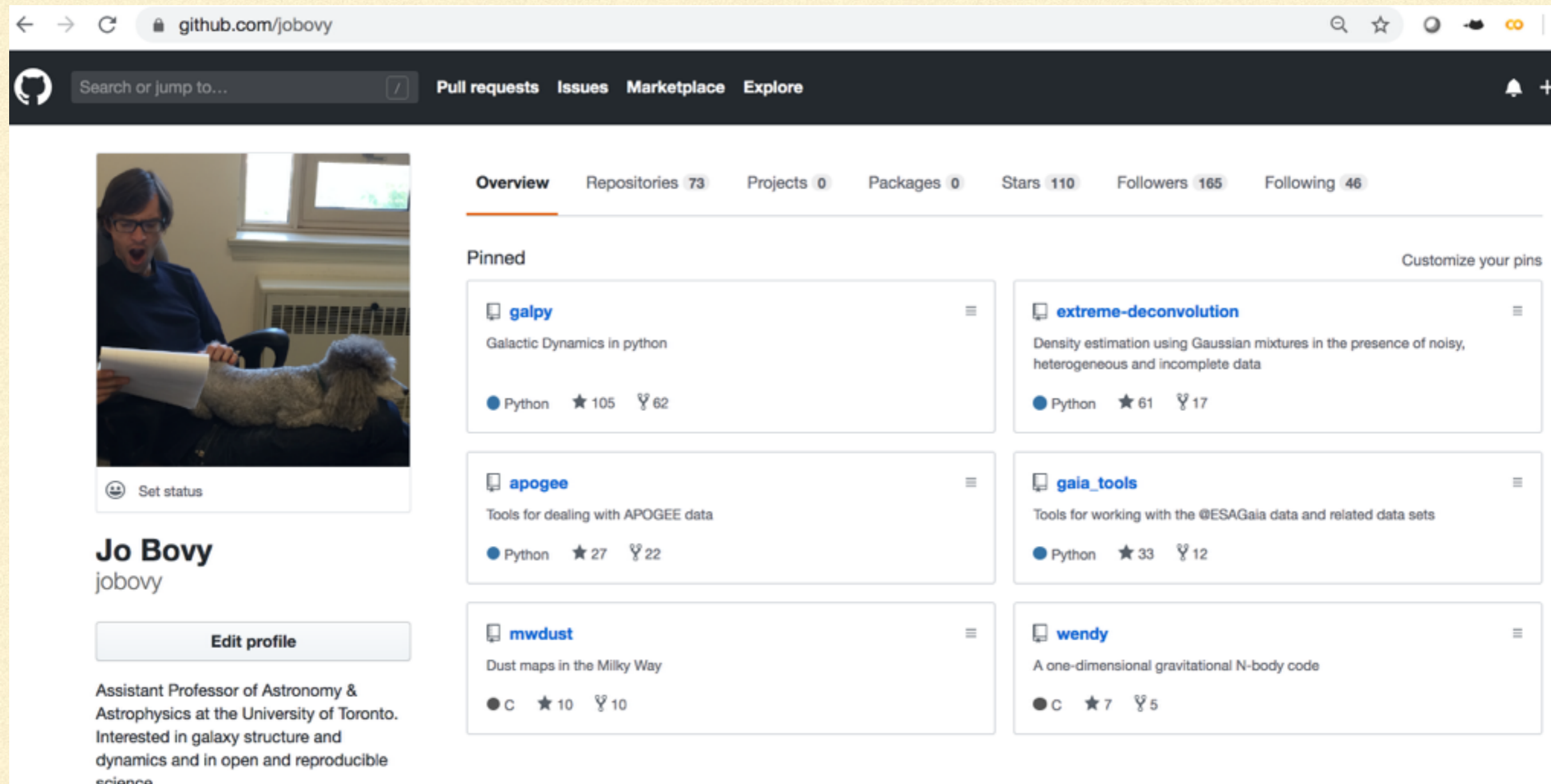

MINI-COURSE ON CODE DEVELOPMENT AND PACKAGING



WHY A MINI-COURSE ON CODE DEVELOPMENT AND PACKAGING?

- Most of astrophysical research these days consists of *a lot* of coding
 - Coding itself not a big part of undergrad/grad curriculum, much less how to package/distribute code
 - Lots of *lost opportunity for re-use of code*
 - Lots of *un- or under-documented code*
 - Lots of *un- or under-tested code*
 - Aim: crash-course on code packaging to help you over the initial hurdle in creating software packages with re-useable, documented, and tested code
-

MY OWN FORAYS IN CODE PACKAGING:



The screenshot shows the GitHub profile page for user 'jobovy'. The browser address bar displays 'github.com/jobovy'. The navigation bar includes 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. The profile section on the left features a profile picture of Jo Bovy with a dog, a 'Set status' button, and the name 'Jo Bovy' with the handle 'jobovy'. Below this is an 'Edit profile' button and a bio: 'Assistant Professor of Astronomy & Astrophysics at the University of Toronto. Interested in galaxy structure and dynamics and in open and reproducible science.' The main content area shows statistics: Overview (selected), Repositories 73, Projects 0, Packages 0, Stars 110, Followers 165, and Following 46. A 'Pinned' section displays six repositories in a grid:

- galpy**: Galactic Dynamics in python. Python, 105 stars, 62 forks.
- extreme-deconvolution**: Density estimation using Gaussian mixtures in the presence of noisy, heterogeneous and incomplete data. Python, 61 stars, 17 forks.
- apogee**: Tools for dealing with APOGEE data. Python, 27 stars, 22 forks.
- gaia_tools**: Tools for working with the @ESAGaia data and related data sets. Python, 33 stars, 12 forks.
- mwdust**: Dust maps in the Milky Way. C, 10 stars, 10 forks.
- wendy**: A one-dimensional gravitational N-body code. C, 7 stars, 5 forks.

LEARNING OBJECTIVES

- Learning in this course will largely happen through *you developing a small Python software package*, taking it through all of the basic stages of package development:
 - Set up the package in the standard way, use git/GitHub to track changes and share the in-development version (*today*)
 - Document the code, automatically host the documentation online (*class 2*)
 - Test the code, automatically run tests (*class 3*)
 - Publish the package (*class 4*)
 - Some advanced topics: write C extension, ... (*class 5*)
-

SOME LOGISTICS

- Meeting room: AB 113
 - Meeting time/date: Tue 10:30am — 12:30pm, Feb 25, Mar 3., Mar. 10, (break), Mar. 24., **Mar 26??**
 - Course website:
<https://github.com/jobovy/code-packaging-minicourse>
 - Slack channel #code-packaging-minicourse
-

ASSIGNMENTS

- *Crucial part of the course*
 - Four basic assignments, working through code examples of what we cover in class, building up your basic software package
 - Posted on course website
 - Second half of each lecture will be in-class work on the assignments / on developing our package
 - For those of you taking the course for credit: send me a link to your package, I will check that you do the assignments by looking through the commit history 😊
-

INTRODUCTION:
WHAT MAKES A GOOD SCIENTIFIC
SOFTWARE PACKAGE?

WHY PACKAGE CODE?

- You write *lots of code* and probably re-use code often or use *very similar* code
 - You may be *copying* bits of code when you re-use them, for example, between files in a single project, or between projects
 - That's prone to lead to confusion:
 - You have many different version lying around, doing the same thing or something slightly different
 - You likely don't *document* your code, even if you *comment* it
 - You almost surely don't *test* your code, because that's *hard*
 - Package your code!
 - Create a single version contained in a git repository to track changes over time, adding features when you need the code to also do things slightly differently from before
 - Re-use incentivizes documentation: tell *present-day you* what *six-months-ago you* meant for the code to do
 - Re-use incentivizes testing your code: make sure that code that you use *a lot* actually works!
 - Better science in the end!
-

WHY *RELEASE* CODE?

- Releasing your code allows *others* to use it
 - Gain exposure for your work and yourself
 - People who use your code will *remember who you are and learn about your work*
 - People who use your code *may want to collaborate on new projects*
 - Help the community progress, especially beginning researchers
 - Existing, released code provides a great jumping off-point for new research
 - Increase the standards of methods used in the field by releasing well-documented, well-tested, robust code
 - Releasing code should include at least a small commitment to help people use it and respond to bug reports (but manage your time!)
-

WHAT MAKES A SUCCESSFUL CODE PACKAGE?

- Essentially two tracks for successful packages:
 - Solve a *hard* problem (complex calculation, highly-optimized code, HPC): used because too difficult to implement oneself
 - Solve a *not-so-hard* problem in a convenient manner: make your code so easy to use that people prefer it to implementing the code themselves
 - We will focus on the 2nd track
 - Properties of a successful package:
 - Small and focused: don't set out to solve all of astrophysics, or to include all code you may want to share in a single package: pick a problem with a well-defined scope and stick to that
 - Make the code easy to install (probably *the* most important thing)
 - Make the code *easy and intuitive* to use
 - Make the code well-tested and robust
 - Degree of up-take has an upper limit set by the subject area, but other than that, ease-of-use and ease-of-installation will largely determine use
-

SOME DOS AND DON'TS: INSTALLATION

- Your code should be installable using standard commands:
 - `pip install X`
 - `conda install X`
 - `python setup.py install` for installing from source
 - Avoid any additional required setup if you can (like putting data files in a particular location) or use reasonable, documented defaults
 - Don't ask people to download the code, edit a `setup.py` or `Makefile` for their system, and then install
 - Attempt to support commonly-used operating systems (lots of people use macs, lots of people use windows)
-

SOME DOS AND DON'TS: DEPENDENCIES

- Use dependencies sparingly, because they will make installation/support/maintenance difficult
 - Support the latest major versions of Python and numpy
 - If you want your package to be heavy used and depended on, you need to be compatible with versions stretching years into the past
 - Make difficult-to-install dependencies optional to avoid them holding up use
 - ```
try:
 import difficult_package
except ImportError:
 _DIFFICULT_PACKAGE_LOADED= False
else:
 _DIFFICULT_PACKAGE_LOADED= True
```
  - Be wary of supporting alternative dependencies for the same task (I've made that mistake in the past!)
-



---

# SOME DOS AND DON'TS: DOCUMENTATION

---

- Write documentation *as soon as you start implementing a feature*. Don't think "I will document it once it's final". Writing the documentation may even help with designing the implementation
  - Remember that nothing is obvious to other users of your code: every argument and keyword needs to be documented, including its expected type
  - Having some documentation is better than none, so it can be sparse, but it's unlikely that you will write *too much documentation*
  - Keep a changelog to keep track of changes after the first implementation:
    - Keep a general HISTORY.txt (HISTORY.md) file for the package to keep track of major changes
    - Keep a history of major changes in the documentation of each function/class/...
-



---

# SOME DOS AND DON'TS: INTERACTING WITH THE USER

---

- Avoid printing anything
  - Avoid writing to files, all output should be returned as a return value
  - Avoid prompting the user for input, all inputs should be arguments of keywords
  - Use keywords with reasonable defaults for as many inputs as possible
  - Use standard Python error reporting, choosing the closest relevant among the standard exceptions:  

```
raise ValueError("MESSAGE")
```
  - Use standard Python warnings reporting (don't print warnings):  

```
import warnings
warnings.warn("WARNING MESSAGE")
```

Warn generously, users can always ignore when you use the standard warnings
-



---

# SOME DOS AND DON'TS: CODE DEVELOPMENT

---

- For any code package that will be used for years by multiple users, *maintenance* becomes a large part of the work
  - Develop your code with maintenance at the back of your mind:
    - *Avoid feature bloat*: Just because you could implement a feature doesn't mean you should. Every new feature will have to be maintained and *will* break, be mindful of future-you's time
    - *Comment liberally*, writing down the rationale for implementation decisions in the code, especially non-obvious ones; throughout your code, use descriptive variable, function, and class names, even for purely internal function (also document internal functions)
    - Try to keep a simple structure for your code, keeping it easy to navigate
  - Having a comprehensive test suite will help you spot maintenance issues as soon as they arise, when they are typically easier to fix
  - Keep your code clean and logical, but *not at the expense of user experience*: when you face the choice between a simple user experience and simple code, choose the user experience
-



---

`git`  
VERSION CONTROL

---



---

# WHY USE VERSION CONTROL?

---

- Version control keeps the history of changes to your code (or documents, images, etc., but text files work best), allowing you to trace changes over time. This frees you from having to track changes manually.
  - Most version control systems use a *central location* for the main copy of your code, which acts as
    - A crucial back-up of your work
    - A central place to share your code with yourself (for use on multiple machines) and others (e.g., collaborators)
  - Branches allow you to keep multiple in-progress versions of your code that can be developed in-parallel and merged later
-



---

# git VERSION CONTROL

---

- git is the latest and greatest version control system, probably the only one you've heard of
  - git has a decentralized approach to version control: by default, each version ("clone") of the code *repository* has the *full history of changes*
  - git is now closely associated with GitHub although technically they are independent from each other
-



---

# QUICK `git` INTRO

---

- We will now run through some basic and advanced features of `git`, based on the prior knowledge of the course participants
-



---

# THE BASIC STRUCTURE OF A PYTHON PACKAGE

---



---

# A PYTHON PACKAGE

---

- As part of this course, you will develop a Python package and eventually publish it to the Python Package Index (PyPI) for distribution to the community
  - We will build this from the ground up, starting with the most basic outline and adding features along the way
  - This is how I always build packages, at most starting from a basic outline from a previous package
  - There are also package templates (e.g., through `cookiecutter`) that typically come with *batteries included*, but these are highly confusing for beginners
-



---

# NAMING YOUR PACKAGE

---

- First decision is a big one: what do you want to name your package?
  - Use a name that is short, memorable, relevant, but most importantly *as unique as possible*:
    - Check whether the name that you have in mind exists on PyPI (so you can eventually `pip install PACKAGENAME`)
    - To be exhaustive, search GitHub and [sourcef4.net](https://sourcef4.net) to see what's already out there with the name you have in mind
  - The name of your package will quickly go in many different places, so give the naming careful thought *now*
-



---

# THE BASIC PACKAGE STRUCTURE

---

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

- Package lives in directory with its name
  - `__init__.py` file in that directory makes it a package (can be empty)
  - `setup.py` file will install the package (+distribute; needs content!)
-



---

# THE BASIC PACKAGE STRUCTURE

- Can write code directly into `__init__.py` or in another file in that directory and import into `__init__.py` (this allows you to use the functions as `from exampy import FUNCTION`)

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

```
__init__.py
from .utils import *
```

- \* Note that if you just write code in `utils.py`, you have to `import exampy.utils`
-



---

# THE BASIC PACKAGE STRUCTURE

- Submodules go into a subdirectory of the package (can also be a file, but then can't split into multiple files, which you will typically want to do)

```
TOP-LEVEL_DIRECTORY/
 exampy/
 submodule1/
 __init__.py
 __init__.py
 utils.py
 setup.py
```

- \* Now you can import the submodule as `import exampy.submodule`
  - \* Structure of the submodule exactly analogous to the top-level module
-



---

# THE `setup.py` FILE

---

- Key to allowing your package to be installed using standard tools
  - Uses `setuptools` to build, install, and package/distribute the code
  - (alternatively, you can use a `setup.cfg` configuration file, but still need `setup.py`)
  - Basic `setup.py` file calls `setuptools.setup` to define the package
-



---

# THE `setup.py` FILE

---

- Basic example:

```
setup.py
import setuptools

setuptools.setup(
 name="exampy",
 version="0.0.1",
 author="Jo Bovy",
 author_email="bovy@astro.utoronto.ca",
 description="A small example package",
 packages=["exampy"]
)
```

- \* From a packaging standpoint, most important here is `packages=["exampy"]` which tells the setup what the package is; can also do `packages=setuptools.find_packages(include=['exampy', 'exampy.*'])`
-



---

# THE `setup.py` FILE

---

- Add `long_description`:

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



---

# THE `setup.py` FILE

---

- Add classifiers:

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



---

# THE `setup.py` FILE

---

- More options:
    - `url:homepage`
    - `license:name of the license`
    - `python_requires:constraints on Python versions, e.g., python_requires='>=3'`
    - `install_requires:basic dependencies, e.g., install_requires=["numpy", "scipy"]`
    - `package_data:non *.py files that need to be added to the package, e.g., package_data={"": ["README.md", "LICENSE"]}`
    - `entry_points:use for scripts that are part of the package`
-



---

# INSTALLING YOUR CODE FOR DEVELOPMENT

---

- Normally use:

```
python setup.py install
```

- For development use:

```
python setup.py develop
```

- or

```
pip install -e .
```

- This allows you to edit the code and use it without constantly re-installing it
-



---

# CODE LICENSES

---



---

# WHY DOES YOUR CODE NEED A LICENSE?

---

- Without a license, your code is assumed to be copyrighted to you, not allowing re-use, modification, or re-distribution
  - You want people to use your code? It needs a license
  - A license also provides legal protection against liability: explicitly deny any liability related to any use of the code
-



---

# TWO CATEGORIES OF LICENSES

---

- Permissive licenses:
    - Allow arbitrary use, modification, and re-distribution provided (typically) that the original license is retained and the original author credited, denying liability
    - Examples: MIT License, BSD 3-clause License (these are *simple*)
    - License of choice for most Python projects
  - Copy-left licenses:
    - Allow use, modification, and re-distribution provided that the re-distributed code is licensed under the equivalent terms
    - Example: GNU General Public License v3
    - Use when you want to make sure that your code remains open source, but make inclusion of your code in other projects difficult
-



---

GitHub

---



---

# GITHUB

- Main website to share code these days (RIP [code.google.com](http://code.google.com))
  - Acts as central 'main' repository for your code that you use to share with yourself (on different machines) and others
  - But much more:
    - Bug tracker: Issues
    - Forks: Copies of your repository by others that branch off of your repository, main way for other users to contribute to your code (they don't need write access)
    - Pull requests: Method for merges between branches in forks and branches in the main repository
    - Formatted README.md gives nice-looking home-page for your code
    - Hooks to other webservices: automated documentation, automated testing, etc.
-



---

# INTERACTING WITH GITHUB

---

- Get a local copy by “cloning” the GitHub repository (get link on GitHub page)

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

- Then basic git commands will automatically work
- When you create a branch, you need to push it to GitHub to tell it about it


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

---




# GITHUB ISSUES


<> Code | **Issues 9** | Pull requests 1 | Actions | Projects 0 | Wiki | Security | Insights | Settings




Write | Preview | AA | B | i | “ ” | <> | 🔗 | ☰ | ☰ | ✓☰ | @ | 📌 | ↶


Leave a comment


Attach files by dragging & dropping, selecting or pasting them. 


 Styling with Markdown is supported


[Submit new issue](#)

 Remember, contributions to this repository should follow its [contributing guidelines](#).

**Assignees**   
No one—assign yourself

**Labels**   
None yet

**Projects**   
None yet

**Milestone**   
No milestone

**Linked pull requests**  
Successfully merging a pull request may close this issue.  
None yet



---

# A GOOD ISSUE

---

- contains a minimal, complete, reproducible example code snippet:
    - minimal: use as little code as possible
    - complete: should be able to be run without any additional stuff
    - reproducible: make sure that this snippet reproduces the error
  - So you likely have to edit down the actual problem you are having into a small, complete code snippet that has the same issue; this can be the basis of a test added to the test suite to make sure the error remains solved
  - Include operating system, Python version, version of dependencies
  - Promptly respond to queries for additional information or requests to test possible solutions
  - You can refer to the issue in commits as `#ITSNUMBER` to link these commits in the issue's page
-




# A NEW REPOSITORY

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)



Owner

 jobovy ▾

Repository name \*

Great repository names are short and memorable. Need inspiration? How about **refactored-telegram**?

Description (optional)

-  **Public**  
Anyone can see this repository. You choose who can commit.
-  **Private**  
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

- Initialize this repository with a README**  
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾

Add a license: **None** ▾



Create repository



---

# ASSIGNMENT I

---



---

# ASSIGNMENT I

---

- Create your package
  - Create it on GitHub
  - Play around with `git`
  - Play around with GitHub
-