Running unittest with typical test directory structure

Asked 13 years, 11 months ago Modified 1 month ago Viewed 407k times

990

The very common directory structure for even a simple Python module seems to be to separate the unit tests into their own `test` directory:

```
new_project/
    antigravity/
        antigravity.py
    test/
        test_antigravity.py
    setup.py
    etc.
```

My question is simply **What's the usual way of actually running the tests?** I suspect this is obvious to everyone except me, but you can't just run `python test_antigravity.py` from the test directory as its `import antigravity` will fail as the module is not on the path.

I know I could modify PYTHONPATH and other search path related tricks, but I can't believe that's the simplest way - it's fine if you're the developer but not realistic to expect your users to use if they just want to check the tests are passing.

The other alternative is just to copy the test file into the other directory, but it seems a bit dumb and misses the point of having them in a separate directory to start with.

So, if you had just downloaded the source to my new project how would you run the unit tests? I'd prefer an answer that would let me say to my users: "To run the unit tests do X."

python unit-testing

Share Improve this question Follow

edited Jun 8, 2022 at 1:27
PythoNic
313 5 13

asked Dec 13, 2009 at 16:10
Major Major
10.4k 4 19 9

6 @EMP The proper solution when you need to set the search path is to... set the search path. What sort of solution were you expecting? – Carl Meyer Feb 17, 2012 at 20:12

9 @CarlMeyer another better solution is to use the `unittest` command line interface as described in my answer below so you don't have to add the directory to the path. – Pierre Jun 27, 2014 at 18:30

59 Same here. I just embarked on writing my very first unit tests in for a tiny Python project and took several days trying to reason with the fact that I can't readily run a test while keeping my sources in a src directory and tests in a test directory, seemingly with any of the existing test frameworks. I'll eventually accept things, figure out a way; but this has been a very frustrating introduction. (And I'm a unit testing veteran outside Python.) – Ates Goral Mar 30, 2016 at 20:43

Tip for coming up with a module structure: don't shadow built-in modules. `antigravity` is a CPython module and `test` is in the Python standard library. That's why I've moved to `tests` . – Joooeey Jan 27 at 16:04

27 Answers

Sorted by: Highest score (default)

903

The best solution in my opinion is to use the `unittest` command line interface which will add the directory to the `sys.path` so you don't have to (done in the `TestLoader` class).

For example for a directory structure like this:

```
new_project
├── antigravity.py
└── test_antigravity.py
```

You can just run:

```
$ cd new_project
$ python -m unittest test_antigravity
```

For a directory structure like yours:

```
new_project
├── antigravity
│   ├── __init__.py         # make it a package
│   └── antigravity.py
└── test
    ├── __init__.py         # also make test a package
    └── test_antigravity.py
```

And in the test modules inside the `test` package, you can import the `antigravity` package and its modules as usual:

```
# import the package
import antigravity

# import the antigravity module
from antigravity import antigravity

# or an object inside the antigravity module
from antigravity.antigravity import my_object
```

Running a single test module:

To run a single test module, in this case `test_antigravity.py`:

```
$ cd new_project
$ python -m unittest test.test_antigravity
```

Just reference the test module the same way you import it.

Running a single test case or test method:

Also you can run a single `TestCase` or a single test method:

```
$ python -m unittest test.test_antigravity.GravityTestCase
$ python -m unittest test.test_antigravity.GravityTestCase.test_method
```

Running all tests:

You can also use test discovery which will discover and run all the tests for you, they must be modules or packages named `test*.py` (can be changed with the `-p, --pattern` flag):

```
$ cd new_project
$ python -m unittest discover
$ # Also works without discover for Python 3
$ # as suggested by @Burrito in the comments
$ python -m unittest
```

This will run all the `test*.py` modules inside the `test` package.

Here you can find the updated official documentation of `discovery`.

Share Improve this answer Follow

edited Oct 4 at 12:36
frankfalse
1,788 2 5 20

answered Jun 17, 2014 at 14:49
Pierre
12.6k 6 44 64

85 `python -m unittest discover` will find and run tests in the `test` directory if they are named `test*.py`. If you named the subdirectory `tests`, use `python -m unittest discover -s tests`, and if you named the test files `antigravity_test.py`, use `python -m unittest discover -s tests -p '*test.py'` File names can use underscores but not dashes. – Mike3d0g May 18, 2015 at 2:49

20 This fails for me on Python 3 with the error `ImportError: No module named 'test.test_antigravity'` because of a conflict with the test sub-module of the unittest library. Maybe an expert can confirm and change the answer sub-directory name to e.g., 'tests' (plural). – expz Dec 22, 2016 at 21:45

18 My `test_antigravity.py` still throws an import error for both `import antigravity` and `from antigravity import antigravity`, as well. I have both `__init_.py` files and I am calling `python3 -m unittest discover` from the `new project` directory. What else could be wrong? – imrek May 2, 2017 at 20:11

34 file `test/__init__.py` is crucial here, even if empty – Francois Aug 2, 2018 at 13:32

7 @Mike3d0g not sure if you meant to imply that the directory name `test` is special...but just for the record, it isn't. :P `python -m unittest discover` works with test files in `tests/` just as well as `test/`. – ryan Oct 11, 2018 at 21:31

61

I've had the same problem for a long time. What I recently chose is the following directory structure:

```
project_path
├── Makefile
├── src
│   ├── script_1.py
│   ├── script_2.py
│   └── script_3.py
└── tests
    ├── __init__.py
    ├── test_script_1.py
    ├── test_script_2.py
    └── test_script_3.py
```

and in the `__init__.py` script of the test folder, I write the following:

```
import os
import sys
PROJECT_PATH = os.getcwd()
SOURCE_PATH = os.path.join(
    PROJECT_PATH,"src"
)
sys.path.append(SOURCE_PATH)
```

Super important for sharing the project is the Makefile, because it enforces running the scripts properly. Here is the command that I put in the Makefile:

```
run_tests:
    python -m unittest discover .
```

The Makefile is important not just because of the command it runs but also because of *where it runs it from*. If you would cd in tests and do `python -m unittest discover .` , it wouldn't work because the **init** script in unit_tests calls os.getcwd(), which would then point to the incorrect absolute path (that would be appended to sys.path and you would be missing your source folder). The scripts would run since discover finds all the tests, but they wouldn't run properly. So the Makefile is there to avoid having to remember this issue.

I really like this approach because I don't have to touch my src folder, my unit tests or my environment variables and everything runs smoothly.

Share Improve this answer Follow

edited Apr 22, 2022 at 17:19
Miles Erickson
2,574 2 17 18

answered Jan 14, 2020 at 11:15
Patrick Da Silva
1,574 1 11 15

2 Since I wrote this answer, I found a way to avoid the sys.path.append workaround. If I find the time I'll update my answer. – Patrick Da Silva Jul 31, 2020 at 20:56

27 "If I find the time I'll update my answer" – Joaquín L. Robles Dec 21, 2020 at 20:42

6 How is it going today? :) – programmar Apr 6, 2021 at 14:25

4 @PatrickDaSilva seems to be a complicated solution you have there. If it is not simple probably it's not worth it :-) – programmar Apr 12, 2021 at 13:22

1 @programmar I recommend you to pick a better solution... Joaquin's comment pretty much sums up my feeling about my answer right now – Patrick Da Silva Apr 13, 2021 at 14:20

54

The simplest solution for your users is to provide an executable script (`runtests.py` or some such) which bootstraps the necessary test environment, including, if needed, adding your root project directory to `sys.path` temporarily. This doesn't require users to set environment variables, something like this works fine in a bootstrap script:

```
import sys, os

sys.path.insert(0, os.path.dirname(__file__))
```

Then your instructions to your users can be as simple as " `python runtests.py` ".

Of course, if the path you need really is `os.path.dirname(__file__)` , then you don't need to add it to `sys.path` at all; Python always puts the directory of the currently running script at the beginning of `sys.path` , so depending on your directory structure, just locating your `runtests.py` at the right place might be all that's needed.

Also, the unittest module in Python 2.7+ (which is backported as unittest2 for Python 2.6 and earlier) now has test discovery built-in, so nose is no longer necessary if you want automated test discovery: your user instructions can be as simple as `python -m unittest discover` .

Share Improve this answer Follow

edited Aug 8, 2019 at 13:03
Guy Avraham
3,502 3 38 51

answered Dec 13, 2009 at 20:40
Carl Meyer
123k 20 106 116

1 I put some tests in a subfolder like as "Major Major". They can run with python -m unittest discover but how can I select to run only one of them. If I run python -m unittest tests/testxxxxx then it fails for path issue. Since dicovery mode solve everything I would expect that there is another trick to solve path issue without handcoding path fix you suggest in first point – Frederic Bazin May 23, 2012 at 16:07

3 @FredericBazin Don't use discovery if you only want a single test or test file, just name the module you want to run. If you name it as a module dotted-path (rather than a file path) it can figure out the search path correctly. See Peter's answer for more details. – Carl Meyer Jul 15, 2014 at 1:01

This hack was usefull in a scenario where I had to run something like `python -m pdb tests\test_antigravity.py` . Inside pdb, I executed `sys.path.insert(0, "antigravity")` which allowed the import statement to resolve as if I was running the module. – ixe013 Apr 23, 2019 at 11:46

26

I generally create a "run tests" script in the project directory (the one that is common to both the source directory and `test`) that loads my "All Tests" suite. This is usually boilerplate code, so I can reuse it from project to project.

run_tests.py:

```
import unittest
import test.all_tests
testSuite = test.all_tests.create_test_suite()
text_runner = unittest.TextTestRunner().run(testSuite)
```

test/all_tests.py (from How do I run all Python unit tests in a directory?)

```
import glob
import unittest

def create_test_suite():
    test_file_strings = glob.glob('test/test_*.py')
    module_strings = ['test.'+str[5:len(str)-3] for str in test_file_strings]
    suites = [unittest.defaultTestLoader.loadTestsFromName(name) \
              for name in module_strings]
    testSuite = unittest.TestSuite(suites)
    return testSuite
```

With this setup, you can indeed just `include antigravity` in your test modules. The downside is you would need more support code to execute a particular test... I just run them all every time.

Share Improve this answer Follow

edited May 23, 2017 at 12:03
Community Bot
1 1

answered Jun 7, 2010 at 19:30
stw_dev
842 11 14

2 I also wanted *a `run tests` script in the project directory* and found a lot cleaner way to do it. Highly recommended. – z33k Mar 26, 2018 at 19:14

18

From the article you linked to:

> Create a test_modulename.py file and put your unittest tests in it. Since the test modules are in a separate directory from your code, you may need to add your module's parent directory to your PYTHONPATH in order to run them:
>
> ```
> $ cd /path/to/googlemaps
>
> $ export PYTHONPATH=$PYTHONPATH:/path/to/googlemaps/googlemaps
>
> $ python test/test_googlemaps.py
> ```
>
> Finally, there is one more popular unit testing framework for Python (it's that important!), nose. nose helps simplify and extend the builtin unittest framework (it can, for example, automagically find your test code and setup your PYTHONPATH for you), but it is not included with the standard Python distribution.

Perhaps you should look at nose as it suggests?

Share Improve this answer Follow

edited Jan 18, 2019 at 21:07
Topher
596 4 15

answered Dec 13, 2009 at 16:25
Mark Byers
817k 195 1588 1454

6 Yes this works (for me), but I'm really asking for the simplest instructions that I can give users to my module to get them to run the tests. Modifying the path might actually be it, but I'm fishing for something more straight-forward. – Major Major Dec 13, 2009 at 16:39

6 So what does your python path look like after you've worked on a hundred projects? Am I supposed to manually go in and clean up my path? If so this is an odious design! – jeremyjjbrown Jun 21, 2014 at 23:16

16

I had the same problem, with a separate unit tests folder. From the mentioned suggestions I add the **absolute source path** to `sys.path` .

The benefit of the following solution is, that one can run the file `test/test_yourmodule.py` without changing at first into the test-directory:

```
import sys, os
testdir = os.path.dirname(__file__)
srcdir = '../antigravity'
sys.path.insert(0, os.path.abspath(os.path.join(testdir, srcdir)))

import antigravity
import unittest
```

Share Improve this answer Follow

edited Aug 25, 2018 at 8:41
per1234
897 5 13

answered Dec 4, 2013 at 9:45
admirableadmin
2,679 1 24 41

14

I noticed that if you run the unittest command line interface from your "src" directory, then imports work correctly without modification.

```
python -m unittest discover -s ../test
```

If you want to put that in a batch file in your project directory, you can do this:

```
setlocal & cd src & python -m unittest discover -s ../test
```

Share Improve this answer Follow

answered Aug 7, 2017 at 23:13
Alan L
1,815 20 28

2 Out of all the answers on this page, this is the only one that worked for me. I suspect other answers are missing some vital info. – RCross Sep 1, 2021 at 14:33

1 It's hilariously stupid that we have to do this. But what can you do.. It's the most simple and easy solution – musicman Oct 3, 2021 at 21:52

The other answers are relying on the python import system. With this answer you are specifying the path for your tests and I think the test runner modifies the path before running the test. – ta32 Feb 27, 2022 at 4:46

13

Solution/Example for Python unittest module

Given the following project structure:

```
ProjectName
 ├── project_name
 │    ├── models
 │    │    └── thing_1.py
 │    └── __main__.py
 └── test
      ├── models
      │    └── test_thing_1.py
      └── __main__.py
```

You can run your project from the root directory with `python project_name`, which calls `ProjectName/project_name/__main__.py`.

To run your tests with `python test`, effectively running `ProjectName/test/__main__.py`, you need to do the following:

1) Turn your `test/models` directory into a package by adding a `__init__.py` file. This makes the test cases within the sub directory accessible from the parent `test` directory.

```
# ProjectName/test/models/__init__.py

from .test_thing_1 import Thing1TestCase
```

2) Modify your system path in `test/__main__.py` to include the `project_name` directory.

```
# ProjectName/test/__main__.py

import sys
import unittest

sys.path.append('../project_name')

loader = unittest.TestLoader()
testSuite = loader.discover('test')
testRunner = unittest.TextTestRunner(verbosity=2)
testRunner.run(testSuite)
```

Now you can successfully import things from `project_name` in your tests.

```
# ProjectName/test/models/test_thing_1.py

import unittest
from project_name.models import Thing1  # this doesn't work without 'sys.path.append'
per step 2 above

class Thing1TestCase(unittest.TestCase):

    def test_thing_1_init(self):
        thing_id = 'ABC'
        thing1 = Thing1(thing_id)
        self.assertEqual(thing_id, thing.id)
```

Share Improve this answer Follow

answered Jun 28, 2017 at 23:45
Derek Soike
11.3k 3 80 74

This seems to be the correct way. This person is doing something similar including the strange `import .<module>` with the period. I am quite shocked at how impossible it is to find a concise description out there. It's like most Python developers don't use submodules and separate test/ and lib/ directories. How is it so complicated and undocumented? Can someone help me make a sample project and put it on github please? (link: github.com/Rhoynar/sample-python/tree/master/tests) – cppProgrammer Oct 28 at 4:19

11

if you run "python setup.py develop" then the package will be in the path. But you may not want to do that because you could infect your system python installation, which is why tools like virtualenv and buildout exist.

Share Improve this answer Follow

edited Jul 23, 2019 at 13:46

answered Dec 13, 2009 at 16:27
Tom Willis
5,260 23 34

7

Python 3+

Adding to @Pierre

Using `unittest` directory structure like this:

```
new_project
├── antigravity
│   ├── __init__.py         # make it a package
│   └── antigravity.py
└── test
    ├── __init__.py         # also make test a package
    └── test_antigravity.py
```

To run the test module `test_antigravity.py`:

```
$ cd new_project
$ python -m unittest test.test_antigravity
```

Or a single `TestCase`

```
$ python -m unittest test.test_antigravity.GravityTestCase
```

Mandatory don't forget the `__init__.py` even if empty otherwise will not work.

Share Improve this answer Follow

edited Oct 8, 2018 at 15:12

answered Sep 28, 2018 at 15:21
imbr
6,417 4 55 70

6

If you use VS Code and your tests are located on the same level as your project then running and debug your code doesn't work out of the box. What you can do is change your launch.json file:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python",
            "type": "python",
            "request": "launch",
            "stopOnEntry": false,
            "pythonPath": "${config:python.pythonPath}",
            "program": "${file}",
            "cwd": "${workspaceRoot}",
            "env": {},
            "envFile": "${workspaceRoot}/.env",
            "debugOptions": [
                "WaitOnAbnormalExit",
                "WaitOnNormalExit",
                "RedirectOutput"
            ]
        }
    ]
}
```

The key line here is envFile

```
"envFile": "${workspaceRoot}/.env",
```

In the root of your project add .env file

Inside of your .env file add path to the root of your project. This will temporarily add

PYTHONPATH=C:\YOUR\PYTHON\PROJECT\ROOT_DIRECTORY

path to your project and you will be able to use debug unit tests from VS Code

Share Improve this answer Follow

edited May 28, 2017 at 0:38

answered May 27, 2017 at 23:45
Vlad Bezden
85k 25 252 184

5

Use `setup.py develop` to make your working directory be part of the installed Python environment, then run the tests.

Share Improve this answer Follow

answered Dec 13, 2009 at 16:24
Ned Batchelder
366k 75 565 664

This gets me an `invalid command 'develop'` and this option isn't mentioned if I ask for `setup.py --help-commands` . Does there need to be something in the `setup.py` itself for this to work? – Major Major Dec 13, 2009 at 16:43

It's OK - the problem was I was missing an `import setuptools` from my `setup.py` file. But I guess that does go to show that this won't work all the time for other people's modules. – Major Major Dec 13, 2009 at 16:54

2 If you have pip, you can use that to install your package in "editable" mode: `pip install -e .` This likewise adds the package to the Python environment without copying the source, allowing you to continue to edit it where it lies. – Eric Smith Feb 4, 2014 at 23:39

`pip install -e .` is the exact same thing as `python setup.py develop` , it just monkeypatches your `setup.py` to use setuptools even if it doesn't actually, so it works either way. – Carl Meyer Jul 15, 2014 at 0:57

5

You can't import from the parent directory without some voodoo. Here's yet another way that works with at least Python 3.6.

First, have a file test/context.py with the following content:

```
import sys
import os
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
```

Then have the following import in the file test/test_antigravity.py:

```
import unittest
try:
    import context
except ModuleNotFoundError:
    import test.context
import antigravity
```

Note that the reason for this try-except clause is that

- *import test.context* fails when run with "python test_antigravity.py" and
- *import context* fails when run with "python -m unittest" from the new_project directory.

With this trickery they both work.

Now you can run **all** the test files within test directory with:

```
$ pwd
/projects/new_project
$ python -m unittest
```

or run an individual test file with:

```
$ cd test
$ python test_antigravity
```

Ok, it's not much prettier than having the content of context.py within test_antigravity.py, but maybe a little. Suggestions are welcome.

Share Improve this answer Follow

edited Oct 20, 2018 at 17:31

answered Oct 20, 2018 at 16:25
tjk
339 3 8

The trick with `context.py` is quite good: it allows to import main modules from tests when unit tests are ran via command line or in VSCode. – AntonK Feb 17 at 11:47

4

It's possible to use wrapper which runs selected or all tests.

For instance:

```
./run_tests antigravity/*.py
```

or to run all tests recursively use globbing (`tests/**/*.py`) (enable by `shopt -s globstar`).

The wrapper can basically use `argparse` to parse the arguments like:

```
parser = argparse.ArgumentParser()
parser.add_argument('files', nargs='*')
```

Then load all the tests:

```
for filename in args.files:
    exec(open(filename).read())
```

then add them into your test suite (using `inspect`):

```
alltests = unittest.TestSuite()
for name, obj in inspect.getmembers(sys.modules[__name__]):
    if inspect.isclass(obj) and name.startswith("FooTest"):
        alltests.addTest(unittest.makeSuite(obj))
```

and run them:

```
result = unittest.TextTestRunner(verbosity=2).run(alltests)
```

Check this example for more details.

See also: How to run all Python unit tests in a directory?

Share Improve this answer Follow

edited Nov 30, 2017 at 13:13

answered Jun 7, 2015 at 11:45

kenorb
158k 88 682 748

3

Following is my project structure:

```
ProjectFolder:
 - project:
     - __init__.py
     - item.py
 - tests:
     - test_item.py
```

I found it better to import in the setUp() method:

```
import unittest
import sys

class ItemTest(unittest.TestCase):

    def setUp(self):
        sys.path.insert(0, "../project")
        from project import item
        # further setup using this import

    def test_item_props(self):
        # do my assertions

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

Share Improve this answer Follow

answered Jul 25, 2017 at 14:01

rolika
381 1 10

3

What's the usual way of actually running the tests

I use Python 3.6.2

```
cd new_project

pytest test/test_antigravity.py
```

To install *pytest*: `sudo pip install pytest`

I didn't set any path variable and my imports are not failing with the same "test" project structure.

I commented out this stuff: `if __name__ == '__main__'` like this:

test_antigravity.py

```
import antigravity

class TestAntigravity(unittest.TestCase):

    def test_something(self):

        # ... test stuff here


# if __name__ == '__main__':
#
#     if __package__ is None:
#
#         import something
#         sys.path.append(path.dirname(path.dirname(path.abspath(__file__))))
#         from .. import antigravity
#
#     else:
#
#         from .. import antigravity
#
#     unittest.main()
```

Share Improve this answer Follow

edited Jul 26, 2017 at 21:55

answered Jul 26, 2017 at 21:43
aliopi
3,702 2 29 24

3

You should really use the pip tool.

Use `pip install -e .` to install your package in development mode. This is a very good practice, recommended by pytest (see their good practices documentation, where you can also find two project layouts to follow).

Share Improve this answer Follow

edited Aug 12, 2019 at 12:23
yanlend
470 4 10

answered Mar 7, 2014 at 7:51
squid
2,597 1 24 19

Why downvote this answer? I read the accepted answer and while it was not bad, `pytest` is way better to run tests, because of the console output you get, in color, with stack trace info and detailed assertion error information. – aliopi Jul 26, 2017 at 21:38

1

If you have multiple directories in your test directory, then you have to add to each directory an `__init__.py` file.

```
/home/johndoe/snakeoil
└── test
    ├── __init__.py
    └── frontend
        └── __init__.py
        └── test_foo.py
    └── backend
        └── __init__.py
        └── test_bar.py
```

Then to run every test at once, run:

```
python -m unittest discover -s /home/johndoe/snakeoil/test -t /home/johndoe/snakeoil
```

Source: `python -m unittest -h`

```
  -s START, --start-directory START
                        Directory to start discovery ('.' default)
  -t TOP, --top-level-directory TOP
                        Top level directory of project (defaults to start
                        directory)
```

Share Improve this answer Follow

answered Oct 30, 2018 at 15:31
Qlimax
5,241 4 28 31

0

This BASH script will execute the python unittest test directory from anywhere in the file system, no matter what working directory you are in.

This is useful when staying in the `./src` or `./example` working directory and you need a quick unit test:

```
#!/bin/bash

this_program="$0"
dirname="`dirname $this_program`"
readlink="`readlink -e $dirname`"

python -m unittest discover -s "$readlink"/test -v
```

No need for a `test/__init__.py` file to burden your package/memory-overhead during production.

Share Improve this answer Follow

answered Aug 13, 2018 at 0:50
John Greene
2,289 3 26 37

0

This way will let you run the test scripts from wherever you want without messing around with system variables from the command line.

This adds the main project folder to the python path, with the location found relative to the script itself, not relative to the current working directory.

```
import sys, os

sys.path.insert(0, os.path.dirname(os.path.dirname(os.path.realpath(__file__))))
```

Add that to the top of all your test scripts. That will add the main project folder to the system path, so any module imports that work from there will now work. And it doesn't matter where you run the tests from.

You can obviously change the project_path_hack file to match your main project folder location.

Share Improve this answer Follow

edited Jun 26, 2019 at 15:44

answered Jun 26, 2019 at 15:25
chasmani
2,382 2 24 35

0

A simple solution for *nix based systems (macOS, Linux); and probably also Git bash on Windows.

```
PYTHONPATH=$PWD python test/test_antigravity.py
```

`print` statement easily works, unlike `pytest test/test_antigravity.py`. A perfect way for "scripts", but not really for unittesting.

Of course, I want to do a proper automated testing, I would consider `pytest` with appropriate settings.

Share Improve this answer Follow

answered Nov 16, 2020 at 12:31
Polv
2,028 1 20 31

0

With `cwd` being the root project dir (`new_project` in your case), you can run the following command without `__init__.py` in any directory:

```
python -m unittest discover -s test
```

But you need `import` in `test_antigravity.py` as:

```
from antigravity import antigravity.your_object
```

instead of:

```
import antigravity.your_object
```

If you don't like `from antigravity` clause, you might like Alan L's answer.

Share Improve this answer Follow

answered Oct 6, 2022 at 3:36
catwith
935 10 14

0

Create a `conftest.py` file in your root or test directory.

```
import sys
# Runs before all tests and imports when you run pytest
def pytest_configure() -> None:
    sys.path.append(__file__) # or __file__.parent for the above directory, or __file__
/ 'src', etc.
```

How to run the tests: call `pytest` from any directory.

Note: I think the best way 99% of the time is to just tell people to update their PYTHONPATH (you already made them install python+all of the requirements, just tell them to update their `.bashrc`).

Note2: pytest can run unittest. (and I think it's generally more useful). unittest does not support `conftest.py` . You can use an `__init__.py` instead but in my experience, this file doesn't always run first, other files sometimes get imported before it is called depending on structure and how the command is called.

Share Improve this answer Follow

edited Jul 9 at 15:36

answered Jul 9 at 15:27
Alex Li
1 1

0

Noobs issues..

The file name issue

I've been googling around for an answer to this issue too but nothing worked for my case.

I'm using Windows and the reason unittest wasn't able to discover the modules is because I used hyphen on the file name. Renaming the file name with underscores **fixed the issue**.

See this stackoverflow issue

Share Improve this answer Follow

edited Jul 27 at 8:20

answered Jul 27 at 8:13
carmel
920 7 24

-1

If you are looking for a command line-only solution:

Based on the following directory structure (generalized with a dedicated source directory):

```
new_project/
    src/
        antigravity.py
    test/
        test_antigravity.py
```

Windows: (in `new_project`)

```
$ set PYTHONPATH=%PYTHONPATH%;%cd%\src
$ python -m unittest discover -s test
```

See this question if you want to use this in a batch for-loop.

Linux: (in `new_project`)

```
$ export PYTHONPATH=$PYTHONPATH:$(pwd)/src  [I think - please edit this answer if you
are a Linux user and you know this]
$ python -m unittest discover -s test
```

With this approach, it is also possible to add more directories to the PYTHONPATH if necessary.

Share Improve this answer Follow

edited Feb 1, 2019 at 11:28

answered Feb 1, 2019 at 9:57
pj.dewitte
421 3 11

-1

I think that the method outlined in **https://docs.python-guide.org/writing/structure/** is very clean:

(quoted here verbatim from that article)

To give the individual tests import context, create a tests/context.py file:

```
import os
import sys
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

import sample
```

Then, within the individual test modules, import the module like so:

```
from .context import sample
```

This will always work as expected, regardless of installation method.

Share Improve this answer Follow

answered Mar 2 at 20:41
Joseph Bolton
36 2 5

-2

unittest in your project have `setup.py` file. try:

```
python3 setup.py build
```

and

```
python3 setup.py develop --user
```

do the work of config paths an so on. try it!

Share Improve this answer Follow

edited Mar 15, 2021 at 22:45
Valentin Vignal
6,631 2 35 77

answered Mar 15, 2021 at 0:57
david92
1