

Testing, Timing, Documentation, Extras

Robert Klöforn

Center for Mathematical Sciences
Lund University

2023

Content

- ▶ Python modules
- ▶ Testing: `pytest` and `unittest`
- ▶ Integration testing
- ▶ Decorators
- ▶ Timing
- ▶ Documentation
- ▶ Generators

Writing Python Modules

- ▶ Every .py file can be used as a module Example:

```
import bisection
```

where bisection.py is a file in a known folder. This is ok when we only have a few code snippets.

bisection.py

```
import numpy

# a simple bisection algorithm
def bisect(f, a, b, tol = 1e-8, maxit=100):
    ...

# important when used as module and sometimes directly
# code that should only be executed when run directly
if __name__ == '__main__':
    ...
```

- ▶ A subdirectory can be a Python module, needs a `__init__.py` file (can be empty).

__init__.py

```
from .bisection import bisect
```

Writing Python Modules

- ▶ Every .py file can be used as a module Example:

```
import bisection
```

where bisection.py is a file in a known folder. This is ok when we only have a few code snippets.

bisection.py

```
import numpy

# a simple bisection algorithm
def bisect(f, a, b, tol = 1e-8, maxit=100):
    ...

# important when used as module and sometimes directly
# code that should only be executed when run directly
if __name__ == '__main__':
    ...
```

- ▶ A subdirectory can be a Python module, needs a `__init__.py` file (can be empty).

__init__.py

```
from .bisection import bisect
```

Software testing

Why do we need software testing?

<https://www.openhub.net/p/dune-project>

<https://www.openhub.net/p/opm>

- ▶ Unit testing (i.e. testing a small piece of code) (today)
- ▶ Integration testing (e.g. test several units together)
- ▶ Continuous Integration (today)
- ▶ System testing (test the whole product, α , β ,...)
- ▶ ...
- ▶ Performance testing
<https://openbenchmarking.org/test/system/opm>

Software testing

Why do we need software testing?

<https://www.openhub.net/p/dune-project>

<https://www.openhub.net/p/opm>

- ▶ Unit testing (i.e. testing a small piece of code) (today)
- ▶ Integration testing (e.g. test several units together)
- ▶ Continuous Integration (today)
- ▶ System testing (test the whole product, α , β ,...)
- ▶ ...
- ▶ Performance testing
<https://openbenchmarking.org/test/system/opm>

Software testing

Why do we need software testing?

<https://www.openhub.net/p/dune-project>

<https://www.openhub.net/p/opm>

- ▶ Unit testing (i.e. testing a small piece of code) (today)
- ▶ Integration testing (e.g. test several units together)
- ▶ Continuous Integration (today)
- ▶ System testing (test the whole product, α , β , ...)
- ▶ ...
- ▶ Performance testing
<https://openbenchmarking.org/test/system/opm>

Software testing

Why do we need software testing?

<https://www.openhub.net/p/dune-project>

<https://www.openhub.net/p/opm>

- ▶ Unit testing (i.e. testing a small piece of code) (today)
- ▶ Integration testing (e.g. test several units together)
- ▶ Continuous Integration (today)
- ▶ System testing (test the whole product, α , β ,...)
- ▶ ...
- ▶ Performance testing
<https://openbenchmarking.org/test/system/opm>

Software testing

Why do we need software testing?

<https://www.openhub.net/p/dune-project>

<https://www.openhub.net/p/opm>

- ▶ Unit testing (i.e. testing a small piece of code) (today)
- ▶ Integration testing (e.g. test several units together)
- ▶ Continuous Integration (today)
- ▶ System testing (test the whole product, α , β ,...)
- ▶ ...
- ▶ Performance testing
<https://openbenchmarking.org/test/system/opm>

Software testing

Why do we need software testing?

<https://www.openhub.net/p/dune-project>

<https://www.openhub.net/p/opm>

- ▶ Unit testing (i.e. testing a small piece of code) (today)
- ▶ Integration testing (e.g. test several units together)
- ▶ Continuous Integration (today)
- ▶ System testing (test the whole product, α , β ,...)
- ▶ ...
- ▶ Performance testing

<https://openbenchmarking.org/test/system/opm>

A brief overview on some techniques

- ▶ gitlab CI, what could you do
 - ▶ <https://gitlab.dune-project.org/core/dune-common>
 - ▶ <https://gitlab.dune-project.org/dune-fem/dune-fem>
 - ▶ <https://gitlab.dune-project.org/infrastructure/dune-nightly-test>
- ▶ github actions and pytest (today)

A brief overview on some techniques

- ▶ gitlab CI, what could you do
 - ▶ <https://gitlab.dune-project.org/core/dune-common>
 - ▶ <https://gitlab.dune-project.org/dune-fem/dune-fem>
 - ▶ <https://gitlab.dune-project.org/infrastructure/dune-nightly-test>
- ▶ github actions and pytest (today)

A brief overview on some techniques

- ▶ gitlab CI, what could you do
 - ▶ <https://gitlab.dune-project.org/core/dune-common>
 - ▶ <https://gitlab.dune-project.org/dune-fem/dune-fem>
 - ▶ <https://gitlab.dune-project.org/infrastructure/dune-nightly-test>
- ▶ github actions and pytest (today)

A brief overview on some techniques

- ▶ gitlab CI, what could you do
 - ▶ <https://gitlab.dune-project.org/core/dune-common>
 - ▶ <https://gitlab.dune-project.org/dune-fem/dune-fem>
 - ▶ <https://gitlab.dune-project.org/infrastructure/dune-nightly-test>
- ▶ github actions and pytest (today)

A brief overview on some techniques

- ▶ gitlab CI, what could you do
 - ▶ <https://gitlab.dune-project.org/core/dune-common>
 - ▶ <https://gitlab.dune-project.org/dune-fem/dune-fem>
 - ▶ <https://gitlab.dune-project.org/infrastructure/dune-nightly-test>
- ▶ github actions and pytest (today)

unittest

Run all unit tests in a specific folder:

```
python -m unittest discover
```

A typical result would look like this:

```
-----  
Ran 2 tests in 0.001s  
  
OK (expected failures=1)
```


unittest

Run all unit tests in a specific folder:

```
python -m unittest discover
```

A typical result would look like this:

```
-----  
Ran 2 tests in 0.001s  
  
OK (expected failures=1)
```

unittest example

A test case with unittest needs to implement a test case class:

test_bisection_unittest.py

```
import unittest
from bisection import bisect

# derive from unittest.TestCase
class TestIdentity(unittest.TestCase):

    # all methods that start with test are executed
    def testRoot(self):
        interval, root = bisect(lambda x: x, -1.2, 1.,tol=1.e-8)
        expected = 0.
        self.assertAlmostEqual(root , expected)

    # this method will be ignored
    def somethingelse(self):
        self.testRoot()

    # test expected failures
    @unittest.expectedFailure
    def testInterval(self):
        interval, root = bisect(lambda x: x, -1.2, -0.5,tol=1.e-8)
        expected = 0.
        self.assertAlmostEqual(root , expected)

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

pytest

Like the unittest discover function **pytest** will find all files in a given folder that start with **test**

```
pytest
```

The output will look like this:

```
===== test session starts =====
platform linux -- Python 3.11.5, pytest-7.4.0, pluggy-1.2.0
rootdir: ....pythontesting
plugins: anyio-3.7.1
collected 7 items

test_bisection.py ..FF                                [ 57%]
test_bisection_unittest.py xx.                        [100%]

===== FAILURES =====
...
===== short test summary info =====
FAILED test_bisection.py::test[inpt1-0] - assert inf < 1e-08
FAILED test_bisection.py::test[inpt2-inf] - assert inf < 1e-08
===== 2 failed, 3 passed, 2 xfailed in 0.50s =====
```

Note: **pytest** also runs all tests defined via **unittest**!

pytest

Like the unittest discover function **pytest** will find all files in a given folder that start with **test**

```
pytest
```

The output will look like this:

```
===== test session starts =====
platform linux -- Python 3.11.5, pytest-7.4.0, pluggy-1.2.0
rootdir: ....pythontesting
plugins: anyio-3.7.1
collected 7 items

test_bisection.py ..FF                                [ 57%]
test_bisection_unittest.py xx.                        [100%]

===== FAILURES =====
...
===== short test summary info =====
FAILED test_bisection.py::test[inpt1-0] - assert inf < 1e-08
FAILED test_bisection.py::test[inpt2-inf] - assert inf < 1e-08
===== 2 failed, 3 passed, 2 xfailed in 0.50s =====
```

Note: **pytest** also runs all tests defined via **unittest**!

pytest

Like the unittest discover function **pytest** will find all files in a given folder that start with **test**

```
pytest
```

The output will look like this:

```
===== test session starts =====
platform linux -- Python 3.11.5, pytest-7.4.0, pluggy-1.2.0
rootdir: ....pythontesting
plugins: anyio-3.7.1
collected 7 items

test_bisection.py ..FF                                [ 57%]
test_bisection_unittest.py xx.                        [100%]

===== FAILURES =====
...
===== short test summary info =====
FAILED test_bisection.py::test[inpt1-0] - assert inf < 1e-08
FAILED test_bisection.py::test[inpt2-inf] - assert inf < 1e-08
===== 2 failed, 3 passed, 2 xfailed in 0.50s =====
```

Note: **pytest** also runs all tests defined via **unittest**!

Why pytest?

It's simple to use with github actions if the repo is public.

The screenshot shows the GitHub Actions interface for a public repository. At the top, a navigation bar includes links for Code, Issues, Pull requests, Actions (highlighted with a red arrow), Projects, Wiki, Security, Insights, and Settings. Below the navigation bar, the heading "Get started with GitHub Actions" is displayed, with a red arrow pointing to the "Actions" tab. Underneath, a subheading "Suggested for this repository" is followed by a search bar and a grid of six workflow suggestions. A green arrow points from the "Suggested for this repository" text to the first suggestion, "Python Package using Anaconda", which is also marked with a green "(2)". The other suggestions are "Publish Python Package", "Django", "Pylint", "Python application", and "Python package". The "Python package" suggestion is marked with a red arrow and a red "2". Each suggestion card includes a title, a description, a "Configure" button, and a Python logo.

<> Code Issues Pull requests **Actions** Projects Wiki Security Insights Settings

Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

Skip this and [set up a workflow yourself](#) →

Search workflows

Suggested for this repository

Python Package using Anaconda
By GitHub Actions

Create and test a Python package on multiple Python versions using Anaconda for package management.

Configure

Python

Publish Python Package
By GitHub Actions

Publish a Python Package to PyPI on release.

Configure

Python

Django
By GitHub Actions

Build and Test a Django Project

Configure

Python

Pylint
By GitHub Actions

Lint a Python application with pylint.

Configure

Python

Python application
By GitHub Actions

Create and test a Python application.

Configure

Python

Python package
By GitHub Actions

Create and test a Python package on multiple Python versions.

Configure

Python

Github actions

pythontesting / .github / workflows / python-package.yml in main

Cancel changes Commit changes...

Edit Preview Code 55% faster with GitHub Copilot Spaces 2 No wrap

```
1 # This workflow will install Python dependencies, run tests and lint with a variety of Python versions
2 # For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing
3
4 name: Python package
5
6 on:
7   push:
8     branches: [ "main" ]
9   pull_request:
10     branches: [ "main" ]
11
12 jobs:
13   build:
14
15     runs-on: ubuntu-latest
16     strategy:
17       fail-fast: false
18     matrix:
19       python-version: ["3.9", "3.10", "3.11"]
20
21     steps:
22     - uses: actions/checkout@v3
23     - name: Set up Python ${ matrix.python-version }
24       uses: actions/setup-python@v3
25       with:
26         python-version: ${ matrix.python-version }
```

Use control + shift + m to toggle the tab key moving focus. Alternatively, use esc then tab to move to the next interactive element on the page.

Use control + space to trigger autocomplete in most situations.

Marketplace Documentation

Search Marketplace for Actions

Featured Actions

- Upload a Build Artifact** 2.5k
By actions
Upload a build artifact that can be used by subsequent workflow steps
- Setup Java JDK** 1.2k
By actions
Set up a specific version of the Java JDK and add the command-line tools to the PATH
- Setup Go environment** 1.2k
By actions
Setup a Go environment and add it to the PATH
- Setup .NET Core SDK** 805
By actions
Used to build and publish .NET source. Set up a specific version of the .NET and authentication to private NuGet repository
- First interaction** 646

pytest example

test_bisection.py

```
# import the pytest module use with the below tests
import pytest

# list Python modules in requirements.txt
import numpy as np
# own modules cannot be in requirements.txt
from bisection import bisect
# a simple test
def test_root():
    tol = 1.e-8
    interval, root = bisect(lambda x: x, -1.2, 1., tol=tol)
    assert abs(root) <= tol

def root_param(a, b, maxit):
    tol = 1.e-8
    interval, root = bisect(lambda x: x, a, b, tol=tol,
                           maxit=maxit)
    print(root)
    return root

# use parameterize decorators to test different parameter sets
@pytest.mark.parametrize("inpt, exptd",
    [((-1.2, 1, 100), 0), ((-1.2, 1, 10), 0), ((-2, -1, 100),
    np.inf)])
def test(inpt, exptd):
    root = root_param(*inpt)
    assert abs(root - exptd) < 1e-8
```

Q: What does the @ symbol indicate?

pytest example

test_bisection.py

```
# import the pytest module use with the below tests
import pytest

# list Python modules in requirements.txt
import numpy as np
# own modules cannot be in requirements.txt
from bisection import bisect
# a simple test
def test_root():
    tol = 1.e-8
    interval, root = bisect(lambda x: x, -1.2, 1., tol=tol)
    assert abs(root) <= tol

def root_param(a, b, maxit):
    tol = 1.e-8
    interval, root = bisect(lambda x: x, a, b, tol=tol,
                           maxit=maxit)
    print(root)
    return root

# use parameterize decorators to test different parameter sets
@pytest.mark.parametrize("inpt, exptd",
    [((-1.2, 1, 100), 0), ((-1.2, 1, 10), 0), ((-2, -1, 100),
    np.inf)])
def test(inpt, exptd):
    root = root_param(*inpt)
    assert abs(root - exptd) < 1e-8
```

Q: What does the @ symbol indicate?

Development strategy

In parallel to coding write tests (or maybe even before). Assign one group member to be responsible for testing.

- ▶ check the code
- ▶ document its use
- ▶ and document what has been tested

Development strategy

In parallel to coding write tests (or maybe even before). Assign one group member to be responsible for testing.

- ▶ check the code
- ▶ document its use
- ▶ and document what has been tested

Development strategy

In parallel to coding write tests (or maybe even before). Assign one group member to be responsible for testing.

- ▶ check the code
- ▶ document its use
- ▶ and document what has been tested

Function decorators 1/2

A function decorator is a short hand for modifying an existing function without changing its name.

Example:

```
def how_sparse(A):  
    return len(A.reshape(-1).nonzero()[0])
```

Make sure it works also for lists:

```
def cast2array(f):  
    def new_function(obj):  
        fA = f(array(obj))  
        return fA  
    return new_function  
  
how_sparse=cast2array(how_sparse)
```

Function decorators 1/2

A function decorator is a short hand for modifying an existing function without changing its name.

Example:

```
def how_sparse(A):  
    return len(A.reshape(-1).nonzero()[0])
```

Make sure it works also for lists:

```
def cast2array(f):  
    def new_function(obj):  
        fA = f(array(obj))  
        return fA  
    return new_function  
  
how_sparse = cast2array(how_sparse)
```

Function decorators 2/2

The decorator way:

```
def cast2array(f):  
    def new_function(obj):  
        fA = f(array(obj))  
        return fA  
    return new_function  
  
@cast2array  
def how_sparse(A):  
    return len(A.reshape(-1).nonzero()[0])
```

Timing – the time module

```
from numpy import array, arange
from numpy import sum as npsum

import time

a = array(arange(0,1000))

# default simple way of timing
start = time.time()

# commonly made mistake, use np.sum instead
# default sum implementation does not know about arrays
s = sum(a)

print(f"Standard sum took {time.time() - start:.6e} sec.")

start = time.time()

s = npsum(a)

print(f"Standard sum took {time.time() - start:6e} sec.")
```


Timing – timeit

```
python -m timeit 'sum(range(100))'
```

```
python -m timeit 'sum(range(100))'
```

```
python -m timeit -n 10 -r 10 'sum(range(100))'
```

Timing – timeit

```
python -m timeit 'sum(range(100))'
```

```
python -m timeit 'sum(range(100))'
```

```
python -m timeit -n 10 -r 10 'sum(range(100))'
```

Timing – timeit

```
python -m timeit 'sum(range(100))'
```

```
python -m timeit 'sum(range(100))'
```

```
python -m timeit -n 10 -r 10 'sum(range(100))'
```

Timing – timeit and ipython

In the ipython shell we can simply write

```
%timeit sum(range(100))
```

We can use the same parameters

```
%timeit -n 100 -r 10 sum(range(100))
```

Note: If no numbers are supplied the suitable values are chosen.

Timing – timeit and ipython

In the ipython shell we can simply write

```
%timeit sum(range(100))
```

We can use the same parameters

```
%timeit -n 100 -r 10 sum(range(100))
```

Note: If no numbers are supplied the suitable values are chosen.

Timing – timeit and ipython

In the ipython shell we can simply write

```
%timeit sum(range(100))
```

We can use the same parameters

```
%timeit -n 100 -r 10 sum(range(100))
```

Note: If no numbers are supplied the suitable values are chosen.

Timing – decorators

Task: Write a decorator for timing specific functions!

Documentation – README.md

An easy way for project documentation is the README.md file. Written in markdown it can display code, math, and normal text.

README.md

```
# pythontesting

This is a simple project to demonstrate testing with Python and
github.

## Testing frameworks

It is recommended to use [pytest][0]. pytest can also
    automatically include tests
written with [unittest][2].

## Documentation

Python documentation is usually created with [sphinx][1].

[0]: https://pytest.org
[1]: https://www.sphinx-doc.org
[2]: https://docs.python.org/3/library/unittest.html
```


Documentation – sphinx

Install sphinx, e.g. using pip

```
pip install sphinx
```

Using conda this may look different.

Then run the script

```
sphinx-quickstart
```

This will create a starting point for your documentation.

You should then make changes and

```
make html
```

to update the documentation.

Documentation – sphinx

Install sphinx, e.g. using pip

```
pip install sphinx
```

Using conda this may look different.

Then run the script

```
sphinx-quickstart
```

This will create a starting point for your documentation.

You should then make changes and

```
make html
```

to update the documentation.

Documentation – sphinx

Install sphinx, e.g. using pip

```
pip install sphinx
```

Using conda this may look different.

Then run the script

```
sphinx-quickstart
```

This will create a starting point for your documentation.

You should then make changes and

```
make html
```

to update the documentation.

Generators

Definition: A generator generates objects (to be passed to `for` loop). Similar to a list except that the objects need not to exist before entering the loop. A generator-like object is `range`:

```
for i in range(100000000):  
    if i > 10:  
        break
```

See also

```
rr = range ( 20 )  
print ( rr )
```

`rr` is not a `list`, but a tool which can be used to generate a list:

```
rrl=list(rr)
```

Generators – Python definition

Creation of generators is possible with the keyword **yield**:

```
def odd_numbers(n):  
    "generator for odd numbers less than n"  
    for k in range(n):  
        if k % 2 == 1:  
            yield k
```

Then use it like this:

```
g = odd_numbers(10)  
for k in g:  
    ... # do something with k
```

Infinite generators

Note: Just as in mathematics, generators may be infinite!

```
def odd_numbers():  
    "generator for all odd numbers"  
    k = 1  
    while True:  
        if k % 2 == 1:  
            yield k  
        k += 1
```

```
on = odd_numbers()  
print(on.__next__())
```

Note: Finite generator objects are exhausted after their use!

Infinite generators

Note: Just as in mathematics, generators may be infinite!

```
def odd_numbers():  
    "generator for all odd numbers"  
    k = 1  
    while True:  
        if k % 2 == 1:  
            yield k  
        k += 1
```

```
on = odd_numbers()  
print(on.__next__())
```

Note: Finite generator objects are exhausted after their use!

Generator tools

`enumerate` is used to *enumerate* a generator:

```
g = odd_numbers(10)
for i, x in enumerate(g):
    print(i,x, end=';')

# result: 0 1 ; 1 3 ; 2 5 ; 3 7 ; 4 9 ;
```

`reversed` creates a generator from a list by going backwards:

```
A = [0, 1, 2]
for elt in reversed(A):
    print(elt ,end=' ')

# result: 2 1 0
```


Generator tools

`enumerate` is used to *enumerate* a generator:

```
g = odd_numbers(10)
for i, x in enumerate(g):
    print(i, x, end=';')

# result: 0 1 ; 1 3 ; 2 5 ; 3 7 ; 4 9 ;
```

`reversed` creates a generator from a list by going backwards:

```
A = [0, 1, 2]
for elt in reversed(A):
    print(elt, end=' ')

# result: 2 1 0
```

Iterator tools

Create a generator from another generator.

```
import itertools as it
on = odd_numbers ()
some_on=it.takewhile(lambda n: n < 50 , on)
```

some_on is another generator generating all odd numbers smaller than 50.

```
import itertools as it
on = odd_numbers ()
some_on = it.islice(on , 3, 20 , 3) # start, stop index and steps
```

list(some_on) returns [7, 13, 19, 25, 31, 37]

Iterator tools

Create a generator from another generator.

```
import itertools as it
on = odd_numbers ()
some_on=it.takewhile(lambda n: n < 50 , on)
```

some_on is another generator generating all odd numbers smaller than 50.

```
import itertools as it
on = odd_numbers ()
some_on = it.islice(on , 3, 20 , 3) # start, stop index and steps
```

list(some_on) returns [7, 13, 19, 25, 31, 37]

List filling patterns

Common programming pattern:

```
L = []  
for k in range(n):  
    L.append ( some_function (k) )
```

use instead:

```
L = [some_function(k) for k in range(n)]
```

This is called list comprehension!

List filling patterns

Common programming pattern:

```
L = []  
for k in range(n):  
    L.append ( some_function (k) )
```

use instead:

```
L = [some_function(k) for k in range(n)]
```

This is called list comprehension!

List filling patterns

Common programming pattern:

```
L = []  
for k in range(n):  
    L.append ( some_function (k) )
```

use instead:

```
L = [some_function(k) for k in range(n)]
```

This is called list comprehension!

Complex List Filling Pattern

```
L = [0, 1]
for k in range(n):
    # call various functions here
    # that compute a "result"
    L.append( result )
```

Use a generator instead!

```
def result_generator(n):
    for k in range(n):
        # call various functions here
        # that compute a "result"
        yield result
```

...and if you really need a list:

```
L = list(result_generator(n)) # no append needed!
```

Complex List Filling Pattern

```
L = [0, 1]
for k in range(n):
    # call various functions here
    # that compute a "result"
    L.append( result )
```

Use a generator instead!

```
def result_generator(n):
    for k in range(n):
        # call various functions here
        # that compute a "result"
        yield result
```

...and if you really need a list:

```
L = list(result_generator(n)) # no append needed!
```


Complex List Filling Pattern

```
L = [0, 1]
for k in range(n):
    # call various functions here
    # that compute a "result"
    L.append( result )
```

Use a generator instead!

```
def result_generator(n):
    for k in range(n):
        # call various functions here
        # that compute a "result"
        yield result
```

...and if you really need a list:

```
L = list(result_generator(n)) # no append needed!
```

Generator comprehension

Just as we had list comprehension, there is also generator comprehension:

```
g = (n for n in range(1000) if not n % 100)
# a generator that generates 0, 100, 200, ...
```

The odd numbers again:

```
on = (n for n in range(1000) if n % 2)
```

Generator comprehension

Just as we had list comprehension, there is also generator comprehension:

```
g = (n for n in range(1000) if not n % 100)
# a generator that generates 0, 100, 200, ...
```

The odd numbers again:

```
on = (n for n in range(1000) if n % 2)
```

Zippping Generators

How to make one generator out of two?

```
from itertools import izip
xg = x_generator()
yg = y_generator()
for x,y in izip(xg ,yg):
    print(x,y)
```

The zipped generator stops as soon as one of the generators is exhausted.

Zippping Generators

How to make one generator out of two?

```
from itertools import izip
xg = x_generator()
yg = y_generator()
for x,y in izip(xg ,yg):
    print(x,y)
```

The zipped generator stops as soon as one of the generators is exhausted.

Further Reading



Wikipedia: Software testing.

https://en.wikipedia.org/wiki/Software_testing.

Accessed 2023.



Python pytest.

<https://pytest.org>.

Accessed 2023.



Python unittest.

<https://docs.python.org/3/library/unittest.html>.

Accessed 2023.



Python timeit.

<https://docs.python.org/3/library/timeit.html>.

Accessed 2023.



Sphinx documentation.

<https://www.sphinx-doc.org/>.

Accessed 2023.