Python Package Template Documentation

Release 1.0.0

Chris Ostrouchov

Apr 23, 2018

Contents:

This is an opinionated attempt to document how I deploy a python application with documentation, testing, pypi, and continuous deployment. This project will be updated as I change my python development practices. Number one this is a learning experience.

This project is a python package itself and full documentation is available on readthedocs. Each of the steps below includes a link to the section in the documentation.

1. setup a bare python package with git repo (`setup.py`, `README.md`, `.gitignore`, `<package>`)
2. setup pypi deployment with git tags `vX.X.X`
3. setup conda deployment with git tags `vX.X.X`
4. setup docker deployment with git tags `vX.X.X`
5. setup testing on each commit with `pytest`
6. setup documentation with `sphinx` on readthedocs and self hosted
7. setup command line interface with `argparse`
8. setup badges for README.md

CHAPTER 1

Packaging

In this section I will talk about how create a simple python package that can be installed using `python setup.py install`. These are the basics sharing your package with other users. In order to get your package to install with `pip` you will need to complete the steps in this guide and *PyPi*. The reason is that this guide only shows how to let someone install your package if they have the package directory on their machine.

This guide was taken from several resources:

- setup.py reference documentation
- pypi sample project
- kennethreitz setup.py
- pypi supports markdown

Is anyone else troubled by the fact that so many links are necissary for simple python package development?!

Overview of typical package

    README.md CHANGELOG.md LICENSE.md setup.py <package>/

        __init__.py

1.1 setup.py

The most important file is the `setup.py` file. All required and optional fields are given `<required>` and `<optional>` respectively.

```
from setuptools import setup, find_packages
from codecs import open
from os import path

here = path.abspath(path.dirname(__file__))

# Get the long description from the README file
with open(path.join(here, 'README.md'), encoding='utf-8') as f:
```

```
    long_description = f.read()

setup(
    name='<required>',
    version='<required>',
    description='<required>',
    long_description=long_description,
    long_description_content_type="text/markdown",
    url='<optional>',
    author='<optional>',
    author_email='<optional>',
    license='<optional>',
    classifiers=[
        # Trove classifiers
        # Full list: https://pypi.python.org/pypi?%3Aaction=list_classifiers
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python',
        'Programming Language :: Python :: 3.6',
        'Programming Language :: Python :: Implementation :: CPython'
    ],
    keywords='<optional>',
    packages=find_packages(exclude=['docs', 'tests']),
    # setuptools > 38.6.0 needed for markdown README.md
    setup_requires=['setuptools>=38.6.0'],
)
```

While setuptools docs detail each option. I still needed some of the keyworks in more plain english. This is not an exhaustive list so make sure to reference the setuptools docs.

name the name of package on pypi and when listed in pip. This is not the name of the package that you import via python. The name of the import will always be the name of the package directory for example `pypkgtemp`.

version make sure that the version numbers when pushing to pypi are unique. Also best to follow semantic versioning.

description keep it short and describe your package

long_description make sure that you have created a README.md file in the project directory. Why use a README.md instead of README.rst? It's simple, Github, Bitbucket, Gitlab, etc. all will display a README.md as the homepage.

url link to git repo url

author give yourself credit!

author_email nobody should really use this address to contact you about the package

license need help choosing a license? use choosealicense

classifiers one day would be nice to know why they are important. list of available tags.

keywords will help with searching for package on pypi

packages which packages to include in python packaging. using `find_packages` is very helpful.

setup_requires list of packages required for setup. Note that versioning uses environment markers.

1.2 LICENSE.md

If you do not include a license it is by default copyrighted and unable to be used by others. This is why it is so important to give your work a license. A great resource for this is choosealicense.com.

1.3 README.md

A README is the first document someone sees when they visit your project make it an inviting document with an overview of everthing the programmer needs.

1.4 CHANGELOG.md

A changelog is something that I did not really adopt in my projects until I started forgeting what I had done in the past week. I git log is not designed for this! Some great advice can be found in Keep a CHANGELOG. Their motto is "Don't let your friends dump git logs into CHANGELOGs™"

At this point you have a simple python package setup! Obviously the readme, changelog, and license are all optional but HIGHLY recommended. Next we will share our package with the whole world through continuous deployment (**:doc:'pypi'_**).

CHAPTER 2

PyPi

PyPi otherwise known as the cheeseshop is the packaging repository for python. PyPi has been going through some great changes recently including a new UI and not requiring registering new projects. PyPi deployment can seem daunting. Twine is your best friend.

PyPi has two repositories. The testing repository and main repository. when you are trying out deploying packages I would advise starting with testing. Older guides for using PyPi will state that you need to pre-register a package. This is no longer the case.

First you will need to create a pypi account. The testing and production repositories require separate accounts. Look for the Register link in the top right of the site. After creating your account keep track of the `username` and `password`.

From this point we have everything needed to do a simple manual deployment to pypi. If you dont want to submit to the testing repository remove `--repository-url https://test.pypi.org/legacy/`. You will be prompted for your username and password.

1. `pip install twine`
2. `python setup.py sdist bdist_wheel`
3. `twine upload --repository-url https://test.pypi.org/legacy/ dist/*`

This is easy! But one issue is that this process is not automated. Really we would like anytime that we create a new release on git that it is pushed to pypi. This is where Gitlab comes to the rescue.

Gitlab has a continuous deployment and continuous integration pipeline that is free to use. This will only work for code that is stored in a Gitlab repo.

Add the following to a `.gitlab-ci.yml` file in the root of your project. The reason that there are two twine upload steps is because there is currently a flaw in the markdown processing (issue). Hopefully this is fixed soon.

2.1 .gitlab-ci.yml

```
variables:
  TWINE_USERNAME: SECURE
  TWINE_PASSWORD: SECURE
```

```
  TWINE_REPOSITORY_URL: https://test.pypi.org/legacy/

stages:
 - deploy

deploy:
  image: python:3.6
  stage: deploy
  script:
    - pip install -U twine setuptools
    - pip list
    - python setup.py sdist bdist_wheel
    - twine upload dist/*.tar.gz
    - twine upload dist/*.whl
  only:
    - /^v\d+\.\d+\.\d+([abc]\d*)?$/  # PEP-440 compliant version (tags)
```

Additionally `settings->CI/CD->Secret variables` the environment variables `TWINE_PASSWORD` and `TWINE_USERNAME` need to be set. At this point whenever a git tag of the form `vX.Y.Z` is pushed to Gitlab a new version will be pushed to PyPi. Make sure that your git tags match the version number! PyPi does not allow you to change a currently existing version of your project. This is a good thing since we should all do our best to follow semantic versioning.

Once you would like to deploy to the main PyPi repository change `TWINE_REPOSITORY_URL` to `https://upload.pypi.org/legacy/`.

So you now have a package that can be shared with the entire world! But you have no testing... the next section *Testing* will show you how to include testing via `pytest`.

CHAPTER 3

Conda

Conda is an alternative package manger to PyPi. It comes with many features that PyPi packaging does not handle well such as including compiled libraries and c dependencies.

While traditional Python packages are stored in pypi.org conda python packages are stored at anaconda.org. These steps do not require that you have already deployed a package to PyPi.

First create an account through *https://anaconda.org <https://anaconda.org>*. Unlike PyPi there is no test repo to submit your package to. Anaconda takes a different philosophy where each user has a collection of packages and jupyter notebooks in their repo. The approach I will show you does not require that you have `conda` installed on your machine. If you would like to experiment with the build tool I would recommend pulling the continuum `conda build` docker container continuumio/miniconda3. The default *continuumio/anaconda3* docker environment is over 3.5 GB unzipped. Why are the continuum docker containers so large?

```
docker pull continuumio/miniconda3
docker run -i -t continuumio/miniconda3 /bin/bash
```

Once you start the docker container you can do the following steps for package deployment to conda. These steps will be automated later with a Gitlab build script. In order to upload packages you will either need to login to your account via `anaconda login` or create an account token will all account access. I would recommend creating an account token so that you can revoke access at any time. To create an account token go to `settings->access` on anaconda.org when you are logged in.

1. *conda install anaconda-client setuptools conda-build -y*
2. *python setup.py bdist_conda*
3. *anaconda -t $ANACONDA_TOKEN upload -u $ANACONDA_USERNAME /opt/conda/conda-bld/linux-64/<package>-<version>-<pyversion>.tar.bz2*

The first step ensures that all packages are the right version and we have the command line anaconda tool. Anaconda has it hidden in their documentation that they have a convenient build tool for python packages that does not require a recipe. When running in a conda environment they have overridden setuptools to include *bdist_conda* for building conda packages. The build command will build the package, run tests, and check that each command created exits. After your package is built you can now upload to conda. If you are building within a docker container chances

are that their is only one conda build so you can shorten the upload command to *anaconda upload /opt/conda/conda-bld/linux-64/<package>*.tar.bz2*. Otherwise you will have to chose the build that is provided at the end of the `python setup.py bdist_conda` output.

From some of my initial tests I was surprised that many packages available on PyPi are not available on *conda* and thus made the builds fail. These errors are most likely due to me know understanding the conda tools well. If your build succeeded you should see the package listed on *https://anaconda.org/<username>*.

Since we are all about automation lets make this process automatic on Gitlab!

```
variables:
  TWINE_USERNAME: SECURE
  TWINE_PASSWORD: SECURE
  TWINE_REPOSITORY_URL: https://test.pypi.org/legacy/
  ANACONDA_USERNAME: SECURE
  ANACONDA_TOKEN: SECURE

stages:
 - deploy

deploy_conda:
  image: continuumio/miniconda3:latest
  stage: deploy
  script:
    - conda install anaconda-client setuptools conda-build -y
    - python setup.py bdist_conda
    - anaconda -t $ANACONDA_TOKEN upload -u $ANACONDA_USERNAME /opt/conda/conda-bld/
↪linux-64/pypkgtemp*.tar.bz2
  only:
    - /^v\d+\.\d+\.\d+([abc]\d*)?$/  # PEP-440 compliant version (tags)
```

CHAPTER 4

Docker

While PyPi and Conda are great ways to distribute python applications to python developers. We would like to have an easier way to distribute applications to linux and OSX users. Containers are a great way of achieving this. For this we will use docker for our builds with a `Dockerfile`. Explaining dockerfiles are outside of the scope of this documentation but here is a general build template for Docker. You will want to put the docker file in the root of your project.

```
FROM python:3.6-slim
MAINTAINER Chris Ostrouchov

ARG VERSION=v1.1.0
ARG USERNAME=costrouc
ARG PROJECT=python-package-template

# Download package, install package, no cache
RUN pip install --no-cache-dir https://gitlab.com/$USERNAME/$PROJECT/repository/
↪$VERSION/archive.tar.gz

ENTRYPOINT ["helloworld"]
CMD ["fizzbuzz", "-n", "10"]
```

Lets explain some of the settings. `FROM` describes the docker container that we derive from. In this case starting with base python is a good start. `MAINTAINER` is exactly what it sounds like it is an easy way to declare the package maintainer. Since dockerfiles work in layers we need to do all our work in one run command to reduce size. Luckily if we have built our python package correctly the run step should be very simple using *pip*. Finally `ENTRYPOINT` and `CMD` set the default command and arguments respectively. Once you have a template you have many choices on where to share your docker container. I will show here how to upload your container to dockerhub. First create an account at docker hub and signup. Then create a repository with your desired repository name. Then follow these simple steps to build an image and push to docker hub.

1. `docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD`
2. `docker build -t $USERNAME/$PACKAGE:$VERSION --build-arg VERSION=$VERSION .`
3. `docker push $USERNAME/$PACKAGE:$VERSION`

It really is as simple as that!

4.1 Gitlab Registry + Docker Hub

Now we would like to automate this with gitlab to deploy our container to gitlab registries and docker hub. Here is the additions to `.gitlab-ci.yml`.

```
 variables:
    DOCKER_PASSWORD: SECURE
    DOCKER_USERNAME: SECURE

stages:
  - test
  - deploy

 deploy_docker:
   image: docker:git
   stage: deploy
   services:
     - docker:dind
   script:
     - docker build -t python-package-template:$CI_COMMIT_TAG --build-arg VERSION=$CI_
↪COMMIT_TAG .
     # push to dockerhub
     - docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD
     - docker tag python-package-template:$CI_COMMIT_TAG costrouc/python-package-
↪template:$CI_COMMIT_TAG
     - docker tag python-package-template:$CI_COMMIT_TAG costrouc/python-package-
↪template:latest
     - docker push costrouc/python-package-template:$CI_COMMIT_TAG
     - docker push costrouc/python-package-template:latest
     # push to gitlab registry
     - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN registry.gitlab.com
     - docker tag python-package-template:$CI_COMMIT_TAG registry.gitlab.com/costrouc/
↪python-package-template:$CI_COMMIT_TAG
     - docker tag python-package-template:$CI_COMMIT_TAG registry.gitlab.com/costrouc/
↪python-package-template:latest
     - docker push registry.gitlab.com/costrouc/python-package-template:$CI_COMMIT_TAG
     - docker push registry.gitlab.com/costrouc/python-package-template:latest
   only:
     - /^v\d+\.\d+\.\d+([abc]\d*)?$/  # PEP-440 compliant version (tags)
```

With this you should now be able to deploy to both gitlab and docker! Of course like before in PyPi and Conda you will need to add your secret environment variables to the gitlab CI/CD.

4.2 Docker Image Size

If you visit the official python docker repository there are many many choices for base images to start from for each version. To simplify your choices you need to pick a python version (2.7, 3.3, 3.5, 3.6, etc.), whether the image is based on a large ubuntu image, debian minimal, or alpine. My advice is that I have found several applications not to work on the extremely minimal alpine but if it does use it (90.4 MB). You should have no problem using the debian minimal image (slim) and this should be your default choice (162 MB). Use the ubuntu image as a last resort as it is HUGE (691 MB).

The size of a docker image will determine how fast a container management framework such as kubernetes can spinup your instance. Smaller is better and "too big" is always relative. I stick to less than 300-400 MB.

CHAPTER 5

Testing

Testing should be required for all source code. While there are many tools available for testing python code `pytest` in my option is the clear winner. I beleive this is due to `pytest` being the most pythonic framework for testing. PyTest gives a guide on integrating it into your project. The following will give a setup that is both simple and opinionated. Another easy win we can get with pytest is code coverage. We will use the add-on package `pytest-cov` for this.

5.1 setup.py

```
setup(
  ...
  setup_requires=['pytest-runner', ...],
  tests_require=['pytest', 'pytest-cov'],
)
```

5.2 setup.cfg

```
[aliases]
test=pytest
```

That is all you need to get `pytest` running! You can run all tests via the command `python setup.py test` or `py.test`. Now that is all assuming you have tests. `pytest` by default will look for tests in the `tests` directory and runs all files with the name `test_<filename>.py`. In order to get the additional coverage report with the tests you need to add some additional arguments `python setup.py test --addopts "--cov=pypkgtemp"`. Read the pytest documentation for more detailed documentation . For example create a file `tests/test_example.py` with the following code.

5.3 test/test_example.py

```
def test_example():
    assert 1 == 1
```

Now run the test via `python setup.py test` and you should see that one test passes. Similarly how we discussed that all new tags of our project should be pushed to PyPi we should also test all commits when they are pushed to gitlab. Adding to the `.gitlab-ci.yml` setup in the **:doc:'pypi'_** documentation.

5.4 .gitlab-ci.yml

```
stages:
  - test
  - deploy

test:
  image: python:3.6
  stage: test
  script:
    - pip install .
    - pip list
    - python setup.py test --addopts "--cov=<package-directory>"
```

With these changes it will test every commit given to Gitlab and will only submit a new package if all tests pass! All of this has been automated for us. In order to get the coverage report setup correctly you will need to tell gitlab the regular expression to use in order to parse the coverage report. For `pytest-cov` this is `^TOTAL\s+\d+\s+\d+\s+(\d+)\%\s*$`.

CHAPTER 6

Documentation

Documenting a python project is a daunting task. Even though I myself have had experience with documentation it always takes me time to get the setup just right. In python the standard way to create documentation is with Sphinx. Sphinx is not straightforward to use and relies heavily on restructured text. Restructured text is a somewhat more verbose markup language than Markdown but is not too hard to learn the syntax. There is a lot to learn to use RST properly with python sadly. Most of the installation instructions follow the awesome An idiot's guide to Python documentation with Sphinx and ReadTheDocs. My changes are that I want to show how to include docstring in the google format and how to additionally deploy documentation on a static site without readthedocs.

First you will install sphinx and create a docs folder in the root of your project. If you want to use the readthedocs theme for the documentation you will need to install the sphinx_rtd_theme.

1. `pip install sphinx sphinx_rtd_theme`
2. `mkdir docs`

Next you will want to setup a basic sphinx project. You will do this by running the command *sphinx-quickstart* within the `docs` folder. Most of the default options are good. You will need to set a `project name`, `version`, and answer yes to `autodoc` since we want our project source code to be documented. At this point you will have very basic sphinx documentation. Next we need to add our package source documentation. This can be done via the `sphinx` tool `sphinx-autodoc`. Add the following to your `Makefile` in the `docs`. Now run `make apidocs` to add the outline of your source code.

6.1 docs/Makefile

Add to the makefile the following lines.

```
apidocs:
     sphinx-apidoc -o source/ ../<package>
```

If you wanted the readthedocs theme instead of the default you will need to modify `docs/conf.py`.

```
...
html_theme = 'sphinx_rtd_theme'
...
```

The default sphinx apidoc is tedious and verbose. I recommend using sphinx napoleon docstrings which has been standardized by google and numpy. In order to use napoleon the extension needs to be added.

6.2 docs/conf.py

```
...
extensions = [
    ...
    'sphinx.ext.napoleon'
]
...
```

Here is an example of simple function being documented in the google style. See the google docstring format for further details.

```
def fizzbuzz(n):
    """A super advanced fizzbuzz function

    Write a program that prints the numbers from 1 to 100. But for
    multiples of three print "Fizz" instead of the number and for the
    multiples of five print "Buzz". For numbers which are multiples of
    both three and five print "FizzBuzz" Prints out fizz and buzz

    Args:
        n (int): number for fizzbuzz to count to

    Returns:
       None: prints to stdout fizzbuzz
    """
    def _fizzbuzz(i):
        if i % 3 == 0 and i % 5 == 0:
            return 'FizzBuzz'
        elif i % 3 == 0:
            return 'Fizz'
        elif i % 5 == 0:
            return 'Buzz'
        else:
            return str(i)
    print("\n".join(_fizzbuzz(i+1) for i in range(n)))
```

If you want math support there is a mathjax extension. Just again modify `conf.py`. If you want latex support when exporting to a pdf follow this math sphinx documentation.

```
...
extensions = [
    ...
    'sphinx.ext.mathjax'
]
...
```

Math can then simply be included inline or in block format. Use the awesome latex markup language to write equations.

```
as some inline text

:math:`\beta \gamma`

or as a block math equation

.. math::

   \beta = \gamma
```

At this point you are ready to go! You can run `make html` within the docs folder and it will build the website in `docs/_build/html`. Okay so great we have the static files for the website but how do I deploy them?! There are two answers and you can choose both: self hosting and readthedocs.org.

6.3 readthedocs.org

First you will signup an account with readthedocs.org. It is not necessary to link an account as readthedocs will work with any publicly available version controlled repo. `Import a project -> Import Manually` and give the project a unique name and specify the repository url. The name that you provide determines the url `<name>.readthedocs.org`. For full documentation see the `https://docs.readthedocs.io/en/latest/getting_started.html`.

Readthedocs will detect and change in the repository and rebuilt the documentation. However often times the default configuration does not work with cutting edge projects and also by default does not install the project when building the documentation. To specify the readthedocs configuration in your project you should use .readthedocs.yml. A basic configuration is specified below. Readthedocs uses docker containers and has many more configuration options. With this you should be all setup! Read the documentation for additional options.

6.4 .readthedocs.yml

```
build:
  image: latest

python:
  version: 3.6
  setup_py_install: true
```

Scientific packages often have dependencies that require c extensions or cython. In order to use readthedocs that has c extension dependencies you will need to mock out all the dependencies in the `conf.py`. This is documented in the readthedocs FAQ. One more reason I would recommend hosting the static site yourself.

```
import sys
from unittest.mock import MagicMock

class Mock(MagicMock):
    @classmethod
    def __getattr__(cls, name):
        return MagicMock()
```

```
MOCK_MODULES = ['pygtk', 'gtk', 'gobject', 'argparse', 'numpy', 'pandas']
sys.modules.update((mod_name, Mock()) for mod_name in MOCK_MODULES)
```

6.5 static documentation site

Sometimes it is nicer to just deploy the static website yourself. With this deployment we get much more flexibility on the resulting documentation. Read the docs is an awesome resource but it does have limitations. For instance one issue I have had is that it does not generate docstrings from cextensions such as cython code and cannot handle packages with c extensions. There are workarounds by mocking the modules. In these cases we can use Gitlab CD/CI for deploying our own static site.

Since we already have a pipeline for our project lets include the static website building. Add the following to `.gitlab-ci.yml`

6.6 .gitlab-ci.yml

```
stages:
  - test
  - deploy
  - docs

pages:
  image: python:3.6
  stage: docs
  script:
    - pip install sphinx sphinx_rtd_theme
    - pip install -e .
    - mkdir public
    - cd docs
    - make apidocs
    - make html
    - cp -r _build/html/* ../public
  artifacts:
    paths:
      - public
  only:
    - master
```

We are using gitlab pages to deploy our website. It should be available at `<username>.gitlab.io/<repo>`. If you would like to add a custom domain follow either my blog at gitlab static site deployment or look at the gitlab cloudflare documentation.

Now you have your documentation completed!

CHAPTER 7

Command Line Interface

There are numerous tools in python to help you create a command line interface. Some of these include `click`, `docopt`, and SO many others. Personally I have found that `argparse` in the standard library does 90% of the things that I need in a command line. Since `argparse` is in the stdlib full documentation is available argparse

7.1 setup.py

```
setup(
     ...
     entry_points={
          'console_scripts': [
              '<command>=<package>.__main__:main'
          ]
      },
      ...
  )
```

7.2 <package>/__main__.py

```
import argparse
import sys


def main():
    parser = argparse.ArgumentParser()
    subparsers = parser.add_subparsers()
    add_subcommand_fizzbuzz(subparsers)
    if len(sys.argv) == 1:
        parser.print_help()
        sys.exit(1)
```

```
    args = parser.parse_args()
    args.func(args)


def add_subcommand_fizzbuzz(subparsers):
    parser = subparsers.add_parser('fizzbuzz', help='do the fizzbuzz!')
    parser.set_defaults(func=handle_subcommand_fizzbuzz)
    parser.add_argument('-n', '--number', type=int, default=100, help='number for␣
↪fizzbuzz to count to')


def handle_subcommand_fizzbuzz(args):
    from pypkgtemp.hello import fizzbuzz
    fizzbuzz(args.number)


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

And there you have the simplest non trivial and scalable argparser. This demonstration shows how to create subcommands and take options with certain types and defaults. You can run the example via `<command> fizzbuzz -n 42` or `<command> fizzbuzz`.

CHAPTER 8

Badges

Obviously the most important part about creating packages is the amount of flair that you have. Badges are the way to achieve this. Many of the sites described in the documentation provide badges such as conda, gitlab, and readthedocs. Sadly PyPi does not provide badges but we can still get them from shields.io.

For the conda badges go to `anaconda.org/<username>/<package>/badges` and you will see a list of available badges that you can use. For gitlab badges go to `https://gitlab.com/<username>/<project>/settings/ci_cd` and scroll to Pipeline status pipeline status and coverage report should be available. The coverage report relies on the fact that you setup coverage described in the testing section. For readthedocs a single badge is provided to show that documentation is building `http://<package>.readthedocs.io/en/latest/?badge=latest`.

Markdown and restructured text are not the best formats for creating tables (org mode is). But for non emacs uses and a way to embed a table in markdown we can just use HTML. I stole this idea from what the pandas developers did.

They use a simple html table.

```
<table>
<tr>
  <td>Latest Release</td>
  <td><img src="https://img.shields.io/pypi/v/pypkgtemp.svg" alt="latest release" /></
↪td>
</tr>
  <td></td>
  <td><img src="https://anaconda.org/costrouc/pypkgtemp/badges/version.svg" alt=
↪"latest release" /></td>
</tr>
<tr>
  <td>Package Status</td>
  <td><img src="https://img.shields.io/pypi/status/pypkgtemp.svg" alt="status" /></td>
</tr>
<tr>
  <td>License</td>
  <td><img src="https://img.shields.io/pypi/l/pypkgtemp.svg" alt="license" /></td>
</tr>
<tr>
```

```
  <td>Build Status</td>
  <td>
    <a href="https://gitlab.com/costrouc/python-package-template/pipelines">
    <img src="https://gitlab.com/costrouc/python-package-template/badges/master/
↪pipeline.svg" alt="gitlab pipeline status" />
    </a>
  </td>
</tr>
<tr>
  <td>Coverage</td>
  <td><img src="https://gitlab.com/costrouc/python-package-template/badges/master/
↪coverage.svg" alt="coverage" /></td>
</tr>
<tr>
  <td>Conda</td>
  <td>
    <a href="https://gitlab.com/costrouc/python-package-template">
    <img src="https://anaconda.org/costrouc/pypkgtemp/badges/downloads.svg" alt=
↪"conda downloads" />
    </a>
  </td>
</tr>
<tr>
  <td>Documentation</td>
  <td>
    <a href="https://costrouc-python-package-template.readthedocs.io/en/latest/">
    <img src="http://costrouc-python-package-template.readthedocs.io/en/latest/?
↪badge=latest" alt="readthedocs documentation" />
    </a>
  </td>
</tr>
</table>
```

Looking at a `README.md` you can see the resulting table.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search