# pytest-nodev Documentation

***Release 0.9.8***

**Alessandro Amici**

2016-07-16

Table of Contents

**Version** 0.9.8

**Date** 2016-07-16

Test-driven source code search for Python.

New to the concept of *test-driven code search*? Jump to the Quickstart for a 2 minutes hands-on overview. Curious about the technique? Head over to the Concepts section. The User's guide documents pytest-nodev usage in details and covers a few more examples.

If you have any feedback or you want to help out head over our main repository: https://github.com/nodev-io/pytest-nodev

CHAPTER 1

# Quickstart

## 1.1 New user FAQ

pytest-nodev is a simple test-driven search engine for Python code, it finds classes and functions that match the behaviour specified by the given tests.

**How does "test-driven code search" work?**

To be more precise pytest-nodev is a pytest plugin that helps you execute feature specification tests on all objects in the Python standard library and in all the modules you have installed.

**Who are pytest-nodev users?**

Python developers who've got better things to do than reinvent existing wheels.

**I need to write a 'parse_bool' function that robustly parses a boolean value from a string. Here is the test I intend to use to validate my own implementation once I write it.**:

```
def test_parse_bool():
    assert not parse_bool('false')
    assert not parse_bool('FALSE')
    assert not parse_bool('0')

    assert parse_bool('true')
    assert parse_bool('TRUE')
    assert parse_bool('1')
```

**Show me how I search for a ready-made implementation with pytest-nodev.**

First, install the latest version of pytest-nodev from the Python Package Index:

```
$ pip install pytest-nodev
```

Then copy your specification test to the `test_parse_bool.py` file and decorate it with `pytest.mark.candidate` as follows:

```
@pytest.mark.candidate('parse_bool')
def test_parse_bool():
    assert not parse_bool('false')
    assert not parse_bool('FALSE')
    assert not parse_bool('0')

    assert parse_bool('true')
    assert parse_bool('TRUE')
    assert parse_bool('1')
```

Finally, instruct pytest to run your test on all functions in the Python standard library:

```
$ py.test test_parse_bool.py --candidates-from-stdlib
======================= test session starts ==========================
platform darwin -- Python 3.5.0, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: /tmp, inifile: setup.cfg
plugins: nodev-1.0.0, timeout-1.0.0
collected 3259 items

test_parse_bool.py xxxxxxxxxxx[...]xxxxxxxxXxxxxxxxx[...]xxxxxxxxxxx

============================== 1 hit ================================

test_parse_bool.py::test_parse_bool[distutils.util:strtobool] HIT

==== 3258 xfailed, 1 xpassed, 27 pytest-warnings in 45.07 seconds ====
```

In less than a minute pytest-nodev collected more than 3000 functions from the standard library and run your specification test on all of them and you've got a HIT. The strtobool function in the distutils.util module passes the test, so now you should thoroughly review it and if you like it you may use it in your code, no need to write your own implementation.

**Wow! Does it work so well all the times?**

To be honest strtobool is a little known gem of the Python standard library that is just perfect for illustrating all the benefits of test-driven code search. Here are some of them in rough order of importance:

- a function imported is a one less function coded—and tested, documented, debugged, ported, maintained...
- it's battle tested code—lot's of old bugs have already been squashed
- it's other people code—there's an upstream to report new bugs to
- it gives you additional useful functionality—for free on top of that
- it's in the Python standard library—no additional dependency required

## 1.2 BIG FAT WARNING!

A lot of functions called with the wrong set of arguments may have unexpected consequences ranging from slightly annoying, think `os.mkdir('false')`, to **utterly catastrophic**, think `shutil.rmtree('/', True)`. Serious use of pytest-nodev, in particular using `--candidates-from-all`, require running the tests with operating-system level isolation, e.g. as a dedicated user or even better inside a dedicated container. The User's guide documents how to run pytest-nodev safely and efficiently.

## 1.3 Project resources

| | |
|---|---|
| Documentation | http://pytest-nodev.readthedocs.org |
| Support | https://stackoverflow.com/search?q=pytest-nodev |
| Development | https://github.com/nodev-io/pytest-nodev |
| Discussion | To be decided, see issue #15 |
| Download | https://pypi.python.org/pypi/pytest-nodev |
| Code quality | |
| nodev website | http://nodev.io |

## 1.4 Contributing

Contributions are very welcome. Please see the CONTRIBUTING document for the best way to help. If you encounter any problems, please file an issue along with a detailed description.

Authors:

- Alessandro Amici - @alexamici

Contributors:

- @kr1

Sponsors:

-

## 1.5 License

pytest-nodev is free and open source software distributed under the terms of the MIT license.

CHAPTER 2

# Concepts

**Warning:** This section is work in progress and there will be areas that are lacking.

## 2.1 Motivation

> "Have a look at this piece of code that I'm writing–I'm sure it has been written before. I wouldn't be surprised to find it verbatim somewhere on GitHub." - @kr1

Every piece of functionality in a software project requires code that lies somewhere in the wide reusability spectrum that goes form extremely custom and strongly tied to the specific implementation to completely generic and highly reusable.

On the *custom* side of the spectrum there is all the code that defines the features of the software and all the choices of its implementation. That one is code that need to be written.

On the other hand a seasoned software developer is trained to spot pieces of functionality that lie far enough on the *generic* side of the range that with high probability a library already implements it **and documents it well enough to be discovered with an internet search**.

In between the two extremes there is a huge gray area populated by pieces of functionality that are not *generic* enough to obviously deserve a place in a library, but are *common* enough that must have been already implemented by someone else for their software. This kind of code is doomed to be re-implemented again and again for the simple reason that **there is no way to search code by functionality**...

Or is it?

## 2.2 Test-driven code search

When developing new functionalities developers spend significant efforts searching for code to reuse, mainly via keyword-based searches, e.g. on StackOverflow and Google. Keyword-based search is quite effective in finding code that is explicitly designed and documented to be reused, e.g. libraries and frameworks, but typically fails to identify reusable functions and classes in the large corpus of auxiliary code of software projects.

TDR aims to address the limits of keyword-based search with test-driven code search that focuses instead on code behaviour and semantics. Developing a new feature in TDR starts with the developer writing the tests that will validate candidate implementations of the desired functionality. Before writing any functional code the tests are run against all functions and classes of all available projects. Any code passing the tests is presented to the developer as a candidate implementation for the target feature.

pytest-nodev is a pytest plugin that enables *test-driven code search* and consequently a software development strategy called *test-driven reuse* or TDR that we call *nodev*, that is an extension of the well known *test-driven development* or TDD.

The idea is that once the developer has written the tests that define the behaviour of a new function to a degree sufficient to validate the implementation they are going to write it is good enough to validate any implementation. Running the tests on a large set of functions may result in a *hit*, that is a function that already implements their feature.

Due to its nature the approach is better suited for discovering smaller functions with a generic signature.

## 2.3 Tests validation

Another use for pytest-nodev is, with a bit of additional work, to validate a project test suite. If a test passes with an unexpected object there are two possibilities, either the test is not strict enough and allows for false positives and needs to be updated, or the *hit* is actually a function you could use instead of your implementation.

Keywords:

- Source code *search by feature*, *search by functionality*, *search by specification* or *nodev*
- *Feature-specification test* and test suite or *Requirement-specification test*
- *Test-driven reuse* or *test-driven code search* or *test-driven source code search*

## 2.4 Bibliography


- "CodeGenie: a tool for test-driven source code search", O.A. Lazzarini Lemos *et al*, Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, 917–918, **2007**, ACM, http://dx.doi.org/10.1145/1297846.1297944
- "Code conjurer: Pulling reusable software out of thin air", O. Hummel *et al*, IEEE Software, (25) 5 45-52, **2008**, IEEE, http://dx.doi.org/10.1109/MS.2008.110 — PDF
- "Finding Source Code on the Web for Remix and Reuse", S.E. Sim *et al*, 251, **2013** — PDF
- "Test-Driven Reuse: Improving the Selection of Semantically Relevant Code", M. Nurolahzade, Ph.D. thesis, **2014**, UNIVERSITY OF CALGARY — PDF

CHAPTER 3

# User's guide

**Warning:** This section is work in progress and there will be areas that are lacking.

Intended audience: python developers who've got better things to do than reinvent wheels.

## 3.1 Installation

Install the latest version of pytest-nodev from the Python Package Index:

```
$ pip install pytest-nodev
```

## 3.2 Basic usage

Write a specification test instrumented with the `candidate` fixture in the `test_example.py` file. Run pytest with one of the `--candidates-from-*` options to select the search space, e.g. to search in the Python standard library:

```
$ py.test --candidates-from-stdlib test_example.py
```

## 3.3 Advanced usage

Use of `--candidates-from-all` may be very dangerous and it is disabled by default.

In order to search safely in all modules we suggest to use docker for OS-level isolation. To kickstart your advanced usage downlaod the nodev-tutorial:

```
$ git clone https://github.com/nodev-io/nodev-tutorial.git
$ cd nodev-tutorial
```

build the nodev docker image with all module from requirements.txt installed:

```
$ docker build -t nodev .
```

and run tests with:

```
$ docker run --rm -it -v `pwd`:/home/pytest nodev --candidates-from-all tests/test_factorial.py
```

Alternatively you can enable it on your regular user only after you have understood the risks and set up appropriate mitigation strategies by setting the `PYTEST_NODEV_MODE` environment variable to `FEARLESS`:

```
$ PYTEST_NODEV_MODE=FEARLESS py.test --candidates-from-all --candidates-includes .*util -- test_examp
```

## 3.4 Command line reference

The plugin adds the following options to pytest command line:

```
nodev:
  --candidates-from-stdlib
                        Collects candidates form the Python standard library.
  --candidates-from-all
                        Collects candidates form the Python standard library
                        and all installed packages. Disabled by default, see
                        the docs.
  --candidates-from-specs=CANDIDATES_FROM_SPECS=[CANDIDATES_FROM_SPECS=...]
                        Collects candidates from installed packages. Space
                        separated list of `pip` specs.
  --candidates-from-modules=CANDIDATES_FROM_MODULES=[CANDIDATES_FROM_MODULES=...]
                        Collects candidates from installed modules. Space
                        separated list of module names.
  --candidates-includes=CANDIDATES_INCLUDES=[CANDIDATES_INCLUDES=...]
                        Space separated list of regexs matching full object
                        names to include, defaults to include all objects
                        collected via `--candidates-from-*`.
  --candidates-excludes=CANDIDATES_EXCLUDES=[CANDIDATES_EXCLUDES=...]
                        Space separated list of regexs matching full object
                        names to exclude.
  --candidates-predicate=CANDIDATES_PREDICATE
                        Full name of the predicate passed to
                        `inspect.getmembers`, defaults to `builtins.callable`.
  --candidates-fail     Show candidates failures.
```

CHAPTER 4

# Design

This chapter documents the high-level design of the product and it is intended for developers contributing to the project.

**Note: Users of the product need not bother with the following. Unless they are curious :)**

## 4.1 Mission and vision

The project mission is to enable test-driven code search for Python with pytest.

Target use cases:

1. test-driven reuse
2. tests validation

Project goals:

1. collect all possible python live objects (modules, functions, classes, singletons, constants...)
2. enable flexible search space definition
3. let users turn normal tests into specification tests, and vice versa, with minimal effort

Project non-goals:

1. protect the user from unintended consequences (clashes with goal 1.), instead document how to use OS-level isolation/containerization
2. help users writing implementation-independent specification tests (think a `contains` function that also tests inside dict values and class attributes)

## 4.2 Software architecture

Logical components:

- the object collector with filtering
- the pytest plugin interface

## 4.3 Version goals

pytest-nodev strives to adhere to semantic versioning.

### 4.3.1 1.0.0 (upcoming release)

Minimal set of features to be operationally useful and to showcase the nodev approach. Reasonably safe to test, but not safe to use without OS-level isolation. No completeness and no performance guarantees.

- Search environment definition:
  - Support defining which modules to search. Command line `--candidates-from-*` options.
  - Support defining which objects to include/exclude by name or via a predicate test function. Command line `--candidates-includes/excludes/predicate` options.
- Object collection:
  - Collect most objects from the defined environment. It is ok to miss some objects for now.
- Test execution:
  - Execute tests instrumented with the `candidate` fixture once for every object collected. The tests are marked `xfail` unless the `--candidates-fail` command line option is given to make standard pytest reporting the most useful.
- Report:
  - Report which objects pass each test.
- Safety:
  - Interrupting hanging tests is delegated to pytest-timeout.
  - Internal modules and objects starting with an underscore are excluded.
  - Potentially dangerous, crashing, hard hanging or simply annoying objects belonging to the standard library are unconditionally blacklisted so that new users can test `--candidates-from-stdlib` without bothering with OS-level isolation.
  - Limited use of `--candidates-from-all`.
- Documentation:
  - Enough to inspire and raise interest in new users.
  - Enough to use it effectively and safely. Give a strategy to get OS-level isolation.