Python Project Template Documentation

Arthur Van de Wiele

Nov 14, 2019

APP'S DOCUMENTATION:

All you need to kick-start a python project should be here, as the template includes code snippets, configuration files and makefile to spend more time on your application and less on the setting up the environment.

This small project aims at offering a personal take on formalizing python project structures, together with good development practices and tools. The main point here is setting up the most important tools a developer needs, and ensure they function out of the box so starting any new project can be done in a breeze.

CI Status

Contributors Arthur Van de Wiele

Project Links

Home: https://github.com/artiev/python-template
Doc: https://py-template.readthedocs.io

CHAPTER

ONE

TABLE OF CONTENT

1.1 Documenting your application

When documenting your application, the quick-start guide should be a one-pager document which is where I should find the absolute minimum information about your app, what its purpose is, how I can install it and how to use it. It's oriented toward a user, not a developer.

1.1.1 User Documentation

When documenting your application, the quick-start guide should be a one-pager document which is where I should find the absolute minimum information about your app, what its purpose is, how I can install it and how to use it. It's oriented toward a user, not a developer.

Brief Description

This project is dedicated to building the best template project to kick-off any python development work. It includes configuration files and libraries to automate code syntax checks, execute tests, trigger builds on https://travis-ci.org, compile code coverage locally and on https://codecov.io.

Additionally, this project also documents all the tools used, and discusses why some tools are discarded, or why some rules are enforced when others are not.

How to install

There is really no install, just go to the project Github page and fork or download the project. If you fork it, remember to send out pull-requests if you see something can be improved !

Link: https://github.com/artiev/python-template/

Main features

See *Workflow* and *Standard Tools* for a complete description of the supported features.

Screenshots

There are no screenshots of the code, feel free to dig in!

How to get help

Fly over to the project's repository on github and raise a question or report an issue. You can also contribute to the project anytime !

Link: https://github.com/artiev/python-template/

Frequently asked questions

How can I install the app?

You need a running python 3.7+ version on your system. And we recommend PIP to install the dependencies listed in *requirements.txt*. If these dependencies are matched, and make is present on your system, you can simply call:

```
> make install
```

What is your favorite python IDE?

I find PyCharm to be feature rich, and it let's me refactor my code in a breeze. Plus its code validation, auto-complete and auto-documentation feature are very responsive. On the downside, PyCharm will eat up your battery in no time.

1.1.2 Developer's Documentation

So you're building your app? Then it's time to document it. I've built up a *starter kit* for your documentation with the kind of stuff you should tackle early on. This section is more oriented toward collaboration, developer's documentation, or internal documentation in general, and will often not be published alongside the quick-start guides and user documentation.

Warning: This template automatically triggers a travis build and a documentation build whenever the code is push to GitHub. Be careful not to expose intellectual property or secret recipes on https://readthedocs.io

Navigating the codebase

Warning: Work in progress. . .

Flow diagrams

Taking the habit of documenting your application's flow control with proper diagrams will allow new-comers to your code find their way through the code base much faster. Let's take an extremely simplified flow diagram of the template's main app, which loops over an index *n* and compute the fibonacci value for that index.

This graph is built automatically during the documentation generation using an external dependency called *graphviz*. The graph image is generated based on a graph description which defined in a separated text file.

The beauty of it, is that you can maintain the graphs as easily as you maintain your code, without the need for any external tools. And the same holds true for automatically generated documentation (see *Documenting your project with Sphinx and reStructuredText*)

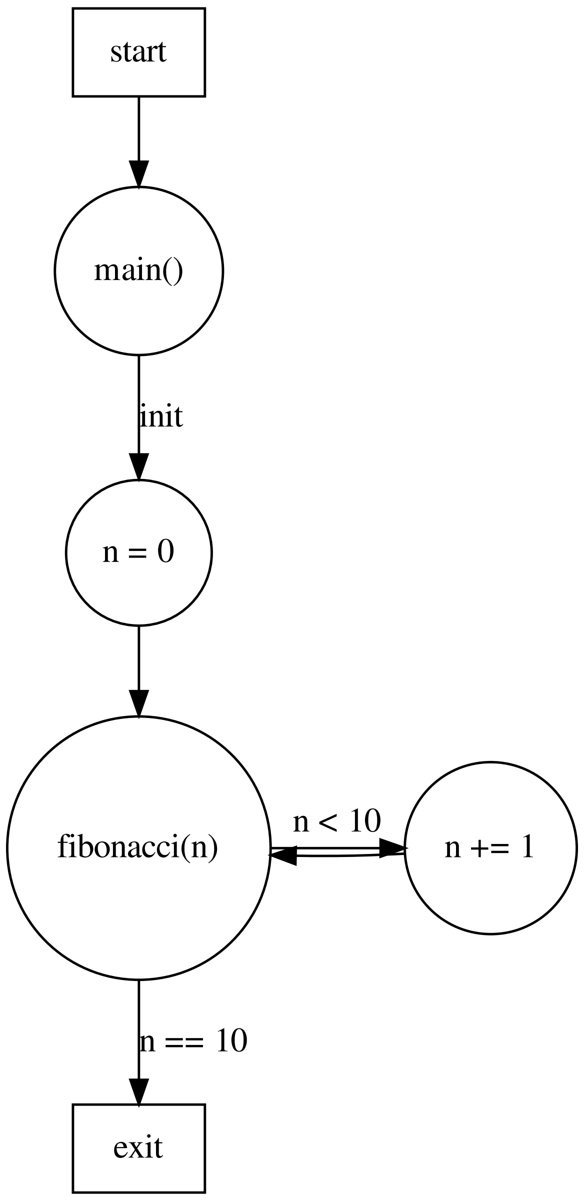


Fig. 1: Example of a state machine

```
digraph sm {
    size = "5,8"
    overlap = False
    pad = 1

    node [ shape = rectangle ]
    start [ label = "start" ]
    exit [ label = "exit" ]

    node [ shape = circle ]
    main [ label = "main()" ]
    init [ label = "n = 0" ]
    increment [ label = "n += 1" ]
    { rank = same; increment }
    fib [ label = "fibonacci(n)" ]


    start -> main
    main -> init [ label="init" ]
    init -> fib
    increment -> fib
    fib -> increment [ label = "n < 10" ]
    fib -> exit [ label = "n == 10" ]

    {
    rank = same;
    fib -> increment [ style=invis ];
    rankdir = LR;
    }
}
```

Important parts of your code

Most of the time, one function should be the entry point of your application. In this instance, I use C as an inspiration for standardizing the use of the `main.main()` function. As python file names matter - in contrast to C - the module holding the *main()* function is also called `main`.

If you are calling the python interpreter directly on a file, then you'll need to add a safeguard:

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

Otherwise, the function is usually given as a gateway, for example if using gunicorn to start a server and deliver a flask application:

```
$ gunicorn app.main:main
```

Warning: The entire project is built using Python 3.7+, and in such an example, gunicorn or any other server needs to be carefully setup to use Python 3 as - at the time of this writing - most operating systems and platforms still operate with Python 2.7 as a default.

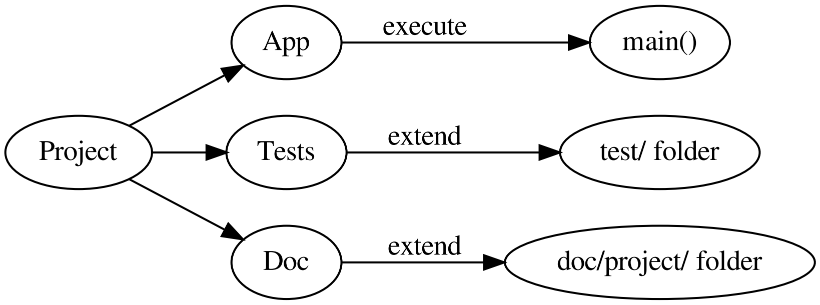


Fig. 2: Entry points.

1.2 Workflow

Having a solid and documented workflow is paramount to maintaining a clean codebase when multiple contributors are involved. In this chapter, I'm highlighting some of the conventions I adopted, while also trying to discuss the reasons leading me to these choices.

The following diagram documents how this template makes use of automation and continuous integration platforms, and where they can impact your development workflow. You will find more information about goals and configuration requirements in the *Continuous Integration* and *Standard Tools* chapters.

1.2.1 Continuous Integration

For any project where quality and maintainability is an important goal, continuous integration is the ultimate quality gate. The stronger the rules and the better the tests, the more stable your project will be throughout its entire development cycle. This chapter deals mostly with configuration of these quality gates using https://travis-ci.org and https://codecov.io platforms. Other chapters offer discussions about the reasons for all these configurations.

Warning: Work in progress…

Travis & Codecov configuration

As briefly discussed in the *Code Coverage based on Unit Tests* section, Travis is configured to run the complete regression (all tests found in the *test/* folder) and computed the coverage using the same tools as the local coverage computation.

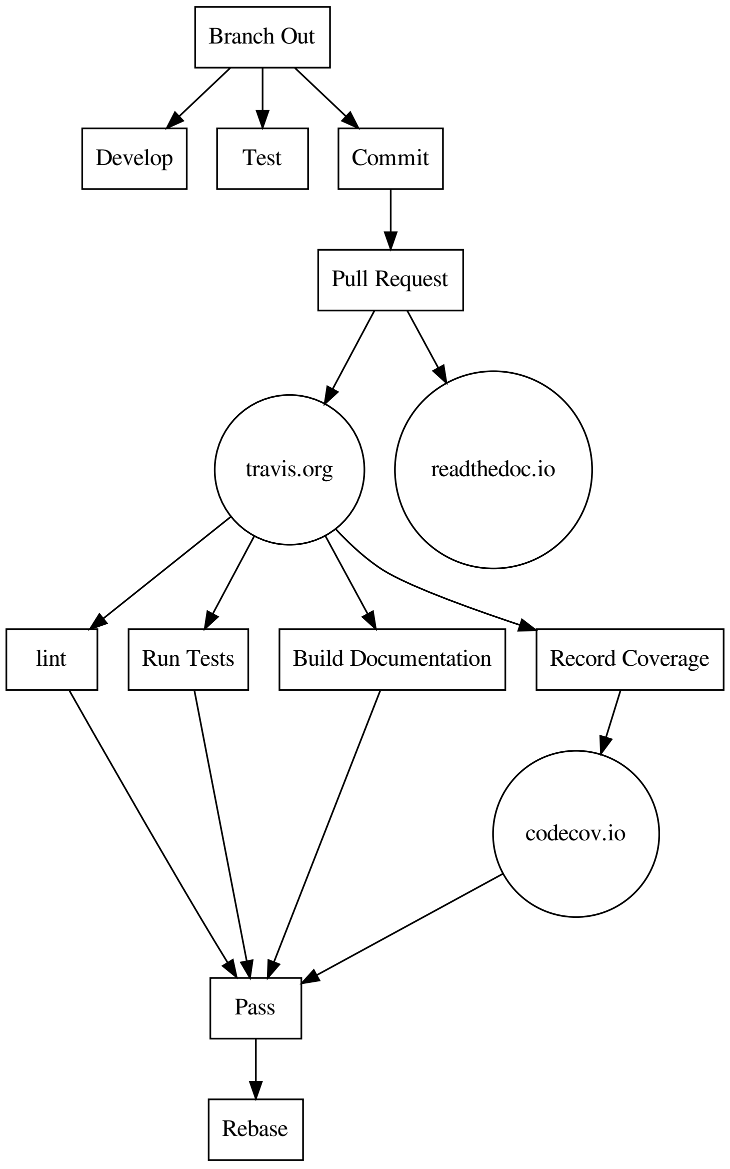


Fig. 3: High-level development flow and Quality Gates‘

The difference comes after the build on Travis is successful, and the *codecov* module kicks in, uploading the coverage statistics to codecov.io. This is achieved by a set of configuration files found at the root of the project.

Linking Travis and Codecov

Computing code coverage locally is a nice tool, but most importantly, the ability to follow the evolution of your coverage throughout the development cycles of your application is a very effective metric to know how your process is evolving, and whether or not your investment on testing in increasing or decreasing.

This is achieved through a configuration file *.codecov.yml* located at the root of the project directory:

Listing 1: .codecov.yml

```yaml
coverage:
  status:
    patch: no
    changes: no
    project:
      default: false
      app:
        paths: "app/"
        target: 75%
      tests:
        paths: "test/"
        target: 95%
      tools:
        paths: "tools/"
        target: 85%
```

And a few special directives to the *.travis.yml*:

Listing 2: .travis.yml

```yaml
language: python

python:
  - "3.7"

addons:
  apt:
    packages:
      - libenchant-dev
      - graphviz

install:
  - "pip install -r requirements.txt"
  - "pip install codecov"

script:
  - "make check PYTHON=\"python\""
  - "make coverage PYTHON=\"python\""
  - "make doc PYTHON=\"python\""

after_success:
  - "codecov"
```

1.2.2 Code Conventions

When multiple contributors are planned to work an a single project, it is important to align the code style and expectations - to some extent - and to that end, this project has pre-configured a linter using the *pylint* module. This chapter offers a few insights on some of the decisions taken during this configuration, and also about project structures in general.

Discussing disabled linter warnings

I - the template's author - follow most of the PEP8 recommendations. But there are some rules which I consider diminishing read-ability of the code, and which are hence not enforced in the linter's configuration (see *.pylintrc*).

In particular, the following conventions are disables by default:

- *bad-whitespace*
- *len-as-condition*
- *no-name-in-module*

bad-whitespace

One reason for me to disable the **bad-whitespace** warnings is the way I write functions definitions and calls, and mathematical formulae. I'd simply call it 'adding negative space' - the absence of code - to enhance readability. The **bad-whitespace** would trigger multiple warnings in the following code.

```
def add_and_multiply( a: float, b: float, c: float = 1 ) -> float:
    """
    Adds a to b, then multiplies the result by c
    """

    result = (a + b) * c
    message = '( {} + {} ) * {} = {}'.format( a, b, c, result )
    print( message )

    return result
```

But I find this much more readable than the following, despite is having the exact same functionality.

```
def add_and_multiply(a: float, b: float, c: float = 1) -> float:
    """ Adds a to b, then multiplies the result by c """
    result = (a + b) * c
    message = '( {} + {} ) * {} = {}'.format(a, b, c, result)
    print(message)
    return result
```

Granted, the code snippets differs in more ways between than just the introduction of that rule, but I will still prefer :

- **`print( message )` over** `print(message)`
- **`add_and_multiply( a: float, b: float, c: float = 1 )` over** `add_and_multiply(a: float, b: float, c: float = 1)`

len-as-condition

In the same vain as the **bad-whitespace** discussion, there is a few simplifications made in python which I consider hindering readability and clarity. Let's take the following example:

```
some_list = [1, 4, 6, 2, 1]

if list:
    print( 'list is not empty' )
```

This makes my skin crawl, what is `if list` supposed to mean ? Now let's see the explicit equivalent, which would trigger the linter:

```
some_list = [1, 4, 6, 2, 1]

if len(list) > 0:
    print( 'list is not empty' )
```

Now, whether one way is faster or more pythonic is not what I care about. Instead, I'll take readability over performance or philosophy most of the time. Because let's be honest, for real performance, write your libraries in C and compile them.

If you'd like to be thorough, then, I do agree that `if len(some_list):  pass` is neither explicit, nor pythonic, and therefore should be avoided. It's not logical to extract a boolean value by camparing it to an integer which is computed from a list, but by deactivating **len-as-condition** entirely, you open the door to such slips. I guess it's up to you to decide if you can be diligent enough.

no-name-in-module

Ok, I guess this is going to be the most controversial point, but I'm tired of the `__init__.py` files cluttering my directories. So I only use them sparsely since Python 3.3, but the linter does no always react in the best of ways (yet?) and throws me a bunch of **no-name-in-module** warnings.

Regular modules increase the risk of side effects you can purposely - or not - introduce in libraries. Now, not letting Python know what module is a module only works if your import scheme are consistent with an certain approach:

- Use absolute imports for all your custom libraries
- Only allow importing an entire module for the standard libraries
- Import only the resources you need from your app/libraries

This seems arbitrary, but in practice, there are quite a few things happening (and NOT happening). Let's have a look:

Dependencies You always highlight specific dependencies:

```
from app.client import CREDENTIALS_ERROR
```

Instead of:

```
import app
```

It has the added benefit to avoid executing code you don't know about, which brings me to the next point.

Execution Side effects are the bane of any collaborative software developer's existence. Now when importing a module with `import app`, Python will implicitly execute the `__init__.py` file and a bunch more things.

```
# module/__init__.py
# [...]
LOGGER = logging.getLogger('my_logger')
HANDLER = RotatingFileHandler('my_log.log', maxBytes=2000,␣
↪backupCount=10)
LOGGER.addHandler(handler)
# [...]

# some other file
import module

# and boom, you've accessed the filesystem to create a log file.
# Ok, granted, the 'module' was crap in the first place ^^
```

Now, for most people, this being an empty file, it does not really matter. But I have seen (and on occasion even used) `__init__.py` files to restrict the import scopes of a module by manually overwriting the `__all__` attribute, in other words, redefining a module's exposed functions and objects.

```
# __init__.py

from .submodule import public_function
from .defines import PUBLIC_SET
from .lib.oop import PublicObject

__all__ = ['public_function', 'PUBLIC_SET', 'PublicObject']
```

You guess where I'm going with this ? Well, I'm being supplied a library and was told to only use the 'public' interface, I'm looking into the code, and find the perfect function, so I import my module, and call 'module .function' somewhere down, and… and nothing, it fails because `__all__` did not expose that particular function.

Don't get me wrong, it's a very nice way to differentiate 'public' and 'private' functions or objects for third parties, but it contradicts my approach to software development: code should only do what it's supposed to do. And in Python, everything is public, so don't break expectations.

Clutter Last but not least, I do my best to divide my project's codes in small and contained libraries. You know, to keep things clean and modular. So I have many folders and files, and I'm working in the console, so I call `tree`:

```
.
├── __init__.py
├── lib
│   ├── bells
│   │   └── __init__.py
│   ├── colors
│   │   └── __init__.py
│   ├── console
│   │   └── __init__.py
│   └── __init__.py
└── module
    ├── client
    │   └── __init__.py
    ├── core
    │   ├── defines
    │   │   └── __init__.py
    │   └── __init__.py
    │
```

(continues on next page)

(continued from previous page)

```
    └── __init__.py
```

> Well, I can't describe that feeling. But that's where Python 3.3+ came handy, by introducing the concept of `namespace` to complement the `regular` package definition, and suffice to say, it suits my needs. And also offer a few interesting options for the future.

And that's why most of my projects only have a limited amount of `__init__.py` files, simply because most of the time I treat folders as namespaces rather than entire modules.

> A namespace package is a composite of various portions, where each portion contributes a subpackage to the parent package. Portions may reside in different locations on the file system. Portions may also be found in zip files, on the network, or anywhere else that Python searches during import. Namespace packages may or may not correspond directly to objects on the file system; they may be virtual modules that have no concrete representation.
>
> Namespace packages do not use an ordinary list for their __path__ attribute. They instead use a custom iterable type which will automatically perform a new search for package portions on the next import attempt within that package if the path of their parent package (or sys.path for a top level package) changes.
>
> With namespace packages, there is no parent/__init__.py file. In fact, there may be multiple parent directories found during import search, where each one is provided by a different portion. Thus parent/one may not be physically located next to parent/two. In this case, Python will create a namespace package for the top-level parent package whenever it or one of its subpackages is imported.

See https://www.python.org/dev/peps/pep-0420/ for more details.

Warning: Work in progress…

1.2.3 Project Structure

It's important to spend a minimum amount of efforts setting up a project properly, especially if many contributors are expected to navigate the project. This section is presenting a few arguments about the overall project structure, while staying clear of the application's architecture, which will highly depend on the application itself and your architectural decisions.

Use namespaces

Warning: Work in progress…

1.3 Standard Tools

This projects uses the *pylint* and *coverage* modules to ensure a decent level of code quality. Both module are configured through their rc files at the root of the project folder. The test coverage is itself dependent on the *pytest* module. Continuous integration through https://travis-ci.org and https://codecov.io is also pre-configured and discussed in another chapter (*Travis & Codecov configuration*). The documentation itself is built using *sphinx* with the *autodoc* extension. A direct connection to https://readthedocs.io is ready to be established.

1.3.1 Code Coverage based on Unit Tests

This template relies on the use of the *coverage* module in conjunction with the *unittest*/*pytest* modules to automatically compile code coverage reports.

Underlined here is the importance of writing tests which not only maximize coverage, but also prevent breaking applications later during refactoring and maintenance stages of a project's lifecycle.

Warning: Code coverage alone is not a good indicator of project health. But it's a start.

Coverage Makefile Targets

The makefile offer the *test* and the *coverage* targets. The former collects and run unit tests while gathering coverage statistics. The second target calls *test* then generates an html report, located under *reports/coverage/* (as well as printing out stats in the console).

Note: For more info on code coverage in python, see https://coverage.readthedocs.io/en/latest/#quick-start

For computing the current coverage of your working directory, simply run:

```
make coverage # depends on clean and test targets
```

Which will runt all the tests, and will output the coverage file by file:

```
Name                                        Stmts   Miss Branch BrPart  Cover   Missing
---------------------------------------------------------------------------------------
app/main.py                                    16      1     10      1    92%   62, 61->
↪62
test/test_app/test_main.py                     12      0      0      0   100%
test/test_methodology/test_examples.py         12      1      0      0    92%   49
test/test_tools/test_path_explorer.py          30      0      4      0   100%
test/test_tools/test_pretty_console.py         38      1      2      0    98%   86
tools/path_explorer.py                         27      0     18      0   100%
tools/pretty_console.py                        35      2      8      1    93%   15, 100,␣
↪99->100
---------------------------------------------------------------------------------------
TOTAL                                         170      5     42      2    97%
```

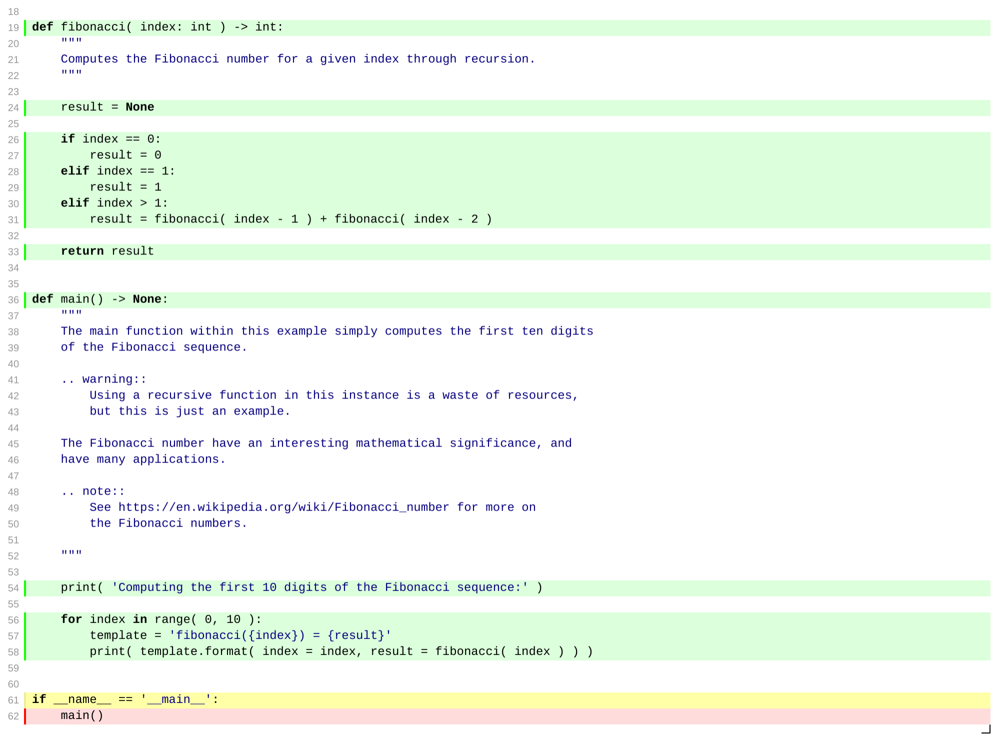
Additionally, the coverage routine also generates an HTML report (see *reports/coverage/index.html*) which has the added benefit of highlighting the code statements executed and skipped.

Note: Code coverage can be easily integrated with travis and codecov using the *codecov* module.

Uploading to Codecov.io

Computing code coverage locally is a nice tool, but most importantly, the ability to follow the evolution of your coverage throughout the development cycles of your application is a very effective metric to know how your process is evolving, and whether or not your investment on testing in increasing or decreasing.

This is achieved through a configuration file *.codecov.yml* located at the root of the project directory:



```python

def fibonacci( index: int ) -> int:
    """
    Computes the Fibonacci number for a given index through recursion.
    """

    result = None

    if index == 0:
        result = 0
    elif index == 1:
        result = 1
    elif index > 1:
        result = fibonacci( index - 1 ) + fibonacci( index - 2 )

    return result


def main() -> None:
    """
    The main function within this example simply computes the first ten digits
    of the Fibonacci sequence.

    .. warning::
        Using a recursive function in this instance is a waste of resources,
        but this is just an example.

    The Fibonacci number have an interesting mathematical significance, and
    have many applications.

    .. note::
        See https://en.wikipedia.org/wiki/Fibonacci_number for more on
        the Fibonacci numbers.

    """

    print( 'Computing the first 10 digits of the Fibonacci sequence:' )

    for index in range( 0, 10 ):
        template = 'fibonacci({index}) = {result}'
        print( template.format( index = index, result = fibonacci( index ) ) )


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

Fig. 4: Coverage report highlighting

Listing 3: .codecov.yml

```yaml
coverage:
  status:
    patch: no
    changes: no
    project:
      default: false
      app:
        paths: "app/"
        target: 75%
      tests:
        paths: "test/"
        target: 95%
      tools:
        paths: "tools/"
        target: 85%
```

And a few special directives to the *.travis.yml*:

Listing 4: .travis.yml

```yaml
language: python

python:
  - "3.7"

addons:
```

(continues on next page)

(continued from previous page)

```
  apt:
    packages:
      - libenchant-dev
      - graphviz

install:
  - "pip install -r requirements.txt"
  - "pip install codecov"

script:
  - "make check PYTHON=\"python\""
  - "make coverage PYTHON=\"python\""
  - "make doc PYTHON=\"python\""

after_success:
  - "codecov"
```

Therefore, the continuous integration build is also setup to upload code coverage statistics to Codecov.io automatically if the build was successful. It has the added benefit of offering much better visualization of your coverage.

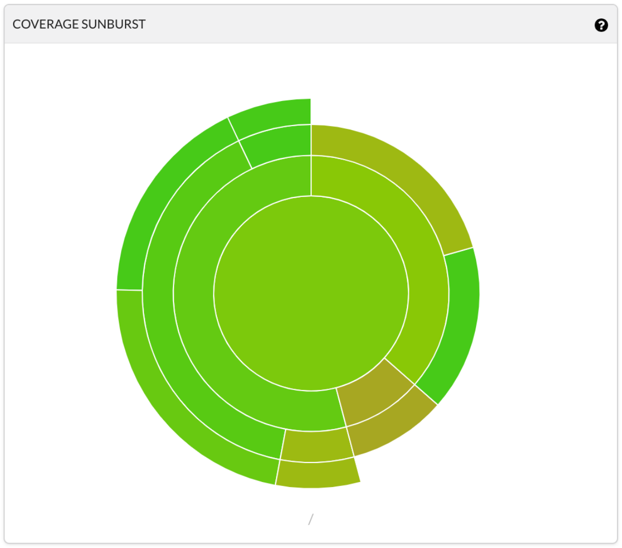


Fig. 5: Coverage 'Pie' for the complete project.

Note: More on Codecov's graphs at https://docs.codecov.io/docs/graphs#section-sunburst

Codecoverage as a Check in GitHub

One of the main advantages to this approach is that you can extrapolate 'quality gates' directly from the coverage targets, which will define whether of not your CI status is a pass or a fail, for example while preparing a new pull request (if you have linked Travis and Codecov to your github project).

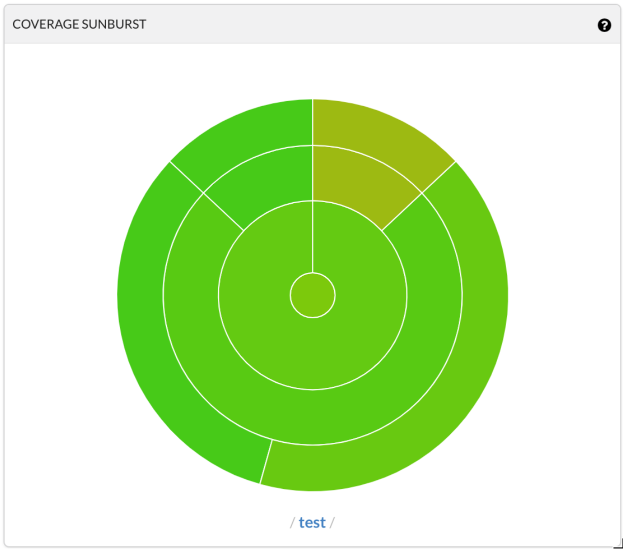


Fig. 6: Coverage 'Pie' for the test/ folder.

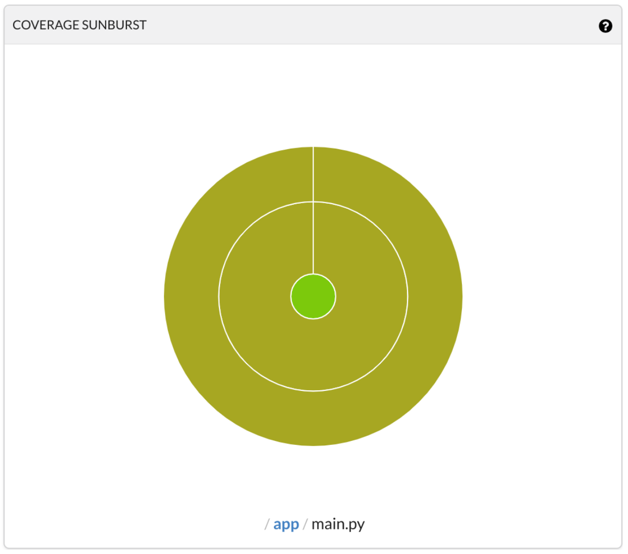
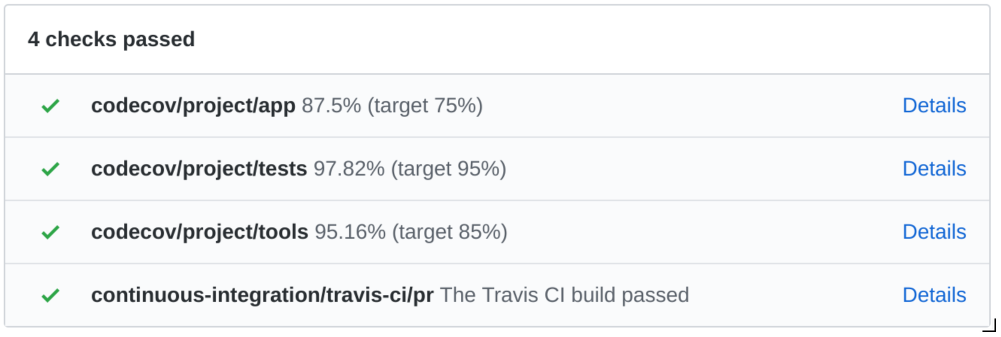


Fig. 7: Coverage 'Pie' for the app folder.



4 checks passed

✓ **codecov/project/app** 87.5% (target 75%) Details

✓ **codecov/project/tests** 97.82% (target 95%) Details

✓ **codecov/project/tools** 95.16% (target 85%) Details

✓ **continuous-integration/travis-ci/pr** The Travis CI build passed Details

Fig. 8: Recorded checks status after a merged PR.

Discussion on Coverage Targets

There is no right or wrong approach to code coverage. Generally speaking, the obvious rule is **more is better** but you should be weary of not confusing code coverage for functional testing. It is but a metric informing you about how much of the code was executed once, but does not necessarily prove the stability, robustness or overall quality of an application or module.

So let's discuss coverage targets with that in mind. What is a coverage target? Well, it's a local or global coverage limit you deem acceptable for a project or portions of it. It implies that a lower score than the expected target should constitute a fail in regards to continuous integration.

Let's dive in this projects's default targets by looking at the Codecov configuration file we looked at earlier:

```
project:
    # [...]
    app:
        paths: "app/"
        target: 75%
    tests:
        paths: "test/"
        target: 95%
    tools:
        paths: "tools/"
        target: 85%
```

In essence, this configuration defines different targets for different sub-folders. Behind these numbers lie a few important concepts:

The app

It's likely a **waste of time to try and cover 100%** of your app - think return on investment - but considering good development practices, and a drive for quality anything below 60% might be considered dangerous on the long game.Now, of course, it will depend on so much more than than: the application life cycle, the expected reuse of modules, shared dependencies, time constrains, etc…

All critical components of your app should be tested, but some tests might be too complicated to automate. I find between 50% and 80% to be a sweet spot for trusting your continuous integration will catch a few bullets for you.

The tests

The **testing code itself should always be close to 100%** coverage except if you're expecting to skip some tests (this is the case in this project). Hence it's good practice to separate that coverage from your other coverage metrics, otherwise it will artificially bump your coverage scores up even if you have 1% of your app actually tested.

The 100% coverage has a lot to do with approaches such as Test Driven Development - which is a good thing - but in my opinion, passing tests is not the only tool in your arsenal. I cover a few concepts about testing in the *Unit Tests* section, which bring me to the following conclusion:

It might not be in your best interest to expect 100% test coverage due to skipped tests and expected fails. This is true here, and the reason why the *test/* coverage target is only set to **95%**.

The tools

There is no reason you should skip testing your environment as well as your application. After all, **breaking your tools is likely to have serious side effects** on your CI and other quality gates.

Yet again, it all boils down to the return on investment. And your environment being a building block for all the effort spent on the application itself, it's likely that that time is a good investment.

Warning: Work in progress…

1.3.2 Linter & Code Quality

The linting process checks all *imported* code from a given starting point (entry file) for common PEP8 rules (not all, see conventions). In the current configuration, the linting process also applies grammar check to all comments on top of the normal code.

Note: For more info on linting in python, see https://pylint.readthedocs.io/en/latest/tutorial.html.

Warning: Work in progress…

1.3.3 Unit Tests

The unit testing scheme of this template is designed to offer the user the possibility to not only create unit tests within the *test/* folder, but also the option to separate unit tests in what I will call *collections*.

The reasoning behind it is that not 100% of the project needs to be tested during coding sessions, as opposed to continuous integration regressions, hence the ability for any contributor to design their own test collections.

When computing code-coverage, the entry point of is always the complete collection of tests located in the *test/* folder. Adding new tests to the collection is as simple as creating a new file within that directory structure.

Note: For more info on unittesting in python, see https://docs.python.org/3/library/unittest.html

Deep dive on the test results

As mentioned previously, usual coverage of your tests should be around 100%, which has a lot to do with approaches such as Test Driven Development - which is a good thing - but in my opinion, passing tests is not the only tool in your arsenal. In fact, a test can have quite a few different outcomes, which all have various degrees of severity, but let's focus on these for now:

- passed
- failed
- skipped
- expected failed
- error

Now you're obviously familiar with the first two, but let's talk about the other three, and why these can be very important.

Skipped

There are a few reasons a test might or should be skipped, some at the tester's discretion and some others due to the level of automation the project relies on. To cite only a few:

- Using **parametrized regressions** can help to reduce test time to use the same test suite for different level of regressions, one common approach to this is skipping some tests.
- If a dependency fails during the current test sequence, and you have specified these dependencies within your tests, then for the sake of clarity, you should skip some tests automatically. More about this in the chapter on unit testing.
- Sometimes, introducing tests long before a modules is ready is the best way to define an API, but it can be hard to maintain a quality gate with these tests if not for purposefully skipping them until the modules are ready.

Expected Fails

Sometimes, having all your tests passing is not the best way to prove a library's proper behaviour. Let's take a few concrete examples:

- Writing tests is often a great source of **documentation** for the next developer. To that end, writing failing test can be an effective way of signaling how not to do something.
- Randomizing **test vectors** to maximize the impact of a single test - running one sequence of tests with an arrays of possible inputs - if a fantastic way to attack stress a module, but sometimes, not all input combinations is supposed to work, and for these combination, specifying an expected fail flag is mandatory.

Errors

As briefly mentioned while talking about coverage targets, your tools and development environment is as susceptible to breakage as your application. This is what differentiate an error from a fail: the error seem to happen within your test framework / environment rather than while executing a specific test. Now - granted - there is a fine line between and error and a fail, and sometimes one can be mistakenly confused with the other, but the difference still exists.

Warning: Work in progress…

1.3.4 Documenting your project with Sphinx and reStructuredText

This project is mixing static and dynamic documentation using the *sphinx* module to generate html documentation. The project is also configured to trigger a complete build of the HTML documentation on https://readthedocs.io after every successful merge into the *master* branch.

1.4 Custom Tools

Unfortunately, sometimes the standard way is not always the most user-friendly, or the optimal process. So despite generally advising against it, I've collected a few tools of my own design to help work out some specific problems I could not solve otherwise.

This template is built upon as many standardized libraries and modules in order to provide stability and state-of-the-art features relying on the great work of other projects.

Unfortunately, sometimes a subset of features might be missing, or is not user friendly enough, and investing some time into building the right tool for the job is worth it. This is the case for the following scripts/tools, which are not standard.

1.4.1 Python Project Cleaner

Introduction

As the name suggests, this collection of function and scripts helps you keep a clean project tree. For me, this allows me to work in the console better, by letting me clean all python cache files and folders in a single command, hence keeping the directory tree pristine.

For example, see the file tree below. I could configure my system so that folders starting with __ would be hidden, but it makes no sense.

```
$ tree t*
├── test
│   ├── pytest.ini
│   ├── test_app
│   │   ├── __pycache__
│   │   │   └── test_main.cpython-37-pytest-5.0.1.pyc
│   │   └── test_main.py
│   ├── test_methodology
│   │   ├── __pycache__
│   │   │   └── test_examples.cpython-37-pytest-5.0.1.pyc
│   │   └── test_examples.py
│   └── test_tools
│       ├── __pycache__
│       │   ├── test_path_explorer.cpython-37-pytest-5.0.1.pyc
│       │   └── test_pretty_console.cpython-37-pytest-5.0.1.pyc
│       ├── test_path_explorer.py
│       └── test_pretty_console.py
└── tools
    ├── path_cleaner.py
    ├── path_explorer.py
    ├── pretty_console.py
    ├── __pycache__
    │   ├── path_explorer.cpython-37.pyc
    │   └── pretty_console.cpython-37.pyc
    └── pycharm_code_style.xml
```

That's where the cleaning routine helps. It navigates through the entire project structure, and locates all the cache files and folders, then deletes them all. So that navigating the project looks more like:

```
$ tree t*
├── test
│   ├── pytest.ini
│   ├── test_app
│   │   └── test_main.py
│   ├── test_methodology
│   │   └── test_examples.py
│   └── test_tools
│       ├── test_path_explorer.py
│       └── test_pretty_console.py
└── tools
    ├── path_cleaner.py
```

(continues on next page)

(continued from previous page)

```
    ├── path_explorer.py
    ├── pretty_console.py
    ├── __pycache__
    │   └── pretty_console.cpython-37.pyc
    └── pycharm_code_style.xml
```

Note: As you might have noticed, currently executing pre-compiled files are not deleted, like it is the case for *pretty_console.cpython-37.pyc* being used by the makefile during the cleaning process.

How to clean the project tree

Generally speaking, most of the template's tools are exposed through the makefile. In this instance, simply run:

```
$ make clean
```

Now, this does a few things on top of cleaning up the cache, so you can always call the script separately. The cleaner tool uses the *click* library to expose a command line interface. For example, as called by the makefile's *clean* target:

```
$   python -m tools.path_cleaner clear-all-cache <START_DIR>
```

You can always get an up-to-date feature description from calling:

```
$ python -m tools.path_cleaner --help
Usage: path_cleaner.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  clear-all-cache               Entry point for cleaning files and then...
  clear-cache-files             Walk through the given path, looking for...
  clear-cache-folders-if-empty  Walk through the given path, looking for...
```

Note: More about the *click* library at https://click.palletsprojects.com/en/

Developer's documentation

This library relies on the path_explorer tool to find and remove python cache files and folders. It offers a command line interface and should not be called as a python module.

`path_cleaner.`**`clear_all_cache`**`(`**args*, ***kwargs*`)`
Entry point for cleaning files and then folders in one CLI call.

`path_cleaner.`**`clear_cache_files`**`(`**args*, ***kwargs*`)`
Walk through the given path, looking for .py[cod] cache files, and tries to delete all of them.

`path_cleaner.`**`clear_cache_folders_if_empty`**`(`**args*, ***kwargs*`)`
Walk through the given path, looking for __pycache__ folders, and tries to delete all of them, but only if they are empty.

The path explorer module is designed to help locate and filter and folders files within the directory structure. It is not restricted to the local project, so expose/use with caution.

`path_explorer.`**`search_path_for_directories_and_files`**(*path: str*, *omit_python_cache: bool = True) -> (<class 'list'>, <class 'list'>*)

Creates two lists containing all found directories, and all files starting from a given path.

`path_explorer.`**`search_path_for_directories_with_partial_match`**(*path: str*, *partial: str*) → list

Gets results from the path explorer, and does a simple string comparison on the complete absolute path for a partial match to the provided string.

`path_explorer.`**`search_path_for_files_with_extensions`**(*path: str*, *extensions: set*) → list

Gets results for the path explorer, and filters out only the files with an extension matching any of the list provided.

1.5 app

1.5.1 main module

The main.py file is currently used for demonstration only within this project, in order to offer a standard entry point to the whole template.

The structure of the app itself is not predefined, and different apps will present different structures, hence the freedom.

In this example, the main file is supposed to be called directly from an interpreter which will call the main() function.

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

`main.`**`fibonacci`**(*index: int*) → int

Computes the Fibonacci number for a given index through recursion.

`main.`**`main`**() → None

The main function within this example simply computes the first ten digits of the Fibonacci sequence.

Warning: Using a recursive function in this instance is a waste of resources, but this is just an example.

The Fibonacci number have an interesting mathematical significance, and have many applications.

Note: See https://en.wikipedia.org/wiki/Fibonacci_number for more on the Fibonacci numbers.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX