

# Python code packaging for scientific software



Jo Bovy  
(UNIVERSITY OF TORONTO)

Rev. b6901f7  
Last changed February 04, 2025  
Last built July 04, 2025

All of the code snippets and examples in these notes are being shared under a [CC0 1.0 Public Domain](#) license, so you can re-use and re-mix them in your own work. Otherwise, the text is shared under a [CC BY-NC-ND 4.0 International](#) Creative Commons License, meaning that you may distribute the work in full only, you are not allowed to modify it in any way, and you may not use it for commercial means or to gain monetary compensation (note that this summary is no substitute for the full license terms).

# CONTENTS:

<b>1</b>	<b>Introduction: What makes a good scientific software package?</b>	<b>1</b>
1.1	“Why should I (want to) release my code?”	2
1.2	The dos and don’ts of software package development	3
<b>2</b>	<b>The basic structure of a Python package</b>	<b>7</b>
2.1	Naming your package	7
2.2	Package layout	8
2.3	The <code>setup.py</code> file	11
2.4	Installing your code	14
2.5	Code licenses	15
<b>3</b>	<b>git and GitHub: version control and social open-source development</b>	<b>17</b>
3.1	Version control	17
3.2	Basic git use	18
3.3	Branches	21
3.4	Some useful advanced git features	24
3.5	Using GitHub to build a community for your code	25
<b>4</b>	<b>Documenting your code and hosting the documentation online</b>	<b>31</b>
4.1	Basics of good documentation	31
4.2	Python docstrings	34
4.3	Using <code>sphinx</code> to write and generate documentation for your package	44
4.4	A brief tour of <code>reStructuredText</code>	54
4.5	Including docstrings into the <code>sphinx</code> documentation	58
4.6	Including <code>jupyter</code> notebooks as part of your documentation	63
4.7	Automatically building and hosting your documentation on <code>readthedocs.io</code>	66
<b>5</b>	<b>Testing your code</b>	<b>71</b>
5.1	Basics of good testing	72
5.2	Writing simple tests	75
5.3	Running a test suite with <code>pytest</code>	80
5.4	Test coverage	87

<b>6</b>	<b>Automatically building and testing your code: continuous integration</b>	<b>95</b>
6.1	Why continuous integration? . . . . .	95
6.2	Continuous integration with Travis CI . . . . .	97
6.3	Continuous integration for Windows: AppVeyor . . . . .	108
6.4	GitHub Actions, the new kid on the block . . . . .	112
6.5	Analyzing test coverage online using Codecov . . . . .	118
6.6	Status badges for your package . . . . .	122
<b>7</b>	<b>Releasing your package</b>	<b>129</b>
7.1	Versioning your code . . . . .	129
7.2	Preparing for your package’s release . . . . .	131
7.3	Uploading your package to the Python Package Index (PyPI) . . . . .	134
7.4	Building and adding binary distributions (“wheels”) to your PyPI release . . . . .	137
7.5	Starting the development of your next version . . . . .	141



## **INTRODUCTION: WHAT MAKES A GOOD SCIENTIFIC SOFTWARE PACKAGE?**

The purpose of these notes is to introduce you to the main tools available for creating, developing, maintaining, and releasing a Python software package, with a particular emphasis on software packages with an academic, scientific audience. While many scientists these days write and run computer code for much of their working hours, relatively few scientists even now write software packages that gain much use outside of their own group of collaborators, even though many of us write code that is generally useful and all of science would benefit if scientists shared their tools more widely. These notes aim to help move the community of scientists along toward more open-source, open-development, well-tested, and well-documented code releases.

My own field is that astrophysics and my experience in writing code for astrophysical applications inevitably colors all opinions expressed in these notes. I have written a few software packages that have found more wide-spread use, primarily [galpy](#), a Python software package for galactic dynamics, but also various [other packages](#) for data handling and analysis and for machine learning. Many of the thoughts expressed in these notes, especially the more opinionated ones, come from my experience developing and maintaining these packages, with many of the lessons espoused in these notes learned the hard way by me. But once you get the hang of how to package your code in a way to make it useful to people, it becomes far easier to create new packages. Making the most of the many wonderful automation services that are freely available these days is a crucial part of making developing and maintaining code easier, and these notes therefore go into significant detail on how to use these services.

Before we begin, it is important to mention that these notes are not intended to teach you how to program in Python, neither the basics nor more advanced concepts, and a mature understanding of Python is assumed. Much of what is discussed would generalize to packages written in languages other than Python, but no attempt is made at generalization.

### 1.1 “Why should I (want to) release my code?”

Even today, many scientists are often reluctant to release code they use, both more general software packages that they use in many projects or the code specific to the analysis in a specific project. The reasons given vary from embarrassment at the state of the code and the fear of being scooped to it being too much of a burden to release code and wanting to keep code private as a competitive advantage. About embarrassment there is little to say except that if you are so embarrassed by your code that you would not want to share it, then perhaps the code is not fit for use in a scientific paper. We should all think about the code we write to do our projects as equal in status to the paper that describes the project and its results, and therefore one should not end up with a code that one is embarrassed about, just like nobody submits a paper that they are embarrassed about. With the increasing complexity of all scientific analyses, actually having access to the code is the *only* hope we have of being able to reproduce analyses from previous work, since published papers fall woefully short in providing a full description of all steps involved. But if we had at least a snapshot of the code used for each paper, we would have a much better chance of reconstructing what happened (perfect reproducibility is unfortunately a very thorny problem, with ever changing versions of the many software packages one relies on and even of the underlying operating systems).

(This point was brought home once again in a recent exchange with an author whose work my collaborators and I are relying on in an ongoing project; we asked the author to clarify a calculation in their paper and its relation to a similar calculation in a previous paper and they replied that they could not state whether the calculation was correct or not, because the paper was written seven years ago. If the actual author of a paper cannot reconstruct what they did a few years later, what hope is there for outside readers? But if the code had been available, it would have been easy to check.)

About the fear of being scooped all I can say is that I do not believe I have ever heard of anybody being scooped by people using their publicly-available code.

Because of the need for reproducibility, keeping code private once it has been used in a scientific publication is unethical. Even if you do not make your code publicly available online, interested parties should be able to request code that you used as part of a scientific publication, provided that it can be shared without placing undue burden on the requestee (but it is hard to imagine that sharing a code that is so precious that one would want to keep it private would be difficult to share).

Thus, it is clear that there are many good reasons from the standpoint of scientific practice and ethics for sharing and releasing your code, but what about *positive* reasons for releasing your code?

I believe that you should want to release your code because it will make you do **better science**, cause you to **gain collaborators**, increase your **professional profile**, help you **share your work** with as wide a community as possible, and allow you to **help the next generation of scientists** bloom.

One of the great advantages of sharing your code and having it be used by other people is that they will find issues in the code. As they say, that’s not a bug, it’s a feature . All code has bugs (this is why we will discuss in detail how to *test your code* (page 71)), and the more eyes on the code and the more people using it, the more likely that major bugs will get caught and fixed (and fixed

*forever* using *continuous integration* (page 95)). In the end, this leads to better science, because code that's used and re-used is more likely to be correct.

If you release your code and it gets used by a community of people, this will increase your professional profile, because people who use your code will know and remember you, and they will remember you fondly, because you help them out in their research without asking anything in return. People working with your code will contact you and describe their own use case, and this will invariably lead to new collaborations.

Sharing your code and sharing the results of your research as *re-usable code* is also a great alternative way to gain exposure for the science that you are doing. People using your code will want to find out what you did with the code and thus they will learn about the research that you are doing. If your code is more generally applicable than the specific science area that you work in, this will attract attention from people who might otherwise not learn about your research. And through your code, you can influence how people do science, for example, by setting defaults to values that you find reasonable.

Finally, an important reason for why I release code is to help make science more accessible. Well-documented, well-tested code that is easy to use can provide an important jumping off point to people new to the field you are in, whether they are students just starting out or more seasoned researchers exploring a different field. By making state-of-the-art tools easily available, we increase the overall level of science done and help researchers explore their ideas. By making code open source and developing in the open, this is the case no matter whether people are at a prestigious institution, or students in remote areas of the world.

## 1.2 The dos and don'ts of software package development

There are many ways in which scientific software packages become successful. Your package may solve a problem that is very difficult to solve and/or that requires specialized routines and a level of code optimization not generally known among scientists; in this case, users will likely put up with a lot of inconveniences. Most high-performance-computing codes fall in this category and they are not typically known for being easy to install or use. However, much more common these days is that a code solves a problem that is not easy to solve oneself, but doable by most scientists working in the field with a moderate amount of effort, at least to write a basic version of the code. Examples of these types of problems in astrophysics are (i) reading different data formats, (ii) doing astronomical coordinate conversions, (iii) convolving spectra with photometric bands to generate photometry, etc. But there is an enormous advantage to the community for these tasks to be performed by generally-used software packages, because, through testing and use, packages become very robust against bugs and because much-used packages have an incentive to get all of the subtleties in calculations correct, which would likely be ignored by an individual writing a basic version of a calculation. In astrophysics, this is a key reason why *astropy* has been such a success, despite in many ways implementing basic astronomical tools that each on their own could be implemented by a graduate student: the combined effort from many individual contributors has made *astropy* robust and general to such a degree that it far surpasses what any individual could write

and it therefore becomes an essential research tool.

In my opinion, the single-most important reason behind a code's success in gaining wide-spread use is being **easy and intuitive to use**. This should be the main driver behind any decision regarding details of implementation, documentation, and installation of your code. Always put the user first. Put *yourself* in the user's position and ask how you would want to use the code if you had not written it. That's difficult to do, but it's important to try. Users want a code to be easy to install, because unless a code solves a problem they need to use and it would take them weeks or months to solve it themselves, users will give up and implement the code themselves if they cannot easily install your code. Functions, classes, methods, and variables in your code should have intuitive names, so it is easy to remember how to use your code without having to constantly look at the (hopefully good and comprehensive!) documentation. The basic functionality of your code should be able to be run with a few lines of code, nobody wants to have to start with a hundred line code-block to get any output from your code (that's okay for advanced use, but basic use should be *brief*). And whenever you face a choice between simplifying the implementation of your code or simplifying its use, you should go for the latter even if at the expense of the former (simple, intuitive implementation is important from a maintenance perspective, but user experience is more important in my opinion).

To summarize, here is a list of dos and don'ts for developing your package:

- **do** allow your code to be installed using standard installation commands (`pip install X` for PyPI packages, `python setup.py build/install/develop` for building from source, `./configure, make, make install` for compiled code).
- **do** have your code and auxiliary files installed in standard locations. Nobody wants to have to use your code from the directory it was downloaded it in; code should be able to be installed and be available system-wide if desired.
- **don't** require use of files in non-standard locations: small data files should simply be copied to the code's directory; the exception to this rule are large data files that are necessary, which may require the user to give a preferred location for these (e.g., through an environment variable or a configuration parameter).
- **do** attempt to make your code work on commonly-used operating systems: Linux/UNIX, Mac OS, Windows if possible (Windows can be tricky, although for pure Python code it should be relatively straightforward if one pays attention to some details, like using `os.path.join` to create file paths rather than writing them out directly).
- **do** support the last few minor Python and `numpy` versions (there is a [numpy NEP](#) with guidelines). To be as stable as possible, avoid using new features in the latest Python version, or at least test for Python version in your code so it does not entirely fail on older versions.
- **don't** require too many dependencies: What's great about Python is the enormous variety of packages available and you may be tempted to include many dependencies that your code can use. But dependencies make code hard to maintain. Most packages you depend on are under ongoing development and will change calling sequences, deprecate features, move code between submodules, etc. and all these changes will break your own code (for example, `scipy` has moved the `logsumexp` function twice since I started developing `galpy`, requiring multiple `if` statements to deal with different `scipy` versions that `galpy` users might have).

- **don't** require hard-to-install dependencies: Dependency-problems are compounded when dependencies are difficult to install. I am old enough to remember when simply installing `numpy` or `scipy` was difficult; luckily that is no longer the case, but many packages can still be difficult to install, especially if they require compiling code and linking to external libraries. If a dependency is difficult to install, it is likely that many of your potential users will fail to install the dependencies; if this means that they cannot run your package, they will not end up using it.
- **do** make all but the basic dependencies optional: Dependencies can be made optional by enclosing their imports in a `try: ... except: ...` statement and only raising an error when your code truly cannot function without the non-imported dependency. This is a great way to make sure that hard-to-install dependencies do not fully block use of your code. If it is likely that users will lack a dependency and this lack would make a simple `import YOUR_PACKAGE` fail, it is a good idea to make that dependency optional. A good practice is to make any dependency that cannot be reliably installed using a simple `pip install optional`.
- **do** document new features from when you write a first, basic versions of them. Documentation should not be an after-thought, something that you will “get to once the code is mature”. Keep documentation up-to-date with changes in the code.
- **do** keep a changelog, documenting all but the most minor changes of your package's functionality.
- **do** use the standard Python mechanism for reporting [errors and exceptions](#) and for raising [warnings](#). Avoid excessive warnings, but err on the side of warning too often rather than too little. Use warnings to point out changes to the code that a user may not be aware of, or to warn about possible unintended usage. Using the standard Python exception/warning syntax allows users to catch and ignore errors and warnings as they see fit.
- **do** write [a comprehensive test suite](#) (page 71) for your package and use [continuous integration services](#) (page 95) to run it automatically whenever you update the code.

This is just an incomplete list of dos and don'ts and I will give more tips in the remainder of these notes.



## THE BASIC STRUCTURE OF A PYTHON PACKAGE

To start, I will explain the basic structure of a Python package, which provides a skeleton onto which you can add the features that we will discuss later in these notes (documentation, automated testing, etc.). One way to learn about package structure is to peruse Python packages on GitHub, e.g., [galpy's GitHub page](#). But because these packages have many complex maintenance features implemented and they may use files specific to GitHub integration, this can be a confusing way to start learning the structure of Python packages (however, once you know the basics, looking at other packages is a great way to discover new features that you may want to use in your own package; in general, you can learn a lot by reading other people's code).

Similarly, there exists a wide variety of package generators, pieces of code that will generate skeleton packages that you can fill in to create your own package. A popular class of such package templates are those generated using the [cookiecutter](#) command-line utility, which allows you to generate package skeletons for many different languages / layouts simply by calling `cookiecutter` with the URL of a template. For example, `astropy` provides a [cookiecutter package template](#) specific to packages in the `astropy` eco-system. In general, I shy away from using such templates, certainly for beginning packagers, because these templates come with a confusing amount of advanced features that obscure the basic structure of a package and that distract from the basic development of the package (when I generate a template using the `astropy` template, I don't even know where to start putting code!). Cookiecutter templates are useful for advanced users creating many packages, but for the purpose of learning about packaging, I think it is better to build the package from the ground up and add each advanced feature individually later.

### 2.1 Naming your package

The first decision you have to make when creating a package is what to name it. Because it is annoying to rename a package later, this is an important decision to make early on and it is worth spending at least a few moments thinking about a good name. A memorable, catchy name will help your package gain attention. Indeed, I am very happy to have snatched the “galpy” name when it was available and even then I only ended up with “galpy”, because the name I had originally wanted to use was `pygd` (for “Python galactic dynamics”), but this name was already taken by [a project on sourceforge.net](#).



Besides being catchy, it is also important for a name to be unique, so you'll want to check that the name is not already in use by another project. For Python packages, it's essential to check that there is no package of the name you are thinking about available on the [Python Packaging Index](#) (PyPI), by searching their database. This is essential, because eventually you'll want to be able to install your package using a simple `pip install PACKAGE_NAME`, because that is the first thing that users will try when they learn that they need to use your package. If `pip install PACKAGE_NAME` installs a *different* package, many users will end up being very confused. So while you can have a different PyPI name from your package name, in the case of a conflict it is better to move away from your intended package name and choose one that is available. For a Python package, being available on PyPI is the most important consideration, but you might also want to check [sourceforge.net](#) to check more generally against names of open-source projects (not necessarily in Python) and search GitHub (although in the case of GitHub, the most important conflicts would be with packages that actually appear to be used by a wider community). To make sure that the name does not disappear while you are developing, you may want to register your package on PyPI as soon as possible, by [publishing a first release](#) (page 129).

As for what to choose as a name, tastes differ. Many Python packages choose to end in `py` to make it clear that they are Python packages (e.g., `numpy`, `scipy`, `astropy`, `galpy`), but this is not a rule and a package name can be anything (indeed, the number of good, available names ending in “py” is rapidly dwindling). You can choose a name that succinctly describes what your package does (this has long been my own preferred naming convention, leading to such dryly named packages as `apogee`, `mw dust`, `gaia_tools`) or you can choose a clever name or acronym (my own forays in this direction are `wendy` and `kimmy`, although nobody ever seems to get them...; also illustrating that you can just end in “y”!). But I would recommend keeping the name of your package relatively short, because even in the age of tab-completion, people using your code will end up typing its name *a lot*.

So we will not have to tediously refer to `PACKAGE_NAME` as the name of our under-construction package, from now on we will use `exampy` as the example (get it?) package. I will use `exampy` throughout these notes to illustrate everything that is being discussed. The `exampy` package is available [here on GitHub](#) and [here on PyPI](#).

## 2.2 Package layout

Once you have decided on a name, it is time to start building your package. Make a directory that will hold your package, which I typically give the name of the package, but this is not required. Later, we will host this entire directory on GitHub and I will refer to it as the “top-level directory”. In this top-level directory, your package will be contained in a sub-directory that has the name of your package, in our example case this is `exampy/`. This directory will contain all of your code. Other sub-directories of the top-level directory will hold documentation and tests and sub-directories will also be automatically be generated when you build and distribute your code (more on that later). We will be using this example package throughout the rest of these notes to illustrate documentation and testing tools, so you may want to follow along and implement this simple package yourself to be able to keep using it in the next chapters. You might want to add it to GitHub as



exampy-GITHUBUSERNAME to distinguish it from the [original package](#).

Files in the top-level directory largely hold meta-information about your package. The top-level directory should have a README file with basic information about the package, it will hold the *license file* (page 15), eventually it will contain configuration files for automated documentation generation and for continuous integration of tests (but not yet!), if you host the package on GitHub it may have one or more files specific to GitHub integration, and it will hold a few files related to the installation and distribution of your code, the most important being the `setup.py` file.

Because we are building the package from the ground up, at first our package will have the following structure

```
TOP-LEVEL_DIRECTORY/  
  exampy/  
  setup.py
```

To make the package into an importable Python module, the package directory needs to contain an `__init__.py` file, which can simply be an empty file created using `touch exampy/__init__.py`. So a full-fledged, bare-bones Python package looks like

```
TOP-LEVEL_DIRECTORY/  
  exampy/  
    __init__.py  
  setup.py
```

Without writing any further code under `exampy/` (but with a basic `setup.py` file that we will describe below), this example package could be installed and imported in a Python session.

The `__init__.py` file contains everything that is imported by `import exampy` or `from exampy import *` (which you should never do!). You can put functions and classes directly in the `__init__.py` file or you can write them in other files (to organize your code more clearly) and import them in `__init__.py` to make them easily accessible. For example, say that we implement a first set of basic math functions in `_math.py` and our package now looks like

```
TOP-LEVEL_DIRECTORY/  
  exampy/  
    __init__.py  
    _math.py  
  setup.py
```

then without adding code to `__init__.py` we need to `from exampy import _math` to gain access to the functions in `_math.py`; `import exampy` would, for example, not allow access to `exampy._math`. If you want the functions to be available under `import exampy` directly, you can import them in the `__init__.py` as follows:

```
# __init__.py  
from ._math import *
```

(although better would be to explicitly import all of the functions that you want to import). This will make functions in `_math.py`, say you have a function `def square(x): return x**2`, as `exampy.square`, available through, e.g., `from exampy import square`. Alternatively, if you want to retain the “`_math`” part of the function, you can do

```
# __init__.py
from . import _math
```

which makes the `square` function available as `exampy._math.square`. In both of these cases, we get the `square` function using a simple `import exampy`. I discuss below why I chose to start the `_math.py` filename with an underscore.

When your code grows in complexity, you likely will want to separate functionality into different submodules, such as `exampy.integrate`, which will contain functions to integrate mathematical functions. As we saw above, such a structure can be generated by having a single file `integrate.py` under the main `exampy/` directory, but to allow for `integrate` to consist of multiple files, it is better to make a directory `integrate` under `exampy` and use an `__init__.py` file in that directory to make it a submodule. In this case, our example package’s layout becomes

```
TOP-LEVEL_DIRECTORY/
  exampy/
    integrate/
      __init__.py
    __init__.py
    _math.py
  setup.py
```

Everything that we have discussed so far for the main `exampy/` directory contents holds for this submodule as well: we can either write code in `integrate/__init__.py` directory or in different files in that directory. For example, imagine that we have a file `integrate/_integrate.py` that implements a simple [Riemann sum](#) `def riemann(func,a,b,n=10): return np.sum(func(np.linspace(a,b,n))*(b-a)/n)`. Then with an empty `integrate/__init__.py` file we have to `import exampy.integrate._integrate` to gain access to `exampy.integrate._integrate.riemann` (or `from exampy.integrate import _integrate` or similar), *or* we can again import the `riemann` function in `integrate/__init__.py` to make it accessible through a simple `from exampy import integrate` call.

The convention I personally follow is to define submodules as much as possible through subdirectories rather than as files, pulling all of a (sub)module’s functionality into its `__init__.py` file to make it accessible to the user. This is why I gave the non-`__init__.py` files in the example above names that start with an underscore. This indicates in the Python universe that these are *internal* parts that should not be accessed directly by users; their functionality is exposed to users by importing it into the (sub)module’s `__init__.py` file. But this is largely a matter of taste, the most important considerations being keeping things simple for the user and keeping the code easily understandable for yourself (in that order!).

The considerations in naming submodules are similar to those discussed in naming the package

as a while [above](#) (page 7): choose short, descriptive names (not clever ones in this case; great examples are `scipy.integrate`, `scipy.interpolate`, which immediately make clear what these submodules do and don't do).

## 2.3 The `setup.py` file

Next, we want to make our package installable using standard Python installation tools. The main tool used for Python packaging is `setuptools`. To use `setuptools`, we write a `setup.py` file that includes all of the information necessary to build, install, and package the code.

Some packages use a `setup.cfg` configuration file to define the necessary information, but even in that case one still needs to write a `setup.py` file that ingests the configuration file and hooks it up to `setuptools`. While this has some advantages, for beginning users I think it is easier to directly write the `setup.py` file, which is instructive and also allows for extensive customization later. Another downside of using a `setup.cfg` file is that it makes it that `python se[TAB]` no longer auto-completes to `python setup.py`! Advanced `setup.py` files can become quite complicated (e.g., take a look at [galpy's setup.py file](#)), so while it is again instructive to look at other packages' `setup.py` files, for beginners this is likely to be highly confusing.

The main thing a `setup.py` file has to do is to call `setuptools.setup()`, which then takes care of supporting all of the basic installation and packaging tools. For our example package `exampy` above, a simple, bare-bones `setup.py` file is the following

```
# setup.py
import setuptools

setuptools.setup(
    name="exampy",
    version="0.1",
    author="Jo Bovy",
    author_email="bovy@astro.utoronto.ca",
    description="A small example Python package",
    packages=["exampy", "exampy/integrate"]
)
```

This basic `setup.py` file defines the name of the package, its version, some basic information about the author and the package, and it tells `setuptools` what the actual package is. If you add this file to the example package, you will now be able to install it, by doing `python setup.py install`, but see [below](#) (page 14) for more on how to install code.

Because installation proceeds by running the `setup.py` as a Python script, `setup.py` can contain arbitrary code to help install your code. Let's take a look at what other keywords we can provide to the `setup()` function. We can provide:

- A `long_description`: This is a detailed description of what the code does (longer than the

description, which should be a single sentence) and what eventually would be published on the package's PyPI site (e.g., see [galpy's PyPI page](#)). Typically, one takes advantage of the fact that we can run arbitrary code in the `setup.py` file to read the contents of the README and use it as the `long_description`, using

```
# setup.py
with open("README.md", "r") as fh:
    long_description = fh.read()
setuptools.setup(
    ...
    long_description=long_description,
    long_description_content_type="text/markdown",
    ...
)
```

in case the README's format is [Markdown](#), and we specify the format as well.

- `url=` with the homepage of the package: typically this is the GitHub site. Additional URLs can be specified as `project_urls=`.
- `license=` with the name of the [open-source license](#) (page 15) (e.g., `license='New BSD'` or `license='MIT'`).
- `classifiers=` which contain meta-data about your project used by PyPI to categorize your package. Commonly-used classifiers concern the development status of your code (e.g., `Development Status :: 4 - Beta`, `Development Status :: 6 - Mature`), the intended audience (e.g., `Intended Audience :: Science/Research`), the license (again) (e.g., `License :: OSI Approved :: MIT License`), the programming language used (e.g., `Programming Language :: Python` or more specifically, `Programming Language :: Python :: 3.7`), and the operating system(s) the code works on (e.g., `Operating System :: OS Independent` for all). As far as I know, nobody ever uses these classifiers and I find it difficult to remember to update them (e.g., between Python versions, or when the code matures to a higher development status), but it is considered good practice to include them. For example, you could have

```
# setup.py
...
setuptools.setup(
    ...
    classifiers=[
        "Development Status :: 6 - Mature",
        "Intended Audience :: Science/Research",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
        "Programming Language :: Python :: 3.5",
        "Programming Language :: Python :: 3.6",
```

(continues on next page)

(continued from previous page)

```
"Programming Language :: Python :: 3.7",
"Topic :: Scientific/Engineering :: Astronomy",
"Topic :: Scientific/Engineering :: Physics"]
)
```

for a mature package used in astrophysics that works on recent Python versions on all operating systems. A full list of classifiers is available [here](#).

These are the main descriptive, meta-data keywords used by the `setup()` function.

Further options of the `setup` function help `setuptools` deal with your package's installation and distribution:

- `packages=` lists the modules and submodules included in your package. For the example above, this would be `packages=["exampy", "exampy/integrate"]`. Rather than listing modules manually, you can use `packages=setuptools.find_packages()` to find them automatically, making sure to only include your own package by doing something like `packages=setuptools.find_packages(include=['exampy', 'exampy.*'])`.
- `python_requires=` specifies the Python versions supported by your code, mainly for use by the `pip` installer. If you are not too worried about this, you can omit this, but if you only support Python 3 (very reasonably these days), you can specify `python_requires='>=3'`.
- `install_requires=` lists the basic dependencies of your code, dependencies without which your code cannot run. When users install your code using `pip`, `pip` uses this list to install any missing dependencies. For example, to specify that your code requires `numpy` and `scipy`, do `install_requires=["numpy", "scipy"]`. You can specify version requirements, such as `numpy>=1.7`, using the standard `pip` syntax. If you have a dependency that is not on PyPI (thus, not `pip` installable), but is, for example, on GitHub, you can specify it in `install_requires` and give the URL in the `dependency_links=` keyword, e.g., `dependency_links=["http://github.com/jobovy/galpy/tarball/main#egg=galpy"]` to link to `galpy`'s GitHub source (of course, `galpy` is `pip` installable). In the example `exampy` package introduced above, we used `numpy` in the `exampy.integrate.riemann` function, so we need to specify `install_requires=["numpy"]`.
- `package_data=` is a dictionary with any data files that are part of your package(s) that need to be copied over to the installation directory (only `.py` files are normally copied to the installation directory) and that will be distributed when the time comes to publish your package. To copy data files to directories outside of the installation directory, use `data_files=`. To include the `README.md` and the `LICENSE` file, do `package_data={"": ["README.md", "LICENSE"]}`.
- `entry_points=` gives non-standard entry points to your code. For example, if you are distributing a command-line script, you can install that and make it executable on a user's `PATH`, by specifying

```
# setup.py

...
setuptools.setup(
    ...
    entry_points={
        'console_scripts': [
            'my_script=my_script:main',
        ]
    }
    ...
)
```

which makes the main function of `my_script` an entry point.

More information on `setup()`'s keywords can be found on the [setuptools documentation page](#).

## 2.4 Installing your code

Now that we have the basic outline of an example package and we have written the `setup.py` file, we are ready to install the code! The standard method for installing a package from its source directory (the top-level directory that contains `setup.py`) is to call

```
python setup.py install
```

which installs it in your system's installation directory (typically under `/usr/local` on UNIX-style systems). You can specify an alternative installation location using, e.g.,

```
python setup.py install --user
```

which installs the code in a directory in your home folder (typically under `~/.local` on UNIX-style systems, with modules installed in `~/.local/lib/pythonX.Y/site-packages`). You can also directly set a prefix using

```
python setup.py install --prefix=~/.local
```

where the chosen prefix here is to have the equivalent of the `--user` option (but `--prefix` can be any directory). An alternative to directly calling `python setup.py` is to use `pip` even for local packages. For example, you can install a local project using

```
pip install .
```

However, when you are actively developing a package, installing in the way discussed above means that every time you update the code, you have to re-install it to gain access to any changes you have

made. To avoid this, you can install the package in “develop” mode, using

```
python setup.py develop
```

or

```
pip install -e .
```

if you are using pip. In “develop” mode, the source is not copied to the installation directory, but rather an entry is made in the installation directory to find the code back in the original directory. This means that any changes you make are immediately available system-wide without requiring a re-installation. Of course, if you have the package already loaded in a Python session, you still have to exit and re-start the session (or use `importlib.reload`). If your package includes compiled code and you make changes to the source code that need to be compiled, you do have to re-compile the code by running `python setup.py develop` again.

## 2.5 Code licenses

Before moving onto the next chapter where I will discuss how to start sharing your code online with others, it is important to briefly discuss code licenses. *All code that is shared online should have a license.* Without a license specifying the terms of the code’s use and re-distribution, all code is considered to be copyrighted to the author, without allowing re-use or re-distribution (code that you put online without a license is *not* in the public domain, indeed, the opposite is the case). Thus, you should choose a license for your code and put the license file in your code’s top-level directory. If code is on GitHub without a license, the GitHub Terms of Service allow people to view and fork the code, but no modifications or re-distribution are permitted (see the [No License](#) GitHub help page). License your code.

There are two main categories of open-source licenses: *permissive* free software licenses and *copy-left* licenses.

Permissive licenses, as their name implies, are very generous in their terms. Typically they allow arbitrary use, modification, and re-distribution provided that the original license is retained, the original author is properly credited, while any liability related to any use of the code is explicitly denied. Examples of permissive licenses are the [MIT License](#) and the [BSD 3-clause License](#), with the MIT License appearing to be the permissive license of choice of recent projects. Permissive licenses allow the broadest use of your code, because they require very little of people using your code. Most of the major Python projects that you know and love use permissive licensing (e.g., `numpy`, `scipy`, `astropy`).

Copy-left licenses are open-source licenses that in addition to denying liability and requesting credit for the original author also require that any modifications of the code be re-distributed under a similar copy-left license. The main used example of a copy-left license is the [GNU General Public License version 3](#) (there is also an older [version 2](#) which is somewhat more permissible). Thus, you can only use copy-left-licensed code in packages that are themselves copy-left licensed. In practice,

this tends to decrease adoption of such packages, even though the philosophy behind this style of license is laudable (it aims to make sure open-source software remains open-source).

**Creative Commons Licenses** are not typically used for software, even though they are in heavy use for sharing other creative content such as websites, class materials, scientific papers, blog posts, etc.

The most important thing is that you give your code a license, with the type of license being of secondary importance; *any license is better than no license*. While it may seem silly to you, explicitly denying liability is an important thing to do when you put code online, to legally protect yourself from mis-use of your code (not that this has ever happened to me, but you never know...). When in doubt, choose a permissive license like the MIT License.



## GIT AND GITHUB: VERSION CONTROL AND SOCIAL OPEN-SOURCE DEVELOPMENT

All code should be under version control to keep track of changes over time and when it comes to version control `git` is the dominant system. A 2018 Stack Overflow survey of developers' version control use found that [90% of developers use git](#), with the second most popular version control being the older [Subversion](#), likely mostly to use legacy code that still lives in Subversion repositories. `git` is most widely supported by code hosting services, with [GitHub](#) only hosting `git` repositories and [BitBucket dropping support](#) for the main `git` alternative `mercurial` in 2020. Basically, `git` is now the only game in town.

### 3.1 Version control

Version control is a system for tracking and making changes to code as the code develops. Version control stores code in a “repository” and when using version control, any changes to the code are logged through a code “commit” that lists the files changed and provides a brief description of the changes; version control software then generates a “diff” with respect to the previous version that gets stored, such that the full history of changes is available for use in the future. Opinions differ on how many changes to include in a single commit (which can consist of changes to multiple files at once), but typically it is best to keep commits as “atomic” as possible, that is, create a commit at the smallest change that is reasonable to call a change or improvement to the code. Commits are often as small as changing a single line, perhaps improving the documentation or fixing a small bug. When making changes to existing code, it is always best to keep commits at the level of small changes, so any issues with the changes can later easily be pinned down to a specific change; change commits should typically consist of a few lines to a few dozen lines of edited code at most. When first implementing a new feature, it may make sense to wait to commit until a draft version of the feature is working and one can thus end up with a larger commit, but even then it is best to first implement a skeleton of the new feature and then edit it with small changes until the feature is fully implemented.

Early version control systems stored the history of changes in a central location while each developer only had a copy of the current version of the code; thus, every code commit and every query of the code's history required interaction with the central location (often remote, requiring an internet

connection). Because this meant that one could not commit while offline and even when online was an impediment to quick progress due to the sometimes slow response time of the central location, this led to often bloated commits. One of the great improvements in `git` is that each copy of the code's repository contains the full history, leading to a decentralized system where there is no need for a central location. Different copies of a `git` repositories are called “clones” and with `git`, clones can communicate among themselves without needing to go through a central place. Of course, the current reality is that most `git` repositories have “main” copies that are stored in online services like GitHub or Bitbucket, with most of the communication between different clones happening through the main hosted copy of the code. Nevertheless, the fact that each clone contains the entire history means that you can easily commit code and investigate the history without requiring interaction with a centralized repository, and this is therefore much faster and robust against network interruptions.

In this chapter, I provide a brief overview of the basic `git` features and commands and discuss how to use GitHub to host your scientific code package. Note that this is not supposed to be an exhaustive guide to using `git`, many such guides already exist.

### 3.2 Basic `git` use

The most basic cycle of `git` use is a cycle of `git pull`, `git commit`, `git push`. These commands, respectively, pull in the latest changes from the remote main version of the code repository (e.g., hosted on GitHub), commit new changes made to the code, and push the changes in this commit(s) back to the remote main version. If you add `git diff` for looking at the not-yet-committed current set of changes, `git status` for interrogating the status of the repository and `git log` for looking at the history of the code, and you have well over 95% of my typical usage of `git`.

To get started with `git`, you can initialize any directory to be a `git` repository using

```
git init
```

which initializes an *empty* `git` repository. That is, even if the directory already has files in them, these are *not* automatically added to the `git` repository, instead, you need to add them yourself. To follow along with this tutorial, you can create, for example, a directory `exampy-GITHUBUSERNAME` where you can build your own version of the `exampy` package that I discussed in the [previous chapter](#) (page 7). Keep in mind that while we use `git init` here to get started with `git`, typically the way you start a new `git` repository is *not* by running this command-line command, but instead by creating a new repository on GitHub and setting it up online in such a way that you can directly clone a local copy and start pulling, committing, and pushing changes (see [below](#) (page 25)). I don't think I have run `git init` once in the last eight years.

As discussed above, you should make a commit as soon as enough changes to the code have accrued to make up a reasonable change to the code (again, this could be as simple as fixing a typo in the code, a single character). Simply running

```
git commit
```

will open a text editor (set by your `git` defaults) that allows you to write a message describing the change; this will perform a single commit for *all* current changes to *all* changed files in the repository. You can avoid the use of the text editor by directly specifying the message as

```
git commit -m "A message describing the atomic change made"
```

which I personally prefer as a fan of the command line (and of speed!). You can list specific files to only commit changes to those files by adding them to the command as

```
git commit -m "A message describing the atomic change made" file1.py  
↪file2.py
```

It is good practice to *always specify the files you are committing changes for* rather than not specifying any files or specifying a folder (which would commit changes to all files in that folder). This way, you don't end up accidentally committing changes made to other files that are unrelated to the current commit (we will see *below* (page 24) how you can even split up changes in a single files into different commits).

Before you can start committing changes to files, you need to tell `git` about the existence of the file in the first place (typically soon after you create the file, in preparation for your first commit of the file). This is done using `git add` which you call with

```
git add file1.py
```

and you can also list multiple files. Even though you specify a directory and you can use wildcards, it is again good practice to always explicitly list all files that you are adding, rather than an entire folder or more, because that way you will invariably end up adding files that you did not want to add and removing them again can be difficult.

When your code is centrally hosted (as it should be!), each coding session should start with a `git pull` to pull in changes in the remote main repository that have not yet been added to your clone of the repository. If you are the sole developer of a code, this may seem silly, but it is again good practice to always do this such that it becomes muscle memory and because even if you are the sole developer, you are likely to be developing the code on multiple machines (a personal laptop, a desktop at work, a remote server for running large jobs, ...) and this keeps the code in sync. When you have cloned the code from GitHub and are working in the main branch, a simple

```
git pull
```

will suffice to pull in remote changes, but in general you can specify both the location of the remote repository and the branch. For example, typically the simple `git pull` will be equivalent to

```
git pull origin main
```

which tells `git` to pull changes from the remote repository referenced as “origin” and to pull changes from the main branch.

After you have made one or more commits, you will want to push these commits back to the remote main repository. Before you do that, it is good practice to again first do `git pull` to pull in any changes to the remote repository that may have occurred while you were coding, so you can resolve any conflicts before pushing your own changes (and possibly having them be rejected if there is a conflict). Once you have done this, you push your commits with

```
git push
```

which is again typically a shortcut for the full

```
git push origin main
```

With just these four `git` commands you can get most of the basic functionality of `git` version control. Further useful basic commands are `git diff`, `git status`, and `git log`. The `git diff` command provides a “diff” showing the difference between your clone’s current state and the last commit; thus, it shows the changes you have made since the last commit. Depending on your setup, this diff will simply have ‘-’ lines and ‘+’ lines to show lines that were removed and added (a change on a single line giving both a removal of the old line and an addition of the edited new line) or they may be colored red and green. Running

```
git diff
```

without any additional arguments goes through all files that were changed, but you can look at changes in a single file or in a set of files by specifying them in the call, e.g., as

```
git diff file1.py
```

Running `git status` gives a brief summary of the current version of your clone. It prints the branch you are on and whether you are up to date with the remote repository’s same branch or how many commits ahead of the remote repository you are. It also prints files that have changed since the last commit and files contained in your clone that have not been declared to `git` (for example, new files before running `git add` will show up in that list; after running `git add` they will be listed as newly added). I use `git status` *a lot* to remind myself of what I have been doing since the last commit.

`git log` prints a log of the history of changes to the code. Run without any options, it will provide a moderately verbose list of all commits, listing the commit hash (the unique identifier of every commit), the commit’s author and date, and the summary that you provided when running `git commit`. But `git log`’s output can be highly customized. To get a very succinct listing do

```
git log --oneline
```

which will list each commit in an abbreviated manner on a single line. Or use the `--pretty=` option to get less or more information, e.g.,

```
git log --pretty=short
```

which is similar to the basic output, but does not include the date.

## 3.3 Branches

A feature of most version control systems and one that is especially easy to use with `git` is the ability to *branch* off the main development branch of your code to focus on developing a single feature, fix a single bug, etc. After you are satisfied with the changes on the branch, these changes are *merged* back into the main development commit history. A crucial part of the implementation of the `git` software is fast and intelligent algorithms to perform such merges automatically, even when the difference between the feature branch and the main branch are substantial. When `git` is unable to automatically merge branches, the repository goes into a suspended state until the user manually resolves any merges that cannot be automatically done.

Branches are an incredibly useful feature of `git`, especially when combined with *forks* discussed [below](#) (page 25), and you should make liberal use of them. Branches allow you to split off things like implementing new features, while still keeping the ability to fix bugs in the main branch without that fix having to wait for the new feature to be ready to go “live”. Branches also allow you to develop new features in the incremental way that you should implement all of your code (with many commits), without necessarily having to worry at first that the new feature is entirely compatible with the existing code or that it passes all existing tests.

The main branch is called `main`. It is good practice to keep this branch as clean as possible, that is, avoid having it be in a state where it contains partially implemented features or bug fixes. The `main` branch should always contain a fully working version of your code. Any significant changes to your code should therefore be done in other branches.

To create a new branch, do

```
git switch -c NEWFEATURE
```

which creates a branch called `NEWFEATURE` (which should be a very brief string describing the new feature, e.g., “`add_cube`” if you are adding a function to compute the cube of a number) and switches the state of the repository to this branch. In detail, this command is a shorthand for the following two commands

```
git branch NEWFEATURE
git switch NEWFEATURE
```

where the first `git branch` command creates the branch, while staying on the current branch (e.g., `main`) and the `git switch` command switches the state of the repository to the new branch. After running this, `git status` will report that you are now on the `NEWFEATURE` branch. Any commit that you make now is logged in the commit history of the branch, which is the same as that of

the branch it branched off from up until the branching point and then starts containing additional commits. Running

```
git branch
```

without any further arguments will show a list of all branches that exist in the local clone of the repository (this is not necessarily the same as the branches that exist in the centrally-hosted repository if those branches haven't been checked-out in the local repository. To switch between branches, run

```
git switch SWITCH_TO_BRANCH
```

where `SWITCH_TO_BRANCH` is the name of the branch you want to switch to (e.g., `git switch main` to go back to `main`). This keeps the branch intact, it simply places the working state of the repository to another branch. This is useful if you are working on a new feature in one branch, but want to fix a bug in another branch. Make sure to commit all changes that you made in a branch before switching to another branch, otherwise there is a good chance that you will accidentally commit a change you meant to commit in the feature branch in the wrong branch!

Once you are ready to merge the changes in your branch back into the `main` branch, you switch back to `main` and run the merge command

```
git switch main
git merge NEWFEATURE
```

`git merge` will attempt to perform the merge automatically, in which case you have to do nothing except to okay a commit that performs the merge (sometimes not even that). If the automatic merge fails, you will get a message like

```
Auto-merging file.py
CONFLICT (content): Merge conflict in file.py
Automatic merge failed; fix conflicts and then commit the result.
```

notifying you that the merge has failed and that you have to resolve conflicts between the branches yourself. This is an annoying situation, but it will happen. The failed merge process will leave your files in a state where they record the attempted merge and why it failed; your `file.py` in this case will have a section that looks like

```
<<<<<<< HEAD:file.py
def cube(x):
    return x**3
=====
def newcube(x):
    return x**3.
>>>>>>> NEWFEATURE:file.py
```

You can then manually resolve these, but it is typically easier to use a tool for this, which you can bring up with

```
git mergetool
```

This command will ask you which tool to use (e.g., `opendiff`; if you use Visual Studio Code it will automatically switch to this mode without having to do `git mergetool`) and will then open the files with conflicts in sequence in the merge-tool to allow you to resolve the changes, with typical output

```
This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse_
↳diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
file.py
```

```
Normal merge conflict for 'file.py':
{local}: modified file
{remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Typically, these tools will show the two versions of the file, labeling all sections that need to be merged and showing which cannot be performed automatically and it will show the merged version of the file, which you can edit to resolve the merge (either through an option, such as “choose main” or “choose NEWFEATURE” or by manually editing the merged file). Once you have resolved the conflicts, you need to perform a simple

```
git commit
```

without any other arguments (i.e., don’t specify any files) to commit the merge.

Once you have merged a branch’s changes back into the main branch, you can delete the branch by running

```
git branch -d NEWFEATURE
```

If you have performed the merge elsewhere (e.g., on GitHub), this command might complain that the NEWFEATURE branch contains changes that have not been merged yet, but if you are sure that all is okay, you can force-delete the branch by switching to an uppercase “D”

```
git branch -D NEWFEATURE
```

Be careful with this though, because if you accidentally delete a branch that you still need, it will be *very difficult* to get it back (although, because it’s git, not necessarily impossible...).

## 3.4 Some useful advanced git features

The `git` features discussed above will allow you to do most of your day-to-day work with `git` version control, but `git` has many advanced features. This is not supposed to be an exhaustive guide to all `git` features, but in this section I briefly discuss some of the more advanced `git` features that I use on a semi-regular basis.

Above, we have used `git switch` to switch branches, but `git switch` is a special version of a more general command `git checkout` that can do much more (for switching branches, `git switch` and `git checkout` are equivalent, but switching to a new branch is done with `git checkout -b NEWBRANCH` instead). One often-used invocation is

```
git checkout -- file.py
```

which discards all changes in `file.py` since the previous commit (you can also run it on the entire repository). This is useful when you’ve made a big mess and the easiest way out is to just give up and start over (this happens to me *a lot*). Again, be careful with this command, because once you discard the changes, it is impossible to get them back. In newer versions of `git`, you can equivalently run

```
git restore file.py
```

Besides checking-out branches, `git checkout` can also check-out a previous commit, by specifying the commit’s hash as

```
git checkout COMMITHASH
```

where `COMMITHASH` is the hash (the number like `625123ab491088d6714809648d8a13ae435b7cf8` that you can get from `git log` or elsewhere). This will leave the repository in a “detached HEAD” state, which doesn’t sound good and which isn’t indeed all that good (if you want to actually start making changes, you will have to create a new branch starting from this commit), but it allows you to switch back to an earlier state of the repository and see what it looked like or run tests etc. for the earlier state. That’s often useful when you are trying to figure out where in the commit history *something went wrong*.

If you are working in a branch and have uncommitted changes and you want to switch to another branch (briefly, say) and you *really* don’t want to commit the uncommitted changes before the switch, you can “stash” them away for future use. For this run

```
git stash
```

which stashes all uncommitted changes and reverts the repository back to the previous commit. Then you can switch to another branch without carrying over the uncommitted change. Once you are ready to start work on the uncommitted changes again, switch back to their branch and do



```
git stash pop
```

to bring back the uncommitted changes. You can stash multiple sets of uncommitted changes and there is support for listing them etc., but in practice that becomes ugly very quickly, so it is best to use `git stash` very sparingly and only for very brief periods of time (e.g., you are in the middle of working on a new feature, someone reports a bug that will just take two minutes to fix, so you switch to a branch to fix the bug before coming back five minutes later to take up the new feature's implementation again).

Finally, `git add`, in addition to adding files to the repository's list of files, can be used to specify what parts of the current changes to "stage" for the next commit by running

```
git add -p
```

which is short for `git add --patch`. Run like this, this will start an interactive session that breaks up all of the current set of changes into atomic chunks (called "hunks" for some reason) and asks you whether you want to "stage the hunk" (i.e., add it to the next commit), "not stage the hunk" (i.e., skip it), split the hunk into multiple sub-hunks if you want finer-grained control, or manually edit the way in which the current hunk is staged for the next commit (typing '?' at any time during the process brings up a helpful explanation of the different options).

Using `git add -p` is useful when you have made a lot of changes since the last commit, perhaps because you need many changes to perform a meaningful test of the new implementation, and you want to break it up into multiple small commits for clarity in your repository's history. In general, it is best to simply make small commits along the way, but if you've found yourself making lots of changes since the last commit, `git add -p` will help you out in keeping a sane code history.

## 3.5 Using GitHub to build a community for your code

What is [GitHub](#)? GitHub is an online service to host software packages using the `git` version control system and that has many of the additional bells and whistles to help with a package's development, maintenance, and community interaction. While there is no direct association between `git` and GitHub (there are other services to host `git` repositories, like BitBucket), at this point `git` and GitHub are heavily associated with one another and it seems to me that the total dominance that `git` has attained over other similar version-control systems has much to do with the exquisite support for hosting `git` repositories that GitHub has provided now for many years.

At its most basic, GitHub provides the central location where the main copy of your repository is stored, the location from which you `git pull` and to which you `git push` changes to the code. As such, it provides a crucial back-up service for your code and a central hub that you can use to keep different copies of your code up-to-date with one another. But GitHub provides many more features than that. For aiding in the development and maintenance of your code, GitHub provides a full online viewer of your code, arranged as a file system that is a central part of your code's GitHub website, which allows you to see the latest version of your code as well as the code at any commit

in its history. It can also display changes made in each commit in an easy-to-understand format and show you differences between the code at different points in the code's history. But GitHub's most important feature is that it provides your code's public face where users of your code will go to learn about the code, to find your code's documentation, to interact with the developer(s), to commit patches to the code, etc. For many modern code packages, their GitHub page is *the* public website of the package.

While you can `import` an existing code repository hosted elsewhere online, typically you start by **creating a new repository** (log into your GitHub account to access this page). This brings up a page that asks basic information about the code repository that you want to create. First you specify the repository's name and this should typically be the name of your software package; GitHub does not require names across GitHub to be unique (only within your own account), but as we have discussed before, *using a globally unique name is important* (page 7). Then you can specify a brief description (which can be easily edited later, but it is good practice to always start with a cogent description) and whether to make the repository public (viewable by all internet users) or private (accessible only to yourself and any explicitly added collaborators). If your intention is for a wide range of users to use your code, you'll want to make it public! But even if this is your plan, you may want to start off creating a simple version of your repository in private if you so desire (I don't judge, as long as you make it public soon). Note that if you are an academic educator or researcher, GitHub has a `program` that gives you a free "Pro" account, which comes with unlimited private repositories. When you make a repository private, you can always change it to public later in the repository's Settings. You can then choose to "Initialize this repository with a README", which is a good thing to do, because it will create a skeleton GitHub repository that contains a README file in Markdown format (therefore, `README.md`) that contains the name and description (and that's all your repository will contain at this point!). Starting out with a `README.md` means that you can then clone the repository to your local machine, and start adding and committing changes without having to locally `git init` an empty repository. You also have the option to add a `.gitignore` file (this is a file that contains rules for files that git should largely ignore, e.g, not list as unknown files to git when you run `git status`; this contains entries like `*.pyc` to ignored compiled bytecode Python files; for a Python project, choose the Python version of the `.gitignore` file). You can also immediately add a code license from a list of open-source licenses, which is a *good idea* (page 15). Then hit "Create repository" and you're done!

If you don't initialize your repository with a README or any other file, it will be created but you will have to finalize the initialization of the repository yourself. This is what you do when you have already started the git repository locally using `git init` and by having added and committed some files. In that case, you need to run

```
git remote add origin https://github.com/GITHUBUSERNAME/REPOSITORYNAME.git
```

to tell your local repository about the newly created GitHub repository and then do

```
git push -u origin main
```

to push your local initialization to GitHub. You can run this command after as many commits as you want, that is, you can even push git repositories with thousands of commits to a newly-created

GitHub repository and the GitHub repository will then contain the entire previous history of the code in the same way as if you had developed it while using GitHub (in that sense, GitHub is simply a viewer of your repository's commit history).

When you have initialized your GitHub repository with a `README.md` file, you typically will create a local copy by running (e.g., for the repository that contains these notes)

```
git clone https://github.com/jobovy/code-packaging-minicourse.git
```

The URL here is standard `https://github.com/GITHUBUSERNAME/REPOSITORYNAME.git`, but the simplest way to obtain it is to go your repository's GitHub page and click the big green "Clone or download" button near the top, which will allow you to copy the URL to your clipboard. As the name implies, `git clone` creates an exact, full copy of the GitHub repository on your local machine. When you obtain your local copy in this way, the local copy is automatically aware of the central GitHub location of your code, such that commands like `git pull` and `git push` work without requiring any immediate further setup.

When you create a branch in your local copy of the repository, you need to tell your local copy how to link up this branch with a branch in the GitHub version of your code. Simply trying to run `git pull` in a newly created branch tells you what you have to do here: You can either always (tediously) specify the remote branch as

```
git pull origin BRANCHNAME
```

(similar for `git push`) where "origin" is a shorthand for the GitHub repository (in general, the central location of your code's repository) or you can save this information using

```
git branch --set-upstream-to=origin/BRANCHNAME BRANCHNAME
```

such that you can again simply do `git pull` and `git push` and changes will be pulled and pushed to the correct branch on GitHub. Note that if you have not yet pushed a newly created branch to GitHub, the `git pull origin BRANCHNAME` command will fail to find the remote branch; in this case, first push the branch with `git push origin BRANCHNAME`.

One of GitHub most crucial features is that it allows other users to easily create their own copy of your code and hosting that on GitHub as well by creating a "fork" of your code. That is, **a fork is a copy of your code that is hosted under another user's account and that is identical to your git repository** (including all commits) up to the point at which the fork was made. This allows other users to make changes to your code using `git` without needing write access to your version of the code, they can push changes to their own version of the code and make these available to other users via GitHub. People who fork your code cannot directly write to your GitHub repository and neither can you write to their fork of your repository (but GitHub prominently links back to the original version). The purpose of most forks is for other users to make changes to the code that will quickly or eventually be merged back into the main GitHub repository, but some codes have forks that are long-lasting and never re-unite with the original repository. To create a fork, navigate to a repository's GitHub page and click the Fork button at the top right. If during work in a fork you want to merge in subsequent changes made in the original repository, you will need to tell the clone

of your fork about the original repository. The original repository is normally called the “upstream” and you add it as a remote repository as, e.g.,

```
git remote add upstream https://github.com/jobovy/code-packaging-  
→minicourse.git
```

if you have forked the repository containing these notes and want to merge in changes made in the original repository. Then you can pull in changes from an “upstream” branch with, e.g.,

```
git pull upstream main
```

to pull in changes from the upstream main branch. Note that the “upstream” in this command is simply the shorthand for the URL that you added with the `git remote add` command.

**The main mechanism for merging changes made in a fork of a repository back into the main repository is through a “pull request”**, essentially a request to `git pull` the changes from the fork into the main repository (although what’s actually done is a `git merge`). A fork is essentially like a set of branches, where all of the original repository’s branches are present as duplicates of the original’s and users can add additional branches. Merging changes made in a fork is essentially the same as merging changes from a branch as we discussed [above](#) (page 21), with the only difference being that the fork is hosted remotely. Every GitHub repository has a tab called “Pull requests”, which lists the currently-open and previously-closed pull requests. To initiate a pull request, either go to the main repository’s `Pull requests` tab and click “New pull request” or go to your fork’s page, which has a `Pull request` button that would initiate the pull request. When you open a pull request, you should give a brief rationale behind the change that is being asked to be merged in. It is good practice not to make changes to a fork’s main branch, but to instead create a new branch to implement changes and then initiate a pull request from this branch to the original main branch. For one thing, this will allow the original repository’s owner to check out your fork’s branch more easily if this becomes necessary in the review or merging process.

When you open a pull request, the original code repository’s owners will likely ask you additional questions about the changes, to edit the changes to abide by the main repository’s coding style, to make sure that documentation/tests are updated (e.g., the log of changes), etc. and GitHub allows this conversation to happen on the page associated with the pull request. Keep in mind that your pull request may be rejected by the code’s owners: maybe it is implementing a new feature that they do not wish to support and maintain in the future (any new feature will entail a maintenance burden that will typically fall on the code’s owners rather than the person implementing it in a fork), or maybe they want more explanation/documentation/tests and you are not willing to provide this. Large pull requests may be difficult to review, so good pull requests are typically small (you may be able to split up a big change into smaller, atomic pull requests if each one can stand on its own). If you are concerned about doing a lot of work that might get rejected, contact the authors through the communication channel(s) that they prefer *before you start the work*, so you can find out whether they would be amenable to a pull request or not (you may want to do this by opening an Issue [see below] if there is no other obvious way to contact the code owners/maintainers).

Merging pull requests proceeds in the same way as merging between branches in a `git` repository, with the main difference being that if the merge can be done automatically, the merge can be done

entirely through the GitHub site of your code, with no need to check out the fork's code on your own machine. If there is a merge conflict, you have to check out the fork's code and manually merge them (although you will likely want to ask the fork's author to do this on their side, unless it requires deeper knowledge of the code than the fork's author can reasonably be expected to have). GitHub has extensive support for helping in the review of pull requests, allowing you to make comments on all of the changes and request additional changes, asking for reviews from particular contributors before approving the changes to be merged, and running any automated tests that you have and reporting their results. If you are expecting to have pull requests be a common way for your code to evolve, it is essential to have an *automated test suite* (page 71) run with *continuous integration* (page 95) that covers most of the lines in your code, to protect against unforeseen issues when changes are merged into the main code's repository.

Pull requests are the most important social aspect of how your code can grow when it's hosted on GitHub, but GitHub has many more features for the community of your users. **A helpful README file is a great way to introduce your code to your users** and READMEs can have a variety of formats that allow nice-looking GitHub sites to be created (don't have one of those drab pure README GitHub sites, use a Markdown README.md or a reStructuredText README.rst to create an attractive first impression for your code). Like the "Pull requests" tab, there is also an **"Issues" tab that provides a venue for users of your code to report issues with its installation or use**. Anybody with a GitHub account can open an Issue, which then goes into a list of open issues to be resolved. Any given issue typically consists of a conversation between the user reporting the issue and the code's maintainer(s) to figure out the root of the issue and commits that resolve the issue. Each issue has a unique number that you can reference in code commits as #NUMBER and GitHub will automatically link this commit to the issue online (you can even close issues through commits, by writing phrases like "fixes #NUMBER" in the commit! But make sure that the commit actually fixes the issue, because otherwise you will have to re-open it). When you are reporting an issue, it is important to write up a useful description of the issue: succinctly explain what the issue is, give the version of the code that you are using and the version of any other relevant component (e.g., the Python version, your operating system, etc.), and try to create a *minimal, reproducible example* of the issue which allows the maintainer to quickly reproduce the issue themselves and which can form the basis of a test added to the code's test suite checking that the issue is and remains resolved. Report any errors in full, using a service like *pastebin* to paste large logs (to not clog up the Issues page). When you open an issue, respond promptly to any follow-up questions (don't open an issue just before going on vacation!) and make sure to close the issue once it has been resolved.

GitHub has many more features than the basic ones that I have discussed here, many of them having to do with the integration with automated documentation and testing tools that I will discuss later in these notes.



## DOCUMENTING YOUR CODE AND HOSTING THE DOCUMENTATION ONLINE

Writing good documentation for your code is essential to allowing others to use it and it is crucial for lowering your own burden of supporting users of your code. Having excellent documentation that is easily accessible and that is up-to-date for the latest version of your code at all times allows people to use your code without having to constantly contact you with questions about the code (which many people will anyway not do, but they will rather simply not use your code if they cannot easily figure it out). Documentation is also important for your own use of the code: a few months after you've written a piece of code, it will not be immediately clear any longer how to use it (for me this can be as quick as a few days!), so by writing good documentation, you will also help *yourself* save time in the future from having to reconstruct how your own code works.

In this chapter, I first discuss the basics of how to write good documentation and then I discuss various software tools that make writing good, up-to-date documentation easy and that allow you to share the documentation online.

### 4.1 Basics of good documentation

Before starting a discussion of what makes for good code documentation, it is worth re-stressing the importance of *making your code easy and intuitive to use*, with many of the basic features taking at most a few lines to run. When that is the case, users will have to consult the documentation much less often than when your code is difficult to use or when even using a basic feature of your code requires them to write dozens of lines of code (e.g., setting up many related objects or many configuration options in a complicated way). It will also make your documentation much easier to write, because you will be able to illustrate your code's use with short, copyable code snippets, which makes the documentation much more pleasurable to read.

What's most important about documentation is that it is **as complete as possible and as up-to-date as possible**. Both of these are difficult to achieve, which is why using automated tools such as those discussed below is useful, because they can help significantly with achieving this goal. It is important that your documentation is as complete as possible, because otherwise users will run into undocumented features and need to contact you or give up. The only reasonable features to ex-



clude from the strict complete-documentation requirement are internal features that users shouldn't use; even then it is good practice to document them (albeit perhaps at a lower level of formatting clarity) for your own and other code developers' use. Documentation should be up-to-date to avoid mis-use of your code after major changes and to again avoid user frustration when they find that the documentation is out-of-date with the code and they cannot figure out how to use it. In addition to using automation tools to help you out, the best way to achieve complete and up-to-date documentation is to start writing documentation as soon as you implement new features and even *before* you implement them. That is, ideally you would write a first draft of the documentation of a function or class before implementing a first version of it, which has the added benefit of requiring you to think through carefully what you want the function or class to do, what inputs to take, and what outputs to return (similarly for tests later, ideally one would write them before writing the code). This is a hard ideal to achieve in practice, but it is good to write at least some documentation in parallel with the first implementation of the code. That way, your documentation will be complete. Keeping it up-to-date requires you to make sure to immediately update the documentation when you change the function.

Good documentation should cover at least the following sub-components:

- *A guide to the installation of your package*, discussing any pre-requisites. Your code should be able to be installed with standard installation commands, but even so it is good to list the commands (especially if you have both `pip` and `conda` installation options available, it is necessary to alert users).
- *A quick-start guide and a set of brief tutorials*: This helps users to get started using your code quickly by copying and pasting example code and it's a good way to show off what your code can do without requiring people to run it.
- *A full API* (Application programming interface): a complete listing of all of your code's functions, classes, and their methods. This is a reference guide that users can consult to learn about exactly how each feature works and what its options are.

Your code's **installation guide** should cover the typical way in which your package is installed. This can be as easy as explicitly stating that your code should be installed with `pip` as

```
pip install exampy
```

(for our example package from [Chapter 2](#) (page 7)). This may seem obvious to you, but it is useful to explicitly give the command, people love to simply copy-and-paste code (and we will show below how to add automatic copy-to-clipboard buttons like the one above). If your code has dependencies that wouldn't be easily and unobtrusively installed by `pip` (which will attempt to install all requirements listed in the `install_requires` part of your `setup.py` file, as we discussed in [Chapter 2](#) (page 11)), then it is useful to list how to install these as well, again giving explicit commands as much as possible, e.g.,

```
conda install numpy scipy
```

or



```
pip install numpy scipy
```

if your dependencies are `numpy` and `scipy`. Especially if your code requires harder-to-install dependencies or non-Python libraries (like the [GSL](#), which provides many scientific functions in C and is often used in C backends of Python packages), it is helpful to give commands for how to install these on different operating systems (the GSL is now luckily available on `conda-forge`, so the easiest way to install it is `conda install -c conda-forge gsl`). The installation guide is also a good place for a ‘frequently-asked-questions’ (FAQ) section with common installation problems. Again, if your code is pure Python with few dependencies, just stating that your code can be installed with `pip install` is likely all you need to say here.

The **quick-start guide** is a way to show off what your package can do and a place to give your users some code snippets that they can start adapting for their own use. When potential users of your package first look at your code they will be deciding whether or not using your package and going to the trouble of installing it and learning to use it is worth it for them. Therefore, a page in your documentation that demonstrates features of your code while also serving as a way to get started is a good way to get people to start using your code. The key to a good quick-start guide is to keep it brief and simple, but also get to interesting use of your code to show off what it can do; achieving these two somewhat competing goals is again easier if your code is *easy and intuitive* to use, because you can do impressive things with your code with very few keystrokes and, thus, you can write a good quick-start tutorial that also shows of what amazing things your code can do. It is difficult to keep these quick-start guides updated, so it is worth spending a bit of time carefully thinking what you want it to cover and to only cover very stable features of your code.

You can complement the quick-start guide with **a more extensive set of tutorials** that go into more detail. In practice, most outside users of your code (i.e., not yourself or your collaborators) will likely only use features that are clearly documented and for which a usage example exists, because most users will not attain a full understanding of all that your code can do (e.g., when combining different aspects of it that aren’t obvious) to allow them to go far beyond the tutorials that you provide. So a set of tutorials is where you can go over all of the most common use cases of your code and all the things that you think people can use your code for. It is important to keep them clear and succinct (with pointers for more advanced use), but it is difficult to write too many tutorials (just like it is difficult to write *too much documentation*), so don’t hold back (keeping your own time in mind of course).

Finally, **a complete API** should contain documentation for every function and class and every method in a class in your code, arranged by sub-module. The objective of this is to fully document your code, so users can get information on the inputs and outputs of all of your code’s functionality. The API should be arranged in a logical manner, grouping functions and classes with similar functionality. This is a part of your documentation where you should do a minimal amount of manual work in the documentation itself, but rather you should use automated tools to directly grab documentation from your code itself, in your functions’ and class’ *docstrings*, which I discuss next.

## 4.2 Python docstrings

Python has a built-in mechanism to attach documentation to modules, functions, and classes and their methods: [docstrings](#). Docstrings are a place to put documentation for users of your code, that is, the type of documentation that we are interested in here. Docstrings are *not* for developers: don't use them to comment on specifics of the implementation or on how the code works, unless this is necessary for users of your code; for developer notes, use regular comments in the code (in Python: lines that start with #).

Docstrings are simply regular strings that by virtue of their placement in the code get attached to a module, function, class, or method as its documentation. They do not need to be explicitly assigned as documentation, rather, the Python interpreter does this assignment automatically when it encounters a string in the correct place. This location is as follows:

- For *functions*: immediately following the statement that defines the function `def func(a, b, c=0):`, that is, between the `def` statement and the function body.
- For *classes*: immediately following the statement that defines the class `class a_class(object):`, that is, between the `class` statement and the class body.
- For *methods in a class*: immediately following their definition using `def`, in the same way as for functions.
- For *modules* and *submodules*: at the very top of the file defining the module.

I will give examples of these using the `exampy` example package that we set up in [Chapter 2](#) (page 8). When the Python interpreter encounters a string in the place specified above, it binds this string to the `__doc__` attribute of the module/function/class/method, where it is available to any user. Variables cannot have docstrings in Python itself (that is, the Python interpreter does not bind these to the variable's `__doc__` attribute), but many documentation tools will pick up docstrings immediately following a variable's assignment in the source code:

```
frac_out= 0.25
"""Fraction of the data that is considered an outlier"""
```

While docstrings can be any string, the convention is to use triple-quoted strings of the type `"""A triple-quoted string"""`, because most docstrings contain multiple lines, which is only allowed for triple-quoted strings. Thus, even if you have a docstring that is just a single line (which should rarely be the case), use a triple-quoted string. A good docstring should contain at least: (a) a brief description of what the module/function/class/method does, (b) an explanation of any input arguments and keywords, and (c) a discussion of any return value(s) or, for functions and methods, the lack thereof (it's useful to know that a function does *not* return anything, unless this is obvious, such as with a class' `__init__` function). You can include extra information such as possible failure modes or references as well. While there are many standard formats for docstrings, one of which I will discuss below, you do not have to follow a standard format, but it is important to use a consistent style throughout your package such that users can easily parse the documentation once they are used to your format.

As an example, we can write a docstring for the top-level module of the `exampy` package. To do this, we edit the `exampy/__init__.py` file such that it now looks like

```
"""exampy: an example Python package"""
from ._math import *
```

and the `"""exampy: an example Python package"""` string then becomes the module's docstring. To verify this, open a Python interpreter and do

```
import exampy
?exampy
```

which shows a message that says something like

```
Type:      module
String form: <module 'exampy' from '/PATH/TO/exampy/exampy/__init__.py'>
File:      /PATH/TO/exampy/exampy/__init__.py
Docstring: exampy: an example Python package
```

and in which you see the docstring that we just defined. You can also verify that it was indeed attached as the module's `__doc__` attribute:

```
print(exampy.__doc__)

exampy: an example Python package
```

You should only use one-line docstrings for modules, submodules, and classes, because these do not have direct inputs and outputs, so all of the documentation can easily fit on a single line (however, you should feel free to have a multi-line docstring if there is more to say). A class' docstring simply describes the purpose of the class, *not* how to initialize the class or details on its methods (although it could contain a list of attributes or methods; this isn't generally considered to be necessary); a class' initialization should be documented as the docstring of the class' `__init__` function, just like any regular method as I discuss below.

Functions and methods typically have inputs and outputs in addition to the brief description, and these inputs and outputs should be separated onto their own line each; to keep a uniform style for your documentation, you should therefore also use multi-line docstrings for functions that have no inputs or outputs, stating explicitly that there are no arguments or keywords and no outputs. Methods in a class are functions that are defined as part of a class and they are essentially the same as regular functions, except that their first argument is `self` as the representation of the class instance. `self` is not typically listed as a documented argument of a method, because it is always the first argument of a method and it always has the same meaning. Therefore, methods and functions follow the same documentation rules. I will discuss documentation for functions below, but keep in mind that the same considerations apply to methods in exactly the same way.

While there are many standard docstring formats, for scientific code packages it is simplest to follow `numpy`'s [docstring convention](#) (also used, e.g., by `scipy`, packages in the `scikit` series, and

astropy). The most basic version of a docstring for a function should contain a description, inputs list, and outputs list in the numpy docstring format looks as follows, using as an example the `exampy.square` function that we defined in `exampy/_math.py` in [Chapter 2](#) (page 8):

```
def square(x):
    """The square of a number

Parameters
-----
x: float
    Number to square

Returns
-----
float
    Square of x
    """
    return x**2.
```

The brief description is followed by a *Parameters* section that lists each argument and keyword with the format

```
parameter: type
    Parameter description
```

Similarly, the return value is described as

```
type
    Description of return value
```

If your function returns multiple values, *Returns* becomes a list as well; in that case, you may want to name your return values for extra clarity and follow the same format for each as that for each input parameter. If your function does not take any arguments or keywords, you can simply state

```
Parameters
-----
None
```

Similarly, if your function does not return anything, you can use

```
Returns
-----
None
    None
```

where the two “None”s are necessary, because the sphinx typesetting of the numpy-style docstrings that I will discuss below breaks the return section up into the two components “type” and “Description of return value”.

If we then run

```
?exampy.square
```

a message shows up that looks as follows

```
Signature: exampy.square(x)
Docstring:
The square of a number

Parameters
-----
x: float
    Number to square

Returns
-----
float
    Square of x
File:      /PATH/TO/exampy/exampy/utils.py
Type:      function
```

We can again check that the docstring was indeed assigned to the function’s `__doc__` attribute with

```
print(exampy.square.__doc__)
```

```
The square of a number

Parameters
-----
x: float
    Number to square

Returns
-----
float
    Square of x
```

For most functions, you will want to include a longer description than the one-line description that we could use for the square function above. In that case, you would still start the docstring with a one-line summary, but also provide an extended description after two line breaks. For example, for a verbose `exampy.square` docstring

```
def square(x):
    """The square of a number

    Calculates and returns the square of any floating-point number;
    note that, as currently written, the function also works for
    arrays of floats, ints, arrays of ints, and more generally,
    any number or array of numbers.

    Parameters
    -----
    x: float
        Number to square

    Returns
    -----
    float
        Square of x
    """
    return x**2.
```

If a function has optional keyword arguments, the documentation should make it clear that these are optional, either by adding , optional after the parameter's type or by stating this in the description of the parameter (but the first method is most clear). You can also specify what the default value of the keyword is, but this is not really necessary, because most documentation tools will display the function's signature, which normally shows the default value. For example, we can document the `exampy.integrate.riemann` function in `exampy/integrate/_integrate.py`; with documentation, that function looks like

```
def riemann(func,a,b,n=10):
    """A simple Riemann-sum approximation to the integral of a function

    Parameters
    -----
    func: callable
        Function to integrate, should be a function of one parameter
    a: float
        Lower limit of the integration range
    b: float
        Upper limit of the integration range
    n: int, optional
        Number of intervals to split [a,b] into for the Riemann sum

    Returns
```

(continues on next page)

(continued from previous page)

```
-----
float
    Integral of func(x) over [a,b]
"""
```

If we then request the documentation for the `riemann` function

```
import exampy.integrate
?exampy.integrate.riemann
```

we get a message that says something like

```
Signature: exampy.integrate.riemann(func, a, b, n=10)
Docstring:
A simple Riemann-sum approximation to the integral of a function

Parameters
-----
func: callable
    Function to integrate, should be a function of one parameter
a: float
    Lower limit of the integration range
b: float
    Upper limit of the integration range
n: int, optional
    Number of intervals to split [a,b] into for the Riemann sum

Returns:
-----
float
    Integral of func(x) over [a,b]
File:      /PATH/TO/exampy/exampy/integrate/_integrate.py
Type:      function
```

and we see that the function signature includes the default value of `n` even though the docstring didn't specify it. If the default value of a keyword is the result of calling a function, such that it isn't immediately clear what the default value is from the function signature or how it is calculated, you probably want to state it in the docstring.

Additional commonly-used sections of a function's docstring are (each following the

```
SECTION
-----
```

format) are:

- **Raises:** a list of exceptions that the function may raise and when it raises them.
- **See Also:** a list of related functions; automated documentation tools will be able to link these automatically if you list them in the same way that you would import and use them (e.g., in `square` above you can refer to `cube`, if such a function exists in the same file, but to refer to `riemann` which is in `exampy.integrate`, you need to say explicitly `exampy.integrate.riemann`).
- **Notes:** extended notes on the function. Use this to list calculation or implementation details that you think the user should be aware of. You can also use this section to give details on the history of the function, keeping track of major changes and when they occurred. For example,

History:

2020-03-01: First implementation - Bovy (UofT)

2020-04-06: Added new keyword `Y` to allow for `Z` - Bovy (UofT)

- **References:** a list of bibliographic references, using the format

```
.. [1] J. Bovy, "galpy: A Python Library for Galactic Dynamics,"  
    Astrophys. J. Supp., vol. 216, pp. 29, 2015.
```

As a full example, we implement a docstring for a new `exampy.cube` function that computes the cube of a number, which is located in `exampy/_math.py` (see [Chapter 2](#) (page 8)) and display it here:

```
print(exampy.cube.__doc__)
```

The cube of a number

Calculates and returns the cube of any floating-point number; note that, as currently written, the function also works for arrays of floats, ints, arrays of ints, and more generally, any number or array of numbers.

Parameters

-----

`x`: float

    Number to cube

Returns

-----

float

    Cube of `x`

Raises

(continues on next page)



(continued from previous page)

```

-----
No exceptions are raised.

See Also
-----
exampy.square: Square of a number
exampy.Pow: a number raised to an arbitrary power

Notes
-----
Implements the standard cube function

.. math:: f(x) = x^3

History:

2020-03-04: First implementation - Bovy (UofT)

References
-----
.. [1] A. Mathematician, "x to the p-th power: squares, cubes, and their
    general form," J. Basic Math., vol. 2, pp. 2-3, 1864.

```

In the Notes section here, I also illustrate how LaTeX math can be used to typeset equations (more on that below).

That docstrings are simply a submodule/function/class/method's `__doc__` attribute means that they can be generated, parsed, and modified programatically. That is, you can also specify a docstring by explicitly setting the `__doc__` attribute, you can automatically extract information from the docstring by parsing it as you can any string in Python, or you can modify the docstring (e.g., adding additional information to it). This is, for example, useful when you are defining functions programatically, e.g., automatically defining a set of functions with similar functionality; then you can add documentation to these automatically generated functions by explicitly setting their `__doc__` attribute. For example, we set the docstring for the `exampy.integrate` submodule by editing `exampy/integrate/__init__.py` to be

```

__doc__ = """exampy.integrate: submodule with utilities for calculating
the integral of functions"""
from ._integrate import *

```

When we then do

```
?exampy.integrate
```

We get a message that says something like

```
Type:      module
String form: <module 'exampy.integrate' from '/PATH/TO/exampy/exampy/
→integrate/__init__.py'>
File:      /PATH/TO/exampy/exampy/integrate/__init__.py
Docstring:
exampy.integrate: submodule with utilities for calculating
the integral of functions
```

that is, we see that the docstring was correctly attached.

As an example of documenting a class, we add a `Pow` class to `exampy/_math.py` to calculate an arbitrary power of a number. We add an `__init__` function to setup the power to raise numbers to and a `__call__` function to raise a number to the object's power. With documentation, the class looks as follows

```
class Pow(object):
    """A class to compute the power of a number"""
    def __init__(self,p=2.):
        """Initialize a PowClass instance

Parameters
-----
p: float, optional
    Power to raise x to
"""
        self._p= p

    def __call__(self,x):
        """Evaluate x^p

Parameters
-----
x: float
    Number to raise to the power p

Returns
-----
float
    x^p
"""
        return x**self._p
```

We see that we follow the class definition statement `class Pow(object):` with a docstring that briefly describes what the class does and then we document the two methods `__init__` and `__call__` just as we would normal functions, leaving the `self` argument undocumented. If we ask for the docstring of a class instance, we get the overall class docstring:

```
po= exampy.Pow(p=4.)
print(po.__doc__)
```

```
A class to compute the power of a number
```

if we ask for the help for the class, we get the class docstring *and* the docstring for how to initialize the function:

```
?exampy.Pow
```

gives something like

```
Init signature: exampy.Pow(p=2.0)
Docstring:      A class to compute the power of a number
Init docstring:
Initialize a PowClass instance

Parameters
-----
p: float, optional
    Power to raise x to
File:          /PATH/TO/exampy/exampy/_math.py
Type:          type
Subclasses:
```

and if we ask for the help for the instance, we get all three docstrings, because calling the instance is also how you access the `__call__` method:

```
?po
```

gives something that looks like

```
Signature:      po(x)
Type:          Pow
String form:    <exampy._math.Pow object at 0x1179e2c18>
File:          /PATH/TO/exampy/exampy/_math.py
Docstring:      A class to compute the power of a number
Init docstring:
Initialize a PowClass instance

Parameters
```

(continues on next page)

(continued from previous page)

```
-----
p: float, optional
    Power to raise x to
Call docstring:
Evaluate x^p

Parameters
-----
x: float
    Number to raise to the power p

Returns
-----
float
    x^p
```

At the risk of sounding like a broken record, I will once more repeat that you should write these docstrings *as soon as you implement a function and ideally before you implement it* and you should implement a full docstring of your preferred format from the get-go. If you do this enough times, it will become second nature and you will not even be able to imagine writing code and documentation in any other way!

### 4.3 Using sphinx to write and generate documentation for your package

Docstrings are *the* way to document each submodule, function, class, and method that your package consists of, but if you want to go further and create an (online) manual, more work is required. The main tool used for this in the Python eco-system is [sphinx](#), which was originally created for the documentation of the Python language and standard library itself, but it is a very general documentation system that can be used for any Python package and beyond. These notes, for example, are created using sphinx.

There are [many ways to install sphinx](#), but the easiest way is to simply install sphinx from PyPI as

```
pip install sphinx
```

In addition to the basic sphinx package, additional sphinx functionality can be obtained by installing a variety of extensions, some of which I will discuss below.

sphinx uses the [reStructuredText](#) mark-up language for writing documentation pages that can be rendered as HTML, LaTeX, or any of a large number of other formats. Thus, a single manual written in reStructuredText can be used to create online HTML help pages, a PDF manual, an e-

book version, etc. The PDF version of these notes that is linked on the [Contents page](#) is generated from the same underlying files as the HTML version. reStructuredText is similar in spirit to the [Markdown](#) mark-up language that is typically used for READMEs and that can be used in GitHub Issues and Pull Request comments, but in detail has a different syntax. I will cover the basics of reStructuredText as part of the discussion in this chapter, but these notes do not try to give a full overview of reStructuredText.

The easiest way to get started with building a sphinx manual for your package is to create a directory that will contain the manual in the top-level directory of your code, named `doc/` or `docs/`. For example, for the `exampy` example package, our top-level directory looks now as follows

```
TOP-LEVEL_DIRECTORY/  
  docs/  
  exampy/  
  README.md  
  setup.py
```

Then go into the `docs/` directory and start the sphinx manual by running

```
sphinx-quickstart
```

This script asks you a few basic questions to set up the skeleton of the manual. It asks whether you want to “Separate source and build directories (y/n) [n]”, which is to say, either have separate `source/` and `build/` directories containing the manual’s source (which you will edit) and the manual’s builds for different formats (which are automatically generated); alternatively, if you say “n”, the builds will go into a `_build/` sub-directory of the `source/` directory. The current default is the latter, but I strongly prefer separating the `source/` and `build/` directories to keep a clean separation between source and automatically-generated builds (for one thing, this makes managing your documentation’s changes with `git` much easier, because you will not as easily accidentally add built files to the `git` repository).

Next, the script asks for the project’s name, which typically should be the name of your package (`exampy` in our example case); the author’s name; the current release (typically you should start with “0.0.1” or “0.1” depending on how fine-grained you want to version your code), and the language of your documentation. All of these can be changed later (some more easily than others). After this, sphinx sets up the basic structure of the manual and how to build it: for the current version of sphinx, the contents of the `docs/` directory after running `sphinx-quickstart` is

```
build/  
source/  
  _static/  
  _templates/  
  conf.py  
  index.rst  
Makefile  
make.bat
```

Here, the `Makefile` and `make.bat` are files that allow different formats of your documentation to be built using `make`, the [standard build tool](#). The `build/` directory is initially empty and will contain builds of different formats of your manual; for example, later it will contain a `html/` directory with the HTML version of your manual and perhaps a `latex/` directory with the LaTeX and PDF versions of your manual.

The `source/` directory is where you write the manual itself as a set of reStructuredText (`.rst`) files. `sphinx-quickstart` has populated a skeleton for the main index page of your manual that is its entry page and lists its contents, it looks something like

```
.. exampy documentation master file, created by
   sphinx-quickstart on Wed Mar  4 20:11:34 2020.
   You can adapt this file completely to your liking, but it should at
   ↪least
   contain the root `toctree` directive.

Welcome to exampy's documentation!
=====

.. toctree::
   :maxdepth: 2
   :caption: Contents:


Indices and tables
=====

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

The statement following the `.. directive` is a comment that is not displayed in the manual. All reStructuredText directives start with `..`, these [directives](#) are used for figures, code examples, math, etc. The “Welcome to exampy’s documentation!” is the main title header on the page and title headers in reStructuredText are indicated by a style of underlining (here “=====”); note that there is no specific order of different underlining styles for different types of headers, reStructuredText automatically figures out what the hierarchy is based on the different styles that you use (so you can use “=====” for the first, main header, then use ‘———’ for each sub-title within this section, another “=====” underlined title for a second title at the highest-level, and more “———” sub-titles below that, perhaps even a ‘\*\*\*\*\*’ underlined sub-sub-title...).

The `.. toctree::` directive is the most important part of the index page and it contains the main table of contents as a set of files that contain the documentation: a sphinx manual is a set of pages that are all part of some table-of-contents (some `toctree`), either listed in this main table of contents, or in a `.. toctree::` directive that gives a table of contents on a page included in the main

`toctree`. sphinx does not like pages that are not part of *any* `toctree` directive and it will warn if it encounters one of these; these pages will still be processed and you can link to them, but sphinx won't include a link to them automatically. To populate the `toctree` we will create a set of pages in the `source/` directory, such as `installation.rst`, `intro.rst`, etc. and to include these in the manual, the main `toctree` looks like

```
.. toctree::
    :maxdepth: 2
    :caption: Contents:

    installation.rst
    intro.rst
```

Note that it is not necessary to include the `.rst` part of the filename, but when you are including different types of files (as we will discuss below), it is useful to make it explicit what the format of each page is. When you include the manual's chapters like this, sphinx automatically grabs the titles from each file to make an entry in the table of contents, but you can also specify a custom title here, by doing

```
.. toctree::
    :maxdepth: 2
    :caption: Contents:

    Installation instructions <installation.rst>
    intro.rst
```

The `toctree` directive has [many options](#). For example, the `:maxdepth: 2` option specifies that the displayed table of contents should include the main sections of each page (like for the table of contents of these notes); setting it to 1 would only include the title of each page/chapter, setting it to 3 would include subsections. Some of the more commonly used options are

- `:caption:` A `CAPTION` a caption for the table of contents; used as the title of the table-of-contents part of the page.
- `:numbered:` by default, entries in the table of contents are not numbered, but setting this option numbers them.
- `:name:` `a_name` a name to use to reference the table of contents, that is, to create internal links to the table of contents using the `:ref:` mechanism discussed below

The final part of the automatically-generated index page is “Indices and tables”, which contains an automatically-generated index and search function. I don't personally find these very useful typically and they can be removed from the `index.rst` file to remove them from the manual without any adverse affect.

The two directories `_static/` and `_templates/` that `sphinx-quickstart` generated in the `source/` directory are empty. For most basic documentation needs, you will not need to populate these, but they are used to: store “static” files in `_static/` such as css style files to customize

the look of your manual's HTML page or any extra Javascript code that you might want to use on your manual's HTML page (this is not typical); and to store changes to the default page templates that sphinx uses to create HTML, etc. pages from your reStructuredText source. sphinx is highly customizable, but typical users do not need to worry about all of these customization options and directories.

Finally, sphinx-quickstart created a `conf.py` file in the `source/` directory. This is the configuration file for your manual, which is a Python script itself and which tells sphinx how to build your manual. At first, it looks like

```
# Configuration file for the Sphinx documentation builder.
#
# This file only contains a selection of the most common options. For a
# full
# list see the documentation:
# https://www.sphinx-doc.org/en/master/usage/configuration.html

# -- Path setup -----
#
# If extensions (or modules to document with autodoc) are in another
# directory,
# add these directories to sys.path here. If the directory is relative to
# the
# documentation root, use os.path.abspath to make it absolute, like shown
# here.
#
# import os
# import sys
# sys.path.insert(0, os.path.abspath('.'))

# -- Project information -----
#
project = 'exampy'
copyright = '2020, Jo Bovy'
author = 'Jo Bovy'

# The full version, including alpha/beta/rc tags
release = '0.1'

# -- General configuration -----
```

(continues on next page)



(continued from previous page)

```

# Add any Sphinx extension module names here, as strings. They can be
# extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
# ones.
extensions = [
]

# Add any paths that contain templates here, relative to this directory.
templates_path = ['_templates']

# List of patterns, relative to source directory, that match files and
# directories to ignore when looking for source files.
# This pattern also affects html_static_path and html_extra_path.
exclude_patterns = []

# -- Options for HTML output -----
↪-----

# The theme to use for HTML and HTML Help pages.  See the documentation.↪
↪for
# a list of builtin themes.
#
html_theme = 'alabaster'

# Add any paths that contain custom static files (such as style sheets)↪
↪here,
# relative to this directory. They are copied after the builtin static↪
↪files,
# so a file named "default.css" will overwrite the builtin "default.css".
html_static_path = ['_static']

```

At first, this configuration file basically just contains the info that you provided to `sphinx-quickstart`: the name of the project, the author, a copyright string created from the current year and the given author, and the version that you provided. Then there is a section for any sphinx extension that you use (like the ones that I discuss below) and a statement about templates being in the `_templates/` directory. The last section is specific to the HTML build of the manual: the theme (default: `alabaster`, but there are [many theme options](#)) and the path where additional static files are located (the `_static/` directory discussed above). If you are creating other formats as well and want to customize these, you can add sections for LaTeX output etc. below this. A full list of options for the `conf.py` file is available [here](#). The `conf.py` file is a list of Python commands and it is executed whenever the documentation is built; this allows you to run arbitrary Python code during the build of your manual (e.g., this is how these notes add the `git` revision hash for the

current version on the main page and in the PDF filename).

To create the documentation, we use `make`. Simply running `make` in the `docs/` directory returns a list of make options:

```
make
```

gives

```
Sphinx v2.2.0
Please use `make target' where target is one of
  html          to make standalone HTML files
  dirhtml       to make HTML files named index.html in directories
  singlehtml    to make a single large HTML file
  pickle        to make pickle files
  json          to make JSON files
  htmlhelp      to make HTML files and an HTML help project
  qthelp        to make HTML files and a qthelp project
  devhelp       to make HTML files and a Devhelp project
  epub          to make an epub
  latex         to make LaTeX files, you can set PAPER=a4 or PAPER=letter
  latexpdf      to make LaTeX and PDF files (default pdflatex)
  latexpdfja    to make LaTeX files and run them through platex/dvipdfmx
  text          to make text files
  man           to make manual pages
  texinfo       to make Texinfo files
  info          to make Texinfo files and run them through makeinfo
  gettext       to make PO message catalogs
  changes       to make an overview of all changed/added/deprecated items
  xml           to make Docutils-native XML files
  pseudoxml     to make pseudoxml-XML files for display purposes
  linkcheck     to check all external links for integrity
  doctest       to run all doctests embedded in the documentation (if
→enabled)
  coverage      to run coverage check of the documentation (if enabled)
```

The option you will use most commonly for online software documentation is:

```
make html
```

which creates the HTML version of your manual in the `build/html/` directory (or in the `source/_build/html/` directory if you did not separate the `build/` and `source/` directories. For the basic version created by `sphinx-quickstart`, this creates an index page that looks like

**exampy**

Navigation

Quick search

Go

Welcome to exampy's  
documentation!

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

©2020, Jo Bovy. | Powered by [Sphinx 2.2.0](#) & [Alabaster 0.7.12](#) | [Page source](#)

We then start writing the documentation, starting with adding the `installation.rst` and `intro.rst` files that I mentioned above. The `installation.rst` file contains the basic installation instructions and looks like

Installation instructions

=====

Dependencies

-----

``exampy`` requires the use of ``numpy`` <<https://numpy.org/>>`\_\_.

Installation

-----

``exampy`` is currently not yet available on PyPI, but it can be installed by downloading the source code or cloning the GitHub repository and running the standard::

```
python setup.py install
```

command or its usual variants (``python setup.py install --user``, ``python setup.py install --prefix=/PATH/TO/INSTALL/DIRECTORY``, etc.).

For more info, please open an Issue on the GitHub page.

where I use the different underlining styles of headers discussed above to create a main page title and two sections. The first sentence contains a link to an external website, the `numpy` website in this case. The words enclosed in `“`”` highlight them as “code” and the separate indented line following a double colon `“::”` give a code block, which will be type-set in a special way (this indented block can consist of multiple lines for a multi-line code-block). Code blocks can also be created using the `.. code-block::` directive, which sets the highlighting language (default: Python) and then gives the code example, e.g.,

```
.. code-block:: python
```

(continues on next page)

(continued from previous page)

```
python setup.py install
```

instead of the indented statement following “::” above. This is illustrated further in the `intro.rst` page, which looks like

### Introduction

=====

```exampy``` is an example Python package that contains some very basic math functions. As an example, we can compute the square of a number as::

```
>>> import exampy
>>> exampy.square(3.)
# 9.
```

Similarly, we can compute the cube of a number:

.. code-block:: python

```
>>> exampy.cube(3.)
# 27.
```

A general method for raising a number to a given power is given by the ```Pow``` class. For example, to get the fourth power of 3, do::

```
>>> po= exampy.Pow(p=4.)
>>> po(3.)
# 81.
```

```exampy``` also includes a simple method for integrating a function, in the ```exampy.integrate``` submodule. This submodule contains the function ```riemann``` that approximates the integral of any one-parameter function as a Riemann sum. ```riemann``` takes as input (i) the function to integrate, (ii) the integration range's lower limit and (iii) the upper limit, and (iv) optionally, the number of intervals to divide the integration range in. For example, the integrate the square function of the range `[0,1]`, do::

```
>>> from exampy import integrate
>>> integrate.riemann(exampy.square,0,1)
# 0.35185185185185186
```

(continues on next page)

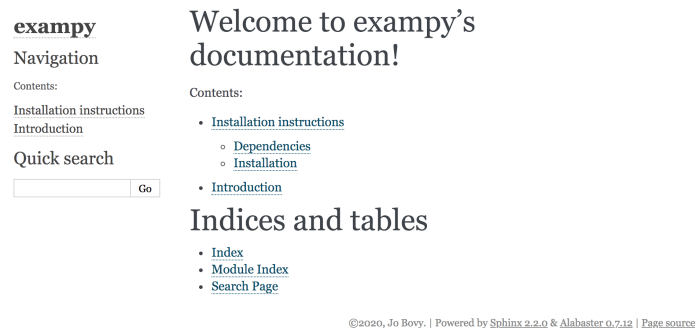
(continued from previous page)

If we increase the number of intervals from the default (which is 10), we get a better approximation to the correct result (which is 1/3)::

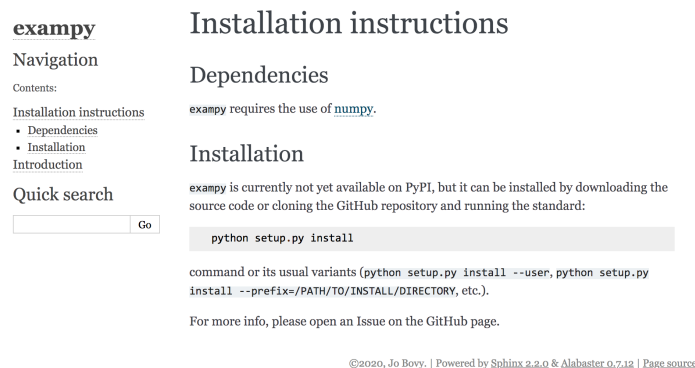
```
>>> integrate.riemann(exampy.square,0,1,n=1000)
# 0.33350016683350014
```

Here, I list the expected output with a comment such that these code-blocks can be copied verbatim into a Python interpreter or into a `jupyter` notebook cell without raising an error. An interactive Python session is represented by including the “>>>” prompt; this can be copied into the Python interpreter or into `jupyter` notebook cells without any problem.

We then also include the `installation.rst` and `intro.rst` pages into the `toctree` directive in `index.rst` as explained above and after that we run `make html` to get a set of pages that looks as follows: `index.html`:



where we see that the different sections in `installation.rst` are included because we set : `maxdepth: 2` in the `toctree` directive. The `installation.html` is:



and `intro.html`:

**exampy**

Navigation

Contents:

Installation instructions

Introduction

Quick search

## Introduction

**exampy** is an example Python package that contains some very basic math functions. As an example, we can compute the square of a number as:

```
>>> import exampy
>>> exampy.square(3.)
# 9.
```

Similarly, we can compute the cube of a number:

```
>>> exampy.cube(3.)
# 27.
```

A general method for raising a number to a given power is given by the `Pow` class. For example, to get the fourth power of 3, do:

```
>>> po= exampy.Pow(p=4.)
>>> po(3.)
# 81.
```

**exampy** also includes a simple method for integrating a function, in the `exampy.integrate` submodule. This submodule contains the function `riemann` that approximates the integral of any one-parameter function as a Riemann sum. `riemann` takes as input (i) the function to integrate, (ii) the integration range's lower limit and (iii) the upper limit, and (iv) optionally, the number of intervals to divide the integration range in. For example, the integrate the square function of the range `[0,1]`, do:

```
>>> from exampy import integrate
>>> integrate.riemann(exampy.square,0,1)
# 0.33185185185185186
```

If we increase the number of intervals from the default (which is 10), we get a better approximation to the correct result (which is  $1/3$ ):

```
>>> integrate.riemann(exampy.square,0,1,n=1000)
# 0.33350816683350814
```

©2020, Jo Bovy. | Powered by [Sphinx 2.2.0](#) & [Alabaster 0.7.12](#) | [Page source](#)

Thus, we now have the basic structure of a manual for our software package, which we can expand upon by adding material and by adding pages. To make the manual more useful and attractive, I give a brief overview of some commonly-used reStructuredText features that help with this and then describe how to automatically include docstrings from your code as part of the documentation.

## 4.4 A brief tour of reStructuredText

reStructuredText has many features that you can all use in sphinx documentation and these notes do not intend to give a comprehensive overview of these. The objective of this brief section is to get you underway with the most commonly used features of reStructuredText when building documentation; consult reStructuredText documentation and guides to learn about more advanced functionality. A good quick-start guide is available on the [reStructuredText webpage](#), where you can also find full documentation and a cheat sheet.

As I have already discussed above, the basic lay-out of a reStructuredText page is set by headings that are indicated by underlined lines. The type of symbol that you use to underline does not matter, reStructuredText automatically figures out the hierarchy of titles and sub-titles based on your use of different underline styles (“`——`” or “`+++++`” or “`=====`” or ...); this of course requires you to use these consistently! If reStructuredText outputs a hierarchy that you did not intend, you probably made a mistake in the consistency of your underline styles.

Basic italic emphasis is done by enclosing a word or sentence in single `*` as “`*emphasize this*`” (rendered as *emphasize this*); bold emphasis is obtained by enclosing in double `**` as “`**strongly emphasize this**`” (rendered as **strongly emphasize this**). Double back-quotes are used for fixed-width formatting, such as used when displaying inline code or as another way of emphasis (as

used often in these notes): “`fixed-width code/emphasis`” (rendered as `fixed-width code/emphasis`).

Lists start with a new paragraph (i.e., two line breaks) and are then simply a numbered or asterisked set of paragraphs. For example, this is an unnumbered list

```
* First list item

* Second list item

* Third list item
```

which is rendered as

- First list item
- Second list item
- Third list item

To get a numbered list, use numbers, e.g.,

```
1. First list item

2. Second list item

3. Third list item
```

which is rendered as

1. First list item
2. Second list item
3. Third list item

Instead of a simple number like this, you can use upper- and lowercase letters, upper- and lowercase roman numerals, and you can add a closing parenthesis “)” or enclose the entire number/letter in parentheses.

Block quotes are done with indentation, e.g. “

Lists start with a new paragraph (i.e., two line breaks) and are then simply a numbered or asterisked set of paragraphs. For example, this is an unnumbered list

(Python code packaging for scientific software, Jo Bovy).”

As already discussed above, code blocks can be written in two ways: (i) by ending the previous paragraph in a double colon “::” and having an indented code block, e.g.,

Using ``exampy``, we can compute the square of a number as::

```
import exampy
exampy.square(3.)
# 9.
```

This uses the default syntax highlighting language, which can be specified in the `conf.py` file by setting the `highlight_language` parameter and by default is currently essentially Python. Alternatively, you can insert a code block using the `code-block` directive where you can also directly specify the language if you want it to be different from the default for a specific code block:

**.. code-block:: language**

used as

Using ``exampy``, we can compute the square of a number as:

**.. code-block:: python3**

```
import exampy
exampy.square(3.)
# 9.
```

Both of these will render as “Using `exampy`, we can compute the square of a number as:

```
import exampy
exampy.square(3.)
# 9.
```

“

Images can be included using the `image` directive that looks like

**.. image:: image\_filename.ext**

which includes the image file with name `image_filename.ext`; options are

**.. image:: image\_filename.ext**

**:alt:** alternate text for the image (like in HTML)

**:height:** height of the image (in length units or % of original)

**:width:** width of the image (in length units or % of original)

**:scale:** integer percentage to scale height and width with

**:align:** "top", "middle", "bottom", "left", "center", or "right"



**:target:** if set, make the image a hyperlink to this URL.

Typically, you should keep images in a separate directory in your `source/` directory; you do not need to tell sphinx about these in any way, sphinx will figure out that this directory exists and copy over its relevant content. As an example, the first webpage image on this page was include using

```
.. image:: images/first-doc-index.png
   :width: 66%
   :align: center
```

You can also include LaTeX math in your manual, LaTeX builds will display this correctly and, using the [MathJax](#) Javascript library, LaTeX math can also be rendered in webpages; to include MathJax support, you need to add the `sphinx.ext.mathjax` extension to your `conf.py`'s `extensions` list. Then you can include inline math using the `:math:` directive as

An example of an inline math equation is Euler's identity `:math:`e^{i\pi}`  
`→+1 = 0`` for complex numbers.

which is rendered as “An example of an inline math equation is Euler’s identity  $e^{i\pi} + 1 = 0$  for complex numbers.” We can also display equations on separate lines or an entire math block using the `math` directive:

Euler's identity is

```
.. math::

    e^{i\pi}+1 = 0
```

which is rendered as

” Euler’s identity is

$$e^{i\pi} + 1 = 0$$

“

While you should avoid too much complex LaTeX math in docstrings, because they are often read as pure text, the manual is typically read as a HTML page with properly typeset math, so you should feel free to use as much LaTeX math as is necessary.

Finally, you likely want to include links to external webpages and internal links to other sections. External links in `reStructuredText` have the format ``link text <link-url>`__`. Internal links are a little trickier. If you want to link internally to a section or other part of the manual, you first need to create a label, for example,

```
.. _install-dependencies:

Dependencies
-----

``exampy`` requires the use of `numpy` <https://numpy.org/>`__.
```

which could be part of the `installation.rst` file. Then, you can internally link to this section as `:ref:`install-dependencies``, that is, using the name *without the initial underscore*. If you link to anything that isn't right before a section as in the example above, you need to give the link a title to be displayed as `:ref:`Link title <label-name>`, where `label-name` is again the name that you used to label *without the leading underscore*.

A full list of possible reStructuredText directives is available [here](#).

## 4.5 Including docstrings into the sphinx documentation

So far, the manual consists solely as manually-written help pages with notes on installation, a quick-start guide, and any examples and tutorials. However, the manual should also include a full reference of the documentation of all functions, classes, and methods in the software package: the API. One could write this manually as well, going through the code and making an API entry for each function, class, or method, copying the docstring by hand, but sphinx has tools to automatically extract docstrings that can be used to (semi-)automatically create the API. Besides requiring significantly less work to set up, this has the advantage that the function signatures and docstrings are always up-to-date with what's written in the code package's documentation itself.

As an example, we add a `reference.rst` file to the documentation that will contain the short API for the `exampy` package. sphinx's automation tools for documenting packages are contained in the `autodoc` extension, which you can access by adding “`sphinx.ext.autodoc`” to the `extensions` list in your `conf.py` file (this extension is part of sphinx, so does not need to be installed separately). The `autodoc` extension contains various *autoX* directives that will automatically grab your package's docstrings and display them in the documentation. Commonly used ones are

```
.. autofunction:: func
    Display the docstring for the function func

.. autoclass:: a_class
    Display the docstring for the class a_class

:members::
```

where `:members:` is a list of member methods (e.g., `method1`, `method2`) to also document, that is, display the docstring for; if the `:members:` option is set without any arguments, all public members (those whose name does not start with an underscore) are shown. Without the `:members:` option, only the class docstring is displayed.

```
.. automethod:: a_method
```

Display the docstring for the method `a_method`

These directives have additional options that are discussed on the autodoc page, but that are not commonly used.

Let's first create an API of just the functions in `exampy` with `autofunction`. A simple API is given by

```
API reference
=====

``exampy``
-----

.. autofunction:: exampy.square

.. autofunction:: exampy.cube

``exampy.integrate``
-----

.. autofunction:: exampy.integrate.riemann
```

Because we have used the `numpy` docstring style, we should add a further extension to create a nicely-formatted version of this docstring in the resulting documentation page, [napoleon](#), an official sphinx extension that supports `numpy` docstrings (also part of sphinx itself and thus not requiring special installation). We add `"sphinx.ext.napoleon"` to the list of extensions in the `conf.py` file. After also adding `reference.rst` in the toctree in the `index.rst` file and running `make html`, the following `references.html` page in the documentation is created:

## exampy

### Navigation

Contents:

[Installation instructions](#)

[Introduction](#)

[API reference](#)

▪ [exampy](#)

▪ [exampy.integrate](#)

### Quick search

## API reference

### exampy

`exampy.square(x)`

The square of a number

**Parameters:** `x` (*float*) – Number to square

**Returns:** Square of `x`

**Return type:** float

`exampy.cube(x)`

The cube of a number

Calculates and returns the cube of any floating-point number; note that, as currently written, the function also works for arrays of floats, ints, arrays of ints, and more generally, any number or array of numbers.

**Parameters:** `x` (*float*) – Number to cube

**Returns:** Cube of `x`

**Return type:** float

**Raises:** No exceptions are raised. –

#### See also:

`exampy.square()`

Square of a number

`exampy.Pow()`

a number raised to an arbitrary power

#### Notes

Implements the standard cube function

$$f(x) = x^3$$

History:

2020-03-04: First implementation - Bovy (UoT)

#### References

- 1 A. Mathematician, “x to the p-th power: squares, cubes, and their general form,” J. Basic Math., vol. 2, pp. 2-3, 1864.

### exampy.integrate

`exampy.integrate.riemann(func, a, b, n=10)`

A simple Riemann-sum approximation to the integral of a function

**Parameters:**

- **func** (*callable*) – Function to integrate, should be a function of one parameter
- **a** (*float*) – Lower limit of the integration range
- **b** (*float*) – Upper limit of the integration range
- **n** (*int, optional*) – Number of intervals to split `[a,b]` into for the Riemann sum

**Returns:** Integral of `func(x)` over `[a,b]`

**Return type:** float

©2020, Jo Bovy. | Powered by [Sphinx 2.2.0](#) & [Alabaster 0.7.12](#) | [Page source](#)

As you can see, the docstrings have been correctly grabbed from the package itself and the `napoleon` extension creates a nicely-formatted listing of each. Also note how the functions in the “See Also” section are live links.

The `autoclass` directive works similar to the `autofunction` one. Without any further options, it simply displays the class’ docstring (that is, the docstring immediately following the `class a_class(object):` statement). You can document class members by listing them in the `:members:` option. For example, to document both a class and its initialization method, do

```
.. autoclass:: a_class
   :members: __init__
```

If you give the `:members:` option without specifying any members directly, *all* member methods will be included in the generated documentation. My recommendation is to always explicitly list the members that you want to document, such that you do not include members without knowing about it (similar to how one should never `from package import *`).

You can also document individual member methods of a class outside of an `autoclass` directive, essentially doing the same as with `autofunction`. For this, use `automethod`, which is the same as `autofunction`, except that you list the name of the as `classname.methodname`.

To illustrate autodoc's handling of classes, we add documentation for the `exampy.Pow` class to the reference page, by updating it to

```
API reference
=====

``exampy``
-----

.. autofunction:: exampy.square

.. autofunction:: exampy.cube

.. autoclass:: exampy.Pow
   :members: __init__

.. automethod:: exampy.Pow.__call__

``exampy.integrate``
-----

.. autofunction:: exampy.integrate.riemann
```

which then creates

## exampy

### Navigation

Contents:

[Installation instructions](#)

[Introduction](#)

[API reference](#)

▪ [exampy](#)

▪ [exampy.integrate](#)

### Quick search

## API reference

### exampy

`exampy.square(x)`

The square of a number

**Parameters:** `x (float)` – Number to square

**Returns:** Square of `x`

**Return type:** float

`exampy.cube(x)`

The cube of a number

Calculates and returns the cube of any floating-point number; note that, as currently written, the function also works for arrays of floats, ints, arrays of ints, and more generally, any number or array of numbers.

**Parameters:** `x (float)` – Number to cube

**Returns:** Cube of `x`

**Return type:** float

**Raises:** No exceptions are raised. –

#### See also:

`exampy.square()`

Square of a number

`exampy.Pow()`

a number raised to an arbitrary power

#### Notes

Implements the standard cube function

$$f(x) = x^3$$

#### History:

2020-03-04: First implementation - Bovy (UofT)

#### References

- 1 A. Mathematician, "x to the p-th power: squares, cubes, and their general form," J. Basic Math., vol. 2, pp. 2-3, 1864.

`class exampy.Pow(p=2.0)`

A class to compute the power of a number

`__init__(p=2.0)`

Initialize a PowClass instance

**Parameters:** `p (float, optional)` – Power to raise `x` to

`Pow.__call__(x)`

Evaluate `x^p`

**Parameters:** `x (float)` – Number to raise to the power `p`

**Returns:** `x^p`

**Return type:** float

### exampy.integrate

`exampy.integrate.riemann(func, a, b, n=10)`

A simple Riemann-sum approximation to the integral of a function

**Parameters:** • `func (callable)` – Function to integrate, should be a function of one parameter

• `a (float)` – Lower limit of the integration range

• `b (float)` – Upper limit of the integration range

• `n (int, optional)` – Number of intervals to split `[a,b]` into for the Riemann sum

**Returns:** Integral of `func(x)` over `[a,b]`

**Return type:** float

©2020, Jo Bovy. | Powered by [Sphinx 2.2.0](#) & [Alabaster 0.7.12](#) | [Page source](#)

It is a matter of taste how exactly one lays out the API. Some packages, such as `numpy` and my own `galpy` package, use an individual page for each function, class, or method. That has the disadvantage of leading to *a lot of files* for a large package. Other options are to use a single page for everything (gets unwieldy) or a single page per submodule or class.

To encourage users of your code to look at the source code (e.g., when they run into issues), you can use the `sphinx.ext.viewcode` [sphinx extension](#). When you add this built-in extension to the extensions list in your `conf.py` file, a link will be added to each documentation function, class, and method that leads to the source code for that function, class, and method. For example, for the API page that we created, adding `sphinx.ext.viewcode` adds “[source]” links:

**exampy**

Navigation

Contents:

- [Installation instructions](#)
- [Introduction](#)
- [Introduction notebook](#)
- [API reference](#)
- [exampy](#)
- [exampy.integrate](#)

Quick search

## API reference

**exampy**

`exampy.square(x)` [\[source\]](#)

The square of a number

**Parameters:** `x (float)` – Number to square

**Returns:** Square of `x`

**Return type:** float

`exampy.cube(x)` [\[source\]](#)

The cube of a number

Calculates and returns the cube of any floating-point number; note that, as currently written, the function also works for arrays of floats, ints, arrays of ints, and more generally, any number or array of numbers.

which when you click on them lead to, e.g.,

**exampy**

Navigation

Contents:

- [Installation instructions](#)
- [Introduction](#)
- [Introduction notebook](#)
- [API reference](#)
- [exampy](#)
- [exampy.integrate](#)

Quick search

## Source code for `exampy._math`

```
def square(x):
    """The square of a number

    Parameters
    -----
    x: float
        Number to square

    Returns
    -----
    float
        Square of x
    """
    return x**2.
```

[\[docs\]](#)

## 4.6 Including jupyter notebooks as part of your documentation

Up until now, the manual consists solely of a set of `.rst` files written in reStructuredText. However, it is also possible to generate documentation pages from `jupyter` notebooks. This has many advantages both from the standpoint of ease of writing the documentation and from making sure that the documentation is as accurate as possible. Writing (parts of) your documentation as a `jupyter` notebook allows you to run the code examples that you include directly, so you are sure that they work without having to copy and paste code. You can also more easily include images, because `jupyter` notebooks can easily generate inline figures that are part of the notebook. Typesetting LaTeX math is also easier using `jupyter` notebooks, because you can directly see the result without having to compile the documentation using `make`. Using `jupyter` notebooks for your documentation’s pages

is so convenient that even if you don't include many code examples, they can still be a good choice. Indeed, these notes themselves are written as a set of `jupyter` notebooks and you can see how they allow complex documentation to be written.

The easiest way to include `jupyter` notebooks in your documentation is by using the `nbsphinx` sphinx extension. After installing this extension with

```
python3 -m pip install nbsphinx
```

(using the additional `--user` option for a user-specific install), you can start including notebooks by simply adding “`nbsphinx`” to the extensions list in your `conf.py` file and then including `.ipynb` files in your documentation's toctree(s) just as you would add `.rst` files. For example, the main toctree for these notes looks as follows:

```
.. toctree::
    :maxdepth: 2
    :caption: Contents:
    :numbered:

    01-Introduction.ipynb
    02-Package-Structure.ipynb
    03-git-and-GitHub.ipynb
    04-Documentation.ipynb
    05-Tests.ipynb
    06-Continuous-Integration.ipynb
    07-Package-Release.ipynb
```

The `nbsphinx` extension will automatically execute notebooks before rendering the documentation *if no output cells are stored*. Thus, if you clear the output of the notebook before saving it, `nbsphinx` will execute it and then render the documentation. This is a great feature to make sure that your documentation's code examples exactly reflect what the current version of the code does (which with regular `reStructuredText` examples can be hard to keep up-to-date). However, if you store any output, then automatic execution is turned off by default.

You can render almost anything in a standard `jupyter` notebook to sphinx-generated documentation: code cells and their output, Markdown cells, and raw cells in different formats. Note that when you are combining both notebooks and regular `reStructuredText` files, the fact that the standard text box in a notebook uses Markdown and not `reStructuredText` can get a little confusing (as it is in writing these notes where I have to use both Markdown and `reStructuredText` in the same notebook to properly display all content!). But generally, any code, LaTeX, and images in the notebook will be seamlessly rendered as HTML, LaTeX, etc. pages.

The `nbsphinx` extension has many [configuration values](#), which you typically can just leave at their defaults. These can be used to control how the notebooks are executed when they are automatically executed, and how to style notebook elements. One element that you may want to change is each code cell's prompt, which shows up by default. To remove the prompt, set



```
nbsphinx_prompt_width = 0 # no prompts in nbsphinx
```

in the `conf.py` file. There is no easy way currently to remove the prompt from the generated LaTeX/PDF documentation.

When you are dealing with notebooks, you will also want to set

```
exclude_patterns = ['.ipynb_checkpoints/*']
```

in your `conf.py` file to tell sphinx to ignore the automatically-generated checkpoint files; without this, sphinx will process these files as well, which can lead to long build times, because these files often change. When you are dealing with notebooks in a git repository (which your documentation should be in), you will also want to install a git plugin to show nicely-formatted diffs of notebooks, because otherwise `git diff` will show you changes in the JSON file that is the underlying representation of each notebook, which isn't a particularly illuminating way to look at changes. Therefore, install `nbdime` for this purpose.

As an example, we write the `intro.rst` page that we created above as a jupyter notebook `intro_notebook.ipynb` that looks like

```

1 Introduction notebook

This page gives a similar introduction as intro.rst, but written as a 'jupyter' notebook.

exampy is an example Python package that contains some very basic math functions. As an example, we can compute the square of a number as

In [1]: import exampy
        exampy.square(3.)
Out[1]: 9.0

Similarly, we can compute the cube of a number:

In [2]: exampy.cube(3.)
Out[2]: 27.0

A general method for raising a number to a given power is given by the Pow class. For example, to get the fourth power of 3, do:

In [3]: po= exampy.Pow(p=4.)
        po(3.)
Out[3]: 81.0

exampy also includes a simple method for integrating a function, in the 'exampy.integrate' submodule. This submodule contains the function 'riemann'
that approximates the integral of any one-parameter function as a Riemann sum. 'riemann' takes as input (i) the function to integrate, (ii) the integration
range's lower limit and (iii) the upper limit, and (iv) optionally, the number of intervals to divide the integration range in. For example, the integrate the square
function of the range [0,1], do:

In [4]: from exampy import integrate
        integrate.riemann(exampy.square,0,1)
Out[4]: 0.35185185185185186

If we increase the number of intervals from the default (which is 10), we get a better approximation to the correct result (which is 1/3):

In [5]: integrate.riemann(exampy.square,0,1,n=1000)
Out[5]: 0.33350016683350014

```

Adding this `intro_notebook.ipynb` file in the toctree in `source/index.rst` as

```

.. toctree::
   :maxdepth: 2
   :caption: Contents:

   installation.rst
   intro.rst
   intro_notebook.ipynb

```

(continues on next page)

(continued from previous page)

reference.rst

adding the `nbsphinx` extension in `source/conf.py`, and running `make html` then gives a documentation page that looks like

**exampy**

Navigation

Contents:

- Installation instructions
- Introduction
- Introduction notebook
- API reference

Quick search

Go

## Introduction notebook

This page gives a similar introduction as [intro.rst](#), but written as a `jupyter` notebook.

`exampy` is an example Python package that contains some very basic math functions. As an example, we can compute the square of a number as

```
[1]: import exampy
     exampy.square(3.)
```

```
[1]: 9.0
```

Similarly, we can compute the cube of a number:

```
[2]: exampy.cube(3.)
```

```
[2]: 27.0
```

A general method for raising a number to a given power is given by the `Pow` class. For example, to get the fourth power of 3, do:

```
[3]: po= exampy.Pow(p=4.)
     po(3.)
```

```
[3]: 81.0
```

`exampy` also includes a simple method for integrating a function, in the `exampy.integrate` submodule. This submodule contains the function `riemann` that approximates the integral of any one-parameter function as a Riemann sum. `riemann` takes as input (i) the function to integrate, (ii) the integration range's lower limit and (iii) the upper limit, and (iv) optionally, the number of intervals to divide the integration range in. For example, to integrate the square function of the range `[0,1]`, do:

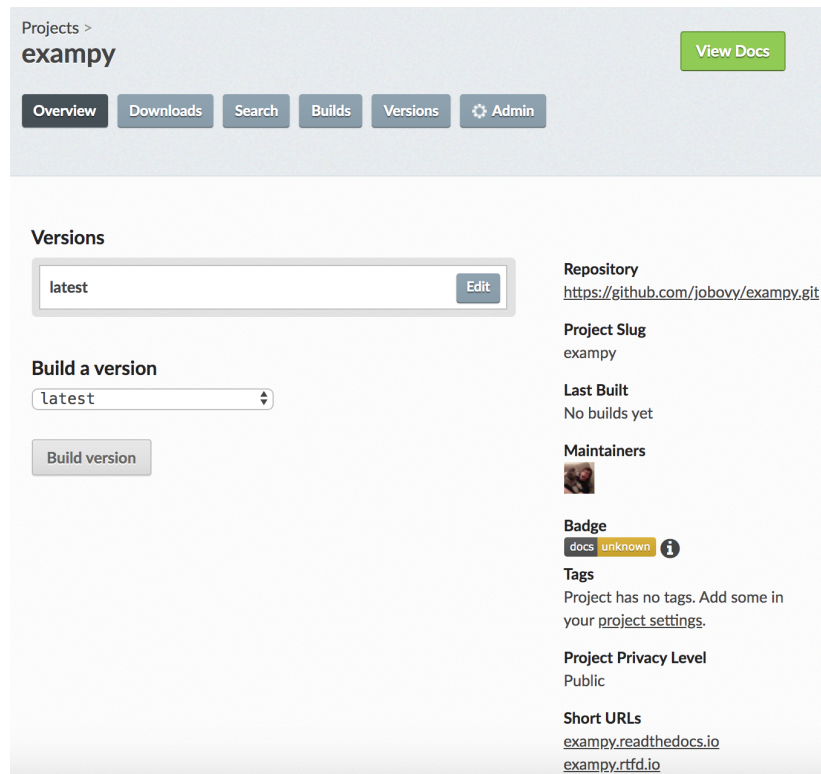
We see that the documentation page looks very similar to the one we created from `intro.rst`.

## 4.7 Automatically building and hosting your documentation on readthedocs.io

To host the documentation you generate using `sphinx` online, you could upload it to a dedicated website (e.g., a [GitHub Pages](#) site), but `readthedocs.io` is a free online service that has become the go-to destination for hosting code documentation online. `readthedocs.io` seamlessly integrates with GitHub and `sphinx` and automatically generates multiple, easily-accessible versions of your code's documentation for different releases and for the development version, the latter of which is updated upon every push of your code to GitHub. If you have working `sphinx`-based documentation for your package, it is easy to get started with `readthedocs.io` and have your documentation online quickly.

To get started, head to <https://readthedocs.io>, click on “Log in”, and sign in with your GitHub account (you could make an account as well, but signing up with your GitHub account makes syncing your `readthedocs.io` account with your GitHub repositories easier). You are brought to your dashboard where you have the option to import a project. Click on that button to obtain

a list of projects that you could import (these are your GitHub repositories; you may have to hit refresh to get the list) and click on the one you want to start building online documentation for. Once you confirm, you are brought to the `readthedocs.io` owner page for your project, which looks as follows at the start:



You can click on “Build a version” to get a first build of the documentation going; this build will likely fail, because it hasn’t been configured yet, but it will set up a webhook to automatically update the documentation when you push changes to GitHub, which is convenient.

To configure a project on `readthedocs.io`, add a `.readthedocs.yml` configuration file, which is documented [here](#). A simple one to get started looks like

```
version: 2

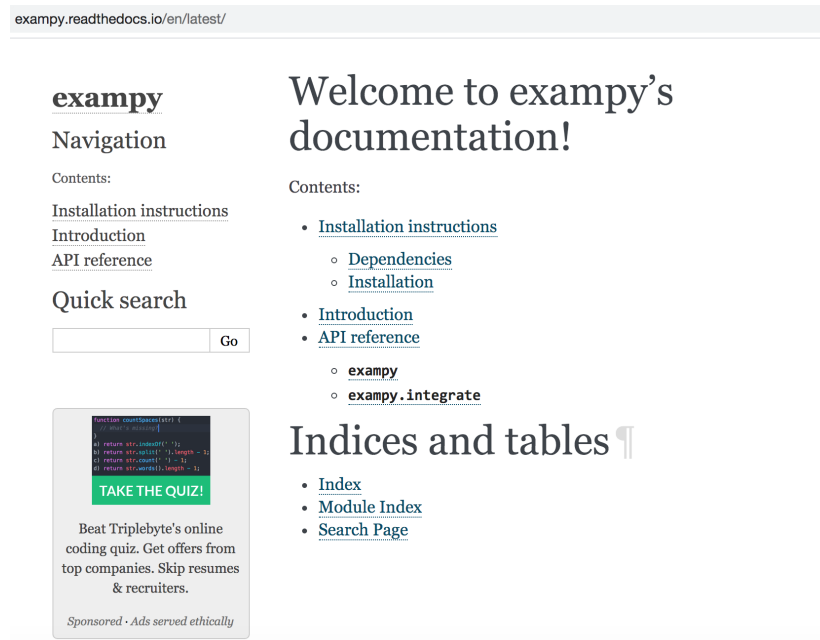
sphinx:
  configuration: docs/source/conf.py

python:
  version: 3
  install:
    - method: pip
      path: .
```

Here, `version:` sets the version of the `readthedocs.io` configuration file format to use (version 2 is the latest, which I will use here, but version 1 is still the default); `sphinx:` tells

`readthedocs.io` to use `sphinx` to build the documentation and states where the `conf.py` file is located (`readthedocs.io` can also find it automatically, but it's always best to be explicit), and the `python:` section configures how to install the package and its dependencies (more on that below).

Because `readthedocs.io` may be using an older version of `sphinx` that does not automatically find your documentation's master file if you named it `source/index.rst` (older versions of `sphinx` assumed it was `source/contents.rst`), you also have to set `master_doc = 'index'` in the `source/conf.py` file to explicitly tell `sphinx` which document contains the main `toctree`. Adding this `.readthedocs.yml` file and this change to `source/conf.py` and pushing these changes to GitHub, you will see that the next build of your documentation commences automatically on `readthedocs.io`. When your documentation builds successfully, you can click on “View Docs” to see the online documentation (at <https://exampy.readthedocs.io/en/latest/> for the example package). This documentation looks like (this is the version *before* we added the `source/intro_notebook.ipynb` jupyter notebook; see below)



As you can see, it looks almost identical to the documentation that we rendered locally above, but now it is available online.

Including a `jupyter` notebook in the online documentation requires us to install `nbsphinx` on the `readthedocs.io` build server. What gets installed as part of the build process is controlled by the `python:` section of the configuration file. Above, we simply asked for the `exampy` package to be installed with `pip install .`, because I specified `method: pip` and `path: .` in the `install:` section. This is a standard Python install and it therefore also installed `exampy`'s requirements listed in the `install_requires` section of its `setup.py` file, but this does not contain the `nbsphinx` requirement. We can add additional requirements using a `requirements.txt` file: such a file could live at the top level of the `git` repository, in which case it would be an alternative way of specifying the package's dependencies (this is standard `pip` usage), but we can also include a `requirements.txt` file in the `docs` directory that is specific to building the documentation. We

will do that for nbxphinx, placing just that one line in `docs/requirements.txt` and updating the `readthedocs.yml` to

```
version: 2

sphinx:
  configuration: docs/source/conf.py

python:
  version: 3
  install:
    - method: pip
      path: .
    - requirements: docs/requirements.txt
```

Pushing these changes to GitHub, the documentation gets built automatically, and it now includes the jupyter notebook that we included.

Before you release a version, `readthedocs.io` will show only the latest version of your documentation, but when you create a release or a `git tag`, your `readthedocs.io` page will include those [versions](#), which you should give names like “1.2.1” etc. On your project’s admin page, you have the option to edit the versions that are shown (for example, to remove old released versions, although unless you have released in error, it’s best to keep old versions around for reference). You can also build the documentation for other branches, which you should use sparingly, but it is useful when you are developing a new feature with extensive documentation and you want to see what the documentation looks like before merging this branch into your main branch.

Typically, the automatic builds on `readthedocs.io` work fine and once set up, you will have to interact very little with the `readthedocs.yml` service, but you do have the option to trigger a build yourself on your project’s admin page. That can be useful when a build failed because of some issue installing a dependency (e.g., it briefly broke), and you want to re-build the documentation without pushing new changes to GitHub. There is [extensive documentation online](#) (obviously ...) that allows you to easily make even complex changes to the way `readthedocs.io` serves your documentation (like serving it from a custom domain).



## TESTING YOUR CODE

Thoroughly testing your code is crucial for allowing others (and yourself!) to use it confidently, for maintaining your code long-term, and for developing new features. Testing your code needs to be done *continuously*, that is, it is not long-term useful to perform a test of the implementation of a new feature or of its integration with the rest of your package if you do not save the test and re-run it every time the code changes; unfortunately, only testing code at its first implementation still remains the main mode in which scientific software is tested. Instead you need to write a comprehensive **test suite** that checks that your code keeps working the way you intend it to work as your package grows and evolves.

Having a comprehensive test suite is essential for being as confident as you can be that your code behaves correctly (of course, no test suite is 100% water tight). But having a comprehensive test suite also helps you maintain and develop your package. A test suite makes maintenance easier, because you can catch new problems early (e.g., an issue stemming from a change in a dependency), when they are typically easier to solve. It is also easy to check that your package works with a new minor (or major, but that will likely be a while) version of Python itself. It also helps you in developing your code, especially in allowing large-scale changes to the underlying framework of your package without changing the package's user interface (or without changing it much). For example, if you want to completely re-vamp the way a complex sub-module of your code with many functions and classes is implemented, this is difficult to do without a comprehensive test suite, because it would be difficult to be sure that no user-facing functionality changed. But with a good test suite, you can be confident that your underlying changes do not break code that people have that uses your package or the way they interact with your code. Having a comprehensive test suite is also absolutely essential if you will be accepting pull requests from outside users (which you should!): without a test suite, it would be very difficult to know that changes proposed in a pull request (especially if there are many) do not break some part of your code (you would be surprised at how easily even a small change in one part of your package can cause problems in unexpected places).

In this chapter, I discuss the basics of testing your Python package, explain how to write good tests for different aspects of your package, and how to build and run a test suite with `pytest`, the current standard for Python testing. In this chapter, we will run this test suite by hand, but in the next chapter, I will discuss how to use online continuous integration services to automatically run your test suite on a set of external servers every time you change your code.

## 5.1 Basics of good testing

Tests of a Python package are regular Python statements, functions, etc., just like any other Python code. They become tests of your software package simply by virtue of *context*: tests are regular Python files that live in a designated spot for tests of your package and that consist of a set of statements checking that your code works as intended. At some level, any use of your package by yourself and other users is a test of your code, because users will (or should) be sanity-checking your code's results and report issues if the results are incorrect or unexpected. But your package should contain a dedicated set of tests, which we refer to as a *test suite*.

The first decision you have to make is where to put your tests. Tests should be part of the same `git` repository that contains the package itself and then there are essentially two options: (i) you can put the tests *outside of the package itself*, by putting them in a sub-directory of the top-level directory of your package's `git` repository, or (ii) you can incorporate the tests *into the package itself*, that is, include them as a sub-directory within your Python package's directory (or in a set of sub-directories, e.g., to separate tests by submodule). The latter option allows you to include the tests in distributions of your code, which may be useful to allow people who have installed your code to run the tests themselves and verify that they pass on their system. However, I believe it is better to choose option (i) and keep the tests outside of the package itself. This is for a few reasons: (a) packages should be as light-weight as possible and a comprehensive test suite will be large (e.g., for `galpy`, the test suite currently contains about 30k lines, while the package itself has about 40k, excluding comments and docstrings), (b) your package should be as well documented and tested as possible, but you will likely not document the tests much and almost certainly not write a test suite for the tests (and so on!), (c) your tests may have difficult-to-install dependencies that you do not want users of your code to have to install, and (d) the main issues that users will run into are installation issues; when they can install and load your code, it's highly unlikely that a test of the code's functionality would fail, so there really isn't too much point in normal users of your code running the test suite themselves if you run it often yourself. Thus, you should include tests in a `tests/` sub-directory of your top-level directory, which therefore now looks like (for the `exampy` example package from [chapter 2](#) (page 8)):

```
TOP-LEVEL_DIRECTORY/  
  docs/  
  exampy/  
  tests/  
  README.md  
  setup.py
```

The `tests/` directory then contains a set of Python `.py` files that contain tests of your code. For a large software package, one option would be to mirror your package's structure of sub-modules and arrange tests in a similar directory tree under `tests/`, but typically it's easy enough to simply include a bunch of `*.py` files in the `tests/` directory itself that test all of your package's functionality in all of its submodules. To seamlessly integrate with the `pytest` test framework, your tests should be arranged as follows:



- Tests should be in files with filenames that start with `test_` and end in `.py`, so for example, `test_basic_math.py` will contain tests of the basic math included in `exampy`'s top-level module (see below).
- Functions that contain tests to be run within the test files should start with `test`; if you use classes to arrange tests, these need to start with `Test`. Your test files can include other functions or classes, for example as helper functions/classes for your tests, but these will not be executed as tests by `pytest` below.

I will focus on the case of using functions to write tests, rather than classes. Using classes for tests is useful when your tests require common, expensive initialization procedures, which can easily be done with a `TestX` test class, but is difficult when using functions. When this is not necessary, it is easier to simply use functions that contain all of the setup for your test, such that they can stand alone and can be run independently from each other. This is useful when diagnosing problems with your tests or code, so you don't always have to run the entire (or a large part of the) test suite.

It is a matter of taste how you arrange tests of different parts of your package into a set of test files. One option is to have a single file per package file that contains the tests of that package file's functionality, but that leaves it unclear where to put tests of the integration of different parts of your package. Another option is to use one file (or a few files, if a logical division can be made) for each submodule and then include tests of the interplay between different submodules in the most relevant submodule's test file(s). However you choose to arrange your tests, use a logical structure that makes it easy to find where tests are located for your future self and for other developers.

Before going on to describe how to write Python tests, it is useful to consider what are *good* properties of tests of a scientific software package and how to construct a good test suite:

- *Many of your tests should be as minimal, short, and atomic as possible:* you should strive to make sure that each function in your package does what you intend it to do and that it does this correctly; this is called [unit testing](#). If your package is well written, it should consist mostly of relatively small units (functions, classes, etc.) that work together to do more complex things. Unit testing makes sure that the smallest units of code in your package work as intended, because having all of the parts of a complex code work is a prerequisite to having the entire complex code working correctly. Testing these small units of your package should be possible with relatively brief tests: they should require as little setup as possible.
- *Your test suite should include integration tests:* Besides making sure that the atomic parts of your code work correctly, you should test that they work together to create more complex workflows that work as expected. Even if all of the individual parts of your code work to your satisfaction, it can still easily be the case that their combination does not work as you intended (this can be as simple as their outputs and inputs not being compatible, or more complicated when, e.g., approximations that seem correct for parts of your code are not good enough for their combination with other parts of your code). These integration tests will typically be more complex pieces of Python code that more directly resemble actual use of your code, but that's okay.
- *Your tests should run in as little time as possible:* Having to wait for a long test-suite to run before knowing whether or not your code contains bugs or other errors will strongly impede

your progress (I should know, `galpy`'s test suite currently runs for close to an hour, even when spreading the tests over six machines!). Thus, you want your tests to run *fast*. Because a test suite for any decent-sized package will contain many individual test functions, each of the individual test functions should run very fast in order for the entire test suite to run quickly.

Of course, for complex scientific software, it is inevitable that some tests will require a longer time to run (long in the context of tests starts at something like one second, and you should definitely avoid any test that requires more than a minute to run). You might want to check that your code works to a certain high level of precision and that level of precision requires a long computation. In such cases, it is useful to also include a shorter version of the same test if this is possible (for example, one that requires less time to get a lower precision result) that can be run before the longer version of the test, such that not-too-subtle bugs can be found using the shorter test without having to wait for the longer test to finish. Of course, this only makes sense if the shorter test is significantly shorter, such that it does not increase the total test runtime much.

- *You should test outputs of your code as well as the important errors and warnings that it raises:* You want to focus your efforts on making sure that the values your code returns are correct, but if your code raises errors in certain cases or warnings, it is important to test whether these are raised appropriately as well.
- *You should test setting non-default values for your functions' keyword parameters:* For all functions with optional parameters, you should test that changing the value of any optional parameters is properly handled by the code (e.g., to make sure you haven't accidentally hard-coded the value of an optional parameter somewhere). If you have more than one optional parameter, it quickly becomes difficult to test all possible combinations of setting or not setting keywords, but there is little harm in combining these and simply using a test that changes the value of *all* keywords from their defaults.
- *You want to have at least one test for each function, but it's typically best to test many different invocations:* Even for simple functions, try multiple values of the input arguments and keywords to more extensively test the function. If your function has conditional statements, this is likely necessary to obtain a high coverage, as we will discuss more below (because a single invocation only tests a single path through your code's conditional statements, thus not testing the other paths).
- *Add a test for each reported issue whose fix requires a code change:* When users report an issue with the code that requires you to change the code, add a test that checks that the issue was fixed. This will prevent the issue from arising again in the future, e.g., when you accidentally undo the fix later. Often the best way to start editing your code in response to a reported issue is to *first* write a test that fails because of the issue and then make the fix in the code; once the test passes you are ready to close the issue.
- *You should document your test suite:* Many tests of a scientific software package will be non-trivial (e.g., checking the code against the result of an analytical calculation). To make sure you remember what you were thinking when you wrote specific test, extensively document your reasoning with code comments. Users will not (typically) be importing and using your

test suite, so there is no need for true docstrings, but keeping a basic level of documentation of all functions in your test suite will make it easier for you and other developers to understand, use, and extend it.

Aside from these general desirable properties of a good test suite, the structure of good tests for a scientific software package will depend strongly on the package itself. While including very simple test cases of general functions is useful for catching major issues (e.g., testing the `exampy.integrate.riemann` function with a constant function), be careful in only depending on the simplest possible tests, because it is likely that these would not catch more subtle issues with your code because they are *too* simple (i.e., even a slightly wrong version of your code might get them right enough). In my own field of astrophysics, we often have analytical solutions for certain inputs of computations that our code can perform in general and it is useful to test that such analytical solutions are correctly reproduced (and it is good to test against non-trivial analytical solutions if they exist). If a computational problem has no known analytical solution, we often still know of certain properties that the solution should have or we have constraints on the solution and we can test that these properties and constraints are satisfied (e.g., when solving the Newtonian equation of motion for a conservative, time-independent force, we know that the energy associated with the solution is conserved). Another way to test your code may be to compare it to an alternative solution, for example, one that only applies in certain cases (but isn't analytic) and/or one that is part of a different package that should be consistent. As with documentation before, it is difficult to write *too many tests* and as long as they run in a reasonable amount of time, erring on the side of testing your code too much is better than not testing it enough!

## 5.2 Writing simple tests

To illustrate how to write basic tests, we will add some tests of the basic `exampy` functionality that we implemented in the previous chapters. A more advanced discussion of test writing is inevitably tied up with the framework that we choose to use to run the tests and, thus, I postpone a more advanced discussion until the next section, where running tests with `pytest` is described in detail.

I start by adding a file `test_basic_math.py` in the `tests/` directory that will contain tests of the basic math contained at the top level of the `exampy` package. Thus, the top-level package directory, expanded to one level looks now as follows

```
TOP-LEVEL_DIRECTORY/  
  docs/  
    build/  
    source/  
  Makefile  
  make.bat  
  exampy/  
    integrate/  
    __init__.py
```

(continues on next page)

(continued from previous page)

```
_math.py
tests/
    test_basic_math.py
README.md
setup.py
```

As discussed above, I keep the tests outside of the package itself here.

The way we check in a test whether the code conforms to our expectations of how it should work is using one or more `assert` statements that assert that a certain behavior holds. For example, the simplest function in `exampy` is `exampy.square` and we check that this function returns the square by checking a few known solutions. This is done in the following function that we add to `tests/test_basic_math.py`

```
def test_square_direct():
    # Direct test that the square works based on known solutions
    import math
    import exampy
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(3.)-10.) < tol, \
        "exampy.square does not agree with known solution"
    return None
```

As you can see, I add a total of three `assert` statements that check the behavior of the square function against the known square of the numbers one, two, and three (which I worked out analytically for you ...). Python `assert` statements have a very simple behavior: in the case of `assert True, msg`, nothing happens and the statement following this is executed; in case of `assert False, msg`, an `AssertionError` is raised and the `msg` string is printed, and `msg` should therefore contain a useful message explaining what went wrong. Therefore, I have written the test of the square function as an `assert` that the result from the square function agrees with the known value to a known absolute tolerance (using the `math.fabs` function). I have chosen to include the two `import` statements to make this test a fully self-contained code example, but when this test file grows, you may want to move these to the top level of the file, such that they do not need to be repeated in each test (however, repeating them makes all of the tests self-contained code snippets, which may be useful when developing and testing the tests). Note that you can also use the `numpy.allclose` function to test whether two numbers or arrays agree to some relative and absolute tolerance (e.g., as `numpy.allclose(exampy.square(2.), 4.)`), but I will not discuss that here.

I will discuss how to run the tests using the `pytest` commandline utility below, but for now you could run the test manually by going to the `tests/` directory, opening a Python terminal, and doing

```
>>> import test_basic_math
>>> test_basic_math.test_square_direct()
```

which produces

```
AssertionError: exampy.square does not agree with known solution
```

We get this error, because I actually got the known solution of  $3^2$  wrong! To fix this, we change the test to

```
def test_square_direct():
    # Direct test that the square works based on known solutions
    import math
    import exampy
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(3.)-9.) < tol, \
        "exampy.square does not agree with known solution"
    return None
```

Running the test again, we now get no output, indicating that the test passed.

As an example of testing a known property of the solution, we add a test of `exampy.cube` that checks that this is an odd function, that is, that for example  $(-2)^3 = -(2)^3$

```
def test_cube_oddfunction():
    # Test of the cube function by checking that it is an odd function
    tol= 1e-10
    assert math.fabs(exampy.cube(1.)+exampy.cube(-1.)) < tol, \
        "exampy.cube is not an odd function"
    assert math.fabs(exampy.cube(2.)+exampy.cube(-2.)) < tol, \
        "exampy.cube is not an odd function"
    assert math.fabs(exampy.cube(3.)+exampy.cube(-3.)) < tol, \
        "exampy.cube is not an odd function"
    return None
```

Opening a Python terminal and running

```
>>> import test_basic_math
>>> test_basic_math.test_cube_oddfunction()
```

then returns nothing, indicating that the test passed. Of course, we could more simply write this function as

```
def test_cube_oddfunction():
    # Test of the cube function by checking that it is an odd function
    tol= 1e-10
    for nn in range(1,10):
        assert math.fabs(exampy.cube(nn)+exampy.cube(-nn)) < tol, \
            "exampy.cube is not an odd function"
    return None
```

and testing all the way up to  $N = 9$ .

In cases where no relevant analytical solution is known or where no strong constraints exist on the solution that provide enough piece of mind that satisfying them makes you confident that your code works, you can also test against more approximate solutions or properties or against a different way of solving the problem that may be available in another package. As an example of this type of testing, I first implement a better approximate integration method in `exampy.integrate`, adding the [Simpson's rule](#) as the function `exampy.integrate.simps`, which (with documentation!) looks like

```
def simps(func,a,b,n=10):
    """Integrate a function using Simpson's rule

Parameters
-----
func: callable
    Function to integrate, should be a function of one parameter
a: float
    Lower limit of the integration range
b: float
    Upper limit of the integration range
n: int, optional
    Number of major intervals to split [a,b] into for the Simpson rule
```

```
Returns
-----
float
    Integral of func(x) over [a,b]
```

```
Notes
-----
Applies Simpson's rule as
```

```
.. math::
    \int_a^b \mathrm{d}x f(x) \approx \frac{(b-a)}{6n} \left[ f(a) + 4f(a+h) + \right.
```

(continues on next page)

(continued from previous page)

```
/2)+2f(a+h)+4f(a+3h/2)+
```

```
\ldots+2f(b-h)+4f(b-h/2)+f(b)\\right]
```

See Also

```
-----
exampy.integrate.riemann: Integrate a function with a simple Riemann sum
"""
    return (2.*np.sum(func(np.linspace(a,b,n+1)))
            -func(a)-func(b) # adjust double-counted first and last
            +4.*np.sum(func(np.linspace(a+(b-a)/n/2,b-(b-a)/n/2,n))))\
            *(b-a)/n/6.
```

(note that I am not particularly concerned with implementing this in an efficient manner here). Suppose we didn't know *any* analytical integrals, then we could still check that `exampy.integrate.simps` at least gives approximately the same answer as `exampy.integrate.riemann`, by including a test in a new `tests/test_integrate.py` file like this

```
import numpy as np
import exampy.integrate

def test_simps_against_riemann():
    # Test that simps and riemann give approximately the same answer
    # for complicated functions
    complicated_func= lambda x: x*np.cos(x**2)/(1+np.exp(-x))
    tol= 1e-4
    n_int= 1000
    assert np.fabs(exampy.integrate.simps(complicated_func,0,1,n=n_int)
                  -exampy.integrate.riemann(complicated_func,0,1,n=n_
→int)) \
                  < tol, \
                  """exampy.integrate.simps gives a different result.
→from """\
                  """exampy.integrate.riemann for a complicated function"
→"""
    return None
```

which tests that the function  $f(x) = x \cos x^2 / (1 + e^{-x})$  is consistently integrated over the interval from zero to one. The point of a test like this is to gain confidence in the validity of the implementation of the more complex method by making sure that it approximately agrees with a simpler method; the thinking being that if they agree, the more complex one is probably implemented correctly, because otherwise it would be chance that they agree well. Of course, this assumes that you have made a decent effort to implement the more complex method correctly, simply copying in the simpler method as the more complex method would obviously also pass this test!

We could also test that our integration routines are consistent with those in an external package, `scipy.integrate`, by adding the following test to `tests/test_integrate.py`

```
def test_simps_against_scipy():
    # Test that exampy.integrate.simps integration agrees with
    # scipy.integrate.quad
    from scipy import integrate as sc_integrate
    complicated_func= lambda x: x*np.cos(x**2)/(1+np.exp(-x))
    tol= 1e-14
    n_int= 1000
    assert np.fabs(exampy.integrate.simps(complicated_func,0,1,n=n_int)
                  -sc_integrate.quad(complicated_func,0,1)[0])\
        < tol, \
        """exampy.integrate.simps gives a different result_
→from """\
        """scipy.integrate.quad for a complicated function"""
    return None
```

If you run this test, you will see that it passes, which given the 1e-14 tolerance demonstrates that our `exampy.integrate.simps` method works very well! It is left as an exercise to write a test that checks whether the error made by Simpson's rule scales as the fifth power of the number of intervals times the fourth derivative, as expected from the [math behind Simpson's rule](#).

## 5.3 Running a test suite with pytest

So far, we have been running the tests that we wrote above by going into the `tests/` directory, opening a Python terminal, and importing and running the tests manually. Test suite runners provide an easier way to run your tests, while also giving useful outputs of the status of your test runs and providing much additional functionality to make test writing and running easier and more comprehensive. While there are different test runners available in the Python ecosystem (notably `nose`), currently the dominant framework is `pytest` and we will focus on `pytest` in these notes. You can install `pytest` as

```
pip install -U pytest
```

To get to know how `pytest` works, we run the tests that we have written so far in `tests/test_basic_math.py` using `pytest` as

```
pytest -v tests/test_basic_math.py
```

in a regular terminal in the top-level of the package. This produces output that looks like



```

===== test session starts
→=====
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 --
→/PATH/
TO/PYTHON/BINARY
cachedir: .pytest_cache
rootdir: /PATH/TO/exampy
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.4.0, remotedata-0.
→3.1
collected 2 items

tests/test_basic_math.py::test_square_direct PASSED
→[ 50%]
tests/test_basic_math.py::test_cube_oddfunction PASSED
→[100%]

===== 2 passed in 0.07s
→=====

```

This output shows the Python and pytest versions that you are using, the directory that you are executing the command from, any plugins (ignore these for now), and then displays a verbose summary of the test run (obtained using the `-v` flag that we passed to the `pytest` invocation). The final summary is that two tests passed and all is well. If we had run the `pytest` command before we fixed the erroneous square of three in the `test_square_direct` function, we would have instead gotten

```

===== test session starts
→=====
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 --
→/PATH/
TO/PYTHON/BINARY
cachedir: .pytest_cache
rootdir: /PATH/TO/exampy
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.4.0, remotedata-0.
→3.1
collected 2 items

tests/test_basic_math.py::test_square_direct FAILED
→[ 50%]
tests/test_basic_math.py::test_cube_oddfunction PASSED
→[100%]

===== FAILURES
→=====

```

(continues on next page)

(continued from previous page)

```

----- test_square_direct -----
→ -----

def test_square_direct():
    # Direct test that the square works based on known solutions
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
>    assert math.fabs(exampy.square(3.)-10.) < tol, \
        "exampy.square does not agree with known solution"
E    AssertionError: exampy.square does not agree with known solution
E    assert 1.0 < 1e-10
E    + where 1.0 = <built-in function fabs>((9.0 - 10.0))
E    + where <built-in function fabs> = math.fabs
E    + and 9.0 = <function square at 0x1080fad90>(3.0)
E    + where <function square at 0x1080fad90> = exampy.square

tests/test_basic_math.py:12: AssertionError
===== 1 failed, 1 passed in 0.10s
→ =====

```

We see that both tests are still run, but that the first one failed. All failures are presented in a detailed FAILURES section that contains a traceback of which assert statement failed and the components of this assert statement are somewhat dissected, which is helpful in diagnosing what went wrong in the test.

Besides the basic math tests, we have also already written tests of the `exampy.integrate` sub-module. To run all tests of the package, we can do either

```
pytest -v tests/test_basic_math.py tests/test_integrate.py
```

that is, we specify both files directly, or

```
pytest -v tests/
```

Both of these give

```

===== test session starts
→ =====
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 --
→ /PATH/
TO/PYTHON/BINARY

```

(continues on next page)

(continued from previous page)

```

cachedir: .pytest_cache
rootdir: /PATH/TO/exampy
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.4.0, remotedata-0.
→3.1
collected 4 items

tests/test_basic_math.py::test_square_direct PASSED
→[ 25%]
tests/test_basic_math.py::test_cube_oddfunction PASSED
→[ 50%]
tests/test_integrate.py::test_simps_against_riemann PASSED
→[ 75%]
tests/test_integrate.py::test_simps_against_scipy PASSED
→[100%]

===== 4 passed in 0.22s
→=====

```

and we see that all four existing tests are run and that they pass. `pytest` has a rather straightforward [set of rules for discovering tests](#) when you specify directories and files and which we summarized [above](#) (page 72).

There are *many* options available when running `pytest`. Useful options of the command-line tool are

- `-x`: Exit upon the first failure. This causes the test run to be interrupted as soon as a test fails. The default is to run all tests and report all passes and all failures.
- `-s`: Print any stdout and stderr output produced by your code. The default behavior is to not print these, but if you have, for example, print statements in your tests (e.g., when debugging the tests) and you want these to show up, you need this option.
- `-k EXPRESSION`: Use this to only run tests with names that match the `EXPRESSION`. For example, running `pytest -v tests/test_basic_math.py -k square` would only run tests with `square` in their name. This is useful if you only want to run a single test or a subset of related tests (e.g., when debugging tests or during the implementation of a new feature).
- `--lf`: Only run tests that failed during the previous invocation of `pytest`. This is helpful when you have a situation where a test unexpectedly fails as part of a bigger test suite and you are trying to fix the code or test to make the test pass again, without having to re-run the entire test suite over and over. Once you've fixed the issue, make sure to run the entire test suite again to make sure that your fix did not accidentally break something else!
- `--disable-pytest-warnings`: When your code emits warnings, `pytest` captures these and prints them as part of a warnings summary at the end of the run. If your code emits *many* warnings, this can clobber the entire output from the test run and setting this option turns off

the warnings summary.

`pytest` has additional functionality to help you write tests for your code. For example, you will want to test that your code correctly raises exceptions in cases where an exception should be raised. `pytest` allows this through the `raises` context manager, which checks that a piece of code raises a certain error. For example, the functions in `exampy.integrate` require that the to-be-integrated function can handle array inputs, which fails when a function is provided that only works for scalar inputs, as in the following snippet

```
>>> import math
>>> import exampy.integrate
>>> print(exampy.integrate.simps(lambda x: math.exp(x),0,1))
```

This raises

```
TypeError: only size-1 arrays can be converted to Python scalars
```

which is not a very useful error message for users of the code. To remedy this, we can edit the source code for the `exampy.integrate.simps` function to catch this error and re-raise it with a more informative error message. The code (without the docstring) becomes

```
def simps(func,a,b,n=10):
    try:
        return (2.*np.sum(func(np.linspace(a,b,n+1)))
                -func(a)-func(b) # adjust double-counted first and last
                +4.*np.sum(func(np.linspace(a+(b-a)/n/2,b-(b-a)/n/2,n))))\
                *(b-a)/n/6.
    except TypeError:
        raise TypeError("Provided func needs to be callable on arrays of_
↪inputs")
```

such that if we run the snippet above, we now get

```
TypeError: Provided func needs to be callable on arrays of inputs
```

(in addition to the full traceback). Now we want to check that our code indeed raises this exception properly when it encounters this situation, so we add a test to `tests/test_integrate.py` that does this with the `raises` context manager; the new test is

```
def test_simps_typerror():
    # Test that exampy.integrate.simps properly raises a TypeError
    # when called with a non-array function
    import math
    import pytest
    with pytest.raises(TypeError):
```

(continues on next page)

(continued from previous page)

```

    out= exampy.integrate.simps(lambda x: math.exp(x),0,1)
    return None

```

This test passes. See for yourself what happens if you use `with pytest.raises(ValueError):`; you should see that the test now fails with an informative (but long!) message about the raised exception. If you want more fine-grained control over the tested exception, for example, to make sure the correct `TypeError` was raised (not just any `TypeError`), you can get access to the full error message with

```
with pytest.raises(TypeError) as excinfo
```

which has the type of the exception raised, the value of the exception message string, and the traceback with the full traceback. You can then for example test that the message of the `TypeError` is exactly the expected one as follows:

```

def test_simps_typerror():
    # Test that exampy.integrate.simps properly raises a TypeError
    # when called with a non-array function
    import math
    import pytest
    with pytest.raises(TypeError) as excinfo:
        out= exampy.integrate.simps(lambda x: math.exp(x),0,1)
        assert str(excinfo.value) == "Provided func needs to be callable on
→arrays of inputs"
    return None

```

Note that the `assert` on the raised error is outside and after the `with pytest.raises(...)` as context manager

The second option is to use the `match=` keyword of `pytest.raises`, which checks whether the error message matches an expected value, expressed as a regular expression. For example, we can do

```

def test_simps_typerror():
    # Test that exampy.integrate.simps properly raises a TypeError
    # when called with a non-array function
    import math
    import pytest
    with pytest.raises(TypeError,match="Provided func needs to be
→callable on arrays of inputs"):
        out= exampy.integrate.simps(lambda x: math.exp(x),0,1)
    return None

```

The `with pytest.raises(...)` as `excinfo:` syntax gives you more freedom to assert things about the raised error, but for most use cases the `match=` method will do.

Similarly, you can test that your code raises the expected warnings, with the `pytest.warns` context manager, described in detail [here](#). You will have to make sure that warnings are printed and not suppressed to properly test warnings, for example, by doing `warnings.simplefilter("always", WARNING_CLASS)` where `WARNING_CLASS` is a type of warning (e.g., `DeprecationWarning`).

Sometimes your test suite will contain tests that the current version of the code does not pass. This could be for an in-progress fix of the code when you have already added the test that the code is fixed, but the code hasn't been fixed yet (this is good practice). Or you may have a test of old behavior that no longer works, but you want to keep the test for historical reasons and for keeping a record that this behavior changed (obviously, you want to do this very sparingly and only for important changes; most tests that have become out-of-date should be edited and/or removed). Functionality for this is part of `pytest`'s framework for [skipping tests](#). What you can do is to label a test as an expected failure, by marking it with the `@pytest.mark.xfail` decorator. For example, say that we have started work on fixing the fact that `exampy.integrate.simps` fails for functions that only work for scalar inputs and we write a test that should pass once this works; we first write this test as

```
def test_simps_scalarfunc():
    # Test that exampy.integrate.simps works even when called with a
    # non-array function
    import math
    tol= 1e-7
    assert np.fabs(exampy.integrate.simps(lambda x: math.exp(x),0,1)
                  -(math.e-1.)) < tol, \
        """exampy.integrate.simps does not work for scalar-
→input"""\
        """functions"""
    return None
```

If we then run the test suite, the test fails with the `TypeError: Provided func needs to be callable on arrays of inputs` error. However, if we mark it as

```
@pytest.mark.xfail
def test_simps_scalarfunc():
    # Test that exampy.integrate.simps works even when called with a
    # non-array function
    import math
    tol= 1e-7
    assert np.fabs(exampy.integrate.simps(lambda x: math.exp(x),0,1)
                  -(math.e-1.)) < tol, \
        """exampy.integrate.simps does not work for scalar-
→input"""\
        """functions"""
    return None
```

the test now **XFAILs**, that is, it is an expected failure. If the test does happen to pass, the test suite

will still report the tests as being successfully run, which is not typically the behavior that you want, because a test that is expected to fail, but that actually passes, is an unexpected behavior of the code (and what tests truly establish is that your code behaves *as expected*, more so than that it behaves *correctly*). By setting the `strict=True` parameter as

```
@pytest.mark.xfail(strict=True)
...
```

an expected failure that passes will now raise a test error. Other options of `pytest.mark.xfail` allow you to provide a reason for the failure as the `reason=` keyword, the specific exception that should be raised as the `raises=` keyword, and you can even make the test not run at all by setting `run=False` (which is useful if the test’s failure crashes the interpreter and thus the entire test suite).

## 5.4 Test coverage

Now that we are well underway to writing a good test suite, an important question crops up: How comprehensive is our test suite? That is, to what extent does the test suite actually test all of the code included in the package and to what degree does it cover different ways different parts of the package work together? Answering these questions is the domain of **test coverage** and in this section I give a brief introduction to the important concepts to consider when worrying about your code’s test coverage and how to use software tools to check your tests’ code coverage.

There are different coverage criteria depending on how thoroughly you want to define “coverage”:

- *Function coverage* (including *class coverage* and *method coverage*): This type of coverage checks whether every function (or class, or method) of your package is called by the test suite (and thus, if your test suite passes all tests, that there is a successful evaluation of each function/class/method of your package). This is the first type of coverage you should aim to have when starting to build a test suite. But if your functions contain conditional parts, then 100% function coverage can still leave many parts of your code untested.
- *Statement coverage*: Going beyond function coverage, this type of coverage demands that each statement in your code is executed. Thus, 100% statement coverage means that every single statement in your software package is executed at least once by your test suite. As I will discuss further below, this is relatively easy to measure, realistic to attain, and achieving 100% statement coverage is what your test suite should aim for. However, statement coverage may still leave parts of your code untested, especially if your code is written in a way that gets around statement coverage (see example below). To more fully characterize test coverage, there are two more advanced coverage types.
- *Branch coverage*: This type of coverage checks that every branch in your code (e.g., the three possible evaluations of an `if: ... elif: ... else: ...` block) gets executed by your tests. Thus, for every conditional part of your code, your tests should go through each possibility. More generally, branch coverage can mean *testing every possible distinct path through your code’s conditional statements*. If your code contains more than a few conditional

statements, the number of possible paths grows exponentially and in practice it becomes impossible to test each possible path, but it is useful to keep in mind that testing different paths through your code is a good idea. Measuring branch coverage is difficult.

- *Condition coverage*: If your code contains conditional statements with complex boolean expressions (e.g., `if (x > 0 and y < -1):`), then this type of coverage checks whether *each boolean sub-expression* evaluates to both `True` and `False` in your test suite. That is, this goes beyond statement coverage, which in this example could be achieved by the total boolean expression evaluating to `True` (and `False` if there is an `else:` statement as well), without going through all possible combinations of boolean sub-expressions. Condition coverage is also difficult to measure, but is a good standard to aim for when writing tests.

While branch coverage implies statement coverage, and statement coverage implies function coverage, statement coverage does not imply branch coverage, because of edge cases such as

```
if math.fabs(x) < 1e-10: x= 0
```

which as a line (which is the unit most tools for measuring coverage use) registers as executed no matter whether the condition `math.fabs(x) < 1e-10` is `True` or `False`. To avoid getting into this situation, always start on a new line after an `if`, `elif`, `else`, `for`, `while` etc. statement (that is, any statement ending in a colon `:`). Neither of condition coverage and branch coverage also necessarily implies the other.

The only useful type of coverage in the end is that which can be easily measured while running your test suite and this is *statement coverage*. While you should aim for as comprehensive as possible condition and branch coverage, because these are difficult to measure they are hard to quantitatively achieve.

The standard Python tool to measure statement coverage is `coverage.py`, which you can install with

```
pip install coverage
```

You can then run your tests while collecting test coverage information by replacing the standard

```
pytest ...
```

call with

```
coverage run -m pytest ...
```

This does not print any information on your test coverage, but simply collects the information for use by other `coverage` commands and other tools (in the `.coverage` file). To obtain a report, run

```
coverage report
```

Thus, if we run



```
coverage run -m pytest -v tests/
coverage report
```

we get

Name	Stmts	Miss	Cover	Missing
-----	-----	-----	-----	-----
exampy/__init__.py	1	0	100%	
exampy/_math.py	9	2	78%	73, 88
exampy/integrate/__init__.py	2	0	100%	
exampy/integrate/_integrate.py	8	0	100%	
tests/test_basic_math.py	13	0	100%	
tests/test_integrate.py	27	1	96%	52
-----	-----	-----	-----	-----
TOTAL	60	3	95%	

As you see, this prints an overview of the number of individual statements, statements not run by the test suite (in absolute and relative terms), and the line numbers for statements not run, for every file. Note that the report also contains the files in the test suite. To limit the report to those files in your package, use the `--source` option:

```
coverage run --source=exampy/ -m pytest -v tests/
coverage report
```

we now get

Name	Stmts	Miss	Cover	Missing
-----	-----	-----	-----	-----
exampy/__init__.py	1	0	100%	
exampy/_math.py	9	2	78%	73, 88
exampy/integrate/__init__.py	2	0	100%	
exampy/integrate/_integrate.py	8	0	100%	
-----	-----	-----	-----	-----
TOTAL	20	2	90%	

Sometimes you want to exclude lines or entire code blocks from coverage, for example, when you have a conditional statement that goes to a general error that you do not feel needs to be tested or when you include a function that is not used in the code and that users should not use, but that you want to keep for later use (in that case, it's probably best to remove it and keep it elsewhere for future use, but this isn't an ideal world...). The standard method for doing this is to add a comment `# pragma: no cover` at the end of the statement. For a regular statement, this excludes the current line from the coverage report, but if you add it at the end of a function definition, a conditional statement (e.g., `if x < 0: # pragma: no cover`), a for loop, or any statement ending in a colon, then the entire following code block will be excluded. So for example, the entire function

```
def fourth_power(x): # pragma: no cover
    return x**4.
```

would be excluded from the coverage report. Once you put in this comment, the line will disappear from the coverage report and you will get lulled into a false sense of security about your code's coverage (which will appear higher than it truly is), so this should be used sparingly for lines and code blocks that truly can be ignored at little risk.

You can more generally exclude lines or files using a configuration file, which is by default a `.coveragerc` file in the directory where you run the command; this file has the standard `.ini` format (but does not end in this extension!), an example is

```
[run]
source= exampy/

[report]
# Regexes for lines to exclude from consideration
exclude_lines =
    # Have to re-enable the standard pragma
    pragma: no cover

    # Don't complain if tests don't hit defensive assertion code:
    raise AssertionError
    raise NotImplementedError

    # Don't complain if non-runnable code isn't run:
    if 0:
    if __name__ == '__main__':

omit =
    exampy/__init__.py
    exampy/integrate/*

ignore_errors = True

[html]
directory = coverage_html_report
```

This sets the runtime option `--source=exampy/` so that we don't have to specify this by hand every time we run `coverage run`, the `[report]` section uses the `exclude_lines` option to exclude source lines that contain these expressions (we have to add the standard `pragma: no cover`, because otherwise it would be overwritten by the rules here), the `omit` option to exclude entire files or directories (use very sparingly! This is a bad example!), and the `ignore_errors` to ignore any errors when trying to find source files. The final section specifies where to place the HTML version of the report, which is created by

```
coverage html
```

instead of `coverage report` and when running with this `.coveragerc` file, the HTML page looks like

Coverage report: 78%

Module ↓	statements	missing	excluded	coverage
exampy/_math.py	9	2	0	78%
<b>Total</b>	<b>9</b>	<b>2</b>	<b>0</b>	<b>78%</b>

coverage.py v5.0.3, created at 2020-03-06 15:30

Going forward, we remove the `omit=` part from the `.coveragerc` file so that all files are included in the test coverage reports.

A different way of running the coverage script that is often more convenient is using the `pytest-cov` plugin, which you can install with

```
pip install pytest-cov
```

then you can get the report in one go as (in our example):

```
pytest -v tests/ --cov=exampy/
```

which produces

```
===== test session starts
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 --
/path/
TO/python
cachedir: .pytest_cache
rootdir: /path/TO/exampy
plugins: arraydiff-0.3, cov-2.8.1, doctestplus-0.3.0, openfiles-0.4.0,
remotedata-0.3.1
collected 6 items

tests/test_basic_math.py::test_square_direct PASSED
[ 16%]
tests/test_basic_math.py::test_cube_oddfunction PASSED
[ 33%]
tests/test_integrate.py::test_simps_against_riemann PASSED
[ 50%]
tests/test_integrate.py::test_simps_against_scipy PASSED
[ 66%]
```

(continues on next page)

(continued from previous page)

```
tests/test_integrate.py::test_simps_typererror PASSED
→ [ 83%]
tests/test_integrate.py::test_simps_scalarfunc XFAIL
→ [100%]

----- coverage: platform darwin, python 3.7.3-final-0 -----
Name                                Stmts   Miss  Cover
-----
exampy/__init__.py                   1       0   100%
exampy/_math.py                      9       2    78%
exampy/integrate/__init__.py         2       0   100%
exampy/integrate/_integrate.py       8       0   100%
-----
TOTAL                               20       2    90%

===== 5 passed, 1 xfailed in 1.15s
→ =====
```

To also get the line numbers of statements not run as part of the test suite, do

```
pytest -v tests/ --cov=exampy/ --cov-report term-missing
```

To get the HTML version, do

```
pytest -v tests/ --cov=exampy/ --cov-report html
```

which without the omitted files looks like

Coverage report: 90%

Module ↓	statements	missing	excluded	coverage
exampy/__init__.py	1	0	0	100%
exampy/_math.py	9	2	0	78%
exampy/integrate/__init__.py	2	0	0	100%
exampy/integrate/_integrate.py	8	0	0	100%
<b>Total</b>	<b>20</b>	<b>2</b>	<b>0</b>	<b>90%</b>

coverage.py v5.0.3, created at 2020-03-06 15:40

If your package contains non-Python code, for example, compiled C code to speed up computations, and you want to check the test coverage of this code as well, you need to use additional tools. This is too advanced of a topic to cover in detail here, but to get you on your way here is an overview of the process for C code: (i) to collect coverage information, you need to compile your C code without optimization (`-O0`) and with the `gcov` coverage option `-coverage` (in compilation and linking). This will allow the `gcov` coverage tool to collect coverage information for any execution of your

code (similar to what `coverage run` does above); (ii) use `lcov` to generate a coverage report, e.g., with commands like `lcov --capture --base-directory . --directory build/temp.linux-x86_64-3.7/exampy/ --no-external --output-file coverage.info` which collects the coverage information into the `coverage.info` file, (iii) generate a HTML page with the coverage reports with, e.g., `genhtml coverage.info --output-directory out`. The online services that I discuss in the next chapter also allow you to combine Python and C coverage reports into a single HTML overview.



## AUTOMATICALLY BUILDING AND TESTING YOUR CODE: CONTINUOUS INTEGRATION

Once you have a test suite for your Python package (even if it is not complete yet), you will want to run it often to check that all tests continue to pass as your package evolves. Doing this automatically is the domain of *continuous integration* and for scientific projects it is most easily done using the free, online services that I discuss in this chapter. Like complete documentation and a comprehensive test suite before, continuous integration is an essential component of a modern software package that is used by more than a few people and/or receives contributions from outside users (e.g., through pull requests). Setting up continuous integration for your software package will end up saving you lots of time by finding issues with your code quickly and making it less likely that you merge a change that has unexpected, bad consequences.

### 6.1 Why continuous integration?

Continuous integration (CI) is the practice of integrating all changes into the “main” copy of a code base (here, a Python package) on a frequent basis (“continuously”). The “main” copy in our case is the GitHub repository of the package, which is the basis of all clones and forks of the code (I assume throughout these notes that the only way the development version of the code is shared is through the GitHub site, even for versions used by the same developer). The main reason to perform continuous integration is to catch any unmergeable changes to the code made in different copies of the code quickly when they can typically be more easily resolved. Continuous integration checks both that the package builds successfully and that it passes the tests in the test suite and this is considered to be a successful integration.

To perform continuous integration efficiently, both the build and test system need to be *automated*, that is, they should be able to be run without any human intervention. We have seen how to build a test suite that can be run with a simple `pytest` command in the [previous chapter](#) (page 71); I will discuss how to automate the build with specific examples below. Automating the building and testing of the code is important for taking any human decisions and mistakes out of the loop and for being able to perform the continuous-integration procedure *very often*.

What do we in practice mean by “continuous”, because obviously we aren’t running the build-

and-test procedure all the time? “Continuous” qualitatively means that we run the build-and-test procedure every time the code or any of its dependencies change. In practice, it is easier to know when one’s own code changes than when the code’s dependencies change and we typically run the continuous-integration procedure upon every push of changes to GitHub. That is, we can make multiple commits and run the integration tests each time we push a set of commits to GitHub. Ideally, we would run the integration tests in conjunction with each commit, but since integration tests can take a long time to run, a compromise of running whenever we think the code is ready for a push to GitHub is good and it is easy to set up with automation services. This does mean that one should push changes to GitHub often, typically at least once a day, to make sure that the continuous integration procedure is run often. The code will also change in response to patches or new features submitted through a pull request and it is good practice to run the continuous-integration procedure *before* merging changes from a pull request. It is easy to set this up to be done automatically and, indeed, having continuous integration set up is essential to being able to merge pull requests for your package, because otherwise it is difficult to know that the proposed changes do not break some unexpected part of your code. One typically runs the continuous-integration procedure for changes to any branch, not just `main`.

The way your code runs also changes when its dependencies change. While one could in principle set up a “continuous” check for whether dependencies have changed, in practice this is easiest to spot by running the continuous integration procedure on a fixed schedule in addition to any runs in response to pushes or pull requests. That way, you can ensure that the integration tests are run even when the code is going through a stretch of minor development. These fixed-schedule tests could be run daily or weekly, depending on how often you think dependencies might change and/or how quickly you think you need to catch this. One way in which the automated build-and-test of your code is useful is that it shows that there is a working version of your code and how to get it to work, which you can point people to who have issues with installing and running your code.

The advantages of continuous integration are many: you will find issues quickly, keep your development version in a working state and, thus, always have a fully-functional version of your code during development (that is, not just at releases), and by making use of automated tools to run your integration tests on different types of machines you are able to easily make sure that your code runs on all systems that you support, not just your own. But there are disadvantages as well, the main one being that setting up and maintaining the continuous-integration system takes quite a bit of time and quite often issues that pop up during the integration procedure are due to the way the build-and-test procedure is set up (which can be quite complex for a larger Python package), rather than being due to a real bug in the package itself. For example, the way you install dependencies in the integration step might break and you then have to fix that to keep the continuous-integration procedure working, even though there is likely nothing wrong with your code. Overall, I think even this type of time is well spent, because it is important to know at all times that your code can be built (including its dependencies).

There are *many* services available to perform continuous integration of code, because continuous integration is a crucial aspect of all modern software and software-backed services, allowing bug fixes and updates to be rolled out quickly and often. Continuous integration of course goes far beyond Python packages and is used in the entire range of software, apps, and online services, where it is often combined with *continuous deployment* (CD, leading to the abbreviation CI/CD), the

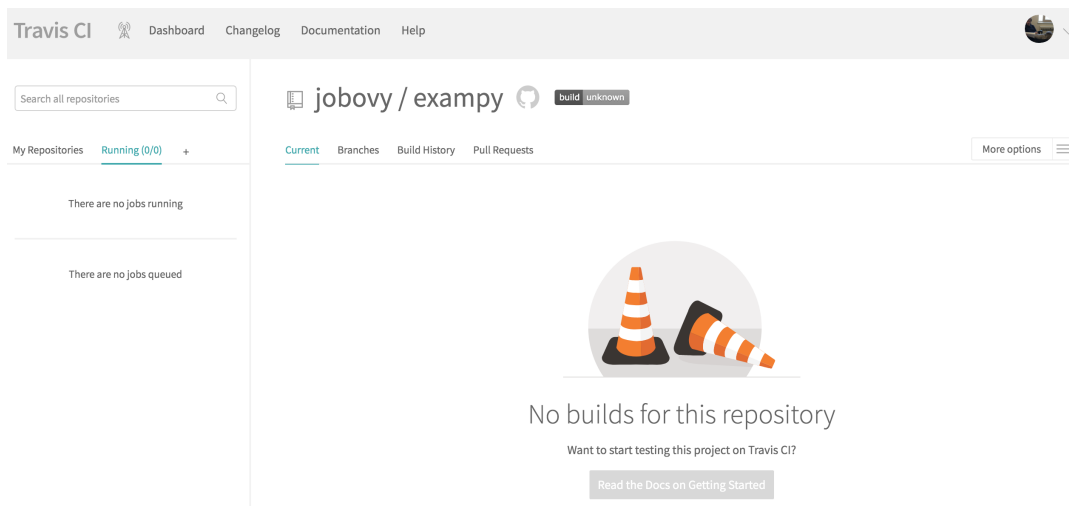


practice of rolling out bug fixes and updates as soon as they are made and pass the integration tests. The most popular CI/CD services are Travis CI, Circle CI, Jenkins, AppVeyor, and since recently GitHub Actions. I will only discuss Travis CI, AppVeyor, and GitHub Actions below, but to a large degree they all work in the same way.

## 6.2 Continuous integration with Travis CI

Travis CI is a continuous-integration service that is seamlessly integrated with GitHub and is free for open-source projects, generously providing you with multiple runners for your build-and-test integrations. The way Travis CI works is that you connect it to your GitHub account, give it access to your repository, and once configured it will automatically run your build and test suites every time you push to any branch and every time somebody opens a pull request. It will notify you when things go wrong (or when things go right! But that's not typically as interesting...) and can send the results from tests on to other services (e.g., those that parse and display your test coverage statistics).

To get started, go to <https://travis-ci.com/> and sign up with your GitHub account. You then need to give Travis CI permission to access your account's information. Once you're back at [travis-ci.com](https://travis-ci.com/), you can click on your profile picture to bring up a big green Activate button, press this to activate the GitHub Apps integration which Travis CI uses to start testing and deploying on Travis CI. Once you have clicked this, you will see a list of your repositories, if you click on the repository that you want to add to Travis CI, you will be brought to its page, which looks like this for the example package `exampy` at first



To start building and testing our package with Travis CI, we have to add a `.travis.yml` configuration file to the top-level directory of our repository and push this to GitHub. This `.travis.yml` file will contain *all* of the information to configure the build-and-test procedure on Travis CI. I will go over all of the parts in due course, so let's start with the simplest possible configuration for our `exampy` project, which looks like

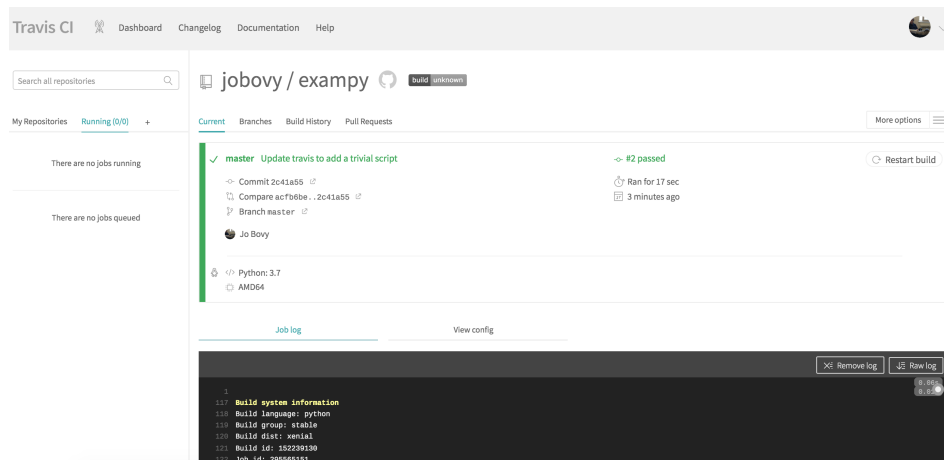
```
language: python

python:
  - "3.7"

install:
  - python setup.py develop

script:
  - echo 0
```

This configuration file does only the most basic things: (i) it states what the language of the code is (Python) and what version to use, (ii) it installs the code in the `install:` section, and (iii) it has a trivial statement in the `script:` section, which will later contain the running of the test suite. The reason that we need to add the trivial statement in the `script:` section is that without this section, Travis CI will label the run as failed. If you add this file to the GitHub repository and push it to GitHub, Travis CI is notified by GitHub that a change occurred in the repository, and Travis CI starts running the continuous integration procedure; the page on Travis CI will change to something like



(where the run should be #1 for you, I messed up my own first run...). If you scroll down, you see the full log of what happened and you will see that Travis CI clones your repository, sets up Python, runs the `install:` section, followed by the `script:` section:

```
154 $ git clone --depth=50 --branch=master https://github.com/jobovy/exampy.git jobovy/exampy
155 Cloning into 'jobovy/exampy'...
156 remote: Enumerating objects: 159, done.
157 remote: Counting objects: 100% (159/159), done.
158 remote: Compressing objects: 100% (80/80), done.
159 remote: Total 159 (delta 72), reused 156 (delta 60), pack-reused 0
160 Resolving objects: 100% (159/159), 19.49 KiB | 0.75 MiB/s, done.
161 Resolving deltas: 100% (72/72), done.
162 $ cd jobovy/exampy
163 $ git checkout -q 2c41a587bd241a63f739174555650e7c7d7eb1
164
165 $ source ~/.virtualenv/python3.7/bin/activate
166 $ python --version
167 Python 3.7.1
168 $ pip --version
169 pip 19.0.3 from /home/travis/virtualenv/python3.7.1/lib/python3.7/site-packages/pip (python 3.7)
170 $ python setup.py develop
171
172 $ echo 0
173 0
174 The command "echo 0" exited with 0.
175
176 Done. Your build exited with 0.
```

Next, we want to run our test suite with `pytest` and print the test coverage information using the `pytest-cov` plugin. To do this, we need to install `pytest` and the `pytest-cov` plugin in the `install:` section as well, and then run our test suite in the `script:` section, such that `.travis.yml` file now looks as follows

```
language: python

python:
  - "3.7"

install:
  - pip install pytest
  - pip install pytest-cov
  - python setup.py develop

script:
  - pytest -v tests/ --cov=exampy/
```

Pushing this change to GitHub, Travis CI automatically clones the updated repository and runs the build-and-test procedure. However, the tests fail, with the pertinent part of the log being

```
220 $ pytest -v tests/ --cov=exampy/
221 ===== test session starts =====
222 platform linux -- Python 3.7.1, pytest-4.3.1, py-1.7.0, pluggy-0.8.0 -- /home/travis/virtualenv/python3.7.1/bin/python
223 cachedir: .pytest_cache
224 rootdir: /home/travis/build/jobovy/exampy, inifile:
225 plugins: cov-2.8.1
226 collected 6 items
227
228 tests/test_basic_math.py::test_square_direct PASSED [ 16%]
229 tests/test_basic_math.py::test_cube_oddfunction PASSED [ 33%]
230 tests/test_integrate.py::test_simps_against_riemann PASSED [ 50%]
231 tests/test_integrate.py::test_simps_against_scipy FAILED [ 66%]
232 tests/test_integrate.py::test_simps_typerror PASSED [ 83%]
233 tests/test_integrate.py::test_simps_scalarfunc XFAIL [100%]
234
235 ===== FAILURES =====
236 test_simps_against_scipy
237
238 def test_simps_against_scipy():
239     # Test that exampy.integrate.simps integration agrees with
240     # scipy.integrate.quad
241     > from scipy import integrate as sc_integrate
242     E ModuleNotFoundError: No module named 'scipy'
243
244 tests/test_integrate.py:22: ModuleNotFoundError
245
246 ===== warnings summary =====
247 exampy/integrate/_integrate.py:186
248 /home/travis/build/jobovy/exampy/exampy/integrate/_integrate.py:56: DeprecationWarning: invalid escape sequence \l
249 """
250
251 -- Docs: https://docs.pytest.org/en/latest/warnings.html
252
253 ----- coverage: platform linux, python 3.7.1-final-0 -----
254 Name                               Stmts   Miss  Cover
255 -----
256 exampy/_init_.py                     1     0   100%
257 exampy/math.py                       9     2    78%
258 exampy/integrate/_init_.py           2     0   100%
259 exampy/integrate/_integrate.py       8     0   100%
260 -----
261 TOTAL                                20     2    90%
262
263 ===== 1 failed, 4 passed, 1 xfailed, 1 warnings in 0.33 seconds =====
264 The command "pytest -v tests/ --cov=exampy/" exited with 1.
```

The tests failed, because we forgot to install `scipy`, which is not a dependency of the `exampy` package itself (and, thus, we didn't list it in the `install_requires` section of the `setup.py` file), but it is a dependency of the test suite, because in one test we compare the `exampy.integrate.simps` procedure to numerical integration in `scipy`. Thus, we need to also install `scipy`, using the following `.travis.yml`

```
language: python
```

(continues on next page)

(continued from previous page)

```
python:
- "3.7"

install:
- pip install pytest
- pip install pytest-cov
- pip install scipy
- python setup.py develop

script:
- pytest -v tests/ --cov=exampy/
```

This time the tests pass without a hitch!

```
172 $ pip install pytest
181 $ pip install pytest-cov
197 $ pip install scipy
204 $ python setup.py develop
227 $ pytest -v tests/ --cov=exampy/
===== test session starts =====
228 platform linux -- Python 3.7.1, pytest-4.3.1, py-1.7.0, pluggy-0.8.0 -- /home/travis/virtualenv/python3.7.1/bin/python
236 cachedir: .pytest_cache
231 rootdir: /home/travis/build/jobovy/exampy, inifile:
232 plugins: cov-2.8.1
233 collected 6 items
234
235 tests/test_basic_math.py::test_square_direct PASSED [ 16%]
236 tests/test_basic_math.py::test_cube_oddfunction PASSED [ 33%]
237 tests/test_integrate.py::test_sims_against_riemann PASSED [ 50%]
238 tests/test_integrate.py::test_sims_against_scipy PASSED [ 66%]
239 tests/test_integrate.py::test_sims_typeerror PASSED [ 83%]
240 tests/test_integrate.py::test_sims_scalarfunc XFAIL [100%]
241
242 ===== warnings summary =====
243 exampy/integrate/_integrate.py:56
244 /home/travis/build/jobovy/exampy/exampy/integrate/_integrate.py:56: DeprecationWarning: invalid escape sequence \l
245 ***
246 -- Docs: https://docs.pytest.org/en/latest/warnings.html
247
248 ----- coverage: platform linux, python 3.7.1-final-0 -----
249
250 Name                               Stmts  Miss  Cover
251 -----
252 exampy/_init_.py                     1      0 100%
253 exampy/math.py                       9      2   78%
254 exampy/integrate/_init_.py           2      0 100%
255 exampy/integrate/_integrate.py       8      0 100%
256
257 TOTAL                                20      2   90%
258
259 ===== 5 passed, 1 xfailed, 1 warnings in 0.52 seconds =====
260 The command "pytest -v tests/ --cov=exampy/" exited with 0.
261
262
263 Done. Your build exited with 0.
```

The `.travis.yml` configuration file has *lots* of possible sections to customize your build and test runs in any way that you want, which are documented in full [here](#). When you are using the linux operating system (the default), you have access to a full Ubuntu distribution and you can run arbitrary commands in the sections `install:`, `before_install:` (which contains commands to run before the installation of your code; technically it would have been better to install the test dependencies in the `before_install:` section, or in the `before_script:` section [see below] because they are not necessary for the code's installation itself), `script:`, `before_script:`, etc. There are also many other sections that allow you to install some dependencies more easily, to define environment variables, to run your build-and-test integrations with different versions of dependencies (and Python itself), and there are sections `after_success:` and `after_failure:` to customize what Travis CI does upon successful or unsuccessful execution of your build-and-test run. In the remainder of this section, I will cover some of the most commonly-used customizations.

As a first example of a customization, let's say we want to explicitly specify the `numpy` version used by our code. To do this, we add an `env:` section that contains an environment variable `NUMPY_VERSION` that we set to the desired version. Then we can use this environment variable in the rest of the configuration file, e.g., as

```
language: python

python:
  - "3.7"

env:
  - NUMPY_VERSION=1.18

before_install:
  - pip install numpy==$NUMPY_VERSION

install:
  - python setup.py develop

before_script:
  - pip install pytest
  - pip install pytest-cov
  - pip install scipy

script:
  - pytest -v tests/ --cov=exampy/
```

where we have now explicitly included the `numpy` dependency install in the `before_install:` section (previously, we used the default `numpy` version available for this Travis CI disk image) and we have moved the installations that are only necessary for the test suite to the `before_script:` section. Pushing this change to GitHub, the log on Travis CI now shows that the requested version of `numpy` is installed, with the relevant part of the log being:

## Python code packaging for scientific software

```
101 $ git clone --depth=50 --branch=master https://github.com/jobovy/exampy.git jobovy/exampy
102
103
104
105
106
107 Setting environment variables from .travis.yml
108 $ export NUMPY_VERSION=1.18
109
110 $ source ~/.virtualenv/python3.7/bin/activate
111 $ python --version
112 Python 3.7.1
113 $ pip --version
114 pip 19.0.3 from /home/travis/.virtualenv/python3.7.1/lib/python3.7/site-packages/pip (python 3.7)
115 $ pip install numpy==$NUMPY_VERSION
116 Collecting numpy==1.18
117   Downloading https://files.pythonhosted.org/packages/26/53/127cb49435bcfd841ba8eafa839931c62a9eac577a641f6c2293d23371/numpy-1.18.0-cp37-m
manylinux_x86_64.whl (28.1MB)
118
119 Installing collected packages: numpy
120 Found existing installation: numpy 1.15.4
121 Uninstalling numpy-1.15.4:
122   Successfully uninstalled numpy-1.15.4
123 Successfully installed numpy-1.18.0
124 $ python setup.py develop
125
126 $ pip install pytest
127
128 $ pip install pytest-cov
129
130 $ pip install scipy
131
132 $ pytest -v tests/ --cov=exampy/
===== test session starts =====
platform linux -- Python 3.7.1, pytest-4.3.1, py-1.7.0, pluggy-0.8.0 -- /home/travis/.virtualenv/python3.7.1/bin/python
cachedir: .pytest_cache
rootdir: /home/travis/build/jobovy/exampy, inifile:
plugins: cov-2.8.1
collected 6 items

tests/test_basic_math.py::test_square_direct PASSED [ 16%]
tests/test_basic_math.py::test_cube_oddfunction PASSED [ 33%]
tests/test_integrate.py::test_simps_against_fineasm PASSED [ 50%]
tests/test_integrate.py::test_simps_against_scipy PASSED [ 66%]
tests/test_integrate.py::test_simps_timestep PASSED [ 83%]
tests/test_integrate.py::test_simps_scalarfunc XFAIL [100%]
```

You can set multiple environment variables in the `env:` section, but note that all definitions should be part of a *single* dash. E.g., to also specify the `scipy` version, do

```
language: python
```

```
python:
  - "3.7"
```

```
env:
  - NUMPY_VERSION=1.18
  - SCIPY_VERSION=1.4
```

```
before_install:
  - pip install numpy==$NUMPY_VERSION
```

```
install:
  - python setup.py develop
```

```
before_script:
  - pip install pytest
  - pip install pytest-cov
  - pip install scipy==$SCIPY_VERSION
```

```
script:
  - pytest -v tests/ --cov=exampy/
```

rather than

```
...
env:
```

(continues on next page)

(continued from previous page)

```
- NUMPY_VERSION=1.18
- SCIPY_VERSION=1.4
...
```

because the latter would create two jobs, one with the `numpy` version set, the other with the `scipy` version set (see below). Another use of environment variables is to split your tests into sets to be executed in parallel, by explicitly setting the test files fed to `pytest` as an environment variable.

One of the great advantages of using a cloud-based continuous-integration system is that you can easily test your code with different versions of Python and your code's dependencies. Travis CI allows you to easily build *matrices* of jobs, by taking multiple values listed in some of the configuration sections. For example, if you list two major Python versions in the `python:` section and three minor `numpy` versions in the `env:` section as above, naively following the recommendations from a recent [proposal](#) for which Python and `numpy` versions packages should support (the recommendation isn't actually to support all of these combinations), Travis CI will run 6 different jobs, one for each combination of Python and `numpy` version. In the example that we have been considering, this gives the following `.travis.yml`:

```
language: python

python:
  - "3.8"
  - "3.7"

env:
  - NUMPY_VERSION=1.18
  - NUMPY_VERSION=1.17
  - NUMPY_VERSION=1.16

before_install:
  - pip install numpy==$NUMPY_VERSION

install:
  - python setup.py develop

before_script:
  - pip install pytest
  - pip install pytest-cov
  - pip install scipy

script:
  - pytest -v tests/ --cov=exampy/
```

Pushing this to GitHub, we see that the Travis CI page for the current build-and-test integration

## Python code packaging for scientific software

looks as follows when the integration test is in progress:

The screenshot shows the Travis CI interface for the repository `jobovy/exampy`. The build status is **passing**. The main panel displays the build matrix for the `master` branch, titled "Make build matrix with two Python versions and three numpy v...". It shows a commit `810868b` and a branch `master`. The build is running for 1 min 11 sec. Below this, a table lists the build jobs:

Job ID	Platform	Python	NUMPY_VERSION	Duration
# 8.1	AMD64	Python: 3.8	NUMPY_VERSION=1.18	37 sec
# 8.2	AMD64	Python: 3.8	NUMPY_VERSION=1.17	1 min 11 sec
# 8.3	AMD64	Python: 3.8	NUMPY_VERSION=1.16	1 min 9 sec
# 8.4	AMD64	Python: 3.7	NUMPY_VERSION=1.18	27 sec
# 8.5	AMD64	Python: 3.7	NUMPY_VERSION=1.17	25 sec
# 8.6	AMD64	Python: 3.7	NUMPY_VERSION=1.16	27 sec

Rather than seeing the log for a single job, we now see an overview of the six created jobs for the six combinations of Python and numpy versions. Some of these run in parallel. Clicking on one of the jobs will bring up the log for that particular job and Travis CI will report the status of each job as either a success or a failure; the entire combination only succeeds if all component jobs succeed (unless you [allow failures](#)). Once all jobs finish running, the final status looks as follows:

The screenshot shows the Travis CI interface for the repository `jobovy/exampy`. The build status is **errored**. The main panel displays the build matrix for the `master` branch, titled "Make build matrix with two Python versions and three numpy v...". It shows a commit `810868b` and a branch `master`. The build is errored. Below this, a table lists the build jobs:

Job ID	Platform	Python	NUMPY_VERSION	Duration
# 8.1	AMD64	Python: 3.8	NUMPY_VERSION=1.18	37 sec
# 8.2	AMD64	Python: 3.8	NUMPY_VERSION=1.17	2 min 34 sec
# 8.3	AMD64	Python: 3.8	NUMPY_VERSION=1.16	2 min 51 sec
# 8.4	AMD64	Python: 3.7	NUMPY_VERSION=1.18	27 sec
# 8.5	AMD64	Python: 3.7	NUMPY_VERSION=1.17	25 sec
# 8.6	AMD64	Python: 3.7	NUMPY_VERSION=1.16	27 sec

We see that one job failed! Another thing you notice when looking at the duration of each job is that the Python v3.8 / numpy v1.18 and all of the Python v3.7 finish in about 30 seconds, the other two Python v3.8 jobs run for about three minutes. Upon inspection of the logs, it becomes clear that the reason for this is that for these two jobs, no binary wheels for numpy are available; I will discuss binary wheels more in [Chapter 7](#) (page 129), but for now the important thing to know about them is that they allow you to install a package with `pip` without building it on your computer (which is what `pip` normally does). Thus, for these two versions, `pip install numpy` builds numpy from source and this takes a while. These wheels aren't available, because the correct reading of the [NEP 29 proposal](#) is that Python 3.8 only needs to support numpy version 1.18 and numpy therefore did not build binary distributions for earlier numpy versions with Python 3.8. For the Python v3.8



/ numpy v1.17 combination, the build from source actually fails on Travis CI, which is the cause of this job's failure.

When your code includes a few dependencies that take a long (more than a dozen or so seconds) time to build from source (e.g., *numpy* and *scipy*), it is beneficial to use the [Anaconda](#) Python distribution, which includes built versions of many packages that you might use. Because the entire Anaconda distribution is large and your Travis CI runs would have to download it every time, it is good to use the [Miniconda](#) version of Anaconda instead, which is a bare-bones version of Anaconda that comes with very few packages pre-installed to create a light-weight distribution. Using Miniconda requires us to add a few lines to the `before_install:` section to download Miniconda, set it up, and use it to install the dependencies. Note that we need to make sure that Miniconda gets set up for the same Python version that we requested in the `python:` section, by using the `$TRAVIS_PYTHON_VERSION` environment variable that Travis CI automatically defines (but note that the Miniconda Python version does not have to be the same as the one specified in the `python:` section; keeping them the same using the `$TRAVIS_PYTHON_VERSION` environment variable is useful to avoid confusion though).

Re-writing the example above using Miniconda looks like

```
language: python

python:
  - "3.8"
  - "3.7"

env:
  - NUMPY_VERSION=1.18.1
  - NUMPY_VERSION=1.17,4
  - NUMPY_VERSION=1.16.6

before_install:
  - wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_
    ↪64.sh -O miniconda.sh
  - bash miniconda.sh -b -p $HOME/miniconda
  - export PATH="$HOME/miniconda/bin:$PATH"
  - hash -r
  - conda config --set always_yes yes --set changeps1 no
  - conda update conda
  - conda config --add channels conda-forge
  - conda create -n test-environment python=$TRAVIS_PYTHON_VERSION
    ↪"numpy==$NUMPY_VERSION" scipy pip
  - source activate test-environment

install:
  - python setup.py develop
```

(continues on next page)

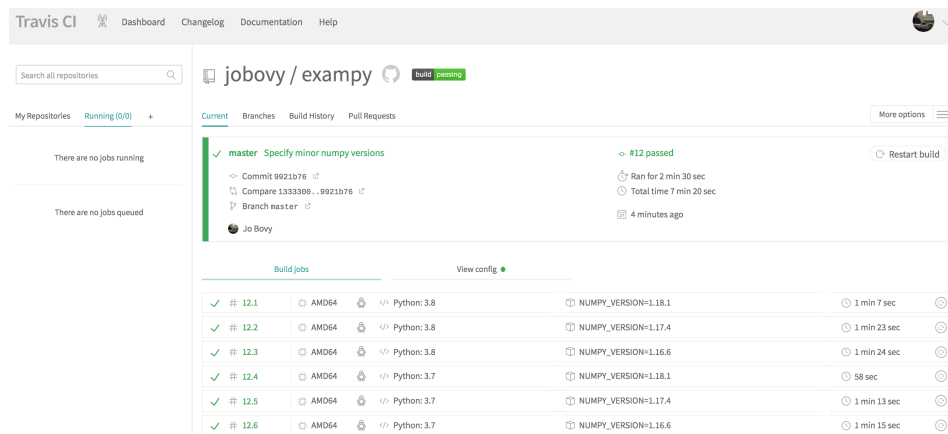
(continued from previous page)

```
before_script:
- pip install pytest
- pip install pytest-cov

script:
- pytest -v tests/ --cov=exampy/
```

Note that I have taken advantage of the fact that `scipy` can also be more easily installed using `conda` to simply include it in the main `conda create` command in the `before_install:` section. I also specified the exact version of `numpy` to use, because otherwise for, e.g., v1.17 `conda` tries to install v1.17.0 (which in this case fails, because it does not exist in Anaconda for Python 3.8).

This run now was successful, with the overview at the end looking like



The screenshot shows the Travis CI web interface for the repository 'jobovy / exampy'. The main build status is 'Build passing' (green). The build details show a successful run for the 'master' branch with the commit '9921b76'. The build passed 12 tests, ran for 2 minutes and 30 seconds, and the total time was 7 minutes and 20 seconds. Below the main build status, there is a table of build jobs. Each job is successful (green checkmark) and runs on an AMD64 architecture with Python 3.8. The jobs are numbered 12.1 through 12.6, and each has a duration of approximately 1 minute. The table also shows the environment variables for each job, including 'NUMPY\_VERSION'.

Job #	Architecture	Python Version	Environment Variables	Duration
12.1	AMD64	Python: 3.8	NUMPY_VERSION=1.16.1	1 min 7 sec
12.2	AMD64	Python: 3.8	NUMPY_VERSION=1.17.4	1 min 23 sec
12.3	AMD64	Python: 3.8	NUMPY_VERSION=1.16.6	1 min 24 sec
12.4	AMD64	Python: 3.7	NUMPY_VERSION=1.18.1	58 sec
12.5	AMD64	Python: 3.7	NUMPY_VERSION=1.17.4	1 min 13 sec
12.6	AMD64	Python: 3.7	NUMPY_VERSION=1.16.6	1 min 15 sec

Each job now takes about a minute to run, showing that using Miniconda significantly speeds up the run compared to building `numpy` from source (which above led to a > 2 minute job time; building `scipy` from source would more than double that).

At this point, your email inbox will be filled with emails from Travis CI telling you about the status of each integration run. You typically do not want to be updated about the status of every run, but simply when the status changes. For example, if a set of runs all end successfully, it's not that useful to get an email about this every time, but if a run suddenly fails, you will want to know that. Similarly, if your runs have been failing, it's useful to know when they start passing again (although that you will more likely be checking on the website directly). To only get notified upon a change in status from success to failure or failure to success, add a section to your `.travis.yml` file at the end that looks like

```
notifications:
  email:
    recipients:
      - YOUR_EMAIL
```

(continues on next page)

(continued from previous page)

```
on_success: change
on_failure: change
```

Other useful parts of the `.travis.yml` file that I will not discuss in detail are:

- **addons:** allows you to specify dependencies that can be installed using, e.g., standard linux tools like `apt-get`. For example, to use the GNU scientific library, use

```
addons:
  apt:
    packages:
      - libgsl0-dev
```

- If your tests make plots, you will run into errors, because the plots cannot be displayed. To avoid those, include the `xvfb` service that allows you to run graphical applications without a display:

```
services:
  - xvfb
```

- Building a matrix of jobs by multiplying options set in sections such as `python:` and `env:` quickly leads to large build matrices. If you just want to include an additional job with different parameters, you can include individual jobs in the `matrix:` section, e.g., include a single job that uses Python v3.6 and `numpy` v1.18.1 with

```
matrix:
  include:
    - python: "3.6"
      env: NUMPY_VERSION=1.18.1
```

You can also [exclude](#) jobs from the matrix, for example, to actually follow the [NEP 29 support proposal](#), you can exclude the unnecessary jobs for Python 3.8 in the Travis CI configuration that I discussed just before the Miniconda discussion, as

```
matrix:
  exclude:
    - python: "3.8"
      env: NUMPY_VERSION=1.17
    - python: "3.8"
      env: NUMPY_VERSION=1.16
```

Full documentation on the build matrix is given [here](#).

Finally, by default, Travis CI will run when you push a new commit or set of commits to GitHub (for any branch) or when someone opens or updates a pull request (in that case, Travis CI will

automatically add status updates on whether the integration tests pass to the pull request’s page). If in addition to this, you want the build of, say, your `main` branch to happen at least weekly whether or not a change occurred, you can do this with a “cron job” by going to your package’s Travis CI page and navigating to the “Settings” under the “More Options” menu. There, there is a “Cron Jobs” part where you can choose a branch, how often to run (daily, weekly, or monthly currently), and whether to always run or run only if there hasn’t been a commit in the last 24 hours. This page looks as follows if you add the `main` branch on a weekly schedule, only running when there hasn’t been a change in the last day:



I will discuss how you can process test coverage with Travis CI [below](#) (page 118).

A final note: getting the installation of your dependencies, the build of your own package, and the test suite to run on Travis CI can be difficult and you will often end up with frustrating failed builds that are difficult to diagnose. In writing this section, I ran into syntax errors and unexpected behavior multiple times. Unfortunately, there is nothing much you can do about this, because as far as I am aware one cannot replicate the Travis CI builds locally to test them before running them on Travis CI. But there is lots of info in the online documentation and on the web to help you out and in the end it’s worth the effort to know that your package is working as you expect at all times.

## 6.3 Continuous integration for Windows: AppVeyor

Travis CI is great for testing your integration suite on a Linux operating system and it also has support for Mac OS X, although in practice running Python packages on Macs is so similar to running on Linux that there is little reason to test Mac OS X separately. But if you want to support Windows users, you will want to build and test your code on Windows, because there are subtle ways in which pure Python programs that you think are universal will fail on Windows (e.g., if you explicitly write paths as `/path/to/file` rather than using `os.path.join`, which would properly put in the backslashes for Windows) and any compiled code will require significant modifications to run under Windows. While Travis CI has some support for Windows, only a limited set of features is available. However, AppVeyor is a continuous-integration system that is built around supporting Windows and it, like Travis CI, allows free use for open-source projects (although under somewhat less generous terms: only a single runner is available at any time).

Overall AppVeyor works similar to Travis CI. To start using it, sign in with your GitHub account, add your project by choosing among your GitHub repositories that will be displayed when you log in, and configure your build with a `.appveyor.yml` file (there is also a user interface, but it’s easier to configure AppVeyor similar to Travis CI with a `.yaml` file and you cannot both use the `.appveyor.yml` file and the user interface). The structure of the `.appveyor.yml` file is broadly similar to the `.travis.yml` file, with sections to set up the environment, the matrix of builds, the installation of dependencies and of the package itself, and a section to run the tests. AppVeyor

builds run in the CMD shell or the [Windows PowerShell](#), a shell that is similar in spirit to Unix-type shells, but quite different in many ways; I will mainly use the CMD shell, because it is more similar to Unix-style shells (but you can mix CMD and Powershell within a single `.appveyor.yml` file). You can write arbitrary commands just as in the `.travis.yml` file. This is a good time to point out that if you want to make your code work on Windows and set up the AppVeyor integration tests, it's helpful to have a Windows machine around to test things locally. If you don't have a Windows computer at hand, you can use a virtual machine, such as [VirtualBox](#).

To illustrate the structure of the `.appveyor.yml` file, let's write the equivalent of the simple single-Python-and-numpy that we got working in the previous section. This looks as follows

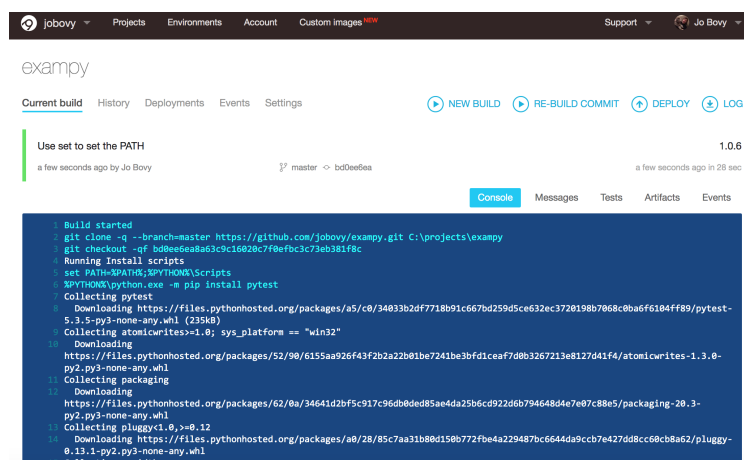
```
build: off

environment:
  PYTHON: "C:\\Python37"

install:
  - "set PATH=%PATH%;%PYTHON%\\Scripts"
  - "%PYTHON%\\python.exe -m pip install pytest"
  - "%PYTHON%\\python.exe -m pip install pytest-cov"
  - "%PYTHON%\\python.exe -m pip install scipy"
  - "%PYTHON%\\python.exe setup.py develop"

test_script:
  - pytest -v tests/ --cov=exampy/
```

Provided that we have added our package to our AppVeyor account, pushing this `.appveyor.yml` file to the package's GitHub repository will trigger a run of the AppVeyor integration test and we see the log in a similar way under the "Current build" as in Travis CI:



```
Build started
git clone -q --branch=master https://github.com/jobovy/exampy.git C:\projects\exampy
git checkout -q b0e6e6a8a63c9c16828c7f0efbc3c73eb381f8c
Running install scripts
set PATH=%PATH%;%PYTHON%\Scripts
%PYTHON%\python.exe -m pip install pytest
Collecting pytest
  Downloading https://files.pythonhosted.org/packages/a5/c0/34833b26f7718b91c667bd259d5ce632ec3720198b7068c0ba6f6104ff89/pytest-5.3.5-py3-none-any.whl (235kB)
Collecting atomicwrites<1.0; sys_platform == "win32"
  Downloading https://files.pythonhosted.org/packages/52/90/6155aa926f43f2b2a22b01be7241be3bfd1cea7d0b3267213e8127d41f4/atomicwrites-1.3.0-py2.py3-none-any.whl
Collecting packaging
  Downloading https://files.pythonhosted.org/packages/62/8a/34641d2bf5c917c96db0ed85ae4da25b6cd922d6b794648d4e7e07c88e5/packaging-20.3-py2.py3-none-any.whl
Collecting pluggy<1.0, >=0.12
  Downloading https://files.pythonhosted.org/packages/a0/28/85c7aa31b80d150b772f0e4a229487bc6644da9ccb7e427dd8cc66c8ba62/pluggy-0.13.1-py2.py3-none-any.whl
Collecting wcwidth
```

What happens in this `.appveyor.yml` file is the following: (i) we turn `build: off`, because `build:` is a MS-specific build process that we don't use for Python packages, (ii) we choose one of the [pre-installed Python](#) versions, (iii) we install dependencies and the package in a similar way as

in `.travis.yml`, with some Windows-specific tweaks, and (iv) we run the test script. Note that we need to add the `C:\\Python37\\Scripts` directory to the `PATH`, because otherwise the `pytest` script is not found. Similar to Travis CI above, getting the AppVeyor integration runs to work can be a bit of trial and error and as you can see above, it took me six tries to get this simple example to work properly!

Note that you can use both CMD and PowerShell syntax in the same `.appveyor.yml`, you simply need to prefix any PowerShell statements with `ps:`, for example, you can do the path assignment in the PowerShell in the example above:

```
build: off

environment:
  PYTHON: "C:\\Python37"

install:
  - ps: $env:PATH="$env:PATH;$env:PYTHON\\Scripts"
  - "%PYTHON%\\python.exe -m pip install pytest"
  - "%PYTHON%\\python.exe -m pip install pytest-cov"
  - "%PYTHON%\\python.exe -m pip install scipy"
  - "%PYTHON%\\python.exe setup.py develop"

test_script:
  - pytest -v tests/ --cov=exampy/
```

Like with Travis CI above, you can define matrices of jobs to run and you can also use Miniconda to run Python, which is [natively installed](#), so you don't have to download Miniconda like on Travis CI. One limitation of the build matrices on AppVeyor is that you cannot automatically multiply different environment variables, but you have to write out all combinations by hand. So to follow the recommendations from the [NEP 29 proposal](#) about which Python/numpy versions to support (see discussion above), we simply write out all four combinations. Using Miniconda for dependencies, this can be achieved with the following `.appveyor.yml`

```
build: off

environment:
  MINICONDA: C:\\Miniconda37-x64

matrix:
  - PYTHON_VERSION: 3.8
    NUMPY_VERSION: 1.18.1

  - PYTHON_VERSION: 3.7
    NUMPY_VERSION: 1.18.1
```

(continues on next page)

(continued from previous page)

```
- PYTHON_VERSION: 3.7
  NUMPY_VERSION: 1.17.4

- PYTHON_VERSION: 3.7
  NUMPY_VERSION: 1.16.6

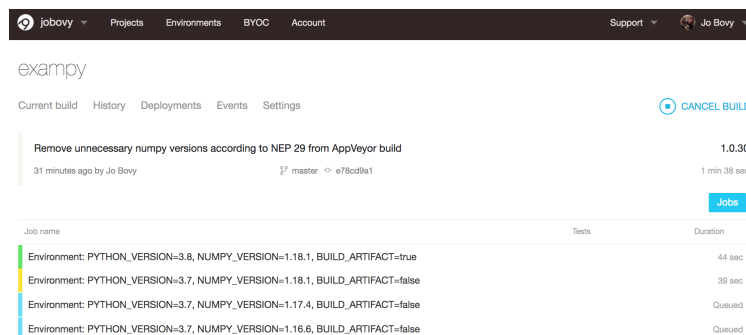
install:
- cmd: call %MINICONDA%\Scripts\activate.bat
- cmd: conda.exe update --yes --quiet conda
- "set PATH=%MINICONDA%;%MINICONDA%\Scripts;%PATH%"
- conda config --set always_yes yes --set changeps1 no
- conda info -a
- "conda create -n test-environment python=%PYTHON_VERSION% numpy==
  ↪ %NUMPY_VERSION% scipy"
- activate test-environment
- python setup.py develop

before_test:
- pip install pytest
- pip install pytest-cov

test_script:
- pytest -v tests/ --cov=exampy/
```

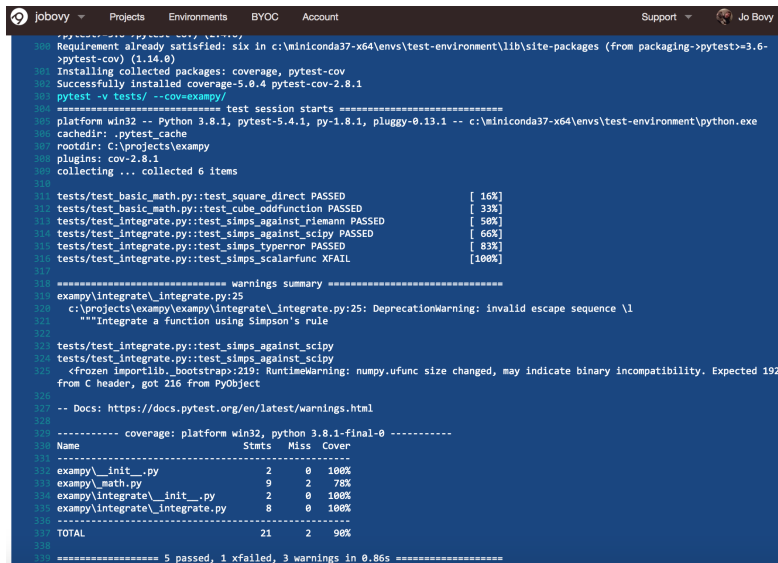
Note that here I use the Miniconda for Python v3.7, even for the tests that use Python v3.8, because at the time of writing there is no direct support for Python v3.8 on AppVeyor. As you can see, the `.appveyor` file is quite similar to the `.travis.yml` file for this setup, except that we need to manually write out this job matrix.

Once we push this new `.appveyor.yml` file to GitHub, the various jobs start running and the “Current build” page changes to an overview of the different jobs:



Soon all of the individual jobs finish successfully and we can inspect their individual logs by clicking on them. For example, the final part of the Python v3.8 / numpy v1.18.1 looks like





```
100 Requirement already satisfied: six in c:\miniconda37-x64\envs\test-environment\lib\site-packages (from packaging->pytest>3.6-
101 >pytest-cov) (1.14.0)
102 Installing collected packages: coverage, pytest-cov
103 Successfully installed coverage-5.0.4 pytest-cov-2.8.1
104 pytest -v tests/ --cov=exampy/
105 ===== test session starts =====
106 platform win32 -- Python 3.8.1, pytest-5.4.1, py-1.8.1, pluggy-0.13.1 -- c:\miniconda37-x64\envs\test-environment\python.exe
107 cachedir: .pytest_cache
108 rootdir: C:\projects\exampy
109 plugins: cov-2.8.1
110 collecting ... collected 6 items
111
112 tests/test_basic_math.py::test_square_direct PASSED [ 16%]
113 tests/test_basic_math.py::test_cube_oddfunction PASSED [ 33%]
114 tests/test_integrate.py::test_simps_against_riemann PASSED [ 50%]
115 tests/test_integrate.py::test_simps_against_scipy PASSED [ 66%]
116 tests/test_integrate.py::test_simps_typeerror PASSED [ 83%]
117 tests/test_integrate.py::test_simps_scalarfunc XFAIL [100%]
118
119 ===== warnings summary =====
120 exampy\integrate\integrate.py:25: DeprecationWarning: invalid escape sequence \l
121     """Integrate a function using Simpson's rule
122
123 tests/test_integrate.py::test_simps_against_scipy
124 tests/test_integrate.py::test_simps_against_scipy
125 <frozen importlib._bootstrap>:219: RuntimeWarning: numpy.ufunc size changed, may indicate binary incompatibility. Expected 192
126 from C header, got 216 from PyObject
127
128 -- Docs: https://docs.pytest.org/en/latest/warnings.html
129
130 ----- coverage: platform win32, python 3.8.1-final-0 -----
131
132 Name                               Stmts   Miss  Cover
133 -----
134 exampy\__init__.py                   2     0   100%
135 exampy\math.py                       9     2    78%
136 exampy\integrate\__init__.py         2     0   100%
137 exampy\integrate\integrate.py        8     0   100%
138 -----
139 TOTAL                                21     2    90%
140
141 ===== 5 passed, 1 xfailed, 3 warnings in 0.86s =====
```

As you can see, the tests pass and we are indeed using Python v3.8.

AppVeyor by default will only email you upon a change in the status, so there is no need to configure that directly. Like Travis CI, AppVeyor will run for any change to any branch and for any opened or updated pull request.

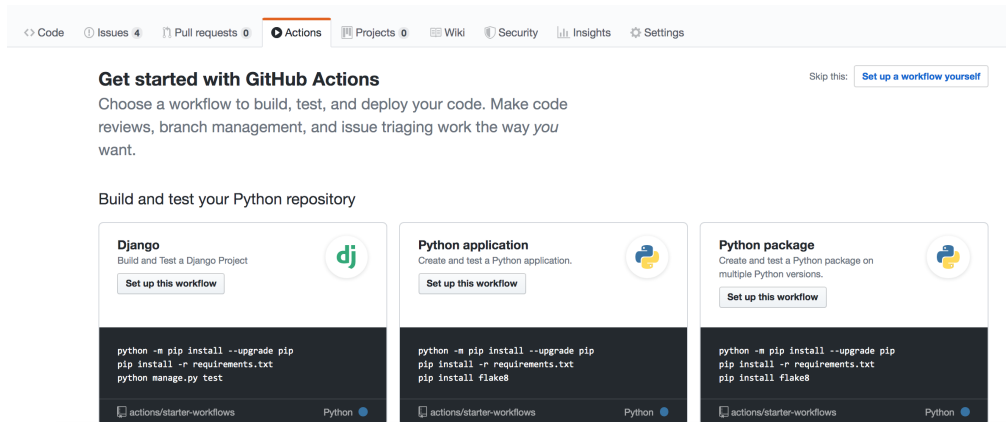
## 6.4 GitHub Actions, the new kid on the block

GitHub itself has recently unveiled a continuous-integration service called [GitHub Actions](#). In many ways GitHub Actions gives similar functionality to Travis CI and AppVeyor, but it has (at least) two main advantages: (1) you can perform a variety of different tasks by defining multiple “workflows” in their own .yml files, rather than having to configure the entire integration process in one .yml file, and (2) you can easily use tasks, or “actions”, written by others to perform steps in your own workflow and you can write and share your own. The first advantage means that you can do things like run the integration tests upon each push or pull request, create binary distributions for your code, automatically respond to Issues and Pull requests, automatically publish your package to a package distribution, publish your documentation (e.g., if you host it yourself), by defining multiple different workflows. The second advantage means that you can significantly simplify the way you write your own workflows by making use of actions that perform atomic operations. These atomic actions are things like checking out the source repository, installing python, installing Mini-conda and setting up a conda environment, but also things like installing LaTeX, spinning up an SSH agent for authentication, uploading files to AWS S3 buckets, etc. This way of using and sharing code is truly following the GitHub spirit.

GitHub Actions are, obviously, seamlessly integrated with GitHub. Similar to Travis CI and AppVeyor above, they are defined using .yml configuration file, but now you can have multiple ones. These files live in the .github/workflows/ directory of your repository. Adding one of these files to your repository will automatically set up GitHub Actions to run for your repository,



so there is nothing to do beyond writing the file. You may have noticed that your package’s GitHub page has an “Actions” tab and if you navigate there, you see a page that looks like



You could get started building your continuous-integration workflow there, but we will simply add a `.yaml` file ourselves to the `.github/workflows/` directory to do this instead.

To define GitHub Actions workflows, it’s easiest to use the online GitHub editor to add a new file or edit an existing file. This is because GitHub will automatically allow you to check the syntax of the file before committing it, by clicking on “preview”, which is handy to not get too many failed builds due to syntax errors (I have found this to happen often with workflow configuration files written in a separate editor and pushed to the repository like other files). Therefore, navigate to the package’s GitHub repository and click on “create a new file” just above the directory listing. We’ll add a file `.github/workflows/test_package.yaml`. To run the same integration tests that we ran on Travis CI and AppVeyor, we can use the following file

```
name: Test exampy

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.7, 3.8]
        numpy-version: [1.16, 1.17, 1.18]
        exclude:
          - python-version: 3.8
            numpy-version: 1.16
          - python-version: 3.8
            numpy-version: 1.17
    steps:
      - uses: actions/checkout@v2
```

(continues on next page)

(continued from previous page)

```
- name: Set up Python ${ matrix.python-version }
  uses: actions/setup-python@v1
  with:
    python-version: ${ matrix.python-version }
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install numpy==${ matrix.numpy-version }
- name: Install package
  run: |
    pip install -e .
- name: Test with pytest
  run: |
    pip install pytest
    pip install pytest-cov
    pip install scipy
    pytest -v tests/ --cov=exampy/
```

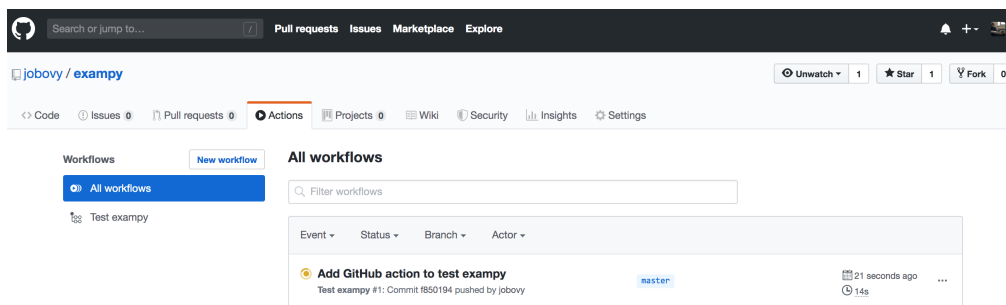
The steps in this workflow are as follows:

- **name:** sets the name for the workflow
- **on:** tells GitHub Actions when to run this workflow, with **on:** `[push]` meaning that it should run this for every push to the repository. Typically, you'll want to also run your tests upon each Pull Request, which can be achieved with **on:** `[push,pull_request]`. You can use [very complex conditions](#) of when to run your workflows.
- **jobs:** sets up the various jobs in this workflow. Every job runs in parallel, so you will typically just have a single one, **build:** here.
- In the job, the **runs\_on:** section defines the operating system, which can be versions of Ubuntu, Windows, or Mac OS X.
- The **strategy:** section allows you to define a build matrix, similar to how this is done for Travis CI and AppVeyor. Here, I create a matrix for two minor Python versions and three minor numpy versions to (partially) follow [NEP 29](#), but according to NEP 29, Python 3.8 only requires numpy v1.18 support, so I exclude the numpy versions 1.16 and 1.17 in the **exclude:** section. Note that there is an **include:** section as well, but unlike on Travis CI where this would add a new build to the matrix, for GitHub Actions, this can only *modify* a job defined in the matrix. That's why I have to exclude the builds I don't want rather than including the one I do want (Python 3.8 and numpy 1.18).
- **steps:** then lists the various steps that need to be run to perform the workflow and this is where you can use pre-defined actions as part of your workflow. For many workflows, the first step will be to clone the repository, which is done in the **uses:** `actions/checkout@v2` step. This is a step that is defined by an action [implemented by GitHub itself](#) and the entire

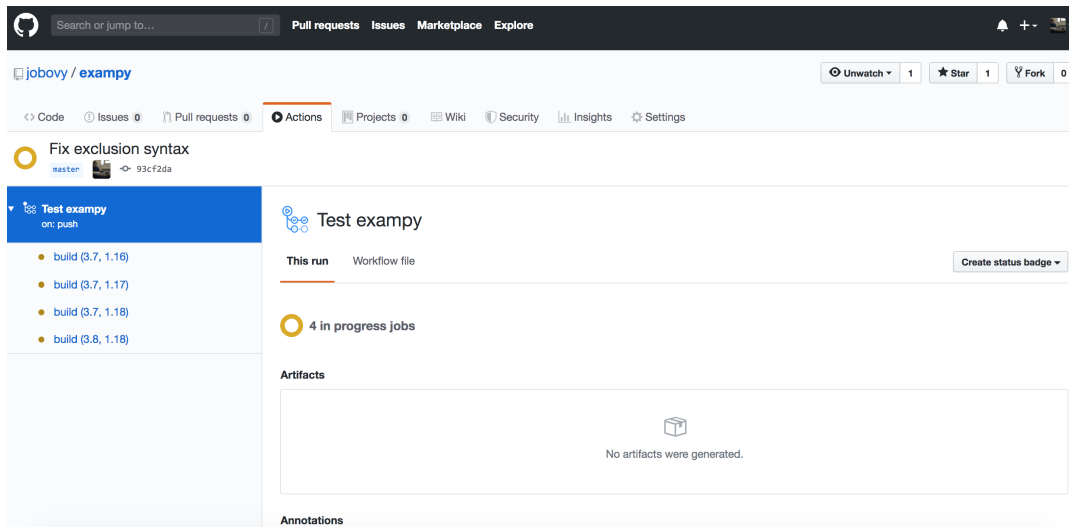
step thus simply consists of you telling the workflow to use this pre-defined action.

- Similarly, the next step uses a pre-defined action to set up a specific Python version. As you can see, we can give steps a `name:`. We also need to provide the action with the Python version that we want to set up and this is done in the `with:` section, which lists parameters for the action. In this case, we set this to the Python version corresponding to the current build in the matrix, using the `${{ matrix.python-version }}` syntax.
- Next, I install the desired `numpy` version. Rather than using a pre-defined action, this step simply runs a set of shell commands in the `run:` section. A single command could directly follow `run:`, but multiple commands as in the example here need to use `run: |` and then list the commands on the next lines. We again access a variable defined in the matrix to install the correct `numpy` version.
- The following two steps are similar, we install the package, and then run the tests. Note that we could have split the tests into two, with one step for installing the test dependencies and one to run the actual tests (similar to what we did for Travis CI and AppVeyor); with GitHub Actions, you can define as many steps as you want!

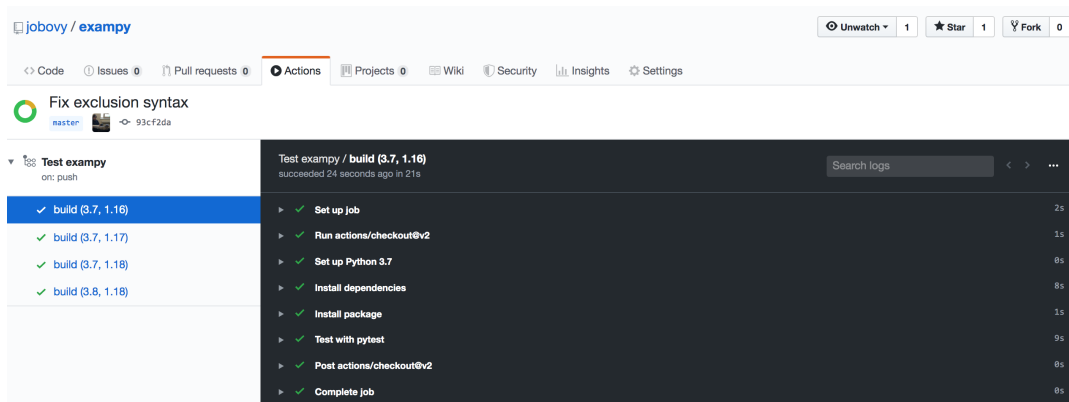
When you commit this file to your repository and navigate to the “Actions” tab again, you see a page that looks like



which shows all of the workflows that are running or have been run for your repository, currently only a single one, but eventually this would contain the entire history of runs. The workflow is labeled by its commit message. Clicking on that, we get an overview page for all of the builds in the current workflow, where as expected four builds are running for different versions of Python and `numpy`:



Clicking on an individual build, you get to a log of what happens in each of the steps, which once the build ends for one of these looks like



By clicking on the little arrows, you can get the detailed log for each step.

Going further and running the integration tests on all three major operating systems (OS) is very easy with GitHub Actions. To do this, we simply change the beginning of the `test_package.yml` file to

```
name: Test exampy

on: [push]

jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        python-version: [3.7, 3.8]
```

(continues on next page)

(continued from previous page)

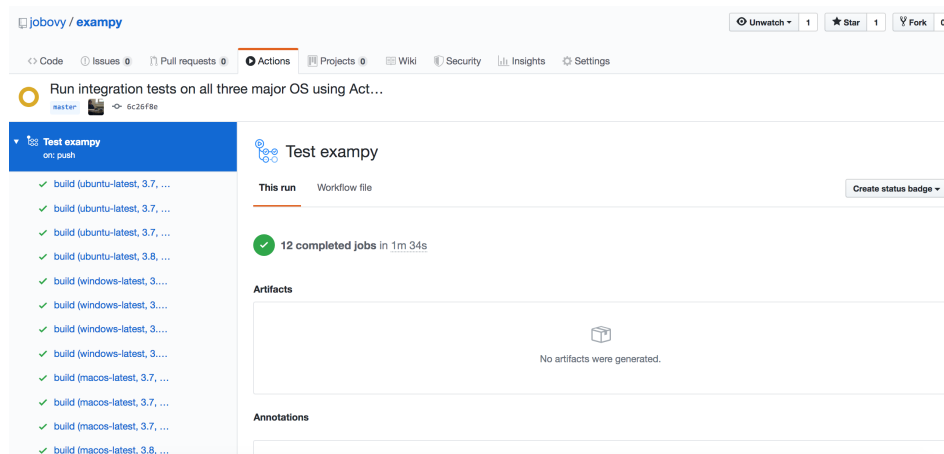
```
numpy-version: [1.16,1.17,1.18]
```

```
exclude:
```

- os: ubuntu-latest  
python-version: 3.8  
numpy-version: 1.16
- os: ubuntu-latest  
python-version: 3.8  
numpy-version: 1.17
- os: windows-latest  
python-version: 3.8  
numpy-version: 1.16
- os: windows-latest  
python-version: 3.8  
numpy-version: 1.17
- os: macos-latest  
python-version: 3.8  
numpy-version: 1.16
- os: macos-latest  
python-version: 3.8  
numpy-version: 1.17

```
steps:
```

where everything following `steps:` stays the same. As you can see, all we have changed is `runs-on:` to choose the OS defined by the matrix, `${{ matrix.os }}`, and we have added a row `os: [ubuntu-latest, windows-latest, macos-latest]` to the `matrix:` section. To exclude the unnecessary components of the matrix, we have to (tediously) exclude every single OS, because one cannot use lists in exclusions. Once you push this change, the workflow that runs looks like



If you want to use Miniconda instead to manage Python and other dependencies, you can use an action that accomplishes this, e.g., [conda-incubator/setup-miniconda](#), which you can use as a step

as, for example,

```
- uses: conda-incubator/setup-miniconda@v2
  with:
    activate-environment: test-environment
    environment-file: environment.yml
    python-version: ${{ matrix.python-version }}
```

which sets up a conda environment named `test-environment` that is defined in the `environment.yml` file (this is a standard conda file). If you want to conda install a version specified in the matrix, e.g., `numpy` in the example that we have been using, you need to do this as `conda install numpy==${{ matrix.numpy-version }}` as part of a later step's run:.

This is only a very small part of what GitHub Actions can do for you, so check out the [documentation](#) to learn more.

## 6.5 Analyzing test coverage online using Codecov

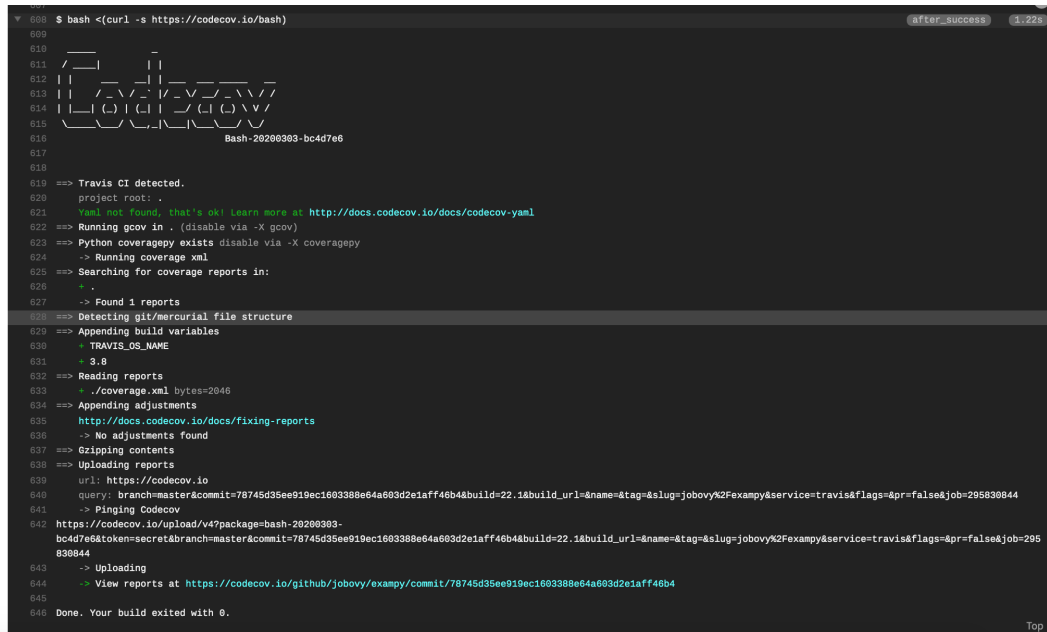
We are now able to run our build-and-test integration on various continuous-integrations services and as part of the logs of these runs we can see the test coverage of our test suite. That's already very useful, but it's easier to analyze your test coverage results if they are displayed in a nicer format, such as the HTML format that `coverage.py` can create. To make this easy, there are various free online services that will ingest your test coverage results every time you run the test suite using continuous integration and display it as a convenient online website. Moreover, these services are also able to combine test coverage results from different, independent jobs that make up your build-and-test integrations. This is useful if you break up a single test suite into multiple parallel jobs; without the ability to combine the test coverage results from the parallel jobs, it would be very difficult to know your test suite's actual coverage.

While the most popular online test-coverage tool still appears to remain [Coveralls](#), I find the alternative [Codecov](#) service far superior, in everything from getting the simplest setup to work, to making more advanced features like combining test-coverage results from different jobs or different languages work, and it has a much more useful and pleasant interface than Coveralls. I will therefore focus on Codecov here.

Codecov is yet another service that is seamlessly integrated with GitHub. To get started, navigate to <https://codecov.io/> and sign up using your GitHub account. When you login, navigate to "Repositories", click on "Add new repository", and find the GitHub repository that you want to add (here `exampy` again). Then all you have left to do is to invoke the "bash Codecov uploader" in your `.travis.yml`'s `after_success:` section (I assume here that you want to upload reports from Travis CI, but the procedure for AppVeyor or GitHub Actions would be similar) as

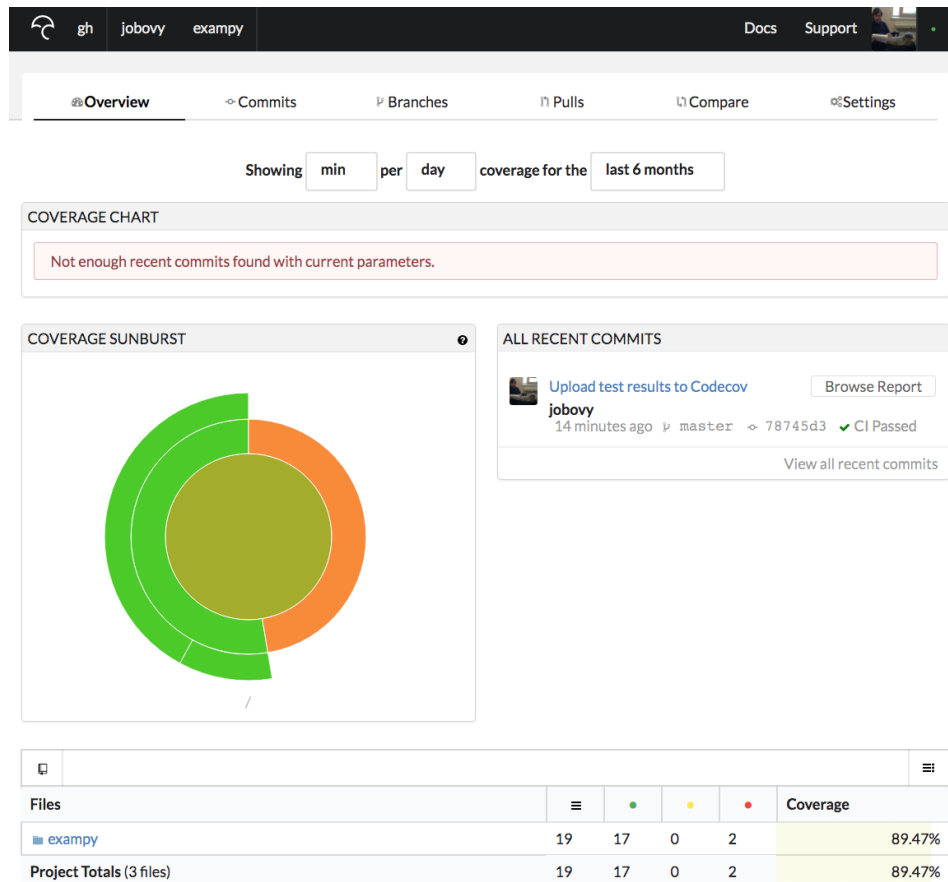
```
after_success:
- bash <(curl -s https://codecov.io/bash)
```

Once you push this update to GitHub, your tests will once again run on Travis CI, and now at the end of a successful integration run, you reports will be uploaded to your Codecov page for your repository. The final part of the Travis CI log for a job looks like (you might have to expand this, by clicking on the arrow in front of `bash <(curl -s https://codecov.io/bash)`):

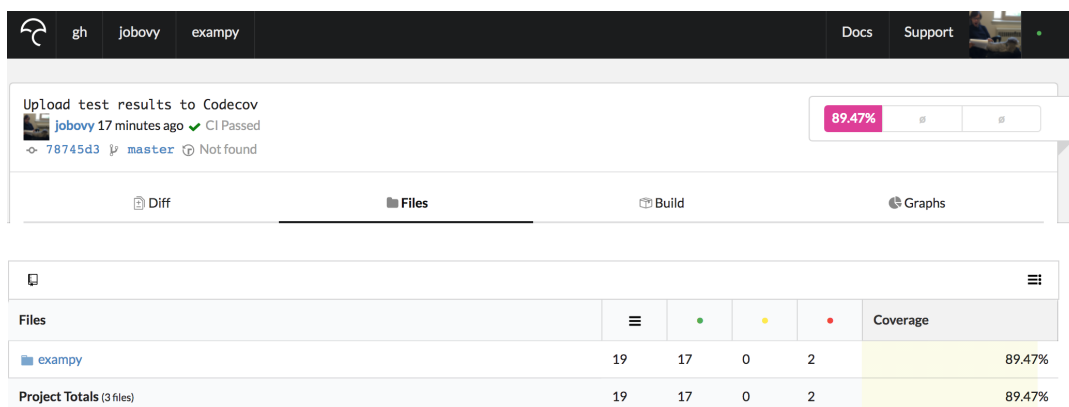


```
609 $ bash <(curl -s https://codecov.io/bash)
610
611
612
613
614
615
616 Bash-20200303-bc4d7e6
617
618
619 ==> Travis CI detected.
620 project root: .
621 xml not found. That's ok! Learn more at http://docs.codecov.io/docs/codecov-yaml
622 ==> Running gcov in . (disable via -X gcov)
623 ==> Python coveragepy exists disable via -X coveragepy
624 -> Running coverage xml
625 ==> Searching for coverage reports in:
626 + .
627 -> Found 1 reports
628 ==> Detecting git/mercurial file structure
629 ==> Appending build variables
630   TRAVIS_OS_NAME
631   3.8
632 ==> Reading reports
633   ./coverage.xml bytes=2046
634 ==> Appending adjustments
635   http://docs.codecov.io/docs/fixing-reports
636 -> No adjustments found
637 ==> Zipping contents
638 ==> Uploading reports
639 url: https://codecov.io
640 query: branch=master&commit=78745d35ee919ec1603388e64a603d2e1aff46b4&build=22.1&build_url=&name=&tag=&slug=jobovyX2Fexampy&service=travis&flags=&pr=false&job=295830844
641 -> Pingin Codecov
642 https://codecov.io/upload/v4?package=bash-20200303-
bc4d7e6&token=secret&branch=master&commit=78745d35ee919ec1603388e64a603d2e1aff46b4&build=22.1&build_url=&name=&tag=&slug=jobovyX2Fexampy&service=travis&flags=&pr=false&job=295
830844
643 -> Uploading
644 -> View reports at https://codecov.io/github/jobovy/exampy/commit/78745d35ee919ec1603388e64a603d2e1aff46b4
645
646 Done. Your build exited with 0.
```

and if we navigate to the repository's Codecov page, we now see an overview of the test coverage of all recent commits (just the one so far); for the `exampy` repository, this is <https://codecov.io/gh/jobovy/exampy>:



Clicking on the latest commit brings us to a page that shows the difference in test coverage with respect to the commit's parent (typically the previous pushed commit for the branch you are on), which is empty at first because there is nothing to compare against. Clicking on “Files” instead, you get an overview of the test coverage of all of the files in your repository, shown as a directory tree or as a simple list of files:

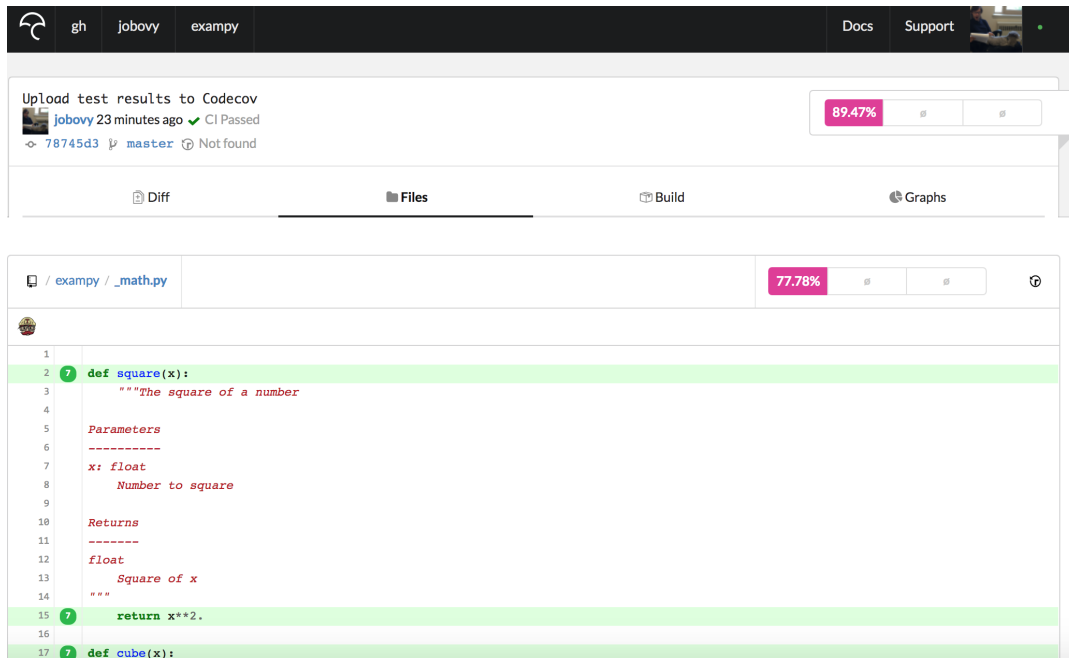


This lists the total number of lines, the total number of covered lines, or partially-covered lines (we don't typically use this), and lines that are not covered. For directories (e.g., the package directory shown above), you see a total for all of the files in that directory (so for the package directory you see the total for all files in the package; note that there appears to be an [issue](#) that still exists despite



being closed on GitHub where Codecov does not include the top-level `__init__.py` file, leading to a one line discrepancy to our results *before* (page 87)).

If we navigate to a particular file, e.g., `exampy/_math.py`, we get a page that starts as



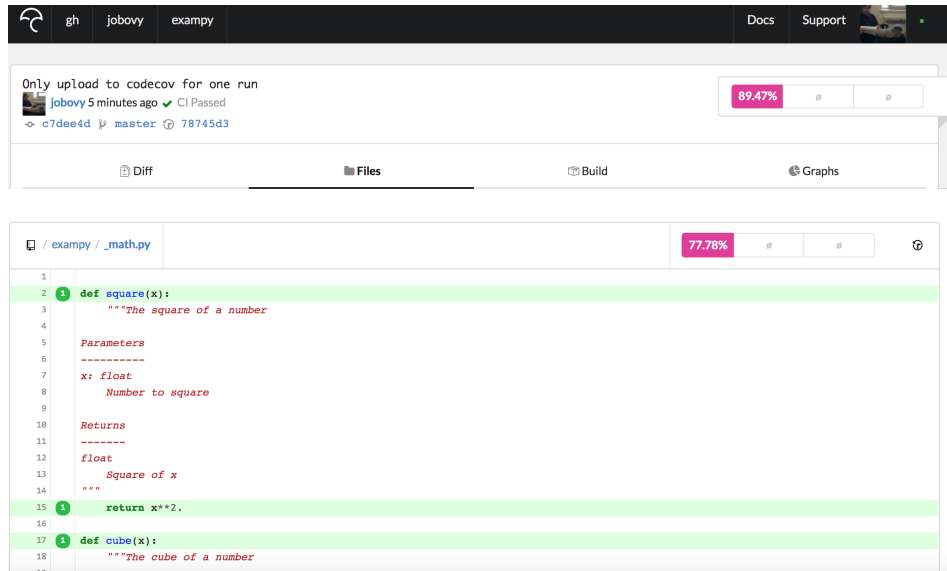
which shows the source code, with lines that are covered labeled in green and lines that are not covered indicated in red (lines that are not code lines like docstrings and comments or that are excluded by the `exclude_lines =` option in the `.coveragerc` are not labeled at all). These pages make it easy to spot which parts of your source code are not covered by the tests.

Note that the way we have added the Codecov bash uploader to the `.travis.yml` file, the reports for each run are uploaded and combined (go to the “Builds” section and you see the various Travis CI jobs that got uploaded). Because these all repeat the same tests, this uploads the same report multiple times (seven times in the `exampy` case here, leading to the number seven in front of the lines in the above image, indicating that the line was executed seven times). Typically, you will want to only upload reports from a single, unique run of your test suite. If you’ve split it into multiple pieces, you want to upload reports for all of those, and they will be automatically combined, but if you run the test suite multiple times for different setups (e.g., different Python versions as in the example here), you want to only upload reports for one. To do this, you can put a conditional statement in front of the upload command, for example

```
after_success:
  - if [[ $TRAVIS_PYTHON_VERSION == 3.8 ]] && [[ $NUMPY_VERSION == 1.18.1
    ↪]]; then bash <(curl -s https://codecov.io/bash); fi
```

which only uploads the reports for Python v3.8 / numpy v1.18.1 (you may want to define these as the environment variables `PYTHON_COVREPORTS_VERSION` and `NUMPY_COVREPORTS_VERSION`, such that they are easy to adjust when you update the versions that you run your tests for (you may use this condition multiple times). Pushing this update to the `.travis.yml` file to GitHub, the integration

suite runs again, and now only the test-coverage results from the first job (with Python v3.8 / numpy v1.18.1) get uploaded to Codecov, which you can check by going to the “Builds” section for this commit’s Codecov page. If we go to the Codecov page for `exampy/_math.py` now, we see a page that starts with



where we now see that the lines are preceded by a “1”, indicating that they were executed once. In general, these numbers tell you how often a line was executed during the test suite, and this is why you want to only upload the reports from a unique run of your test suite, such that the numbers given are the actual number of times the lines are executed without being multiplied by the number of times the test suite was run. The more often a line is executed, the more robust it probably is.

As I discussed in the [section on test coverage](#) (page 87), you can also generate test-coverage reports for parts of your code written in C (or in any other language). To upload these to Codecov in addition to the Python test-coverage results, simply run the uploader as

```
bash <(curl -s https://codecov.io/bash) -X gcov
```

and Codecov will automatically combine the Python and C reports into a single online report.

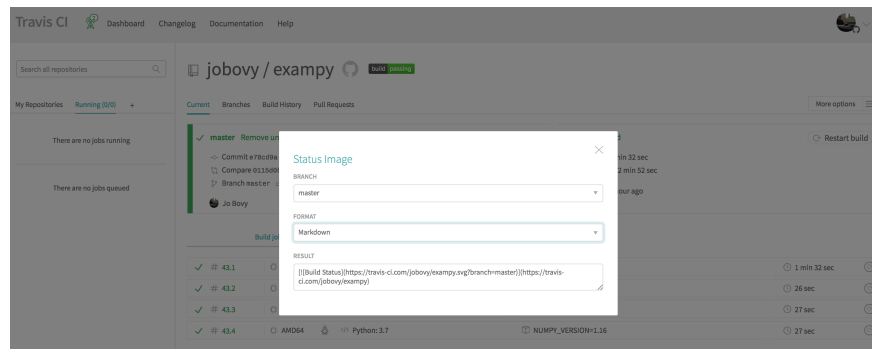
## 6.6 Status badges for your package

I have discussed how to run continuous-integration builds for your package with various different services and how to collect and display the test-coverage statistics from these builds. When you have different CI services running for your code, it can be easy to lose track of the status of each one. To help with keeping track, it’s useful to add status badges for all of the services that you are using to your package’s GitHub page to get a quick overview.

In addition to showing the status of your CI runs and other services that you use, clicking them typically also directly leads to your project’s page on the service in question, making them also an

easy way to navigate to your package’s page for various services.

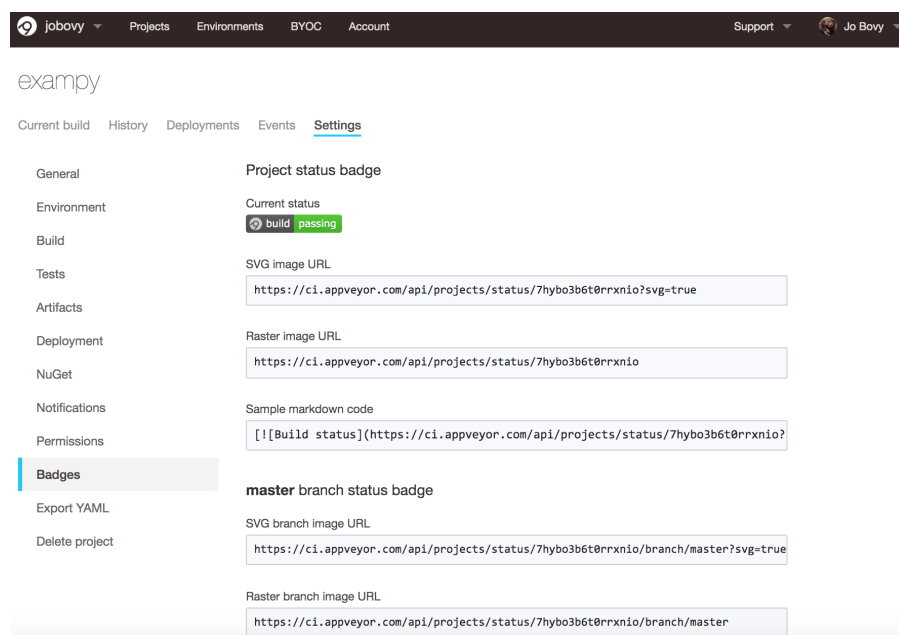
To get a status badge for your Travis CI runs, head to your package’s Travis CI page (you need to be logged in) and click on the displayed badge next to your package’s name. This brings up a dialog box that allows you to specify the branch for which you want to display the status (typically `main`) and the format of the file that you will paste the badge code into (“Markdown” if your README is in Markdown format, “RST” if it is in reStructuredText format, etc.); this dialog box looks like



Then add the resulting code snippet to your README, e.g.. for `exampy` in Markdown format, the snippet is

```
[[Build Status](https://travis-ci.com/jobovy/exampy.svg?
→branch=main)](https://travis-ci.com/jobovy/exampy)
```

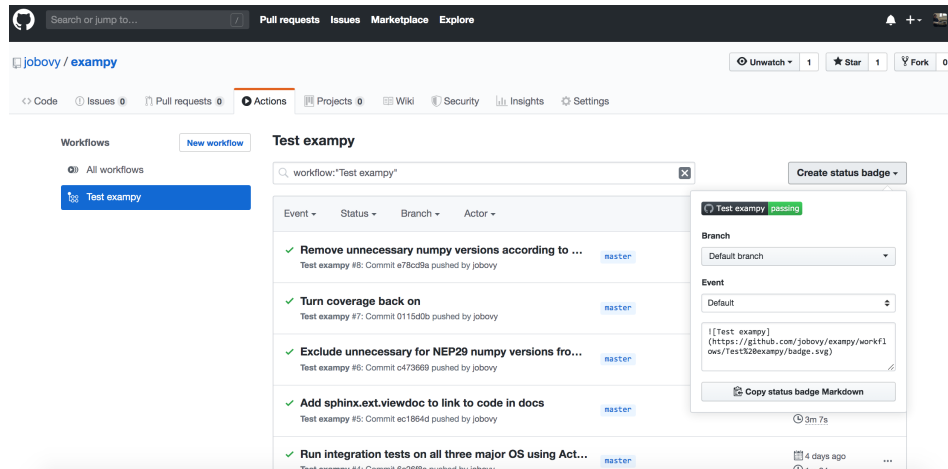
For AppVeyor, the process is similar: go to your package’s AppVeyor page, click on “Settings”, and then “Badges”, which gives you URLs for various badge images and a sample Markdown code snippet:



For `exampy`, the snippet in Markdown is

```
[![Build status](https://ci.appveyor.com/api/projects/status/7hybo3b6t0rrxnio?svg=true)](https://ci.appveyor.com/project/jobovy/exampy)
```

For GitHub Actions, navigate to the “Actions” tab on your package’s GitHub page and click on the left on the workflow for which you want to get the badge (you can get badges for all different workflows). Then on the right you will see a “Create status badge” button that when you click it brings up a similar dialog as on Travis CI:

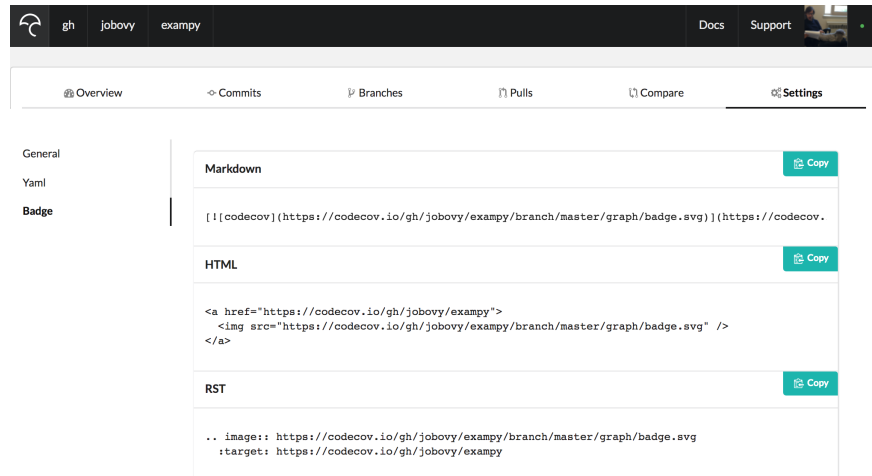


Now the Markdown snippet is

```
[![Test exampy](https://github.com/jobovy/exampy/workflows/Test%20exampy/badge.svg)]
```

which creates a badge that has the workflow name in it.

You can also create a badge for your code’s test-coverage statistics from Codecov. To get the badge, go to your package’s Codecov page (you need to be logged in), navigate to “Settings” and then “Badge”, which brings up a page with the code snippet for embedding the badge in a variety of file formats



For Markdown, the snippet now is

```
[![codecov](https://codecov.io/gh/jobovy/exampy/branch/main/graph/badge.svg)](https://codecov.io/gh/jobovy/exampy)
```

which brings up a badge that includes the fraction of your package’s statements that are covered by the test suite, colored according to how high it is (try to get it green!).

You can also get a badge that shows the status of your documentation’s build on [readthedocs.io](https://readthedocs.org/), which we discussed in [Chapter 4](#) (page 66). Again, navigate to the admin page for your package’s [readthedocs.io](https://readthedocs.org/) setup where you will see a badge on the right-hand side; clicking on the “i” symbol next to it brings up a dialog box with various badge options:

The screenshot shows the Read the Docs interface for a project named 'exampy'. At the top, there's a dark header with the 'Read the Docs' logo and a user profile 'bovy'. Below the header, the project name 'exampy' is displayed with a 'View Docs' button. A navigation bar contains links for Overview, Downloads, Search, Builds, Versions, and Admin. The main content area is divided into two columns. The left column shows the 'reStructuredText' and 'Markdown' source code for a documentation status badge. The right column displays project metadata: Repository (https://github.com/jobovy/exampy.git), Project Slug (exampy), Last Built (1 hour, 37 minutes ago, passed), Maintainers (with a profile picture), Badge (docs passing), Tags (Project has no tags), and Project Privacy Level (Public).

For example, the Markdown format is

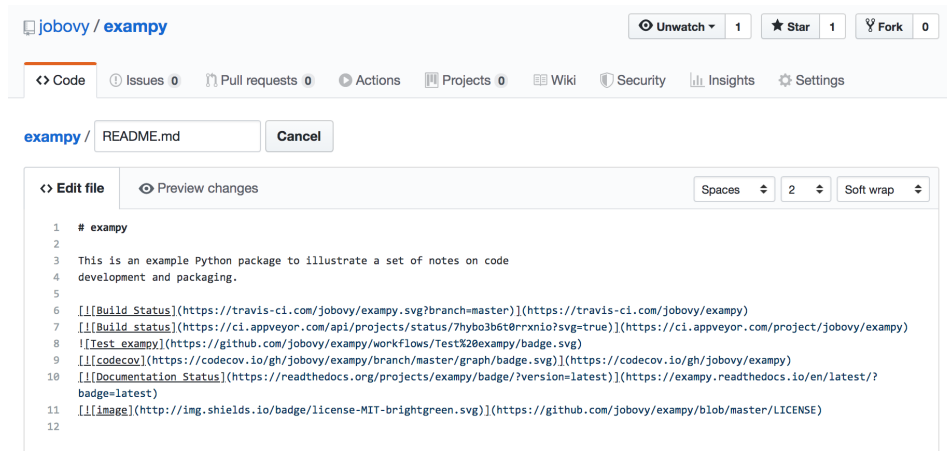
```
[![Documentation Status](https://readthedocs.org/projects/exampy/badge/?
→version=latest)](https://exampy.readthedocs.io/en/latest/?badge=latest)
```

You can create more badges to show off other aspects of your package. You can create badges by hand using the [shields.io](https://shields.io/) service, which has a URL-based interface to request badges with different text, color, shape, etc. For example, you could show off your package's license by linking it through a badge with

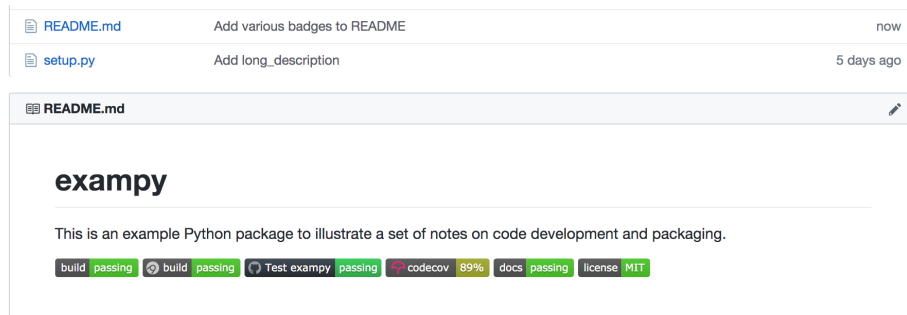
```
[![image](http://img.shields.io/badge/license-MIT-brightgreen.svg)](https:
→//github.com/jobovy/exampy/blob/main/LICENSE)
```

Any text needs to be encoded as a URL (search for “encode URL” to get a working online service to do this for you). Note that [shields.io](https://shields.io/) can also create badges for certain services that are automatically created rather than hand-crafted. For example, we will use this to create a badge for the PyPI release of our package in the next chapter.

Adding all of these badges to the README, for exampy like



we get something that looks like:



(note that this is a static image, while the embedded badges above are “live”, so if exampy has changed since writing this, they make give different statuses).





## RELEASING YOUR PACKAGE

Once you have *implemented a bunch of functions, classes, and methods* (page 8) in your package and made sure your package can be *installed in the standard way* (page 14), once you have set up *version control with git* (page 17) and you are using *GitHub to develop your code online* (page 25), once you have written *documentation* (page 31) explaining how to *install and use your code* (page 44), you have created an *API for all of the functionality in your package* (page 58) and you are *hosting your package’s documentation online* (page 66), once you have written a *comprehensive test suite* (page 71) that checks that *your code works as you expect* (page 75), this test suite *covers a large fraction of your package* (page 87) and you are using *continuous integration to run your test suite* (page 95) automatically upon every change that you make to your code, you are ready to release a first version of your package to the world!

Many scientists still release their code by creating a source distribution (a “tarball”) and linking to it on their website, but this fails to take advantage of the large amount of support in the Python community (and beyond) to make releasing different versions of your code easy and allowing your code to be installed in standard ways on different platforms with minimal headaches for the users of your package. The primary venue for releasing Python packages is the Python Package Index (PyPI) and it is releasing your package on PyPI that I focus on in this chapter. An additional popular method for releasing your code, especially if it contains compiled code and/or dependencies that are difficult to install, is to make your package installable by Anaconda’s `conda install` command. The most common way to do this for scientific packages is to use `conda-forge`, but describing how to get your package onto `conda-forge` is beyond the scope of these notes.

### 7.1 Versioning your code

Before you release a first version of your code, you should decide on how you want to label different versions of your code, because once you release the first version, you’ll want to start work on the next version! When we first created the *setup.py file* (page 11), we included a version string, but I will now discuss versioning in a little more detail.

Your code’s version could be any unique string that states the version (e.g., “v1”), but to make your code’s version easy to interpret for a wide variety of users and compatible with standard

Python functions to process version numbers, your code should follow the [standard Python versioning scheme](#). Python version strings should follow the format

```
[N! ]N(.N)*[{a|b|rc}N][.postN][.devN]
```

which I will briefly unpack here. We'll ignore the `[N! ]` part, the [version epoch](#), which is only used when packages change the way they version and which the author of these notes has never seen in practice.

For released versions, you only have to pay attention to the

```
N(.N)*
```

part of the version string. This states that any released version string can be any string of integers separated by periods (e.g., “0.1.0”). While arbitrary lengths are in principle allowed, most packages use either two integers separated by a period (e.g., “1.0”) or three integers, (e.g., “2.1.3”). Upon a release, either the last integer is increased by 1 or one of the earlier integers in the sequence is increased and all following integers are set to 0. So, for example,

- version “0.1” could be increased to either “0.2” or to “1.0” in the next release,
- version “1.3.2” could be increased to either of “2.0.0”, “1.4.0”, or “1.3.3”.

Note that the version string does *not* contain “v”, that is, the version is “1.0.0”, not “v1.0.0” (that can be easy to forget!).

Standard Python usage of version numbers is *not* to use semantic versioning as advocated by [semver](#), but to use a more loosely defined structure of “major.minor” or “major.minor.patch” where the “major” version is increased only when a package’s structure changes dramatically (causing much backwards incompatibility, e.g., Python 2 to Python 3) or when the package matures significantly to deserve a “1.x.x” version (having had “0.x.x” until then; e.g., [scipy’s relatively recent version](#) “1.0.0”, 16 years after the first release). Therefore, for most Python packages, major version changes are rare (but some packages update the major version more often, e.g., [astropy](#)). Typical releases will therefore increase the minor version number, e.g., going from version “1.1” to “1.2”. The third integer, called “patch” above, indicates small changes such as bug fixes or updates because of dependency changes. Such patch releases would typically be made from a branch of the code created at the latest minor version release to track bug fixes and other small changes, while development of the next minor version continues in the main development branch.

The remainder of the possible version string

```
[{a|b|rc}N][.postN][.devN]
```

deals with pre- and post-release updates to a package. The `[{a|b|rc}N]` part indicates pre-release versions, with `a` indicating an alpha release, `b` a beta release, and `rc` a “release candidate”. The `N` that can optionally follow each of these indicates the `N`-th version (e.g., “a2” would be the second alpha release). Unless you are developing a package with many users and/or that provides essential infrastructure where each release needs to be thoroughly tested, you will typically not make use of

these versions (but you could if you want). The `[.postN]` is reserved for changes to your code that happen “post release”. These are changes, not to the source of your code itself, but to the distribution of your code. You can for example create a `.post1` release if you forgot to include the License file in your release; if you simply want to add the file, you should create a new release (because a release, once made, is final), but you are not changing any of the code. Finally, the `[.devN]` is used to indicate development versions of your code. For example, “1.1.dev” is the development version that will lead to release “1.1” eventually. Note that there are a bunch of [alternative syntaxes](#) allowed by the standard (for example, you can use “alpha” instead of “a”), but for new versions it’s best to follow the actual standard discussed here.

One of the advantages of all packages using this standard versioning scheme is that this can be easily parsed by automated parsers, to check, for example, which version of a dependency users have installed on their systems. One way to do this is to use `setuptools`’ `pkg_resources`, which has a function `parse_version` (documented [here](#)), that returns an object that represents the version and that can be compared to other such objects. For example, we can check that the installed version of `scipy` is at least later than version “0.19” by doing

```
from pkg_resources import parse_version
import scipy
print(parse_version(scipy.__version__) >= parse_version('0.19'))
```

`parse_version` correctly deals with development, pre-release, and post-release versions.

You should include your package’s version in the `setup.py` file and define the `__version__` attribute in your package’s top-level `__init__.py` file (`exampy/__init__.py` in the example package); you will likely also want to define it in the documentation’s `source/conf.py`. Because your package’s version then appears in multiple places in your code, you need to remember to update it everywhere when you increment the version number. One way to do this is to make it a part of your release checklist (discussed below), you can also make use of automated tools like [bumpversion](#), which can be configured to automatically update the version string in multiple files whenever you invoke `bumpversion`.

## 7.2 Preparing for your package’s release

In preparation for your package’s initial and each subsequent release, you’ll want to add or update some files in your repository. It’s useful to list files that need to be updated every release as part of a *release checklist* (which you can add as a `RELEASE_CHECKLIST.md` file in your repository as well) that explicitly lists all of the steps involved in releasing each version of your package, from preparing the release, over uploading your package to PyPI, to setting up the next development version.

Some of the files that you will want to add or update for every release are:

- A `HISTORY.txt` or `HISTORY.md` (for formatted GitHub content) file that lists major changes and additions to your package since the previous release. For the first release, this file can

simply state that this is the first release, but at each subsequent release you should update this file with all of the major changes since the previous release. It's easiest to keep this file up to date by adding to it as you develop each version, rather than writing it at the end. If you find yourself having to write it at the end, go through all Issues and Pull Requests to find major changes, or look through the `git` log up to the last release (although if you are making the recommended frequent commits, this will be a long and detailed log). One way to help yourself keep track of major changes is to submit Pull Requests for all of these even if you are making the changes yourself and you could just merge them directly.

If your documentation includes a page summarizing major changes, you'll want to update that one as well.

- Update the version number to that of the release you are going to make, typically at least in the `setup.py` file and the top level `__init__.py` file. You should have been using a version “x.y.z.dev” while developing version “x.y.z” and in that case this is as simple as removing the “.dev” part of the version string.
- To obtain fine-grained control over which files are included in the source distribution, you need to use a `MANIFEST.in` file. Many files are included by default, most importantly: all Python files included in your package (through the `packages` keyword of the `setuptools.setup` command in your `setup.py` file), the `setup.py` file itself, any files specified in `setuptools.setup`'s `package_data` keyword, scripts included in the setup, C source files for any C extensions specified in `setuptools.setup`, various `README` formats (without extension, or with `.txt`, `.rst`, or `.md` extension), and the `MANIFEST.in` file itself. Test files following `test/test*.py` are also included (not `tests/test*.py`). The `MANIFEST.in` file then gives you the option to include extra files or to exclude files from the standard list. To include a file, add a line `include FILE` in `MANIFEST.in` where `FILE` is the path to the file, which can include standard wildcards. To exclude a file, add a line `exclude FILE` (e.g., to exclude the tests, add `exclude test/test*.py`). One file you'll want to include is the `LICENSE` file. If you have other useful files in your repository, e.g., `INSTALL.md` with installation instructions (typically given in the `README` file) or `AUTHORS.md` with contributors to your code, you can also include them in the `MANIFEST.in` file. If your package includes a C extension, you'll have to include the header files, because they will not by default be copied to the source distribution.

As an example, for the `exampy` example package, I created the following `MANIFEST.in`

```
include LICENSE README.md
exclude tests/test*.py
```

to include the `LICENSE` file, to make sure to include the `README.md` file (it should be included anyway, but it does not hurt to make sure), and to exclude the tests.

If you have added files to your package since the last release, you might have to add some of them to `MANIFEST.in` to make sure that they are included in the source distribution. You can get a list of all files added since your last release by running

```
git diff --name-status PREV_RELEASE_HASH | grep ^A
```

where `PREV_RELEASE_HASH` is the `git` hash corresponding to the previous release.

Once you're happy with all of the necessary additions and changes, you should make sure to have commit all of the changes to your repository and push them to GitHub such that your automated tests can run one last time to check that all is well.

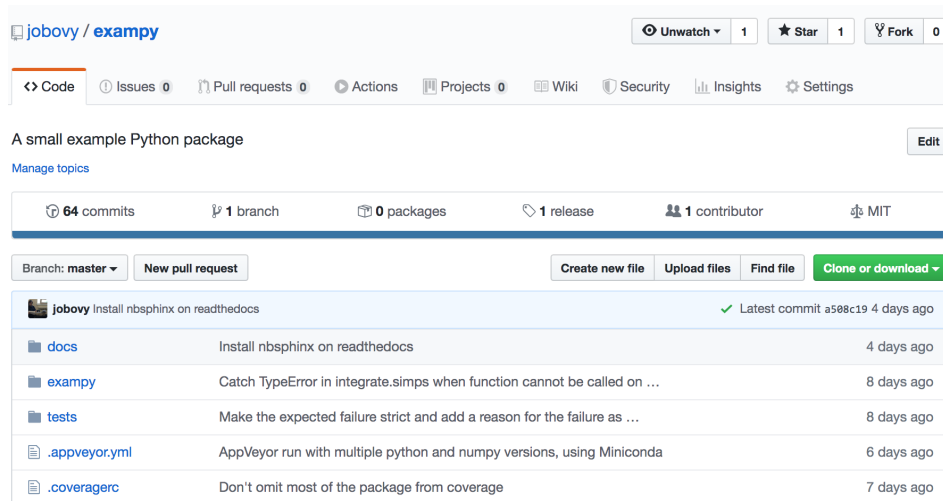
Provided that your tests pass, you can then move on to the next step, which is tagging the next release and creating it on GitHub. To create a tag, simply run

```
git tag x.y.z
```

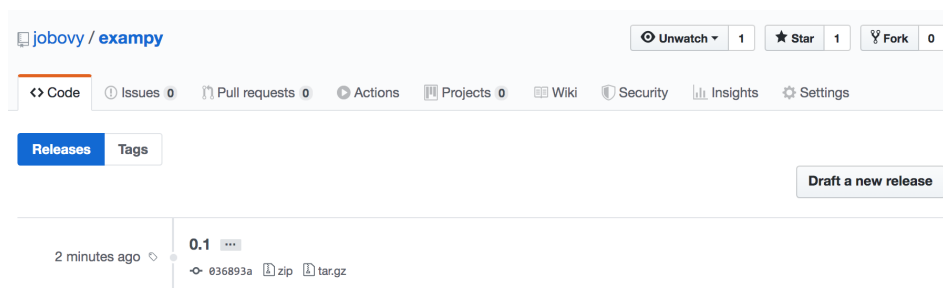
where `x.y.z` is the version of the release. Push this to GitHub with

```
git push --tags
```

Then you can go to your package's GitHub page and navigate to the "Releases" tab, in the row above the directory structure of your repository:



When you have pushed the tag, you will see that GitHub has created a release, that currently just looks like a stub:



You can create the actual release by clicking on this, clicking on "Edit tag", and giving the release

a title and short description (as usual, using Markdown formatting), and hitting “Publish release” on the page that looks like this:

jobovy / exampy

Unwatch 1 Star 1 Fork 0

<> Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings

Releases Tags

0.1

✓ Existing tag

Version 0.1

Write Preview

First release of the “exampy” package, a very simple example Python package that accompanies a set of notes on how to create a scientific Python package.

Attach files by dragging & dropping, selecting or pasting them.

**Tagging suggestions**  
It's common practice to prefix your version names with the letter v. Some good tag names might be v1.0 or v2.3.4.  
  
If the tag isn't meant for production use, add a pre-release version after the version name. Some good pre-release versions might be v0.2-alpha or v5.9-beta.3.

**Semantic versioning**  
If you're new to releasing software, we highly recommend reading about [semantic versioning](#).

Then you have created the release of your code on GitHub!

## 7.3 Uploading your package to the Python Package Index (PyPI)

Next, we will release our package to the Python Package Index PyPI. To get started, navigate to PyPI at <https://pypi.org/> to [register for an account](#); we'll use TestPyPI to test that our release looks okay and works as expected, so also navigate to TestPyPI at <https://test.pypi.org/> to [register for an account on TestPyPI](#) (TestPyPI is entirely separate from PyPI, so your PyPI account is separate from your TestPyPI account). Interacting with PyPI and TestPyPI is done, of course, using a Python package called `twine`, which you can install with

```
pip install twine
```

At first, we will simply package the source of our package and upload it to PyPI as a first release. To create a source distribution, do

```
python setup.py sdist
```

in your package's top-level directory. To make sure you aren't accidentally including files in your development version that are not part of the package, it is useful to do this from a fresh clone of your repository, checking out the tag with

```
git checkout x.y.z
```

where `x.y.z` is the version of the tag that you created above. Running the `python setup.py sdist` command creates a source distribution for the current released version in a `dist/` directory, with a filename of something like `dist/PACKAGENAME-VERSION.tar.gz`, e.g., `dist/exampy-0.1.tar.gz` for the first release of the example `exampy` package. If this is not your first release, you'll want to remove old releases from the `dist/` directory before continuing (but you should be working with a fresh clone, so an empty `dist/` directory).

The first thing you want to check is that the `long_description` that you have specified in your `setup.py`'s `setuptools.setup` command can be correctly displayed by PyPI; to check this, run

```
twine check dist/*
```

which performs this check for all source distributions in the `dist/` directory (you can also manually specify the latest one as the filename). It's useful to run this check *before* creating the final release, so you can fix any issues before tagging the release. Add it to your release checklist for the next release!

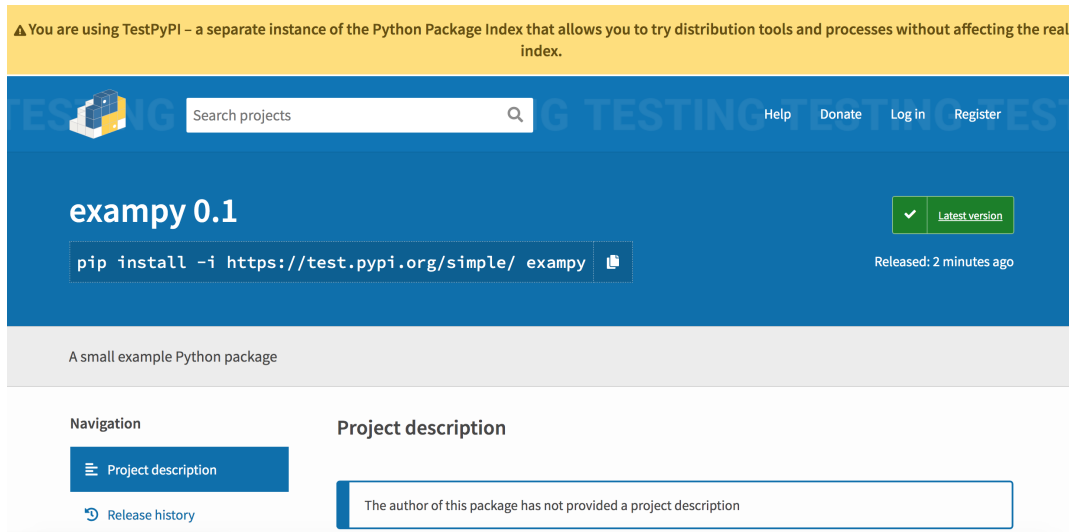
Next, upload the source distribution to TestPyPI. TestPyPI is a clone of PyPI and you can therefore check that your package release upload and the way it appears on the website look okay before publishing the final release. Once you upload a version to PyPI you can no longer easily change it, so for every release you should check with TestPyPI to make sure you are not making any mistakes (e.g., formatting errors in the `long_description` that will form the webpage for your package on PyPI. You can upload your source distribution by doing

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

which uploads every release file in the `dist/` directory; currently this is only the source distribution, but when you build binary distributions (as I will discuss below), those would also be uploaded. You'll be prompted for your username and password and if all goes well you should see the following output (for the example package's first release)

```
Enter your username: USERNAME
Enter your password:
Uploading distributions to https://test.pypi.org/legacy/
Uploading exampy-0.1.tar.gz
100%| 5.46k/5.46k [00:01<00:00, 2.85kB/s]
```

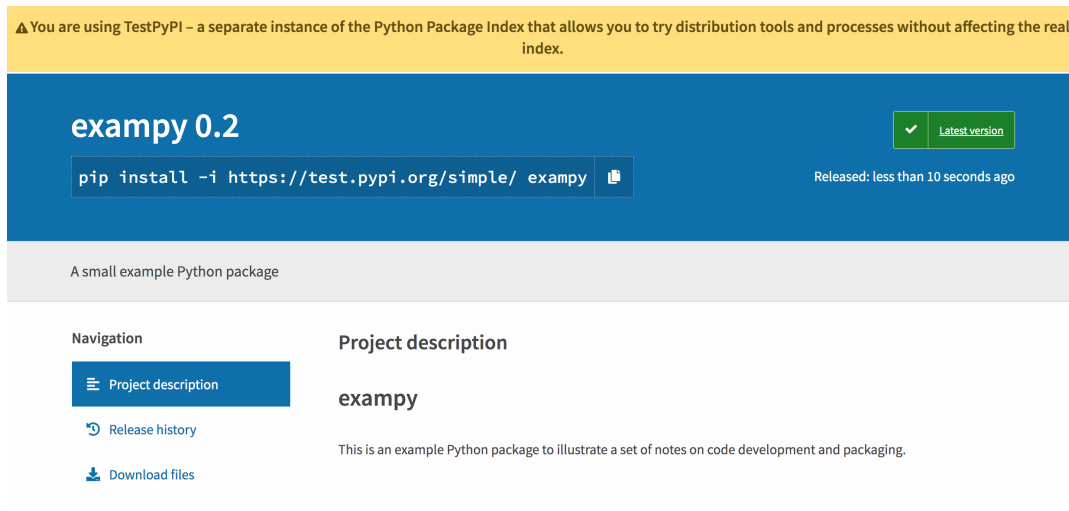
Then navigate to your project's page on TestPyPI (e.g., <https://test.pypi.org/project/exampy/>) and check that all looks well. You'll want to check that (a) your `long_description`, which is typically the `README.md` or `README.rst` that you have on GitHub as I discussed in [Chapter 2](#) (page 11), is rendered correctly as the webpage of your release, and (b) that the source distribution was properly uploaded by checking the "Download files" tab. For the first release of the `exampy` package, this test page looks like



which immediately shows a problem with this release: I had not provided a `long_description`, so there is no web page! The `twine check dist/*` command above would have already told us that, but now it can no longer be ignored! To fix this, I create a new release of `exampy` that includes the `long_description` keyword in its setup (this requires a full new release, because it changes a file, `setup.py`, that is part of the source and its installation). After (a) removing the previous release from `dist/`, (b) creating the new source release using `python setup.py sdist`, and (c) running `twine check dist/*` to check that it now passes, we run the upload command again

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

and now the TestPyPI page looks like



This now looks as we expect it to look, similar to the GitHub page. TestPyPI is a fully functional Python package index, so you can install with `pip` from TestPyPI using, e.g.,

```
pip install -i https://test.pypi.org/simple/ exampy
```

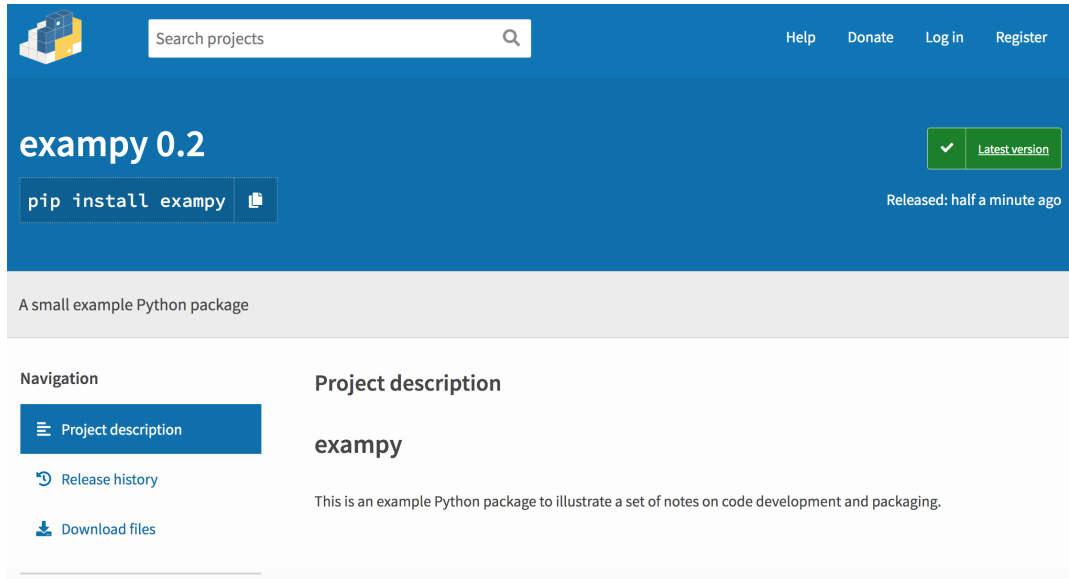


which is useful to check your package installs without a hitch using `pip`.

Once you're happy with the way your package release looks on TestPyPI, it's time to upload your release to PyPI itself. This you do with

```
twine upload dist/*
```

This works the same way as uploading to TestPyPI. Once the upload is finished, you can navigate to your package's PyPI site, e.g., <https://pypi.org/project/exampy/>, which looks something like



Once your package is on PyPI, you may want to add a *badge* (page 122) to your package's README on GitHub that shows the latest PyPI release and links to it. This can be done with the snippet:

```
[![image](http://img.shields.io/pypi/v/exampy.svg)](https://pypi.python.org/pypi/exampy/)
```

## 7.4 Building and adding binary distributions (“wheels”) to your PyPI release

So far I have only discussed how to release your package's source code, which means that users who install your package have to build the code themselves from the source. While this is relatively straightforward for pure Python packages, if your package is large or if it contains compiled C extensions, the build process can be complex, prone to errors, and long. For example, in *Chapter 6* (page 95) we saw that having to build `numpy` from source as part of the Travis CI continuous-integration procedure for the `exampy` example package made the run take substantially longer. To make the installation of your code easier, you should create binary distributions called “wheels” of your package and upload these to PyPI. When a user then asks to install your package using `pip` and a wheel is available for their Python version and platform, the wheel will be installed instead

of building the package from source. If your code contains extensions that need to be compiled and you want to support Windows natively, creating pre-built binary releases of your package is essential, because many Windows users will not have a compiler installed.

The type and number of binary wheels that you can create for your package depends on the code that is included in your package. If your package is a pure Python package (that is, it does not contain compiled extensions, just \*.py files), then you can create a single wheel that can be used on any platform (e.g., Windows, Mac OS X, or Linux). If your package contains compiled extensions, you have to create wheels for each platform *and* for each minor Python version that you want to support separately (because the Python C code is specific to each minor version); you therefore have a lot more work then!

To build wheels, first you need to install the (wait for it) `wheel` package with

```
pip install wheel
```

Then you can create a binary wheel by running

```
python setup.py bdist_wheel
```

This command will automatically determine whether to build a platform-agnostic wheel if your package is pure Python or to build a platform/Python-version specific wheel if your package contains compiled code. Build wheels are added to the `dist/` directory, which is the reason why the `twine upload` commands above specified `dist/*`; this way both the source distribution and any built wheels are uploaded simultaneously. The example `exampy` package is pure Python, so when you run the command a wheel

```
exampy-0.2-py3-none-any.whl
```

is created. The name of the wheel has the format `PACKAGENAME-PACKAGEVERSION-PYTHONVERSION-PYTHONCVERSION-PLATFORM.whl`. Because the package is pure Python, the Python C version that you are using is irrelevant and is therefore set to `none` and the platform is arbitrary, thus, `any`. Because the only major Python version that is currently supported is Python 3, you only need to build a wheel for Python 3; this can be created using any minor Python 3.x version (note that if your package requires features only available in a later 3.x version, the wheel will build and be installable for earlier versions, but the code will likely fail at runtime). When Python 2 was still supported (only a few months ago at the time of writing), you could also create a separate wheel with Python 2 or, if your package directly supported both Python 2 and 3, you could build a universal wheel with `python setup.py bdist_wheel --universal`, which would work for both major Python versions and would have the name `exampy-0.2-py2.py3-none-any.whl` for the example package. However, with Python 2 no longer being supported, there is no need for new packages to create Python 2 wheels (unless they really think users who cannot upgrade want to use it).

If your package includes compiled code, then that code will be compiled as part of the `python setup.py bdist_wheel` execution and a compiled, built wheel specific to the minor Python version that you use to run the command and the platform will be created. For example, suppose the

exampy package contained a compiled C extension and we ran `python setup.py bdist_wheel` using Python 3.7 on a Mac, we would create a wheel with the name

```
exampy-0.2-cp37-cp37m-macosx_10_9_x86_64.whl
```

where now the `PYTHONVERSION` in `PACKAGENAME-PACKAGEVERSION-PYTHONVERSION-PYTHONCVERSION-PLATFORM.whl` is `cp37`, indicating “CPython 3.7”, since there are other implementations of Python such as [PyPy](#) (not to be confused with PyPI!). The Python C API version is also CPython 3.7, hence the second `cp37`, and the platform is the specific Mac version that you ran the command on, in this case Mac OS X 10.9, 64 bit (the `x86_64` part). You can create multiple wheels for different Python versions by running `python setup.py bdist_wheel` with different versions of Python (e.g., using different conda environments or different virtualenv environments). Each of these would be added to the `dist/` directory and all would be uploaded to PyPI using `twine upload dist/*`.

When you create a wheel on a Linux operating system, the wheel’s platform will be something like `linux_x86_64`, which does not specify the Linux distribution that you are using. For this reason, PyPI does not accept such Linux wheels, it only accepts `manylinux1` wheels built on Linux. These are wheels that are defined in [Python Enhancement Proposal 513](#) that are only allowed to link to a small subset of C libraries, such that they can be easily installed on many different Linux distributions. Unless you really care about making your code easy to install on Linux, I would recommend not bothering creating such `manylinux1` wheels (Linux users are typically quite sophisticated and should therefore be able to work with a compiler and install dependencies; the only reason to create a binary wheel would be to speed up continuous integration run for packages that have to install your package if building your package from source is slow).

If you need to create multiple compiled wheels for different Python versions and for both Mac OS X and Windows platforms, you can either use a Mac or Windows computer that you have access to, create all different wheels, and upload them all to PyPI (you can just run

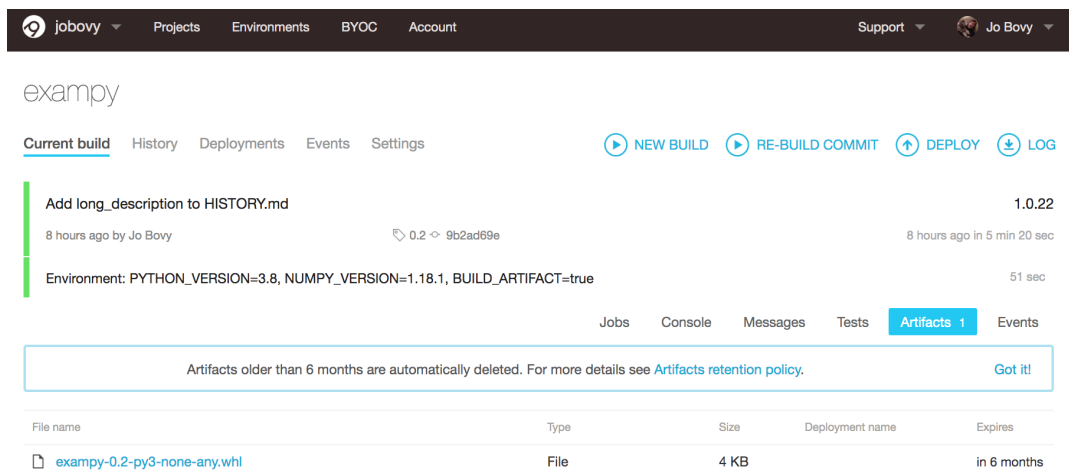
```
twine upload dist/* --skip-existing
```

on multiple machines, and files will just be added), or you can use external services to build all of the wheels for you, collect them, and upload them to PyPI. Using external services is good, because it makes sure that the build environment is clean and it means that you don’t have to use multiple machines. Because the most important compiled wheel to create is the Windows one, I will describe how to use AppVeyor to do this, but you could use Travis CI to do the same for Mac OS X wheels.


The way you can create wheels using AppVeyor is to create them at the end of your [AppVeyor integration runs](#) (page 108) and add them as an “Artifact”, which can be downloaded from the AppVeyor page for your run. Thus, if you haven’t already set up continuous integration using AppVeyor, you need to do that first. There are multiple ways to create artifacts on AppVeyor, but one flexible way is to use the command-line command `Push-AppveyorArtifact`. Typically, you would only create a wheel if your integrations tests pass and to do this, you should add the following section to your `.appveyor.yml` file

```
after_test:
- ps: |
    pip install wheel
    python setup.py bdist_wheel
    Get-ChildItem dist\*.whl | % { Push-AppveyorArtifact $_.FullName -
    ↪FileName $_.Name }
```

This simply installs the `wheel` package, builds the wheel, and then finds the wheel in `dist\` (the Windows path!) and pushes it as an artifact. To find this artifact, navigate to your package's AppVeyor web page, go to the build and the job that creates the artifact (each job will by default create an artifact), and click on the “Artifacts” tab. For the example package, the page that you get to looks like



The screenshot shows the AppVeyor web interface for a project named 'exampy'. The top navigation bar includes 'jobovy', 'Projects', 'Environments', 'BYOC', 'Account', 'Support', and a user profile 'Jo Bovy'. Below the project name, there are tabs for 'Current build', 'History', 'Deployments', 'Events', and 'Settings'. The 'Current build' tab is active, showing a build summary for version 1.0.22, created 8 hours ago by Jo Bovy. The build environment is specified as PYTHON\_VERSION=3.8, NUMPY\_VERSION=1.18.1, and BUILD\_ARTIFACT=true. The 'Artifacts' tab is selected, displaying a table of artifacts. A message states: 'Artifacts older than 6 months are automatically deleted. For more details see [Artifacts retention policy](#). Got it!'

File name	Type	Size	Deployment name	Expires
 <a href="#">exampy-0.2-py3-none-any.whl</a>	File	4 KB		in 6 months

You see that the wheel is available as an artifact and you can download it there. Because `exampy` is pure Python, this is the simple pure-Python wheel that we discussed above, but if the package contains compiled code, the wheel would be specific to the Python version that it was built with and to the Windows operating system.

To create Windows wheels for different Python minor versions, you can use a matrix AppVeyor build with multiple Python versions as discussed in [Chapter 6](#) (page 108). If you run multiple jobs for the same Python version (e.g., like in the example in Chapter 6, where I run jobs with three different `numpy` versions for each Python version), you may only want to create artifacts for one of the jobs for each Python version (not that you couldn't create them for each job, since they are specific to a job, but it would be somewhat wasteful). To do this, you could add an environment variable `BUILD_ARTIFACT` that is either `"true"` or `"false"`, depending on whether you want the job to create the artifact, and then use

```
after_test:
- ps: |
    if ($env:BUILD_ARTIFACT -eq "true") {
        pip install wheel
```

(continues on next page)

(continued from previous page)

```
python setup.py bdist_wheel
Get-ChildItem dist\*.whl | % { Push-AppveyorArtifact $_.FullName -
→FileName $_.Name }
}
```

To add these AppVeyor built artifacts to your release, you can just download the one(s) created for the AppVeyor run corresponding to the `git` tag of your release into your local `dist/` folder and then upload them from your machine using `twine upload` when you upload the source distribution as well. You can set things up such that wheels are automatically uploaded to PyPI from AppVeyor itself, but you probably want to finely control exactly what gets uploaded to your official release on PyPI, which is difficult when the upload happens automatically. Unless you release so often that you need the automation, it's easiest to just download the wheels and upload them to PyPI yourself.

## 7.5 Starting the development of your next version

Once you have created your release and uploaded all files to PyPI, you will want to set up your package for the development of your next version. The following is a list of some thing you may want to do now to make this easier:

- Update the version number everywhere in your package to the next version (including the `.dev` end to indicate that this version is currently in development). Remember, the version string likely appears at least in your `setup.py` file, the top-level `__init__.py`, and your documentations `source/conf.py` file.
- If you want to keep open the possibility to patch small bugs in the currently-released version, while developing larger changes in the main development branch of your repository (typically `main`), you should create a maintenance branch, called something like `maintenance/a.b.x`, where `a.b` is the minor version you just released; typically, you'll only want to do this for every increase in `b` in this version, so then you create a maintenance branch `maintenance/a.b.x` (e.g., `maintenance/1.3.x`) to be able to fix bugs in that branch and use it to make patch releases in the `a.b.x` series (e.g., `1.3.1` would fix minor bugs in `1.3.0`).
- Add a new section at the top of your history file (e.g., `HISTORY.md`) for the next version where you can start recording major updates to the code for the next release.

And now you are ready to keep developing your package for many happy releases to come!