Plumbum Shell Combinators Documentation

Release 1.8.2

Tomer Filiba

Jun 01, 2023

CONTENTS

Ever wished the compactness of shell scripts be put into a **real** programming language? Say hello to *Plumbum Shell Combinators*. Plumbum (Latin for *lead*, which was used to create pipes back in the day) is a small yet feature-rich library for shell script-like programs in Python. The motto of the library is **"Never write shell scripts again"**, and thus it attempts to mimic the **shell syntax** (*shell combinators*) where it makes sense, while keeping it all **Pythonic and cross-platform**.

Apart from *shell-like syntax* and *handy shortcuts*, the library provides local and *remote* command execution (over SSH), local and remote file-system *paths*, easy working-directory and environment *manipulation*, quick access to ANSI *colors*, and a programmatic *Command-Line Interface (CLI)* application toolkit. Now let's see some code!

CHAPTER

ONE

NEWS

- **2023.01.01**: Version 1.8.1 released with hatchling replacing setuptools for the build system, and support for Path objects in local.
- **2022.10.05**: Version 1.8.0 released with `NO_COLOR`/`FORCE_COLOR`, `all_markers` & future annotations for the CLI, some command enhancements, & Python 3.11 testing.
- **2021.12.23**: Version 1.7.2 released with very minor fixes, final version to support Python 2.7 and 3.5.
- **2021.11.23**: Version 1.7.1 released with a few features like reverse tunnels, color group titles, and a glob path fix. Better Python 3.10 support.
- **2021.02.08**: Version 1.7.0 released with a few new features like `.with_cwd`, some useful bugfixes, and lots of cleanup.
- **2020.03.23**: Version 1.6.9 released with several Path fixes, final version to support Python 2.6.
- **2019.10.30**: Version 1.6.8 released with `local.cmd`, a few command updates, `Set` improvements, and `TypedEnv`.
- **2018.08.10**: Version 1.6.7 released with several minor additions, mostly to CLI apps, and `run_*` modifiers added.
- **2018.02.12**: Version 1.6.6 released with one more critical bugfix for a error message regression in 1.6.5.
- **2017.12.29**: Version 1.6.5 released with mostly bugfixes, including a critical one that could break pip installs on some platforms. English cli apps now load as fast as before the localization update.
- **2017.11.27**: Version 1.6.4 released with new CLI localization support. Several bugfixes and better pathlib compatibility, along with better separation between Plumbum's internal packages.
- **2016.12.31**: Version 1.6.3 released to provide Python 3.6 compatibility. Mostly bugfixes, several smaller improvements to paths, and a provisional config parser added.
- **2016.12.3**: Version 1.6.2 is now available through conda-forge, as well.
- **2016.6.25**: Version 1.6.2 released. This is mostly a bug fix release, but a few new features are included. Modifiers allow some new arguments, and `Progress` is improved. Better support for SunOS and other OS's.
- **2015.12.18**: Version 1.6.1 released. The release mostly contains smaller fixes for CLI, 2.6/3.5 support, and colors. PyTest is now used for tests, and Conda is supported.
- **2015.10.16**: Version 1.6.0 released. Highlights include Python 3.5 compatibility, the `plumbum.colors` package, `Path` becoming a subclass of `str` and a host of bugfixes. Special thanks go to Henry for his efforts.
- **2015.07.17**: Version 1.5.0 released. This release brings a host of bug fixes, code cleanups and some experimental new features (be sure to check the changelog). Also, say hi to Henry Schreiner, who has joined as a member of the project.
- *Change Log*
- *Quick reference guide*

CHAPTER

TWO

CHEAT SHEET

2.1 Basics

```
>>> from plumbum import local
>>> ls = local["ls"]
>>> ls
LocalCommand(<LocalPath /bin/ls>)
>>> ls()
'build.py\ndist\ndocs\nLICENSE\nplumbum\nREADME.rst\nsetup.py\ntests\ntodo.txt\n'
>>> notepad = local["c:\\windows\\notepad.exe"]
>>> notepad()                                   # Notepad window pops up
''                                              # Notepad window is closed by user,␣
↪command returns
```

Instead of writing `xxx = local["xxx"]` for every program you wish to use, you can also *import commands*:

```
>>> from plumbum.cmd import grep, wc, cat, head
>>> grep
LocalCommand(<LocalPath /bin/grep>)
```

Or, use the `local.cmd` syntactic-sugar:

```
>>> local.cmd.ls
LocalCommand(<LocalPath /bin/ls>)
>>> local.cmd.ls()
'build.py\ndist\ndocs\nLICENSE\nplumbum\nREADME.rst\nsetup.py\ntests\ntodo.txt\n'
```

See *Local Commands*.

2.2 Piping

```
>>> chain = ls["-a"] | grep["-v", "\\.py"] | wc["-l"]
>>> print(chain)
/bin/ls -a | /bin/grep -v '\.py' | /usr/bin/wc -l
>>> chain()
'13\n'
```

See *Pipelining*.

2.3 Redirection

```
>>> ((cat < "setup.py") | head["-n", 4])()
'#!/usr/bin/env python3\nimport os\n\ntry:\n'
>>> (ls["-a"] > "file.list")()
''
>>> (cat["file.list"] | wc["-l"])()
'17\n'
```

See *Input/Output Redirection*.

2.4 Working-directory manipulation

```
>>> local.cwd
<Workdir /home/tomer/workspace/plumbum>
>>> with local.cwd(local.cwd / "docs"):
...     chain()
...
'15\n'
```

A more explicit, and thread-safe way of running a command in a different directory is using the `.with_cwd()` method:

```
.. code-block:: python
```

```
>>> ls_in_docs = local.cmd.ls.with_cwd("docs")
>>> ls_in_docs()
'api\nchangelog.rst\n_cheatsheet.rst\ncli.rst\ncolorlib.rst\n_color_list.html\ncolors.
↪rst\nconf.py\nindex.rst\nlocal_commands.rst\nlocal_machine.rst\nmake.bat\nMakefile\n_
↪news.rst\npaths.rst\nquickref.rst\nremote.rst\n_static\n_templates\ntyped_env.rst\
↪nutils.rst\n'
```

See *Paths* and *The Local Object*.

2.5 Foreground and background execution

```
>>> from plumbum import FG, BG
>>> (ls["-a"] | grep["\\.py"]) & FG         # The output is printed to stdout directly
build.py
.pydevproject
setup.py
>>> (ls["-a"] | grep["\\.py"]) & BG         # The process runs "in the background"
<Future ['/bin/grep', '\\.py'] (running)>
```

See *Background and Foreground*.

2.6 Command nesting

```
>>> from plumbum.cmd import sudo
>>> print(sudo[ifconfig["-a"]])
/usr/bin/sudo /sbin/ifconfig -a
>>> (sudo[ifconfig["-a"]] | grep["-i", "loop"]) & FG
lo        Link encap:Local Loopback
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
```

See *Command Nesting*.

2.7 Remote commands (over SSH)

Supports openSSH-compatible clients, PuTTY (on Windows) and Paramiko (a pure-Python implementation of SSH2):

```
>>> from plumbum import SshMachine
>>> remote = SshMachine("somehost", user = "john", keyfile = "/path/to/idrsa")
>>> r_ls = remote["ls"]
>>> with remote.cwd("/lib"):
...     (r_ls | grep["0.so.0"])()
...
'libusb-1.0.so.0\nlibusb-1.0.so.0.0.0\n'
```

See *Remote*.

2.8 CLI applications

```
import logging
from plumbum import cli

class MyCompiler(cli.Application):
    verbose = cli.Flag(["-v", "--verbose"], help = "Enable verbose mode")
    include_dirs = cli.SwitchAttr("-I", list = True, help = "Specify include directories
↪")

    @cli.switch("-loglevel", int)
    def set_log_level(self, level):
        """Sets the log-level of the logger"""
        logging.root.setLevel(level)

    def main(self, *srcfiles):
        print("Verbose:", self.verbose)
        print("Include dirs:", self.include_dirs)
        print("Compiling:", srcfiles)

if __name__ == "__main__":
    MyCompiler.run()
```

2.8.1 Sample output

```
$ python3 simple_cli.py -v -I foo/bar -Ispam/eggs x.cpp y.cpp z.cpp
Verbose: True
Include dirs: ['foo/bar', 'spam/eggs']
Compiling: ('x.cpp', 'y.cpp', 'z.cpp')
```

See *Command-Line Interface (CLI)*.

2.9 Colors and Styles

```
from plumbum import colors
with colors.red:
    print("This library provides safe, flexible color access.")
    print(colors.bold | "(and styles in general)", "are easy!")
print("The simple 16 colors or",
      colors.orchid & colors.underline | '256 named colors,',
      colors.rgb(18, 146, 64) | "or full rgb colors" ,
      'can be used.')
print("Unsafe " + colors.bg.dark_khaki + "color access" + colors.bg.reset + " is␣
↪available too.")
```

2.9.1 Sample output

See *Colors*.

CHAPTER

THREE

DEVELOPMENT AND INSTALLATION

The library is developed on GitHub, and will happily accept patches from users. Please use the GitHub's built-in issue tracker to report any problem you encounter or to request features. The library is released under the permissive MIT license.

3.1 Requirements

Plumbum supports **Python 3.6-3.10** and **PyPy** and is continually tested on **Linux**, **Mac**, and **Windows** machines through GitHub Actions. Any Unix-like machine should work fine out of the box, but on Windows, you'll probably want to install a decent coreutils environment and add it to your `PATH`, or use WSL(2). I can recommend mingw (which comes bundled with Git for Windows), but cygwin should work too. If you only wish to use Plumbum as a Popen-replacement to run Windows programs, then there's no need for the Unix tools.

Note that for remote command execution, an **openSSH-compatible** client is required (also bundled with *Git for Windows*), and a `bash`-compatible shell and a coreutils environment is also expected on the host machine.

This project uses `setuptools` to build wheels; and `setuptools_scm` is required for building SDists. These dependencies will be handled for you by PEP 518 compatible builders, like build and `pip 10+`.

3.2 Download

You can **download** the library from the Python Package Index (in a variety of formats), or run `pip install plumbum` directly. If you use Anaconda, you can also get it from the `conda-forge` channel with `conda install -c conda-forge plumbum`.

CHAPTER

FOUR

USER GUIDE

The user guide covers most of the features of Plumbum, with lots of code-snippets to get you swimming in no time. It introduces the concepts and "syntax" gradually, so it's recommended you read it in order. A quick *reference guide is available*.

4.1 Local Commands

Plumbum exposes a special singleton object named `local`, which represents your local machine and serves as a factory for command objects:

```
>>> from plumbum import local
>>>
>>> ls = local["ls"]
>>> ls
<LocalCommand C:\Program Files\Git\bin\ls.exe>
>>> notepad = local["c:\\windows\\notepad.exe"]
>>> notepad
<LocalCommand c:\windows\notepad.exe>
```

If you don't specify a full path, the program is searched for in your system's `PATH` (and if no match is found, a `CommandNotFound` exception is raised). Otherwise, the full path is used as given. Once you have a `Command` object, you can execute it like a normal function:

```
>>> ls()
'README.rst\nplumbum\nsetup.py\ntests\ntodo.txt\n'
>>> ls("-a")
'.\n..\n.git\n.gitignore\n.project\n.pydevproject\nREADME.rst\n[...]'
```

For convenience with the common case, you can use the `.cmd` magic property instead of the subscription syntax:

```
>>> ls = local.cmd.ls
>>> ls
<LocalCommand C:\Program Files\Git\bin\ls.exe>
```

New in version 1.7: The `.cmd` commands provider object If you use the `.get()` method instead of `[]`, you can include fallbacks to try if the first command does not exist on the machine. This can be used to get one of several equivalent commands, or it can be used to check for common locations of a command if not in the path. For example:

```
pandoc = local.get('pandoc',
                   '~/AppData/Local/Pandoc/pandoc.exe',
```

(continues on next page)

(continued from previous page)

```
                    '/Program Files/Pandoc/pandoc.exe',
                    '/Program Files (x86)/Pandoc/pandoc.exe')
```

An exception is still raised if none of the commands are found. Unlike `[]` access, an exception will be raised if the executable does not exist.

New in version 1.6: The `.get` method With just a touch of magic, you can *import* commands from the mock module `cmd`, like so:

```
>>> from plumbum.cmd import grep, cat
>>> cat
<LocalCommand C:\Program Files\Git\bin\cat.exe>
```

Note: There's no real module named `plumbum.cmd`; it's a dynamically-created "module", injected into `sys.modules` to enable the use of `from plumbum.cmd import foo`. As of version 1.1, you can actually `import plumbum.cmd`, for consistency, but it's not recommended.

It is important to stress that `from plumbum.cmd import foo` translates to `local["foo"]` behind the scenes.

If underscores (_) appear in the name, and the name cannot be found in the path as-is, the underscores will be replaced by hyphens (-) and the name will be looked up again. This allows you to import `apt_get` for `apt-get`.

4.1.1 Pipelining

In order to form pipelines and other chains, we must first learn to *bind arguments* to commands. As you've seen, *invoking* a command runs the program; by using square brackets (`__getitem__`), we can create bound commands:

```
>>> ls["-l"]
BoundCommand(<LocalCommand C:\Program Files\Git\bin\ls.exe>, ('-l',))
>>> grep["-v", ".py"]
BoundCommand(<LocalCommand C:\Program Files\Git\bin\grep.exe>, ('-v', '.py'))
```

You can think of bound commands as commands that "remember" their arguments. Creating a bound command does not run the program; in order to run it, you'll need to call (invoke) it, like so: `ls["-l"]()` (in fact, `ls["-l"]()` is equivalent to `ls("-l")`).

Now that we can bind arguments to commands, forming pipelines is easy and straight-forwards, using `|` (bitwise-or):

```
>>> chain = ls["-l"] | grep[".py"]
>>> print(chain)
C:\Program Files\Git\bin\ls.exe -l | C:\Program Files\Git\bin\grep.exe .py
>>>
>>> chain()
'-rw-r--r--    1 sebulba  Administ        0 Apr 27 11:54 setup.py\n'
```

Note: Unlike common posix shells, plumbum only captures stderr of the last command in a pipeline. If any of the other commands writes a large amount of text to the stderr, the whole pipeline will stall (large amount equals to >64k on posix systems). This can happen with bioinformatics tools that write progress information to stderr. To avoid this issue, you can discard stderr of the first commands or redirect it to a file.

```
>>> chain = (bwa["mem", ...] >= "/dev/null") | samtools["view", ...]
```

4.1.2 Input/Output Redirection

We can also use redirection into files (or any object that exposes a real `fileno()`). If a string is given, it is assumed to be a file name, and a file with that name is opened for you. In this example, we're reading from `stdin` into `grep world`, and redirecting the output to a file named `tmp.txt`:

```
>>> import sys
>>> ((grep["world"] < sys.stdin) > "tmp.txt")()
hello
hello world
what has the world become?
foo                                  # Ctrl+D pressed
''
```

Note: Parentheses are required here! `grep["world"] < sys.stdin > "tmp.txt"` would be evaluated according to the rules for chained comparison operators and result an exception.

Right after `foo`, Ctrl+D was pressed, which caused `grep` to finish. The empty string at the end is the command's `stdout` (and it's empty because it actually went to a file). Lo and behold, the file was created:

```
>>> cat("tmp.txt")
'hello world\nwhat has the world become?\n'
```

If you need to send input into a program (through its `stdin`), instead of writing the data to a file and redirecting this file into `stdin`, you can use the shortcut << (shift-left):

```
>>> (cat << "hello world\nfoo\nbar\spam" | grep["oo"]) ()
'foo\n'
```

4.1.3 Exit Codes

If the command we're running fails (returns a non-zero exit code), we'll get an exception:

```
>>> cat("non/existing.file")
Traceback (most recent call last):
  [...]
ProcessExecutionError: Unexpected exit code: 1
Command line: | /bin/cat non/existing.file
Stderr:       | /bin/cat: non/existing.file: No such file or directory
```

In order to avoid such exceptions, or when a different exit code is expected, just pass `retcode = xxx` as a keyword argument. If `retcode` is `None`, no exception checking is performed (any exit code is accepted); otherwise, the exit code is expected to match the one you passed:

```
>>> cat("non/existing.file", retcode = None)
''
>>> cat("non/existing.file", retcode = 17)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  [...]
ProcessExecutionError: Unexpected exit code: 1
Command line: | /bin/cat non/existing.file
Stderr:       | /bin/cat: non/existing.file: No such file or directory
```

Note: If you wish to accept several valid exit codes, `retcode` may be a tuple or a list. For instance, `grep("foo", "myfile.txt", retcode = (0, 2))`

If you need to have both the output/error and the exit code (using exceptions would provide either but not both), you can use the `run` method, which will provide all of them

```
>>>  cat["non/existing.file"].run(retcode=None)
(1, '', '/bin/cat: non/existing.file: No such file or directory\n')
```

If you need the value of the exit code, there are two ways to do it. You can call `.run(retcode=None)` (or any other valid retcode value) on a command, you will get a tuple `(retcode, stdout, stderr)` (see *Run and Popen*. If you just need the retcode, or want to check the retcode, there are two special objects that can be applied to your command to run it and get or test the retcode. For example:

```
>>> cat["non/existing.file"] & RETCODE
1
>>> cat["non/existing.file"] & TF
False
>>> cat["non/existing.file"] & TF(1)
True
```

Note: If you want to run these commands in the foreground (see *Background and Foreground*), you can give `FG=True` to `TF` or `RETCODE`. For instance, `cat["non/existing.file"] & TF(1,FG=True)`

New in version 1.5: The `TF` and `RETCODE` modifiers

4.1.4 Run and Popen

Notice that calling commands (or chained-commands) only returns their `stdout`. In order to get hold of the exit code or `stderr`, you'll need to use the *run* method, which returns a 3-tuple of the exit code, `stdout`, and `stderr`:

```
>>> ls.run("-a")
(0, '.\n..\n.git\n.gitignore\n.project\n.pydevproject\nREADME.rst\nplumbum\[...]', '')
```

You can also pass `retcode` as a keyword argument to `run` in the same way discussed above.

And, if you want to want to execute commands "in the background" (i.e., not wait for them to finish), you can use the *popen* method, which returns a normal `subprocess.Popen` object:

```
>>> p = ls.popen("-a")
>>> p.communicate()
('.\n..\n.git\n.gitignore\n.project\n.pydevproject\nREADME.rst\nplumbum\n[...]', '')
```

You can read from its `stdout`, `wait()` for it, `terminate()` it, etc.

4.1.5 Background and Foreground

In order to make programming easier, there are two special objects called FG and BG, which are there to help you. FG runs programs in the foreground (they receive the parent's stdin, stdout and stderr), and BG runs programs in the background (much like popen above, but it returns a *Future* object, instead of a subprocess.Popen one). FG is especially useful for interactive programs like editors, etc., that require a TTY or input from the user.

```
>>> from plumbum import FG, BG
>>> ls["-l"] & FG
total 5
-rw-r--r--    1 sebulba  Administ     4478 Apr 29 15:02 README.rst
drwxr-xr-x    2 sebulba  Administ     4096 Apr 27 12:18 plumbum
-rw-r--r--    1 sebulba  Administ        0 Apr 27 11:54 setup.py
drwxr-xr-x    2 sebulba  Administ        0 Apr 27 11:54 tests
-rw-r--r--    1 sebulba  Administ       18 Apr 27 11:54 todo.txt
```

Note: The output of ls went straight to the screen

```
>>> ls["-a"] & BG
<Future ['C:\\Program Files\\Git\\bin\\ls.exe', '-a'] (running)>
>>> f = _
>>> f.ready()
False
>>> f.wait()
>>> f.stdout
'.\n..\n.git\n.gitignore\n.project\n.pydevproject\nREADME.rst\nplumbum\n[...]'
```

If you want to redirect the output, you can pass those arguments to the BG modifier. So the command ls & BG(stdout=sys.stdout, stderr=sys.stderr) has exactly the same effect as ls & in a terminal.

You can also start a long running process and detach it in nohup mode using the NOHUP modifier:

```
>>> ls["-a"] & NOHUP
```

If you want to redirect the input or output to something other than nohup.out, you can add parameters to the modifier:

```
>>> ls["-a"] & NOHUP(stdout='/dev/null') # Or None
```

New in version 1.6: The NOHUP modifier

You can also use the TEE modifier, which causes output to be redirected to the screen (like FG), but also provides access to the output (like BG).

4.1.6 Command Nesting

The arguments of commands can be strings (or any object that can meaningfully-convert to a string), as we've seen above, but they can also be other **commands**! This allows nesting commands into one another, forming complex command objects. The classic example is sudo:

```
>>> from plumbum.cmd import sudo
>>> print(sudo[ls["-l", "-a"]])
/usr/bin/sudo /bin/ls -l -a
```

(continues on next page)

(continued from previous page)

```
>>> sudo[ls["-l", "-a"]]()
'total 22\ndrwxr-xr-x    8 sebulba  Administ     4096 May  9 20:46 .\n[...]'
```

In fact, you can nest even command-chains (i.e., pipes and redirections), e.g., `sudo[ls | grep["\\.py"]]`; however, that would require that the top-level program be able to handle these shell operators, and this is not the case for `sudo`. `sudo` expects its argument to be an executable program, and it would complain about `|` not being one. So, there's a inherent difference between between `sudo[ls | grep["\\.py"]]` and `sudo[ls] | grep["\\.py"]` (where the pipe is unnested) – the first would fail, the latter would work as expected.

Some programs (mostly shells) will be able to handle pipes and redirections – an example of such a program is `ssh`. For instance, you could run `ssh["somehost", ls | grep["\\.py"]]()`; here, both `ls` and `grep` would run on `somehost`, and only the filtered output would be sent (over SSH) to our machine. On the other hand, an invocation such as `(ssh["somehost", ls] | grep["\\.py"])()` would run `ls` on `somehost`, send its entire output to our machine, and `grep` would filter it locally.

We'll learn more about remote command execution *later*. In the meanwhile, we should learn that command nesting works by *shell-quoting* (or *shell-escaping*) the nested command. Quoting normally takes place from the second level of nesting:

```
>>> print(ssh["somehost", ssh["anotherhost", ls | grep["\\.py"]]])
/bin/ssh somehost /bin/ssh anotherhost /bin/ls '|' /bin/grep "'\\.py'"
```

In this example, we first ssh to `somehost`, from it we ssh to `anotherhost`, and on that host we run the command chain. As you can see, `|` and the backslashes have been quoted, to prevent them from executing on the first-level shell; this way, they would safey get to the second-level shell.

For further information, see the *api docs*.

4.2 Paths

Apart from commands, Plumbum provides an easy to use path class that represents file system paths. Paths are returned from several plumbum commands, and local paths can be directly created by `local.path()`. Paths are always absolute and are immutable, may refer to a remote machine, and can be used like a `str`. In many respects, paths provide a similar API to pathlib in the Python 3.4+ standard library, with a few improvements and extra features.

New in version 1.6: Paths now support more pathlib like syntax, several old names have been depreciated, like `.basename`

The primary ways to create paths are from `.cwd`, `.env.home`, or `.path(...)` on a local or remote machine, with `/`, `//` or `[]` for composition.

Note: The path returned from `.cwd` can also be used in a context manager and has a `.chdir(path)` function. See *The Local Object* for an example.

Paths provide a variety of functions that allow you to check the status of a file:

```
>>> p = local.path("c:\\windows")
>>> p.exists()
True
>>> p.is_dir()
True
```

(continues on next page)

(continued from previous page)

```
>>> p.is_file()
False
```

Besides checking to see if a file exists, you can check the type of file using `.is_dir()`, `is_file()`, or `is_symlink()`. You can access details about the file using the properties `.dirname`, `.drive`, `.root`, `.name`, `.suffix`, and `.stem` (all suffixes). General stats can be obtained with `.stat()`.

You can use `.with_suffix(suffix, depth=1)` to replace the last `depth` suffixes with a new suffix. If you specify None for the depth, it will replace all suffixes (for example, `.tar.gz` is two suffixes). Note that a name like `file.name.10.15.tar.gz` will have "5" suffixes. Also available is `.with_name(name)`, which will will replace the entire name. `preferred_suffix(suffix)` will add a suffix if one does not exist (for default suffix situations).

Paths can be composed using `/` or `[]`:

```
>>> p / "notepad.exe"
<LocalPath c:\windows\notepad.exe>
>>> (p / "notepad.exe").is_file()
True
>>> (p / "notepad.exe").with_suffix(".dll")
<LocalPath c:\windows\notepad.dll>
>>> p["notepad.exe"].is_file()
True
>>> p["../some/path"]["notepad.exe"].with_suffix(".dll")
<LocalPath c:\windows\notepad.dll>
```

You can also iterate over directories to get the contents:

```
>>> for p2 in p:
...     print(p2)
...
c:\windows\addins
c:\windows\appcompat
c:\windows\apppatch
...
```

Paths also supply `.iterdir()`, which may be faster on Python 3.5.

Globing can be easily performed using // (floor division)::

```
>>> p // "*.dll"
[<LocalPath c:\windows\masetupcaller.dll>, ...]
>>> p // "*/*.dll"
[<LocalPath c:\windows\apppatch\acgenral.dll>, ...]
>>> local.cwd / "docs" // "*.rst"
[<LocalPath d:\workspace\plumbum\docs\cli.rst>, ...]
```

New in version 1.6: Globing a tuple will glob for each of the items in the tuple, and return the aggregated result.

Files can be opened and read directly::

```
>>> with(open(local.cwd / "docs" / "index.rst")) as f:
...     print(read(f))
<...output...>
```

New in version 1.6: Support for treating a path exactly like a `str`, so they can be used directly in `open()`.

Paths also supply `.delete()`, `.copy(destination, override=False)`, and `.move(destination)`. On systems that support it, you can also use `.symlink(destination)`, `.link(destination)`, and `.unlink()`. You can change permissions with `.chmod(mode)`, and change owners with `.chown(owner=None, group=None, recursive=None)`. If `recursive` is `None`, this will be recursive only if the path is a directory.

For **copy**, **move**, or **delete** in a more general helper function form, see the *utils modules*.

Relative paths can be computed using `.relative_to(source)` or `mypath - basepath`, though it should be noted that relative paths are not as powerful as absolute paths, and are primarily for recording a path or printing.

For further information, see the *api docs*.

4.3 The Local Object

So far we've only seen running local commands, but there's more to the `local` object than this; it aims to "fully represent" the *local machine*.

First, you should get acquainted with `which`, which performs program name resolution in the system `PATH` and returns the first match (or raises an exception if no match is found):

```
>>> local.which("ls")
<LocalPath C:\Program Files\Git\bin\ls.exe>
>>> local.which("nonexistent")
Traceback (most recent call last):
  [...]
plumbum.commands.CommandNotFound: ('nonexistent', [...])
```

Another member is `python`, which is a command object that points to the current interpreter (`sys.executable`):

```
>>> local.python
<LocalCommand c:\python310\python.exe>
>>> local.python("-c", "import sys;print(sys.version)")
'3.10.0 (default, Feb 2 2022, 02:22:22) [MSC v.1931 64 bit (Intel)]\r\n'
```

4.3.1 Working Directory

The `local.cwd` attribute represents the current working directory. You can change it like so:

```
>>> local.cwd
<Workdir d:\workspace\plumbum>
>>> local.cwd.chdir("d:\\workspace\\plumbum\\docs")
>>> local.cwd
<Workdir d:\workspace\plumbum\docs>
```

You can also use it as a *context manager*, so it behaves like `pushd`/`popd`:

```
>>> ls_l = ls | wc["-l"]
>>> with local.cwd("c:\\windows"):
...     print(f"{local.cwd}:{ls_l()}")
...     with local.cwd("c:\\windows\\system32"):
...         print(f"{local.cwd}:{ls_l()}")
...
c:\windows: 105
```

(continues on next page)

(continued from previous page)

```
c:\windows\system32: 3013
>>> print(f"{local.cwd}:{ls_l()}")
d:\workspace\plumbum: 9
```

Finally, A more explicit and thread-safe way of running a command in a different directory is using the `.with_cwd()` method:

```
>>> ls_in_docs = local.cmd.ls.with_cwd("docs")
>>> ls_in_docs()
'api\nchangelog.rst\n_cheatsheet.rst\ncli.rst\ncolorlib.rst\n_color_list.html\ncolors.
↪rst\nconf.py\nindex.rst\nlocal_commands.rst\nlocal_machine.rst\nmake.bat\nMakefile\n_
↪news.rst\npaths.rst\nquickref.rst\nremote.rst\n_static\n_templates\ntyped_env.rst\
↪nutils.rst\n'
```

4.3.2 Environment

Much like `cwd`, `local.env` represents the *local environment*. It is a dictionary-like object that holds **environment variables**, which you can get/set intuitively:

```
>>> local.env["JAVA_HOME"]
'C:\\Program Files\\Java\\jdk1.6.0_20'
>>> local.env["JAVA_HOME"] = "foo"
```

And similarity to `cwd` is the context-manager nature of `env`; each level would have it's own private copy of the environment:

```
>>> with local.env(FOO="BAR"):
...     local.python("-c", "import os; print(os.environ['FOO'])")
...     with local.env(FOO="SPAM"):
...         local.python("-c", "import os; print(os.environ['FOO'])")
...     local.python("-c", "import os; print(os.environ['FOO'])")
...
'BAR\r\n'
'SPAM\r\n'
'BAR\r\n'
>>> local.python("-c", "import os;print(os.environ['FOO'])")
Traceback (most recent call last):
  [...]
ProcessExecutionError: Unexpected exit code: 1
Command line: | /usr/bin/python3 -c "import os; print(os.environ['FOO'])"
Stderr:       | Traceback (most recent call last):
              |   File "<string>", line 1, in <module>
              |   File "/usr/lib/python3.10/os.py", line 725, in __getitem__
              |     raise KeyError(key) from None
              | KeyError: 'FOO'
```

In order to make cross-platform-ness easier, the `local.env` object provides some convenience properties for getting the username (`.user`), the home path (`.home`), and the executable path (`path`) as a list. For instance:

```
>>> local.env.user
'sebulba'
```

(continues on next page)

(continued from previous page)

```
>>> local.env.home
<Path c:\Users\sebulba>
>>> local.env.path
[<Path c:\python39\lib\site-packages\gtk-2.0\runtime\bin>, <Path c:\Users\sebulba\bin>, .
↪..]
>>>
>>> local.which("python")
<Path c:\python39\python.exe>
>>> local.env.path.insert(0, "c:\\python310")
>>> local.which("python")
<Path c:\python310\python.exe>
```

For further information, see the *api docs*.

4.4 Remote

Just like running local commands, Plumbum supports running commands on remote systems, by executing them over SSH.

4.4.1 Remote Machines

Forming a connection to a remote machine is very straight forward:

```
>>> from plumbum import SshMachine
>>> rem = SshMachine("hostname", user = "john", keyfile = "/path/to/idrsa")
>>> # ...
>>> rem.close()
```

Or as a context-manager:

```
>>> with SshMachine("hostname", user = "john", keyfile = "/path/to/idrsa") as rem:
...     pass
```

Note: `SshMachine` requires `ssh` (openSSH or compatible) installed on your system in order to connect to remote machines. The remote machine must have bash as the default shell (or any shell that supports the `2>&1` syntax for stderr redirection). Alternatively, you can use the pure-Python implementation of *ParamikoMachine*.

Only the `hostname` parameter is required, all other parameters are optional. If the host has your `id-rsa.pub` key in its `authorized_keys` file, or if you've set up your `~/.ssh/config` to login with some user and `keyfile`, you can simply use `rem = SshMachine("hostname")`.

Much like the *local object*, remote machines expose `which()`, `path()`, `python`, `cwd` and `env`. You can also run remote commands, create SSH tunnels, upload/download files, etc. You may also refer to `the full API`, as this guide will only survey the features.

Note: PuTTY users on Windows should use the dedicated `PuttyMachine` instead of `SshMachine`. See also *ParamikoMachine*.

New in version 1.0.1.

Working Directory and Environment

The `cwd` and `env` attributes represent the remote machine's working directory and environment variables, respectively, and can be used to inspect or manipulate them. Much like their local counterparts, they can be used as context managers, so their effects can be contained.

```
>>> rem.cwd
<Workdir /home/john>
>>> with rem.cwd(rem.cwd / "Desktop"):
...     print(rem.cwd)
/home/john/Desktop
>>> rem.env["PATH"]
/bin:/sbin:/usr/bin:/usr/local/bin
>>> rem.which("ls")
<RemotePath /bin/ls>
```

Tunneling

SSH tunneling is a very useful feature of the SSH protocol. It allows you to connect from your machine to a remote server process, while having your connection authenticated and encrypted out-of-the-box. Say you run on `machine-A`, and you wish to connect to a server program running on `machine-B`. That server program binds to `localhost:8888` (where `localhost` refers naturally to to `machine-B`). Using Plumbum, you can easily set up a tunnel from port 6666 on `machine-A` to port 8888 on `machine-B`:

```
>>> tun = rem.tunnel(6666, 8888)
>>> # ...
>>> tun.close()
```

Or as a context manager:

```
>>> with rem.tunnel(6666, 8888):
...     pass
```

You can now connect a socket to `machine-A:6666`, and it will be securely forwarded over SSH to `machine-B:8888`. When the tunnel object is closed, all active connections will be dropped.

4.4.2 Remote Commands

Like local commands, remote commands are created using indexing (`[]`) on a remote machine object. You can either pass the command's name, in which case it will be resolved by through `which`, or the path to the program.

```
>>> rem["ls"]
<RemoteCommand(<RemoteMachine ssh://hostname>, '/bin/ls')>
>>> rem["/usr/local/bin/python3.2"]
<RemoteCommand(<RemoteMachine ssh://hostname>, '/usr/local/bin/python3.2')>
>>> r_ls = rem["ls"]
>>> r_grep = rem["grep"]
>>> r_ls()
'foo\nbar\spam\n'
```

Nesting Commands

Remote commands can be nested just like local ones. In fact, that's how the `SshMachine` operates behind the scenes - it nests each command inside `ssh`. Here are some examples:

```
>>> r_sudo = rem["sudo"]
>>> r_ifconfig = rem["ifconfig"]
>>> print(r_sudo[r_ifconfig["-a"]]())
eth0      Link encap:Ethernet HWaddr ...
[...]
```

You can nest multiple commands, one within another. For instance, you can connect to some machine over SSH and use that machine's SSH client to connect to yet another machine. Here's a sketch:

```
>>> from plumbum.cmd import ssh
>>> print(ssh["localhost", ssh["localhost", "ls"]])
/usr/bin/ssh localhost /usr/bin/ssh localhost ls
>>>
>>> ssh["localhost", ssh["localhost", "ls"]]()
'bin\nDesktop\nDocuments\n...'
```

Piping

Piping works for remote commands as well, but there's a caveat to note here: the plumbing takes place on the local machine! Consider this code for instance

```
>>> r_grep = rem["grep"]
>>> r_ls = rem["ls"]
>>> (r_ls | r_grep["b"])()
'bin\nPublic\n'
```

Although `r_ls` and `r_grep` are remote commands, the data is sent from `r_ls` to the local machine, which then sends it to the remote one for running `grep`. This will be fixed in a future version of Plumbum.

It should be noted, however, that piping remote commands into local ones is perfectly fine. For example, the previous code can be written as

```
>>> from plumbum.cmd import grep
>>> (r_ls | grep["b"])()
'bin\nPublic\n'
```

Which is even more efficient (no need to send data back and forth over SSH).

Redirection

Redirection to and from remote paths is not currently supported, but you can redirect to and from local paths, with the familiar syntax explained in *the corresponding section for local commands*. Note that if the redirection target/source is given as a string, it is automatically interpreted as a path on the local machine.

4.4.3 Paramiko Machine

New in version 1.1.

`SshMachine` relies on the system's `ssh` client to run commands; this means that for each remote command you run, a local process is spawned and an SSH connection is established. While relying on a well-known and trusted SSH client is the most stable option, the incurred overhead of creating a separate SSH connection for each command may be too high. In order to overcome this, Plumbum provides integration for paramiko, an open-source, pure-Python implementation of the SSH2 protocol. This is the `ParamikoMachine`, and it works along the lines of the `SshMachine`:

```
>>> from plumbum.machines.paramiko_machine import ParamikoMachine
>>> rem = ParamikoMachine("192.168.1.143")
>>> rem["ls"]
RemoteCommand(<ParamikoMachine paramiko://192.168.1.143>, <RemotePath /bin/ls>)
>>> r_ls = rem["ls"]
>>> r_ls()
'bin\nDesktop\nDocuments\nDownloads\nexamples.desktop\nMusic\nPictures\n...'
>>> r_ls("-a")
'.\n..\n.adobe\n.bash_history\n.bash_logout\n.bashrc\nbin...'
```

Note: Using `ParamikoMachine` requires paramiko to be installed on your system. Also, you have to explicitly import it (`from plumbum.machines.paramiko_machine import ParamikoMachine`) as paramiko is quite heavy.

Refer to `the API docs` for more details.

The main advantage of using `ParamikoMachine` is that only a single, persistent SSH connection is created, over which commands execute. Moreover, paramiko has a built-in SFTP client, which is used instead of `scp` to copy files (employed by the `.download()`/`.upload()` methods), and tunneling is much more light weight: In the `SshMachine`, a tunnel is created by an external process that lives for as long as the tunnel is to remain active. The `ParamikoMachine`, however, can simply create an extra *channel* on top of the same underlying connection with ease; this is exposed by `connect_sock()`, which creates a tunneled TCP connection and returns a socket-like object

Warning: Piping and input/output redirection don't really work with `ParamikoMachine` commands. You'll get all kinds of errors, like `'ChannelFile' object has no attribute 'fileno'` or `I/O operation on closed file` – this is due to the fact that Paramiko's channels are not real, OS-level files, so they can't interact with `subprocess.Popen`.

This will be solved in a future release; in the meanwhile, you can use the machine's `.session()` method, like so

```
>>> s = mach.session()
>>> s.run("ls | grep b")
(0, 'bin\nPublic\n', '')
```

Tunneling Example

On `192.168.1.143`, I ran the following sophisticated server (notice it's bound to `localhost`):

```
>>> import socket
>>> s=socket.socket()
>>> s.bind(("localhost", 12345))
>>> s.listen(1)
>>> s2,_=s.accept()
>>> while True:
...     data = s2.recv(1000)
...     if not data:
...         break
...     s2.send("I eat " + data)
...
```

On my other machine, I connect (over SSH) to this host and then create a tunneled connection to port 12345, getting back a socket-like object:

```
>>> rem = ParamikoMachine("192.168.1.143")
>>> s = rem.connect_sock(12345)
>>> s.send("carrot")
6
>>> s.recv(1000)
'I eat carrot'
>>> s.send("babies")
6
>>> s.recv(1000)
'I eat babies'
>>> s.close()
```

4.4.4 Remote Paths

Analogous to local paths, remote paths represent a file-system path of a remote system, and expose a set of utility functions for iterating over subpaths, creating subpaths, moving/copying/ renaming paths, etc.

```
>>> p = rem.path("/bin")
>>> p / "ls"
<RemotePath /bin/ls>
>>> (p / "ls").is_file()
True
>>> rem.path("/dev") // "sd*"
[<RemotePath /dev/sda>, < RemotePath /dev/sdb>, <RemotePath /dev/sdb1>, <RemotePath /dev/
↪sdb2>]
```

Note: See the *Utilities* guide for copying, moving and deleting remote paths

For further information, see the *api docs*.

4.5 Utilities

The `utils` module contains a collection of useful utility functions. Note that they are not imported into the namespace of `plumbum` directly, and you have to explicitly import them, e.g. `from plumbum.path.utils import copy`.

- *`copy(src, dst)`* - Copies `src` to `dst` (recursively, if `src` is a directory). The arguments can be either local or remote paths – the function will sort out all the necessary details.
  - If both paths are local, the files are copied locally
  - If one path is local and the other is remote, the function uploads/downloads the files
  - If both paths refer to the same remote machine, the function copies the files locally on the remote machine
  - If both paths refer to different remote machines, the function downloads the files to a temporary location and then uploads them to the destination
- *`move(src, dst)`* - Moves `src` onto `dst`. The arguments can be either local or remote – the function will sort our all the necessary details (as in `copy`)
- *`delete(*paths)`* - Deletes the given sequence of paths; each path may be a string, a local/remote path object, or an iterable of paths. If any of the paths does not exist, the function silently ignores the error and continues. For example

```
from plumbum.path.utils import delete
delete(local.cwd // "*/*.pyc", local.cwd // "*/__pycache__")
```

- `gui_open(path)` - Opens a file in the default editor on Windows, Mac, or Linux. Uses `os.startfile` if available (Windows), `xdg_open` (GNU), or `open` (Mac).

4.6 Command-Line Interface (CLI)

The other side of *executing programs* with ease is **writing CLI programs** with ease. Python scripts normally use `optparse` or the more recent `argparse`, and their derivatives; but all of these are somewhat limited in their expressive power, and are quite **unintuitive** (and even **unpythonic**). Plumbum's CLI toolkit offers a **programmatic approach** to building command-line applications; instead of creating a parser object and populating it with a series of "options", the CLI toolkit translates these primitives into Pythonic constructs and relies on introspection.

From a bird's eye view, CLI applications are classes that extend `plumbum.cli.Application`. They define a `main()` method and optionally expose methods and attributes as command-line `switches`. Switches may take arguments, and any remaining positional arguments are given to the `main` method, according to its signature. A simple CLI application might look like this:

```
from plumbum import cli

class MyApp(cli.Application):
    verbose = cli.Flag(["v", "verbose"], help = "If given, I will be very talkative")

    def main(self, filename):
        print(f"I will now read {filename}")
        if self.verbose:
            print("Yadda " * 200)

if __name__ == "__main__":
    MyApp.run()
```

And you can run it:

```
$ python3 example.py foo
I will now read foo

$ python3 example.py --help
example.py v1.0

Usage: example.py [SWITCHES] filename
Meta-switches:
    -h, --help                  Prints this help message and quits
    --version                   Prints the program's version and quits

Switches:
    -v, --verbose               If given, I will be very talkative
```

So far you've only seen the very basic usage. We'll now start to explore the library.

New in version 1.6.1: You can also directly run the app, as `MyApp()`, without arguments, instead of calling `.main()`.

4.6.1 Application

The `Application` class is the "container" of your application. It consists of the `main()` method, which you should implement, and any number of CLI-exposed switch functions or attributes. The entry-point for your application is the classmethod `run`, which instantiates your class, parses the arguments, invokes all switch functions, and then calls `main()` with the given positional arguments. In order to run your application from the command-line, all you have to do is

```
if __name__ == "__main__":
    MyApp.run()
```

Aside from `run()` and `main()`, the `Application` class exposes two built-in switch functions: `help()` and `version()` which take care of displaying the help and program's version, respectively. By default, `--help` and `-h` invoke `help()`, and `--version` and `-v` invoke `version()`; if any of these functions is called, the application will display the message and quit (without processing any other switch).

You can customize the information displayed by `help()` and `version` by defining class-level attributes, such as `PROGNAME`, `VERSION` and `DESCRIPTION`. For instance,

```
class MyApp(cli.Application):
    PROGNAME = "Foobar"
    VERSION = "7.3"
```

Colors

New in version 1.6.

Colors are supported. You can use a colored string on `PROGNAME`, `VERSION` and `DESCRIPTION` directly. If you set `PROGNAME` to a color, you can get auto-naming and color. The color of the usage string is available as `COLOR_USAGE`. The color of `Usage:` line itself may be specified using `COLOR_USAGE_TITLE`, otherwise it defaults to `COLOR_USAGE`.

Different groups can be colored with a dictionaries `COLOR_GROUPS` and `COLOR_GROUP_TITLES`.

For instance, the following is valid:

```
class MyApp(cli.Application):
    PROGNAME = colors.green
    VERSION = colors.blue | "1.0.2"
    COLOR_GROUPS = {"Switches": colors.blue | "Meta-switches" : colors.yellow}
    COLOR_GROUP_TITLES = {"Switches": colors.bold | colors.blue, "Meta-switches" :␣
↪colors.bold & colors.yellow}
    opts =  cli.Flag("--ops", help=colors.magenta | "This is help")
```

4.6.2 Switch Functions

The decorator `switch` can be seen as the "heart and soul" of the CLI toolkit; it exposes methods of your CLI application as CLI-switches, allowing them to be invoked from the command line. Let's examine the following toy application:

```
class MyApp(cli.Application):
    _allow_root = False       # provide a default

    @cli.switch("--log-to-file", str)
    def log_to_file(self, filename):
        """Sets the file into which logs will be emitted"""
        logger.addHandler(FileHandle(filename))

    @cli.switch(["-r", "--root"])
    def allow_as_root(self):
        """If given, allow running as root"""
        self._allow_root = True

    def main(self):
        if os.geteuid() == 0 and not self._allow_root:
            raise ValueError("cannot run as root")
```

When the program is run, the switch functions are invoked with their appropriate arguments; for instance, `$ ./myapp.py --log-to-file=/tmp/log` would translate to a call to `app.log_to_file("/tmp/log")`. After all switches were processed, control passes to `main`.

Note: Methods' docstrings and argument names will be used to render the help message, keeping your code as DRY as possible.

There's also `autoswitch`, which infers the name of the switch from the function's name, e.g.:

```
@cli.autoswitch(str)
def log_to_file(self, filename):
    pass
```

Will bind the switch function to `--log-to-file`.

Arguments

As demonstrated in the example above, switch functions may take no arguments (not counting `self`) or a single argument. If a switch function accepts an argument, it must specify the argument's *type*. If you require no special validation, simply pass `str`; otherwise, you may pass any type (or any callable, in fact) that will take a string and convert it to a meaningful object. If conversion is not possible, the type (or callable) is expected to raise either `TypeError` or `ValueError`.

For instance:

```
class MyApp(cli.Application):
    _port = 8080

    @cli.switch(["-p"], int)
    def server_port(self, port):
        self._port = port

    def main(self):
        print(self._port)
```

```
$ ./example.py -p 17
17
$ ./example.py -p foo
Argument of -p expected to be <type 'int'>, not 'foo':
    ValueError("invalid literal for int() with base 10: 'foo'",)
```

The toolkit includes two additional "types" (or rather, *validators*): `Range` and `Set`. `Range` takes a minimal value and a maximal value and expects an integer in that range (inclusive). `Set` takes a set of allowed values, and expects the argument to match one of these values. You can set `case_sensitive=False`, or add `all_markers={"*", "all"}` if you want to have a "trigger all markers" marker. Here's an example:

```
class MyApp(cli.Application):
    _port = 8080
    _mode = "TCP"

    @cli.switch("-p", cli.Range(1024,65535))
    def server_port(self, port):
        self._port = port

    @cli.switch("-m", cli.Set("TCP", "UDP", case_sensitive = False))
    def server_mode(self, mode):
        self._mode = mode

    def main(self):
        print(self._port, self._mode)
```

```
$ ./example.py -p 17
Argument of -p expected to be [1024..65535], not '17':
    ValueError('Not in range [1024..65535]',)
$ ./example.py -m foo
Argument of -m expected to be Set('udp', 'tcp'), not 'foo':
    ValueError("Expected one of ['UDP', 'TCP']",)
```

Note: The toolkit also provides some other useful validators: `ExistingFile` (ensures the given argument is an existing file), `ExistingDirectory` (ensures the given argument is an existing directory), and `NonexistentPath` (ensures the given argument is not an existing path). All of these convert the argument to a *local path*.

Repeatable Switches

Many times, you would like to allow a certain switch to be given multiple times. For instance, in `gcc`, you may give several include directories using `-I`. By default, switches may only be given once, unless you allow multiple occurrences by passing `list = True` to the `switch` decorator:

```
class MyApp(cli.Application):
    _dirs = []

    @cli.switch("-I", str, list = True)
    def include_dirs(self, dirs):
        self._dirs = dirs

    def main(self):
        print(self._dirs)
```

```
$ ./example.py -I/foo/bar -I/usr/include
['/foo/bar', '/usr/include']
```

Note: The switch function will be called **only once**, and its argument will be a list of items

Mandatory Switches

If a certain switch is required, you can specify this by passing `mandatory = True` to the `switch` decorator. The user will not be able to run the program without specifying a value for this switch.

Dependencies

Many times, the occurrence of a certain switch depends on the occurrence of another, e.g., it may not be possible to give `-x` without also giving `-y`. This constraint can be achieved by specifying the `requires` keyword argument to the `switch` decorator; it is a list of switch names that this switch depends on. If the required switches are missing, the user will not be able to run the program.

```
class MyApp(cli.Application):
    @cli.switch("--log-to-file", str)
    def log_to_file(self, filename):
        logger.addHandler(logging.FileHandler(filename))

    @cli.switch("--verbose", requires = ["--log-to-file"])
    def verbose(self):
        logger.setLevel(logging.DEBUG)
```

```
$ ./example --verbose
Given --verbose, the following are missing ['log-to-file']
```

Warning: The toolkit invokes the switch functions in the same order in which the switches were given on the command line. It doesn't go as far as computing a topological order on the fly, but this will change in the future.

Mutual Exclusion

Just as some switches may depend on others, some switches mutually-exclude others. For instance, it does not make sense to allow `--verbose` and `--terse`. For this purpose, you can set the `excludes` list in the `switch` decorator.

```
class MyApp(cli.Application):
    @cli.switch("--log-to-file", str)
    def log_to_file(self, filename):
        logger.addHandler(logging.FileHandler(filename))

    @cli.switch("--verbose", requires = ["--log-to-file"], excludes = ["--terse"])
    def verbose(self):
        logger.setLevel(logging.DEBUG)

    @cli.switch("--terse", requires = ["--log-to-file"], excludes = ["--verbose"])
    def terse(self):
        logger.setLevel(logging.WARNING)
```

```
$ ./example --log-to-file=log.txt --verbose --terse
Given --verbose, the following are invalid ['--terse']
```

Grouping

If you wish to group certain switches together in the help message, you can specify `group = "Group Name"`, where `Group Name` is any string. When the help message is rendered, all the switches that belong to the same group will be grouped together. Note that grouping has no other effects on the way switches are processed, but it can help improve the readability of the help message.

4.6.3 Switch Attributes

Many times it's desired to simply store a switch's argument in an attribute, or set a flag if a certain switch is given. For this purpose, the toolkit provides `SwitchAttr`, which is data descriptor that stores the argument in an instance attribute. There are two additional "flavors" of `SwitchAttr`: `Flag` (which toggles its default value if the switch is given) and `CountOf` (which counts the number of occurrences of the switch):

```
class MyApp(cli.Application):
    log_file = cli.SwitchAttr("--log-file", str, default = None)
    enable_logging = cli.Flag("--no-log", default = True)
    verbosity_level = cli.CountOf("-v")

    def main(self):
        print(self.log_file, self.enable_logging, self.verbosity_level)
```

```
$ ./example.py -v --log-file=log.txt -v --no-log -vvv
log.txt False 5
```

Environment Variables

New in version 1.6.

You can also set a `SwitchAttr` to take an environment variable as an input using the envname parameter. For example:

```
class MyApp(cli.Application):
    log_file = cli.SwitchAttr("--log-file", str, envname="MY_LOG_FILE")

    def main(self):
        print(self.log_file)
```

```
$ MY_LOG_FILE=this.log ./example.py
this.log
```

Giving the switch on the command line will override the environment variable value.

4.6.4 Main

The `main()` method takes control once all the command-line switches have been processed. It may take any number of *positional argument*; for instance, in `cp -r /foo /bar`, `/foo` and `/bar` are the *positional arguments*. The number of positional arguments that the program would accept depends on the signature of the method: if the method takes 5 arguments, 2 of which have default values, then at least 3 positional arguments must be supplied by the user and at most 5. If the method also takes varargs (`*args`), the number of arguments that may be given is unbound:

```
class MyApp(cli.Application):
    def main(self, src, dst, mode = "normal"):
        print(src, dst, mode)
```

```
$ ./example.py /foo /bar
/foo /bar normal
$ ./example.py /foo /bar spam
/foo /bar spam
$ ./example.py /foo
Expected at least 2 positional arguments, got ['/foo']
$ ./example.py /foo /bar spam bacon
Expected at most 3 positional arguments, got ['/foo', '/bar', 'spam', 'bacon']
```

Note: The method's signature is also used to generate the help message, e.g.

```
Usage:  [SWITCHES] src dst [mode='normal']
```

With varargs:

```
class MyApp(cli.Application):
    def main(self, src, dst, *eggs):
        print(src, dst, eggs)
```

```
$ ./example.py a b c d
a b ('c', 'd')
$ ./example.py --help
Usage:  [SWITCHES] src dst eggs...
Meta-switches:
    -h, --help                Prints this help message and quits
    -v, --version             Prints the program's version and quits
```

Positional argument validation

New in version 1.6.

You can supply positional argument validators using the `cli.positional` decorator. Simply pass the validators in the decorator matching the names in the main function. For example:

```
class MyApp(cli.Application):
    @cli.positional(cli.ExistingFile, cli.NonexistentPath)
    def main(self, infile, *outfiles):
        "infile is a path, outfiles are a list of paths, proper errors are given"
```

You can also use annotations to specify the validators. For example:

```
class MyApp(cli.Application):
    def main(self, infile : cli.ExistingFile, *outfiles : cli.NonexistentPath):
    "Identical to above MyApp"
```

Annotations are ignored if the positional decorator is present.

Switch Abbreviations

The cli supports switches which have been abbreviated by the user, for example, "–h", "–he", or "–hel" would all match an actual switch name of"–help", as long as no ambiguity arises from multiple switches that might match the same abbreviation. This behavior is disabled by default but can be enabled by defining the class-level attribute `ALLOW_ABBREV` to True. For example:

```
class MyApp(cli.Application):
    ALLOW_ABBREV = True
    cheese = cli.Flag(["cheese"], help = "cheese, please")
    chives = cli.Flag(["chives"], help = "chives, instead")
```

With the above definition, running the following will raise an error due to ambiguity:

```
$ python3 example.py --ch   # error! matches --cheese and --chives
```

However, the following two lines are equivalent:

```
$ python3 example.py --che
$ python3 example.py --cheese
```

4.6.5 Sub-commands

New in version 1.1.

A common practice of CLI applications, as they span out and get larger, is to split their logic into multiple, pluggable *sub-applications* (or *sub-commands*). A classic example is version control systems, such as git, where `git` is the *root* command, under which sub-commands such as `commit` or `push` are nested. Git even supports `alias`-ing, which creates allows users to create custom sub-commands. Plumbum makes writing such applications really easy.

Before we get to the code, it is important to stress out two things:

- Under Plumbum, each sub-command is a full-fledged `cli.Application` on its own; if you wish, you can execute it separately, detached from its so-called root application. When an application is run independently, its `parent` attribute is `None`; when it is run as a sub-command, its `parent` attribute points to its parent application. Likewise, when an parent application is executed with a sub-command, its `nested_command` is set to the nested application; otherwise it's `None`.
- Each sub-command is responsible of **all** arguments that follow it (up to the next sub-command). This allows applications to process their own switches and positional arguments before the nested application is invoked. Take, for instance, `git --foo=bar spam push origin --tags`: the root application, `git`, is in charge of the switch `--foo` and the positional argument `spam`, and the nested application `push` is in charge of the arguments that follow it. In theory, you can nest several sub-applications one into the other; in practice, only a single level is normally used.

Here is an example of a mock version control system, called `geet`. We're going to have a root application `Geet`, which has two sub-commands – `GeetCommit` and `GeetPush`: these are attached to the root application using the `subcommand` decorator

```
class Geet(cli.Application):
    """The l33t version control"""
    VERSION = "1.7.2"

    def main(self, *args):
        if args:
            print(f"Unknown command {args[0]}")
            return 1   # error exit code
        if not self.nested_command:           # will be ``None`` if no sub-command follows
            print("No command given")
            return 1   # error exit code

@Geet.subcommand("commit")                    # attach 'geet commit'
class GeetCommit(cli.Application):
    """creates a new commit in the current branch"""

    auto_add = cli.Flag("-a", help = "automatically add changed files")
    message = cli.SwitchAttr("-m", str, mandatory = True, help = "sets the commit message
↪")

    def main(self):
        print("doing the commit...")

@Geet.subcommand("push")                      # attach 'geet push'
class GeetPush(cli.Application):
    """pushes the current local branch to the remote one"""
    def main(self, remote, branch = None):
        print("doing the push...")
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    Geet.run()
```

Note:

- Since `GeetCommit` is a `cli.Application` on its own right, you may invoke `GeetCommit.run()` directly (should that make sense in the context of your application)
- You can also attach sub-commands "imperatively", using `subcommand` as a method instead of a decorator: `Geet.subcommand("push", GeetPush)`

Here's an example of running this application:

```
$ python3 geet.py --help
geet v1.7.2
The l33t version control

Usage: geet.py [SWITCHES] [SUBCOMMAND [SWITCHES]] args...
Meta-switches:
    -h, --help                Prints this help message and quits
    -v, --version             Prints the program's version and quits

Subcommands:
    commit                    creates a new commit in the current branch; see
                              'geet commit --help' for more info
    push                      pushes the current local branch to the remote
                              one; see 'geet push --help' for more info

$ python3 geet.py commit --help
geet commit v1.7.2
creates a new commit in the current branch

Usage: geet commit [SWITCHES]
Meta-switches:
    -h, --help                Prints this help message and quits
    -v, --version             Prints the program's version and quits

Switches:
    -a                        automatically add changed files
    -m VALUE:str              sets the commit message; required

$ python3 geet.py commit -m "foo"
committing...
```

4.6.6 Configuration parser

Another common task of a cli application is provided by a configuration parser, with an INI backend: `Config` (or `ConfigINI` to explicitly request the INI backend). An example of it's use:

```
from plumbum import cli

with cli.Config('~/.myapp_rc') as conf:
    one = conf.get('one', '1')
    two = conf.get('two', '2')
```

If no configuration file is present, this will create one and each call to `.get` will set the value with the given default. The file is created when the context manager exits. If the file is present, it is read and the values from the file are selected, and nothing is changed. You can also use `[]` syntax to forcibly set a value, or to get a value with a standard `ValueError` if not present. If you want to avoid the context manager, you can use `.read` and `.write` as well.

The ini parser will default to using the `[DEFAULT]` section for values, just like Python's ConfigParser on which it is based. If you want to use a different section, simply separate section and heading with a `.` in the key. `conf['section.item']` would place `item` under `[section]`. All items stored in an `ConfigINI` are converted to `str`, and `str` is always returned.

4.6.7 Terminal Utilities

Several terminal utilities are available in `plumbum.cli.terminal` to assist in making terminal applications.

`get_terminal_size(default=(80,25))` allows cross platform access to the terminal size as a tuple `(width, height)`. Several methods to ask the user for input, such as `readline`, `ask`, `choose`, and `prompt` are available.

`Progress(iterator)` allows you to quickly create a progress bar from an iterator. Simply wrap a slow iterator with this and iterate over it, and it will produce a nice text progress bar based on the user's screen width, with estimated time remaining displayed. If you need to create a progress bar for a fast iterator but with a loop containing code, use `Progress.wrap` or `Progress.range`. For example:

```
for i in Progress.range(10):
    time.sleep(1)
```

If you have something that produces output, but still needs a progress bar, pass `has_output=True` to force the bar not to try to erase the old one each time.

A command line image plotter (`Image`) is provided in `plumbum.cli.image`. It can plot a PIL-like image `im` using:

```
Image().show_pil(im)
```

The Image constructor can take an optional size (defaults to the current terminal size if None), and a `char_ratio`, a height to width measure for your current font. It defaults to a common value of 2.45. If set to None, the ratio is ignored and the image will no longer be constrained to scale proportionately. To directly plot an image, the `show` method takes a filename and a double parameter, which doubles the vertical resolution on some fonts. The `show_pil` and `show_pil_double` methods directly take a PIL-like object. To plot an image from the command line, the module can be run directly: `python3 -m plumbum.cli.image myimage.png`.

For the full list of helpers or more information, see the *api docs*.

4.6.8 See Also

- filecopy.py example
- geet.py – a runnable example of using sub-commands
- RPyC has changed its bash-based build script to Plumbum CLI. Notice how short and readable it is.
- A blog post describing the philosophy of the CLI module

4.7 TypedEnv

Plumbum provides this utility class to facilitate working with environment variables. Similar to how `plumbum.cli.Application` parses command line arguments into pythonic data types, `plumbum.typed_env.TypedEnv` parses environment variables:

class MyEnv(TypedEnv):
username = TypedEnv.Str("USER", default='anonymous') path = TypedEnv.CSV("PATH", separator=":", type=local.path) tmp = TypedEnv.Str(["TMP", "TEMP"]) # support 'fallback' var-names is_travis = TypedEnv.Bool("TRAVIS", default=False) # True is 'yes/true/1' (case-insensitive)

We can now instantiate this class to access its attributes:

```
>>> env = MyEnv()
>>> env.username
'ofer'

>>> env.path
[<LocalPath /home/ofer/bin>,
 <LocalPath /usr/local/bin>,
 <LocalPath /usr/local/sbin>,
 <LocalPath /usr/sbin>,
 <LocalPath /usr/bin>,
 <LocalPath /sbin>,
 <LocalPath /bin>]

>>> env.tmp
Traceback (most recent call last):
  [...]
KeyError: 'TMP'

>>> env.is_travis
False
```

Finally, our `TypedEnv` object allows us ad-hoc access to the rest of the environment variables, using dot-notation:

```
>>> env.HOME
'/home/ofer'
```

We can also update the environment via our `TypedEnv` object:

```
>>> env.tmp = "/tmp"
>>> env.tmp
'/tmp'
```

```
>>> from os import environ
>>> env.TMP
'/tmp'
```

```
>>> env.is_travis = True
>>> env.TRAVIS
'yes'
```

```
>>> env.path = [local.path("/a"), local.path("/b")]
>>> env.PATH
'/a:/b'
```

4.7.1 TypedEnv as an Abstraction Layer

The `TypedEnv` class is very useful for separating your application from the actual environment variables. It provides a layer where parsing and normalizing can take place in a centralized fashion.

For example, you might start with this simple implementation:

```
class CiBuildEnv(TypedEnv):
    job_id = TypedEnv.Str("BUILD_ID")
```

Later, as the application gets more complicated, you may expand your implementation like so:

```
class CiBuildEnv(TypedEnv):
    is_travis = TypedEnv.Bool("TRAVIS", default=False)
    _travis_job_id = TypedEnv.Str("TRAVIS_JOB_ID")
    _jenkins_job_id = TypedEnv.Str("BUILD_ID")

    @property
    def job_id(self):
        return self._travis_job_id if self.is_travis else self._jenkins_job_id
```

4.7.2 TypedEnv vs. local.env

It is important to note that `TypedEnv` is separate and unrelated to the `LocalEnv` object that is provided via `local.env`.

While `TypedEnv` reads and writes directly to `os.environ`, `local.env` is a frozen copy taken at the start of the python session.

While `TypedEnv` is focused on parsing environment variables to be used by the current process, `local.env`'s primary purpose is to manipulate the environment for child processes that are spawned via plumbum's *local commands*.

4.8 Colors

New in version 1.6.

The purpose of the `plumbum.colors` library is to make adding text styles (such as color) to Python easy and safe. Color is often a great addition to shell scripts, but not a necessity, and implementing it properly is tricky. It is easy to end up with an unreadable color stuck on your terminal or with random unreadable symbols around your text. With the color module, you get quick, safe access to ANSI colors and attributes for your scripts. The module also provides an API for creating other color schemes for other systems using escapes.

Note: Enabling color

`ANSIStyle` assumes that only a terminal can display color, and looks at the value of the environment variable `TERM`. You can force the use of color globally by setting `colors.use_color=4` (The levels 0-4 are available, with 0 being off). See this *note* for more options.

4.8.1 Quick start

Colors (`red`, `green`, etc.), attributes (`bold`, `underline`, etc.) and general styles (`warn`, `info`, etc.) are in `plumbum.colors`. Combine styles with `&`, apply to a string with `|`. So, to output a warning you would do

```
from plumbum.colors import warn
print(warn | "This is a warning.")
```

To create a custom style you would do

```
from plumbum import colors
print(colors.green & colors.bold | "This is green and bold.")
```

You can use rgb colors, too:

```
print(colors.rgb(0,255,0) | "This is also green.")
```

4.8.2 Generating Styles

Styles are accessed through the `plumbum.colors` object. This has the following available objects:

`fg` and `bg`
: The foreground and background colors, reset to default with `colors.fg.reset` or `~colors.fg` and likewise for `bg`.

`bold`, `dim`, `underline`, `italics`, `reverse`, `strikeout`, and `hidden`
: All the *ANSI* modifiers are available, as well as their negations, such as `~colors.bold` or `colors.bold.reset`, etc.

`reset`
: The global reset will restore all properties at once.

`do_nothing`
: Does nothing at all, but otherwise acts like any `Style` object. It is its own inverse. Useful for `cli` properties.

Styles loaded from a stylesheet dictionary, such as `warn` and `info`.
These allow you to set standard styles based on behavior rather than colors, and you can load a new stylesheet with `colors.load_stylesheet(...)`.

Recreating and loading the default stylesheet would look like this:

```
>>> default_styles = dict(
...  warn="fg red",
...  title="fg cyan underline bold",
...  fatal="fg red bold",
...  highlight="bg yellow",
...  info="fg blue",
...  success="fg green")

>>> colors.load_stylesheet(default_styles)
```

The `colors.from_ansi(code)` method allows you to create a Style from any ansi sequence, even complex or combined ones.

Colors

The `colors.fg` and `colors.bg` allow you to access and generate colors. Named foreground colors are available directly as methods. The first 16 primary colors, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, etc, as well as `reset`, are available. All 256 color names are available, but do not populate directly, so that auto-completion gives reasonable results. You can also access colors using strings and do `colors.fg[string]`. Capitalization, underscores, and spaces (for strings) will be ignored.

You can also access colors numerically with `colors.fg[n]` for the extended 256 color codes. `colors.fg.rgb(r,g,b)` will create a color from an input red, green, and blue values (integers from 0-255). `colors.fg.rgb(code)` will allow you to input an html style hex sequence.

Anything you can access from `colors.fg` can also be accessed directly from `colors`.

4.8.3 256 Color Support

While this library supports full 24 bit colors through escape sequences, the library has special support for the "full" 256 colorset through numbers, names or HEX html codes. Even if you use 24 bit color, the closest name is displayed in the `repr`. You can access the colors as as `colors.fg.Light_Blue`, `colors.fg.lightblue`, `colors.fg[12]`, `colors.fg('Light_Blue')`, `colors.fg('LightBlue')`, or `colors.fg('#0000FF')`. You can also iterate or slice the `colors`, `colors.fg`, or `colors.bg` objects. Slicing even intelligently downgrades to the simple version of the codes if it is within the first 16 elements. The supported colors are:

If you want to enforce a specific representation, you can use `.basic` (8 color), `.simple` (16 color), `.full` (256 color), or `.true` (24 bit color) on a style, and the colors in that Style will conform to the output representation and name of the best match color. The internal RGB colors are remembered, so this is a non-destructive operation.

Note: Some terminals only support a subset of colors, so keep this in mind when using a larger color set. The standard Ubuntu terminal handles 24 bit color, the Mac terminal only handles 256 colors, and Colorama on Windows only handles 8. See this gist for information about support in terminals. If you need to limit the output color, you can set `colors.use_color` to 0 (no colors), 1 (8 colors), 2 (16 colors), or 3 (256 colors), or 4 (24-bit colors). This option will be automatically guessed for you on initialization.

4.8.4 Style manipulations

Safe color manipulations refer to changes that reset themselves at some point. Unsafe manipulations must be manually reset, and can leave your terminal color in an unreadable state if you forget to reset the color or encounter an exception. The library is smart and will try to restore the color when Python exits.

Note: If you do get the color unset on a terminal, the following, typed into the command line, will restore it:

```
$ python3 -m plumbum.colors
```

This also supports command line access to unsafe color manipulations, such as

```
$ python3 -m plumbum.colors blue
$ python3 -m plumbum.colors bg red
$ python3 -m plumbum.colors fg 123
$ python3 -m plumbum.colors bg reset
$ python3 -m plumbum.colors underline
```

You can use any path or number available as a style.

Unsafe Manipulation

Styles have two unsafe operations: Concatenation (with + and a string) and calling `.now()` without arguments (directly calling a style without arguments is also a shortcut for `.now()`). These two operations do not restore normal color to the terminal by themselves. To protect their use, you can use a context manager around any unsafe operation.

An example of the usage of unsafe `colors` manipulations inside a context manager:

```
from plumbum import colors

with colors:
    colors.fg.red.now()
    print('This is in red')  .. raw:: html

<p><font color="#800000">This is in red</font><br/>
<font color="#008000">This is in green <span style="text-decoration: underline;">and now␣
↪also underlined!</span></font><br/>
<font color="#008000"><span style="text-decoration: underline;">Underlined</span> and␣
↪not underlined but still green.</font><br/>
This is completely restored, even if an exception is thrown! </p>

    colors.green.now()
    print('This is green ' + colors.underline + 'and now also underlined!')
    print('Underlined' + colors.underline.reset + ' and not underlined but still red')
print('This is completely restored, even if an exception is thrown!')
```

Output:

We can use `colors` instead of `colors.fg` for foreground colors. If we had used `colors.fg` as the context manager, then non-foreground properties, such as `colors.underline` or `colors.bg.yellow`, would not have been reset. Each attribute, as well as `fg`, `bg`, and `colors` all have inverses in the ANSI standard. They are accessed with `~` or `.reset`, and can be used to manually make these operations safer, but there is a better way.

Safe Manipulation

All other operations are safe; they restore the color automatically. The first, and hopefully already obvious one, is using a specific style rather than a `colors` or `colors.fg` object in a `with` statement. This will set the color (using `sys.stdout` by default) to that color, and restore color on leaving.

The second method is to manually wrap a string. This can be done with `color | "string"` or `color["string"]`. These produce strings that can be further manipulated or printed.

Finally, you can also print a color to stdout directly using `color.print("string")`. This has the same syntax as the print function.

An example of safe manipulations:

```
colors.fg.yellow('This is yellow', end='')
print(' And this is normal again.')
with colors.red:
    print('Red color!')
    with colors.bold:
        print("This is red and bold.")
    print("Not bold, but still red.")
print("Not red color or bold.")
print(colors.magenta & colors.bold | "This is bold and colorful!", "And this is not.")
```

Output:

Style Combinations

You can combine styles with & and they will create a new combined style. Colors will not be "summed" or otherwise combined; the rightmost color will be used (this matches the expected effect of applying the styles individually to the strings). However, combined styles are intelligent and know how to reset just the properties that they contain. As you have seen in the example above, the combined style `(colors.magenta & colors.bold)` can be used in any way a normal style can.

4.8.5 New color systems

The library was written primarily for ANSI color sequences, but can also easily be subclassed to create new color systems. See *Colorlib design* for information on how the system works. An HTML version is available as `plumbum.colorlib.htmlcolors`.

4.8.6 See Also

- colored Another library with 256 color support
- colorful A fairly new library with a similar feature set
- **colorama A library that supports colored text on Windows,**
  can be combined with Plumbum.colors (if you force `use_color`, doesn't support all extended colors)
- rich A very powerful modern library for all sorts of styling.

4.9 Change Log

4.9.1 1.8.2

- Fix author metadata on PyPI package and add static check (#648)
- Add testing for Python 3.12 beta 1 (#650)
- Use Ruff for linting (#643)
- Paths: Add type hinting for Path (#646)

4.9.2 1.8.1

- Accept path-like objects (#627)
- Move the build backend to hatchling and hatch-vcs. Users should be unaffected. Third-party packaging may need to adapt to the new build system. (#607)

4.9.3 1.8.0

- Drop Python 2.7 and 3.5 support, add 3.11 support (#573)
- Lots of extended checks and fixes for problems exposed.
- Color: support `NO_COLOR/FORCE_COLOR` (#575)
- Commands: New `iter_lines buffer_size` parameter (#582)
- Commands: cache remote commands (#583)
- SSH: Support reverse tunnels and dynamically allocated ports (#608)
- CLI: add `Set(..., all_markers={"*", "all"})` and fix support for other separators (#619)
- CLI: support future annotations (#621)
- Color: fix the ABC (#617)
- Exceptions: fix for exception pickling (#586)
- Fix for StdinDataRedirection and modifiers (#605)

4.9.4 1.7.2

- Commands: avoid issue mktemp issue on some BSD variants (#571)
- Better specification of dependency on pywin32 (#568)
- Some DeprecationWarnings changed to FutureWarnings (#567)

4.9.5 1.7.1

- Paths: glob with local paths no longer expands the existing path too (#552)
- Paramiko: support reverse tunnels (#562)
- SSHMachine: support forwarding Unix sockets in `.tunnel()` (#550)
- CLI: Support `COLOR_GROUP_TITLES` (#553)
- Fix a deprecated in Python 3.10 warning (#563)
- Extended testing and checking on Python 3.10 and various PyPy versions. Nox is supported for easier new-user development.

4.9.6 1.7.0

- Commands: support `.with_cwd()` (#513)
- Commands: make `iter_lines` deal with decoding errors during iteration (#525)
- Commands: fix handling of env-vars passed to plumbum BoundEnvCommands (#513)
- Commands: fix support for win32 in `iter_lines` (#500)
- Paths: fix incorrect `__getitem__` method in Path (#506)
- Paths: Remote path stat had odd OSError (#505)
- Paths: Fix `RemotePath.copy()` (#527)
- Paths: missing `__fspath__` added (#498)
- SSH: better error reporting on SshSession error (#515)
- Internal: redesigned CI, major cleanup to setuptools distribution, Black formatting, style checking throughout.

4.9.7 1.6.9

- Last version to support Python 2.6; added python_requires for future versions (#507)
- Paths: Fix bug with subscription operations (#498), (#506)
- Paths: Fix resolve (#492)
- Commands: Fix resolve (#491)
- Commands: Add context manager on popen (#495)
- Several smaller fixes (#500), (#505)

4.9.8 1.6.8

- Exceptions: Changed ProcessExecutionError's formatting to be more user-friendly (#456)
- Commands: support for per-line timeout with iter_lines (#454)
- Commands: support for piping stdout/stderr to a logger (#454)
- Paths: support composing paths using subscription operations (#455)
- CLI: Improved 'Set' validator to allow non-string types, and CSV params (#452)

- TypedEnv: Facility for modeling environment-variables into python data types (#451)
- Commands: execute local/remote commands via a magic `.cmd` attribute (#450)

4.9.9 1.6.7

- Commands: Added `run_*` methods as an alternative to modifiers (#386)
- CLI: Added support for `ALLOW_ABREV` (#401)
- CLI: Added `DESCRIPTION_MORE`, preserves spacing (#378)
- Color: Avoid throwing error in atexit in special cases (like pytest) (#393)
- Including Python 3.7 in testing matrix.
- Smaller bugfixes and other testing improvements.

4.9.10 1.6.6

- Critical Bugfix: High-speed (English) translations could break the CLI module (#371)
- Small improvement to wheels packaging

4.9.11 1.6.5

- Critical Bugfix: Syntax error in image script could break pip installs (#366)
- CLI: Regression fix: English apps now load as fast as in 1.6.3 (#364)
- CLI: Missing colon restored in group names
- Regression fix: Restored non-setuptools installs (but really, why would you not have setuptools?) (#367)

4.9.12 1.6.4

- CLI: Support for localization (#339), with:
  - Russian by Pavel Pletenev (#339)
  - Dutch by Roel Aaij (#351)
  - French by Joel Closier (#352)
  - German by Christoph Hasse (#353)
  - Pulls with more languages welcome!
- CLI: Support for `MakeDirectory` (#339)
- Commands: Fixed unicode input/output on Python 2 (#341)
- Paths: More updates for pathlib compatibility (#325)
- Terminal: Changed `prompt()`'s default value for `type` parameter from `int` to `str` to match existing docs (#327)
- Remote: Support ~ in PATH for a remote (#293)
- Remote: Fixes for globbing with spaces in filename on a remote server (#322)
- Color: Fixes to image plots, better separation (#324)

- Python 3.3 has been removed from Travis and Appveyor.
- Several bugs fixed

4.9.13 1.6.3

- Python 3.6 is now supported, critical bug fixed (#302)
- Commands: Better handling of return codes for pipelines (#288)
- Paths: Return split support (regression) (#286) - also supports dummy args for better `str` compatibility
- Paths: Added support for Python 3.6 path protocol
- Paths: Support Python's `in` syntax
- CLI: Added Config parser (provisional) (#304)
- Color: image plots with `python -m plumbum.cli.image` (#304)
- SSH: No longer hangs for `timeout` seconds on failure (#306)
- Test improvements, especially on non-linux systems

4.9.14 1.6.2

- CLI: `Progress` now has a clear keyword that hides the bar on completion
- CLI: `Progress` without clear now starts on next line without having to manually add `\n`.
- Commands: modifiers now accept a timeout parameter (#281)
- Commands: `BG` modifier now allows `stdout`/`stderr` redirection (to screen, for example) (#258)
- Commands: Modifiers no longer crash on repr (see #262)
- Remote: `nohup` works again, typo fixed (#261)
- Added better support for SunOS and other OS's. (#260)
- Colors: Context manager flushes stream now, provides more consistent results
- Other smaller bugfixes, better support for Python 3.6+

4.9.15 1.6.1

- CLI: `Application` subclasses can now be run directly, instead of calling `.run()`, to facilitate using as entry points (#237)
- CLI: `gui_open` added to allow easy opening of paths in default gui editor (#239)
- CLI: More control over help message (#233)
- Remote: `cwd` is now stashed to reduce network usage (similar to Plumbum <1.6 behavior), and absolute paths are faster, (#238)
- Bugfix: Pipelined return codes now give correct attribution (#243)
- Bugfix: `Progress` works on Python 2.6 (#230)
- Bugfix: Colors now work with more terminals (#231)
- Bugfix: Getting an executable no longer returns a directory (#234)

- Bugfix: Iterdir now works on Python <3.5
- Testing is now expanded and fully written in Pytest, with coverage reporting.
- Added support for Conda (as of 1.6.2, use the `-c conda-forge` channel)

4.9.16 1.6.0

- Added support for Python 3.5, PyPy, and better Windows and Mac support, with CI testing (#218, #217, #226)
- Colors: Added colors module, support for colors added to cli (#213)
- Machines: Added `.get()` method for checking several commands. (#205)
- Machines: `local.cwd` now is the current directory even if you change the directory with non-Plumbum methods (fixes unexpected behavior). (#207)
- SSHMachine: Better error message for SSH (#211)
- SSHMachine: Support for FreeBSD remote (#220)
- Paths: Now a subclass of `str`, can be opened directly (#228)
- Paths: Improved pathlib compatibility with several additions and renames (#223)
- Paths: Added globbing multiple patterns at once (#221)
- Commands: added `NOHUP` modifier (#221)
- CLI: added positional argument validation (#225)
- CLI: added `envname`, which allows you specify an environment variable for a `SwitchAttr` (#216)
- CLI terminal: added `Progress`, a command line progress bar for iterators and ranges (#214)
- Continued to clean out Python 2.5 hacks

4.9.17 1.5.0

- Removed support for Python 2.5. (Travis CI does not support it anymore)
- CLI: add `invoke`, which allows you to programmatically run applications (#149)
- CLI: add `--help-all` and various cosmetic fixes: (#125), (#126), (#127)
- CLI: add `root_app` property (#141)
- Machines: `getattr` now raises `AttributeError` instead of `CommandNotFound` (#135)
- Paramiko: `keep_alive` support (#186)
- Paramiko: does not support piping explicitly now (#160)
- Parmaiko: Added pure SFTP backend, gives STFP v4+ support (#188)
- Paths: bugfix to `cwd` interaction with `Path` (#142)
- Paths: read/write now accept an optional encoding parameter (#148)
- Paths: Suffix support similar to the Python 3.4 standard library `pathlib` (#198)
- Commands: renamed `setenv` to `with_env` (#143)
- Commands: pipelines will now fail with `ProcessExecutionError` if the source process fails (#145)
- Commands: added `TF` and `RETCODE` modifiers (#202)

- Experimental concurrent machine support in `experimental/parallel.py`
- Several minor bug fixes, including Windows and Python 3 fixes (#199, #195)

4.9.18 1.4.2

- Paramiko now supports Python 3, enabled support in Plumbum
- Terminal: added `prompt()`, bugfix to `get_terminal_size()` (#113)
- CLI: added `cleanup()`, which is called after `main()` returns
- CLI: bugfix to `CountOf` (#118)
- Commands: Add a TEE modifier (#117)
- Remote machines: bugfix to `which`, bugfix to remote environment variables (#122)
- Path: `read()`/`write()` now operate on bytes

4.9.19 1.4.1

- Force `/bin/sh` to be the shell in `SshMachine.session()` (#111)
- Added `islink()` and `unlink()` to path objects (#100, #103)
- Added `access` to path objects
- Faster `which` implementation (#98)
- Several minor bug fixes

4.9.20 1.4

- Moved `atomic` and `unixutils` into the new `fs` package (file-system related utilities)
- Dropped `plumbum.utils` legacy shortcut in favor of `plumbum.path.utils`
- Bugfix: the left-hand-side process of a pipe wasn't waited on, leading to zombies (#89)
- Added `RelativePath` (the result of `Path.relative_to`)
- Fixed more text alignment issues in `cli.Application.help()`
- Introduced `ask()` and `choose` to `cli.terminal`
- Bugfix: Path comparison operators were wrong
- Added connection timeout to `RemoteMachine`

4.9.21 1.3

- `Command.popen`: a new argument, `new_session` may be passed to `Command.popen`, which runs the given in a new session (`setsid` on POSIX, `CREATE_NEW_PROCESS_GROUP` on Windows)
- `Command.Popen`: args can now also be a list (previously, it was required to be a tuple). See
- `local.daemonize`: run commands as full daemons (double-fork and `setsid`) on POSIX systems, or detached from their controlling console and parent (on Windows).
- `list_processes`: return a list of running process (local/remote machines)

- `SshMachine.nohup`: "daemonize" remote commands via `nohup` (not really a daemon, but good enough)
- `atomic`: Atomic file operations (`AtomicFile`, `AtomicCounterFile` and `PidFile`)
- `copy` and `move`: the `src` argument can now be a list of files to move, e.g., `copy(["foo", "bar"], "/usr/bin")`
- list local and remote processes
- cli: better handling of text wrapping in the generated help message
- cli: add a default `main()` method that checks for unknown subcommands
- terminal: initial commit (`get_terminal_size`)
- packaging: the package was split into subpackages; it grew too big for a flat namespace. imports are not expected to be broken by this change
- SshMachine: added `password` parameter, which relies on sshpass to feed the password to `ssh`. This is a security risk, but it's occasionally necessary. Use this with caution!
- Commands now have a `machine` attribute that points to the machine they run on
- Commands gained `setenv`, which creates a command with a bound environment
- Remote path: several fixes to `stat` (`StatRes`)
- cli: add lazily-loaded subcommands (e.g., `MainApp.subcommand("foo", "my.package.foo.FooApp")`), which are imported on demand
- Paths: added relative_to and split, which (respectively) computes the difference between two paths and splits paths into lists of nodes
- cli: `Predicate` became a class decorator (it exists solely for pretty-printing anyway)
- PuttyMachine: bugfix

4.9.22 1.2

- Path: added chmod
- Path: added link and symlink
- Path: `walk()` now applies filter recursively (#64)
- Commands: added Append redirect
- Commands: fix `_subprocess` issue (#59)
- Commands: add `__file__` to module hack (#66)
- Paramiko: add 'username' and 'password'
- Paramiko: add 'timeout' and 'look_for_keys'
- Python 3: fix #56 and #55

4.9.23 1.1

- Paramiko integration (#10)
- CLI: now with built-in support for sub-commands. See also: #43
- The "import hack" has moved to the package's `__init__.py`, to make it importable directly (#45)
- Paths now support `chmod` (on POSIX platform) (#49)
- The argument name of a `SwitchAttr` can now be given to it (defaults to `VALUE`) (#46)

4.9.24 1.0.1

- Windows: path are no longer converted to lower-case, but `__eq__` and `__hash__` operate on the lower-cased result (#38)
- Properly handle empty strings in the argument list (#41)
- Relaxed type-checking of `LocalPath` and `RemotePath` (#35)
- Added `PuttyMachine` for Windows users that relies on `plink` and `pscp` (instead of `ssh` and `scp`) (#37)

4.9.25 1.0.0

- Rename `cli.CountingAttr` to `cli.CountOf`
- Moved to Travis continuous integration
- Added `unixutils`
- Added `chown` and `uid/gid`
- Lots of fixes and updates to the doc
- Full list of issues

4.9.26 0.9.0

Initial release

4.10 Quick reference guide

This is a cheatsheet for common tasks in Plumbum.

4.10.1 CLI

Optional arguments

Utility	Usage
`Flag`	True or False descriptor
`SwitchAttr`	A value as a descriptor
`CountOf`	Counting version of `Flag`
`@switch`	A function that runs when passed
`@autoswitch`	A switch that gets its name from the function decorated
`@validator`	A positional argument validator on main (or use Py3 attributes)

Validators

Anything that produces a `ValueError` or `TypeError`, is applied to argument. Some special ones included:

Validator	Usage
`Range`	A number in some range
`Set`	A choice in a set
`ExistingFile`	A file (converts to Path)
`ExistingDirectory`	A directory
`NonexistentPath`	Not a file or directory

Common options

Option	Used in	Usage
First argument	Non-auto	The name, or list of names, includes dash(es)
Second argument	All	The validator
docstring	`switch`, `Application`	The help message
`help=`	All	The help message
`list=True`	`switch`	Allow multiple times (passed as list)
`requires=`	All	A list of optional arguments to require
`excludes=`	All	A list of optional arguments to exclude
`group=`	All	The name of a group
`default=`	All	The default if not given
`envname=`	`SwitchAttr`	The name of an environment variable to check
`mandatory=True`	Switches	Require this argument to be passed

Special member variables

4.10.2 Paths

Idiom	Description
`local.cwd`	Common way to make paths
/ Construct	Composition of parts
// Construct	Grep for files
Sorting	Alphabetical
Iteration	By parts
To str	Canonical full path
Subtraction	Relative path
`in`	Check for file in folder

Property	Description	Compare to Pathlib
`.name`	The file name	✓
`.basename`	DEPRECATED	
`.stem`	Name without extension	✓
`.dirname`	Directory name	
`.root`	The file tree root	✓
`.drive`	Drive letter (Windows)	✓
`.suffix`	The suffix	✓
`.suffixes`	A list of suffixes	✓
`.uid`	User ID	
`.gid`	Group ID	
`.parts`	Tuple of `split`	✓
`.parents`	The ancestors of the path	✓
`.parent`	The ancestor of the path	✓

Method	Description	Compare to Pathlib
`.up(count = 1)`	Go up count directories	
`.walk(filter=*, dir_filter=*)`	Traverse directories	
`.as_uri(scheme=None)`	Universal Resource ID	✓
`.join(part, ...)`	Put together paths (/)	`.joinpath`
`.list()`	Files in directory	(shortcut)
`.iterdir()`	Fast iterator over dir	✓
`.is_dir()`	If path is dir	✓
`.isdir()`	DEPRECATED	
`.is_file()`	If is file	✓
`.isfile()`	DEPRECATED	
`.is_symlink()`	If is symlink	✓
`.islink()`	DEPRECATED	
`.exists()`	If file exists	✓
`.stat()`	Return OS stats	✓
`.with_name(name)`	Replace filename	✓
`.with_suffix(suffix, depth=1)`	Replace suffix	✓ (no depth)
`.preferred_suffix(suffix)`	Replace suffix if no suffix	
`.glob(pattern)`	Search for pattern	✓
`.split()`	Split into directories	`.parts`
`.relative_to(source)`	Relative path (-)	✓
`.resolve(strict=False)`	Does nothing	✓
`.access(mode = 0)`	Check access permissions	

Method (changes files)	Description	Compare to Pathlib
`.link(dst)`	Make a hard link	
`.symlink(dst)`	Make a symlink	`.symlink_to`
`.unlink()`	Unlink a file (delete)	✓
`.delete()`	Delete file	`.unlink`
`.move(dst)`	Move file	
`.rename(newname)`	Change the file name	✓
`.copy(dst, override=False)`	Copy a file	
`.mkdir()`	Make a directory	✓ (+ more args)
`.open(mode="r")`	Open a file for reading	✓ (+ more args)
`.read(encoding=None)`	Read a file to text	`.read_text`
`.write(data, encoding=None)`	Write to a file	`.write_text`
`.touch()`	Touch a file	✓ (+ more args)
`.chown(owner=None, group=None, recursive=None)`	Change owner	
`.chmod(mode)`	Change permissions	✓

4.10.3 Colors

You pick colors from `fg` or `bg`, also can `reset`

Main colors: `black` `red` `green` `yellow` `blue` `magenta` `cyan` `white`

Default styles: `warn` `title` `fatal` `highlight` `info` `success`

Attrs: `bold` `dim` `underline` `italics` `reverse` `strikeout` `hidden`

CHAPTER

FIVE

API REFERENCE

The API reference (generated from the *docstrings* within the library) covers all of the exposed APIs of the library. Note that some "advanced" features and some function parameters are missing from the guide, so you might want to consult with the API reference in these cases.

5.1 Package plumbum.cli

exception plumbum.cli.application.**ShowHelp**

exception plumbum.cli.application.**ShowHelpAll**

exception plumbum.cli.application.**ShowVersion**

class plumbum.cli.application.**Application**(*executable=None*)

The base class for CLI applications; your "entry point" class should derive from it, define the relevant switch functions and attributes, and the `main()` function. The class defines two overridable "meta switches" for version (`-v`, `--version`) help (`-h`, `--help`), and help-all (`--help-all`).

The signature of the main function matters: any positional arguments (e.g., non-switch arguments) given on the command line are passed to the `main()` function; if you wish to allow unlimited number of positional arguments, use varargs (`*args`). The names of the arguments will be shown in the help message.

The classmethod `run` serves as the entry point of the class. It parses the command-line arguments, invokes switch functions and enters `main`. You should **not override** this method.

Usage:

```
class FileCopier(Application):
    stat = Flag("p", "copy stat info as well")

    def main(self, src, dst):
        if self.stat:
            shutil.copy2(src, dst)
        else:
            shutil.copy(src, dst)

if __name__ == "__main__":
    FileCopier.run()
```

There are several class-level attributes you may set:

- `PROGNAME` - the name of the program; if `None` (the default), it is set to the name of the executable (`argv[0]`); can be in color. If only a color, will be applied to the name.

- VERSION - the program's version (defaults to 1.0, can be in color)
- DESCRIPTION - a short description of your program (shown in help). If not set, the class' __doc__ will be used. Can be in color.
- DESCRIPTION_MORE - a detailed description of your program (shown in help). The text will be printed by paragraphs (specified by empty lines between them). The indentation of each paragraph will be the indentation of its first line. List items are identified by their first non-whitespace character being one of '-', '*', and '/'; so that they are not combined with preceding paragraphs. Bullet '/' is "invisible", meaning that the bullet itself will not be printed to the output.
- USAGE - the usage line (shown in help).
- COLOR_USAGE_TITLE - The color of the usage line's header.
- COLOR_USAGE - The color of the usage line.
- COLOR_GROUPS - A dictionary that sets colors for the groups, like Meta-switches, Switches, and Subcommands.
- COLOR_GROUP_TITLES - A dictionary that sets colors for the group titles. If the dictionary is empty, it defaults to COLOR_GROUPS.
- SUBCOMMAND_HELPMSG - Controls the printing of extra "see subcommand -h" help message. Default is a message, set to False to remove.
- ALLOW_ABBREV - Controls whether partial switch names are supported, for example '–ver' will match '–verbose'. Default is False for backward consistency with previous plumbum releases. Note that ambiguous abbreviations will not match, for example if –foothis and –foothat are defined, then –foo will not match.

A note on sub-commands: when an application is the root, its parent attribute is set to None. When it is used as a nested-command, parent will point to its direct ancestor. Likewise, when an application is invoked with a sub-command, its nested_command attribute will hold the chosen sub-application and its command-line arguments (a tuple); otherwise, it will be set to None

classmethod unbind_switches(**switch_names*)

Unbinds the given switch names from this application. For example

```
class MyApp(cli.Application):
    pass
MyApp.unbind_switches("--version")
```

classmethod subcommand(*name*, *subapp=None*)

Registers the given sub-application as a sub-command of this one. This method can be used both as a decorator and as a normal classmethod:

```
@MyApp.subcommand("foo")
class FooApp(cli.Application):
    pass
```

Or

```
MyApp.subcommand("foo", FooApp)
```

New in version 1.1.

New in version 1.3: The sub-command can also be a string, in which case it is treated as a fully-qualified class name and is imported on demand. For example,

MyApp.subcommand("foo", "fully.qualified.package.FooApp")

classmethod autocomplete(*argv*)

This is supplied to make subclassing and testing argument completion methods easier

classmethod run(*argv=None*, *exit=True*)

Runs the application, taking the arguments from `sys.argv` by default if nothing is passed. If `exit` is `True` (the default), the function will exit with the appropriate return code; otherwise it will return a tuple of `(inst, retcode)`, where `inst` is the application instance created internally by this function and `retcode` is the exit code of the application.

Note: Setting `exit` to `False` is intendend for testing/debugging purposes only – do not override it in other situations.

---

classmethod invoke(**args*, ***switches*)

Invoke this application programmatically (as a function), in the same way `run()` would. There are two key differences: the return value of `main()` is not converted to an integer (returned as-is), and exceptions are not swallowed either.

Parameters

- **args** – any positional arguments for `main()`
- **switches** – command-line switches are passed as keyword arguments, e.g., `foo=5` for `--foo=5`

main(**args*)

Implement me (no need to call super)

cleanup(*retcode*)

Called after `main()` and all sub-applications have executed, to perform any necessary cleanup.

Parameters
retcode – the return code of `main()`

helpall()

Prints help messages of all sub-commands and quits

help()

Prints this help message and quits

version()

Prints the program's version and quits

exception plumbum.cli.switches.**SwitchError**

A general switch related-error (base class of all other switch errors)

exception plumbum.cli.switches.**PositionalArgumentsError**

Raised when an invalid number of positional arguments has been given

exception plumbum.cli.switches.**SwitchCombinationError**

Raised when an invalid combination of switches has been given

exception plumbum.cli.switches.**UnknownSwitch**

Raised when an unrecognized switch has been given

exception plumbum.cli.switches.**MissingArgument**

Raised when a switch requires an argument, but one was not provided

exception plumbum.cli.switches.**MissingMandatorySwitch**

Raised when a mandatory switch has not been given

exception plumbum.cli.switches.**WrongArgumentType**

Raised when a switch expected an argument of some type, but an argument of a wrong type has been given

exception plumbum.cli.switches.**SubcommandError**

Raised when there's something wrong with sub-commands

plumbum.cli.switches.**switch**(*names*, *argtype=None*, *argname=None*, *list=False*, *mandatory=False*, *requires=()*, *excludes=()*, *help=None*, *overridable=False*, *group='Switches'*, *envname=None*)

A decorator that exposes functions as command-line switches. Usage:

```
class MyApp(Application):
    @switch(["-l", "--log-to-file"], argtype = str)
    def log_to_file(self, filename):
        handler = logging.FileHandler(filename)
        logger.addHandler(handler)

    @switch(["--verbose"], excludes=["--terse"], requires=["--log-to-file"])
    def set_debug(self):
        logger.setLevel(logging.DEBUG)

    @switch(["--terse"], excludes=["--verbose"], requires=["--log-to-file"])
    def set_terse(self):
        logger.setLevel(logging.WARNING)
```

Parameters

- **names** – The name(s) under which the function is reachable; it can be a string or a list of string, but at least one name is required. There's no need to prefix the name with `-` or `--` (this is added automatically), but it can be used for clarity. Single-letter names are prefixed by `-`, while longer names are prefixed by `--`
- **envname** – Name of environment variable to extract value from, as alternative to argv
- **argtype** – If this function takes an argument, you need to specify its type. The default is `None`, which means the function takes no argument. The type is more of a "validator" than a real type; it can be any callable object that raises a `TypeError` if the argument is invalid, or returns an appropriate value on success. If the user provides an invalid value, `plumbum.cli.WrongArgumentType()`
- **argname** – The name of the argument; if `None`, the name will be inferred from the function's signature
- **list** – Whether or not this switch can be repeated (e.g. `gcc -I/lib -I/usr/lib`). If `False`, only a single occurrence of the switch is allowed; if `True`, it may be repeated indefinitely. The occurrences are collected into a list, so the function is only called once with the collections. For instance, for `gcc -I/lib -I/usr/lib`, the function will be called with `["/lib", "/usr/lib"]`.
- **mandatory** – Whether or not this switch is mandatory; if a mandatory switch is not given, `MissingMandatorySwitch` is raised. The default is `False`.
- **requires** – A list of switches that this switch depends on ("requires"). This means that it's invalid to invoke this switch without also invoking the required ones. In the example above, it's illegal to pass `--verbose` or `--terse` without also passing `--log-to-file`.

By default, this list is empty, which means the switch has no prerequisites. If an invalid combination is given, `SwitchCombinationError` is raised.

Note that this list is made of the switch *names*; if a switch has more than a single name, any of its names will do.

Note: There is no guarantee on the (topological) order in which the actual switch functions will be invoked, as the dependency graph might contain cycles.

- **excludes** – A list of switches that this switch forbids ("excludes"). This means that it's invalid to invoke this switch if any of the excluded ones are given. In the example above, it's illegal to pass `--verbose` along with `--terse`, as it will result in a contradiction. By default, this list is empty, which means the switch has no prerequisites. If an invalid combination is given, `SwitchCombinationError` is raised.

  Note that this list is made of the switch *names*; if a switch has more than a single name, any of its names will do.

- **help** – The help message (description) for this switch; this description is used when `--help` is given. If `None`, the function's docstring will be used.
- **overridable** – Whether or not the names of this switch are overridable by other switches. If `False` (the default), having another switch function with the same name(s) will cause an exception. If `True`, this is silently ignored.
- **group** – The switch's *group*; this is a string that is used to group related switches together when `--help` is given. The default group is `Switches`.

Returns

The decorated function (with a `_switch_info` attribute)

`plumbum.cli.switches.`**autoswitch**(**args*, ***kwargs*)

A decorator that exposes a function as a switch, "inferring" the name of the switch from the function's name (converting to lower-case, and replacing underscores with hyphens). The arguments are the same as for `switch`.

`class plumbum.cli.switches.`**SwitchAttr**(*names*, *argtype=<class 'str'>*, *default=None*, *list=False*, *argname='VALUE'*, ***kwargs*)

A switch that stores its result in an attribute (descriptor). Usage:

```
class MyApp(Application):
    logfile = SwitchAttr(["-f", "--log-file"], str)

    def main(self):
        if self.logfile:
            open(self.logfile, "w")
```

Parameters

- **names** – The switch names
- **argtype** – The switch argument's (and attribute's) type
- **default** – The attribute's default value (`None`)
- **argname** – The switch argument's name (default is `"VALUE"`)
- **kwargs** – Any of the keyword arguments accepted by `switch`

class plumbum.cli.switches.**Flag**(*names*, *default=False*, ***kwargs*)

A specialized `SwitchAttr` for boolean flags. If the flag is not given, the value of this attribute is `default`; if it is given, the value changes to `not default`. Usage:

```
class MyApp(Application):
    verbose = Flag(["-v", "--verbose"], help = "If given, I'll be very talkative")
```

Parameters

- **names** – The switch names
- **default** – The attribute's initial value (`False` by default)
- **kwargs** – Any of the keyword arguments accepted by `switch`, except for `list` and `argtype`.

class plumbum.cli.switches.**CountOf**(*names*, *default=0*, ***kwargs*)

A specialized `SwitchAttr` that counts the number of occurrences of the switch in the command line. Usage:

```
class MyApp(Application):
    verbosity = CountOf(["-v", "--verbose"], help = "The more, the merrier")
```

If `-v -v -vv` is given in the command-line, it will result in `verbosity = 4`.

Parameters

- **names** – The switch names
- **default** – The default value (0)
- **kwargs** – Any of the keyword arguments accepted by `switch`, except for `list` and `argtype`.

class plumbum.cli.switches.**positional**(**args*, ***kargs*)

Runs a validator on the main function for a class. This should be used like this:

```
class MyApp(cli.Application):
    @cli.positional(cli.Range(1,10), cli.ExistingFile)
    def main(self, x, *f):
        # x is a range, f's are all ExistingFile's)
```

Or, Python 3 only:

```
class MyApp(cli.Application):
    def main(self, x : cli.Range(1,10), *f : cli.ExistingFile):
        # x is a range, f's are all ExistingFile's)
```

If you do not want to validate on the annotations, use this decorator (even if empty) to override annotation validation.

Validators should be callable, and should have a `.choices()` function with possible choices. (For future argument completion, for example)

Default arguments do not go through the validator.

#TODO: Check with MyPy

class plumbum.cli.switches.**Validator**

choices(*partial=''*)

Should return set of valid choices, can be given optional partial info

class plumbum.cli.switches.**Range**(*start*, *end*)

A switch-type validator that checks for the inclusion of a value in a certain range. Usage:

```
class MyApp(Application):
    age = SwitchAttr(["--age"], Range(18, 120))
```

Parameters

- **start** – The minimal value
- **end** – The maximal value

choices(*partial=''*)

Should return set of valid choices, can be given optional partial info

class plumbum.cli.switches.**Set**(**values: str | Callable[[str], str]*, *case_sensitive: bool = False*, *csv: bool | str = False*, *all_markers: collections.abc.Set[str] = frozenset({})*)

A switch-type validator that checks that the value is contained in a defined set of values. Usage:

```
class MyApp(Application):
    mode = SwitchAttr(["--mode"], Set("TCP", "UDP", case_sensitive = False))
    num = SwitchAttr(["--num"], Set("MIN", "MAX", int, csv = True))
```

Parameters

- **values** – The set of values (strings), or other callable validators, or types, or any other object that can be compared to a string.
- **case_sensitive** – A keyword argument that indicates whether to use case-sensitive comparison or not. The default is `False`
- **csv** – splits the input as a comma-separated-value before validating and returning a list. Accepts `True`, `False`, or a string for the separator
- **all_markers** – When a user inputs any value from this set, all values are iterated over. Something like {"*", "all"} would be a potential setting for this option.

choices(*partial=''*)

Should return set of valid choices, can be given optional partial info

class plumbum.cli.switches.**Predicate**(*func*)

A wrapper for a single-argument function with pretty printing

5.1.1 Terminal-related utilities

plumbum.cli.terminal.**readline**(*message: str = ''*) → str

Gets a line of input from the user (stdin)

plumbum.cli.terminal.**ask**(*question: str*, *default: bool | None = None*) → bool

Presents the user with a yes/no question.

Parameters

- **question** – The question to ask
- **default** – If `None`, the user must answer. If `True` or `False`, lack of response is interpreted as the default option

Returns
the user's choice

plumbum.cli.terminal.**choose**(*question*, *options*, *default=None*)

Prompts the user with a question and a set of options, from which the user needs to choose.

Parameters

- **question** – The question to ask
- **options** – A set of options. It can be a list (of strings or two-tuples, mapping text to returned-object) or a dict (mapping text to returned-object).``
- **default** – If None, the user must answer. Otherwise, lack of response is interpreted as this answer

Returns
The user's choice

Example:

```
ans = choose("What is your favorite color?", ["blue", "yellow", "green"], default =
↪"yellow")
# `ans` will be one of "blue", "yellow" or "green"

ans = choose("What is your favorite color?",
        {"blue" : 0x0000ff, "yellow" : 0xffff00 , "green" : 0x00ff00}, default =␣
↪0x00ff00)
# this will display "blue", "yellow" and "green" but return a numerical value
```

plumbum.cli.terminal.**prompt**(*question*, *type=<class 'str'>*, *default=NotImplemented*, *validator=<function <lambda>>*)

Presents the user with a validated question, keeps asking if validation does not pass.

Parameters

- **question** – The question to ask
- **type** – The type of the answer, defaults to str
- **default** – The default choice
- **validator** – An extra validator called after type conversion, can raise ValueError or return False to trigger a retry.

Returns
the user's choice

plumbum.cli.terminal.**get_terminal_size**(*default: Tuple[int, int] = (80, 25)*) → Tuple[int, int]

Get width and height of console; works on linux, os x, windows and cygwin

Adapted from https://gist.github.com/jtriley/1108174 Originally from: http://stackoverflow.com/questions/566746/how-to-get-console-window-width-in-python

class plumbum.cli.terminal.**Progress**(*iterator=None*, *length=None*, *timer=True*, *body=False*, *has_output=False*, *clear=True*)

start()

This should initialize the progress bar and the iterator

done()

Is called when the iterator is done.

display()

Called to update the progress bar

5.1.2 Terminal size utility

plumbum.cli.termsize.**get_terminal_size**(*default: Tuple[int, int] = (80, 25)*) → Tuple[int, int]

Get width and height of console; works on linux, os x, windows and cygwin

Adapted from https://gist.github.com/jtriley/1108174 Originally from: http://stackoverflow.com/questions/566746/how-to-get-console-window-width-in-python

5.1.3 Progress bar

class plumbum.cli.progress.**ProgressBase**(*iterator=None*, *length=None*, *timer=True*, *body=False*, *has_output=False*, *clear=True*)

Base class for progress bars. Customize for types of progress bars.

Parameters

- **iterator** – The iterator to wrap with a progress bar
- **length** – The length of the iterator (will use __len__ if None)
- **timer** – Try to time the completion status of the iterator
- **body** – True if the slow portion occurs outside the iterator (in a loop, for example)
- **has_output** – True if the iteration body produces output to the screen (forces rewrite off)
- **clear** – Clear the progress bar afterwards, if applicable.

abstract start()

This should initialize the progress bar and the iterator

property value

This is the current value, as a property so setting it can be customized

abstract display()

Called to update the progress bar

increment()

Sets next value and displays the bar

time_remaining()

Get the time remaining for the progress bar, guesses

str_time_remaining()

Returns a string version of time remaining

abstract done()

Is called when the iterator is done.

classmethod range(**value*, ***kargs*)

Fast shortcut to create a range based progress bar, assumes work done in body

classmethod wrap(*iterator*, *length=None*, ***kargs*)

Shortcut to wrap an iterator that does not do all the work internally

class plumbum.cli.progress.**Progress**(*iterator=None*, *length=None*, *timer=True*, *body=False*, *has_output=False*, *clear=True*)

start()

This should initialize the progress bar and the iterator

done()

Is called when the iterator is done.

display()

Called to update the progress bar

class plumbum.cli.progress.**ProgressIPy**(**args*, ***kargs*)

start()

This should initialize the progress bar and the iterator

property value

This is the current value, -1 allowed (automatically fixed for display)

display()

Called to update the progress bar

done()

Is called when the iterator is done.

class plumbum.cli.progress.**ProgressAuto**(**args*, ***kargs*)

Automatically selects the best progress bar (IPython HTML or text). Does not work with qtconsole (as that is correctly identified as identical to notebook, since the kernel is the same); it will still iterate, but no graphical indication will be displayed.

Parameters

- **iterator** – The iterator to wrap with a progress bar
- **length** – The length of the iterator (will use `__len__` if None)
- **timer** – Try to time the completion status of the iterator
- **body** – True if the slow portion occurs outside the iterator (in a loop, for example)

5.2 Package plumbum.commands

plumbum.commands.base.**iter_lines**(*proc*, *retcode=0*, *timeout=None*, *linesize=-1*, *line_timeout=None*, *buffer_size=None*, *mode=None*, *_iter_lines=<function _iter_lines_posix>*)

Runs the given process (equivalent to run_proc()) and yields a tuples of (out, err) line pairs. If the exit code of the process does not match the expected one, `ProcessExecutionError` is raised.

Parameters

- **retcode** – The expected return code of this process (defaults to 0). In order to disable exit-code validation, pass `None`. It may also be a tuple (or any iterable) of expected exit codes.

- **timeout** – The maximal amount of time (in seconds) to allow the process to run. None means no timeout is imposed; otherwise, if the process hasn't terminated after that many seconds, the process will be forcefully terminated an exception will be raised
- **linesize** – Maximum number of characters to read from stdout/stderr at each iteration. -1 (default) reads until a b'n' is encountered.
- **line_timeout** – The maximal amount of time (in seconds) to allow between consecutive lines in either stream. Raise an ProcessLineTimedOut if the timeout has been reached. None means no timeout is imposed.
- **buffer_size** – Maximum number of lines to keep in the stdout/stderr buffers, in case of a ProcessExecutionError. Default is None, which defaults to DEFAULT_BUFFER_SIZE (which is infinite by default). 0 will disable bufferring completely.
- **mode** – Controls what the generator yields. Defaults to DEFAULT_ITER_LINES_MODE (which is BY_POSITION by default) - BY_POSITION (default): yields (out, err) line tuples, where either item may be None - BY_TYPE: yields (fd, line) tuples, where fd is 1 (stdout) or 2 (stderr)

Returns

An iterator of (out, err) line tuples.

plumbum.commands.base.**run_proc**(*proc*, *retcode*, *timeout=None*)

Waits for the given process to terminate, with the expected exit code

Parameters

- **proc** – a running Popen-like object, with all the expected methods.
- **retcode** – the expected return (exit) code of the process. It defaults to 0 (the convention for success). If None, the return code is ignored. It may also be a tuple (or any object that supports __contains__) of expected return codes.
- **timeout** – the number of seconds (a float) to allow the process to run, before forcefully terminating it. If None, not timeout is imposed; otherwise the process is expected to terminate within that timeout value, or it will be killed and ProcessTimedOut will be raised

Returns

A tuple of (return code, stdout, stderr)

plumbum.commands.base.**shquote**(*text*)

Quotes the given text with shell escaping (assumes as syntax similar to sh)

exception plumbum.commands.base.**RedirectionError**

Raised when an attempt is made to redirect an process' standard handle, which was already redirected to/from a file

__weakref__

list of weak references to the object (if defined)

class plumbum.commands.base.**BaseCommand**

Base of all command objects

__str__()

Return str(self).

__or__(*other*)

Creates a pipe with the other command

__gt__(*file*)

Redirects the process' stdout to the given file

__rshift__(*file*)

Redirects the process' stdout to the given file (appending)

__ge__(*file*)

Redirects the process' stderr to the given file

__lt__(*file*)

Redirects the given file into the process' stdin

__lshift__(*data*)

Redirects the given data into the process' stdin

__getitem__(*args*)

Creates a bound-command with the given arguments. Shortcut for bound_command.

bound_command(**args*)

Creates a bound-command with the given arguments

__call__(**args*, ***kwargs*)

A shortcut for *run(args)*, returning only the process' stdout

with_env(***env*)

Returns a BoundEnvCommand with the given environment variables

with_cwd(*path*)

Returns a BoundEnvCommand with the specified working directory. This overrides a cwd specified in a wrapping *machine.cwd()* context manager.

setenv(***env*)

Returns a BoundEnvCommand with the given environment variables

formulate(*level=0*, *args=()*)

Formulates the command into a command-line, i.e., a list of shell-quoted strings that can be executed by Popen or shells.

Parameters

- **level** – The nesting level of the formulation; it dictates how much shell-quoting (if any) should be performed
- **args** – The arguments passed to this command (a tuple)

Returns

A list of strings

popen(*args=()*, ***kwargs*)

Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)

- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A Popen-like object

nohup(*cwd='.'*, *stdout='nohup.out'*, *stderr=None*, *append=True*)

Runs a command detached.

bgrun(*args=()*, ***kwargs*)

Runs the given command as a context manager, allowing you to create a pipeline (not in the UNIX sense) of programs, parallelizing their work. In other words, instead of running programs one after the other, you can start all of them at the same time and wait for them to finish. For a more thorough review, see Lightweight Asynchronism.

Example:

```
from plumbum.cmd import mkfs

with mkfs["-t", "ext3", "/dev/sda1"] as p1:
    with mkfs["-t", "ext3", "/dev/sdb1"] as p2:
        pass
```

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

For the arguments, see *run*.

Returns
A Popen object, augmented with a `.run()` method, which returns a tuple of (return code, stdout, stderr)

run(*args=()*, ***kwargs*)

Runs the given command (equivalent to popen() followed by run_proc). If the exit code of the process does not match the expected one, ProcessExecutionError is raised.

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **retcode** – The expected return code of this process (defaults to 0). In order to disable exit-code validation, pass None. It may also be a tuple (or any iterable) of expected exit codes.

  Note: this argument must be passed as a keyword argument.

- **timeout** – The maximal amount of time (in seconds) to allow the process to run. None means no timeout is imposed; otherwise, if the process hasn't terminated after that many seconds, the process will be forcefully terminated an exception will be raised

  ---

  Note: this argument must be passed as a keyword argument.

- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A tuple of (return code, stdout, stderr)

`run_bg`(***kwargs*)

Run this command in the background. Uses all arguments from the BG construct :py:class: *plumbum.commands.modifiers.BG*

`run_fg`(***kwargs*)

Run this command in the foreground. Uses all arguments from the FG construct :py:class: *plumbum.commands.modifiers.FG*

`run_tee`(***kwargs*)

Run this command using the TEE construct. Inherits all arguments from TEE :py:class: *plumbum.commands.modifiers.TEE*

`run_tf`(***kwargs*)

Run this command using the TF construct. Inherits all arguments from TF :py:class: *plumbum.commands.modifiers.TF*

`run_retcode`(***kwargs*)

Run this command using the RETCODE construct. Inherits all arguments from RETCODE :py:class: *plumbum.commands.modifiers.RETCODE*

`run_nohup`(***kwargs*)

Run this command using the NOHUP construct. Inherits all arguments from NOHUP :py:class: *plumbum.commands.modifiers.NOHUP*

`class plumbum.commands.base.Pipeline`(*srccmd*, *dstcmd*)

`__init__`(*srccmd*, *dstcmd*)

`__repr__`()

Return repr(self).

`formulate`(*level=0*, *args=()*)

Formulates the command into a command-line, i.e., a list of shell-quoted strings that can be executed by `Popen` or shells.

Parameters

- **level** – The nesting level of the formulation; it dictates how much shell-quoting (if any) should be performed
- **args** – The arguments passed to this command (a tuple)

Returns
A list of strings

`popen`(*args=()*, ***kwargs*)

Spawns the given command, returning a `Popen`-like object.

Note: When processes run in the **background** (either via `popen` or `& BG`), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)

- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A Popen-like object

class plumbum.commands.base.**BaseRedirection**(*cmd*, *file*)

__init__(*cmd*, *file*)

__repr__()
Return repr(self).

formulate(*level=0*, *args=()*)
Formulates the command into a command-line, i.e., a list of shell-quoted strings that can be executed by Popen or shells.

Parameters

- **level** – The nesting level of the formulation; it dictates how much shell-quoting (if any) should be performed
- **args** – The arguments passed to this command (a tuple)

Returns
A list of strings

popen(*args=()*, ***kwargs*)
Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A Popen-like object

class plumbum.commands.base.**BoundCommand**(*cmd*, *args*)

__init__(*cmd*, *args*)

__repr__()
Return repr(self).

formulate(*level=0*, *args=()*)
Formulates the command into a command-line, i.e., a list of shell-quoted strings that can be executed by Popen or shells.

Parameters

- **level** – The nesting level of the formulation; it dictates how much shell-quoting (if any) should be performed
- **args** – The arguments passed to this command (a tuple)

Returns
A list of strings

popen(*args=()*, ***kwargs*)

Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A Popen-like object

class plumbum.commands.base.**BoundEnvCommand**(*cmd*, *env=None*, *cwd=None*)

__init__(*cmd*, *env=None*, *cwd=None*)

__repr__()

Return repr(self).

formulate(*level=0*, *args=()*)

Formulates the command into a command-line, i.e., a list of shell-quoted strings that can be executed by Popen or shells.

Parameters

- **level** – The nesting level of the formulation; it dictates how much shell-quoting (if any) should be performed
- **args** – The arguments passed to this command (a tuple)

Returns
A list of strings

popen(*args=()*, *cwd=None*, *env=None*, ***kwargs*)

Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A Popen-like object

class plumbum.commands.base.**ConcreteCommand**(*executable*, *encoding*)

__init__(*executable*, *encoding*)

__str__()

Return str(self).

__repr__()

Return repr(self).

formulate(*level=0*, *args=()*)

Formulates the command into a command-line, i.e., a list of shell-quoted strings that can be executed by Popen or shells.

Parameters

- **level** – The nesting level of the formulation; it dictates how much shell-quoting (if any) should be performed
- **args** – The arguments passed to this command (a tuple)

Returns

A list of strings

popen(*args=()*, ***kwargs*)

Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns

A Popen-like object

class plumbum.commands.base.**StdinRedirection**(*cmd*, *file*)

class plumbum.commands.base.**StdoutRedirection**(*cmd*, *file*)

class plumbum.commands.base.**StderrRedirection**(*cmd*, *file*)

class plumbum.commands.base.**AppendingStdoutRedirection**(*cmd*, *file*)

class plumbum.commands.base.**StdinDataRedirection**(*cmd*, *data*)

__init__(*cmd*, *data*)

formulate(*level=0*, *args=()*)

Formulates the command into a command-line, i.e., a list of shell-quoted strings that can be executed by Popen or shells.

Parameters

- **level** – The nesting level of the formulation; it dictates how much shell-quoting (if any) should be performed

- **args** – The arguments passed to this command (a tuple)

Returns
A list of strings

popen(*args=()*, ***kwargs*)

Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A Popen-like object

class plumbum.commands.modifiers.**Future**(*proc*, *expected_retcode*, *timeout=None*)

Represents a "future result" of a running process. It basically wraps a Popen object and the expected exit code, and provides poll(), wait(), returncode, stdout, and stderr.

__init__(*proc*, *expected_retcode*, *timeout=None*)

__repr__()

Return repr(self).

poll()

Polls the underlying process for termination; returns False if still running, or True if terminated

ready()

Polls the underlying process for termination; returns False if still running, or True if terminated

wait()

Waits for the process to terminate; will raise a plumbum.commands.ProcessExecutionError in case of failure

property stdout

The process' stdout; accessing this property will wait for the process to finish

property stderr

The process' stderr; accessing this property will wait for the process to finish

property returncode

The process' returncode; accessing this property will wait for the process to finish

__weakref__

list of weak references to the object (if defined)

class plumbum.commands.modifiers.**PipeToLoggerMixin**

This mixin allows piping plumbum commands' output into a logger. The logger must implement a log(level, msg) method, as in logging.Logger

Example:

```
class MyLogger(logging.Logger, PipeToLoggerMixin):
    pass

logger = MyLogger("example.app")
```

Here we send the output of an install.sh script into our log:

```
local['./install.sh'] & logger
```

We can choose the log-level for each stream:

```
local['./install.sh'] & logger.pipe(out_level=logging.DEBUG, err_level=logging.
↪DEBUG)
```

Or use a convenience method for it:

```
local['./install.sh'] & logger.pipe_debug()
```

A prefix can be added to each line:

```
local['./install.sh'] & logger.pipe(prefix="install.sh: ")
```

If the command fails, an exception is raised as usual. This can be modified:

```
local['install.sh'] & logger.pipe_debug(retcode=None)
```

An exception is also raised if too much time (DEFAULT_LINE_TIMEOUT) passed between lines in the stream, This can also be modified:

```
local['install.sh'] & logger.pipe(line_timeout=10)
```

If we happen to use logbook:

```
class MyLogger(logbook.Logger, PipeToLoggerMixin):
    from logbook import DEBUG, INFO  # hook up with logbook's levels
```

pipe(*out_level=None*, *err_level=None*, *prefix=None*, *line_timeout=None*, ***kw*)

Pipe a command's stdout and stderr lines into this logger.

Parameters

- **out_level** – the log level for lines coming from stdout
- **err_level** – the log level for lines coming from stderr

Optionally use *prefix* for each line.

pipe_info(*prefix=None*, ***kw*)

Pipe a command's stdout and stderr lines into this logger (both at level INFO)

pipe_debug(*prefix=None*, ***kw*)

Pipe a command's stdout and stderr lines into this logger (both at level DEBUG)

__rand__(*cmd*)

Pipe a command's stdout and stderr lines into this logger. Log levels for each stream are determined by DEFAULT_STDOUT and DEFAULT_STDERR.

__weakref__

list of weak references to the object (if defined)

exception plumbum.commands.processes.**ProcessExecutionError**(*argv*, *retcode*, *stdout*, *stderr*, *message=None*, *, *host=None*)

Represents the failure of a process. When the exit code of a terminated process does not match the expected result, this exception is raised by `run_proc`. It contains the process' return code, stdout, and stderr, as well as the command line used to create the process (`argv`)

__init__(*argv*, *retcode*, *stdout*, *stderr*, *message=None*, *, *host=None*)

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

exception plumbum.commands.processes.**ProcessTimedOut**(*msg*, *argv*)

Raises by `run_proc` when a `timeout` has been specified and it has elapsed before the process terminated

__init__(*msg*, *argv*)

__weakref__

list of weak references to the object (if defined)

exception plumbum.commands.processes.**ProcessLineTimedOut**(*msg*, *argv*, *machine*)

Raises by `iter_lines` when a `line_timeout` has been specified and it has elapsed before the process yielded another line

__init__(*msg*, *argv*, *machine*)

__weakref__

list of weak references to the object (if defined)

exception plumbum.commands.processes.**CommandNotFound**(*program*, *path*)

Raised by *local.which* and `RemoteMachine.which` when a command was not found in the system's `PATH`

__init__(*program*, *path*)

__weakref__

list of weak references to the object (if defined)

plumbum.commands.processes.**run_proc**(*proc*, *retcode*, *timeout=None*)

Waits for the given process to terminate, with the expected exit code

Parameters

- **proc** – a running Popen-like object, with all the expected methods.
- **retcode** – the expected return (exit) code of the process. It defaults to 0 (the convention for success). If `None`, the return code is ignored. It may also be a tuple (or any object that supports `__contains__`) of expected return codes.
- **timeout** – the number of seconds (a `float`) to allow the process to run, before forcefully terminating it. If `None`, not timeout is imposed; otherwise the process is expected to terminate within that timeout value, or it will be killed and `ProcessTimedOut` will be raised

Returns

A tuple of (return code, stdout, stderr)

plumbum.commands.processes.**iter_lines**(*proc*, *retcode=0*, *timeout=None*, *linesize=-1*, *line_timeout=None*, *buffer_size=None*, *mode=None*, *_iter_lines=<function _iter_lines_posix>*)

Runs the given process (equivalent to run_proc()) and yields a tuples of (out, err) line pairs. If the exit code of the process does not match the expected one, `ProcessExecutionError` is raised.

Parameters

- **retcode** – The expected return code of this process (defaults to 0). In order to disable exit-code validation, pass `None`. It may also be a tuple (or any iterable) of expected exit codes.
- **timeout** – The maximal amount of time (in seconds) to allow the process to run. `None` means no timeout is imposed; otherwise, if the process hasn't terminated after that many seconds, the process will be forcefully terminated an exception will be raised
- **linesize** – Maximum number of characters to read from stdout/stderr at each iteration. `-1` (default) reads until a b'n' is encountered.
- **line_timeout** – The maximal amount of time (in seconds) to allow between consecutive lines in either stream. Raise an `ProcessLineTimedOut` if the timeout has been reached. `None` means no timeout is imposed.
- **buffer_size** – Maximum number of lines to keep in the stdout/stderr buffers, in case of a ProcessExecutionError. Default is `None`, which defaults to DEFAULT_BUFFER_SIZE (which is infinite by default). `0` will disable bufferring completely.
- **mode** – Controls what the generator yields. Defaults to DEFAULT_ITER_LINES_MODE (which is BY_POSITION by default) - BY_POSITION (default): yields `(out, err)` line tuples, where either item may be `None` - BY_TYPE: yields `(fd, line)` tuples, where `fd` is 1 (stdout) or 2 (stderr)

Returns

An iterator of (out, err) line tuples.

5.3 Package plumbum.machines

class plumbum.machines.env.**EnvPathList**(*path_factory*, *pathsep*)

__init__(*path_factory*, *pathsep*)

append(*path*)

Append object to the end of the list.

extend(*paths*)

Extend list by appending elements from the iterable.

insert(*index*, *path*)

Insert object before index.

index(*path*)

Return first index of value.

Raises ValueError if the value is not present.

__contains__(*path*)

Return key in self.

remove(*path*)

Remove first occurrence of value.

Raises ValueError if the value is not present.

class plumbum.machines.env.**BaseEnv**(*path_factory*, *pathsep*, *, *_curr*)

The base class of LocalEnv and RemoteEnv

__init__(*path_factory*, *pathsep*, *, *_curr*)

__call__(**args*, ***kwargs*)

A context manager that can be used for temporal modifications of the environment. Any time you enter the context, a copy of the old environment is stored, and then restored, when the context exits.

Parameters

- **args** – Any positional arguments for `update()`
- **kwargs** – Any keyword arguments for `update()`

__iter__()

Returns an iterator over the items `(key, value)` of current environment (like dict.items)

__hash__()

Return hash(self).

__len__()

Returns the number of elements of the current environment

__contains__(*name*)

Tests whether an environment variable exists in the current environment

__getitem__(*name*)

Returns the value of the given environment variable from current environment, raising a `KeyError` if it does not exist

keys()

Returns the keys of the current environment (like dict.keys)

items()

Returns the items of the current environment (like dict.items)

values()

Returns the values of the current environment (like dict.values)

get(*name*, **default*)

Returns the keys of the current environment (like dict.keys)

__delitem__(*name*)

Deletes an environment variable from the current environment

__setitem__(*name*, *value*)

Sets/replaces an environment variable's value in the current environment

pop(*name*, **default*)

Pops an element from the current environment (like dict.pop)

clear()

Clears the current environment (like dict.clear)

update(**args*, ***kwargs*)

Updates the current environment (like dict.update)

getdict()

Returns the environment as a real dictionary

property path

The system's PATH (as an easy-to-manipulate list)

property home

Get or set the home path

property user

Return the user name, or None if it is not set

class plumbum.machines.local.**PlumbumLocalPopen**(**args*, ***kwargs*)

iter_lines(*retcode=0*, *timeout=None*, *linesize=-1*, *line_timeout=None*, *buffer_size=None*, *mode=None*, *_iter_lines=<function _iter_lines_posix>*)

Runs the given process (equivalent to run_proc()) and yields a tuples of (out, err) line pairs. If the exit code of the process does not match the expected one, ProcessExecutionError is raised.

Parameters

- **retcode** – The expected return code of this process (defaults to 0). In order to disable exit-code validation, pass None. It may also be a tuple (or any iterable) of expected exit codes.
- **timeout** – The maximal amount of time (in seconds) to allow the process to run. None means no timeout is imposed; otherwise, if the process hasn't terminated after that many seconds, the process will be forcefully terminated an exception will be raised
- **linesize** – Maximum number of characters to read from stdout/stderr at each iteration. -1 (default) reads until a b'n' is encountered.
- **line_timeout** – The maximal amount of time (in seconds) to allow between consecutive lines in either stream. Raise an ProcessLineTimedOut if the timeout has been reached. None means no timeout is imposed.
- **buffer_size** – Maximum number of lines to keep in the stdout/stderr buffers, in case of a ProcessExecutionError. Default is None, which defaults to DEFAULT_BUFFER_SIZE (which is infinite by default). 0 will disable bufferring completely.
- **mode** – Controls what the generator yields. Defaults to DEFAULT_ITER_LINES_MODE (which is BY_POSITION by default) - BY_POSITION (default): yields (out, err) line tuples, where either item may be None - BY_TYPE: yields (fd, line) tuples, where fd is 1 (stdout) or 2 (stderr)

Returns

An iterator of (out, err) line tuples.

__init__(**args*, ***kwargs*)

class plumbum.machines.local.**LocalEnv**

The local machine's environment; exposes a dict-like interface

__init__()

expand(*expr*)

Expands any environment variables and home shortcuts found in `expr` (like `os.path.expanduser` combined with `os.path.expandvars`)

Parameters
expr – An expression containing environment variables (as `$FOO`) or home shortcuts (as `~/.bashrc`)

Returns
The expanded string

expanduser(*expr*)

Expand home shortcuts (e.g., `~/foo/bar` or `~john/foo/bar`)

Parameters
expr – An expression containing home shortcuts

Returns
The expanded string

class plumbum.machines.local.**LocalCommand**(*executable*, *encoding='auto'*)

__init__(*executable*, *encoding='auto'*)

popen(*args=()*, *cwd=None*, *env=None*, ***kwargs*)

Spawns the given command, returning a `Popen`-like object.

Note: When processes run in the **background** (either via `popen` or `& BG`), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the `Popen` constructor

Returns
A `Popen`-like object

class plumbum.machines.local.**LocalMachine**

The *local machine* (a singleton object). It serves as an entry point to everything related to the local machine, such as working directory and environment manipulation, command creation, etc.

Attributes:

- `cwd` - the local working directory
- `env` - the local environment
- `custom_encoding` - the local machine's default encoding (`sys.getfilesystemencoding()`)

__init__()

classmethod which(*progname*)

Looks up a program in the `PATH`. If the program is not found, raises `CommandNotFound`

Parameters
progname – The program's name. Note that if underscores (_) are present in the name, and the exact name is not found, they will be replaced in turn by hyphens (-) then periods (.), and the name will be looked up again for each alternative

Returns
A LocalPath

path(**parts*)

A factory for *LocalPaths*. Usage: `p = local.path("/usr", "lib", "python2.7")`

__contains__(*cmd*)

Tests for the existence of the command, e.g., `"ls" in plumbum.local`. `cmd` can be anything acceptable by `__getitem__`.

__getitem__(*cmd*)

Returns a *Command* object representing the given program. `cmd` can be a string or a *LocalPath*; if it is a path, a command representing this path will be returned; otherwise, the program name will be looked up in the system's `PATH` (using `which`). Usage:

```
ls = local["ls"]
```

daemonic_popen(*command*, *cwd='/'*, *stdout=None*, *stderr=None*, *append=True*)

On POSIX systems:

Run `command` as a UNIX daemon: fork a child process to setpid, redirect std handles to /dev/null, umask, close all fds, chdir to `cwd`, then fork and exec `command`. Returns a `Popen` process that can be used to poll/wait for the executed command (but keep in mind that you cannot access std handles)

On Windows:

Run `command` as a "Windows daemon": detach from controlling console and create a new process group. This means that the command will not receive console events and would survive its parent's termination. Returns a `Popen` object.

Note: this does not run `command` as a system service, only detaches it from its parent.

New in version 1.3.

list_processes()

Returns information about all running processes (on POSIX systems: using `ps`)

New in version 1.3.

pgrep(*pattern*)

Process grep: return information about all processes whose command-line args match the given regex pattern

session(*new_session=False*)

Creates a new `ShellSession` object; this invokes `/bin/sh` and executes commands on it over stdin/stdout/stderr

tempdir()

A context manager that creates a temporary directory, which is removed when the context exits

as_user(*username=None*)

Run nested commands as the given user. For example:

```
head = local["head"]
head("-n1", "/dev/sda1")    # this will fail...
with local.as_user():
    head("-n1", "/dev/sda1")
```

Parameters
username – The user to run commands as. If not given, root (or Administrator) is assumed

as_root()

A shorthand for *as_user("root")*

python = LocalCommand(/home/docs/checkouts/readthedocs.org/user_builds/plumbum/envs/stable/bin/python)

A command that represents the current python interpreter (`sys.executable`)

plumbum.machines.local.**local = <plumbum.machines.local.LocalMachine object>**

The *local machine* (a singleton object). It serves as an entry point to everything related to the local machine, such as working directory and environment manipulation, command creation, etc.

Attributes:

- `cwd` - the local working directory
- `env` - the local environment
- `custom_encoding` - the local machine's default encoding (`sys.getfilesystemencoding()`)

exception plumbum.machines.session.**ShellSessionError**

Raises when something goes wrong when calling `ShellSession.popen`

__weakref__

list of weak references to the object (if defined)

exception plumbum.machines.session.**SSHCommsError**(*argv*, *retcode*, *stdout*, *stderr*, *message=None*, *, *host=None*)

Raises when the communication channel can't be created on the remote host or it times out.

exception plumbum.machines.session.**SSHCommsChannel2Error**(*argv*, *retcode*, *stdout*, *stderr*, *message=None*, *, *host=None*)

Raises when channel 2 (stderr) is not available

exception plumbum.machines.session.**IncorrectLogin**(*argv*, *retcode*, *stdout*, *stderr*, *message=None*, *, *host=None*)

Raises when incorrect login credentials are provided

exception plumbum.machines.session.**HostPublicKeyUnknown**(*argv*, *retcode*, *stdout*, *stderr*, *message=None*, *, *host=None*)

Raises when the host public key isn't known

class plumbum.machines.session.**MarkedPipe**(*pipe*, *marker*)

A pipe-like object from which you can read lines; the pipe will return report EOF (the empty string) when a special marker is detected

__init__(*pipe*, *marker*)

close()

'Closes' the marked pipe; following calls to `readline` will return ""

readline()

Reads the next line from the pipe; returns "" when the special marker is reached. Raises EOFError if the underlying pipe has closed

class plumbum.machines.session.**SessionPopen**(*proc*, *argv*, *isatty*, *stdin*, *stdout*, *stderr*, *encoding*, *, *host*)

A shell-session-based Popen-like object (has the following attributes: stdin, stdout, stderr, returncode)

__init__(*proc*, *argv*, *isatty*, *stdin*, *stdout*, *stderr*, *encoding*, *, *host*)

poll()

Returns the process' exit code or None if it's still running

wait()

Waits for the process to terminate and returns its exit code

communicate(*input=None*)

Consumes the process' stdout and stderr until the it terminates.

Parameters
input – An optional bytes/buffer object to send to the process over stdin

Returns
A tuple of (stdout, stderr)

class plumbum.machines.session.**ShellSession**(*proc*, *encoding='auto'*, *isatty=False*, *connect_timeout=5*, *, *host=None*)

An abstraction layer over *shell sessions*. A shell session is the execution of an interactive shell (/bin/sh or something compatible), over which you may run commands (sent over stdin). The output of is then read from stdout and stderr. Shell sessions are less "robust" than executing a process on its own, and they are susceptible to all sorts of malformatted-strings attacks, and there is little benefit from using them locally. However, they can greatly speed up remote connections, and are required for the implementation of SshMachine, as they allow us to send multiple commands over a single SSH connection (setting up separate SSH connections incurs a high overhead). Try to avoid using shell sessions, unless you know what you're doing.

Instances of this class may be used as *context-managers*.

Parameters

- **proc** – The underlying shell process (with open stdin, stdout and stderr)
- **encoding** – The encoding to use for the shell session. If "auto", the underlying process' encoding is used.
- **isatty** – If true, assume the shell has a TTY and that stdout and stderr are unified
- **connect_timeout** – The timeout to connect to the shell, after which, if no prompt is seen, the shell process is killed

__init__(*proc*, *encoding='auto'*, *isatty=False*, *connect_timeout=5*, *, *host=None*)

alive()

Returns True if the underlying shell process is alive, False otherwise

close()

Closes (terminates) the shell session

__weakref__

list of weak references to the object (if defined)

popen(*cmd*)

Runs the given command in the shell, adding some decoration around it. Only a single command can be executed at any given time.

Parameters
cmd – The command (string or `Command` object) to run

Returns
A `SessionPopen` instance

run(*cmd*, *retcode=0*)

Runs the given command

Parameters

- **cmd** – The command (string or `Command` object) to run
- **retcode** – The expected return code (0 by default). Set to `None` in order to ignore erroneous return codes

Returns
A tuple of (return code, stdout, stderr)

5.3.1 Remote Machines

class plumbum.machines.remote.**RemoteEnv**(*remote*)

The remote machine's environment; exposes a dict-like interface

__init__(*remote*)

__delitem__(*name*)

Deletes an environment variable from the current environment

__setitem__(*name*, *value*)

Sets/replaces an environment variable's value in the current environment

pop(*name*, **default*)

Pops an element from the current environment (like dict.pop)

update(**args*, ***kwargs*)

Updates the current environment (like dict.update)

expand(*expr*)

Expands any environment variables and home shortcuts found in `expr` (like `os.path.expanduser` combined with `os.path.expandvars`)

Parameters
expr – An expression containing environment variables (as `$FOO`) or home shortcuts (as `~/.bashrc`)

Returns
The expanded string

expanduser(*expr*)

Expand home shortcuts (e.g., `~/foo/bar` or `~john/foo/bar`)

Parameters
expr – An expression containing home shortcuts

Returns
The expanded string

`getdelta()`
Returns the difference between the this environment and the original environment of the remote machine

`class plumbum.machines.remote.RemoteCommand(remote, executable, encoding='auto')`

`__init__(remote, executable, encoding='auto')`

`__repr__()`
Return repr(self).

`popen(args=(), **kwargs)`
Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via popen or & BG), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns
A Popen-like object

`nohup(cwd='.', stdout='nohup.out', stderr=None, append=True)`
Runs a command detached.

`exception plumbum.machines.remote.ClosedRemoteMachine`

`__weakref__`
list of weak references to the object (if defined)

`class plumbum.machines.remote.BaseRemoteMachine(encoding='utf8', connect_timeout=10, new_session=False)`

Represents a *remote machine*; serves as an entry point to everything related to that remote machine, such as working directory and environment manipulation, command creation, etc.

Attributes:

- cwd - the remote working directory
- env - the remote environment
- custom_encoding - the remote machine's default encoding (assumed to be UTF8)
- connect_timeout - the connection timeout

There also is a _cwd attribute that exists if the cwd is not current (del if cwd is changed).

`class RemoteCommand(remote, executable, encoding='auto')`

`__init__(remote, executable, encoding='auto')`

__repr__()

Return repr(self).

nohup(*cwd='.'*, *stdout='nohup.out'*, *stderr=None*, *append=True*)

Runs a command detached.

popen(*args=()*, ***kwargs*)

Spawns the given command, returning a Popen-like object.

Note: When processes run in the **background** (either via `popen` or `& BG`), their stdout/stderr pipes might fill up, causing them to hang. If you know a process produces output, be sure to consume it every once in a while, using a monitoring thread/reactor in the background. For more info, see #48

Parameters

- **args** – Any arguments to be passed to the process (a tuple)
- **kwargs** – Any keyword-arguments to be passed to the Popen constructor

Returns

A Popen-like object

__init__(*encoding='utf8'*, *connect_timeout=10*, *new_session=False*)

__repr__()

Return repr(self).

close()

closes the connection to the remote machine; all paths and programs will become defunct

path(**parts*)

A factory for *RemotePaths*. Usage: `p = rem.path("/usr", "lib", "python2.7")`

which(*progname*)

Looks up a program in the PATH. If the program is not found, raises CommandNotFound

Parameters

progname – The program's name. Note that if underscores (_) are present in the name, and the exact name is not found, they will be replaced in turn by hyphens (-) then periods (.), and the name will be looked up again for each alternative

Returns

A RemotePath

__getitem__(*cmd*)

Returns a *Command* object representing the given program. cmd can be a string or a *RemotePath*; if it is a path, a command representing this path will be returned; otherwise, the program name will be looked up in the system's PATH (using which). Usage:

```
r_ls = rem["ls"]
```

property python

A command that represents the default remote python interpreter

session(*isatty=False*, ***, *new_session=False*)

Creates a new ShellSession object; this invokes the user's shell on the remote machine and executes commands on it over stdin/stdout/stderr

download(*src*, *dst*)

Downloads a remote file/directory (`src`) to a local destination (`dst`). `src` must be a string or a *RemotePath* pointing to this remote machine, and `dst` must be a string or a `LocalPath`

upload(*src*, *dst*)

Uploads a local file/directory (`src`) to a remote destination (`dst`). `src` must be a string or a `LocalPath`, and `dst` must be a string or a *RemotePath* pointing to this remote machine

popen(*args*, ***kwargs*)

Spawns the given command on the remote machine, returning a `Popen`-like object; do not use this method directly, unless you need "low-level" control on the remote process

list_processes()

Returns information about all running processes (on POSIX systems: using `ps`)

New in version 1.3.

pgrep(*pattern*)

Process grep: return information about all processes whose command-line args match the given regex pattern

tempdir()

A context manager that creates a remote temporary directory, which is removed when the context exits

class plumbum.machines.ssh_machine.**SshTunnel**(*session*, *lport*, *dport*, *reverse*)

An object representing an SSH tunnel (created by `SshMachine.tunnel`)

__init__(*session*, *lport*, *dport*, *reverse*)

__repr__()

Return repr(self).

close()

Closes(terminates) the tunnel

property lport

Tunneled port or socket on the local machine.

property dport

Tunneled port or socket on the remote machine.

property reverse

Represents if the tunnel is a reverse tunnel.

class plumbum.machines.ssh_machine.**SshMachine**(*host*, *user=None*, *port=None*, *keyfile=None*, *ssh_command=None*, *scp_command=None*, *ssh_opts=()*, *scp_opts=()*, *password=None*, *encoding='utf8'*, *connect_timeout=10*, *new_session=False*)

An implementation of *remote machine* over SSH. Invoking a remote command translates to invoking it over SSH

```
with SshMachine("yourhostname") as rem:
    r_ls = rem["ls"]
    # r_ls is the remote `ls`
    # executing r_ls() translates to `ssh yourhostname ls`
```

Parameters

- **host** – the host name to connect to (SSH server)
- **user** – the user to connect as (if None, the default will be used)
- **port** – the server's port (if None, the default will be used)
- **keyfile** – the path to the identity file (if None, the default will be used)
- **ssh_command** – the ssh command to use; this has to be a Command object; if None, the default ssh client will be used.
- **scp_command** – the scp command to use; this has to be a Command object; if None, the default scp program will be used.
- **ssh_opts** – any additional options for ssh (a list of strings)
- **scp_opts** – any additional options for scp (a list of strings)
- **password** – the password to use; requires sshpass be installed. Cannot be used in conjunction with ssh_command or scp_command (will be ignored). NOTE: THIS IS A SECURITY RISK!
- **encoding** – the remote machine's encoding (defaults to UTF8)
- **connect_timeout** – specify a connection timeout (the time until shell prompt is seen). The default is 10 seconds. Set to None to disable
- **new_session** – whether or not to start the background session as a new session leader (setsid). This will prevent it from being killed on Ctrl+C (SIGINT)

__init__(*host*, *user=None*, *port=None*, *keyfile=None*, *ssh_command=None*, *scp_command=None*, *ssh_opts=()*, *scp_opts=()*, *password=None*, *encoding='utf8'*, *connect_timeout=10*, *new_session=False*)

__str__()

Return str(self).

popen(*args*, *ssh_opts=()*, *env=None*, *cwd=None*, ***kwargs*)

Spawns the given command on the remote machine, returning a Popen-like object; do not use this method directly, unless you need "low-level" control on the remote process

nohup(*command*)

Runs the given command using nohup and redirects std handles, allowing the command to run "detached" from its controlling TTY or parent. Does not return anything. Depreciated (use command.nohup or daemonic_popen).

daemonic_popen(*command*, *cwd='.'*, *stdout=None*, *stderr=None*, *append=True*)

Runs the given command using nohup and redirects std handles, allowing the command to run "detached" from its controlling TTY or parent. Does not return anything.

New in version 1.6.0.

session(*isatty=False*, *new_session=False*)

Creates a new ShellSession object; this invokes the user's shell on the remote machine and executes commands on it over stdin/stdout/stderr

tunnel(*lport*, *dport*, *lhost='localhost'*, *dhost='localhost'*, *connect_timeout=5*, *reverse=False*)

Creates an SSH tunnel from the TCP port (lport) of the local machine (lhost, defaults to "localhost", but it can be any IP you can bind()) to the remote TCP port (dport) of the destination machine (dhost, defaults to "localhost", which means *this remote machine*). This function also supports Unix sockets, in

which case the local socket should be passed in as `lport` and the local bind address should be `None`. The same can be done for a remote socket, by following the same pattern with `dport` and `dhost`. The returned `SshTunnel` object can be used as a *context-manager*.

The more conventional use case is the following:

```
+---------+          +---------+
| Your    |          | Remote  |
| Machine |          | Machine |
+----o----+          +----  ----+
     |                    ^
     |                    |
   lport                dport
     |                    |
     \_____SSH TUNNEL____/
            (secure)
```

Here, you wish to communicate safely between port `lport` of your machine and port `dport` of the remote machine. Communication is tunneled over SSH, so the connection is authenticated and encrypted.

The more general case is shown below (where `dport != "localhost"`):

```
+---------+          +-------------+      +-------------+
| Your    |          | Remote      |      | Destination |
| Machine |          | Machine     |      | Machine     |
+----o----+          +----  ----o---+      +----  --------+
     |                    ^    |               ^
     |                    |    |               |
lhost:lport               |    |          dhost:dport
     |                    |    |               |
     \_____SSH TUNNEL_____/    \_____SOCKET____/
            (secure)              (not secure)
```

Usage:

```
rem = SshMachine("megazord")

with rem.tunnel(1234, "/var/lib/mysql/mysql.sock", dhost=None):
    sock = socket.socket()
    sock.connect(("localhost", 1234))
    # sock is now tunneled to the MySQL socket on megazord
```

download(*src*, *dst*)

Downloads a remote file/directory (`src`) to a local destination (`dst`). `src` must be a string or a *RemotePath* pointing to this remote machine, and `dst` must be a string or a `LocalPath`

upload(*src*, *dst*)

Uploads a local file/directory (`src`) to a remote destination (`dst`). `src` must be a string or a `LocalPath`, and `dst` must be a string or a *RemotePath* pointing to this remote machine

class plumbum.machines.ssh_machine.**PuttyMachine**(*host*, *user=None*, *port=None*, *keyfile=None*, *ssh_command=None*, *scp_command=None*, *ssh_opts=()*, *scp_opts=()*, *encoding='utf8'*, *connect_timeout=10*, *new_session=False*)

PuTTY-flavored SSH connection. The programs `plink` and `pscp` are expected to be in the path (or you may provide your own `ssh_command` and `scp_command`)

Arguments are the same as for `plumbum.machines.remote.SshMachine`

__init__(*host*, *user=None*, *port=None*, *keyfile=None*, *ssh_command=None*, *scp_command=None*, *ssh_opts=()*, *scp_opts=()*, *encoding='utf8'*, *connect_timeout=10*, *new_session=False*)

__str__()

Return str(self).

session(*isatty=False*, *new_session=False*)

Creates a new `ShellSession` object; this invokes the user's shell on the remote machine and executes commands on it over stdin/stdout/stderr

class plumbum.machines.paramiko_machine.**ParamikoPopen**(*argv*, *stdin*, *stdout*, *stderr*, *encoding*, *stdin_file=None*, *stdout_file=None*, *stderr_file=None*)

__init__(*argv*, *stdin*, *stdout*, *stderr*, *encoding*, *stdin_file=None*, *stdout_file=None*, *stderr_file=None*)

class plumbum.machines.paramiko_machine.**ParamikoMachine**(*host*, *user=None*, *port=None*, *password=None*, *keyfile=None*, *load_system_host_keys=True*, *missing_host_policy=None*, *encoding='utf8'*, *look_for_keys=None*, *connect_timeout=None*, *keep_alive=0*, *gss_auth=False*, *gss_kex=None*, *gss_deleg_creds=None*, *gss_host=None*, *get_pty=False*, *load_system_ssh_config=False*)

An implementation of *remote machine* over Paramiko (a Python implementation of openSSH2 client/server). Invoking a remote command translates to invoking it over SSH

```
with ParamikoMachine("yourhostname") as rem:
    r_ls = rem["ls"]
    # r_ls is the remote `ls`
    # executing r_ls() is equivalent to `ssh yourhostname ls`, only without
    # spawning a new ssh client
```

Parameters

- **host** – the host name to connect to (SSH server)
- **user** – the user to connect as (if `None`, the default will be used)
- **port** – the server's port (if `None`, the default will be used)
- **password** – the user's password (if a password-based authentication is to be performed) (if `None`, key-based authentication will be used)
- **keyfile** – the path to the identity file (if `None`, the default will be used)
- **load_system_host_keys** – whether or not to load the system's host keys (from `/etc/ssh` and `~/.ssh`). The default is `True`, which means Paramiko behaves much like the `ssh` command-line client
- **missing_host_policy** – the value passed to the underlying `set_missing_host_key_policy` of the client. The default is `None`, which means `set_missing_host_key_policy` is not invoked and paramiko's default behavior (reject) is employed

- **encoding** – the remote machine's encoding (defaults to UTF8)
- **look_for_keys** – set to False to disable searching for discoverable private key files in ~/.ssh
- **connect_timeout** – timeout for TCP connection

Note: If Paramiko 1.15 or above is installed, can use GSS_API authentication

---

Parameters

- **gss_auth** (*bool*) – True if you want to use GSS-API authentication
- **gss_kex** (*bool*) – Perform GSS-API Key Exchange and user authentication
- **gss_deleg_creds** (*bool*) – Delegate GSS-API client credentials or not
- **gss_host** (*str*) – The targets name in the kerberos database. default: hostname
- **get_pty** (*bool*) – Execute remote commands with allocated pseudo-tty. default: False
- **load_system_ssh_config** (*bool*) – read system SSH config for ProxyCommand configuration. default: False

class RemoteCommand(*remote*, *executable*, *encoding='auto'*)

__or__(**_*)

Creates a pipe with the other command

__gt__(**_*)

Redirects the process' stdout to the given file

__rshift__(**_*)

Redirects the process' stdout to the given file (appending)

__ge__(**_*)

Redirects the process' stderr to the given file

__lt__(**_*)

Redirects the given file into the process' stdin

__lshift__(**_*)

Redirects the given data into the process' stdin

__init__(*host*, *user=None*, *port=None*, *password=None*, *keyfile=None*, *load_system_host_keys=True*, *missing_host_policy=None*, *encoding='utf8'*, *look_for_keys=None*, *connect_timeout=None*, *keep_alive=0*, *gss_auth=False*, *gss_kex=None*, *gss_deleg_creds=None*, *gss_host=None*, *get_pty=False*, *load_system_ssh_config=False*)

__str__()

Return str(self).

close()

closes the connection to the remote machine; all paths and programs will become defunct

property sftp

Returns an SFTP client on top of the current SSH connection; it can be used to manipulate files directly, much like an interactive FTP/SFTP session

session(*isatty=False*, *term='vt100'*, *width=80*, *height=24*, ***, *new_session=False*)

Creates a new `ShellSession` object; this invokes the user's shell on the remote machine and executes commands on it over stdin/stdout/stderr

popen(*args*, *stdin=None*, *stdout=None*, *stderr=None*, *new_session=False*, *env=None*, *cwd=None*)

Spawns the given command on the remote machine, returning a `Popen`-like object; do not use this method directly, unless you need "low-level" control on the remote process

download(*src*, *dst*)

Downloads a remote file/directory (`src`) to a local destination (`dst`). `src` must be a string or a *RemotePath* pointing to this remote machine, and `dst` must be a string or a `LocalPath`

upload(*src*, *dst*)

Uploads a local file/directory (`src`) to a remote destination (`dst`). `src` must be a string or a `LocalPath`, and `dst` must be a string or a *RemotePath* pointing to this remote machine

connect_sock(*dport*, *dhost='localhost'*, *ipv6=False*)

Returns a Paramiko `Channel`, connected to dhost:dport on the remote machine. The `Channel` behaves like a regular socket; you can `send` and `recv` on it and the data will pass encrypted over SSH. Usage:

```
mach = ParamikoMachine("myhost")
sock = mach.connect_sock(12345)
data = sock.recv(100)
sock.send("foobar")
sock.close()
```

5.4 Package plumbum.path

class plumbum.path.base.**FSUser**(*val*, *name=None*)

A special object that represents a file-system user. It derives from `int`, so it behaves just like a number (`uid`/`gid`), but also have a `.name` attribute that holds the string-name of the user, if given (otherwise `None`)

static __new__(*cls*, *val*, *name=None*)

class plumbum.path.base.**Path**

An abstraction over file system paths. This class is abstract, and the two implementations are `LocalPath` and *RemotePath*.

__repr__()

Return repr(self).

__truediv__(*other: Any*) → _PathImpl

Joins two paths

__getitem__(*key*)

Return self[key].

__floordiv__(*expr*)

Returns a (possibly empty) list of paths that matched the glob-pattern under this path

__iter__()

Iterate over the files in this directory

__eq__(*other: object*) → bool

Return self==value.

__ne__(*other*)

Return self!=value.

__gt__(*other*)

Return self>value.

__ge__(*other*)

Return self>=value.

__lt__(*other*)

Return self<value.

__le__(*other*)

Return self<=value.

__hash__()

Return hash(self).

__fspath__()

Added for Python 3.6 support

__contains__(*item*)

Paths should support checking to see if an file or folder is in them.

up(*count=1*)

Go up in `count` directories (the default is 1)

walk(*filter=<function Path.<lambda>>*, *dir_filter=<function Path.<lambda>>*)

traverse all (recursive) sub-elements under this directory, that match the given filter. By default, the filter accepts everything; you can provide a custom filter function that takes a path as an argument and returns a boolean

Parameters

- **filter** – the filter (predicate function) for matching results. Only paths matching this predicate are returned. Defaults to everything.
- **dir_filter** – the filter (predicate function) for matching directories. Only directories matching this predicate are recursed into. Defaults to everything.

abstract property name: str

The basename component of this path

property basename

Included for compatibility with older Plumbum code

abstract property stem: str

The name without an extension, or the last component of the path

abstract property dirname: _PathImpl

The dirname component of this path

abstract property root: str

The root of the file tree (/ on Unix)

abstract property drive: str

The drive letter (on Windows)

abstract property suffix: str

The suffix of this file

abstract property suffixes: List[str]

This is a list of all suffixes

abstract property uid: *FSUser*

The user that owns this path. The returned value is a `FSUser` object which behaves like an `int` (as expected from `uid`), but it also has a `.name` attribute that holds the string-name of the user

abstract property gid: *FSUser*

The group that owns this path. The returned value is a `FSUser` object which behaves like an `int` (as expected from `gid`), but it also has a `.name` attribute that holds the string-name of the group

abstract as_uri(*scheme: str | None = None*) → str

Returns a universal resource identifier. Use `scheme` to force a scheme.

abstract join(**parts: Any*) → _PathImpl

Joins this path with any number of paths

abstract list() → List[_PathImpl]

Returns the files in this directory

abstract iterdir() → Iterable[_PathImpl]

Returns an iterator over the directory. Might be slightly faster on Python 3.5 than .list()

abstract is_dir() → bool

Returns `True` if this path is a directory, `False` otherwise

isdir()

Included for compatibility with older Plumbum code

abstract is_file() → bool

Returns `True` if this path is a regular file, `False` otherwise

isfile() → bool

Included for compatibility with older Plumbum code

islink()

Included for compatibility with older Plumbum code

abstract is_symlink() → bool

Returns `True` if this path is a symbolic link, `False` otherwise

abstract exists() → bool

Returns `True` if this path exists, `False` otherwise

abstract stat() → stat_result

Returns the os.stats for a file

abstract with_name(*name: Any*) → _PathImpl

Returns a path with the name replaced

abstract with_suffix(*suffix: str*, *depth: int | None = 1*) → _PathImpl

Returns a path with the suffix replaced. Up to last `depth` suffixes will be replaced. None will replace all suffixes. If there are less than `depth` suffixes, this will replace all suffixes. `.tar.gz` is an example where `depth=2` or `depth=None` is useful

preferred_suffix(*suffix*)

Adds a suffix if one does not currently exist (otherwise, no change). Useful for loading files with a default suffix

abstract glob(*pattern: str | Iterable[str]*) → List[_PathImpl]

Returns a (possibly empty) list of paths that matched the glob-pattern under this path

abstract delete()

Deletes this path (recursively, if a directory)

abstract move(*dst*)

Moves this path to a different location

rename(*newname*)

Renames this path to the `new name` (only the basename is changed)

abstract copy(*dst*, *override=None*)

Copies this path (recursively, if a directory) to the destination path "dst". Raises TypeError if dst exists and override is False. Will overwrite if override is True. Will silently fail to copy if override is None (the default).

abstract mkdir(*mode=511*, *parents=True*, *exist_ok=True*)

Creates a directory at this path.

Parameters

- **mode** – **Currently only implemented for local paths!** Numeric mode to use for directory creation, which may be ignored on some systems. The current implementation reproduces the behavior of `os.mkdir` (i.e., the current umask is first masked out), but this may change for remote paths. As with `os.mkdir`, it is recommended to call *chmod()* explicitly if you need to be sure.
- **parents** – If this is true (the default), the directory's parents will also be created if necessary.
- **exist_ok** – If this is true (the default), no exception will be raised if the directory already exists (otherwise `OSError`).

Note that the defaults for `parents` and `exist_ok` are the opposite of what they are in Python's own `pathlib` - this is to maintain backwards-compatibility with Plumbum's behaviour from before they were implemented.

abstract open(*mode: str = 'r'*, ***, *encoding: str | None = None*) → IOBase

opens this path as a file

abstract read(*encoding: str | None = None*) → str

returns the contents of this file as a `str`. By default the data is read as text, but you can specify the encoding, e.g., `'latin1'` or `'utf8'`

abstract write(*data: AnyStr*, *encoding: str | None = None*) → None

writes the given data to this file. By default the data is written as-is (either text or binary), but you can specify the encoding, e.g., `'latin1'` or `'utf8'`

abstract touch()

Update the access time. Creates an empty file if none exists.

abstract chown(*owner=None*, *group=None*, *recursive=None*)

Change ownership of this path.

Parameters

- **owner** – The owner to set (either `uid` or `username`), optional
- **group** – The group to set (either `gid` or `groupname`), optional
- **recursive** – whether to change ownership of all contained files and subdirectories. Only meaningful when `self` is a directory. If `None`, the value will default to `True` if `self` is a directory, `False` otherwise.

abstract chmod(*mode*)

Change the mode of path to the numeric mode.

Parameters
mode – file mode as for os.chmod

abstract access(*mode: int | str = 0*) → bool

Test file existence or permission bits

Parameters
mode – a bitwise-or of access bits, or a string-representation thereof: `'f'`, `'x'`, `'r'`, `'w'` for `os.F_OK`, `os.X_OK`, `os.R_OK`, `os.W_OK`

abstract link(*dst*)

Creates a hard link from `self` to `dst`

Parameters
dst – the destination path

abstract symlink(*dst*)

Creates a symbolic link from `self` to `dst`

Parameters
dst – the destination path

abstract unlink()

Deletes a symbolic link

split(**_args*, ***_kargs*)

Splits the path on directory separators, yielding a list of directories, e.g, `"/var/log/messages"` will yield `['var', 'log', 'messages']`.

property parts

Splits the directory into parts, including the base directory, returns a tuple

relative_to(*source*)

Computes the "relative path" require to get from `source` to `self`. They satisfy the invariant `source_path + (target_path - source_path) == target_path`. For example:

```
/var/log/messages - /var/log/messages = []
/var/log/messages - /var              = [log, messages]
/var/log/messages - /                 = [var, log, messages]
/var/log/messages - /var/tmp          = [.., log, messages]
```

(continues on next page)

(continued from previous page)

```
/var/log/messages - /opt              = [.., var, log, messages]
/var/log/messages - /opt/lib          = [.., .., var, log, messages]
```

__sub__(*other*)

Same as `self.relative_to(other)`

resolve(*strict=False*)

Added to allow pathlib like syntax. Does nothing since Plumbum paths are always absolute. Does not (currently) resolve symlinks.

property parents

Pathlib like sequence of ancestors

property parent

Pathlib like parent of the path.

__weakref__

list of weak references to the object (if defined)

class plumbum.path.base.**RelativePath**(*parts*)

Relative paths are the "delta" required to get from one path to another. Note that relative path do not point at anything, and thus are not paths. Therefore they are system agnostic (but closed under addition) Paths are always absolute and point at "something", whether existent or not.

Relative paths are created by subtracting paths (`Path.relative_to`)

__init__(*parts*)

__str__()

Return str(self).

__repr__()

Return repr(self).

__eq__(*other*)

Return self==value.

__ne__(*other*)

Return self!=value.

__gt__(*other*)

Return self>value.

__ge__(*other*)

Return self>=value.

__lt__(*other*)

Return self<value.

__le__(*other*)

Return self<=value.

__hash__()

Return hash(self).

__weakref__

list of weak references to the object (if defined)

class plumbum.path.local.**LocalPath**(**parts*)

The class implementing local-machine paths

static __new__(*cls*, **parts*)

property name

The basename component of this path

property dirname

The dirname component of this path

property suffix

The suffix of this file

property suffixes

This is a list of all suffixes

property uid

The user that owns this path. The returned value is a `FSUser` object which behaves like an `int` (as expected from `uid`), but it also has a `.name` attribute that holds the string-name of the user

property gid

The group that owns this path. The returned value is a `FSUser` object which behaves like an `int` (as expected from `gid`), but it also has a `.name` attribute that holds the string-name of the group

join(**others*)

Joins this path with any number of paths

list()

Returns the files in this directory

iterdir()

Returns an iterator over the directory. Might be slightly faster on Python 3.5 than .list()

is_dir()

Returns `True` if this path is a directory, `False` otherwise

is_file()

Returns `True` if this path is a regular file, `False` otherwise

is_symlink()

Returns `True` if this path is a symbolic link, `False` otherwise

exists()

Returns `True` if this path exists, `False` otherwise

stat()

Returns the os.stats for a file

with_name(*name*)

Returns a path with the name replaced

property stem

The name without an extension, or the last component of the path

with_suffix(*suffix*, *depth=1*)

Returns a path with the suffix replaced. Up to last `depth` suffixes will be replaced. None will replace all suffixes. If there are less than `depth` suffixes, this will replace all suffixes. `.tar.gz` is an example where `depth=2` or `depth=None` is useful

glob(*pattern*)

Returns a (possibly empty) list of paths that matched the glob-pattern under this path

delete()

Deletes this path (recursively, if a directory)

move(*dst*)

Moves this path to a different location

copy(*dst*, *override=None*)

Copies this path (recursively, if a directory) to the destination path "dst". Raises TypeError if dst exists and override is False. Will overwrite if override is True. Will silently fail to copy if override is None (the default).

mkdir(*mode=511*, *parents=True*, *exist_ok=True*)

Creates a directory at this path.

Parameters

- **mode** – **Currently only implemented for local paths!** Numeric mode to use for directory creation, which may be ignored on some systems. The current implementation reproduces the behavior of `os.mkdir` (i.e., the current umask is first masked out), but this may change for remote paths. As with `os.mkdir`, it is recommended to call *chmod()* explicitly if you need to be sure.
- **parents** – If this is true (the default), the directory's parents will also be created if necessary.
- **exist_ok** – If this is true (the default), no exception will be raised if the directory already exists (otherwise `OSError`).

Note that the defaults for `parents` and `exist_ok` are the opposite of what they are in Python's own `pathlib` - this is to maintain backwards-compatibility with Plumbum's behaviour from before they were implemented.

open(*mode='r'*, *encoding=None*)

opens this path as a file

read(*encoding=None*, *mode='r'*)

returns the contents of this file as a `str`. By default the data is read as text, but you can specify the encoding, e.g., `'latin1'` or `'utf8'`

write(*data*, *encoding=None*, *mode=None*)

writes the given data to this file. By default the data is written as-is (either text or binary), but you can specify the encoding, e.g., `'latin1'` or `'utf8'`

touch()

Update the access time. Creates an empty file if none exists.

chown(*owner=None*, *group=None*, *recursive=None*)

Change ownership of this path.

Parameters

- **owner** – The owner to set (either `uid` or `username`), optional
- **group** – The group to set (either `gid` or `groupname`), optional
- **recursive** – whether to change ownership of all contained files and subdirectories. Only meaningful when `self` is a directory. If `None`, the value will default to `True` if `self` is a directory, `False` otherwise.

chmod(*mode*)

Change the mode of path to the numeric mode.

Parameters
mode – file mode as for os.chmod

access(*mode=0*)

Test file existence or permission bits

Parameters
mode – a bitwise-or of access bits, or a string-representation thereof: `'f'`, `'x'`, `'r'`, `'w'` for `os.F_OK`, `os.X_OK`, `os.R_OK`, `os.W_OK`

link(*dst*)

Creates a hard link from `self` to `dst`

Parameters
dst – the destination path

symlink(*dst*)

Creates a symbolic link from `self` to `dst`

Parameters
dst – the destination path

unlink()

Deletes a symbolic link

as_uri(*scheme='file'*)

Returns a universal resource identifier. Use `scheme` to force a scheme.

property drive

The drive letter (on Windows)

property root

The root of the file tree (`/` on Unix)

class plumbum.path.local.**LocalWorkdir**

Working directory manipulator

__hash__()

Return hash(self).

static __new__(*cls*)

chdir(*newdir*)

Changes the current working directory to the given one

Parameters
newdir – The destination director (a string or a `LocalPath`)

getpath()

Returns the current working directory as a `LocalPath` object

__call__(*newdir*)

A context manager used to `chdir` into a directory and then `chdir` back to the previous location; much like pushd/popd.

Parameters
newdir – The destination directory (a string or a `LocalPath`)

class plumbum.path.remote.**StatRes**(*tup*)

POSIX-like stat result

__init__(*tup*)

__weakref__

list of weak references to the object (if defined)

class plumbum.path.remote.**RemotePath**(*remote*, **parts*)

The class implementing remote-machine paths

static __new__(*cls*, *remote*, **parts*)

property name

The basename component of this path

property dirname

The dirname component of this path

property suffix

The suffix of this file

property suffixes

This is a list of all suffixes

property uid

The user that owns this path. The returned value is a `FSUser` object which behaves like an `int` (as expected from `uid`), but it also has a `.name` attribute that holds the string-name of the user

property gid

The group that owns this path. The returned value is a `FSUser` object which behaves like an `int` (as expected from `gid`), but it also has a `.name` attribute that holds the string-name of the group

join(**parts*)

Joins this path with any number of paths

list()

Returns the files in this directory

iterdir()

Returns an iterator over the directory. Might be slightly faster on Python 3.5 than .list()

is_dir()

Returns `True` if this path is a directory, `False` otherwise

is_file()

Returns `True` if this path is a regular file, `False` otherwise

is_symlink()

Returns `True` if this path is a symbolic link, `False` otherwise

exists()

Returns `True` if this path exists, `False` otherwise

stat()

Returns the os.stats for a file

with_name(*name*)

Returns a path with the name replaced

with_suffix(*suffix*, *depth=1*)

Returns a path with the suffix replaced. Up to last `depth` suffixes will be replaced. None will replace all suffixes. If there are less than `depth` suffixes, this will replace all suffixes. `.tar.gz` is an example where `depth=2` or `depth=None` is useful

glob(*pattern*)

Returns a (possibly empty) list of paths that matched the glob-pattern under this path

delete()

Deletes this path (recursively, if a directory)

unlink()

Deletes a symbolic link

move(*dst*)

Moves this path to a different location

copy(*dst*, *override=False*)

Copies this path (recursively, if a directory) to the destination path "dst". Raises TypeError if dst exists and override is False. Will overwrite if override is True. Will silently fail to copy if override is None (the default).

mkdir(*mode=None*, *parents=True*, *exist_ok=True*)

Creates a directory at this path.

Parameters

- **mode** – **Currently only implemented for local paths!** Numeric mode to use for directory creation, which may be ignored on some systems. The current implementation reproduces the behavior of `os.mkdir` (i.e., the current umask is first masked out), but this may change for remote paths. As with `os.mkdir`, it is recommended to call *chmod()* explicitly if you need to be sure.
- **parents** – If this is true (the default), the directory's parents will also be created if necessary.
- **exist_ok** – If this is true (the default), no exception will be raised if the directory already exists (otherwise `OSError`).

Note that the defaults for `parents` and `exist_ok` are the opposite of what they are in Python's own `pathlib` - this is to maintain backwards-compatibility with Plumbum's behaviour from before they were implemented.

read(*encoding=None*)

returns the contents of this file as a `str`. By default the data is read as text, but you can specify the encoding, e.g., `'latin1'` or `'utf8'`

write(*data*, *encoding=None*)

writes the given data to this file. By default the data is written as-is (either text or binary), but you can specify the encoding, e.g., `'latin1'` or `'utf8'`

touch()

Update the access time. Creates an empty file if none exists.

chown(*owner=None*, *group=None*, *recursive=None*)

Change ownership of this path.

Parameters

- **owner** – The owner to set (either `uid` or `username`), optional
- **group** – The group to set (either `gid` or `groupname`), optional
- **recursive** – whether to change ownership of all contained files and subdirectories. Only meaningful when `self` is a directory. If `None`, the value will default to `True` if `self` is a directory, `False` otherwise.

chmod(*mode*)

Change the mode of path to the numeric mode.

Parameters
mode – file mode as for os.chmod

access(*mode=0*)

Test file existence or permission bits

Parameters
mode – a bitwise-or of access bits, or a string-representation thereof: `'f'`, `'x'`, `'r'`, `'w'` for `os.F_OK`, `os.X_OK`, `os.R_OK`, `os.W_OK`

link(*dst*)

Creates a hard link from `self` to `dst`

Parameters
dst – the destination path

symlink(*dst*)

Creates a symbolic link from `self` to `dst`

Parameters
dst – the destination path

open(*mode='r'*, *bufsize=-1*, ***, *encoding=None*)

Opens this path as a file.

Only works for ParamikoMachine-associated paths for now.

as_uri(*scheme='ssh'*)

Returns a universal resource identifier. Use `scheme` to force a scheme.

property stem

The name without an extension, or the last component of the path

property root

The root of the file tree (`/` on Unix)

property drive

The drive letter (on Windows)

class plumbum.path.remote.**RemoteWorkdir**(*remote*)

Remote working directory manipulator

static __new__(*cls*, *remote*)

__hash__()

Return hash(self).

chdir(*newdir*)

Changes the current working directory to the given one

getpath()

Returns the current working directory as a *remote path* <*plumbum.path.remote.RemotePath*> object

__call__(*newdir*)

A context manager used to `chdir` into a directory and then `chdir` back to the previous location; much like `pushd`/`popd`.

Parameters

newdir – The destination director (a string or a *RemotePath*)

5.4.1 Utils

plumbum.path.utils.**delete**(**paths*)

Deletes the given paths. The arguments can be either strings, *local paths*, *remote paths*, or iterables of such. No error is raised if any of the paths does not exist (it is silently ignored)

plumbum.path.utils.**move**(*src*, *dst*)

Moves the source path onto the destination path; `src` and `dst` can be either strings, *LocalPaths* or *RemotePath*; any combination of the three will work.

New in version 1.3: `src` can also be a list of strings/paths, in which case `dst` must not exist or be a directory.

plumbum.path.utils.**copy**(*src*, *dst*)

Copy (recursively) the source path onto the destination path; `src` and `dst` can be either strings, *LocalPaths* or *RemotePath*; any combination of the three will work.

New in version 1.3: `src` can also be a list of strings/paths, in which case `dst` must not exist or be a directory.

plumbum.path.utils.**gui_open**(*filename*)

This selects the proper gui open function. This can also be achieved with webbrowser, but that is not supported.

5.5 Package plumbum.fs

File system utilities

Atomic file operations

class plumbum.fs.atomic.**AtomicFile**(*filename*, *ignore_deletion=False*)

Atomic file operations implemented using file-system advisory locks (`flock` on POSIX, `LockFile` on Windows).

Note: On Linux, the manpage says `flock` might have issues with NFS mounts. You should take this into account.

New in version 1.3.

reopen()

Close and reopen the file; useful when the file was deleted from the file system by a different process

locked(*blocking=True*)

A context manager that locks the file; this function is reentrant by the thread currently holding the lock.

Parameters
blocking – if `True`, the call will block until we can grab the file system lock. if `False`, the call may fail immediately with the underlying exception (`IOError` or `WindowsError`)

delete()

Atomically delete the file (holds the lock while doing it)

read_atomic()

Atomically read the entire file

read_shared()

Read the file **without** holding the lock

write_atomic(*data*)

Writes the given data atomically to the file. Note that it overwrites the entire file; `write_atomic("foo")` followed by `write_atomic("bar")` will result in only `"bar"`.

class plumbum.fs.atomic.**AtomicCounterFile**(*atomicfile*, *initial=0*)

An atomic counter based on AtomicFile. Each time you call `next()`, it will atomically read and increment the counter's value, returning its previous value

Example:

```
acf = AtomicCounterFile.open("/some/file")
print(acf.next())  # e.g., 7
print(acf.next())  # 8
print(acf.next())  # 9
```

New in version 1.3.

classmethod open(*filename*)

Shortcut for `AtomicCounterFile(AtomicFile(filename))`

reset(*value=None*)

Reset the counter's value to the one given. If `None`, it will default to the initial value provided to the constructor

next()

Read and increment the counter, returning its previous value

exception plumbum.fs.atomic.**PidFileTaken**(*msg*, *pid*)

This exception is raised when PidFile.acquire fails to lock the pid file. Note that it derives from `SystemExit`, so unless explicitly handled, it will terminate the process cleanly

class plumbum.fs.atomic.**PidFile**(*filename*)

A PID file is a file that's locked by some process from the moment it starts until it dies (the OS will clear the lock when the process exits). It is used to prevent two instances of the same process (normally a daemon) from running concurrently. The PID file holds its process' PID, so you know who's holding it.

New in version 1.3.

acquire()

Attempt to acquire the PID file. If it's already locked, raises `PidFileTaken`. You should normally acquire the file as early as possible when the program starts

release()

Release the PID file (should only happen when the program terminates)

class plumbum.fs.mounts.**MountEntry**(*dev*, *point*, *fstype*, *options*)

Represents a mount entry (device file, mount point and file system type)

plumbum.fs.mounts.**mount_table**()

returns the system's current mount table (a list of MountEntry objects)

plumbum.fs.mounts.**mounted**(*fs*)

Indicates if a the given filesystem (device file or mount point) is currently mounted

5.6 Package plumbum.colors

Factory for styles. Holds font styles, FG and BG objects representing colors, and imitates the FG ColorFactory to a large degree.

plumbum.colors.**__dir__**()

Default dir() implementation.

plumbum.colors.**__format__**(*format_spec*, /)

Default object formatter.

plumbum.colors.**__init_subclass__**()

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

plumbum.colors.**__new__**(**args*, ***kwargs*)

Create and return a new object. See help(type) for accurate signature.

plumbum.colors.**__reduce__**()

Helper for pickle.

plumbum.colors.**__reduce_ex__**(*protocol*, /)

Helper for pickle.

plumbum.colors.**__sizeof__**()

Size of object in memory, in bytes.

plumbum.colors.**__subclasshook__**()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

5.6.1 plumbum.colorlib

The `ansicolor` object provides `bg` and `fg` to access colors, and attributes like bold and underlined text. It also provides `reset` to recover the normal font.

class plumbum.colorlib.**ANSIStyle**(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This is a subclass for ANSI styles. Use it to get color on sys.stdout tty terminals on posix systems.

Set `use_color = True/False` if you want to control color for anything using this Style.

__str__()

Base Style does not implement a __str__ representation. This is the one required method of a subclass.

exception plumbum.colorlib.**ColorNotFound**

Thrown when a color is not valid for a particular method.

__weakref__

list of weak references to the object (if defined)

class plumbum.colorlib.**HTMLStyle**(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This was meant to be a demo of subclassing Style, but actually can be a handy way to quickly color html text.

**end = '
\n'**

The endline character. Override if needed in subclasses.

__str__()

Base Style does not implement a __str__ representation. This is the one required method of a subclass.

class plumbum.colorlib.**Style**(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This class allows the color changes to be called directly to write them to stdout, `[]` calls to wrap colors (or the `.wrap` method) and can be called in a with statement.

color_class

alias of *Color*

end = '\n'

The endline character. Override if needed in subclasses.

ANSI_REG = re.compile('\x1b\\[([\\d;]+)m')

The regular expression that finds ansi codes in a string.

property stdout

This property will allow custom, class level control of stdout. It will use current sys.stdout if set to None (default). Unfortunately, it only works on an instance..

__init__(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This is usually initialized from a factory.

invert()

This resets current color(s) and flips the value of all attributes present

property reset

Shortcut to access reset as a property.

__copy__()

Copy is supported, will make dictionary and colors unique.

__invert__()

This allows ~color.

__add__(*other*)

Adding two matching Styles results in a new style with the combination of both. Adding with a string results in the string concatenation of a style.

Addition is non-commutative, with the rightmost Style property being taken if both have the same property. (Not safe)

__radd__(*other*)

This only gets called if the string is on the left side. (Not safe)

wrap(*wrap_this*)

Wrap a string in this style and its inverse.

__and__(*other*)

This class supports `color & color2` syntax, and `color & "String" syntax too.`

__rand__(*other*)

This class supports `"String:" & color` syntax.

__ror__(*other*)

Support for "String" | color syntax

__or__(*other*)

This class supports `color | color2` syntax. It also supports `"color | "String"` syntax too.

__call__()

This is a shortcut to print color immediately to the stdout. (Not safe)

now()

Immediately writes color to stdout. (Not safe)

print(**printables*, ***kargs*)

This acts like print; will print that argument to stdout wrapped in Style with the same syntax as the print function in 3.4.

print_(**printables*, ***kargs*)

DEPRECATED: Shortcut from classic Python 2

__getitem__(*wrapped*)

The [] syntax is supported for wrapping

__enter__()

Context manager support

__exit__(*_type*, *_value*, *_traceback*)

Runs even if exception occurred, does not catch it.

property ansi_codes

Generates the full ANSI code sequence for a Style

property ansi_sequence

This is the string ANSI sequence.

__repr__()

Return repr(self).

__eq__(*other*)

Equality is true only if reset, or if attributes, fg, and bg match.

abstract __str__()

Base Style does not implement a __str__ representation. This is the one required method of a subclass.

classmethod from_ansi(*ansi_string*, *filter_resets=False*)

This generated a style from an ansi string. Will ignore resets if filter_resets is True.

add_ansi(*sequence*, *filter_resets=False*)

Adds a sequence of ansi numbers to the class. Will ignore resets if filter_resets is True.

classmethod string_filter_ansi(*colored_string*)

Filters out colors in a string, returning only the name.

classmethod string_contains_colors(*colored_string*)

Checks to see if a string contains colors.

to_representation(*rep*)

This converts both colors to a specific representation

limit_representation(*rep*)

This only converts if true representation is higher

property basic

The color in the 8 color representation.

property simple

The color in the 16 color representation.

property full

The color in the 256 color representation.

property true

The color in the true color representation.

__hash__ = None

class plumbum.colorlib.**StyleFactory**(*style*)

Factory for styles. Holds font styles, FG and BG objects representing colors, and imitates the FG ColorFactory to a large degree.

__init__(*style*)

property use_color

Shortcut for setting color usage on Style

from_ansi(*ansi_sequence*)

Calling this is a shortcut for creating a style from an ANSI sequence.

property stdout

This is a shortcut for getting stdout from a class without an instance.

get_colors_from_string(*color=''*)

Sets color based on string, use . or space for separator, and numbers, fg/bg, htmlcodes, etc all accepted (as strings).

filter(*colored_string*)

Filters out colors in a string, returning only the name.

contains_colors(*colored_string*)

Checks to see if a string contains colors.

extract(*colored_string*)

Gets colors from an ansi string, returns those colors

plumbum.colorlib.**main**()

Color changing script entry. Call using python3 -m plumbum.colors, will reset if no arguments given.

5.6.2 plumbum.colorlib.styles

This file provides two classes, *Color* and *Style*.

`Color` is rarely used directly, but merely provides the workhorse for finding and manipulating colors.

With the `Style` class, any color can be directly called or given to a with statement.

class plumbum.colorlib.styles.**Color**(*r_or_color=None*, *g=None*, *b=None*, *fg=True*)

Loaded with `(r, g, b, fg)` or `(color, fg=fg)`. The second signature is a short cut and will try full and hex loading.

This class stores the idea of a color, rather than a specific implementation. It provides as many different tools for representations as possible, and can be subclassed to add more representations, though that should not be needed for most situations. `.from_` class methods provide quick ways to create colors given different representations. You will not usually interact with this class.

Possible colors:

```
reset = Color() # The reset color by default
background_reset = Color(fg=False) # Can be a background color
blue = Color(0,0,255) # Red, Green, Blue
green = Color.from_full("green") # Case insensitive name, from large colorset
red = Color.from_full(1) # Color number
white = Color.from_html("#FFFFFF") # HTML supported
yellow = Color.from_simple("red") # Simple colorset
```

The attributes are:

reset

True it this is a reset color (following attributes don't matter if True)

rgb

The red/green/blue tuple for this color

simple

If true will stay to 16 color mode.

number

The color number given the mode, closest to rgb if not rgb not exact, gives position of closest name.

fg

This is a foreground color if True. Background color if False.

__init__(*r_or_color=None*, *g=None*, *b=None*, *fg=True*)

This works from color values, or tries to load non-simple ones.

number

Number of the original color, or closest color

representation

0 for off, 1 for 8 colors, 2 for 16 colors, 3 for 256 colors, 4 for true color

exact

This is false if the named color does not match the real color

classmethod from_simple(*color*, *fg=True*)

Creates a color from simple name or color number

classmethod from_full(*color*, *fg=True*)

Creates a color from full name or color number

classmethod from_hex(*color*, *fg=True*)

Converts #123456 values to colors.

property name

The (closest) name of the current color

property name_camelcase

The camelcase name of the color

__repr__()

This class has a smart representation that shows name and color (if not unique).

__eq__(*other*)

Reset colors are equal, otherwise rgb have to match.

property ansi_sequence

This is the ansi sequence as a string, ready to use.

property ansi_codes

This is the full ANSI code, can be reset, simple, 256, or full color.

property hex_code

This is the hex code of the current color, html style notation.

__str__()

This just prints it's simple name

to_representation(*val*)

Converts a color to any representation

limit_representation(*val*)

Only converts if val is lower than representation

__hash__ = None

class plumbum.colorlib.styles.**Style**(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This class allows the color changes to be called directly to write them to stdout, `[]` calls to wrap colors (or the `.wrap` method) and can be called in a with statement.

color_class

The class of color to use. Never hardcode `Color` call when writing a Style method.

alias of *Color*

end = '\n'

The endline character. Override if needed in subclasses.

ANSI_REG = re.compile('\x1b\\[([\\d;]+)m')

The regular expression that finds ansi codes in a string.

property stdout

This property will allow custom, class level control of stdout. It will use current sys.stdout if set to None (default). Unfortunately, it only works on an instance..

__init__(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This is usually initialized from a factory.

invert()

This resets current color(s) and flips the value of all attributes present

property reset

Shortcut to access reset as a property.

__copy__()

Copy is supported, will make dictionary and colors unique.

__invert__()

This allows ~color.

__add__(*other*)

Adding two matching Styles results in a new style with the combination of both. Adding with a string results in the string concatenation of a style.

Addition is non-commutative, with the rightmost Style property being taken if both have the same property. (Not safe)

__radd__(*other*)

This only gets called if the string is on the left side. (Not safe)

wrap(*wrap_this*)

Wrap a string in this style and its inverse.

__and__(*other*)

This class supports `color & color2` syntax, and `color & "String" syntax too.`

__rand__(*other*)

This class supports `"String:" & color` syntax.

__ror__(*other*)

Support for "String" | color syntax

__or__(*other*)

This class supports `color | color2` syntax. It also supports `"color | "String"` syntax too.

__call__()

This is a shortcut to print color immediately to the stdout. (Not safe)

now()

Immediately writes color to stdout. (Not safe)

print(**printables*, ***kargs*)

This acts like print; will print that argument to stdout wrapped in Style with the same syntax as the print function in 3.4.

print_(**printables*, ***kargs*)

DEPRECATED: Shortcut from classic Python 2

__getitem__(*wrapped*)

The [] syntax is supported for wrapping

__enter__()

Context manager support

__exit__(*_type*, *_value*, *_traceback*)

Runs even if exception occurred, does not catch it.

property ansi_codes

Generates the full ANSI code sequence for a Style

property ansi_sequence

This is the string ANSI sequence.

__repr__()

Return repr(self).

__eq__(*other*)

Equality is true only if reset, or if attributes, fg, and bg match.

abstract __str__()

Base Style does not implement a __str__ representation. This is the one required method of a subclass.

classmethod from_ansi(*ansi_string*, *filter_resets=False*)

This generated a style from an ansi string. Will ignore resets if filter_resets is True.

add_ansi(*sequence*, *filter_resets=False*)

Adds a sequence of ansi numbers to the class. Will ignore resets if filter_resets is True.

classmethod string_filter_ansi(*colored_string*)

Filters out colors in a string, returning only the name.

classmethod string_contains_colors(*colored_string*)

Checks to see if a string contains colors.

to_representation(*rep*)

This converts both colors to a specific representation

limit_representation(*rep*)

This only converts if true representation is higher

property basic

The color in the 8 color representation.

property simple

The color in the 16 color representation.

property full

The color in the 256 color representation.

property true

The color in the true color representation.

__hash__ = None

class plumbum.colorlib.styles.**ANSIStyle**(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This is a subclass for ANSI styles. Use it to get color on sys.stdout tty terminals on posix systems.

Set `use_color = True/False` if you want to control color for anything using this Style.

__str__()

Base Style does not implement a __str__ representation. This is the one required method of a subclass.

class plumbum.colorlib.styles.**HTMLStyle**(*attributes=None*, *fgcolor=None*, *bgcolor=None*, *reset=False*)

This was meant to be a demo of subclassing Style, but actually can be a handy way to quickly color html text.

**end = '
\n'**

The endline character. Override if needed in subclasses.

__str__()

Base Style does not implement a __str__ representation. This is the one required method of a subclass.

exception plumbum.colorlib.styles.**ColorNotFound**

Thrown when a color is not valid for a particular method.

__weakref__

list of weak references to the object (if defined)

exception plumbum.colorlib.styles.**AttributeNotFound**

Similar to color not found, only for attributes.

__weakref__

list of weak references to the object (if defined)

5.6.3 plumbum.colorlib.factories

Color-related factories. They produce Styles.

class plumbum.colorlib.factories.**ColorFactory**(*fg*, *style*)

This creates color names given fg = True/False. It usually will be called as part of a StyleFactory.

__init__(*fg*, *style*)

__getattr__(*item*)

Full color names work, but do not populate __dir__.

full(*name*)

Gets the style for a color, using standard name procedure: either full color name, html code, or number.

simple(*name*)

Return the extended color scheme color for a value or name.

rgb(*r*, *g=None*, *b=None*)

Return the extended color scheme color for a value.

hex(*hexcode*)

Return the extended color scheme color for a value.

ansi(*ansiseq*)

Make a style from an ansi text sequence

__getitem__(*val*)

Shortcut to provide way to access colors numerically or by slice. If end <= 16, will stay to simple ANSI version.

__call__(*val_or_r=None*, *g=None*, *b=None*)

Shortcut to provide way to access colors.

__iter__()

Iterates through all colors in extended colorset.

__invert__()

Allows clearing a color with ~

__enter__()

This will reset the color on leaving the with statement.

__exit__(*_type: Any*, *_value: Any*, *_traceback: Any*) → None

This resets a FG/BG color or all styles, due to different definition of RESET for the factories.

__repr__()

Simple representation of the class by name.

__weakref__

list of weak references to the object (if defined)

class plumbum.colorlib.factories.**StyleFactory**(*style*)

Factory for styles. Holds font styles, FG and BG objects representing colors, and imitates the FG ColorFactory to a large degree.

__init__(*style*)

property use_color

Shortcut for setting color usage on Style

from_ansi(*ansi_sequence*)

Calling this is a shortcut for creating a style from an ANSI sequence.

property stdout

This is a shortcut for getting stdout from a class without an instance.

get_colors_from_string(*color=''*)

Sets color based on string, use . or space for separator, and numbers, fg/bg, htmlcodes, etc all accepted (as strings).

filter(*colored_string*)

Filters out colors in a string, returning only the name.

contains_colors(*colored_string*)

Checks to see if a string contains colors.

extract(*colored_string*)

Gets colors from an ansi string, returns those colors

5.6.4 plumbum.colorlib.names

Names for the standard and extended color set. Extended set is similar to vim wiki, colored, etc. Colors based on wikipedia.

You can access the index of the colors with names.index(name). You can access the rgb values with `r=int(html[n][1:3],16)`, etc.

`plumbum.colorlib.names.`**`color_codes_simple = [0, 1, 2, 3, 4, 5, 6, 7, 60, 61, 62, 63, 64, 65, 66, 67]`**

Simple colors, remember that reset is #9, second half is non as common.

class `plumbum.colorlib.names.`**`FindNearest`**(*r: int*, *g: int*, *b: int*)

This is a class for finding the nearest color given rgb values. Different find methods are available.

`__init__`(*r: int*, *g: int*, *b: int*) → None

`only_basic`()

This will only return the first 8 colors! Breaks the colorspace into cubes, returns color

`all_slow`(*color_slice: slice = slice(None, None, None)*) → int

This is a slow way to find the nearest color.

`only_colorblock`() → int

This finds the nearest color based on block system, only works for 17-232 color values.

`only_simple`() → int

Finds the simple color-block color.

`only_grey`() → int

Finds the greyscale color.

`all_fast`() → int

Runs roughly 8 times faster than the slow version.

`__weakref__`

list of weak references to the object (if defined)

`plumbum.colorlib.names.`**`from_html`**(*color: str*) → Tuple[int, int, int]

Convert html hex code to rgb.

`plumbum.colorlib.names.`**`to_html`**(*r*, *g*, *b*)

Convert rgb to html hex code.

5.7 Colorlib design

New in version 1.6.

The purpose of this document is to describe the system that plumbum.colors implements. This system was designed to be flexible and to allow implementing new color backends. Hopefully this document will allow future work on colorlib to be as simple as possible.

Note: Enabling color

`plumbum.colors` tries to guess the color output settings of your system. You can force the use of color globally by setting `colors.use_color=True` See *256 Color Support* for more options.

5.7.1 Generating colors

Styles are accessed through the `colors` object, which is an instance of a StyleFactory. The `colors` object is actually an imitation module that wraps `plumbum.colorlib.ansicolors` with module-like access. Thus, things like from `plumbum.colors.bg import red` work also. The library actually lives in `plumbum.colorlib`.

Style Factory

The `colors` object has the following available objects:

`fg` and `bg`
: The foreground and background colors, reset to default with `colors.fg.reset` or `~colors.fg` and likewise for `bg`. These are `ColorFactory` instances.

`bold`, `dim`, `underline`, `italics`, `reverse`, `strikeout`, and `hidden`
: All the *ANSI* modifiers are available, as well as their negations, such as `~colors.bold` or `colors.bold.reset`, etc. (These are generated automatically based on the Style attached to the factory.)

`reset`
: The global reset will restore all properties at once.

`do_nothing`
: Does nothing at all, but otherwise acts like any `Style` object. It is its own inverse. Useful for `cli` properties.

The `colors` object can be used in a with statement, which resets all styles on leaving the statement body. Although factories do support some of the same methods as a Style, their primary purpose is to generate Styles. The colors object has a `use_color` property that can be set to force the use of color. A `stdout` property is provided to make changing the output of color statement easier. A `colors.from_ansi(code)` method allows you to create a Style from any ansi sequence, even complex or combined ones.

Color Factories

The `colors.fg` and `colors.bg` are `ColorFactory`'s. In fact, the colors object itself acts exactly like the `colors.fg` object, with the exception of the properties listed above.

Named foreground colors are available directly as methods. The first 16 primary colors, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, etc, as well as `reset`, are available. All 256 color names are available, but do not populate factory directly, so that auto-completion gives reasonable results. You can also access colors using strings and do `colors[string]`. Capitalization, underscores, and spaces (for strings) will be ignored.

You can also access colors numerically with `colors(n)` or `colors[n]` with the extended 256 color codes. The former will default to simple versions of colors for the first 16 values. The later notation can also be used to slice. Full hex codes can be used, too. If no match is found, these will be the true 24 bit color value.

The `fg` and `bg` also can be put in with statements, and they will restore the foreground and background color only, respectively.

`colors.rgb(r,g,b)` will create a color from an input red, green, and blue values (integers from 0-255). `colors.rgb(code)` will allow you to input an html style hex sequence. These work on `fg` and `bg` too. The `repr` of styles is smart and will show you the closest color to the one you selected if you didn't exactly select a color through RGB.

5.7.2 Style manipulations

Safe color manipulations refer to changes that reset themselves at some point. Unsafe manipulations must be manually reset, and can leave your terminal color in an unreadable state if you forget to reset the color or encounter an exception. If you do get the color unset on a terminal, the following, typed into the command line, will restore it:

```
$ python3 -m plumbum.colors
```

This also supports command line access to unsafe color manipulations, such as

```
$ python3 -m plumbum.colors blue
$ python3 -m plumbum.colors bg red
$ python3 -m plumbum.colors fg 123
$ python3 -m plumbum.colors bg reset
$ python3 -m plumbum.colors underline
```

You can use any path or number available as a style.

Unsafe Manipulation

Styles have two unsafe operations: Concatenation (with + and a string) and calling `.now()` without arguments (directly calling a style without arguments is also a shortcut for `.now()`). These two operations do not restore normal color to the terminal by themselves. To protect their use, you should always use a context manager around any unsafe operation.

An example of the usage of unsafe `colors` manipulations inside a context manager:

```
from plumbum import colors

with colors:
    colors.fg.red.now()
    print('This is in red')
    colors.green.now()
    print('This is green ' + colors.underline + 'and now also underlined!')
    print('Underlined' + colors.underline.reset + ' and not underlined but still red')
print('This is completely restored, even if an exception is thrown!')
```

Output:

We can use `colors` instead of `colors.fg` for foreground colors. If we had used `colors.fg` as the context manager, then non-foreground properties, such as `colors.underline` or `colors.bg.yellow`, would not have reset those properties. Each attribute, as well as `fg`, `bg`, and `colors` all have inverses in the ANSI standard. They are accessed with `~` or `.reset`, and can be used to manually make these operations safer, but there is a better way.

Safe Manipulation

All other operations are safe; they restore the color automatically. The first, and hopefully already obvious one, is using a Style rather than a `colors` or `colors.fg` object in a `with` statement. This will set the color (using sys.stdout by default) to that color, and restore color on leaving.

The second method is to manually wrap a string. This can be done with `color.wrap("string")` or `color["string"]`. These produce strings that can be further manipulated or printed.

Finally, you can also print a color to stdout directly using `color.print("string")`. This has the same syntax as the print function.

An example of safe manipulations:

```
colors.fg.yellow('This is yellow', end='')
print(' And this is normal again.')
with colors.red:
    print('Red color!')
    with colors.bold:
        print("This is red and bold.")
    print("Not bold, but still red.")
print("Not red color or bold.")
print((colors.magenta & colors.bold)["This is bold and colorful!"], "And this is not.")
```

Output:

Style Combinations

You can combine styles with & and they will create a new combined Style object. Colors will not be "summed" or otherwise combined; the rightmost color will be used (this matches the expected effect of applying the Styles individually to the strings). However, combined Styles are intelligent and know how to reset just the properties that they contain. As you have seen in the example above, the combined style `(colors.magenta & colors.bold)` can be used in any way a normal Style can. Since wrapping is done with `|`, the Python order of operations causes styles to be combined first, then wrapping is done last.

5.7.3 256 Color Support

While this library supports full 24 bit colors through escape sequences, the library has special support for the "full" 256 colorset through numbers, names or HEX html codes. Even if you use 24 bit color, the closest name is displayed in the `repr`. You can access the colors as as `colors.fg.Light_Blue`, `colors.fg.lightblue`, `colors.fg[12]`, `colors.fg('Light_Blue')`, `colors.fg('LightBlue')`, or `colors.fg('#0000FF')`. You can also iterate or slice the `colors`, `colors.fg`, or `colors.bg` objects. Slicing even intelligently downgrades to the simple version of the codes if it is within the first 16 elements. The supported colors are:

If you want to enforce a specific representation, you can use `.basic` (8 color), `.simple` (16 color), `.full` (256 color), or `.true` (24 bit color) on a Style, and the colors in that Style will conform to the output representation and name of the best match color. The internal RGB colors are remembered, so this is a non-destructive operation.

To limit the use of color to one of these styles, set `colors.use_color` to 1 for 8 colors, 2 for 16 colors, 3 for 256 colors, or 4 for true color. It will be guessed based on your system on initialisation.

5.7.4 The Classes

The library consists of three primary classes, the `Color` class, the `Style` class, and the `StyleFactory` class. The following portion of this document is primarily dealing with the working of the system, and is meant to facilitate extensions or work on the system.

The `Color` class provides meaning to the concept of color, and can provide a variety of representations for any color. It can be initialised from r,g,b values, or hex codes, 256 color names, or the simple color names via classmethods. If initialized without arguments, it is the reset color. It also takes an fg True/False argument to indicate which color it is. You probably will not be interacting with the Color class directly, and you probably will not need to subclass it, though new extensions to the representations it can produce are welcome.

The `Style` class hold two colors and a dictionary of attributes. It is the workhorse of the system and is what is produced by the `colors` factory. It holds `Color` as `.color_class`, which can be overridden by subclasses (again, this usually is not needed). To create a color representation, you need to subclass `Style` and give it a working `__str__` definition. `ANSIStyle` is derived from `Style` in this way.

The factories, `ColorFactory` and `StyleFactory`, are factory classes that are meant to provide simple access to 1 style Style classes. To use, you need to initialize an object of `StyleFactory` with your intended Style. For example, `colors` is created by:

```
colors = StyleFactory(ANSIStyle)
```

Subclassing Style

For example, if you wanted to create an HTMLStyle and HTMLcolors, you could do:

```
class HTMLStyle(Style):
    attribute_names = dict(bold='b', li='li', code='code')
    end = '<br/>\n'

    def __str__(self):
        result = ''

        if self.bg and not self.bg.reset:
            result += f'<span style="background-color: {self.bg.hex_code}">'
        if self.fg and not self.fg.reset:
            result += f'<font color="{self.fg.hex_code}">'
        for attr in sorted(self.attributes):
            if self.attributes[attr]:
                result += '<' + self.attribute_names[attr] + '>'

        for attr in reversed(sorted(self.attributes)):
            if not self.attributes[attr]:
                result += '</' + self.attribute_names[attr].split()[0] + '>'
        if self.fg and self.fg.reset:
            result += '</font>'
        if self.bg and self.bg.reset:
            result += '</span>'

        return result

htmlcolors = StyleFactory(HTMLStyle)
```

This doesn't support global resets, since that's not how HTML works, but otherwise is a working implementation. This is an example of how easy it is to add support for other output formats.

An example of usage:

```
>>>  htmlcolors.bold & htmlcolors.red | "This is colored text"
'<font color="#800000"><b>This is colored text</b></font>'
```

The above color table can be generated with:

```
for color in htmlcolors:
    htmlcolors.li(
        "&#x25a0;" | color,
        color.fg.hex_code | htmlcolors.code,
        color.fg.name_camelcase)
```

Note: `HTMLStyle` is implemented in the library, as well, with the `htmlcolors` object available in `plumbum.colorlib`. It was used to create the colored output in this document, with small changes because `colors.reset` cannot be supported with HTML.

5.7.5 See Also

- colored Another library with 256 color support
- **colorama A library that supports colored text on Windows,**
  can be combined with Plumbum.colors (if you force `use_color`, doesn't support all extended colors)

Note: The `local` object is an instance of a `machine`.

CHAPTER

SIX

ABOUT

The original purpose of Plumbum was to enable local and remote program execution with ease, assuming nothing fancier than good-old SSH. On top of this, a file-system abstraction layer was devised, so that working with local and remote files would be seamless.

I've toyed with this idea for some time now, but it wasn't until I had to write build scripts for a project I've been working on that I decided I've had it with shell scripts and it's time to make it happen. Plumbum was born from the scraps of the `Path` class, which I wrote for the aforementioned build system, and the `SshContext` and `SshTunnel` classes that I wrote for RPyC. When I combined the two with *shell combinators* (because shell scripts do have an edge there) the magic happened and here we are.

CHAPTER

SEVEN

CREDITS

The project has been inspired by **PBS** (now called sh) of Andrew Moffat, and has borrowed some of his ideas (namely treating programs like functions and the nice trick for importing commands). However, I felt there was too much magic going on in PBS, and that the syntax wasn't what I had in mind when I came to write shell-like programs. I contacted Andrew about these issues, but he wanted to keep PBS this way. Other than that, the two libraries go in different directions, where Plumbum attempts to provide a more wholesome approach.

Plumbum also pays tribute to Rotem Yaari who suggested a library code-named `pyplatform` for that very purpose, but which had never materialized.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W