

Where do the Python unit tests go?

[closed]

Asked 16 years, 3 months ago Modified 2 years, 11 months ago

Viewed 142k times



577



Closed. This question is [opinion-based](#). It is not currently accepting answers.



Want to improve this question? Update the question so it can be answered with facts and citations by [editing this post](#).

Closed 3 years ago.

[Improve this question](#)

If you're writing a library, or an app, where do the unit test files go?

It's nice to separate the test files from the main app code, but it's awkward to put them into a "tests" subdirectory inside of the app root directory, because it makes it harder to import the modules that you'll be testing.

Is there a best practice here?

python

unit-testing

code-organization

Share

Improve this question

Follow

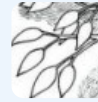
edited Mar 2, 2010 at 12:33



Chen Levy

16.3k ● 18 ● 78 ● 94

asked Sep 14, 2008 at 5:41



readonly

355k ● 109 ● 206 ● 206

18 Answers

Sorted by:

Highest score (default)



259

For a file `module.py`, the unit test should normally be called `test_module.py`, following Pythonic naming conventions.



There are several commonly accepted places to put `test_module.py`:



1. In the same directory as `module.py`.
2. In `../tests/test_module.py` (at the same level as the code directory).
3. In `tests/test_module.py` (one level under the code directory).

I prefer #1 for its simplicity of finding the tests and importing them. Whatever build system you're using can easily be configured to run files starting with `test_`. Actually, the [default unittest pattern used for test discovery is `test*.py`](#).

Share Improve this answer

Follow

edited Jul 5, 2015 at 11:12



Cristian Ciupitu

20.8k ● 7 ● 54 ● 79

answered Sep 15, 2008 at 12:52



user6868

8,938 ● 2 ● 19 ● 3

12 The [load_tests_protocol](#) searches for files named test*.py by default. Also, [this top Google result](#) and [this unittest documentation](#) both use the test_module.py format.
– [dfarrell07](#) Aug 13, 2013 at 21:24 ✎

8 Using foo_test.py can save a keystroke or two when using tab-completion since you don't have a bunch of files starting with 'test_'. – [juniper-](#) Aug 26, 2013 at 17:10

13 @juniper, following your thoughts module_test would appear in auto completion when you are coding. That can be confusing or annoying. – [Igor Medeiros](#) Oct 11, 2013 at 4:21

23 When deploying code, we don't want to deploy the tests to our production locations. So, having them in one directory './tests/test_blah.py' is easy to yank when we do deployments. ALSO, some tests take sample data files, and having those in a test directory is vital lest we deploy test data. – [Kevin J. Rice](#) May 4, 2015 at 14:10

3 @KevinJ.Rice Shouldn't you be testing that the code works on your production locations? – [endolith](#) Jun 18, 2017 at 15:00



Only 1 test file



If there has only 1 test files, putting it in a top-level directory is recommended:



```
module/  
  lib/  
    __init__.py  
    module.py  
    test.py
```

Run the test in CLI

```
python test.py
```

Many test files

If has many test files, put it in a `tests` folder:

```
module/  
  lib/  
    __init__.py  
    module.py  
  tests/  
    test_module.py  
    test_module_function.py
```

```
# test_module.py  
  
import unittest  
from lib import module  
  
class TestModule(unittest.TestCase):  
    def test_module(self):  
        pass
```

```
if __name__ == '__main__':  
    unittest.main()
```

Run the test in CLI

```
# In top-level /module/ folder  
python -m tests.test_module  
python -m tests.test_module_function
```

Use [unittest discovery](#)

`unittest discovery` will find all test in package folder.

Create a `__init__.py` in `tests/` folder

```
module/  
  lib/  
    __init__.py  
    module.py  
  tests/  
    __init__.py  
    test_module.py  
    test_module_function.py
```

Run the test in CLI

```
# In top-level /module/ folder  
  
# -s, --start-directory (default current directory)  
# -p, --pattern (default test*.py)  
  
python -m unittest discover
```

Reference

- `pytest` [Good Practices](#) for test layout
- `unittest`

Unit test framework

- [nose](#)
- [nose2](#)
- [pytest](#)

Share Improve this answer

edited Jul 23, 2019 at 4:59

Follow


answered Apr 30, 2014 at 10:58



[Steely Wing](#)

17.5k ● 9 ● 62 ● 55

2 I have the same structure that you have outlined under "Many Test Files" but when I try `from lib import module` I get an error: `ImportError: No module named 'lib'`
– [Shervin Rad](#) Jan 6, 2022 at 17:52

2 @ShervinRad You need run in the `module` folder, not `lib` or `tests` folder, python use the relative path to import.
– [Steely Wing](#) Jan 7, 2022 at 7:20 



52



A common practice is to put the tests directory in the same parent directory as your module/package. So if your module was called `foo.py` your directory layout would look like:

```
parent_dir/  
  foo.py  
  tests/
```

Of course there is no one way of doing it. You could also make a tests subdirectory and import the module using [absolute import](#).

Wherever you put your tests, I would recommend you use [nose](#) to run them. [Nose](#) searches through your directories for tests. This way, you can put tests wherever they make the most sense organizationally.

Share Improve this answer

Follow

edited Jan 9, 2015 at 5:31



user559633

answered Sep 14, 2008 at 6:46



Cristian

43.9k ● 26 ● 89 ● 99

18 I'd like to do this but I can't make it work. To run the tests, I'm in the `parent_dir`, and type: `"python tests\foo_test.py"`, and in `foo_test.py`: `"from ..foo import this, that, theother"` That fails with: `"ValueError: Attempted relative import in non-package"` Both `parent_dir` and `tests` have an `__init__.py` in them, so I'm not sure why they aren't packages. I suspect it's because the top-level script you run from the command line cannot be

considered (part of) a package, even if it's in a dir with an **init.py**. So how do I run the tests? I'm working on Windows today, bless my cotton socks. – [Jonathan Hartley](#) May 7, 2009 at 16:35

- 4 The best way-- I've found-- to run unit tests is to install your library/program then run unit tests with nose. I would recommend virtualenv and virtualenvwrapper to make this a lot easier. – [Cristian](#) May 16, 2009 at 9:27
-

@Tartley - you need a **init.py** file in your 'tests' directory for absolution imports to work. I have this method working with nose, so I'm not sure why you are having trouble. – [cmcginty](#) Jun 27, 2009 at 1:42

- 4 Thanks Casey - but I do have an **init.py** file in all relevant directories. I don't know what I do wrong, but I have this problem on all my Python projects and I can't understand why nobody else does. Oh deary. – [Jonathan Hartley](#) Jul 22, 2009 at 8:23
-

- 9 One solution for my problem, with Python2.6 or newer, us to run the tests from the root of your project using: `python -m project.package.tests.module_tests` (instead of `python project/package/tests/module_tests.py`). This puts the test module's directory on the path, so the tests can then do a relative import to their parent directory to get the module-under-test. – [Jonathan Hartley](#) Jul 22, 2009 at 8:25
-



32



We had the very same question when writing Pythoscope (<https://pypi.org/project/pythoscope/>), which generates unit tests for Python programs. We polled people on the testing in python list before we chose a directory, there were many different opinions. In the end we chose to put a "tests" directory in the same directory as the source



code. In that directory we generate a test file for each module in the parent directory.

Share Improve this answer

edited Mar 18, 2019 at 22:58

Follow



CivFan

15k ● 11 ● 47 ● 66

answered May 2, 2009 at 17:08



Paul Hildebrandt

2,754 ● 28 ● 26



28



Every once in a while I find myself checking out the topic of test placement, and every time the majority recommends a separate folder structure beside the library code, but I find that every time the arguments are the same and are not that convincing. I end up putting my test modules somewhere beside the core modules.

The main reason for doing this is: **refactoring**.

When I move things around I do want test modules to move with the code; it's easy to lose tests if they are in a separate tree. Let's be honest, sooner or later you end up with a totally different folder structure, like [django](#), [flask](#) and many others. Which is fine if you don't care.

The main question you should ask yourself is this:

Am I writing:

- a) reusable library or

- b) building a project than bundles together some semi-separated modules?

If a:

A separate folder and the extra effort to maintain its structure may be better suited. No one will complain about your tests getting *deployed to production*.

But it's also just as easy to exclude tests from being distributed when they are mixed with the core folders; [put this in the setup.py](#):

```
find_packages("src", exclude=["*.tests", "*.tests.*",
```

If b:

You may wish — as every one of us do — that you are writing reusable libraries, but most of the time their life is tied to the life of the project. Ability to easily maintain your project should be a priority.

Then if you did a good job and your module is a good fit for another project, it will probably get copied — not forked or made into a separate library — into this new project, and moving tests that lay beside it in the same folder structure is easy in comparison to fishing up tests in a mess that a separate test folder had become. (You may argue that it shouldn't be a mess in the first place but let's be realistic here).

So the choice is still yours, but I would argue that with mixed up tests you achieve all the same things as with a separate folder, but with less effort on keeping things tidy.

Share Improve this answer

Follow

edited Aug 28, 2017 at 14:43



Serp C


883 ● 1 ● 10 ● 24

answered Sep 28, 2016 at 7:31



Janusz Skonieczny

18.9k ● 11 ● 57 ● 64

1 what's the problem with deploying tests to production, actually? in my experience, it's very useful: allows users to actually run tests on their environment... when you get bug reports, you can ask the user to run the test suite to make sure everything is alright there and send hot patches for the test suite directly... besides, it's not because you put your tests in `module.tests` that it *will* end up in production, unless you did something wrong in your setup file... – [anarcat](#) Sep 15, 2017 at 13:09 

2 As I wrote in my answer I usually do not separate tests. But. Putting tests into distribution package may lead to executing stuff you normally would not want in production env (eg. some module level code). It of course depends on how tests are written, but to be on the safe side it you leave them off, no one will run something harmful by mistake. I'm not against including tests in distributions, but I understand that as a rule of thumb it is safer. And putting them in separate folder makes it super easy not-to include them in dist.
– [Janusz Skonieczny](#) Oct 30, 2017 at 7:19



26



I also tend to put my unit tests in the file itself, as Jeremy Cantrell above notes, although I tend to not put the test function in the main body, but rather put everything in an

```
if __name__ == '__main__':  
    do tests...
```

block. This ends up adding documentation to the file as 'example code' for how to use the python file you are testing.

I should add, I tend to write very tight modules/classes. If your modules require very large numbers of tests, you can put them in another, but even then, I'd still add:

```
if __name__ == '__main__':  
    import tests.thisModule  
    tests.thisModule.runtests
```

This lets anybody reading your source code know where to look for the test code.

Share Improve this answer

edited Sep 19, 2008 at 17:04

Follow


answered Sep 19, 2008 at 16:46



Thomas Andrews

1,597 ● 1 ● 13 ● 32

1 "as Jeremy Cantrell above notes" where? – [endolith](#) Aug 13, 2014 at 15:46

I like this way of working. The simplest of editors can be configured to run your file with a hot-key, so when you're viewing the code, you can run the tests in an instant. Unfortunately in languages other than Python this can look horrible, so if you're in a C++ or Java shop your colleagues may frown on this. It also doesn't work well with code coverage tools. – [Keeely](#) Jun 15, 2017 at 9:57 



14

I use a `tests/` directory, and then import the main application modules using relative imports. So in `MyApp/tests/foo.py`, there might be:



```
from .. import foo
```



to import the `MyApp.foo` module.



Share Improve this answer

answered Sep 14, 2008 at 18:18

Follow



[John Millikin](#)

201k ● 41 ● 215 ● 227

12 "ValueError: Attempted relative import in non-package"
– [cprn](#) Jul 25, 2016 at 19:12



13

I don't believe there is an established "best practice".

I put my tests in another directory outside of the app code. I then add the main app directory to `sys.path`



(allowing you to import the modules from anywhere) in my test runner script (which does some other stuff as well) before running all the tests. This way I never have to remove the tests directory from the main code when I release it, saving me time and effort, if an ever so tiny amount.

Share Improve this answer

answered Sep 14, 2008 at 6:46

Follow



[dwestbrook](#)

5,268 ● 3 ● 26 ● 20

4 *I then add the main app directory to sys.path* The problem is if your tests contain some helper modules (like test web server) and you need to import such helper modules from within your proper tests. – [Piotr Dobrogost](#) Nov 29, 2011 at 19:21

1 This is what it looks like for me:

```
os.sys.path.append(os.dirname('..'))
```

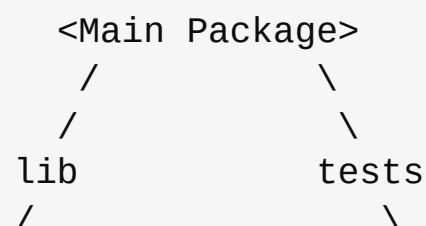
– [Mattwmaster58](#) Jul 9, 2018 at 4:25



13



From my experience in developing Testing frameworks in Python, I would suggest to put python unit tests in a separate directory. Maintain a symmetric directory structure. This would be helpful in packaging just the core libraries and not package the unit tests. Below is implemented through a schematic diagram.



```
[module1.py, module2.py, [ut_module1.py,  
module3.py module4.py, ut_module3.py,  
__init__.py]
```

In this way when you package these libraries using an rpm, you can just package the main library modules (only). This helps maintainability particularly in agile environment.

Share Improve this answer

answered Mar 28, 2014 at 4:22

Follow



[Rahul Biswas](#)

195 ● 1 ● 9

-
- 2 I've realized that one potential advantage of this approach is that the tests can be developed (and maybe even version controlled) as an independent application of their own. Of course, to every advantage there is a disadvantage -- maintaining symmetry is basically doing "double entry accounting," and makes refactoring more of a chore. – [Jasha](#)
Jun 1, 2018 at 11:41
-



I recommend you check some main Python projects on GitHub and get some ideas.

13



When your code gets larger and you add more libraries it's better to create a test folder in the same directory you have setup.py and mirror your project directory structure for each test type (unittest, integration, ...)



For example if you have a directory structure like:

```
myPackage/  
  myapp/  
    moduleA/  
      __init__.py  
      module_A.py  
    moduleB/  
      __init__.py  
      module_B.py  
  setup.py
```

After adding test folder you will have a directory structure like:

```
myPackage/  
  myapp/  
    moduleA/  
      __init__.py  
      module_A.py  
    moduleB/  
      __init__.py  
      module_B.py  
  test/  
    unit/  
      myapp/  
        moduleA/  
          module_A_test.py  
        moduleB/  
          module_B_test.py  
    integration/  
      myapp/  
        moduleA/  
          module_A_test.py  
        moduleB/  
          module_B_test.py  
  setup.py
```

Many properly written Python packages use the same structure. A very good example is the Boto package.

Check <https://github.com/boto/boto>

Share Improve this answer

answered May 9, 2016 at 17:54

Follow



Arash

619 ● 7 ● 10

-
- 2 It is recommended to use "tests" instead of "test", because "test" is a Python build in module.
docs.python.org/2/library/test.html – brodul Dec 14, 2017 at 9:59
-

Not always.. for example `matplotlib` has it under `matplotlib/lib/matplotlib/tests` (github.com/matplotlib/matplotlib/tree/...) , `sklearn` has it under `scikitearn/sklearn/tests` (github.com/scikit-learn/scikit-learn/tree/master/sklearn/tests) – alpha_989 Jan 28, 2018 at 19:52



How I do it...

9

Folder structure:



```
project/  
  src/  
    code.py  
  tests/  
  setup.py
```



Setup.py points to src/ as the location containing my projects modules, then i run:

```
setup.py develop
```

Which adds my project into site-packages, pointing to my working copy. To run my tests i use:

```
setup.py tests
```

Using whichever test runner I've configured.

Share Improve this answer

edited Apr 5, 2011 at 21:36

Follow

answered Mar 2, 2010 at 12:50



Dale Reidy

1,199 ● 10 ● 22

You seem to have over-ridden the term "module". Most Python programmers would probably think that the module is the file you called `code.py`. It would make more sense to call the top level directory "project". – [blokeley](#) Apr 5, 2011 at 11:50



5

I prefer toplevel tests directory. This does mean imports become a little more difficult. For that I have two solutions:



1. Use setuptools. Then you can pass

```
test_suite='tests.runalltests.suite' into
```

```
setup() , and can run the tests simply: python
```

```
setup.py test
```

2. Set PYTHONPATH when running the tests:

```
PYTHONPATH=. python tests/runalltests.py
```



Here's how that stuff is supported by code in M2Crypto:

- <http://svn.osafoundation.org/m2crypto/trunk/setup.py>
- <http://svn.osafoundation.org/m2crypto/trunk/tests/alltests.py>

If you prefer to run tests with nosetests you might need do something a little different.

Share Improve this answer

Follow

edited Oct 7, 2011 at 1:18



bstpierre

31.1k ● 15 ● 72 ● 104

answered Dec 19, 2008 at 23:48



Heikki Toivonen



5



I put my tests in the same directory as the code under test (CUT). In projects where I can tweak pytest with my plugin, for `foo.py` I use `foo.pt` for the tests which makes editing a particular module and its test together really easy: `vi foo.*`.



Where I can't do this, I use `foo_ut.py` or similar. You can still use `vi foo*` though that will also catch `foobar.py` and `foobar_ut.py` if those exist.

In either case I tweak the test discovery process to find these.

This puts the tests right beside the code in a directory listing, making it obvious that tests are there, and makes opening the tests as easy as it can possibly be when they're in a separate file. (For editors started from the command line, as described above; for GUI systems, click on the code file and the adjacent (or very nearly adjacent) test file.

As [others have pointed out](#), this also makes it easier to refactor and to extract the code for use elsewhere should that ever be necessary.

I really dislike the idea of putting tests in a completely different directory tree; why make it harder than necessary for developers to open up the tests when they're opening the file with the CUT? It's not like the vast majority of developers are so keen on writing or tweaking tests that they'll ignore any barrier to doing that, instead of using the barrier as an excuse. (Quite the opposite, in my experience; even when you make it as easy as possible I know many developers who can't be bothered to write tests.)

Share Improve this answer

edited Dec 31, 2021 at 5:07

Follow

answered Dec 12, 2018 at 10:10



cjs

27.1k ● 10 ● 91 ● 110



We use

3

```
app/src/code.py
app/testing/code_test.py
app/docs/..
```



In each test file we insert `../src/` in `sys.path`. It's not the nicest solution but works. I think it would be great if someone came up w/ something like maven in java that gives you standard conventions that just work, no matter what project you work on.

Share Improve this answer

Follow

edited Aug 20, 2019 at 8:07



buhtz

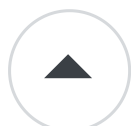
12.1k ● 20 ● 87 ● 183

answered Sep 24, 2008 at 17:44



André

13.3k ● 3 ● 35 ● 45



If the tests are simple, simply put them in the docstring -- most of the test frameworks for Python will be able to use that:

1



```
>>> import module
>>> module.method('test')
'testresult'
```



For other more involved tests, I'd put them either in

`../tests/test_module.py` or in `tests/test_module.py`.

Share Improve this answer

answered Sep 16, 2008 at 21:10

Follow



tholo



In C#, I've generally separated the tests into a separate assembly.

1



In Python -- so far -- I've tended to either write doctests, where the test is in the docstring of a function, or put them in the `if __name__ == "__main__"` block at the bottom of the module.



Share Improve this answer

edited Nov 7, 2008 at 12:01

Follow



Blair Conrad

241k ● 25 ● 136 ● 112

answered Sep 15, 2008 at 3:09



George V. Reilly

16.3k ● 7 ● 45 ● 39



0



When writing a package called "foo", I will put unit tests into a separate package "foo_test". Modules and subpackages will then have the same name as the SUT package module. E.g. tests for a module foo.x.y are found in foo_test.x.y. The `__init__.py` files of each testing package then contain an AllTests suite that includes all test suites of the package. setuptools provides a convenient way to specify the main testing package, so that after "python setup.py develop" you can just use



"python setup.py test" or "python setup.py test -s foo_test.x.SomeTestSuite" to the just a specific suite.

Share Improve this answer

answered Sep 15, 2008 at 14:56

Follow



[Sebastian Rittau](#)

21.2k ● 3 ● 26 ● 21



-3



I've recently started to program in Python, so I've not really had chance to find out best practice yet. But, I've written a module that goes and finds all the tests and runs them.

So, I have:



```
app/  
  appfile.py  
test/  
  appfileTest.py
```

I'll have to see how it goes as I progress to larger projects.

Share Improve this answer

answered Sep 14, 2008 at 18:02

Follow



[quamrana](#)

39.3k ● 13 ● 55 ● 76

5 Hi, the camelCase is a little weird in the python world.
– [Stuart Axon](#) Oct 5, 2015 at 19:15



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.