INF200_H21_Ju04

June 2, 2022

1 INF200 Lecture No. Ju04

1.0.1 Hans Ekkehard Plesser

1.0.2 2 June 2022

1.1 Today's topics

- Coding
  - Deleting from lists
  - Adding new objects conditionally
  - A little more on random selection
- Packaging and documentation
  - Packaging your code for distribution
  - Choosing version numbers
  - Running tests with Tox and running tests on GitLab
  - Creating documentation with Sphinx

---

2 Deleting from lists

- Removing elements from a list inside a loop over the list is dangerous
- It can confuse the list iteration
- Example: remove numbers that can be divide by 2 or 3

2.1 A correct loop

[1]:
```
d = list(range(10))
for n in d:
    print('Testing', n, end='')
    if n % 2 == 0 or n % 3 == 0:
        print(' ... divisible', end='')
    print()
```

```
Testing 0 … divisible
Testing 1
Testing 2 … divisible
Testing 3 … divisible
Testing 4 … divisible
Testing 5
Testing 6 … divisible
Testing 7
Testing 8 … divisible
Testing 9 … divisible
```

2.2 A confused loop

[2]:
```
d = list(range(10))
for n in d:
    print('Testing', n)
    if n % 2 == 0 or n % 3 == 0:
        d.remove(n)
print(d)
```

```
Testing 0
Testing 2
Testing 4
Testing 6
Testing 8
[1, 3, 5, 7, 9]
```

2.3 A better solution: keep the good ones

[3]:
```
d = list(range(10))
d = [n for n in d if not (n % 2 == 0 or n % 3 == 0)]
print(d)
```

```
[1, 5, 7]
```

---

3 Adding new objects conditionally (aka "birth")

[4]:
```
import random
```

3.1 A class allowing objects to create new objects of the class

New objects are created - with a given probability p - may fail to be created if the chosen "weight" is negative - `clone()` returns new object or `None`

[5]:
```
class Q:

    p = 0.5

    def __init__(self, w):
        assert w > 0
        self.w = w

    def __repr__(self):
        return f'Q({self.w:.2g})'

    def clone(self):
        if random.random() < self.p:
            nw = random.gauss(1, 1)
            if nw > 0:
                return Q(nw)
        return None
```

[6]:
```
random.seed(12345)
q = Q(10)
[q.clone() for _ in range(10)]
```

[6]:
```
[Q(2.9), Q(1.1), Q(1.4), Q(2.2), None, Q(1.5), None, Q(1.4), None, Q(0.89)]
```

3.2 A class hierarchy with similar properties

- `A` is an abstract base class
- Only objects of subclasses `B` and `C` can be instantiated
- Cloning is still done in the base class using `type()`

[7]:
```
class A:

    p = None

    def __init__(self, w):
        assert w > 0
        self.w = w

    def __repr__(self):
        return f'{type(self).__name__}({self.w:.2g})'

    def clone(self):
        if random.random() < self.p:
            nw = random.gauss(1, 1)
            if nw > 0:
                return type(self)(nw)
        return None

class B(A):
    p = 0.5

class C(A):
    p = 0.3
```

[8]:
```
random.seed(12345)
b = B(10)
[b.clone() for _ in range(10)]
```

[8]:
```
[B(2.9), B(1.1), B(1.4), B(2.2), None, B(1.5), None, B(1.4), None, B(0.89)]
```

[9]:
```
random.seed(12345)
c = C(10)
[c.clone() for _ in range(10)]
```

[9]:
```
[None, C(1.4), None, C(0.25), None, C(1.8), None, C(1.7), None, None]
```

3.3 A function to produce many clones

- Takes a list of objects
- Gives every object the opportunity to clone
- Returns list of only those objects that were cloned (drops `None`s)

3.3.1 First implementation: explicit loop

[10]:
```
def mc1(d):
    r = []
    for x in d:
        xc = x.clone()
        if xc:
            r.append(xc)
    return r
```

3.3.2 Second implementation: list comprehension (requires Python >= 3.8)

- Use *assignment expression* in Python 3.8

[11]:
```
def mc2(d):
    return [xc for x in d if (xc := x.clone())]
```

3.3.3 Test both implementations

[12]:
```
for f in [mc1, mc2]:
    random.seed(12345)
    d = [B(10), C(20), B(10)]
    new_d = f(d)
    d.extend(new_d)
    print(d)
```

```
[B(10), C(20), B(10), B(2.9), C(1.1), B(1.4)]
[B(10), C(20), B(10), B(2.9), C(1.1), B(1.4)]
```

4 A little more on random selection

4.1 Case 1: Dead or alive

- An animal has a probability p to die
- How do we decided if the animal will die in a given year?
  - Draw uniformly distributed random number from $[0, 1)$ and compare to p

4.2 Case 2: Choosing between multiple alternaives

- Literature: Knuth, The Art of Computer Programming, vol 2, ch 3.3-3.4
- In a simluation, we want to choose between four alternatives with probabilities p_0, p_1, p_2, p_3
- Note $\sum_{n=0}^{3} p_n = 1$ by definition
- Cumulative probabilities $P_n = \sum_{k=0}^{n} p_k$ divide unit interval in sections corresponding to events 0, 1, 2, 3
- Specifically, we choose a random number r and select

$$\begin{cases} \text{event } 0 & \text{if } r < P_0 \\ \text{event } n & \text{if } P_{n-1} \leq r < P_n \quad \text{for } n > 0 \end{cases} \tag{1}$$

- The following code will select from `len(p)` alternatives with probabilities `p[0]`, `p[1]`, …

[13]:
```python
def random_select(p):
    r = random.random()
    n = 0
    while r >= p[n]:
        r -= p[n]
        n += 1
    return n
```

4.2.1 Simpler approach for our simulations

- Animals move in all four directions with *same* probability
- Can use `random.choice()` to pick one element from a list with equal probability

---

5 Packaging your code for distribution

Let us say you have spent the last year creating some really great Python code, and now you want to share it with others. What do we need to do? - Need to put "everything together" into a nice "parcel" - Need to handle *dependencies* (e.g., that our code needs NumPy) - Need to "spread the word (code)"

Python solution: *Packaging*

5.1 Packages *vs* Packages

You might have noted that we now have to different things called *packages*, they are either - Collections of modules (*import packages*) - A collection of code neatly packaged for sharing with others (*distribution packages*)

Yes, having the same name for two different things is confusing. Programmers are horrible at naming conventions, we just have to deal with that

The Python Packaging User Guide Glossary defines a Distribution Package as

```
"A versioned archive file that contains Python packages, modules, and other resource files tha
```

5.1.1 Where to share distribution packages?

You have now created a nice distribution package of your code (we will check out the details soon), how do you share it? - If it is only with a few people, email, direct transfer, etc is fine - If you want to keep the code open for everyone to see, github/bitbucket is a nice way to do it - Alternatively, you can use the Python Package Index (PyPI), aka the "CheeseShop" - If you want to make it easily available for Conda users, consider creating a Conda package as well - Discussion of Conda vs PIP by Jake Vanderplas

5.1.2 Python packaging: a convoluted history

- Creating distribution packages for Python has a long and difficult history
- Pure Python packages reasonably simple, but packages depending, e.g., on optimized numerics libraries such as NumPy were difficult
- Various approaches over time, e.g., setuptools, distutil, eggs, wheels, …
  - also external package managers such as conda
- Relatively recent standardization
  - PEP 517 — A build-system independent format for source trees
  - PEP 518 — Specifying Minimum Build System Requirements for Python Projects
- Still a lot of outdated or partially up-to-date information out there

5.2 How do we prepare our code for distribution?

We cover only the basics here.

- Description in the following built on
  - https://packaging.python.org/tutorials/packaging-projects/
  - https://docs.python.org/3/distributing/index.html
  - https://setuptools.readthedocs.io/en/latest/userguide/quickstart.html
  - https://setuptools.readthedocs.io/en/latest/userguide/declarative_config.html
- For more information, see also
  - https://packaging.python.org/guides/distributing-packages-using-setuptools/
  - https://packaging.python.org/guides/
- Or this guide by Yngve Moe, one of the INF200 examiners
  - https://github.com/yngvem/python-project-structure/

Key idea of a distribution package We want to make sharing Python-based projects easy - Collect - Source code: Python modules, import packages, tests - Example scripts - Documentation - …

- Provide *metadata* about the code, e.g.,
  - Purpose, Dependencies, Author information

    - License information, Version information, …
- Provide a *build archive*
- Support easy installation to predefined locations

Example: Typical distribution package directory layout

```
biolab_project/
   docs/
   examples/
      experiment_01.py
      ...
   src/
       biolab/
          __init__.py
          bacteria.py
          ...
   tests/
       test_bacteria.py
       ...
   .gitlab-ci.yml
   LICENSE
   pyproject.toml
   README.md
   setup.cfg
   setup.py
   tox.ini
```

In our example, `biolab` is an import package included in our distribution, it is the source code. In this example `tests` is placed next to the source code package.

In addition to the `biolab` package we have a folder called `examples`, with some scripts the user can look at to see how the `chutes` packaged can be used. Note that `examples` is *not* a package, as it does not have an `__init__.py` file, it is just a regular folder. If you have a Jupyter notebook with examples, it could also be placed here. `docs` contains documentation, see below.

Configuration files

- `LICENSE` includes the license for your code.
    - Choose your license carefully!
    - Do not try to write your own license (unless you are a lawyer, maybe …).
    - Three major categories of open source licenses
        * Viral licenses, e.g., GNU Public License (GPL)
        * Permissive licenses, e.g., BSD or MIT licenses
        * Other licenses
    - See also
        * https://opensource.org/licenses
        * https://choosealicense.com
- A `README.md` contains a description of the distribution package, and usually contain some

information to the user about how to install it and where to look for examples/documentation. The file type is flexible, but Markdown is common

- `pyproject.toml` describes the build system for creating your package. It should usually be just

  ```
  [build-system]
  requires = ["setuptools", "wheel"]
  build-backend = "setuptools.build_meta"
  ```

- `setup.cfg` is the main configuration file describing our package, discussed in more detail below. It is a *declarative* (*static*) configuration file. This is the preferred modern way of providing the configuration information. For details, see comments in `biolab_project/setup.cfg` and links above.

- `setup.py` is an *imperative* (*dynamic*) file used in the past to provide information about package configuration. It can fill the same role as `setup.cfg` and was widely used in the past. If you have a `setup.cfg` file, then `setup.py` can be omitted or should only contain

  ```
  import setuptools
  setuptools.setup()
  ```

- `tox.ini` and `.gitlab-ci.yml` configure testing and will be discussed below.

5.3 Building a distribution package

- Open a terminal or Anaconda prompt, or open a Terminal in PyCharm
- Go to the top-level directory for your project, here `biolab_project`
- Make sure you have the correct conda environment activated
- Run

```
python -m build
```

- Python `setuptools` will do all the work
  - Files created in this process are placed in directory `build`
  - The files you can distribute will be placed in `dist`
- In our case we get in `dist` (on Windows, we get `zip` archive instead of `tar.gz`)
  - `biolab-0.1-tar.gz` (plain archive with source files)
  - `biolab-0.1-py3-none-any.whl` (Python Wheel)
- Wheels are current standard Python package distribution archives
  - can handle depdencies on C libraries
  - `none-any` can be replaced with system-specific names if building with C libraries
  - For more, see https://realpython.com/python-wheels/
- Material in `build` or `dist` could confuse PyCharm code inspection, so set `Mark directory as > Excluded` for those directories.
- **Do not commit** the `build` and `dist` directories!
- You could now upload your package to PyPi using `twine`, but we will skip that part in this course.

5.4 Installing a package

5.4.1 “Manually” from a plain archive

This is the old-fashioned (pre-wheel) way of doing it.

1. Unpack the `tar.gz` or `zip` file
2. Move into the directory that we unpacked
3. Run

   ```
   python setup.py install
   ```

- This will install in the default location for packages in your current Python environment.
- Packages installed like this **cannot** be uninstalled easily.

5.4.2 Installing the `pip` way

- If package is available on PyPi, just

  ```
  pip install xyz
  ```

- To install from a local file

  ```
  pip install biolab-0.1.0-py3-none-any.whl
  ```

- Also installs to default location
- Package can be uninstalled with `pip uninstall biolab`

5.5 Choosing version numbers

- Semantic Versioning is a widely used approach

5.5.1 Semantic versioning principles

The following are key principles of semantic versioning from semver.org. See that page for details.

1. Software using Semantic Versioning MUST declare a public API … it SHOULD be precise and comprehensive.
2. A normal version number MUST take the form X.Y.Z where … X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically.
3. Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications MUST be released as a new version.
4. Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.
5. Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.
6. **Patch version Z** (x.y.Z | $x > 0$) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
7. **Minor version Y** (x.Y.z | $x > 0$) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality

or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.

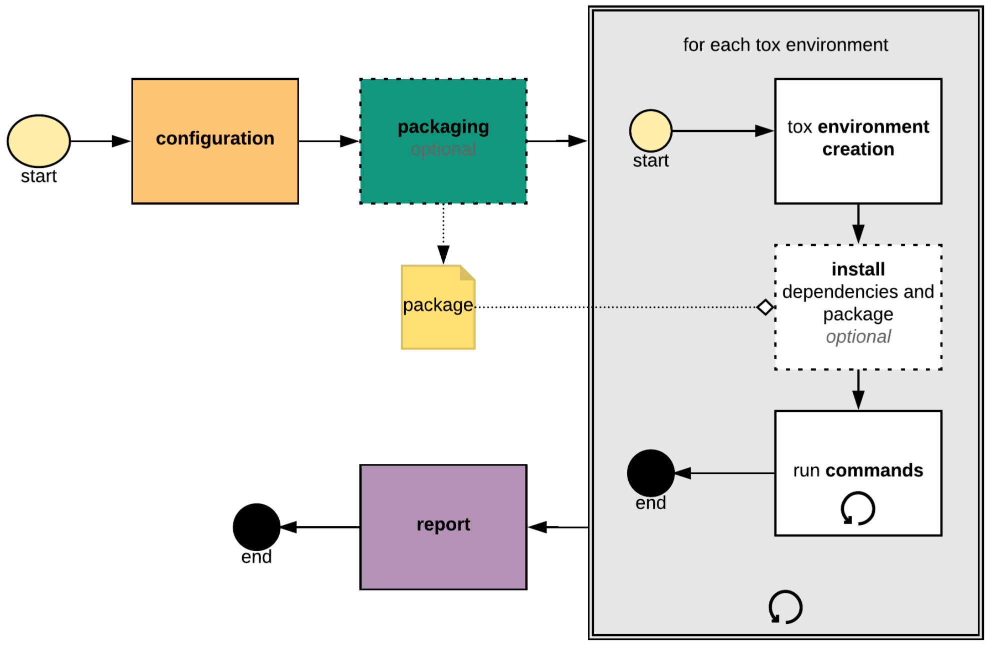
8. **Major version X** (X.y.z | X > 0) MUST be incremented if any backwards *incompatible* changes are introduced to the public API. It MAY also include minor and patch level changes. Patch and minor versions MUST be reset to 0 when major version is incremented.

---

6 Running tests with Tox and running tests on GitLab

6.1 Tox

- Tox manages environments for controlled running of tests
- Sets up environment according to `setup.cfg` and `tox.ini` specifications and tests in this well-defined enviroment



- Configured by `tox.ini`, see `biolab_project/tox.ini` for example
- Run in terminal as

```
tox
```

- Places all its files in `.tox` directory, delete this if changes you make to Tox configuration seem to have no effect
- Mark `.tox` directory as `Excluded` in PyCharm
- **Do not commit** the `.tox` directory!

6.2 GitLab test runners

- GitLab (and Github, Travis, Jenkins, ...) can tests automatically for us
- Test on every push to repository
- Good practice, also known as Continuous Integration Testing (CI)

- Can be extended to Continuous Delivery/Deployment (CD)
- Requires `.gitlab-ci.yml` file at *top level* of repository
- See `biolab_project/.gitlab-ci.yml` for an example

6.3 Notes

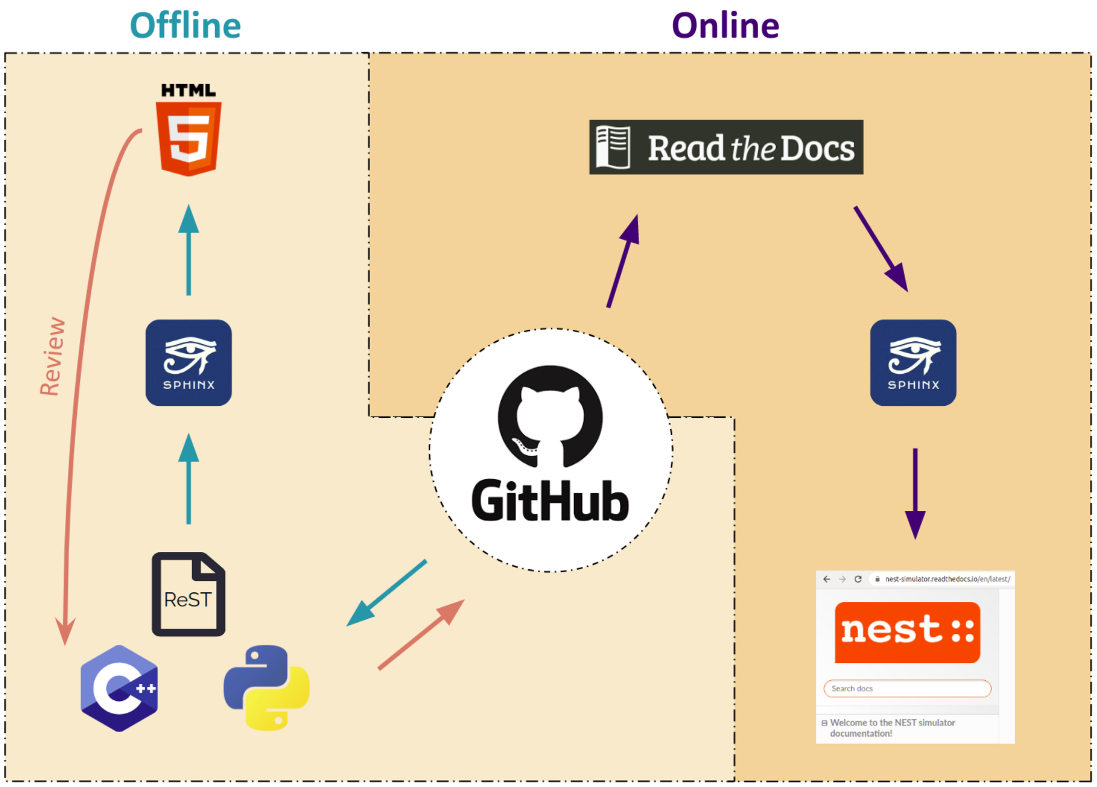
- At present, GitLab test runners will not work for you (GitLab asks for credit card information to check you identity)
- I hope this problem will be fixed soon

---

7 Creating documentation with Sphinx

7.1 What is Sphinx?

- Sphinx is a tool for generating documentation for your code
- Can compile documentation to many different formats: LaTeX, pdf, html, etc.
- Can read out docstrings in your code and include in the documentation
- Sphinx-generated documentation can easily be served online, e.g., via ReadTheDocs
  - ReadTheDocs can pick up material from your GitLab/Github repo
  - Automatically updates documentation on very push
  - Can handle multiple versions
  - Configured by `.readthedocs.yml` file at top level in repository
  - Requires ReadTheDocs account
  - We will skip this in this course



7.2 Getting started: `sphinx-quickstart`

1. Open `Terminal` within PyCharm
   - Alternative: open `Terminal` under OSX/Linux or `Anaconda Prompt` under Windows and navigate to your `biosim-axx-Name1-Name2` folder (use `cd` to change directories)
2. Ensure your `inf200` conda environment is activated
3. Run the following command

```
sphinx-quickstart --ext-autodoc --ext-coverage --ext-mathjax --ext-viewcode docs
```

1. Accept default answers for questions by pressing ENTER and enter sensible values for
   - Project Name
   - Author Names(s)
   - Project version
2. Don't worry if you make a mistake, you can fix it in the `docs/conf.py` file
3. Open file `conf.py` in the `docs` directory and change the following lines (approx line 15)

   ```
   #import os
   #import sys
   #sys.path.insert(0, os.path.abspath('.'))
   ```

   to

   ```
   import os
   import sys
   ```

```
sys.path.insert(0, os.path.abspath('../src'))
autoclass_content = 'both'
```

The first line ensures that Sphinx finds all code in the project directory, the second that documentation will be generated for all constructors.

4. Finally, add the following line at the end of `conf.py`

```
latex_elements = {'papersize': 'a4paper'}
```

7.3 Write documentation sources

1. Edit the `docs/index.rst` file and add additional documentation `*.rst` files
2. For a worked example, see `Project/SampleProjects/biolab_project`
3. For more information on restructured text, see
   - Sphinx ReStructuredText primer
   - Another ReStructuredText primer
   - Full Sphinx ReStructuredText documentation
4. For some suggestions on writing documentation, see the NEST Documentation Guide

7.4 Generate documention

1. Open a Terminal (e.g. inside PyCharm) and navigate to the `docs` folder inside your project.
2. Run

   ```
   make html
   ```

   This will create basic documentation, which you by opening `docs/_build/html/index.html` in a web browser.
3. If the command above does not work in the terminal you opened in PyCharm, try opening a normal Terminal, navigate to the `docs` directory and try again.
4. To create documentation in other formats, run, e.g.

   ```
   make epub
   make latexpdf
   ```

   The resulting documentation will be in the `epub` and `latex` directories, respectively. Creating these formats may require additional software on your computer, especially a working TeX system, e.g.

   - Windows: MikTeX
   - OSX: MacTex

   Under Windows, you may have to run

   ```
   make latex
   cd _build/latex
   pdflatex biosim
   ```

   If Sphinx tells you that Perl.exe is missing to build the LaTeX file, you can install Perl using `conda install perl` or install Perl from https://strawberryperl.com (not tested yet).
5. Run `make html` again after you made changes to the documentation.

6. Run `make clean` to remove any generated documentation and temporary files if you run into problems.
7. See `*.rst` files in `biolab` project for how to automatically generate documentation from docstrings.

7.4.1 Keep Sphinx-generated documentation out of Git repo!

The documentation that is generated in the `docs/_build` directory **should not be committed to your git repository**!

`docs/_build` should automatically be ignored by git if you have put the right `.gitignore` file in place (copied from course repo `project_description/sample.gitignore`.

If the `docs/_build` directory is not ignored by git, proceed as follows: 1. If you have not yet put `.gitignore` in place, do it now and see if `docs/_build` is ignored afterwards. 1. If the `docs/_build` build directory is still not ignored, there are a few possibilities: 1. The `docs` directory has a different name, e.g. `Docs` or `doc`. Rename it to `docs`. 1. The `docs` directory is not at the top level within the `BioSim_Txx_Name1_Name2` folder. Move it to the top level. 1. The `.gitignore` file is not at the top level within the `BioSim_Txx_Name1_Name2` folder. Move it there. 1. If none of this helps, contact Hans Ekkehard! 1. Commit your changes if you changed `.gitignore` or moved a directory.

7.5 Formatting options for docstrings

Instead of the standard format for docstrings, e.g.,

```
def repeat(text, copies):
    """
    Repeat given text a given number of times.

    :param text: a string
    :param copies: an integer
    :return: string, text concatenated copies times
    """
```

one can also use NumPy-style docstrings which look like this

```
"""
Repeat given text a given number of times.

Parameters
----------
text : str
    Text to be repeated
copies : int
    Number of repetitions

Returns
-------
str
    Text concatenated copies times.
```

"""

For more on the NumPyDoc format, see - http://numpydoc.readthedocs.io/en/latest/format.html - http://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html

To work with NumPyDoc docstrings, you need to do the following: 1. In `docs/conf.py`, around line 35, add `'sphinx.ext.napoleon'` to the list of `extensions`. 1. In PyCharm, open Preferences, go to `Tools > Python integrated tools` and select `Docstring format` NumPy

7.6 Further documentation on Sphinx

- Sphinx homepage
- "Guided tour" to documenting with Sphinx
- Sphinx tutorial from the Matplotlib folks
- Documentation tutorial by Brenadn Hasz
- A lot of projects using Sphinx for documentation

```
[ ]:
```