
Paver Manual

Release 1.3.4

Kevin Dangoor

January 16, 2018

Contents

1	Features	3
2	Status	5
3	Installation	7
4	Help and Development	9
4.1	Running test suite	9
5	License	11
6	Contents	13
6.1	Foreword: Why Paver?	13
6.2	Paver's Features	14
6.3	Getting Started with Paver	15
6.4	pavement.py in depth	21
6.5	The Paver Standard Library	26
6.6	Paver Command Line	32
6.7	Tips and Tricks	33
6.8	Articles about Paver	33
6.9	Complete API Reference	33
6.10	Paver Changelog	33
6.11	Credits	41
7	Indices and tables	43
	Python Module Index	45



Japanese translation available thanks to Tetsuya Morimoto. Latest stable documentation is on PyPI, latest development docs are [on GitHub<https://github.com/paver/paver/tree/master/docs>](https://github.com/paver/paver/tree/master/docs)‘_

Paver is a Python-based software project scripting tool along the lines of Make or Rake. It is not designed to handle the dependency tracking requirements of, for example, a C program. It *is* designed to help out with all of your other repetitive tasks (run documentation generators, moving files around, downloading things), all with the convenience of Python’s syntax and massive library of code.

If you’re developing applications in Python, you get even more... Most public Python projects use distutils or setuptools to create source tarballs for distribution. (Private projects can take advantage of this, too!) Have you ever wanted to generate the docs before building the source distribution? With Paver, you can, trivially. Here’s a complete pavement.py:

```
from paver.easy import *
from paver.setuputils import setup

setup(
    name="MyCoolProject",
    packages=['mycool'],
    version="1.0",
    url="https://www.blueskyonmars.com/",
    author="Kevin Dangoor",
    author_email="dangoor@gmail.com"
)

@task
@needs(['html', "distutils.command.sdist"])
def sdist():
    """Generate docs and source distribution."""
    pass
```

With that pavement file, you can just run `paver sdist`, and your docs will be rebuilt automatically before creating the source distribution. It’s also easy to move the generated docs into some other directory (and, of course, you can tell Paver where your docs are stored, if they’re not in the default location.)

- Build files are *just Python*
- *One file with one syntax*, `pavement.py`, knows how to manage your project
- *File operations* are unbelievably easy, thanks to the built-in version of Jason Orendorff's `path.py`.
- Need to do something that takes 5 lines of code? *It'll only take 5 lines of code..*
- Completely encompasses *distutils and setuptools* so that you can customize behavior as you need to.
- Wraps *Sphinx* for generating documentation, and adds utilities that make it easier to incorporate fully tested sample code.
- Wraps *Subversion* for working with code that is checked out.
- Wraps *virtualenv* to allow you to trivially create a bootstrap script that gets a virtual environment up and running. This is a great way to install packages into a contained environment.
- Can use all of these other libraries, but *requires none of them*
- Easily transition from `setup.py` without making your users learn about or even install Paver! (See the *Getting Started Guide* for an example).

See how it works! Check out the *Getting Started Guide*.

Paver was created by [Kevin Dangoor](#) of [SitePen](#).

CHAPTER 2

Status

Paver has been in use in production settings since mid-2008, and significant attention is paid to backwards compatibility since the release of 1.0.

See the *changelog* for more information about recent improvements.

The easiest way to get Paver is if you have `pip` or `distutils_` installed.

```
pip install Paver
```

or

```
easy_install Paver
```

Without `setuptools`, it's still pretty easy. Download the Paver .tgz file from [Paver's Cheeseshop page](#), untar it and run:

```
python setup.py install
```


You can get help from the [mailing list](#).

If you'd like to help out with Paver, you can check the code out from github:

```
git clone https://github.com/paver/paver.git
```

Ideally, create a fork, fix an issue from [Paver's list of issues](#) (or create an issue Yourself) and send a pull request.

Your help is appreciated!

4.1 Running test suite

Paver contains both unit and integration test suite. Unittests are run by either `paver test` or `paver unit`. Integration tests can be run by `paver integrate`.

Using older, system-wide installed paver to run tests on development version can lead to bad interactions (see [issue 33](#)). Please, run paver test suite using development version itself, by:

- Creating virtual environment with `--no-site-packages`
- and
- Installing development version with `python setup.py develop`
- or
- Using embedded minilib, thus invoking commands with `setup.py` instead of `paver`

CHAPTER 5

License

Paver is licensed under a BSD license. See the LICENSE.txt file in the distribution.

6.1 Foreword: Why Paver?

Paver occupies a sweet spot in the python toolset, the design is sound, it's easier than mud to work with at a basic level, and it has a nice grade of descent into more advanced things.

—David Eyk

I didn't want to make a new build tool. Honestly. The main reason that I created Paver is...

6.1.1 The Declarative/Imperative Divide

When you solve the same problem repeatedly, patterns emerge and you're able to easily see ways to reduce how much effort is involved in solving that problem. Often times, you'll end up with a **declarative solution**. Python's standard distutils is a declarative solution. In your call to the `setup()` function, you declare some metadata about your project and provide a list of the parts and you end up with a nice command line interface. That command line interface knows how to make redistributable tarballs (and eggs if you have setuptools), install the package, build C extensions and more.

`zc.buildout` does a similar thing for deployments. Declare the parts of your system and specify a recipe for each part (each recipe with its own collection of options) and you can build up a consistent system with one command.

These tools sound great, don't they? They are great. As long as you don't need to customize the capabilities they provide. For example, it's not uncommon that you'll need to move some file here, create some directory there, etc. Then what?

In an **imperative system**, you'd just add the few lines of code you need.

For distutils and `zc.buildout` and other similar tools, the answer usually involves extra boilerplate surrounding the code in question and then installing that code somewhere. You basically have to create declarative syntax for something that is a one-off rather than a larger, well-understood problem. And, for distutils and `zc.buildout`, you have to use two entirely different mechanisms.

6.1.2 Consistency for Project Related Tasks

And that's the next thing that bothered me with the state of Python tools. It would be nice to have a consistent interface in command line and configuration for the tools that I use to work with my projects. Every tool I bring in to the project adds new command line interfaces, more config files (and some of those config files duplicate project metadata!).

6.1.3 That's Why Paver Is Here

Paver is set up to provide declarative handling of common tasks with as easy an escape hatch to imperative programming as possible (just add a function with a decorator in the same file). Your project-related configuration options are all together and all accessible to different parts of your build and deployment setup. And the language used for everything is Python, so you're not left guessing how to do a `for` loop.

Of course, rebuilding the great infrastructure provided by tools like `distutils` makes no sense. So, Paver just uses `distutils` and other tools directly.

Paver also goes beyond just providing an extension mechanism for `distutils`. It adds a bunch of useful capabilities for things like working with files and directories, elegantly handling sample code for your documentation and building bootstrap scripts to allow your software to easily be installed in an isolated `virtualenv`.

I'm already using Paver for SitePen's Support service user interface project and I use Paver to manage Paver itself! It's been working out great for me, and it's set up in such a way that whatever kind of scripting your project needs it should be pretty simple with Paver.

6.1.4 Making the Switch is Easy

Finally, I've put some time into making sure that moving a project from `distutils` to Paver is easy for everyone involved (people making the projects and people using the projects). Check out the *Getting Started Guide* to see an example of how a project moves from `distutils` to Paver (even maintaining the `python setup.py install` command that everyone's used to!)

Thanks for reading!

Kevin Dangoor May 2008

6.2 Paver's Features

6.2.1 Files Are Just Python

Python has a very concise, readable syntax. There's no need to create some mini-language for describing your builds. Quite often it seems like these mini-languages are missing features that you need. By using Python as its syntax, you can always be sure that you can express whatever it is you need to do easily. A `for` loop is just a `for` loop:

```
for fn in ["f1.txt", "f2.txt", "f3.txt"]:  
    p = path(fn)  
    p.remove()
```

6.2.2 One File with One Syntax

When putting together a Python project today, you get into a collection of tools to get the job done. `distutils` and `setuptools` are the standards for getting packages put together. `zc.buildout` and `virtualenv` are used for installation into isolated deployment environments. `Sphinx` provides a great way to document Python projects.

To put together a total system, you need each of these parts. But they each have their own way of working. The goal with the *Paver Standard Library* is to make the common tools have a more integrated feel, so you don't have to guess as much about how to get something done.

As of today, Paver is tightly integrated with distutils and setuptools, and can easily work as a drop-in, more easily scriptable replacement for setup.py.

6.2.3 Easy file operations

Paver includes a customized version of Jason Orendorff's awesome path.py module. Operations on files and directories could hardly be easier, and the methods have been modified to support "dry run" behavior.

6.2.4 Small bits of behavior take small amounts of work

Imagine you need to do something that will take you 5 lines of Python code. With some of the tools that Paver augments, it'll take you a lot more effort than those 5 lines of code. You have to read about the API for making new commands or recipes or otherwise extending the package. The goal when using Paver is to have a five line change take about five lines to make.

For example, let's say you need to perform some extra work when and 'sdist' is run. Good luck figuring out the best way to do that with distutils. With Paver, it's just:

```
@task
def sdist():
    # perform fancy file manipulations
    blah.blah.blah()

    # *now* run the sdist
    call_task("setuptools.command.sdist")
```

6.2.5 Can Take Advantage of Libraries But Doesn't Require Them

The Paver Standard Library includes support for a lot of the common tools, but you don't necessarily need all of those tools, and certainly not on every project. Paver is designed to have no other requirements but to automatically take advantage of other tools when they're available.

6.3 Getting Started with Paver

Often, the easiest way to get going with a new tool is to see an example in action, so that's how we'll get started with Paver. In the Paver distribution, there are samples under docs/samples. The Getting Started samples are in the "started" directory under there.

6.3.1 The Old Way

Our first sample is called "The Old Way" (and it's in the docs/samples/started/oldway directory). It's a fairly typical project with one Python package and some docs, and we want to be able to distribute it.

Python's distutils makes it easy indeed to create a distributable package. We create a setup.py file that looks like this:

```
#<== include('started/oldway/setup.py')==>
#<==end==>
```

With that simple setup script, you can run:

```
python setup.py sdist
```

to build a source distribution:

```
# <==
# sh('cd docs/samples/started/oldway; python setup.py sdist',
#     insert_output=False)
# sh('ls -l docs/samples/started/oldway/dist')
# ==>
# <==end==>
```

Then your users can run the familiar:

```
python setup.py install
```

to install the package, or use `setuptools`' even easier:

```
easy_install "TheOldWay"
```

for packages that are up on the Python Package Index.

The Old Way's Docs

The Old Way project is at least a bit modern in that it uses [Sphinx](#) for documentation. When you use `sphinx-quickstart` to get going with your docs, Sphinx will give you a Makefile that you can run to generate your HTML docs. So, generating the HTML docs is easy:

```
make html
```

Except, in this project (as in Paver itself), we want to include the HTML files in a docs directory in the package for presenting help to the users. We end up creating a shell script to do this:

```
# <== include("started/oldway/builddocs.sh") ==>
# <==end==>
```

Of course, creating a script like this means that we have to actually remember to run it. We could change this script to “`buildsdist.sh`” and add a `python setup.py sdist` to the end of the file. But, wouldn't it be nicer if we could just use `python setup.py sdist` directly?

You can [create new distutils commands](#), but do you really want to drop stuff like that in the `distutils/command` package in your Python library directory? And how would you call the `sdist` command anyway? [setuptools](#) helps, but it still requires setting up a module and entry point for this collection of commands.

Work with me here

I just *know* there are some people reading this and thinking “man, what a contrived example!”. Building, packaging, distributing and deploying of projects is quite custom for every project. Part of the point of Paver is to make it easy to handle whatever weird requirements arise in your project. This example may seem contrived, but it should give you an idea of how easy Paver makes it to get your tasks done.

6.3.2 The New Way

Let's bring in Paver now to clean up our scripting a bit. Converting a project to use Paver is really, really simple. Recall the setup function from our Old Way setup.py:

```
# <== include("started/oldway/setup.py", "setup")==>
# <==end==>
```

Getting Started with Paver

setup.py is a standard Python script. It's just called setup.py as a convention. Paver works a bit more like Make or Rake. To use Paver, you run `paver <taskname>` and the paver command will look for a pavement.py file in the current directory. pavement.py is a standard Python module. A typical pavement will import from paver.easy to get a bunch of convenience functions and objects and then import other modules that include useful tasks:

```
# <== include('started/newway/pavement.py', 'imports')==>
# <==end==>
```

Converting from setup.py to pavement.py is easy. Paver provides a special `options` object that holds all of your build options. `options` is just a dictionary that allows attribute-style access and has some special searching abilities. The options for distutils operations are stored in a `setup` section of the options. And, as a convenience, Paver provides a `setup` function that sets the values in that options section (and goes a step further, by making all of the distutils/setuptools commands available as Paver tasks). Here's what the conversion looks like:

```
# <== include('started/newway/pavement.py', 'setup')==>
# <==end==>
```

Paver is compatible with distutils

Choosing to use Paver does not mean giving up on distutils or setuptools. Paver lets you continue to use distutils and setuptools commands. When you import a module that has Paver tasks in it, those tasks automatically become available for running. If you want access to distutils and setuptools commands as well, you can either use the `paver.setuputils.setup` function as described above, or call `paver.setuputils.install_distutils_tasks()`.

We can see this in action by looking at `paver help`:

```
# <== sh('cd docs/samples/started/newway; paver help')==>
# <==end==>
```

That command is listing all of the available tasks, and you can see near the top there are tasks from distutils.command. All of the standard distutils commands are available.

There's one more thing we need to do before our Python package is properly redistributable: tell distutils about our special files. We can do that with a simple MANIFEST.in:

```
# <== include('started/newway/MANIFEST.in')==>
# <==end==>
```

With that, we can run `paver sdist` and end up with the equivalent output file:

```
# <==
# sh('cd docs/samples/started/newway; paver sdist',
#     insert_output=False)
# sh('ls -l docs/samples/started/newway/dist')
```

```
# ==>
# <==end==>
```

It also means that users of The New Way can also run `paver install` to install the package on their system. Neat.

But people are used to `setup.py`!

`python setup.py install` has been around a long time. And while you could certainly put a README file in your package telling people to run `paver install`, we all know that no one actually reads docs. (Hey, thanks for taking the time to read this!)

No worries, though. You can run `paver generate_setup` to get a `setup.py` file that you can ship in your tarball. Then your users can run `python setup.py install` just like they're used to, and Paver will take over.

But people don't have Paver yet!

There are millions of Python installations that don't have Paver yet, but have Python and `distutils`. How can they run a Paver-based install?

Easy, you just run `paver minilib` and you will get a file called `paver-minilib.zip`. That file has enough of Paver to allow someone to install most projects. The Paver-generated `setup.py` knows to look for that file and use it if it sees it.

Worried about bloating your package? The `paver-minilib` is not large:

```
# <==
# sh('cd docs/samples/started/newway ; paver minilib',
#     insert_output=False)
# sh('ls -l docs/samples/started/newway/paver-minilib.zip')
# ==>
# <==end==>
```

Paver itself is bootstrapped with a generated `setup` file and a `paver-minilib`.

Hey! Didn't you just create more work for me?

You might have noticed that we now have three commands to run in order to get a proper distribution for The New Way. Well, you can actually run them all at once: `paver generate_setup minilib sdist`. That's not terrible, but it's also not great. You don't want to end up with a broken distribution just because you forgot one of the tasks.

By design, one of the easiest things to do in Paver is to extend the behavior of an existing "task", and that includes `distutils` commands. All we need to do is create a new `sdist` task in our `pavement.py`:

```
# <== include('started/newway/pavement.py', 'sdist')==>
# <==end==>
```

The `@task` decorator just tells Paver that this is a task and not just a function. The `@needs` decorator specifies other tasks that should run before this one. You can also use the `call_task(taskname)` function within your task if you wish. The function name determines the name of the task. The docstring is what shows up in Paver's help output.

With that task in our `pavement.py`, `paver sdist` is all it takes to build a source distribution after generating a `setup` file and `minilib`.

Note: If you are depending on distutils task (via `@needs`), you have to call `setup()` before task is defined. Under the hood, `setup` call installs `distutils/setupools` task and make them available, so do not make it conditional.

Tackling the Docs

Until the tools themselves provide tasks and functions that make creating pavements easier, Paver’s Standard Library will include a collection of modules that help out for commonly used tools. Sphinx is one package for which Paver has built-in support.

To use Paver’s Sphinx support, you need to have Sphinx installed and, in your `pavement.py`, import `paver.doctools`. Just performing the import will make the doctools-related tasks available. `paver help html` will tell us how to use the `html` command:

```
# <== sh('paver help paver.doctools.html')==>
# <==end==>
```

According to that, we’ll need to set the `builddir` setting, since we’re using a `builddir` called “`_build`”. Let’s add this to our `pavement.py`:

```
# <== include('started/newway/pavement.py', 'sphinx')==>
# <==end==>
```

And with that, `paver html` is now equivalent to `make html` using the Makefile that Sphinx gave us.

Getting rid of our docs shell script

You may remember that shell script we had for moving our generated docs to the right place:

```
# <== include('started/oldway/builddocs.sh')==>
# <==end==>
```

Ideally, we’d want this to happen whenever we generate the docs. We’ve already seen how to override tasks, so let’s try that out here:

```
# <== include('started/newway/pavement.py', 'html')==>
# <==end==>
```

There are a handful of interesting things in here. The equivalent of ‘`make html`’ is the `@needs('paver.doctools.html')`, since that’s the task we’re overriding.

Inside our task, we’re using “`path`”. This is a customized version of Jason Orendorff’s `path` module. All kinds of file and directory operations become super-simple using this module.

We start by deleting our destination directory, since we’ll be copying new generated files into that spot. Next, we look at the built docs directory that we’ll be moving:

```
# <== include('started/newway/pavement.py', 'html.builtdocs')==>
# <==end==>
```

One cool thing about `path` objects is that you can use the natural and comfortable ‘`/`’ operator to build up your paths.

The next thing we see here is the accessing of options. The `options` object is available to your tasks. It’s basically a dictionary that offers attribute-style access and can search for variables (which is why you can type `options.builddir`

instead of the longer `options.sphinx.builddir`). That property of options is also convenient for being able to share properties between sections.

And with that, we eliminate the shell script as a separate file.

Fixing another wart in The Old Way

In the documentation for The Old Way, we actually included the function body directly in the docs. But, we had to cut and paste it there. Sphinx does offer a way to include an external file in your documentation. Paver includes a better way.

There are a couple of parts to the documentation problem:

1. It's good to have your code in separate files from your docs so that the code can be complete, runnable and, above all, testable programs so that you can be sure that everything works.
2. You want your writing and the samples included with your writing to stand up as reasonable, coherent documents. Python's doctest style does not always lend itself to coherent documents.
3. It's nice to have the code sample that you're writing about included inline with the documents as you're writing them. It's easier to write when you can easily see what you're writing about.

#1 and #3 sound mutually exclusive, but they're not. Paver has a two part strategy to solve this problem. Let's look at part of the `index.rst` document file to see the first part:

```
# <== include("started/newway/docs/index.rst", "mainpart")==>
# <==end==>
```

In The New Way's `index.rst`, you can see the same mechanism being used that is used in this Getting Started guide. Paver includes Ned Batchelder's `Cog` package. `Cog` lets you drop snippets of Python into a file and have those snippets generate stuff that goes into the file. Unlike a template language, `Cog` is designed so that you can leave the markers in and regenerate as often as you need to. With a template language, you have the template and the finalized output, but not a file that has both.

So, as I'm writing this Getting Started document, I can glance up and see the `index.rst` contents right inline. You'll notice The # [[[cog part in there is calling an `include()` function. This is the second part offered by Paver. Paver lets you specify an "includedir" for use with `Cog`. This lets you include files relative to that directory. And, critically, it also lets you mark off sections of those files so that you can easily include just the part you want. In the example above, we're picking up the 'code' section of the `newway/thecode.py` file. Let's take a look at that file:

```
# <== sh("cat docs/samples/started/newway/newway/thecode.py") ==>
# <==end==>
```

Paver has a `Cog`-like syntax for defining named sections. So, you just use the `include` function with the relative filename and the section you want, and it will be included. Sections can even be nested (and you refer to nested sections using familiar dotted notation).

Bonus Deployment Example

pavements are just standard Python. The syntax for looping and things like that are just what you're used to. The options are standard Python so they can contain lists and other objects. Need to deploy to multiple hosts? Just put the hosts in the options and loop over them.

Let's say we want to deploy The New Way project's HTML files to a couple of servers. This is similar to what I do for Paver itself, though I only have one server. First, we'll set up some variables to use for our deploy task:


```
# <== include('started/newway/pavement.py', 'deployoptions')==>
# <==end==>
```

As you can see, we can put whatever kinds of objects we wish into the options. Now for the deploy task itself:

```
# <== include("started/newway/pavement.py", "deploy")==>
# <==end==>
```

You'll notice the new “cmdopts” decorator. Let's say that you have sensitive information like a password that you don't want to include in your pavement. You can easily make it a command line option for that task using cmdopts. `options.deploy.username` will be set to whatever the user enters on the command line.

It's also worth noting that when looking up options, Paver gives priority to options in a section with the same name as the task. So, `options.username` will prefer `options.deploy.username` even if there is a `username` in another section.

Our deploy task uses a simple for loop to run an `rsync` command for each host. Let's do a dry run providing a username to see what the commands will be:

```
# <== sh("cd docs/samples/started/newway; paver -n deploy -u kevin")==>
# <==end==>
```

Where to go from here

The first thing to do is to just get started using Paver. As you've seen above, it's easy to get Paver into your workflow, even with existing projects.

Use the `paver help` command.

If you really want more detail now, you'll want to read more about *pavement files* and the *Paver Standard Library*.

6.4 pavement.py in depth

Paver is meant to be a hybrid declarative/imperative system for getting stuff done. You declare things via the options in `pavement.py`. And, in fact, many projects can get away with nothing but options in `pavement.py`. Consider, for example, an early version of Paver's own pavement file:

```
from paver.easy import *
import paver.doctools

options(
    setup=dict(
        name='paver',
        version="0.3",
        description='Python build tool',
        author='Kevin Dangoor',
        author_email='dangoor+paver@gmail.com',
        #url='',
        packages=['paver'],
        package_data=setuptools.find_package_data("paver", package="paver",
                                                  only_in_packages=False),
        install_requires=[],
        test_suite='nose.collector',
        zip_safe=False,
        entry_points="""
        [console_scripts]
```

```
paver = paver.command:main
"""
),

sphinx=Bunch(
    builddir="build",
    sourcedir="source"
)

)

@task
@needs('paver.doctools.html')
def html():
    """Build Paver's documentation and install it into paver/docs"""
    builtdocs = path("docs") / options.sphinx.builddir / "html"
    destdir = path("paver") / "docs"
    destdir.rmtree()
    builtdocs.move(destdir)
```

This file has both declarative and imperative aspects. The options define enough information for distutils, setuptools and Sphinx to do their respective jobs. This would function just fine without requiring you to define any tasks.

However, for Paver's 'paverdoc' built-in task to work, we need Sphinx's generated HTML to end up inside of Paver's package tree. So, we override the html task to move the generated files.

6.4.1 Defining Tasks

Tasks are just simple functions. You designate a function as being a task by using the @task decorator.

You can also specify that a task depends on another task running first with the @needs decorator. A given task will only run once as a dependency for other tasks.

6.4.2 Manually Calling Tasks

Sometimes, you need to do something *before* running another task, so the @needs decorator doesn't quite do the job. Of course, tasks are just Python functions. So, you can just call the other task like a function!

6.4.3 How Task Names Work

Tasks have both a long name and a short name. The short name is just the name of the function. The long name is the fully qualified Python name for the function object.

For example, the Sphinx support includes a task function called "html". That task's long name is "paver.doctools.html".

If you `import paver.doctools` in your `pavement.py`, you'll be able to use either name to refer to that task.

Tasks that you define in your pavement are always available by their short names. Tasks defined elsewhere are available by their short names unless there is a conflict where two tasks are trying to use the same short name.

Tasks are always available unambiguously via their long names.

6.4.4 Task Parameters

Tasks don't have to take any parameters. However, Paver allows you to work in a fairly clean, globals-free manner(*). Generally speaking, the easiest way to work with paver is to just do `from paver.easy import *`, but if you don't like having so much in your namespace, you can have any attribute from `tasks.environment` sent into your function. For example:

```
@task
def my_task(options, info):
    # this task will get the options and the "info" logging function
    # sent in
    pass
```

(*): well, there *is* one global: `tasks.environment`.

6.4.5 Command Line Arguments

Tasks can specify that they accept command line arguments, via three other decorators. The `@consume_args` decorator tells Paver that *all* command line arguments following this task's name should be passed to the task. If you'd like specifying a number of consumed arguments, let use `@consume_nargs`. This later is similar by default to the previous, but also accept as an `int` argument the number of command line arguments the decorated task will consume. You can either look up the arguments in `options.args`, or just specify `args` as a parameter to your function. For example, Paver's "help" task is declared like this:

```
@task
@consume_args
def help(args, help_function):
    pass

@task
@consume_nargs(3)
def mytask(args):
    pass
```

The "args" parameter is just an attribute on `tasks.environment` (as is `help_function`), so it is passed in using the same rules described in the previous section.

New in version 1.1.0: `@consume_nargs` decorator superseeds `@consume_args`, and optionally accepts an `int` as argument: the number of command line argument the decorated task will consume.

More generally, you're not trying to consume all of the remainder of the command line but to just accept certain specific arguments. That's what the `cmdopts` decorator is for:

```
@task
@cmdopts([
    ('username=', 'u', 'Username to use when logging in to the servers')
])
def deploy(options):
    pass
```

`@cmdopts` takes a list of tuples, each with long option name, short option name and help text. If there's an "=" after the long option name, that means that the option takes a parameter. Otherwise, the option is assumed to be boolean. The command line options set in this manner are all added to the `options` under a namespace matching the name of the task. In the case above, `options.deploy.username` would be set if the user ran `paver deploy -u my-user-name`. Note that an equivalent command line would be `paver deploy.username=my-user-name deploy`

For fine-tuning, you may add `optparse.Option` instances:

```
@tasks.task
@tasks.cmdopts([
    make_option("-f", "--foo", help="foo")
])
def foo_instead_of_spam_and_eggs(options):
    pass
```

You may share `@cmdopts` between tasks. To do that and to avoid confusion, You have to add `share_with` argument:

```
@task
@cmdopts([
    ('username=', 'u', 'Username to use when logging in to the servers')
])
def deploy_to_linux(options):
    pass

@task
@needs(['deploy_to_linux'])
@cmdopts([
    ('username=', 'u', 'Username to use when logging in to the servers')
], share_with=['deploy_to_linux'])
def deploy(options):
    pass
```

For sharing, following must be fulfilled:

- Both long and short option names must be same
- `share_with` argument must be specified on top-level task

Otherwise, `PavementError` is raised.

You can combine both `@consume_args` and `@cmdopts` together:

```
@task
@cmdopts([
    ('username=', 'u', 'Username to use when logging in to the servers')
])
@consume_args
def exec(options):
    pass
```

- `paver exec -u root` will result in `options.username = 'root'`, `options.args = []`
- `paver exec -u root production` will result in `options.username = 'root'`, `options.args = ['production']`
- `paver exec production -u root` will result in `options.args = ['production', '-u', 'root']` with no `options.username` attribute.
- `paver exec -u root production -u other` will result in `options.username = 'root'`, `options.args = ['production', '-u', 'other']`

6.4.6 Hiding tasks

Some tasks may only exist as a shared dependency between other tasks and may not make sense to be called directly.

There is no way to provide that, however you can hide them from `paver help` to reduce noise. Just decorate function with `@no_help` decorator:

```
@task
@no_help
def hidden_dependency():
    pass
```

Of course, this should not be used usually. If task is not to be called at all, why not just make them a g'old function?

6.4.7 More complex dependencies

`@needs` might not cover all your needs. For example, depending on argument or environment, you might decide to call an appropriate task in the middle of another one.

There are two key options for fixing that:

1. `@might_call` decorator, which indicates that task might invoke `call_task` on one or more of the specified tasks. This allows you to provide command line options to task that might be called (it is inserted in dependency chain):

```
@task
@cmdopts([
    ('username=', 'u', 'Whom to greet')
], share_with=['deploy_to_linux'])
def say_hello(options):
    if not hasattr(options, "username"):
        print 'SPAM'
    else:
        print 'Hello, my dear user %s' % options.username

@task
@might_call('say_hello')
def greet_user(options):
    setup_environment()

    call_task('say_hello')

    do_cleanup()
```

2. Providing options and arguments to another tasks directly. Options are provided with final assigned value:

```
@task
@cmdopts([
    ('long-username=', 'u', 'Whom to greet')
], share_with=['deploy_to_linux'])
def say_hello(options):
    if not hasattr(options, "username"):
        print 'SPAM'
    else:
        print 'Hello, my dear user %s' % options.long_username

@task
def greet_user(options):
    call_task('say_hello', options={
```

```
'long_username' : 'Kitty'
})
```

Console arguments (args) should be passed as in console:

```
@task
@consume_args
def say_hello(args):
    print 'Hello to ALL the users: %s' % ', '.join(args)

@task
def greet_user(options):
    call_task('say_hello', args = [
        'Arthur Pewtey',
        'The Reverend Arthur Belling',
    ])
```

6.5 The Paver Standard Library

Paver includes several modules that will help you get your project scripting done more quickly.

6.5.1 distutils and setuptools (paver.setuputils)

Paver makes it very easy to use and extend Python's standard distribution mechanisms. The *Getting Started Guide* has a useful example of moving from distutils and setuptools to Paver. Integrates distutils/setuptools with Paver.

```
paver.setuputils.find_package_data (where='.', package="", exclude=('*.py', '*.pyc', '*~',
    '*.bak', '*.swp*'), exclude_directories=('.', 'CVS',
    '_darcs', './build', './dist', 'EGG-INFO', '*.egg-info'),
    only_in_packages=True, show_ignored=False)
```

Return a dictionary suitable for use in `package_data` in a distutils `setup.py` file.

The dictionary looks like:

```
{'package': [files]}
```

Where `files` is a list of all the files in that package that don't match anything in `exclude`.

If `only_in_packages` is true, then top-level directories that are not packages won't be included (but directories under packages will).

Directories matching any pattern in `exclude_directories` will be ignored; by default directories with leading `..`, `CVS`, and `_darcs` will be ignored.

If `show_ignored` is true, then all the files that aren't included in package data are shown on stderr (for debugging purposes).

Note patterns use wildcards, or can be exact paths (including leading `./`), and all searching is case-insensitive.

This function is by Ian Bicking.

```
paver.setuputils.install_distutils_tasks ()
    Makes distutils and setuptools commands available as Paver tasks.
```

```
paver.setuputils.setup (**kw)
```

Updates `options.setup` with the keyword arguments provided, and installs the `distutils` tasks for this pavement. You can use `paver.setuputils.setup` as a direct replacement for the `distutils.core.setup` or `setuptools.setup` in a traditional `setup.py`.

6.5.2 File Handling in Paver (`paver.path`)

Wrapper around `path.py` to add dry run support and other paver integration.

```
paver.path.pushd (*args, **kws)
```

A context manager for stepping into a directory and automatically coming back to the previous one. The original directory is returned. Usage is like this:

```
from paver.easy import *

@task
def my_task():
    with pushd('new/directory') as old_dir:
        ...do stuff...
```

6.5.3 Documentation Tools (`paver.doctools`)

Tasks and utility functions and classes for working with project documentation.

```
class paver.doctools.Includer (basedir, cog=None, include_markers=None)
```

Looks up `SectionedFiles` relative to the `basedir`.

When called with a filename and an optional section, the `Includer` will:

1. look up that file relative to the `basedir` in a cache
2. load it as a `SectionedFile` if it's not in the cache
3. return the whole file if section is `None`
4. return just the section desired if a section is requested

If a `cog` object is provided at initialization, the text will be output (via `cog's out`) rather than returned as a string.

You can pass in `include_markers` which is a dictionary that maps file extensions to the single line comment character for that file type. If there is an include marker available, then output like:

```
# section 'sectionname' from 'file.py'
```

There are some default include markers. If you don't pass in anything, no include markers will be displayed. If you pass in an empty dictionary, the default ones will be displayed.

```
class paver.doctools.SectionedFile (filename=None, from_string=None)
```

Loads a file into memory and keeps track of all of the sections found in the file. Sections are started with a line that looks like this:

```
[[[section SECTIONNAME]]]
```

Anything else can appear on the line outside of the brackets (so if you're in a source code file, you can put the section marker in a comment). The entire lines containing the section markers are not included when you request the text from the file.

An end of section marker looks like this:

```
[[[endsection]]]
```

Sections can be nested. If you do nest sections, you will use dotted notation to refer to the inner sections. For example, a “dessert” section within an “order” section would be referred to as “order.dessert”.

The SectionedFile provides dictionary-style access to the sections. If you have a SectionedFile named ‘sf’, sf[sectionname] will give you back a string of that section of the file, including any inner sections. There won’t be any section markers in that string.

You can get the text of the whole file via the all property (for example, sf.all).

Section names must be unique across the file, but inner section names are kept track of by the full dotted name. So you can have a “dessert” section that is contained within two different outer sections.

Ending a section without starting one or ending the file without ending a section will yield BuildFailures.

all

Property to get access to the whole file.

paver.doctools.cog (options)

Runs the cog code generator against the files matching your specification. By default, cog will run against any .rst files in your Sphinx document root. Full documentation for Cog is here:

<https://nedbatchelder.com/code/cog/>

In a nutshell, you put blocks in your file that look like this:

```
[[[cog cog.outl("Hi there!") ]]] [[end]]
```

Cog will replace the space between]]] and [[end]]] with the generated output. In this case, Hi there!

Here are the options available for the cog task. These are looked up in the ‘cog’ options section by default. The ‘sphinx’ option set is also searched.

basedir directory to look in for files to cog. If not set, ‘docroot’ is looked up.

pattern file glob to look for under basedir. By default, *.rst

includedir If you have external files to include in your documentation, setting includedir to the root of those files will put a paver.doctools.Includer in your Cog namespace as ‘include’. This lets you easily include files and sections of files. Here’s an example usage:

```
[[[cog include('filename_under_includedir.py', 'mysection')]]]
[[end]]
```

defines Dictionary of objects added to your Cog namespace. (can supersede ‘include’ and ‘sh’ defined by includedir.)

beginspec String used at the beginning of the code generation block. Default: [[[cog

endspec String used at the end of the code generation block. Default:]]]

endoutput String used at the end of the generated output Default: [[end]]]

delete_code Remove the generator code. Note that this will mean that the files that get changed cannot be changed again since the code will be gone. Default: False

include_markers Dictionary mapping file extensions to the single line comment marker for that file. There are some defaults. For example, ‘py’ maps to ‘#’. If there is a known include marker for a given file, then a comment will be displayed along the lines of:

```
# section ‘SECTIONNAME’ in file ‘foo.py’
```


If this option is not set, these lines will not be displayed at all. If this option is set to an empty dictionary, the default include markers will be displayed. You can also pass in your own extension -> include marker settings.

`paver.doctools.doc_clean()`

Clean (delete) the built docs. Specifically, this deletes the build directory under the docroot. See the html task for the options list.

`paver.doctools.html()`

Build HTML documentation using Sphinx. This uses the following options in a “sphinx” section of the options.

docroot the root under which Sphinx will be working. Default: docs

builddir directory under the docroot where the resulting files are put. default: .build

sourcedir directory under the docroot for the source files default: (empty string)

apidir directory under the sourcedir for the auto-generated API docs (empty = don’t create them) default: api

`paver.doctools.uncog(options)`

Remove the Cog generated code from files. Often, you will want to do this before committing code under source control, because you don’t generally want generated code in your version control system.

This takes the same options as the cog task. Look there for more information.

6.5.4 Miscellaneous Tasks (paver.misctasks)

These are some other tasks that are located in the paver.misctasks module. Miscellaneous tasks that don’t fit into one of the other groupings.

`paver.misctasks.generate_setup(options)`

Generates a setup.py file that uses paver behind the scenes. This setup.py file will look in the directory that the user is running it in for a paver-minilib.zip and will add that to sys.path if available. Otherwise, it will just assume that paver is available.

`paver.misctasks.minilib(options)`

Create a Paver mini library that contains enough for a simple pavement.py to be installed using a generated setup.py. This is a good temporary measure until more people have deployed paver. The output file is ‘paver-minilib.zip’ in the current directory.

Options:

versioned_name if set to True, paver version will be added into minilib’s filename (ie paver-minilib-1.1.0.zip) purpose is to avoid import error while using different versions of minilib with easy_install (default False)

extra_files list of other paver modules to include (don’t include the .py extension). By default, the following modules are included: defaults, path, release, setuputils, misctasks, options, tasks, easy

extra_packages list of unrelated packages to include. By default, Paver’s own dependencies are included. Package must be installed and importable

6.5.5 Virtualenv Support (paver.virtual)

Paver makes it easy to set up virtualenv environments for development and deployment. Virtualenv gives you a place to install Python packages and keep them separate from your main system’s Python installation.

6.5.6 Using virtualenv with tasks

You may specify which virtual environment should particular task use. Do this with `@virtualenv` decorator:

```
from paver.easy import task
from paver.virtual import virtualenv

@task
@virtualenv(dir="virtualenv")
def t1():
    import some_module_existing_only_in_virtualenv
```

6.5.7 paver.virtual Tasks

Tasks for managing virtualenv environments.

`paver.virtual.bootstrap()`

Creates a virtualenv bootstrap script. The script will create a bootstrap script that populates a virtualenv in the current directory. The environment will have paver, the packages of your choosing and will run the paver command of your choice.

This task looks in the virtualenv options for:

script_name name of the generated script

packages_to_install packages to install with pip/easy_install. The version of paver that you are using is included automatically. This should be a list of strings.

paver_command_line run this paver command line after installation (just the command line arguments, not the paver command itself).

dest_dir the destination directory for the virtual environment (defaults to '.')

no_site_packages don't give access to the global site-packages dir to the virtual environment (default; deprecated)

system_site_packages give access to the global site-packages dir to the virtual environment

unzip_setuptools unzip Setuptools when installing it (defaults to False)

distribute use Distribute instead of Setuptools. Set environment variable `VIRTUALENV_DISTRIBUTE` to make it the default.

index_url base URL of Python Package Index

trusted_host specify whether the given `index_url` is a trusted host to avoid deprecated warnings

no_index ignore package index (only looking at `find_links` URL(s) instead)

find_links additional URL(s) to search for packages. This should be a list of strings.

prefer_easy_install prefer `easy_install` to `pip` for package installation if both are installed (defaults to False)

`paver.virtual.virtualenv(dir)`

Run decorated task in specified virtual environment.

6.5.8 Using with Subversion (paver.svn)

Convenience functions for working with svn.

This module does not include any tasks, only functions.

At this point, these functions do not use any kind of library. They require the svn binary on the path.

`paver.svn.checkout` (*url*, *dest*, *revision=""*)

Checks out the specified URL to the given destination.

`paver.svn.checkup` (*url*, *dest*, *revision=""*)

Does a checkout or update, depending on whether the destination exists and is up to date (if a revision is passed in). Returns true if a checkout or update was performed. False otherwise.

`paver.svn.export` (*url*, *dest*, *revision=""*)

Exports the specified URL to the given destination.

`paver.svn.info` (*path=""*)

Retrieves the svn info for the path and returns a dictionary of the values. Names are normalized to lower case with spaces converted to underscores.

`paver.svn.update` (*path=""*, *revision=""*)

Run an svn update on the given path.

6.5.9 Using with Bazaar-NG (bzd) (paver.bzd)

6.5.10 Using with Mercurial (hg) (paver.hg)

Convenience functions for working with mercurial

This module does not include any tasks, only functions.

At this point, these functions do not use any kind of library. They require the hg binary on the PATH.

`paver.hg.branches` (*repo_path*, *closed=False*)

List branches for the target repository.

Parameters: *repo_path* (string): The local path to a mercurial repository. *closed=False* (bool): Whether to include closed branches in the

branch list.

Returns: A python tuple. The first item of the tuple is the current branch. The second item of the tuple is a list of the branches

`paver.hg.clone` (*url*, *dest_folder*, *rev=None*)

Clone a mercurial repository.

Parameters:

url (string): The path to clone the repository from. Could be local or remote.

dest_folder (string): The local folder where the repository will be cloned.

rev=None (string or None): If specified, the revision to clone to. If omitted or *None*, all changes will be cloned.

Returns: None

`paver.hg.latest_tag` (*repo_path*, *relative_to='tip'*)

Get the latest tag from a mercurial repository.

Parameters: *repo_path* (string): The local path to a mercurial repository. *relative_to='tip'* (string): If provided, the revision to use as

a reference. Defaults to 'tip'.

Returns: The string name of the latest tag.

`paver.hg.pull` (*repo_path*, *rev=None*, *url=None*)

Pull changes into a mercurial repository.

Parameters: `repo_path` (string): The local path to a mercurial repository. `rev=None` (string or None): If specified, the revision to pull to.

If omitted or *None*, all changes will be pulled.

`url=None` (string or None): If specified, the repository to pull from. If omitted or *None*, the default location of the repository will be used.

Returns: None

`paver.hg.update` (*repo_path*, *rev='tip'*, *clean=False*)

Update a mercurial repository to a revision.

Parameters: `repo_path` (string): The local path to a mercurial repository. `rev='tip'` (string): If provided, the revision to update to. If

omitted, 'tip' will be used.

`clean=False` (bool): If *True*, the update will discard uncommitted changes.

Returns: None

6.5.11 SSH Remote Access Support (`paver.ssh`)

Functions for accessing remote hosts.

At present, these are implemented by calling `ssh`'s command line programs.

`paver.ssh.scp` (*source*, *dest*)

Copy the source file to the destination.

6.6 Paver Command Line

Paver does sophisticated command line parsing globally and for each task:

```
paver [-q] [-n] [-v] [-f pavement] [-h] [option.name=key] [taskname] [taskoptions]_
↪ [taskname...]
```

The command line options are:

- q** quiet... don't display much info (info and debug messages are not shown)
- n** dry run... don't actually run destructive commands
- v** verbose... display debug level output
- h** display the command line options and list of available tasks. Note that `-h` can be provided for any task to display the command line options and detailed help for that task.
- f <pavement>** use a different file than "pavement.py"

If you run `paver` without a task, it will only run the "auto" task, if there is one. Otherwise, Paver will do nothing.

`paver help` is the equivalent of `paver -h`, and `paver help taskname` is the equivalent of `paver taskname -h`.

You can set build options via the command line by providing `optionname=value`. The option names can be in dotted notation, so `foo.bar.baz=something` will set `options['foo']['bar']['baz'] = 'something'` in the options. If you need to enter a value with multiple words, put quotes around the part with the space.

Important and useful: Options are set at the point in which they appear in the command line. That means that you can set an option before one task and then set it to another value for the next task.

6.7 Tips and Tricks

6.7.1 Using a Config File For Settings

Many people like to have their configuration metadata available in a *data file*, rather than a Python program. This is easy to do with Paver:

```
from paver.easy import *

@task
def auto():
    config_data = (read config data using config parser of choice)
    # assuming config_data is a dictionary
    options.update(config_data)
```

The auto task is automatically run when the pavement is launched. You can use Python's standard `ConfigParser` module, if you'd like to store the information in an `.ini` file.

6.8 Articles about Paver

- [Converting from Make to Paver](#) - Doug Hellman's look at how he moved from Make to Paver for Python Module of the Week.
- [Writing Technical Documentation with Sphinx, Paver and Cog](#) - Doug Hellman's in-depth article about how he produces the Python Module of the Week series.
- [Initial announcement](#) and background about the project
- [Release 0.7 announcement](#)
- [Paver 1.0a1 announcement](#) (the ill-fated release that was recalled)

6.9 Complete API Reference

The following is a complete API reference generated from source, as a companion to the "Paver Standard Library" chapter.

6.10 Paver Changelog

6.10.1 1.3.4 (Dec 31, 2017)

- Minilib can now be include arbitrary packages (#28)

- Six is now bundled in minilib (#193)
- `install_requires` is now not overridden and six is properly declared as a dependency (#194)
- Regression: Installation using `setup.py install` with minilib will *not* install six since it will be recognised as a dependency from minilib (#193)

6.10.2 1.3.3 (Dec 29, 2017)

- Properly exclude cache files from release

6.10.3 1.3.2 (Dec 28, 2017)

- Properly specify six in release dependencies

6.10.4 1.3.1 (Dec 28, 2017)

- Same as 1.3.1, but with properly bumped versions in source code
- Releases are now done from Travis CI

6.10.5 1.3.0 (Dec 28, 2017, tagged, but not released)

- ***Removed support for Python 2.6, 3.2, 3.3 and Jython 2.6*** (#179)
- Unvendor six (#180)
- https everything (#181)
- Mercurial convenience commands (#159)
- Add support for trusted hosts (#146)
- Minilib can now be directly executed (#145)
- Fix task grouping (#158)

6.10.6 1.2.4 (February 23, 2015)

- Make path comparison better (github issue #78)
- Add `last_tag` task
- six upgraded to 1.6.1

6.10.7 1.2.3 (August 10, 2014)

- ***Removed support for Python 2.5***. 2.6 is deprecated and will be removed in next release.
- Fixed `shell.py` missing from minilib (github issue #122)
- Added `env` keyword to `sh`. (github issue #132)
- When both `@cmdopts` and `@consume_nargs` are used, the options before the

args are parsed by the task's parser and given to it (github issue #126) * Support list and tuple as `sh` argument (github issue #92)

6.10.8 1.2.2 (January 12, 2014)

- Fixed `version.py` missing from minilib (github issue #112)

6.10.9 1.2.1 (June 2, 2013)

- Fixed most of the regressions from 1.2:
- documentation was missing from tarball (github issue #95)
- `path.push` missing in `paver.easy` (github issue #97, thanks to leonhandreke)
- broken backward compatibility with python2.5 and custom tasks (github issue #94)
- Variety of Python 3 problems (thanks to Afrever)
- Ignore non-system-default characters when `sh()`ing command with bad output

6.10.10 1.2 (February 24, 2013)

- ***Python 3 support***, thanks to @rkuppe
- pypy support now tested on Travis
- `pip` now preferred over `easy_install` (github issue #81, thanks to pmcncr)
- `virtual.bootstrap` enhancements: support for `find-links`, `index-url`, `system-site-packages` and `distribute` options (github issue #79, thanks to pmcncr)
- new `tasks.consume_nargs()` decorator, similar to `tasks.consume_args()` but accepting an argument: the number of arguments that the decorated function will consume. If no argument is passed to `consume_nargs` decorator, all comand-line arguments will be consumed.

6.10.11 1.1.1 (August 25, 2012)

- `path.py` fix for Jython compatibility (github issue #70, thanks to Arfrever)
- bundled cog updated to version 2.2 for Jython compatibility
- fixes regression for `setuptools` intallation (i.e. using `-root` parameter, github issue #71, thanks to Afrever for the report and yedpodtrzitko for the fix)
- basic jython compatibility tested

6.10.12 1.1.0 (July 30, 2012)

- Minilib is now imported only if full paver is not available (github issue #13)
- Option instances may now be passed to `@cmdopts` (github issues #41 and #42, thanks to David Cramer)
- `--propagate-traceback` option for debugging `BuildFailure`'s (github issue #43)
- Fix misleading error message when non-task is passed to `@needs` (github issue #37)
- `@no_help` to provide a way to hide task from `paver help` (github issue #36)
- `@might_call` for more complex dependencies (see docs, not only github issue #16)
- bundled `path.py` upgraded to patched version 2.2.2 (github issue #15)

- correctly handle dependencies in `install_requires` directive for `setup.py install` command (github issue #49)
- fix creating virtualenv (github issue #44)
- fix virtualenv example in docs (github issue #48)
- `path.rename()` do not call `rename` twice (github issue #47)
- updated `path.py` to resolve issues with bounding os functions with CPython 2.7.3 (github issue #59, thanks to Pedro Romano)
- minimal version of python raised to Python 2.5 (github issue #52)
- always import + do not allow to overwrite basic tasks (eg. `help`) (github issue #58)
- if virtualenv is not available, `PaverImportError` is raised instead of generic `Exception` (github issue #30)

6.10.13 1.0.5 (October 21, 2011)

- Ability to share command line options between tasks (github issue #7)
- Flush after print (github issue #17, thanks to Honza Kral)
- Minilib is now compatible with `zipimport` (github issue #19, thanks to Piet Delport)
- Auto task is now properly not called when target task is decorated with `no_auto` (github issue #4)

6.10.14 1.0.4 (January 16, 2011)

- Fixed md5 deprecation warnings in the bundled cog (thanks to jszakmeister, issue #56)
- Project moved to github
- Fixed problems with negative command-line options for `distutils` (thanks to Nao Nakashima for bugreport, github issue #2)
- Japanese translation moved to <https://github.com/paver/paver-docs-jp>
- Tasks take `cmdopts` even from grandparents (thanks to aureliano, github issue #4)
- Task description is taken from the first sentence, where the end of the sentence is dot followed by alphanumeric character (google code bug #44). Description is also stripped now.

6.10.15 1.0.3 (June 1, 2010)

- Fixed deadlock problem when there's a lot of output from a subprocess (thanks to Jeremy Rossi)
- Fixed unit tests (thanks to Elias Alma)

6.10.16 1.0.2 (March 8, 2010)

- FIXED A command that outputs to `stderr` containing formatting directives (`%s`) or something that looks like one would cause an error. Thanks to `disturbyte` for the patch.
- Tasks can take normal keyword arguments
- Returns exit code 1 if any tasks fail
- `stderr` is no longer swallowed up by `sh()` (issue #37, thanks to Marc Sibson for the patch)

6.10.17 1.0.1 (May 4, 2009)

This release was made possible by Adam Lowry who helped improve the code and reviewed committed many of the patches.

- Fixed sending nonpositional arguments first with `consume_args` (issue #31).
- Fixed use of `setuputils` without defining `options.setup` (issue #24).
- Python 2.4 compatibility fixes (issue #28)
- `sh()` failures are logged to `stderr`.
- `sh()` accepts a `cwd` keyword argument (issue #29).
- `virtualenv` bootstrap generation accepts `no_site_packages`, `unzip_setuptools`, and destination directory arguments in options.
- `Distutils` config files were being ignored (issue #36) (thanks to Matthew Scott for the patch)
- The exit code was 0 whenever the first task passes, even if later tasks fail (issue #35) (thanks to Matt for the patch)
- Tasks can take normal keyword arguments (issue #33) (thanks to Chris Burroughs for the patch with test!)

6.10.18 1.0 (March 22, 2009)

- If there is a task called “default”, it is run if Paver is run with no tasks listed on the command line.
- The auto task is run, even if no tasks are specified on the command line.
- `distutils`’ log output is now routed through Paver’s logging functions, which means that the output is now displayed once more (and is controlled via Paver’s command line arguments.)
- The `paver.setuputils.setup` function will automatically call `install_distutils_tasks`. This makes it a very convenient way to upgrade from `distutils/setuptools` to Paver.
- Nicer looking error when you run Paver with an unknown task name.
- fix the `md5` deprecation warning in `paver.path` for real (forgot to delete the offending import). Also fixed an import loop when you try to import `paver.path`.
- Improved docs for 1.0
- Paver now requires Sphinx 0.6 for the docs. In Paver’s `conf.py` for Sphinx, there is an `autodoc Documenter` for handling Paver Tasks properly.

6.10.19 1.0b1 (March 13, 2009)

- added `call_task` to environment and `paver.easy`, so it should be easy to call `distutils` tasks, for example. (Normally, with Paver 1.0, you just call Paver tasks like normal functions.)
- added `setup()` function to `paver.setuputils` that is a shortcut for setting options in `options.setup`. This means that you switch from `distutils` to Paver just by renaming the file and changing the import.
- the `-h` command line argument and “help” task have been unified. You’ll get the same output regardless of which one you use.
- the auto task is no longer called when you run the help task (issue #21). As part of this, a new “no_auto” decorator has been created so that any task can be marked as not requiring the auto behavior.
- `consume_args` and `PavementError` are now included in `paver.easy` (thanks to Marc Sibson)

- more methods in `paver.path` now check for existence or lack thereof and won't fail as a result. (`mkdir` and `makedirs` both check that the directory does not exist, `rmdir` and `rmtree` check to be sure that it does.) This is because the goal is ultimately to create or remove something... `paver` just makes sure that it either exists or doesn't.
- fix md5 deprecation warning in `paver.path` (issue #22)

6.10.20 1.0a4 (March 6, 2009)

- `call_pavement` would raise an exception if the pavement being called is in the current directory
- the new `paver.path25` module was missing from the `paver-minilib.zip`

6.10.21 1.0a3 (March 6, 2009)

- Added automatic running of “auto” task. If there's a task with the name “auto”, it is run automatically. Using this mechanism, you can write code that sets up the options any way you wish, and without using globals at all (because the auto task can be given options as a parameter).
- When generating `egg_info` running “paver”, the full path to the Paver script was getting included in `egg_info/SOURCES.txt`. This causes installation problems on Windows, at the very least. Paver will now instead place the pavement that is being run in there. This likely has the beneficial side effect of not requiring a `MANIFEST.in` file just to include the pavement.
- the options help provided via the `cmdopts` decorator now appears
- pavements can now refer to `__file__` to get their own filename. You can also just declare `pavement_file` as an argument to your task function, if you wish.
- `call_pavement` now switches directories to the location of the pavement and then switches back when returning
- if you try to run a function as a task, you'll now get a more appropriate and descriptive `BuildFailure`, rather than an `AttributeError`
- `paver` can now again run tasks even when there is no pavement present. any task accessible via `paver.easy` (which now also includes `misctasks`) will work.
- added the `pushd` context manager (Python 2.5+). This will switch into another directory on the way in and then change back to the old directory on the way out. Suggested by Steve Howe, with the additional suggestion from Juergen Hermann to return the old directory:

```
with pushd('newdirectory') as olddirectory:
    ...do something...
```

6.10.22 1.0a2 (February 26, 2009)

- The bug that caused 1.0a1 to be recalled (`distutils` command options) has been fixed thanks to Greg Thornton.
- If you provide an invalid long task name, you will no longer get an `AttributeError`. Thanks to Marc Sibson.
- If a task has an uncaught exception, the debug-level output is displayed *and* Paver will exit with a return code of 1. No further tasks are executed. Thanks to Marc Sibson.
- The version number is no longer displayed, so that you can reasonably pipe the output elsewhere. A new `-version` option will display the version as before.
- Eliminate `DeprecationWarnings` in `paver.ssh` and `paver.svn`. Thanks to Marc Sibson.

- The html task will always be defined now when you import `paver.doctools` but will yield a `BuildFailure` if Sphinx is not installed. Hopefully this will lead to clearer errors for people. Thanks to Marc Sibson.
- The Getting Started Guide has been improved for 1.0. Additionally, the “newway” sample now has a MANIFEST.in which provides useful knowledge for people.

6.10.23 1.0a1 (January 28, 2009)

(note: 1.0a1 was recalled because it was unable to properly handle `distutils` command line options.)

- **COMPATIBILITY BREAK:** `paver.misctasks` is no longer imported by default, even when using `paver.easy`
- **DEPRECATIONS:** `paver.runtime` and `paver.defaults` have been deprecated. Watch the warnings for info on how to change to the new `paver.easy` module.
- **COMPATIBILITY WARNING:** By default, the `sh()` function will now raise a `BuildFailure` exception if the return code of the process is non-zero. Passing `ignore_error=True` will switch back to the previous behavior. Thanks to Marc Sibson.
- There is a new `call_pavement` function (automatically imported with `from paver.easy import *`) that can call another pavement file. The new pavement gets its own environment/options but runs in the same process.
- You can now specify an alternate file to run rather than “`pavement.py`” using the `-f` or `-file` global option. Thanks to Marc Sibson.
- Regardless of logging level, output for a task is captured. If there is a `BuildFailure`, then that captured output is displayed.
- The new `paver.tasks` module encapsulates everything needed for running tasks in a file. The `distutils` ties have been reduced.
- `@needs` now accepts a list of requirements in the form `@needs('task1', 'task2')` (passing in a list still works as well)
- Added `paver.bzr` (support for Bazaar-NG related operations), courtesy of Bryan Forbes.
- The `error()` function is now exported, for logging of errors (thanks to Marc Sibson)
- Added handy `paver.svn.export` function for exporting an `svn` repository revision (thanks to Marc Sibson)
- The “`scripts`” directory has been renamed “`distutils_scripts`” to avoid name collision on Windows.

6.10.24 0.8.1 (June 2, 2008)

- Fix bug in `minilib` on Windows (error in `rmtree`). Also simplifies the `minilib` implementation. Patch from Juergen Hermann.
- Fix bug in `virtualenv` bootstrap generation (patches from Michael Greene and Juergen Hermann. Michael Greene’s is the one that was applied.)

6.10.25 0.8 (May 19, 2008)

- Installation on Windows was broken due to a `/` at the end of the `/paver/tests` path in `MANIFEST.in`
- Options can now be set on the command line using the syntax `option.name=value`. Options are set at the point in which they appear on the command line, so you can set one value before `task1` and then another value before `task2`.

- Option ordering can now take an explicit dictionary or Bunch added to the ordering. This allows you to put in new options without changing the global options dictionary and more closely resembles how options would be looked up in a buildout.
- `call_task` now supports an optional “options” argument that allows you to pass in a dictionary or Bunch that is added to the front of the option search ordering.

6.10.26 0.7.3 (May 16, 2008)

- Added `include_markers` parameter to the `paver.doctools.Includer` to display a nice comment with the name of the file and section. This can look more attractive than the raw cog. By default, this is turned off. Set `options.cog.include_markers` to an empty dictionary, and the default include markers will be used.
- Added `options.cog.delete_code` to remove the generator code when cogging. Default: `false`
- Paver 0.7.2 could not be installed by `zc.buildout` on the Mac due to a problem with the `py2app` command under that environment.
- cog and tests were missing from shipped distributions (bug 229324, fixed with a patch from Krys Wilken.)
- Added `svn.checkup` function that does a checkout or update. This is like an `svn:externals` that’s a bit more readable and easier to control, in my opinion.

6.10.27 0.7.2 (May 8, 2008)

- Fixed Python 2.4 compatibility. The `paver-minilib.zip` file contained 2.5 .pyc files. .pyc files are not compatible between major Python versions. The new version contains .py files.

6.10.28 0.7.1 (May 8, 2008)

- 0.7 had a broken `paver-minilib.zip` (missing `misctasks.py`, which is now part of the standard `minilib`)

6.10.29 0.7 (May 7, 2008)

Breaking changes:

- “targets” have become “tasks”, because that name is a clearer description.
- `paver.sphinxdoc` has been renamed `paver.doctools`

New features and changes:

- `runtime.OPTIONS` is gone now. The old voodoo surrounding the `options()` function has been replaced with a distinctly non-magical `__call__ = update` in the `Namespace` class.
- `distutils.core.setup` is now the command line driver
- `distutils/setuputils` commands can be seamlessly intermingled with `Tasks`
- `tasks` can have command line settable options via the `cmdopts` decorator. Additionally, they can use the `consume_args` decorator to collect up all command line arguments that come after the task name.
- Two new tasks: `cog` and `uncog`. These run Ned Batchelder’s Cog code generator (included in the Paver package), by default against your Sphinx documentation. The idea is that you can keep your code samples in separate files (with unit tests and all) and incorporate them into your documentation files. Unlike the Sphinx include directives, using Cog lets you work on your documentation with the code samples in place.

- `paver.doctools.SectionedFile` provides a convenient way to mark off sections of a file, usually for documentation purposes, so that those sections can be included in another documentation file.
- `paver.doctools.Includer` knows how to look up `SectionedFiles` underneath a directory and to cache their sections.
- options are now a “Namespace” object that will search the sections for values. By default, the namespace is searched starting with top-level items (preserving current behavior) followed by a section named the same as the task, followed by all of the other sections. The order can be changed by calling `options.order`.
- option values that are callable will be called and that value returned. This is a simple way to provide lazy evaluation of options.
- Added `minilib` task that creates a `paver-minilib.zip` file that can be used to distribute programs that use Paver for their builds so that `setup.py` will run even without Paver fully installed.
- Added `generate_setup` task that creates a `setup.py` file that will actually run Paver. This will detect `paver-minilib.zip` if it’s present.
- The “help” task has been greatly improved to provide a clearer picture of the tasks, options and commands available.
- Add the ability to create `virtualenv` bootstrap scripts
- The “help” property on tasks has changed to “description”
- output is now directed through `distutils.log`
- Ever improving docs, including a new Getting Started guide.
- Changes to Paver’s bootstrap setup so that Paver no longer uses `distutils` for its bootstrapping.

There were no versions 0.5 and 0.6.

6.10.30 0.4 (April 22, 2008)

- First public release.
- Removes `setuptools` dependency
- More docs
- Paver can now be run even without a `pavement.py` file for commands like `help` and `paverdocs`

6.11 Credits

- Kevin Dangoor is the original author, designer and coder.
- The Paver project has gotten patches from Juergen Hermann, Marc Sibson, Greg Thornton, Michael Greene, Krys Wilken, Bryan Forbes and Ryan Wilcox.
- Ned Batchelder’s “cog” package is included for handling inclusion of files into the docs.
- Ian Bicking provided a lot of great input prior to Paver’s initial release pointers to code that he’s been using both at Open Planning and prior to working there.
- Ian is also the original author of the `paver.setuputils.find_package_data` function.
- Jason Orendorff wrote the original `path.py` module that `paver.path` is based upon.
- Michael Foord’s `Mock` module is included to assist in testing.

- Ian Bicking, Jim Fulton, Philip Eby and Georg Brandl lead the various projects that the Paver Standard Library takes advantage of. Without those projects, Paver users would have to do a lot more work.
- Though it seems almost too obvious for a Python project, Guido van Rossum deserves credit for making and steering a language that is so flexible and clean for such a wide variety of tasks.

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

p

`paver.doctools`, 27
`paver.hg`, 31
`paver.misctasks`, 29
`paver.path`, 27
`paver.setuputils`, 26
`paver.ssh`, 32
`paver.svn`, 30
`paver.virtual`, 30

A

all (paver.doctools.SectionedFile attribute), 28

B

bootstrap() (in module paver.virtual), 30

branches() (in module paver.hg), 31

C

checkout() (in module paver.svn), 31

checkup() (in module paver.svn), 31

clone() (in module paver.hg), 31

cog() (in module paver.doctools), 28

D

doc_clean() (in module paver.doctools), 29

E

export() (in module paver.svn), 31

F

find_package_data() (in module paver.setuputils), 26

G

generate_setup() (in module paver.misctasks), 29

H

html() (in module paver.doctools), 29

I

Includer (class in paver.doctools), 27

info() (in module paver.svn), 31

install_distutils_tasks() (in module paver.setuputils), 26

L

latest_tag() (in module paver.hg), 31

M

minilib() (in module paver.misctasks), 29

P

paver.doctools (module), 27

paver.hg (module), 31

paver.misctasks (module), 29

paver.path (module), 27

paver.setuputils (module), 26

paver.ssh (module), 32

paver.svn (module), 30

paver.virtual (module), 30

pull() (in module paver.hg), 31

pushd() (in module paver.path), 27

S

scp() (in module paver.ssh), 32

SectionedFile (class in paver.doctools), 27

setup() (in module paver.setuputils), 26

U

uncog() (in module paver.doctools), 29

update() (in module paver.hg), 32

update() (in module paver.svn), 31

V

virtualenv() (in module paver.virtual), 30