py Documentation

Release 1.4.31

holger krekel et. al.

Oct 25, 2017

Contents

see *CHANGELOG* for latest changes.

Note: Since version 1.4, the testing tool "py.test" is part of its own pytest distribution.

Contents:

CHAPTER 1

installation info in a nutshell

PyPI name: py

Pythons: CPython 2.6, 2.7, 3.3, 3.4, PyPy-2.3

Operating systems: Linux, Windows, OSX, Unix

Requirements: setuptools or Distribute

Installers: `easy_install` and `pip`

hg repository: https://bitbucket.org/hpk42/py

easy install or pip `py`

Both Distribute and setuptools provide the `easy_install` installation tool with which you can type into a command line window:

```
easy_install -U py
```

to install the latest release of the py lib. The `-U` switch will trigger an upgrade if you already have an older version installed.

Note: As of version 1.4 py does not contain py.test anymore - you need to install the new **'pytest'_** distribution.

Working from version control or a tarball

To follow development or start experiments, checkout the complete code and documentation source with mercurial:

```
hg clone https://bitbucket.org/hpk42/py
```

Development takes place on the 'trunk' branch.

You can also go to the python package index and download and unpack a TAR file:

```
http://pypi.python.org/pypi/py/
```

activating a checkout with setuptools

With a working Distribute or setuptools installation you can type:

```
python setup.py develop
```

in order to work inline with the tools and the lib of your checkout.

Mailing list and issue tracker

- py-dev developers list and commit mailing list.
- #pylib on irc.freenode.net IRC channel for random questions.
- bitbucket issue tracker use this bitbucket issue tracker to report bugs or request features.

CHAPTER 2

py.path

The ‘py’ lib provides a uniform high-level api to deal with filesystems and filesystem-like interfaces: `py.path`. It aims to offer a central object to fs-like object trees (reading from and writing to files, adding files/directories, examining the types and structure, etc.), and out-of-the-box provides a number of implementations of this API.

py.path.local - local file system path

basic interactive example

The first and most obvious of the implementations is a wrapper around a local filesystem. It’s just a bit nicer in usage than the regular Python APIs, and of course all the functionality is bundled together rather than spread over a number of modules.

Example usage, here we use the `py.test.ensuretemp()` function to create a `py.path.local` object for us (which wraps a directory):

```
>>> import py
>>> temppath = py.test.ensuretemp('py.path_documentation')
>>> foopath = temppath.join('foo') # get child 'foo' (lazily)
>>> foopath.check() # check if child 'foo' exists
False
>>> foopath.write('bar') # write some data to it
>>> foopath.check()
True
>>> foopath.read()
'bar'
>>> foofile = foopath.open() # return a 'real' file object
>>> foofile.read(1)
'b'
```

reference documentation

class `py._path.local.`**`LocalPath`**(*path=None*, *expanduser=False*)
object oriented interface to os.path and other local filesystem related information.

exception **`ImportMismatchError`**
raised on pyimport() if there is a mismatch of __file__'s

`LocalPath.`**`samefile`**(*other*)
return True if 'other' references the same file as 'self'.

`LocalPath.`**`remove`**(*rec=1*, *ignore_errors=False*)
remove a file or directory (or a directory tree if rec=1). if ignore_errors is True, errors while removing directories will be ignored.

`LocalPath.`**`computehash`**(*hashtype='md5'*, *chunksize=524288*)
return hexdigest of hashvalue for this file.

`LocalPath.`**`new`**(***kw*)
create a modified version of this path. the following keyword arguments modify various path parts:

```
a:/some/path/to/a/file.ext
xx                           drive
xxxxxxxxxxxxxxxxx            dirname
                  xxxxxxxx   basename
                  xxxx       purebasename
                       xxx   ext
```

`LocalPath.`**`dirpath`**(**args*, ***kwargs*)
return the directory path joined with any given path arguments.

`LocalPath.`**`join`**(**args*, ***kwargs*)
return a new path by appending all 'args' as path components. if abs=1 is used restart from root if any of the args is an absolute path.

`LocalPath.`**`open`**(*mode='r'*, *ensure=False*, *encoding=None*)
return an opened file with the given mode.

If ensure is True, create parent directories if needed.

`LocalPath.`**`listdir`**(*fil=None*, *sort=None*)
list directory contents, possibly filter by the given fil func and possibly sorted.

`LocalPath.`**`size`**()
return size of the underlying file object

`LocalPath.`**`mtime`**()
return last modification time of the path.

`LocalPath.`**`copy`**(*target*, *mode=False*)
copy path to target.

`LocalPath.`**`rename`**(*target*)
rename this path to target.

`LocalPath.`**`dump`**(*obj*, *bin=1*)
pickle object into path location

`LocalPath.`**`mkdir`**(**args*)
create & return the directory joined with args.

`LocalPath.`**`write_binary`**(*data*, *ensure=False*)
write binary data into path. If ensure is True create missing parent directories.

`LocalPath.`**`write_text`**`(`*data*, *encoding*, *ensure=False*`)`
: write text data into path using the specified encoding. If ensure is True create missing parent directories.

`LocalPath.`**`write`**`(`*data*, *mode='w'*, *ensure=False*`)`
: write data into path. If ensure is True create missing parent directories.

`LocalPath.`**`ensure`**`(`**args*, ***kwargs*`)`
: ensure that an args-joined path exists (by default as a file). if you specify a keyword argument 'dir=True' then the path is forced to be a directory path.

`LocalPath.`**`stat`**`(`*raising=True*`)`
: Return an os.stat() tuple.

`LocalPath.`**`lstat`**`()`
: Return an os.lstat() tuple.

`LocalPath.`**`setmtime`**`(`*mtime=None*`)`
: set modification time for the given path. if 'mtime' is None (the default) then the file's mtime is set to current time.

  Note that the resolution for 'mtime' is platform dependent.

`LocalPath.`**`chdir`**`()`
: change directory to self and return old current directory

`LocalPath.`**`as_cwd`**`(`**args*, ***kwds*`)`
: return context manager which changes to current dir during the managed "with" context. On __enter__ it returns the old dir.

`LocalPath.`**`realpath`**`()`
: return a new path which contains no symbolic links.

`LocalPath.`**`atime`**`()`
: return last access time of the path.

`LocalPath.`**`chmod`**`(`*mode*, *rec=0*`)`
: change permissions to the given mode. If mode is an integer it directly encodes the os-specific modes. if rec is True perform recursively.

`LocalPath.`**`pypkgpath`**`()`
: return the Python package path by looking for the last directory upwards which still contains an __init__.py. Return None if a pkgpath can not be determined.

`LocalPath.`**`pyimport`**`(`*modname=None*, *ensuresyspath=True*`)`
: return path as an imported python module.

  If modname is None, look for the containing package and construct an according module name. The module will be put/looked up in sys.modules. if ensuresyspath is True then the root dir for importing the file (taking __init__.py files into account) will be prepended to sys.path if it isn't there already. If ensuresyspath=="append" the root dir will be appended if it isn't already contained in sys.path. if ensuresyspath is False no modification of syspath happens.

`LocalPath.`**`sysexec`**`(`**argv*, ***popen_opts*`)`
: return stdout text from executing a system child process, where the 'self' path points to executable. The process is directly invoked and not through a system shell.

classmethod `LocalPath.`**`sysfind`**`(`*name*, *checker=None*, *paths=None*`)`
: return a path object found by looking at the systems underlying PATH specification. If the checker is not None it will be invoked to filter matching paths. If a binary cannot be found, None is returned Note: This is probably not working on plain win32 systems but may work on cygwin.

LocalPath.**basename**
basename part of path.

LocalPath.**bestrelpath**(*dest*)
return a string which is a relative path from self (assumed to be a directory) to dest such that self.join(bestrelpath) == dest and if not such path can be determined return dest.

LocalPath.**chown**(*user*, *group*, *rec=0*)
change ownership to the given user and group. user and group may be specified by a number or by a name. if rec is True change ownership recursively.

LocalPath.**common**(*other*)
return the common part shared with the other path or None if there is no common part.

LocalPath.**dirname**
dirname part of path.

LocalPath.**ensure_dir**(**args*)
ensure the path joined with args is a directory.

LocalPath.**ext**
extension of the path (including the '.').

LocalPath.**fnmatch**(*pattern*)
return true if the basename/fullname matches the glob-'pattern'.

valid pattern characters:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

If the pattern contains a path-separator then the full path is used for pattern matching and a '*' is prepended to the pattern.

if the pattern doesn't contain a path-separator the pattern is only matched against the basename.

classmethod LocalPath.**get_temproot**()
return the system's temporary directory (where tempfiles are usually created in)

LocalPath.**load**()
(deprecated) return object unpickled from self.read()

LocalPath.**mklinkto**(*oldname*)
posix style hard link to another name.

LocalPath.**mksymlinkto**(*value*, *absolute=1*)
create a symbolic link with the given value (pointing to another name).

LocalPath.**move**(*target*)
move this path to target.

LocalPath.**parts**(*reverse=False*)
return a root-first list of all ancestor directories plus the path itself.

LocalPath.**purebasename**
pure base name of the path.

LocalPath.**read**(*mode='r'*)
read and return a bytestring from reading the path.

LocalPath.**read_binary**()
read and return a bytestring from reading the path.

LocalPath.**read_text**(*encoding*)
read and return a Unicode string from reading the path.

LocalPath.**readlines**(*cr=1*)
read and return a list of lines from the path. if cr is False, the newline will be removed from the end of each line.

LocalPath.**readlink**()
return value of a symbolic link.

LocalPath.**relto**(*relpath*)
return a string which is the relative part of the path to the given 'relpath'.

LocalPath.**visit**(*fil=None*, *rec=None*, *ignore=<class 'py._path.common.NeverRaised'>*, *bf=False*, *sort=False*)
yields all paths below the current one

fil is a filter (glob pattern or callable), if not matching the path will not be yielded, defaulting to None (everything is returned)

rec is a filter (glob pattern or callable) that controls whether a node is descended, defaulting to None

ignore is an Exception class that is ignoredwhen calling dirlist() on any of the paths (by default, all exceptions are reported)

bf if True will cause a breadthfirst search instead of the default depthfirst. Default: False

sort if True will sort entries within each directory level.

classmethod LocalPath.**mkdtemp**(*rootdir=None*)
return a Path object pointing to a fresh new temporary directory (which we created ourself).

classmethod LocalPath.**make_numbered_dir**(*prefix='session-'*, *rootdir=None*, *keep=3*, *lock_timeout=172800*)
return unique directory with a number greater than the current maximum one. The number is assumed to start directly after prefix. if keep is true directories with a number less than (maxnum-keep) will be removed.

py.path.svnurl and py.path.svnwc

Two other py.path implementations that the py lib provides wrap the popular Subversion revision control system: the first (called 'svnurl') by interfacing with a remote server, the second by wrapping a local checkout. Both allow you to access relatively advanced features such as metadata and versioning, and both in a way more user-friendly manner than existing other solutions.

Some example usage of py.path.svnurl:

```
.. >>> import py
.. >>> if not py.test.config.option.urlcheck: raise ValueError('skipchunk')
>>> url = py.path.svnurl('http://codespeak.net/svn/py')
>>> info = url.info()
>>> info.kind
'dir'
>>> firstentry = url.log()[-1]
>>> import time
>>> time.strftime('%Y-%m-%d', time.gmtime(firstentry.date))
'2004-10-02'
```

Example usage of `py.path.svnwc`:

```
.. >>> if not py.test.config.option.urlcheck: raise ValueError('skipchunk')
>>> temp = py.test.ensuretemp('py.path_documentation')
>>> wc = py.path.svnwc(temp.join('svnwc'))
>>> wc.checkout('http://codespeak.net/svn/py/dist/py/path/local')
>>> wc.join('local.py').check()
True
```

svn path related API reference

class `py._path.svnwc.`**`SvnWCCommandPath`**

path implementation offering access/modification to svn working copies. It has methods similar to the functions in os.path and similar to the commands of the svn client.

`strpath`

string path

`rev`

revision

`url`

url of this WC item

`dump`(*obj*)

pickle object into path location

`svnurl`()

return current SvnPath for this WC-item.

`switch`(*url*)

switch to given URL.

`checkout`(*url=None*, *rev=None*)

checkout from url to local wcpath.

`update`(*rev='HEAD'*, *interactive=True*)

update working copy item to given revision. (None -> HEAD).

`write`(*content*, *mode='w'*)

write content into local filesystem wc.

`dirpath`(**args*)

return the directory Path of the current Path.

`ensure`(**args*, ***kwargs*)

ensure that an args-joined path exists (by default as a file). if you specify a keyword argument 'directory=True' then the path is forced to be a directory path.

`mkdir`(**args*)

create & return the directory joined with args.

`add`()

add ourself to svn

`remove`(*rec=1*, *force=1*)

remove a file or a directory tree. 'rec'ursive is ignored and considered always true (because of underlying svn semantics.

`copy`(*target*)

copy path to target.

rename (*target*)
 rename this path to target.

lock ()
 set a lock (exclusive) on the resource

unlock ()
 unset a previously set lock

cleanup ()
 remove any locks from the resource

status (*updates=0*, *rec=0*, *externals=0*)
 return (collective) Status object for this file.

diff (*rev=None*)
 return a diff of the current path against revision rev (defaulting to the last one).

blame ()
 return a list of tuples of three elements: (revision, commiter, line)

commit (*msg=''*, *rec=1*)
 commit with support for non-recursive commits

propset (*name*, *value*, **args*)
 set property name to value on this path.

propget (*name*)
 get property name on this path.

propdel (*name*)
 delete property name on this path.

proplist (*rec=0*)
 return a mapping of property names to property values. If rec is True, then return a dictionary mapping sub-paths to such mappings.

revert (*rec=0*)
 revert the local changes of this path. if rec is True, do so recursively.

new (***kw*)
 create a modified version of this path. A 'rev' argument indicates a new revision. the following keyword arguments modify various path parts:

  http://host.com/repo/path/file.ext |—————————| dirname

  |——| basename |–| purebasename

  |–| ext

join (**args*, ***kwargs*)
 return a new Path (with the same revision) which is composed of the self Path followed by 'args' path components.

info (*usecache=1*)
 return an Info structure with svn-provided information.

listdir (*fil=None*, *sort=None*)
 return a sequence of Paths.

 listdir will return either a tuple or a list of paths depending on implementation choices.

open (*mode='r'*)
 return an opened file with the given mode.

log(*rev_start=None*, *rev_end=1*, *verbose=False*)
return a list of LogEntry instances for this path. rev_start is the starting revision (defaulting to the first one). rev_end is the last revision (defaulting to HEAD). if verbose is True, then the LogEntry instances also know which files changed.

size()
Return the size of the file content of the Path.

mtime()
Return the last modification time of the file.

basename
basename part of path.

bestrelpath(*dest*)
return a string which is a relative path from self (assumed to be a directory) to dest such that self.join(bestrelpath) == dest and if not such path can be determined return dest.

check(***kw*)
check a path for existence and properties.

Without arguments, return True if the path exists, otherwise False.

valid checkers:

```
file=1    # is a file
file=0    # is not a file (may not even exist)
dir=1     # is a dir
link=1    # is a link
exists=1  # exists
```

You can specify multiple checker definitions, for example:

```
path.check(file=1, link=1)  # a link pointing to a file
```

common(*other*)
return the common part shared with the other path or None if there is no common part.

dirname
dirname part of path.

ensure_dir(**args*)
ensure the path joined with args is a directory.

ext
extension of the path (including the '.').

fnmatch(*pattern*)
return true if the basename/fullname matches the glob-'pattern'.

valid pattern characters:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

If the pattern contains a path-separator then the full path is used for pattern matching and a '*' is prepended to the pattern.

if the pattern doesn't contain a path-separator the pattern is only matched against the basename.

load()
(deprecated) return object unpickled from self.read()

move(*target*)
move this path to target.

parts(*reverse=False*)
return a root-first list of all ancestor directories plus the path itself.

purebasename
pure base name of the path.

read(*mode='r'*)
read and return a bytestring from reading the path.

read_binary()
read and return a bytestring from reading the path.

read_text(*encoding*)
read and return a Unicode string from reading the path.

readlines(*cr=1*)
read and return a list of lines from the path. if cr is False, the newline will be removed from the end of each line.

relto(*relpath*)
return a string which is the relative part of the path to the given 'relpath'.

samefile(*other*)
return True if other refers to the same stat object as self.

visit(*fil=None*, *rec=None*, *ignore=<class 'py._path.common.NeverRaised'>*, *bf=False*, *sort=False*)
yields all paths below the current one

fil is a filter (glob pattern or callable), if not matching the path will not be yielded, defaulting to None (everything is returned)

rec is a filter (glob pattern or callable) that controls whether a node is descended, defaulting to None

ignore is an Exception class that is ignoredwhen calling dirlist() on any of the paths (by default, all exceptions are reported)

bf if True will cause a breadthfirst search instead of the default depthfirst. Default: False

sort if True will sort entries within each directory level.

class py._path.svnurl.**SvnCommandPath**
path implementation that offers access to (possibly remote) subversion repositories.

open(*mode='r'*)
return an opened file with the given mode.

dirpath(**args*, ***kwargs*)
return the directory path of the current path joined with any given path arguments.

mkdir(**args*, ***kwargs*)
create & return the directory joined with args. pass a 'msg' keyword argument to set the commit message.

copy(*target*, *msg='copied by py lib invocation'*)
copy path to target with checkin message msg.

rename(*target*, *msg='renamed by py lib invocation'*)
rename this path to target with checkin message msg.

remove (*rec=1*, *msg='removed by py lib invocation'*)
remove a file or directory (or a directory tree if rec=1) with checkin message msg.

export (*topath*)
export to a local path

topath should not exist prior to calling this, returns a py.path.local instance

ensure (**args*, ***kwargs*)
ensure that an args-joined path exists (by default as a file). If you specify a keyword argument 'dir=True' then the path is forced to be a directory path.

info ()
return an Info structure with svn-provided information.

listdir (*fil=None*, *sort=None*)
list directory contents, possibly filter by the given fil func and possibly sorted.

log (*rev_start=None*, *rev_end=1*, *verbose=False*)
return a list of LogEntry instances for this path. rev_start is the starting revision (defaulting to the first one). rev_end is the last revision (defaulting to HEAD). if verbose is True, then the LogEntry instances also know which files changed.

basename
basename part of path.

bestrelpath (*dest*)
return a string which is a relative path from self (assumed to be a directory) to dest such that self.join(bestrelpath) == dest and if not such path can be determined return dest.

check (***kw*)
check a path for existence and properties.

Without arguments, return True if the path exists, otherwise False.

valid checkers:

```
file=1    # is a file
file=0    # is not a file (may not even exist)
dir=1     # is a dir
link=1    # is a link
exists=1  # exists
```

You can specify multiple checker definitions, for example:

```
path.check(file=1, link=1)  # a link pointing to a file
```

common (*other*)
return the common part shared with the other path or None if there is no common part.

dirname
dirname part of path.

ensure_dir (**args*)
ensure the path joined with args is a directory.

ext
extension of the path (including the '.').

fnmatch (*pattern*)
return true if the basename/fullname matches the glob-'pattern'.

valid pattern characters:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

If the pattern contains a path-separator then the full path is used for pattern matching and a '*' is prepended to the pattern.

if the pattern doesn't contain a path-separator the pattern is only matched against the basename.

join(**args*)
return a new Path (with the same revision) which is composed of the self Path followed by 'args' path components.

load()
(deprecated) return object unpickled from self.read()

move(*target*)
move this path to target.

mtime()
Return the last modification time of the file.

new(***kw*)
create a modified version of this path. A 'rev' argument indicates a new revision. the following keyword arguments modify various path parts:

```
http://host.com/repo/path/file.ext
|-----------------------|          dirname
                         |------| basename
                         |--|     purebasename
                             |--| ext
```

parts(*reverse=False*)
return a root-first list of all ancestor directories plus the path itself.

propget(*name*)
return the content of the given property.

proplist()
list all property names.

purebasename
pure base name of the path.

read(*mode='r'*)
read and return a bytestring from reading the path.

read_binary()
read and return a bytestring from reading the path.

read_text(*encoding*)
read and return a Unicode string from reading the path.

readlines(*cr=1*)
read and return a list of lines from the path. if cr is False, the newline will be removed from the end of each line.

relto(*relpath*)
return a string which is the relative part of the path to the given 'relpath'.

samefile(*other*)
return True if other refers to the same stat object as self.

size()
Return the size of the file content of the Path.

url
url of this svn-path.

visit(*fil=None*, *rec=None*, *ignore=<class 'py._path.common.NeverRaised'>*, *bf=False*, *sort=False*)
yields all paths below the current one

fil is a filter (glob pattern or callable), if not matching the path will not be yielded, defaulting to None (everything is returned)

rec is a filter (glob pattern or callable) that controls whether a node is descended, defaulting to None

ignore is an Exception class that is ignoredwhen calling dirlist() on any of the paths (by default, all exceptions are reported)

bf if True will cause a breadthfirst search instead of the default depthfirst. Default: False

sort if True will sort entries within each directory level.

class py._path.svnwc.**SvnAuth**(*username*, *password*, *cache_auth=True*, *interactive=True*)
container for auth information for Subversion

Common vs. specific API, Examples

All Path objects support a common set of operations, suitable for many use cases and allowing to transparently switch the path object within an application (e.g. from "local" to "svnwc"). The common set includes functions such as *path.read()* to read all data from a file, *path.write()* to write data, *path.listdir()* to get a list of directory entries, *path.check()* to check if a node exists and is of a particular type, *path.join()* to get to a (grand)child, *path.visit()* to recursively walk through a node's children, etc. Only things that are not common on 'normal' filesystems (yet), such as handling metadata (e.g. the Subversion "properties") require using specific APIs.

A quick 'cookbook' of small examples that will be useful 'in real life', which also presents parts of the 'common' API, and shows some non-common methods:

Searching *.txt* files

Search for a particular string inside all files with a .txt extension in a specific directory.

```
>>> dirpath = temppath.ensure('testdir', dir=True)
>>> dirpath.join('textfile1.txt').write('foo bar baz')
>>> dirpath.join('textfile2.txt').write('frob bar spam eggs')
>>> subdir = dirpath.ensure('subdir', dir=True)
>>> subdir.join('textfile1.txt').write('foo baz')
>>> subdir.join('textfile2.txt').write('spam eggs spam foo bar spam')
>>> results = []
>>> for fpath in dirpath.visit('*.txt'):
...     if 'bar' in fpath.read():
...         results.append(fpath.basename)
>>> results.sort()
>>> results
['textfile1.txt', 'textfile2.txt', 'textfile2.txt']
```

Working with Paths

This example shows the `py.path` features to deal with filesystem paths Note that the filesystem is never touched, all operations are performed on a string level (so the paths don't have to exist, either):

```
>>> p1 = py.path.local('/foo/bar')
>>> p2 = p1.join('baz/qux')
>>> p2 == py.path.local('/foo/bar/baz/qux')
True
>>> sep = py.path.local.sep
>>> p2.relto(p1).replace(sep, '/') # os-specific path sep in the string
'baz/qux'
>>> p2.bestrelpath(p1).replace(sep, '/')
'../..'
>>> p2.join(p2.bestrelpath(p1)) == p1
True
>>> p3 = p1 / 'baz/qux' # the / operator allows joining, too
>>> p2 == p3
True
>>> p4 = p1 + ".py"
>>> p4.basename == "bar.py"
True
>>> p4.ext == ".py"
True
>>> p4.purebasename == "bar"
True
```

This should be possible on every implementation of `py.path`, so regardless of whether the implementation wraps a UNIX filesystem, a Windows one, or a database or object tree, these functions should be available (each with their own notion of path seperators and dealing with conversions, etc.).

Checking path types

Now we will show a bit about the powerful 'check()' method on paths, which allows you to check whether a file exists, what type it is, etc.:

```
>>> file1 = temppath.join('file1')
>>> file1.check() # does it exist?
False
>>> file1 = file1.ensure(file=True) # 'touch' the file
>>> file1.check()
True
>>> file1.check(dir=True) # is it a dir?
False
>>> file1.check(file=True) # or a file?
True
>>> file1.check(ext='.txt') # check the extension
False
>>> textfile = temppath.ensure('text.txt', file=True)
>>> textfile.check(ext='.txt')
True
>>> file1.check(basename='file1') # we can use all the path's properties here
True
```

Setting svn-properties

As an example of 'uncommon' methods, we'll show how to read and write properties in an `py.path.svnwc` instance:

```
.. >>> if not py.test.config.option.urlcheck: raise ValueError('skipchunk')
>>> wc.propget('foo')
''
>>> wc.propset('foo', 'bar')
>>> wc.propget('foo')
'bar'
>>> len(wc.status().prop_modified) # our own props
1
>>> msg = wc.revert() # roll back our changes
>>> len(wc.status().prop_modified)
0
```

SVN authentication

Some uncommon functionality can also be provided as extensions, such as SVN authentication:

```
.. >>> if not py.test.config.option.urlcheck: raise ValueError('skipchunk')
>>> auth = py.path.SvnAuth('anonymous', 'user', cache_auth=False,
...             interactive=False)
>>> wc.auth = auth
>>> wc.update() # this should work
>>> path = wc.ensure('thisshouldnotexist.txt')
>>> try:
...     path.commit('testing')
... except py.process.cmdexec.Error, e:
...     pass
>>> 'authorization failed' in str(e)
True
```

Known problems / limitations

- The SVN path objects require the "svn" command line, there is currently no support for python bindings. Parsing the svn output can lead to problems, particularly regarding if you have a non-english "locales" setting.
- While the path objects basically work on windows, there is no attention yet on making unicode paths work or deal with the famous "8.3" filename issues.

CHAPTER 3

py.code: higher level python code and introspection objects

py.code provides higher level APIs and objects for Code, Frame, Traceback, ExceptionInfo and source code construction. The py.code library tries to simplify accessing the code objects as well as creating them. There is a small set of interfaces a user needs to deal with, all nicely bundled together, and with a rich set of 'Pythonic' functionality.

Contents of the library

Every object in the py.code library wraps a code Python object related to code objects, source code, frames and tracebacks: the py.code.Code class wraps code objects, py.code.Source source snippets, py.code. Traceback` exception tracebacks, ``py.code.Frame frame objects (as found in e.g. tracebacks) and py.code.ExceptionInfo the tuple provided by sys.exc_info() (containing exception and traceback information when an exception occurs). Also in the library is a helper function py.code.compile() that provides the same functionality as Python's built-in 'compile()' function, but returns a wrapped code object.

The wrappers

py.code.Code

Code objects are instantiated with a code object or a callable as argument, and provide functionality to compare themselves with other Code objects, get to the source file or its contents, create new Code objects from scratch, etc.

A quick example:

```
>>> import py
>>> c = py.code.Code(py.path.local.read)
>>> c.path.basename
'common.py'
>>> isinstance(c.source(), py.code.Source)
True
>>> str(c.source()).split('\n')[0]
"def read(self, mode='r'):"
```

class `py.code.`**`Code`**(*rawcode*)

wrapper around Python code objects

`path`

return a path object pointing to source code (note that it might not point to an actually existing file).

`fullsource`

return a py.code.Source object for the full source file of the code

`source`()

return a py.code.Source object for the code object's source only

`getargs`(*var=False*)

return a tuple with the argument names for the code object

if 'var' is set True also return the names of the variable and keyword arguments when present

py.code.Source

Source objects wrap snippets of Python source code, providing a simple yet powerful interface to read, deindent, slice, compare, compile and manipulate them, things that are not so easy in core Python.

Example:

```
>>> s = py.code.Source("""\
...   def foo():
...     print "foo"
... """)
>>> str(s).startswith('def') # automatic de-indentation!
True
>>> s.isparseable()
True
>>> sub = s.getstatement(1) # get the statement starting at line 1
>>> str(sub).strip() # XXX why is the strip() required?!?
'print "foo"'
```

class `py.code.`**`Source`**(**parts*, ***kwargs*)

a immutable object holding a source code fragment, possibly deindenting it.

`strip`()

return new source object with trailing and leading blank lines removed.

`putaround`(*before='`'*, *after='`'*, *indent=' `'*)

return a copy of the source object with 'before' and 'after' wrapped around it.

`indent`(*indent=' `'*)

return a copy of the source object with all lines indented by the given indent-string.

`getstatement`(*lineno*, *assertion=False*)

return Source statement which contains the given linenumber (counted from 0).

`getstatementrange`(*lineno*, *assertion=False*)

return (start, end) tuple which spans the minimal statement region which containing the given lineno.

`deindent`(*offset=None*)

return a new source object deindented by offset. If offset is None then guess an indentation offset from the first non-blank line. Subsequent lines which have a lower indentation offset will be copied verbatim as they are assumed to be part of multilines.

`isparseable`(*deindent=True*)

return True if source is parseable, heuristically deindenting it by default.

compile (*filename=None*, *mode='exec'*, *flag=0*, *dont_inherit=0*, *_genframe=None*)
: return compiled code object. if filename is None invent an artificial filename which displays the source/line position of the caller frame.

py.code.Traceback

Tracebacks are usually not very easy to examine, you need to access certain somewhat hidden attributes of the traceback's items (resulting in expressions such as 'fname = tb.tb_next.tb_frame.f_code.co_filename'). The Traceback interface (and its TracebackItem children) tries to improve this.

Example:

```
>>> import sys
>>> try:
...   py.path.local(100) # illegal argument
... except:
...   exc, e, tb = sys.exc_info()
>>> t = py.code.Traceback(tb)
>>> first = t[1] # get the second entry (first is in this doc)
>>> first.path.basename # second is in py/path/local.py
'local.py'
>>> isinstance(first.statement, py.code.Source)
True
>>> str(first.statement).strip().startswith('raise ValueError')
True
```

class py.code.**Traceback** (*tb*)
: Traceback objects encapsulate and offer higher level access to Traceback entries.

Entry
: alias of TracebackEntry

cut (*path=None*, *lineno=None*, *firstlineno=None*, *excludepath=None*)
: return a Traceback instance wrapping part of this Traceback

by provding any combination of path, lineno and firstlineno, the first frame to start the to-be-returned traceback is determined

this allows cutting the first part of a Traceback instance e.g. for formatting reasons (removing some uninteresting bits that deal with handling of the exception/traceback)

filter (*fn=<function <lambda>>*)
: return a Traceback instance with certain items removed

fn is a function that gets a single argument, a TracebackItem instance, and should return True when the item should be added to the Traceback, False when not

by default this removes all the TracebackItems which are hidden (see ishidden() above)

getcrashentry ()
: return last non-hidden traceback entry that lead to the exception of a traceback.

recursionindex ()
: return the index of the frame/TracebackItem where recursion originates if appropriate, None if no recursion occurred

append ()
: L.append(object) – append object to end

count (*value*) → integer – return number of occurrences of value

extend()
L.extend(iterable) – extend list by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert()
L.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse()
L.reverse() – reverse *IN PLACE*

sort()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

py.code.Frame

Frame wrappers are used in `py.code.Traceback` items, and will usually not directly be instantiated. They provide some nice methods to evaluate code 'inside' the frame (using the frame's local variables), get to the underlying code (frames have a code attribute that points to a `py.code.Code` object) and examine the arguments.

Example (using the 'first' TracebackItem instance created above):

```
>>> frame = first.frame
>>> isinstance(frame.code, py.code.Code)
True
>>> isinstance(frame.eval('self'), py.path.local)
True
>>> [namevalue[0] for namevalue in frame.getargs()]
['cls', 'path']
```

class py.code.**Frame**(*frame*)
Wrapper around a Python frame holding f_locals and f_globals in which expressions can be evaluated.

statement
statement this frame is at

eval(*code*, ***vars*)
evaluate 'code' in the frame

'vars' are optional additional local variables

returns the result of the evaluation

exec_(*code*, ***vars*)
exec 'code' in the frame

'vars' are optiona; additional local variables

repr(*object*)
return a 'safe' (non-recursive, one-line) string repr for 'object'

getargs(*var=False*)
return a list of tuples (name, value) for all arguments

if 'var' is set True also include the variable and keyword arguments when present

py.code.ExceptionInfo

A wrapper around the tuple returned by sys.exc_info() (will call sys.exc_info() itself if the tuple is not provided as an argument), provides some handy attributes to easily access the traceback and exception string.

Example:

```
>>> import sys
>>> try:
...   foobar()
... except:
...   excinfo = py.code.ExceptionInfo()
>>> excinfo.typename
'NameError'
>>> isinstance(excinfo.traceback, py.code.Traceback)
True
>>> excinfo.exconly()
"NameError: name 'foobar' is not defined"
```

class `py.code.`**`ExceptionInfo`**(*tup=None*, *exprinfo=None*)
wraps sys.exc_info() objects and offers help for navigating the traceback.

`type` = **None**
the exception class

`value` = **None**
the exception instance

`tb` = **None**
the exception raw traceback

`typename` = **None**
the exception type name

`traceback` = **None**
the exception traceback (py.code.Traceback instance)

`exconly`(*tryshort=False*)
return the exception as a string

when 'tryshort' resolves to True, and the exception is a py.code._AssertionError, only the actual exception part of the exception representation is returned (so 'AssertionError: ' is removed from the beginning)

`errisinstance`(*exc*)
return True if the exception is an instance of exc

`getrepr`(*showlocals=False*, *style='long'*, *abspath=False*, *tbfilter=True*, *funcargs=False*)
return str()able representation of this exception info. showlocals: show locals per traceback entry style: long|short|no|native traceback style tbfilter: hide entries (where __tracebackhide__ is true)

in case of style==native, tbfilter and showlocals is ignored.

class `py.code.`**`Traceback`**(*tb*)
Traceback objects encapsulate and offer higher level access to Traceback entries.

`Entry`
alias of `TracebackEntry`

`cut`(*path=None*, *lineno=None*, *firstlineno=None*, *excludepath=None*)
return a Traceback instance wrapping part of this Traceback

by provding any combination of path, lineno and firstlineno, the first frame to start the to-be-returned traceback is determined this allows cutting the first part of a Traceback instance e.g. for formatting reasons (removing some uninteresting bits that deal with handling of the exception/traceback)

`filter`(*fn=<function <lambda>>*)
return a Traceback instance with certain items removed

fn is a function that gets a single argument, a TracebackItem instance, and should return True when the item should be added to the Traceback, False when not

by default this removes all the TracebackItems which are hidden (see ishidden() above)

`getcrashentry`()
return last non-hidden traceback entry that lead to the exception of a traceback.

`recursionindex`()
return the index of the frame/TracebackItem where recursion originates if appropriate, None if no recursion occurred

`append`()
L.append(object) – append object to end

`count`(*value*) → integer – return number of occurrences of value

`extend`()
L.extend(iterable) – extend list by appending elements from the iterable

`index`(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

`insert`()
L.insert(index, object) – insert object before index

`pop`([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`remove`()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`reverse`()
L.reverse() – reverse *IN PLACE*

`sort`()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

CHAPTER 4

py.io

The ‘py’ lib provides helper classes for capturing IO during execution of a program.

IO Capturing examples

py.io.StdCapture

Basic Example:

```
>>> import py
>>> capture = py.io.StdCapture()
>>> print "hello"
>>> out,err = capture.reset()
>>> out.strip() == "hello"
True
```

For calling functions you may use a shortcut:

```
>>> import py
>>> def f(): print "hello"
>>> res, out, err = py.io.StdCapture.call(f)
>>> out.strip() == "hello"
True
```

py.io.StdCaptureFD

If you also want to capture writes to the stdout/stderr filedescriptors you may invoke:

```
>>> import py, sys
>>> capture = py.io.StdCaptureFD(out=False, in_=False)
>>> sys.stderr.write("world")
>>> out,err = capture.reset()
>>> err
'world'
```

py.io object reference

class `py.io.`**`StdCaptureFD`**(*out=True*, *err=True*, *mixed=False*, *in_=True*, *patchsys=True*, *now=True*)

This class allows to capture writes to FD1 and FD2 and may connect a NULL file to FD0 (and prevent reads from sys.stdin). If any of the 0,1,2 file descriptors is invalid it will not be captured.

`resume`()
: resume capturing with original temp files.

`done`(*save=True*)
: return (outfile, errfile) and stop capturing.

`readouterr`()
: return snapshot value of stdout/stderr capturings.

`call`(*func*, **args*, ***kwargs*)
: return a (res, out, err) tuple where out and err represent the output/error output during function execution. call the given function with args/kwargs and capture output/error during its execution.

`reset`()
: reset sys.stdout/stderr and return captured output as strings.

`suspend`()
: return current snapshot captures, memorize tempfiles.

class `py.io.`**`StdCapture`**(*out=True*, *err=True*, *in_=True*, *mixed=False*, *now=True*)

This class allows to capture writes to sys.stdout|stderr "in-memory" and will raise errors on tries to read from sys.stdin. It only modifies sys.stdout|stderr|stdin attributes and does not touch underlying File Descriptors (use StdCaptureFD for that).

`done`(*save=True*)
: return (outfile, errfile) and stop capturing.

`resume`()
: resume capturing with original temp files.

`readouterr`()
: return snapshot value of stdout/stderr capturings.

`call`(*func*, **args*, ***kwargs*)
: return a (res, out, err) tuple where out and err represent the output/error output during function execution. call the given function with args/kwargs and capture output/error during its execution.

`reset`()
: reset sys.stdout/stderr and return captured output as strings.

`suspend`()
: return current snapshot captures, memorize tempfiles.

class `py.io.`**`TerminalWriter`**(*file=None*, *stringio=False*, *encoding=None*)

CHAPTER 5

py.log documentation and musings

Foreword

This document is an attempt to briefly state the actual specification of the `py.log` module. It was written by Francois Pinard and also contains some ideas for enhancing the py.log facilities.

NOTE that `py.log` is subject to refactorings, it may change with the next release.

This document is meant to trigger or facilitate discussions. It shamelessly steals from the Agile Testing comments, and from other sources as well, without really trying to sort them out.

Logging organisation

The `py.log` module aims a niche comparable to the one of the logging module found within the standard Python distributions, yet with much simpler paradigms for configuration and usage.

Holger Krekel, the main `py` library developer, introduced the idea of keyword-based logging and the idea of logging *producers* and *consumers*. A log producer is an object used by the application code to send messages to various log consumers. When you create a log producer, you define a set of keywords that are then used to both route the logging messages to consumers, and to prefix those messages.

In fact, each log producer has a few keywords associated with it for identification purposes. These keywords form a tuple of strings, and may be used to later retrieve a particular log producer.

A log producer may (or may not) be associated with a log consumer, meant to handle log messages in particular ways. The log consumers can be `STDOUT`, `STDERR`, log files, syslog, the Windows Event Log, user defined functions, etc. (Yet, logging to syslog or to the Windows Event Log is only future plans for now). A log producer has never more than one consumer at a given time, but it is possible to dynamically switch a producer to use another consumer. On the other hand, a single log consumer may be associated with many producers.

Note that creating and associating a producer and a consumer is done automatically when not otherwise overriden, so using `py` logging is quite comfortable even in the smallest programs. More typically, the application programmer will likely design a hierarchy of producers, and will select keywords appropriately for marking the hierarchy tree. If a node of the hierarchical tree of producers has to be divided in sub-trees, all producers in the sub-trees share, as a common prefix, the keywords of the node being divided. In other words, we go further down in the hierarchy of producers merely by adding keywords.

Using the py.log library

To use the `py.log` library, the user must import it into a Python application, create at least one log producer and one log consumer, have producers and consumers associated, and finally call the log producers as needed, giving them log messages.

Importing

Once the `py` library is installed on your system, a mere:

```
import py
```

holds enough magic for lazily importing the various facilities of the `py` library when they are first needed. This is really how `py.log` is made available to the application. For example, after the above `import py`, one may directly write `py.log.Producer(...)` and everything should work fine, the user does not have to worry about specifically importing more modules.

Creating a producer

There are three ways for creating a log producer instance:

- As soon as `py.log` is first evaluated within an application program, a default log producer is created, and made available under the name `py.log.default`. The keyword `default` is associated with that producer.
- The `py.log.Producer()` constructor may be explicitly called for creating a new instance of a log producer. That constructor accepts, as an argument, the keywords that should be associated with that producer. Keywords may be given either as a tuple of keyword strings, or as a single space-separated string of keywords.
- Whenever an attribute is *taken* out of a log producer instance, for the first time that attribute is taken, a new log producer is created. The keywords associated with that new producer are those of the initial producer instance, to which is appended the name of the attribute being taken.

The last point is especially useful, as it allows using log producers without further declarations, merely creating them *on-the-fly*.

Creating a consumer

There are many ways for creating or denoting a log consumer:

- A default consumer exists within the `py.log` facilities, which has the effect of writing log messages on the Python standard output stream. That consumer is associated at the very top of the producer hierarchy, and as such, is called whenever no other consumer is found.
- The notation `py.log.STDOUT` accesses a log consumer which writes log messages on the Python standard output stream.
- The notation `py.log.STDERR` accesses a log consumer which writes log messages on the Python standard error stream.
- The `py.log.File()` constructor accepts, as argument, either a file already opened in write mode or any similar file-like object, and creates a log consumer able to write log messages onto that file.

- The `py.log.Path()` constructor accepts a file name for its first argument, and creates a log consumer able to write log messages into that file. The constructor call accepts a few keyword parameters:
  - `append`, which is `False` by default, may be used for opening the file in append mode instead of write mode.
  - `delayed_create`, which is `False` by default, maybe be used for opening the file at the latest possible time. Consequently, the file will not be created if it did not exist, and no actual log message gets written to it.
  - `buffering`, which is 1 by default, is used when opening the file. Buffering can be turned off by specifying a 0 value. The buffer size may also be selected through this argument.
- Any user defined function may be used for a log consumer. Such a function should accept a single argument, which is the message to write, and do whatever is deemed appropriate by the programmer. When the need arises, this may be an especially useful and flexible feature.
- The special value `None` means no consumer at all. This acts just like if there was a consumer which would silently discard all log messages sent to it.

Associating producers and consumers

Each log producer may have at most one log consumer associated with it. A log producer gets associated with a log consumer through a `py.log.setconsumer()` call. That function accepts two arguments, the first identifying a producer (a tuple of keyword strings or a single space-separated string of keywords), the second specifying the precise consumer to use for that producer. Until this function is called for a producer, that producer does not have any explicit consumer associated with it.

Now, the hierarchy of log producers establishes which consumer gets used whenever a producer has no explicit consumer. When a log producer has no consumer explicitly associated with it, it dynamically and recursively inherits the consumer of its parent node, that is, that node being a bit closer to the root of the hierarchy. In other words, the rightmost keywords of that producer are dropped until another producer is found which has an explicit consumer. A nice side-effect is that, by explicitly associating a consumer with a producer, all consumer-less producers which appear under that producer, in the hierarchy tree, automatically *inherits* that consumer.

Writing log messages

All log producer instances are also functions, and this is by calling them that log messages are generated. Each call to a producer object produces the text for one log entry, which in turn, is sent to the log consumer for that producer.

The log entry displays, after a prefix identifying the log producer being used, all arguments given in the call, converted to strings and space-separated. (This is meant by design to be fairly similar to what the `print` statement does in Python). The prefix itself is made up of a colon-separated list of keywords associated with the producer, the whole being set within square brackets.

Note that the consumer is responsible for adding the newline at the end of the log entry. That final newline is not part of the text for the log entry.

CHAPTER 6

py.xml: simple pythonic xml/html file generation

Motivation

There are a plethora of frameworks and libraries to generate xml and html trees. However, many of them are large, have a steep learning curve and are often hard to debug. Not to speak of the fact that they are frameworks to begin with.

a pythonic object model , please

The py lib offers a pythonic way to generate xml/html, based on ideas from xist which uses python class objects to build xml trees. However, xist's implementation is somewhat heavy because it has additional goals like transformations and supporting many namespaces. But its basic idea is very easy.

generating arbitrary xml structures

With `py.xml.Namespace` you have the basis to generate custom xml-fragments on the fly:

```
class ns(py.xml.Namespace):
    "my custom xml namespace"
doc = ns.books(
    ns.book(
        ns.author("May Day"),
        ns.title("python for java programmers"),),
    ns.book(
        ns.author("why"),
        ns.title("Java for Python programmers"),),
    publisher="N.N",
    )
print doc.unicode(indent=2).encode('utf8')
```

will give you this representation:

```
<books publisher="N.N">
  <book>
    <author>May Day</author>
    <title>python for java programmers</title></book>
  <book>
    <author>why</author>
    <title>Java for Python programmers</title></book></books>
```

In a sentence: positional arguments are child-tags and keyword-arguments are attributes.

On a side note, you'll see that the unicode-serializer supports a nice indentation style which keeps your generated html readable, basically through emulating python's white space significance by putting closing-tags rightmost and almost invisible at first glance :-)

basic example for generating html

Consider this example:

```
from py.xml import html  # html namespace

paras = "First Para", "Second para"

doc = html.html(
   html.head(
        html.meta(name="Content-Type", value="text/html; charset=latin1")),
   html.body(
        [html.p(p) for p in paras]))

print unicode(doc).encode('latin1')
```

Again, tags are objects which contain tags and have attributes. More exactly, Tags inherit from the list type and thus can be manipulated as list objects. They additionally support a default way to represent themselves as a serialized unicode object.

If you happen to look at the py.xml implementation you'll note that the tag/namespace implementation consumes some 50 lines with another 50 lines for the unicode serialization code.

CSS-styling your html Tags

One aspect where many of the huge python xml/html generation frameworks utterly fail is a clean and convenient integration of CSS styling. Often, developers are left alone with keeping CSS style definitions in sync with some style files represented as strings (often in a separate .css file). Not only is this hard to debug but the missing abstractions make it hard to modify the styling of your tags or to choose custom style representations (inline, html.head or external). Add the Browers usual tolerance of messyness and errors in Style references and welcome to hell, known as the domain of developing web applications :-)

By contrast, consider this CSS styling example:

```
class my(html):
    "my initial custom style"
    class body(html.body):
        style = html.Style(font_size = "120%")

    class h2(html.h2):
        style = html.Style(background = "grey")
    class p(html.p):
        style = html.Style(font_weight="bold")

doc = my.html(
    my.head(),
    my.body(
        my.h2("hello world"),
        my.p("bold as bold can")
    )
)

print doc.unicode(indent=2)
```

This will give you a small'n mean self contained represenation by default:

```
<html>
  <head/>
  <body style="font-size: 120%">
    <h2 style="background: grey">hello world</h2>
    <p style="font-weight: bold">bold as bold can</p></body></html>
```

Most importantly, note that the inline-styling is just an implementation detail of the unicode serialization code. You can easily modify the serialization to put your styling into the `html.head` or in a separate file and autogenerate CSS-class names or ids.

Hey, you could even write tests that you are using correct styles suitable for specific browser requirements. Did i mention that the ability to easily write tests for your generated html and its serialization could help to develop _stable_ user interfaces?

More to come ...

For now, i don't think we should strive to offer much more than the above. However, it is probably not hard to offer *partial serialization* to allow generating maybe hundreds of complex html documents per second. Basically we would allow putting callables both as Tag content and as values of attributes. A slightly more advanced Serialization would then produce a list of unicode objects intermingled with callables. At HTTP-Request time the callables would get called to complete the probably request-specific serialization of your Tags. Hum, it's probably harder to explain this than to actually code it :-)

CHAPTER 7

Miscellaneous features of the py lib

Mapping the standard python library into py

The py.std object allows lazy access to standard library modules. For example, to get to the print-exception functionality of the standard library you can write:

```
py.std.traceback.print_exc()
```

without having to do anything else than the usual import py at the beginning. You can access any other top-level standard library module this way. This means that you will only trigger imports of modules that are actually needed. Note that no attempt is made to import submodules.

Support for interaction with system utilities/binaries

Currently, the py lib offers two ways to interact with system executables. py.process.cmdexec() invokes the shell in order to execute a string. The other one, py.path.local's 'sysexec()' method lets you directly execute a binary.

Both approaches will raise an exception in case of a return- code other than 0 and otherwise return the stdout-output of the child process.

The shell based approach

You can execute a command via your system shell by doing something like:

```
out = py.process.cmdexec('ls -v')
```

However, the cmdexec approach has a few shortcomings:

- it relies on the underlying system shell
- it neccessitates shell-escaping for expressing arguments

- it does not easily allow to "fix" the binary you want to run.
- it only allows to execute executables from the local filesystem

local paths have `sysexec`

In order to synchronously execute an executable file you can use `sysexec`:

```
binsvn.sysexec('ls', 'http://codespeak.net/svn')
```

where `binsvn` is a path that points to the `svn` commandline binary. Note that this function does not offer any shell-escaping so you have to pass in already separated arguments.

finding an executable local path

Finding an executable is quite different on multiple platforms. Currently, the `PATH` environment variable based search on unix platforms is supported:

```
py.path.local.sysfind('svn')
```

which returns the first path whose `basename` matches `svn`. In principle, *sysfind* deploys platform specific algorithms to perform the search. On Windows, for example, it may look at the registry (XXX).

To make the story complete, we allow to pass in a second `checker` argument that is called for each found executable. For example, if you have multiple binaries available you may want to select the right version:

```
def mysvn(p):
    """ check that the given svn binary has version 1.1. """
    line = p.execute('--version'').readlines()[0]
    if line.find('version 1.1'):
        return p
binsvn = py.path.local.sysfind('svn', checker=mysvn)
```

Cross-Python Version compatibility helpers

The `py.builtin` namespace provides a number of helpers that help to write python code compatible across Python interpreters, mainly Python2 and Python3. Type `help(py.builtin)` on a Python prompt for a the selection of builtins.

CHAPTER 8

1.4.31

- fix local().copy(dest, mode=True) to also work with unicode.
- pass better error message with svn EEXIST paths

CHAPTER 9

1.4.30

- fix issue68 an assert with a multiline list comprehension was not reported correctly. Thanks Henrik Heibuerger.

CHAPTER 10

1.4.29

- fix issue55: revert a change to the statement finding algorithm which is used by pytest for generating tracebacks. Thanks Daniel Hahler for initial analysis.
- fix pytest issue254 for when traceback rendering can't find valid source code. Thanks Ionel Cristian Maries.

CHAPTER 11

1.4.28

- fix issue64 – dirpath regression when “abs=True” is passed. Thanks Gilles Dartiguelongue.

CHAPTER 12

1.4.27

- fix issue59: point to new repo site
- allow a new ensuresyspath=”append” mode for py.path.local.pyimport() so that a neccessary import path is appended instead of prepended to sys.path
- strike undocumented, untested argument to py.path.local.pypkgpath
- speed up py.path.local.dirpath by a factor of 10

CHAPTER 13

1.4.26

- avoid calling normpath twice in py.path.local
- py.builtin._reraise properly reraises under Python3 now.
- fix issue53 - remove module index, thanks jenisys.
- allow posix path separators when "fnmatch" is called. Thanks Christian Long for the complete PR.

CHAPTER 14

1.4.25

- fix issue52: vaguely fix py25 compat of py.path.local (it's not officially supported), also fix docs
- fix pytest issue 589: when checking if we have a recursion error check for the specific "maximum recursion depth" text of the exception.

CHAPTER 15

1.4.24

- Fix retrieving source when an else: line has an other statement on the same line.
- add localpath read_text/write_text/read_bytes/write_bytes methods as shortcuts and clearer bytes/text interfaces for read/write. Adapted from a PR from Paul Moore.

CHAPTER 16

1.4.23

- use newer apipkg version which makes attribute access on alias modules resolve to None rather than an ImportError. This helps with code that uses inspect.getframeinfo() on py34 which causes a complete walk on sys.modules thus triggering the alias module to resolve and blowing up with ImportError. The negative side is that something like "py.test.X" will now result in None instead of "importerror: pytest" if pytest is not installed. But you shouldn't import "py.test" anyway anymore.
- adapt one svn test to only check for any exception instead of specific ones because different svn versions cause different errors and we don't care.

CHAPTER 17

1.4.22

- refactor class-level registry on ForkedFunc child start/finish event to become instance based (i.e. passed into the constructor)

CHAPTER 18

1.4.21

- ForkedFunc now has class-level register_on_start/on_exit() methods to allow adding information in the boxed process. Thanks Marc Schlaich.
- ForkedFunc in the child opens in “auto-flush” mode for stdout/stderr so that when a subprocess dies you can see its output even if it didn’t flush itself.
- refactor traceback generation in light of pytest issue 364 (shortening tracebacks). you can now set a new traceback style on a per-entry basis such that a caller can force entries to be isplayed as short or long entries.
- win32: py.path.local.sysfind(name) will preferrably return files with extensions so that if “X” and “X.bat” or “X.exe” is on the PATH, one of the latter two will be returned.

CHAPTER 19

1.4.20

- ignore unicode decode errors in xmlescape. Thanks Anatoly Bubenkoff.
- on python2 modify traceback.format_exception_only to match python3 behaviour, namely trying to print unicode for Exception instances
- use a safer way for serializing exception reports (helps to fix pytest issue413)

CHAPTER 20

Changes between 1.4.18 and 1.4.19

- merge in apipkg fixes
- some micro-optimizations in py/_code/code.py for speeding up pytest runs. Thanks Alex Gaynor for initiative.
- check PY_COLORS=1 or PY_COLORS=0 to force coloring/not-coloring for py.io.TerminalWriter() independently from capabilities of the output file. Thanks Marc Abramowitz for the PR.
- some fixes to unicode handling in assertion handling. Thanks for the PR to Floris Bruynooghe. (This helps to fix pytest issue 319).
- depend on setuptools presence, remove distribute_setup

CHAPTER 21

Changes between 1.4.17 and 1.4.18

- introduce path.ensure_dir() as a synonym for ensure(..., dir=1)
- some unicode/python3 related fixes wrt to path manipulations (if you start passing unicode particular in py2 you might still get problems, though)

CHAPTER 22

Changes between 1.4.16 and 1.4.17

- make py.io.TerminalWriter() prefer colorama if it is available and avoid empty lines when separator-lines are printed by being defensive and reducing the working terminalwidth by 1
- introduce optional "expanduser" argument to py.path.local to that local("~", expanduser=True) gives the home directory of "user".

CHAPTER 23

Changes between 1.4.15 and 1.4.16

- fix issue35 - define __gt__ ordering between a local path and strings
- fix issue36 - make chdir() work even if os.getcwd() fails.
- add path.exists/isdir/isfile/islink shortcuts
- introduce local path.as_cwd() context manager.
- introduce p.write(ensure=1) and p.open(ensure=1) where ensure triggers creation of neccessary parent dirs.

CHAPTER 24

Changes between 1.4.14 and 1.4.15

- majorly speed up some common calling patterns with LocalPath.listdir()/join/check/stat functions considerably.
- fix an edge case with fnmatch where a glob style pattern appeared in an absolute path.

CHAPTER 25

Changes between 1.4.13 and 1.4.14

- fix dupfile to work with files that don’t carry a mode. Thanks Jason R. Coombs.

CHAPTER 26

Changes between 1.4.12 and 1.4.13

- fix getting statementrange/compiling a file ending in a comment line without newline (on python2.5)
- for local paths you can pass "mode=True" to a copy() in order to copy permission bits (underlying mechanism is using shutil.copymode)
- add paths arguments to py.path.local.sysfind to restrict search to the diretories in the path.
- add isdir/isfile/islink to path.stat() objects allowing to perform multiple checks without calling out multiple times
- drop py.path.local.__new__ in favour of a simpler __init__
- iniconfig: allow "name:value" settings in config files, no space after "name" required
- fix issue 27 - NameError in unlikely untested case of saferepr

CHAPTER 27

Changes between 1.4.11 and 1.4.12

- fix python2.4 support - for pre-AST interpreters re-introduce old way to find statements in exceptions (closes pytest issue 209)
- add tox.ini to distribution
- fix issue23 - print ,* args information in tracebacks, thanks Manuel Jacob

CHAPTER 28

Changes between 1.4.10 and 1.4.11

- use _ast to determine statement ranges when printing tracebacks - avoiding multi-second delays on some large test modules
- fix an internal test to not use class-denoted **pytest_funcarg__**
- fix a doc link to bug tracker
- try to make terminal.write() printing more robust against unicodeencode/decode problems, amend according test
- introduce py.builtin.text and py.builtin.bytes to point to respective str/unicode (py2) and bytes/str (py3) types
- fix error handling on win32/py33 for ENODIR

CHAPTER 29

Changes between 1.4.9 and 1.4.10

- terminalwriter: default to encode to UTF8 if no encoding is defined on the output stream
- issue22: improve heuristic for finding the statementrange in exceptions

CHAPTER 30

Changes between 1.4.8 and 1.4.9

- fix bug of path.visit() which would not recognize glob-style patterns for the “rec” recursion argument
- changed iniconfig parsing to better conform, now the chars ”;” and “#” only mark a comment at the stripped start of a line
- include recent apipkg-1.2
- change internal terminalwriter.line/reline logic to more nicely support file spinners

CHAPTER 31

Changes between 1.4.7 and 1.4.8

- fix issue 13 - correct handling of the tag name object in xmlgen
- fix issue 14 - support raw attribute values in xmlgen
- fix windows terminalwriter printing/re-line problem
- update distribute_setup.py to 0.6.27

CHAPTER 32

Changes between 1.4.6 and 1.4.7

- fix issue11 - own test failure with python3.3 / Thanks Benjamin Peterson
- help fix pytest issue 102

CHAPTER 33

Changes between 1.4.5 and 1.4.6

- help to fix pytest issue99: unify output of ExceptionInfo.getrepr(style="native") with ...(style="long")
- fix issue7: source.getstatementrange() now raises proper error if no valid statement can be found
- fix issue8: fix code and tests of svnurl/svnwc to work on subversion 1.7 - note that path.status(updates=1) will not properly work svn-17's status –xml output is broken.
- make source.getstatementrange() more resilent about non-python code frames (as seen from jnja2)
- make trackeback recursion detection more resilent about the eval magic of a decorator library
- iniconfig: add support for ; as comment starter
- properly handle lists in xmlgen on python3
- normalize py.code.getfslineno(obj) to always return a (string, int) tuple defaulting to ("", -1) respectively if no source code can be found for obj.

CHAPTER 34

Changes between 1.4.4 and 1.4.5

- improve some unicode handling in terminalwriter and capturing (used by pytest)

CHAPTER 35

Changes between 1.4.3 and 1.4.4

- a few fixes and assertion related refinements for pytest-2.1
- guard py.code.Code and getfslineno against bogus input and make py.code.Code objects for object instance by looking up their __call__ function.
- make exception presentation robust against invalid current cwd

CHAPTER 36

Changes between 1.4.2 and 1.4.3

- fix terminal coloring issue for skipped tests (thanks Amaury)
- fix issue4 - large calls to ansi_print (thanks Amaury)

CHAPTER 37

Changes between 1.4.1 and 1.4.2

- fix (pytest) issue23 - tmpdir argument now works on Python3.2 and WindowsXP (which apparently starts to offer os.symlink now)
- better error message for syntax errors from compiled code
- small fix to better deal with (un-)colored terminal output on windows

CHAPTER 38

Changes between 1.4.0 and 1.4.1

- fix issue1 - py.error.* classes to be pickleable
- fix issue2 - on windows32 use PATHEXT as the list of potential extensions to find find binaries with py.path.local.sysfind(commandname)
- fix (pytest-) issue10 and refine assertion reinterpretation to avoid breaking if the __nonzero__ of an object fails
- fix (pytest-) issue17 where python3 does not like “import *” leading to misrepresentation of import-errors in test modules
- fix py.error.* attribute pypy access issue
- allow path.samefile(arg) to succeed when arg is a relative filename
- fix (pytest-) issue20 path.samefile(relpath) works as expected now
- fix (pytest-) issue8 len(long_list) now shows the lenght of the list

CHAPTER 39

Changes between 1.3.4 and 1.4.0

- py.test was moved to a separate "pytest" package. What remains is a stub hook which will proxy `import py.test` to `pytest`.
- all command line tools ("py.cleanup/lookup/countloc/..." moved to "pycmd" package)
- removed the old and deprecated "py.magic" namespace
- use apipkg-1.1 and make py.apipkg.initpkg|ApiModule available
- add py.iniconfig module for brain-dead easy ini-config file parsing
- introduce py.builtin.any()
- path objects have a .dirname attribute now (equivalent to os.path.dirname(path))
- path.visit() accepts breadthfirst (bf) and sort options
- remove deprecated py.compat namespace

CHAPTER 40

Changes between 1.3.3 and 1.3.4

- fix issue111: improve install documentation for windows
- fix issue119: fix custom collectability of __init__.py as a module
- fix issue116: –doctestmodules work with __init__.py files as well
- fix issue115: unify internal exception passthrough/catching/GeneratorExit
- fix issue118: new –tb=native for presenting cpython-standard exceptions

CHAPTER 41

Changes between 1.3.2 and 1.3.3

- fix issue113: assertion representation problem with triple-quoted strings (and possibly other cases)
- make conftest loading detect that a conftest file with the same content was already loaded, avoids surprises in nested directory structures which can be produced e.g. by Hudson. It probably removes the need to use –confcutdir in most cases.
- fix terminal coloring for win32 (thanks Michael Foord for reporting)
- fix weirdness: make terminal width detection work on stdout instead of stdin (thanks Armin Ronacher for reporting)
- remove trailing whitespace in all py/text distribution files

CHAPTER 42

Changes between 1.3.1 and 1.3.2

New features

- fix issue103: introduce py.test.raises as context manager, examples:

```
with py.test.raises(ZeroDivisionError):
    x = 0
    1 / x

with py.test.raises(RuntimeError) as excinfo:
    call_something()

# you may do extra checks on excinfo.value|type|traceback here
```

  (thanks Ronny Pfannschmidt)

- Funcarg factories can now dynamically apply a marker to a test invocation. This is for example useful if a factory provides parameters to a test which are expected-to-fail:

```
def pytest_funcarg__arg(request):
    request.applymarker(py.test.mark.xfail(reason="flaky config"))
    ...

def test_function(arg):
    ...
```

- improved error reporting on collection and import errors. This makes use of a more general mechanism, namely that for custom test item/collect nodes `node.repr_failure(excinfo)` is now uniformly called so that you can override it to return a string error representation of your choice which is going to be reported as a (red) string.
- introduce '–junitprefix=STR' option to prepend a prefix to all reports in the junitxml file.

Bug fixes / Maintenance

- make tests and the `pytest_recwarn` plugin in particular fully compatible to Python2.7 (if you use the `recwarn` funcarg warnings will be enabled so that you can properly check for their existence in a cross-python manner).
- refine –pdb: ignore xfailed tests, unify its TB-reporting and don't display failures again at the end.
- fix assertion interpretation with the ** operator (thanks Benjamin Peterson)
- fix issue105 assignment on the same line as a failing assertion (thanks Benjamin Peterson)
- fix issue104 proper escaping for test names in junitxml plugin (thanks anonymous)
- fix issue57 -f|–looponfail to work with xpassing tests (thanks Ronny)
- fix issue92 collectonly reporter and –pastebin (thanks Benjamin Peterson)
- fix py.code.compile(source) to generate unique filenames
- fix assertion re-interp problems on PyPy, by defering code compilation to the (overridable) Frame.eval class. (thanks Amaury Forgeot)
- fix py.path.local.pyimport() to work with directories
- streamline py.path.local.mkdtemp implementation and usage
- don't print empty lines when showing junitxml-filename
- add optional boolean ignore_errors parameter to py.path.local.remove
- fix terminal writing on win32/python2.4
- py.process.cmdexec() now tries harder to return properly encoded unicode objects on all python versions
- install plain py.test/py.which scripts also for Jython, this helps to get canonical script paths in virtualenv situations
- make path.bestrelpath(path) return ".", note that when calling X.bestrelpath the assumption is that X is a directory.
- make initial conftest discovery ignore "–" prefixed arguments
- fix resultlog plugin when used in an multicpu/multihost xdist situation (thanks Jakub Gustak)
- perform distributed testing related reporting in the xdist-plugin rather than having dist-related code in the generic py.test distribution
- fix homedir detection on Windows
- ship distribute_setup.py version 0.6.13

CHAPTER 43

Changes between 1.3.0 and 1.3.1

New features

- issue91: introduce new py.test.xfail(reason) helper to imperatively mark a test as expected to fail. Can be used from within setup and test functions. This is useful especially for parametrized tests when certain configurations are expected-to-fail. In this case the declarative approach with the @py.test.mark.xfail cannot be used as it would mark all configurations as xfail.
- issue102: introduce new –maxfail=NUM option to stop test runs after NUM failures. This is a generalization of the '-x' or '–exitfirst' option which is now equivalent to '–maxfail=1'. Both '-x' and '–maxfail' will now also print a line near the end indicating the Interruption.
- issue89: allow py.test.mark decorators to be used on classes (class decorators were introduced with python2.6) and also allow to have multiple markers applied at class/module level by specifying a list.
- improve and refine letter reporting in the progress bar: . pass f failed test s skipped tests (reminder: use for dependency/platform mismatch only) x xfailed test (test that was expected to fail) X xpassed test (test that was expected to fail but passed)

  You can use any combination of 'fsxX' with the '-r' extended reporting option. The xfail/xpass results will show up as skipped tests in the junitxml output - which also fixes issue99.
- make py.test.cmdline.main() return the exitstatus instead of raising SystemExit and also allow it to be called multiple times. This of course requires that your application and tests are properly teared down and don't have global state.

Fixes / Maintenance

- improved traceback presentation: - improved and unified reporting for "–tb=short" option - Errors during test module imports are much shorter, (using –tb=short style) - raises shows shorter more relevant tracebacks - –fulltrace now more systematically makes traces longer / inhibits cutting

- improve support for raises and other dynamically compiled code by manipulating python's linecache.cache instead of the previous rather hacky way of creating custom code objects. This makes it seemlessly work on Jython and PyPy where it previously didn't.
- fix issue96: make capturing more resilient against Control-C interruptions (involved somewhat substantial refactoring to the underlying capturing functionality to avoid race conditions).
- fix chaining of conditional skipif/xfail decorators - so it works now as expected to use multiple @py.test.mark.skipif(condition) decorators, including specific reporting which of the conditions lead to skipping.
- fix issue95: late-import zlib so that it's not required for general py.test startup.
- fix issue94: make reporting more robust against bogus source code (and internally be more careful when presenting unexpected byte sequences)

CHAPTER 44

Changes between 1.2.1 and 1.3.0

- deprecate –report option in favour of a new shorter and easier to remember -r option: it takes a string argument consisting of any combination of 'xfsX' characters. They relate to the single chars you see during the dotted progress printing and will print an extra line per test at the end of the test run. This extra line indicates the exact position or test ID that you directly paste to the py.test cmdline in order to re-run a particular test.
- allow external plugins to register new hooks via the new pytest_addhooks(pluginmanager) hook. The new release of the pytest-xdist plugin for distributed and looponfailing testing requires this feature.
- add a new pytest_ignore_collect(path, config) hook to allow projects and plugins to define exclusion behaviour for their directory structure - for example you may define in a conftest.py this method:

  ```
  def pytest_ignore_collect(path):
      return path.check(link=1)
  ```

  to prevent even a collection try of any tests in symlinked dirs.
- new pytest_pycollect_makemodule(path, parent) hook for allowing customization of the Module collection object for a matching test module.
- extend and refine xfail mechanism: `@py.test.mark.xfail(run=False)` do not run the decorated test `@py.test.mark.xfail(reason="...")` prints the reason string in xfail summaries specifiying `--runxfail` on command line virtually ignores xfail markers
- expose (previously internal) commonly useful methods: py.io.get_terminal_with() -> return terminal width py.io.ansi_print(...) -> print colored/bold text on linux/win32 py.io.saferepr(obj) -> return limited representation string
- expose test outcome related exceptions as py.test.skip.Exception, py.test.raises.Exception etc., useful mostly for plugins doing special outcome interpretation/tweaking
- (issue85) fix junitxml plugin to handle tests with non-ascii output
- fix/refine python3 compatibility (thanks Benjamin Peterson)
- fixes for making the jython/win32 combination work, note however: jython2.5.1/win32 does not provide a command line launcher, see http://bugs.jython.org/issue1491 . See pylib install documentation for how to work around.

- fixes for handling of unicode exception values and unprintable objects
- (issue87) fix unboundlocal error in assertionold code
- (issue86) improve documentation for looponfailing
- refine IO capturing: stdin-redirect pseudo-file now has a NOP close() method
- ship distribute_setup.py version 0.6.10
- added links to the new capturelog and coverage plugins

CHAPTER 45

Changes between 1.2.1 and 1.2.0

- refined usage and options for “py.cleanup”:

```
py.cleanup     # remove "*.pyc" and "*$py.class" (jython) files
py.cleanup -e .swp -e .cache # also remove files with these extensions
py.cleanup -s  # remove "build" and "dist" directory next to setup.py files
py.cleanup -d  # also remove empty directories
py.cleanup -a  # synonym for "-s -d -e 'pip-log.txt'"
py.cleanup -n  # dry run, only show what would be removed
```

- add a new option “py.test –funcargs” which shows available funcargs and their help strings (docstrings on their respective factory function) for a given test path
- display a short and concise traceback if a funcarg lookup fails
- early-load “conftest.py” files in non-dot first-level sub directories. allows to conveniently keep and access test-related options in a `test` subdir and still add command line options.
- fix issue67: new super-short traceback-printing option: “–tb=line” will print a single line for each failing (python) test indicating its filename, lineno and the failure value
- fix issue78: always call python-level teardown functions even if the according setup failed. This includes refinements for calling setup_module/class functions which will now only be called once instead of the previous behaviour where they’d be called multiple times if they raise an exception (including a Skipped exception). Any exception will be re-corded and associated with all tests in the according module/class scope.
- fix issue63: assume <40 columns to be a bogus terminal width, default to 80
- fix pdb debugging to be in the correct frame on raises-related errors
- update apipkg.py to fix an issue where recursive imports might unnecessarily break importing
- fix plugin links

CHAPTER 46

Changes between 1.2 and 1.1.1

- moved dist/looponfailing from py.test core into a new separately released pytest-xdist plugin.
- new junitxml plugin: –junitxml=path will generate a junit style xml file which is processable e.g. by the Hudson CI system.
- new option: –genscript=path will generate a standalone py.test script which will not need any libraries installed. thanks to Ralf Schmitt.
- new option: –ignore will prevent specified path from collection. Can be specified multiple times.
- new option: –confcutdir=dir will make py.test only consider conftest files that are relative to the specified dir.
- new funcarg: "pytestconfig" is the pytest config object for access to command line args and can now be easily used in a test.
- install 'py.test' and *py.which* with a `-$VERSION` suffix to disambiguate between Python3, python2.X, Jython and PyPy installed versions.
- new "pytestconfig" funcarg allows access to test config object
- new "pytest_report_header" hook can return additional lines to be displayed at the header of a test run.
- (experimental) allow "py.test path::name1::name2::..." for pointing to a test within a test collection directly. This might eventually evolve as a full substitute to "-k" specifications.
- streamlined plugin loading: order is now as documented in customize.html: setuptools, ENV, commandline, conftest. also setuptools entry point names are turned to canonical namees ("pytest_*")
- automatically skip tests that need 'capfd' but have no os.dup
- allow pytest_generate_tests to be defined in classes as well
- deprecate usage of 'disabled' attribute in favour of pytestmark
- deprecate definition of Directory, Module, Class and Function nodes in conftest.py files. Use pytest collect hooks instead.
- collection/item node specific runtest/collect hooks are only called exactly on matching conftest.py files, i.e. ones which are exactly below the filesystem path of an item

- change: the first pytest_collect_directory hook to return something will now prevent further hooks to be called.
- change: figleaf plugin now requires –figleaf to run. Also change its long command line options to be a bit shorter (see py.test -h).
- change: pytest doctest plugin is now enabled by default and has a new option –doctest-glob to set a pattern for file matches.
- change: remove internal py._* helper vars, only keep py._pydir
- robustify capturing to survive if custom pytest_runtest_setup code failed and prevented the capturing setup code from running.
- make py.test.* helpers provided by default plugins visible early - works transparently both for pydoc and for interactive sessions which will regularly see e.g. py.test.mark and py.test.importorskip.
- simplify internal plugin manager machinery
- simplify internal collection tree by introducing a RootCollector node
- fix assert reinterpreation that sees a call containing "keyword=..."
- fix issue66: invoke pytest_sessionstart and pytest_sessionfinish hooks on slaves during dist-testing, report module/session teardown hooks correctly.
- fix issue65: properly handle dist-testing if no execnet/py lib installed remotely.
- skip some install-tests if no execnet is available
- fix docs, fix internal bin/ script generation

CHAPTER 47

Changes between 1.1.1 and 1.1.0

- introduce automatic plugin registration via 'pytest11' entrypoints via setuptools' pkg_resources.iter_entry_points
- fix py.test dist-testing to work with execnet >= 1.0.0b4
- re-introduce py.test.cmdline.main() for better backward compatibility
- svn paths: fix a bug with path.check(versioned=True) for svn paths, allow '%' in svn paths, make svnwc.update() default to interactive mode like in 1.0.x and add svnwc.update(interactive=False) to inhibit interaction.
- refine distributed tarball to contain test and no pyc files
- try harder to have deprecation warnings for py.compat.* accesses report a correct location

CHAPTER 48

Changes between 1.1.0 and 1.0.2

- adjust and improve docs
- remove py.rest tool and internal namespace - it was never really advertised and can still be used with the old release if needed. If there is interest it could be revived into its own tool i guess.
- fix issue48 and issue59: raise an Error if the module from an imported test file does not seem to come from the filepath - avoids "same-name" confusion that has been reported repeatedly
- merged Ronny's nose-compatibility hacks: now nose-style setup_module() and setup() functions are supported
- introduce generalized py.test.mark function marking
- reshuffle / refine command line grouping
- deprecate parser.addgroup in favour of getgroup which creates option group
- add –report command line option that allows to control showing of skipped/xfailed sections
- generalized skipping: a new way to mark python functions with skipif or xfail at function, class and modules level based on platform or sys-module attributes.
- extend py.test.mark decorator to allow for positional args
- introduce and test "py.cleanup -d" to remove empty directories
- fix issue #59 - robustify unittest test collection
- make bpython/help interaction work by adding an __all__ attribute to ApiModule, cleanup initpkg
- use MIT license for pylib, add some contributors
- remove py.execnet code and substitute all usages with 'execnet' proper
- fix issue50 - cached_setup now caches more to expectations for test functions with multiple arguments.
- merge Jarko's fixes, issue #45 and #46
- add the ability to specify a path for py.lookup to search in
- fix a funcarg cached_setup bug probably only occuring in distributed testing and "module" scope with teardown.

- many fixes and changes for making the code base python3 compatible, many thanks to Benjamin Peterson for helping with this.
- consolidate builtins implementation to be compatible with >=2.3, add helpers to ease keeping 2 and 3k compatible code
- deprecate py.compat.doctest|subprocess|textwrap|optparse
- deprecate py.magic.autopath, remove py/magic directory
- move pytest assertion handling to py/code and a pytest_assertion plugin, add "–no-assert" option, deprecate py.magic namespaces in favour of (less) py.code ones.
- consolidate and cleanup py/code classes and files
- cleanup py/misc, move tests to bin-for-dist
- introduce delattr/delitem/delenv methods to py.test's monkeypatch funcarg
- consolidate py.log implementation, remove old approach.
- introduce py.io.TextIO and py.io.BytesIO for distinguishing between text/unicode and byte-streams (uses underlying standard lib io.* if available)
- make py.unittest_convert helper script available which converts "unittest.py" style files into the simpler assert/direct-test-classes py.test/nosetests style. The script was written by Laura Creighton.
- simplified internal localpath implementation

CHAPTER 49

Changes between 1.0.1 and 1.0.2

- fixing packaging issues, triggered by fedora redhat packaging, also added doc, examples and contrib dirs to the tarball.
- added a documentation link to the new django plugin.

CHAPTER 50

Changes between 1.0.0 and 1.0.1

- added a 'pytest_nose' plugin which handles nose.SkipTest, nose-style function/method/generator setup/teardown and tries to report functions correctly.
- capturing of unicode writes or encoded strings to sys.stdout/err work better, also terminalwriting was adapted and somewhat unified between windows and linux.
- improved documentation layout and content a lot
- added a "–help-config" option to show conftest.py / ENV-var names for all longopt cmdline options, and some special conftest.py variables. renamed 'conf_capture' conftest setting to 'option_capture' accordingly.
- fix issue #27: better reporting on non-collectable items given on commandline (e.g. pyc files)
- fix issue #33: added –version flag (thanks Benjamin Peterson)
- fix issue #32: adding support for "incomplete" paths to wcpath.status()
- "Test" prefixed classes are *not* collected by default anymore if they have an __init__ method
- monkeypatch setenv() now accepts a "prepend" parameter
- improved reporting of collection error tracebacks
- simplified multicall mechanism and plugin architecture, renamed some internal methods and argnames

CHAPTER 51

Changes between 1.0.0b9 and 1.0.0

- more terse reporting try to show filesystem path relatively to current dir
- improve xfail output a bit

CHAPTER 52

Changes between 1.0.0b8 and 1.0.0b9

- cleanly handle and report final teardown of test setup
- fix svn-1.6 compat issue with py.path.svnwc().versioned() (thanks Wouter Vanden Hove)
- setup/teardown or collection problems now show as ERRORs or with big "E"'s in the progress lines. they are reported and counted separately.
- dist-testing: properly handle test items that get locally collected but cannot be collected on the remote side - often due to platform/dependency reasons
- simplified py.test.mark API - see keyword plugin documentation
- integrate better with logging: capturing now by default captures test functions and their immediate setup/teardown in a single stream
- capsys and capfd funcargs now have a readouterr() and a close() method (underlyingly py.io.StdCapture/FD objects are used which grew a readouterr() method as well to return snapshots of captured out/err)
- make assert-reinterpretation work better with comparisons not returning bools (reported with numpy from thanks maciej fijalkowski)
- reworked per-test output capturing into the pytest_iocapture.py plugin and thus removed capturing code from config object
- item.repr_failure(excinfo) instead of item.repr_failure(excinfo, outerr)

CHAPTER 53

Changes between 1.0.0b7 and 1.0.0b8

- pytest_unittest-plugin is now enabled by default
- introduced pytest_keyboardinterrupt hook and refined pytest_sessionfinish hooked, added tests.
- workaround a buggy logging module interaction ("closing already closed files"). Thanks to Sridhar Ratnakumar for triggering.
- if plugins use "py.test.importorskip" for importing a dependency only a warning will be issued instead of exiting the testing process.
- many improvements to docs: - refined funcargs doc , use the term "factory" instead of "provider" - added a new talk/tutorial doc page - better download page - better plugin docstrings - added new plugins page and automatic doc generation script
- fixed teardown problem related to partially failing funcarg setups (thanks MrTopf for reporting), "pytest_runtest_teardown" is now always invoked even if the "pytest_runtest_setup" failed.
- tweaked doctest output for docstrings in py modules, thanks Radomir.

CHAPTER 54

Changes between 1.0.0b3 and 1.0.0b7

- renamed py.test.xfail back to py.test.mark.xfail to avoid two ways to decorate for xfail
- re-added py.test.mark decorator for setting keywords on functions (it was actually documented so removing it was not nice)
- remove scope-argument from request.addfinalizer() because request.cached_setup has the scope arg. TOOWTDI.
- perform setup finalization before reporting failures
- apply modified patches from Andreas Kloeckner to allow test functions to have no func_code (#22) and to make "-k" and function keywords work (#20)
- apply patch from Daniel Peolzleithner (issue #23)
- resolve issue #18, multiprocessing.Manager() and redirection clash
- make __name__ == "__channelexec__" for remote_exec code

CHAPTER 55

Changes between 1.0.0b1 and 1.0.0b3

- plugin classes are removed: one now defines hooks directly in conftest.py or global pytest_*.py files.
- added new pytest_namespace(config) hook that allows to inject helpers directly to the py.test.* namespace.
- documented and refined many hooks
- added new style of generative tests via pytest_generate_tests hook that integrates well with function arguments.

CHAPTER 56

Changes between 0.9.2 and 1.0.0b1

- introduced new “funcarg” setup method, see doc/test/funcarg.txt
- introduced plugin architecuture and many new py.test plugins, see doc/test/plugins.txt
- teardown_method is now guaranteed to get called after a test method has run.
- new method: py.test.importorskip(mod,minversion) will either import or call py.test.skip()
- completely revised internal py.test architecture
- new py.process.ForkedFunc object allowing to fork execution of a function to a sub process and getting a result back.

XXX lots of things missing here XXX

CHAPTER 57

Changes between 0.9.1 and 0.9.2

- refined installation and metadata, created new setup.py, now based on setuptools/ez_setup (thanks to Ralf Schmitt for his support).
- improved the way of making py.* scripts available in windows environments, they are now added to the Scripts directory as ".cmd" files.
- py.path.svnwc.status() now is more complete and uses xml output from the 'svn' command if available (Guido Wesdorp)
- fix for py.path.svn* to work with svn 1.5 (Chris Lamb)
- fix path.relto(otherpath) method on windows to use normcase for checking if a path is relative.
- py.test's traceback is better parseable from editors (follows the filenames:LINENO: MSG convention) (thanks to Osmo Salomaa)
- fix to javascript-generation, "py.test –runbrowser" should work more reliably now
- removed previously accidentally added py.test.broken and py.test.notimplemented helpers.
- there now is a py.__version__ attribute

CHAPTER 58

Changes between 0.9.0 and 0.9.1

This is a fairly complete list of changes between 0.9 and 0.9.1, which can serve as a reference for developers.

- allowing + signs in py.path.svn urls [39106]
- fixed support for Failed exceptions without excinfo in py.test [39340]
- added support for killing processes for Windows (as well as platforms that support os.kill) in py.misc.killproc [39655]
- added setup/teardown for generative tests to py.test [40702]
- added detection of FAILED TO LOAD MODULE to py.test [40703, 40738, 40739]
- fixed problem with calling .remove() on wcpaths of non-versioned files in py.path [44248]
- fixed some import and inheritance issues in py.test [41480, 44648, 44655]
- fail to run greenlet tests when pypy is available, but without stackless [45294]
- small fixes in rsession tests [45295]
- fixed issue with 2.5 type representations in py.test [45483, 45484]
- made that internal reporting issues displaying is done atomically in py.test [45518]
- made that non-existing files are igored by the py.lookup script [45519]
- improved exception name creation in py.test [45535]
- made that less threads are used in execnet [merge in 45539]
- removed lock required for atomical reporting issue displaying in py.test [45545]
- removed globals from execnet [45541, 45547]
- refactored cleanup mechanics, made that setDaemon is set to 1 to make atexit get called in 2.5 (py.execnet) [45548]
- fixed bug in joining threads in py.execnet's servemain [45549]
- refactored py.test.rsession tests to not rely on exact output format anymore [45646]

- using repr() on test outcome [45647]
- added 'Reason' classes for py.test.skip() [45648, 45649]
- killed some unnecessary sanity check in py.test.collect [45655]
- avoid using os.tmpfile() in py.io.fdcapture because on Windows it's only usable by Administrators [45901]
- added support for locking and non-recursive commits to py.path.svnwc [45994]
- locking files in py.execnet to prevent CPython from segfaulting [46010]
- added export() method to py.path.svnurl
- fixed -d -x in py.test [47277]
- fixed argument concatenation problem in py.path.svnwc [49423]
- restore py.test behaviour that it exits with code 1 when there are failures [49974]
- don't fail on html files that don't have an accompanying .txt file [50606]
- fixed 'utestconvert.py < input' [50645]
- small fix for code indentation in py.code.source [50755]
- fix _docgen.py documentation building [51285]
- improved checks for source representation of code blocks in py.test [51292]
- added support for passing authentication to py.path.svn* objects [52000, 52001]
- removed sorted() call for py.apigen tests in favour of [].sort() to support Python 2.3 [52481]

CHAPTER 59

Indices and tables

- genindex
- search

Index