Pytest Cheat Sheet

Pytest

Pytest is a testing framework that allows users to write test codes using Python programming language. It is useful to write simple and scalable test cases for databases, APIs, or UI.

Advantages

- It can run multiple tests in parallel, which reduces the execution time of the test suite.
- It has its own way to detect the test file and test functions automatically, if not mentioned explicitly.
- It allows to skip a subset of the tests during execution; allows to run a subset of the entire test suite.
- It is free and open source.
- It is compatible with other test frameworks like nose and unittest.
- It supports fixtures and classes which makes it easier to create common test objects available throughout a module/session/function/class.
- It supports parameterization, which is instrumental in executing the same tests with different configurations using a simple marker.

Installation

Install Pytest using pip

```
>>> pip install pytest
```

Naming conventions to be followed

- File name should start with test_
- Every code to be executed with pytest should be wrapped as a function
- Function name should start with test_ or end with _test
- Class name should start with Test and should not have __init__

Finding version

```
>>> pytest --version
```

Pytest help

```
>>> pytest -h
```

Pytest command line logging

Pytest supports the following command line logging options.

```
logging:
  --log-level=LEVEL     level of messages to catch/display.
                        Not set by default, so it depends on the
root/parent log handler's effective level, where it is "WARNING" by
default.

--log-format=LOG_FORMAT
                        log format as used by the logging module.

--log-date-format=LOG_DATE_FORMAT
                        log date format as used by the logging
module.

--log-cli-level=LOG_CLI_LEVEL
                        cli logging level.

--log-cli-format=LOG_CLI_FORMAT
                        log format as used by the logging module.

--log-cli-date-format=LOG_CLI_DATE_FORMAT
                        log date format as used by the logging
module.

--log-file=LOG_FILE   path to a file when logging will be written to.

--log-file-level=LOG_FILE_LEVEL
                        log file logging level.

--log-file-format=LOG_FILE_FORMAT
                        log format as used by the logging module.

--log-file-date-format=LOG_FILE_DATE_FORMAT
                        log date format as used by the logging
module.
```

```
--log-auto-indent=LOG_AUTO_INDENT
                        Auto-indent multiline messages passed to the
logging module. Accepts true|on, false|off or an integer.
```

Stopping after the first (or N) failures

Pytest options to exit on failures,

```
-x, --exitfirst       exit instantly on first error or failed test.
```

Command line usage,

```
>>> pytest -x
>>> pytest --maxfail=2
```

Specifying tests / selecting tests

- To run tests in a module

```
>>> pytest tests/test_example_func.py
```

- To run all the tests below somepath

```
>>> pytest tests/
```

- To run a specific test function

```
>>> pytest test_example_func.py::test_roadster
```

- To run a specific test method

```
>>> py.test test_example_class.py::TestTeslaModelTag::test_roadster
```

Matching expressions

- To run only the tests with names that matches a string expression

```
-k EXPRESSION
only run tests which match the given substring expression. An
expression is a python evaluatable expression where all names are
substring-matched against test names and their parent classes.
Example: -k 'test_method or test_other' matches all test functions and
classes whose name contains 'test_method' or 'test_other', while -k
'not test_method' matches those that don't contain 'test_method' in
their names.
-k 'not test_method and not test_other' will eliminate the matches.
The matching is case-insensitive.
```

e.g. "TeslaModelTag and not roadster"

This will select all the test methods under TeslaModelTag except for the ones with “roadster” in their method names like TestMyClass.test_roadster

```
>>> pytest -k "TeslaModelTag and not roadster"
```

Modifying traceback printing

- To show local variables in traceback

```
>>> pytest --showlocals
```

- To show local variables (shortcut)

```
>>> pytest -l
```

- To show 'long' tracebacks for the first and last entry, but 'short' style for the other entries

```
>>> pytest --tb=auto
```

- To show exhaustive, informative traceback formatting

```
>>> pytest --tb=long
```

- To show shorter traceback format

```
>>> pytest --tb=short
```

- To show only one line per failure

```
>> pytest --tb=line
```

- To show Python standard library formatting

```
>>> pytest --tb=native
```

- To hide traceback

```
>>> pytest --tb=no
```

Dropping to PDB on failures

Pytest command line options include,

```
--pdb                 start the interactive Python debugger on errors
or KeyboardInterrupt.
--pdbcls=modulename:classname
                      start a custom interactive Python debugger on
errors. For example: --pdbcls=IPython.terminal.debugger:TerminalPdb
--trace               Immediately break when running each test.
--capture=method      per-test capturing method: one of
fd|sys|no|tee-sys.
-s                    shortcut for --capture=no.
```

- To drop to PDB on the first failure and then proceed to the rest of the test session

```
>>> pytest --pdb
```

- To drop to PDB on the first failure and then end the test session

```
>>> pytest -x --pdb
```

- To drop to PDB after certain number of failures

```
>>> pytest --pdb --maxfail=3
```

Setting a breakpoint / aka set_trace()

To set a breakpoint and enter pdb.set_trace(), pytest.set_trace() can be used.

```
import pytest

def test_speed_limit_front_motor(frequency, slip, pole=4):
    slip_factor = (1 - slip / 100)
    pytest.set_trace()
    sync_speed = 120 * frequency * slip_factor / pole
```

Same is demonstrated in the following example,

```
>>> pytest test_parametrize_ids.py -v -s
```

Running tests based on past failures and last modified time

Command line options on pytest include,

```
--lf, --last-failed   rerun only the tests that failed at the last
run (or all if none failed)

--ff, --failed-first  run all tests, but run the last failures first.
                      This may re-order tests and thus lead to
repeated fixture setup/teardown.

--nf, --new-first     run tests from new files first, then the rest
of the tests sorted by file mtime

--cache-show=[CACHESHOW]
                      show cache contents, don't perform collection
or tests. Optional argument: glob (default: '*').

--cache-clear         remove all cache contents at start of test run.
```

```
--lfnf={all,none}, --last-failed-no-failures={all,none}
                      which tests to run with no previously (known)
failures.

--sw, --stepwise      exit on test failure and continue from last
failing test next time

--sw-skip, --stepwise-skip
                      ignore the first failing test but stop on the
next failing test
```

The same is demonstrated in the examples,

```
>>> pytest test_dependency_with_parametrize_dynamic.py -v -s --lf
>>> pytest test_dependency_with_parametrize_dynamic.py -v -s --sw
```

Pytest reporting formats

Command line options supported by pytest for reporting include,

```
reporting:
  --durations=N         show N slowest setup/test durations (N=0 for
all).

  --durations-min=N     Minimal duration in seconds for inclusion in
slowest list. Default 0.005

-v, --verbose         increase verbosity.

--no-header           disable header

--no-summary          disable summary

-q, --quiet           decrease verbosity.

--verbosity=VERBOSE   set verbosity. Default is 0.

-r chars              show extra test summary info as specified by
chars: (f)ailed, (E)rror, (s)kipped, (x)failed, (X)passed, (p)assed,
(P)assed with output,
```

```
(a)ll except passed (p/P), or (A)ll. (w)arnings are enabled by
default (see --disable-warnings), 'N' can be used to reset the list.
                        (default: 'fE').

--disable-warnings, --disable-pytest-warnings
                        disable warnings summary

-l, --showlocals      show locals in tracebacks (disabled by
default).

--tb=style            traceback print mode
(auto/long/short/line/native/no).

--show-capture={no,stdout,stderr,log,all}
                        Controls how captured stdout/stderr/log is
shown on failed tests. Default is 'all'.

--full-trace          don't cut any tracebacks (default is to cut).

--color=color         color terminal output (yes/no/auto).

--code-highlight={yes,no}
                        Whether code should be highlighted (only if -
-color is also enabled)

--pastebin=mode       send failed|all info to bpaste.net pastebin
service.

--junit-xml=path      create junit-xml style report file at given
path.

--junit-prefix=str    prepend prefix to classnames in junit-xml
output

--html=path           create html report file at given path.

--self-contained-html
                        create a self-contained html file containing
all necessary styles, scripts, and images - this means that the
report may not render or
                        function where CSP restrictions are in place
(see https://developer.mozilla.org/docs/Web/Security/CSP)

--css=path            append given css file content to report style
file.
```

Creating JUnit XML format files

To create result files which can be read by Jenkins or other Continuous integration servers,

--junitxml=path_of_xml_file.xml invocation is used.

```
>>> pytest test_parametrize_ids.py --junitxml=parametrize_ids.xml
```

Markers

Markers are used to set various features/attributes to test functions.

They are used to group test functions.

```
import pytest

@pytest.mark.model_3
def test_acceleration_3():
    assert model_3['top_accerl'] < 5, 'Takes more than 5 seconds'

@pytest.mark.model_3
def test_awd_3():
    assert model_3['awd'], 'All Wheel Drive not enabled'
```

Customized markers are registered in pytest.ini file as follows,

```
[pytest]
markers =
    model_3: Run model_3 test cases
    model_s: Run model_s test cases
```

The same is demonstrated in the following example,

```
>>> pytest -m model_3 -v -s
```

Built-in markers

- usefixtures - use fixtures on a test function or class
- filterwarnings - filter certain warnings of a test function
- skip - always skip a test function
- skipif - skip a test function if a certain condition is met
- xfail - produce an “expected failure” outcome if a certain condition is met
- parametrize - perform multiple calls to the same test function

To list all the markers, custom and built-in, the following command is used,

```
>>> pytest --markers
```

Fixtures

Fixtures are functions, which will run before each test function to which it is applied.

Pytest command line options include,

```
--fixtures, --funcargs
                      show available fixtures, sorted by plugin
appearance (fixtures with leading '_' are only shown with '-v')
--fixtures-per-test   show fixtures per test
```

The fixtures option is used on the command line as,

```
>>> pytest --fixtures
```

They are created when first requested by a test, and are destroyed based on their scope.

```
import pytest
import random

@pytest.fixture(scope='module')
def frequency():
    return random.randint(120, 240)
```

The same is demonstrated in the following example

```
>>> pytest test_fixture.py -v -s
```

Command line options using conftest.py

Command line values can be passed to the test methods by using the configuration file, conftest.py

```
def pytest_addoption(parser):
    parser.addoption("--poles", action="store", type=int,help="Number
of poles in induction motor;")

@pytest.fixture(scope='session')
def poles(pytestconfig):
    return pytestconfig.getoption("poles")
```

Parametrizing test method

To execute the same test method over and over again for different sets of test data, parametrize marker is used.

- Test data is passed as a list.
- In case of multiple arguments, the test data is passed as a list of tuples, each tuple representing one set of arguments.
- By default, pytest generates the test ids.

```
import pytest

torque_values = [('acceleration',400,250),('acceleration',330,400)]

@pytest.mark.parametrize("stage, front_trq, rear_trq",torque_values)
def test_stability(stage, front_trq, rear_trq):
    if stage == 'acceleration':
        assert front_trq < rear_trq, "Unstable acceleration"
```

The same is demonstrated in the example,

```
>>> pytest test_parametrize.py -v -s
```

Parametrize with customized ids

The ids used for the test data in parametrize can be customized.

The ids can be made customized statically by passing a list of strings to the ids param of parametrize fixture.

```
import pytest
@pytest.mark.parametrize("slip,frequency",slip_frequency_values,
ids = ['state1','state2','state3','state4', 'state5'])
def test_speed_limit_front_motor(frequency, slip, pole=4):
    slip_factor = (1 - slip / 100)
    sync_speed = 120 * frequency * slip_factor / pole
    assert sync_speed < 6100, "Speed exceeds rated speed"
```

The same is demonstrated in the following example,

```
>>> pytest test_parametrize_ids.py -v -s
```

Also, the ids can be customized dynamically by means of functions.

```
import pytest
def id_gen(val):
    if 2 <= val <= 3:
        return "slip: {}".format(val)
    if 120 <= val <= 240:
        return "frequency: {}".format(val)

@pytest.mark.parametrize("slip,frequency",slip_frequency_values,
ids=id_gen)
def test_speed_limit_back_motor(frequency, slip, pole=4):
    slip_factor = (1 - slip / 100)
    sync_speed = 120 * frequency * slip_factor / pole
    assert sync_speed < 5950, "Speed exceeds rated speed"
```

The same is demonstrated in the following example,

```
>>> pytest test_parametrize_custom_ids.py -v -s
```

Parametrization using pytest_generate_tests

A test argument can also be dynamically parametrized from the conftest.py by means of pytest_generate_tests hook.

```
# conftest.py
def pytest_generate_tests(metafunc):
    variants = ['40', '60', '60D', '70', '70D', '85', '85D', 'P85+',
'P85D', '90D', 'P90D', '100D', 'P100D']
    if "variant" in metafunc.fixturenames:
        metafunc.parametrize("variant", variants, scope="session")
```

In the test method the argument is parametrized by passing its name as,

```
   def test_all_wheel_drive(variant):
    assert "D" in variant, "All Wheel Drive not supported"

    def test_performance_pack(variant):
    assert "+" in variant, '21" wheel not available'
```

The same is demonstrated in,

```
>>> pytest test_generate_test_hook.py -v -s
```

Usage of “indirect” in parametrization

The key word argument indirect is used in parametrize to manipulate parameters being passed to the underlying test function.

```
import pytest

@pytest.fixture(scope="session")
def slip_factor(request):
    return 1 - request.param / 100

@pytest.fixture(scope="session")
def sync_speed_factor(request):
    return 120 * request.param / POLE

@pytest.mark.parametrize("slip_factor,sync_speed_factor",
slip_frequency_values, indirect = ['slip_factor',
'sync_speed_factor'])
def test_speed_limit_back_motor(slip_factor, sync_speed_factor):
    sync_speed = slip_factor * sync_speed_factor
    assert sync_speed < 5950, "Speed exceeds rated speed"
```

The same is demonstrated in the example,

```
>>> pytest test_parametrize_indirect.py -v -s
```

Creating dependent test cases

pytest-dependency is a pytest plugin to manage dependencies of tests.

pytest-dependency plugin can be installed using,

```
>>> pip install pytest-dependency
```

The test_ludicrous_mode is marked as a dependent of the test function, test_all_wheel_drive.

test_ludicrous_mode will be executed only if test_all_wheel_drive passes and will be skipped if test_a fails.

Thus, the execution of the test_b is based on the result status of test_a.

```
import pytest

MODEL_S_VARIANT = '90D'

@pytest.mark.dependency()
def test_all_wheel_drive():
    assert "D" in MODEL_S_VARIANT, "All Wheel Drive not supported"


@pytest.mark.dependency(depends=["test_all_wheel_drive"])
def test_ludicrous_mode():
    assert "P" in MODEL_S_VARIANT, "Ludicrous mode not enabled"
```

The same is demonstrated in the example,

```
>>> pytest test_dependency.py -v -s
```

Dependency in parametrized test cases

While implementing dependency in parametrized test cases, care must be taken as to pass the exact test id of the test function/method depended.

This could be done as,

```
import pytest

variants = ['P85D', '90D', 'P90D', '100D', 'P100D']

@pytest.mark.parametrize("variant", [
        pytest.param('P85D',
marks=pytest.mark.dependency(name="awd01")),
        pytest.param('90D',
marks=pytest.mark.dependency(name="awd02")),
        pytest.param('P90D',
marks=pytest.mark.dependency(name="awd03")),
        pytest.param('100D',
marks=pytest.mark.dependency(name="awd04"))
    ])
def test_all_wheel_drive(variant):
    assert "D" in variant, "All Wheel Drive not supported"

@pytest.mark.parametrize("variant", [
        pytest.param('P85D',
marks=pytest.mark.dependency(name="lmod01", depends=["awd01"])),
        pytest.param('90D',
marks=pytest.mark.dependency(name="lmod02", depends=["awd02"])),
        pytest.param('P90D',
marks=pytest.mark.dependency(name="lmod03", depends=["awd03"])),
        pytest.param('100D',
marks=pytest.mark.dependency(name="lmod04", depends=["awd04"])),
    ])
def test_ludicrous_mode(variant):
    assert "P" in variant, "Ludicrous mode not enabled"
```

The same is demonstrated in,

```
>>> pytest test_dependency_with_parametrize_static.py -v -s
```

When the value to be parametrized is dynamic, the following method could be adopted,

```
import pytest

variants = ['P85D', '90D', 'P90D', '100D', 'P100D']


def variant_id(case, depend_case=None):
    case_id, count = [], 0
    if not depend_case:
        for variant in variants:
            case_id.append(pytest.param(variant,
            marks=pytest.mark.dependency(name=case + str(count))))
            count += 1
    else:
        for variant in variants:
            case_id.append(pytest.param(variant,
            marks=pytest.mark.dependency(name=case + str(count),
                                         depends=[depend_case +
                                         str(count)])))
    return case_id

@pytest.mark.parametrize("variant", variant_id("awd"))
def test_all_wheel_drive(variant):
    assert "D" in variant, "All Wheel Drive not supported"

@pytest.mark.parametrize("variant", variant_id("perf_pack"))
def test_performance_pack(variant):
    assert "+" in variant, '21" wheel not available'

    @pytest.mark.parametrize("variant", variant_id("lmode", "awd"))
def test_ludicrous_mode(variant):
    assert "P" in variant, "Ludicrous mode not enabled"
```

The same is demonstrated in the example,

```
>>> pytest test_dependency_with_parametrize_dynamic.py -v -s
```

Generating HTML report

pytest-html is a plugin for pytest that generates a HTML report for test results.

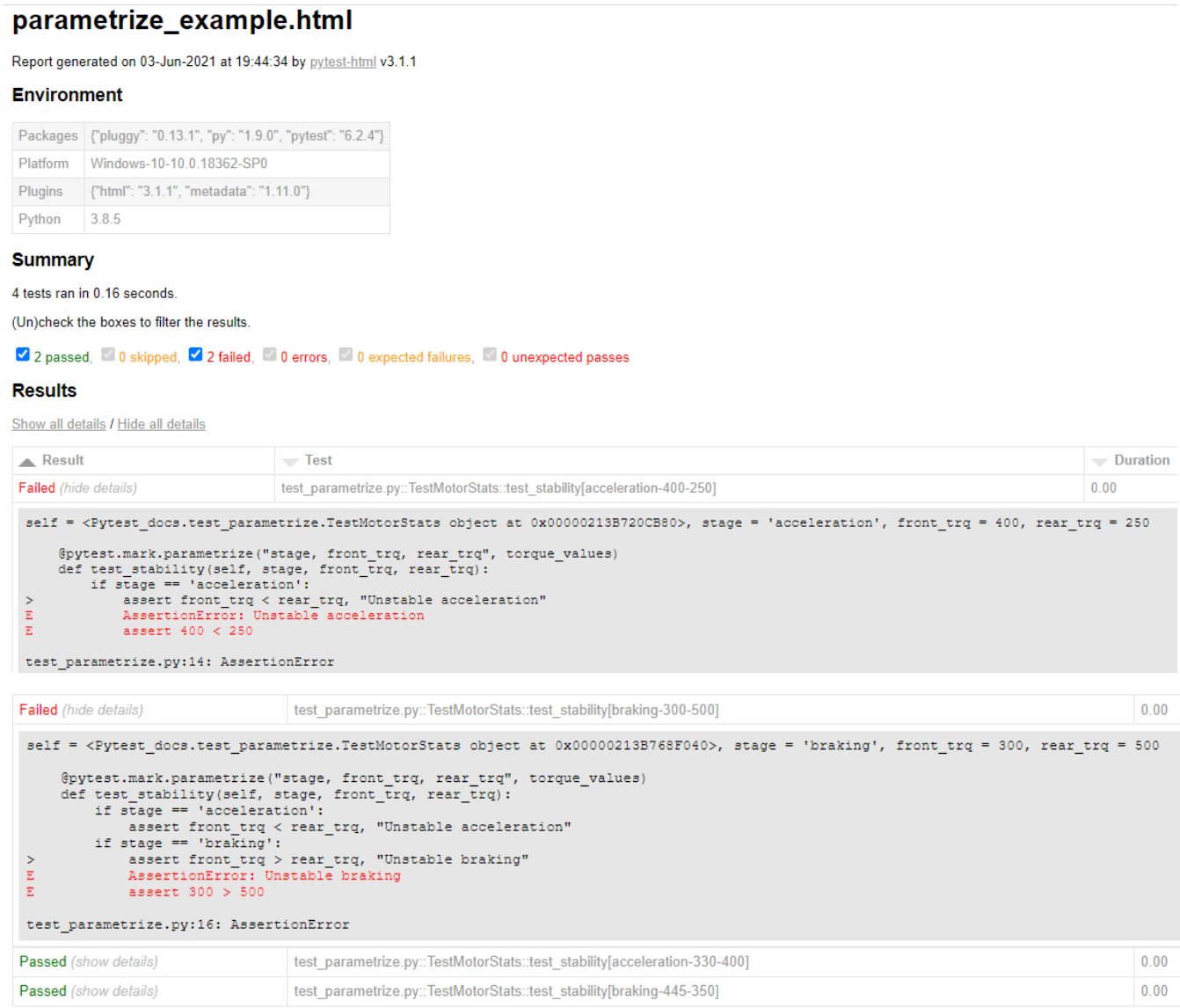
It can be installed using pip.

```
>>> pip install pytest-html
```

A simple html report can be generated by passing file path to the --html parameter.

The same is demonstrated in,

```
>>> pytest test_parametrize.py --html=parametrize_example.html
```



parametrize_example.html

Report generated on 03-Jun-2021 at 19:44:34 by pytest-html v3.1.1

Environment

Packages	{"pluggy": "0.13.1", "py": "1.9.0", "pytest": "6.2.4"}
Platform	Windows-10-10.0.18362-SP0
Plugins	{"html": "3.1.1", "metadata": "1.11.0"}
Python	3.8.5

Summary

4 tests ran in 0.16 seconds.

(Un)check the boxes to filter the results.

2 passed, 0 skipped, 2 failed, 0 errors, 0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration
Failed *(hide details)*	test_parametrize.py::TestMotorStats::test_stability[acceleration-400-250]	0.00

```
self = <Pytest_docs.test_parametrize.TestMotorStats object at 0x00000213B720CB80>, stage = 'acceleration', front_trq = 400, rear_trq = 250

    @pytest.mark.parametrize("stage, front_trq, rear_trq", torque_values)
    def test_stability(self, stage, front_trq, rear_trq):
        if stage == 'acceleration':
>           assert front_trq < rear_trq, "Unstable acceleration"
E           AssertionError: Unstable acceleration
E           assert 400 < 250

test_parametrize.py:14: AssertionError
```

Failed *(hide details)*	test_parametrize.py::TestMotorStats::test_stability[braking-300-500]	0.00

```
self = <Pytest_docs.test_parametrize.TestMotorStats object at 0x00000213B768F040>, stage = 'braking', front_trq = 300, rear_trq = 500

    @pytest.mark.parametrize("stage, front_trq, rear_trq", torque_values)
    def test_stability(self, stage, front_trq, rear_trq):
        if stage == 'acceleration':
            assert front_trq < rear_trq, "Unstable acceleration"
        if stage == 'braking':
>           assert front_trq > rear_trq, "Unstable braking"
E           AssertionError: Unstable braking
E           assert 300 > 500

test_parametrize.py:16: AssertionError
```

Passed *(show details)*	test_parametrize.py::TestMotorStats::test_stability[acceleration-330-400]	0.00
Passed *(show details)*	test_parametrize.py::TestMotorStats::test_stability[braking-445-350]	0.00

The report could be enhanced with customized CSS using --css option too.

References

Official documentation - https://docs.pytest.org/en/6.2.x/getting-started.html

Fixtures - https://docs.pytest.org/en/latest/how-to/fixtures.html

Markers - https://docs.pytest.org/en/6.2.x/example/markers.html

Dependency - https://pytest-dependency.readthedocs.io/en/stable/usage.html#

Parametrize - https://docs.pytest.org/en/stable/example/parametrize.html