Managing programming projects

Kylie A. Bemis

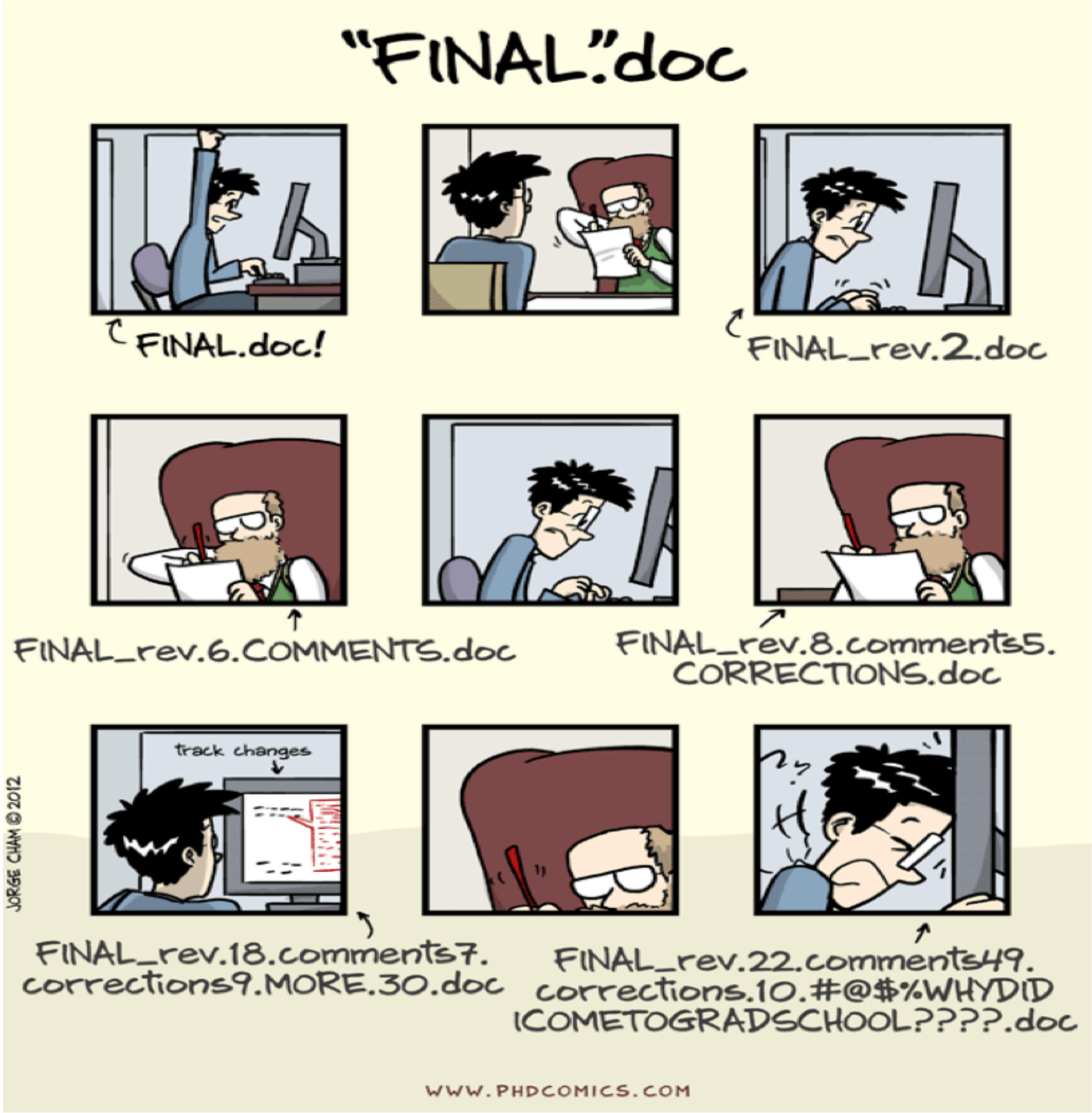
Northeastern University
Khoury College of Computer Sciences

Goals for today

- Source control
- Testing code
- Packages and modules

SOURCE CONTROL

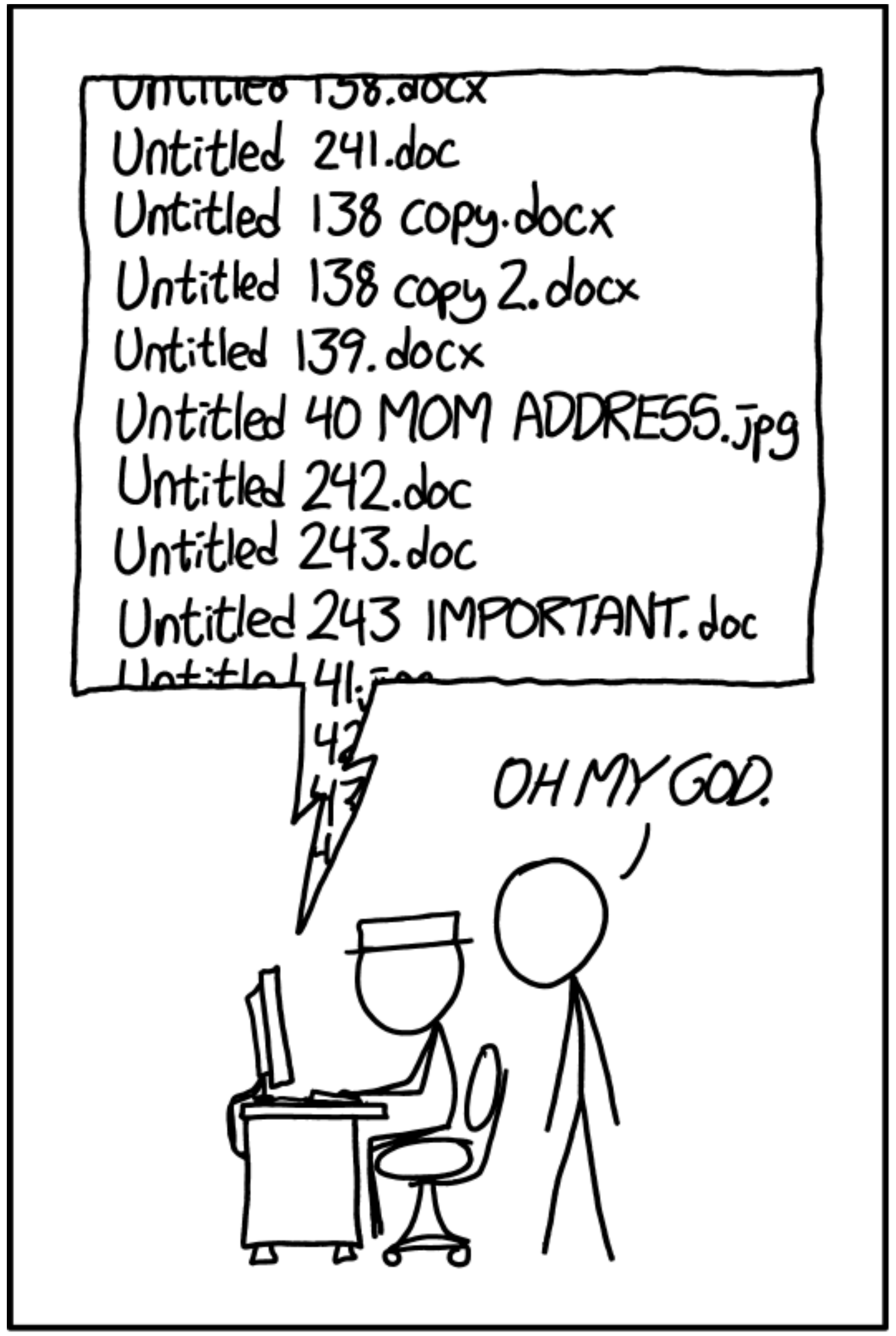
How do you manage files in a project?


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.corrections9.MORE.30.doc
FINAL_rev.22.comments49.corrections.10.#@$%WHYDID ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

title: "notFinal.doc" - originally published 10/12/2012

Version control

- Complex projects produce many files
- Need to **track changes** and **versions**
- Need to **share with collaborators**



PROTIP: NEVER LOOK IN SOMEONE ELSE'S DOCUMENTS FOLDER.

Goals of version control

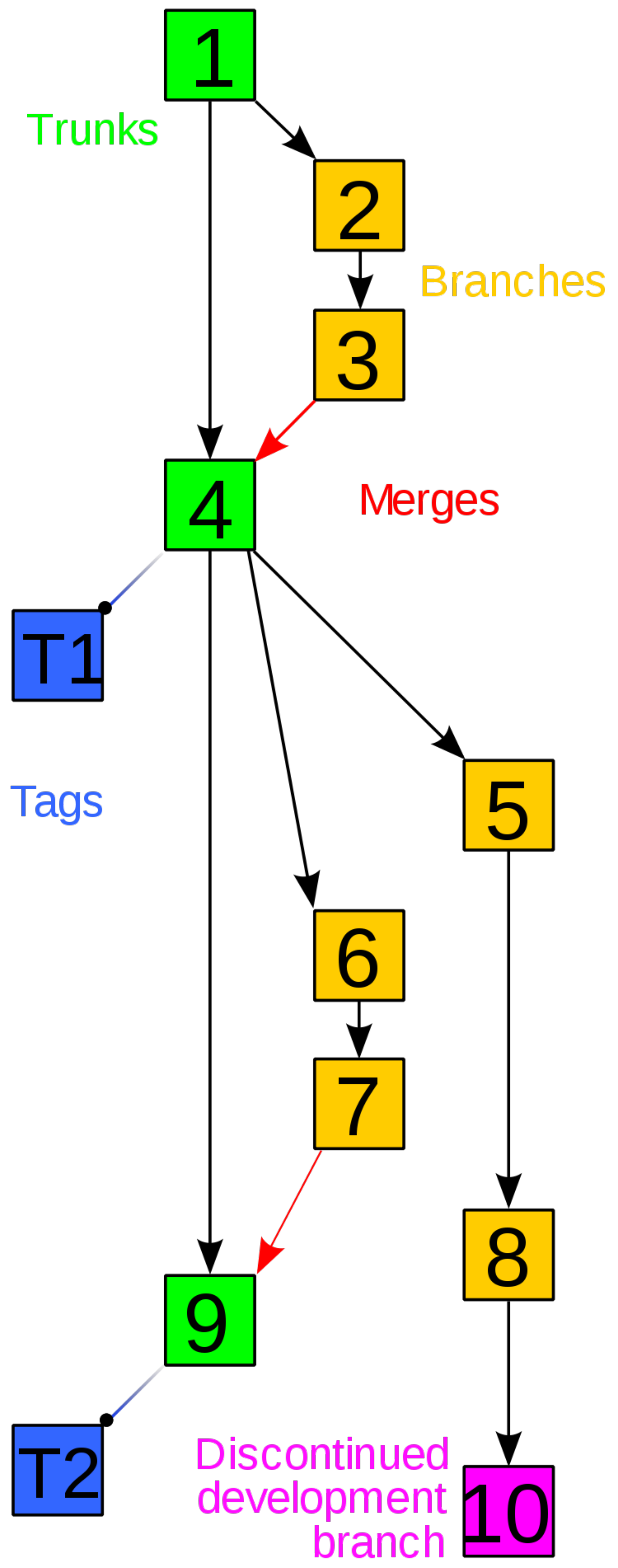
- Track changes made to a project
  - Track **changes across multiple files**
  - Track **creation** and **deletion** of files
  - **Revert** and **merge** changes as necessary
- Allow multiple **branches** of progress
- **Synchronize work** with collaborators

Vocabulary

- A **repository** (“repo”) stores a project tracked by version control and its history
- A **commit** is a snapshot of a set of ***changes***
- The project ***head*** is the most recent commit
- Changes can be **pushed** and **pulled** from one repository to another

History and branches

- Revisions depend on earlier revisions
  - Each revision is linked to the revisions it depends on
- Progress may *fork* into separate **branches**
  - Develop new features or prepare bug patches
  - **Merge changes** back to the ***trunk*** or "main" branch
- Project history forms a ***graph***



Git and Github

- **Git** is a popular version control system
  - Distributed version control system
  - Repo is mirrored on each developer’s machine
  - No need to rely on a central server
- **Github** hosts online Git repositories
  - Free hosting of open-source projects
  - Share work with collaborators

Installing Git

- First check if Git is already installed
- **macOS** download:
  - https://git-scm.com/download/mac
- **Windows** download:
  - https://gitforwindows.org

Setting up Git

- Git needs to know who is making changes
- Configure your credentials:
  - `git config --global user.name <your name>`
  - `git config --global user.email <your email>`
  - `git config --global --list`

Using Git

- Any directory with a git history is a repo
- Initialize a git repo in a directory:
  - Navigate to a directory
  - `git init`
- Any files in the repo can now be tracked

Using Git and Github

- Share work on a remote repository
- Create a new repo on Github
- Clone the repo locally
  - Copy the web URL from the Github repo page
  - `git clone <URL>`
- Work locally and push to Github

Understanding Git

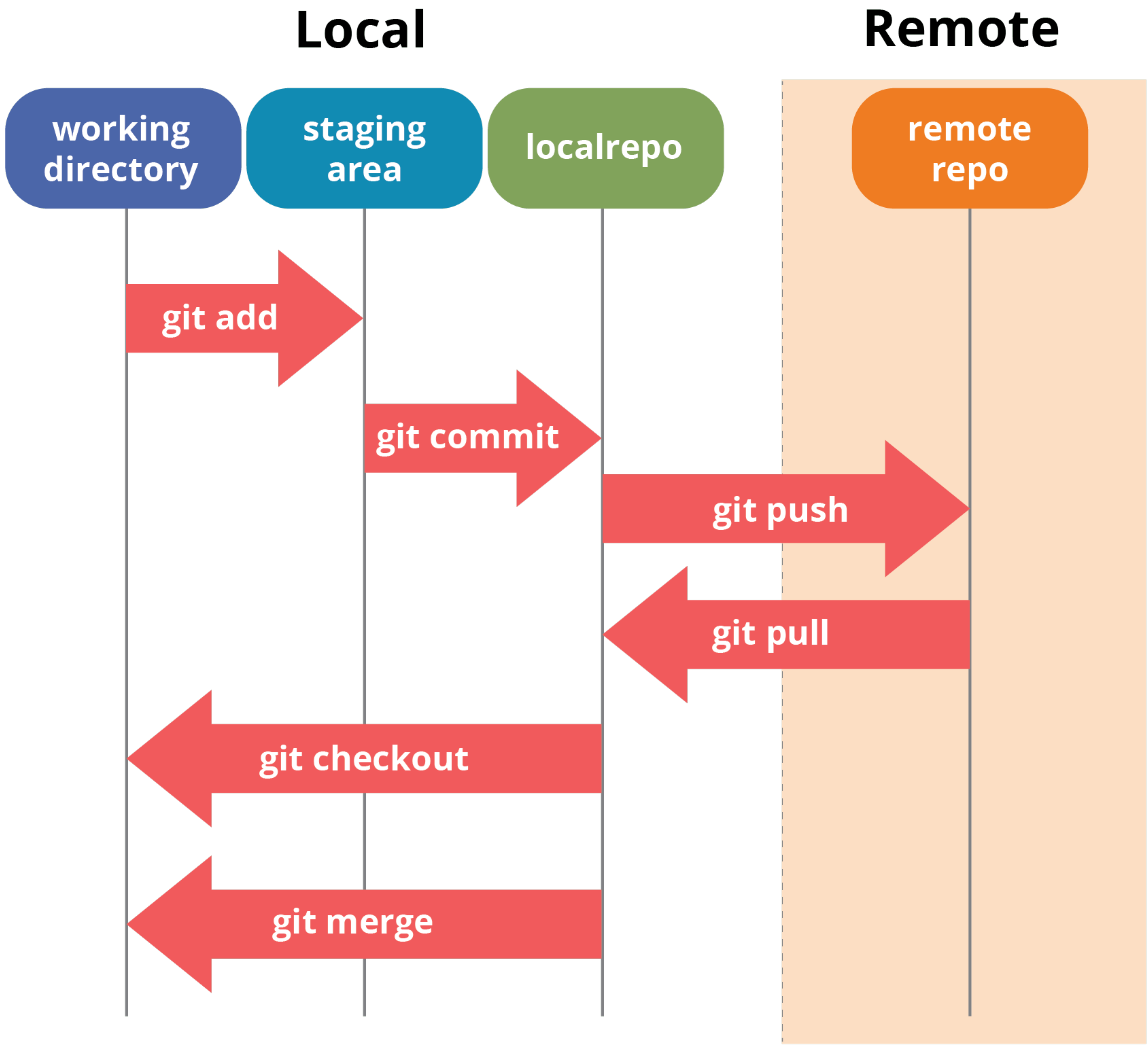
- **Working directory** is the directory on your machine where the repo lives

- **Staging area** is the set of files that has changed since your last ***commit***

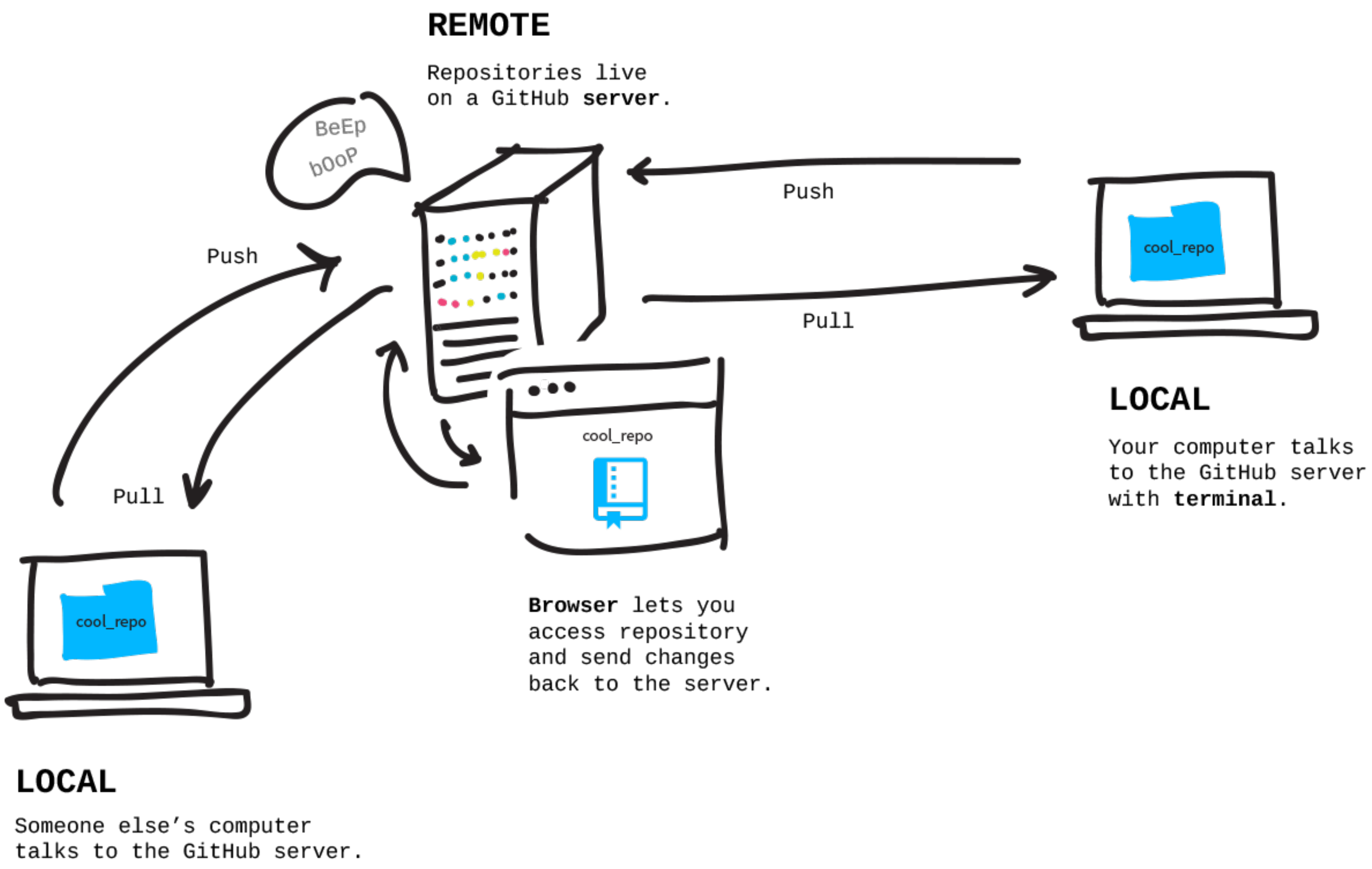
- The **local repository** is the repo on your machine (including its complete history)

- A **remote repository** is a version of the repo on a remote site such as Github

Git workflow



Visualizing local and remote repos



REMOTE

Repositories live
on a GitHub **server**.

LOCAL

Your computer talks
to the GitHub server
with **terminal**.

Browser lets you
access repository
and send changes
back to the server.

LOCAL

Someone else's computer
talks to the GitHub server.

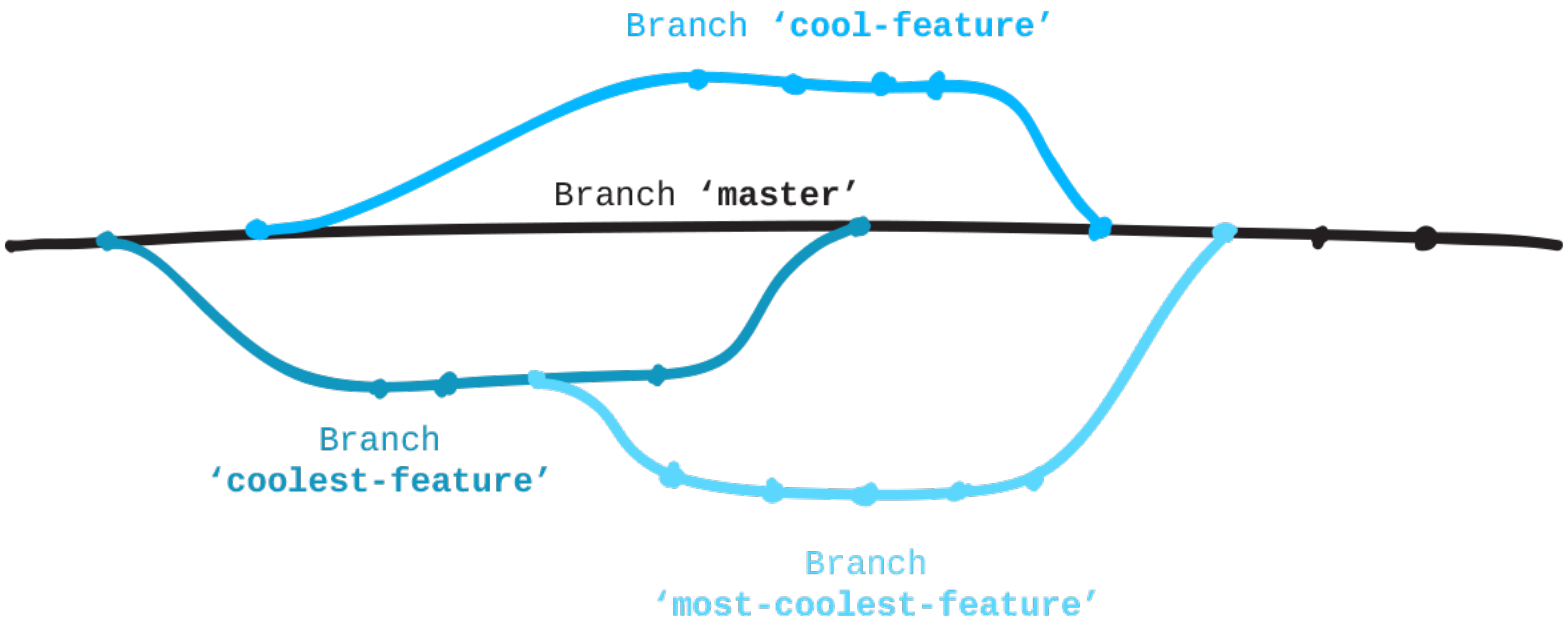
Basic Git commands

- `git add` will add new or changed files to the staging area
  - `git add --all` to add all new or changed files
- `git commit` creates a commit out of the staged changes
  - `git commit -m “notes here”` to commit with a short message
- `git push/pull <remote> <branch>` pushes or pulls commits from your local repo to a remote repo
  - E.g., `git push origin main`

A typical workflow

1. Fetch your teammates' changes from Github with `git pull`
2. Make changes to your local repo
3. Stage changes for commit with `git add`
4. Commit staged changes to your local repo with `git commit`
5. Push your changes to Github with `git push`
6. Repeat steps 1-5 and always `pull` before `push`ing!

Use branches to organize development



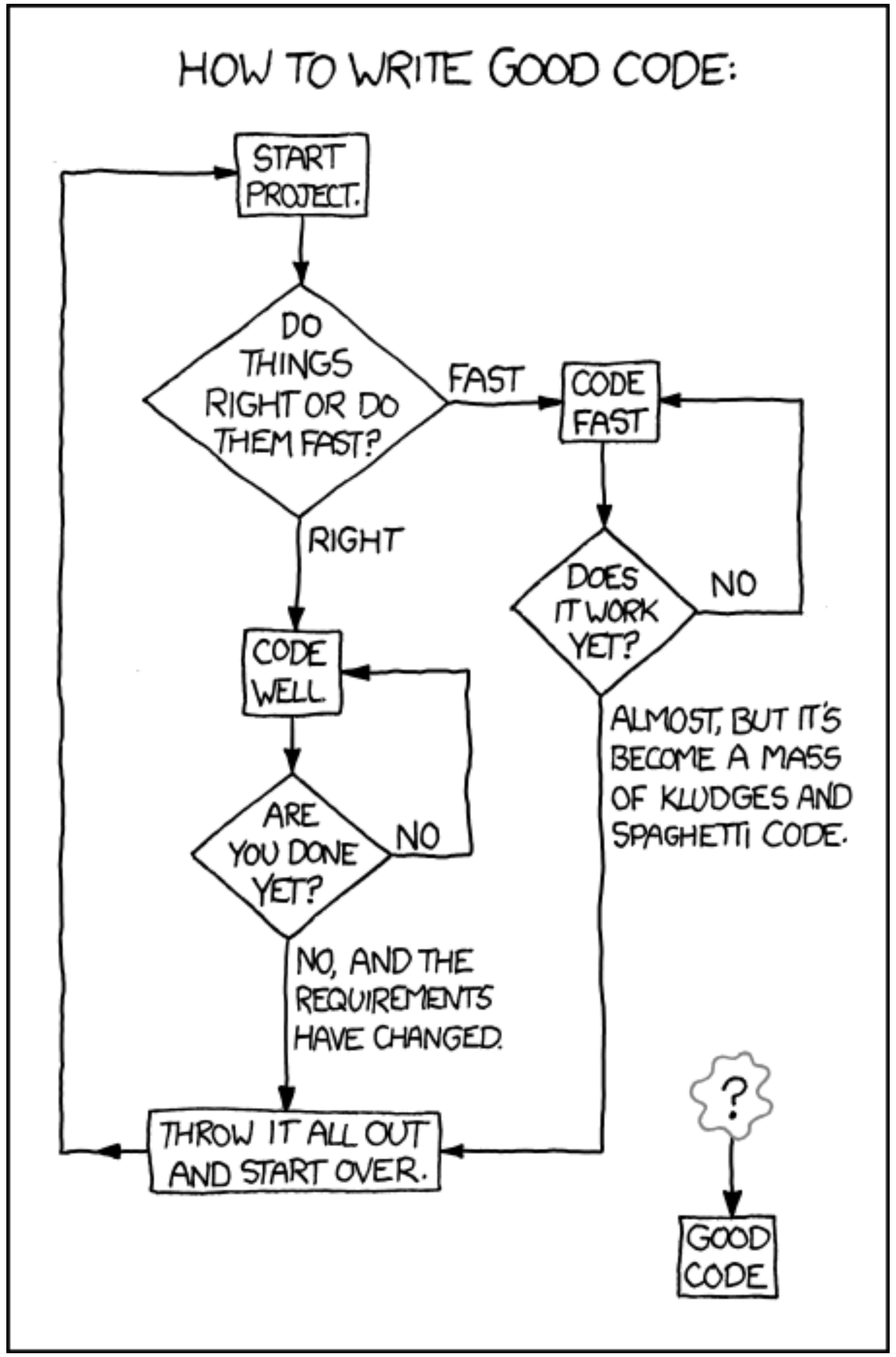
Basic branch commands

- `git branch` lists available branches
- `git checkout -b <name>` will create a new branch
- `git checkout <name>` switches to a different branch
- `git merge <name>` merges a branch into the current one
  - Commits from the other branch are copied into the current one
  - You may need to manually fix merge conflicts

TESTING CODE

Testing code

- Need to make sure **code works**
  - As intended
  - As expected
  - And stays that way
- Need to define requirements


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Unit testing

- Test a **<u>unit</u>** of code or functionality
  - Unit tests are *small*
  - Test a single requirement each
- Formalize code requirements
  - *Define* expected result
  - Test that code returns expected result

A simple unit test

```
def fact(n):
    """Factorial of n (i.e., n!)"""
    if n == 1:
        return n
    else:
        return n * fact(n-1)

assert fact(1) == 1 # test 1! = 1
```

More unit tests

```
def fact(n):
    """Factorial of n (i.e., n!)"""
    if n == 1:
        return n
    else:
        return n * fact(n-1)

assert fact(1) == 1 # test 1!

assert fact(2) == 2 # test 2!

assert fact(9) == 362880 # test 9!
```

Good unit tests should…

- Be small and self-contained
- Be automated and repeatable
- Be easy to implement
- Test a single unit of code
- Run quickly

Using unit tests

- Use to guide development
- Test current and future implementations
  - Run all unit tests after major and minor changes
  - Make sure nothing breaks
- Simplifies code maintenance

Designing a unit test

- Consider a unit of required functionality
- Create a test case
  - What is the required result
- Make it self-contained
  - Isolate setup and teardown

Testing with unittest

```
import unittest

def test_LList_instance():
    x = LList()
    x.append(1.11)
    x.append(2.22)
    x.append(3.33)
    return x

class TestLList(unittest.TestCase):

    def test_getitem(self):
        x = test_LList_instance()
        self.assertEquals(x[0], 1.11)
        self.assertEquals(x[1], 2.22)
        self.assertEquals(x[2], 3.33)
```

Testing with `unittest`

```python
import unittest

def test_LList_instance():
    x = LList()
    x.append(1.11)
    x.append(2.22)
    x.append(3.33)
    return x

class TestLList(unittest.TestCase):

    def test_getitem(self):
        x = test_LList_instance()
        self.assertEquals(x[0], 1.11)
        self.assertEquals(x[1], 2.22)
        self.assertEquals(x[2], 3.33)
```

Isolate setup

Create test case

Define a test

Testing frameworks

- Unit testing is a practice
- Various frameworks exist to automate creating and running tests…
  - `unittest`
  - `pytest`
  - `robot`, etc.
- Practicing unit testing is more important than the framework you choose

PACKAGES AND MODULES

Python modules

- File of Python code with filename ending in ".py"
- Collection of Python definitions and statements
  - **Decompose** complex codebase into collection of related functions
  - Easier to **re-use** and **maintain**
- Everything in a module shares a **similar purpose**

Using modules

- Save your module as "my_module.py"
- Import module for use in another script
- Objects from module referred to by alias

```
import mymodule
mymodule.myfunction()
```

Use module name as alias to prefix its functions

Import a module with an alias

- Save your module as "my_module.py"
- Import module for use in another script
- Objects from module referred to by alias

```
import mymodule as my

my.myfunction()
```

Specify a different alias to refer to module

Import specific objects from a module

- Save your module as "my_module.py"
- Import module for use in another script
- Import specific objects

```
from mymodule import myfunction
myfunction()
```

No alias needed for specific function imports

Standard library modules

- math
- random
- itertools
- string
- datetime
- os
- sys
- etc.

Python packages

- **Packages** may contain multiple related modules
- Organized as a directory of modules
  - /mypkg
    - __init__.py
    - mymodule1.py
    - mymodule2.py
    - mymodule3.py

Using packages

- Import module from a package
- Package hierarchy indicated by **dot notation**

```
import mypkg.mymodule1
mypkd.mymodule.myfunction()
```

Use package.module name as alias to prefix functions

Import from a package using an alias

- Import module from a package
- Package hierarchy indicated by **dot notation**

```
import mypkg.mymodule1 as my

my.myfunction()
```

Specify a different alias to refer to module

Import specific items from a package

- Import module from a package
- Package hierarchy indicated by **dot notation**

```
from mypkg.mymodule1 import myfunction
myfunction()
```

No alias needed for specific function imports

Creating a package

- Create a package directory
  - /mypkg
    - __init__.py
    - mymodule1.py
    - mymodule2.py
    - mymodule3.py
- Include __init__.py
  - Indicates this directory is a Python package
  - Does not need to include anything

Creating a package (2)

- A package may contain subpackages
    - /pkg
        - /subpkg1
            - __init__.py
            - module1A.py
            - module1B.py
        - /subpkg2
            - __init__.py
            - module2A.py
            - module2Bpy

Creating a package (2)

- A package may contain subpackages
  - /pkg
    - /subpkg1
      - __init__.py
      - module1A.py
      - module1B.py
    - /subpkg2
      - __init__.py
      - module2A.py
      - module2Bpy

How to import between subpackages?

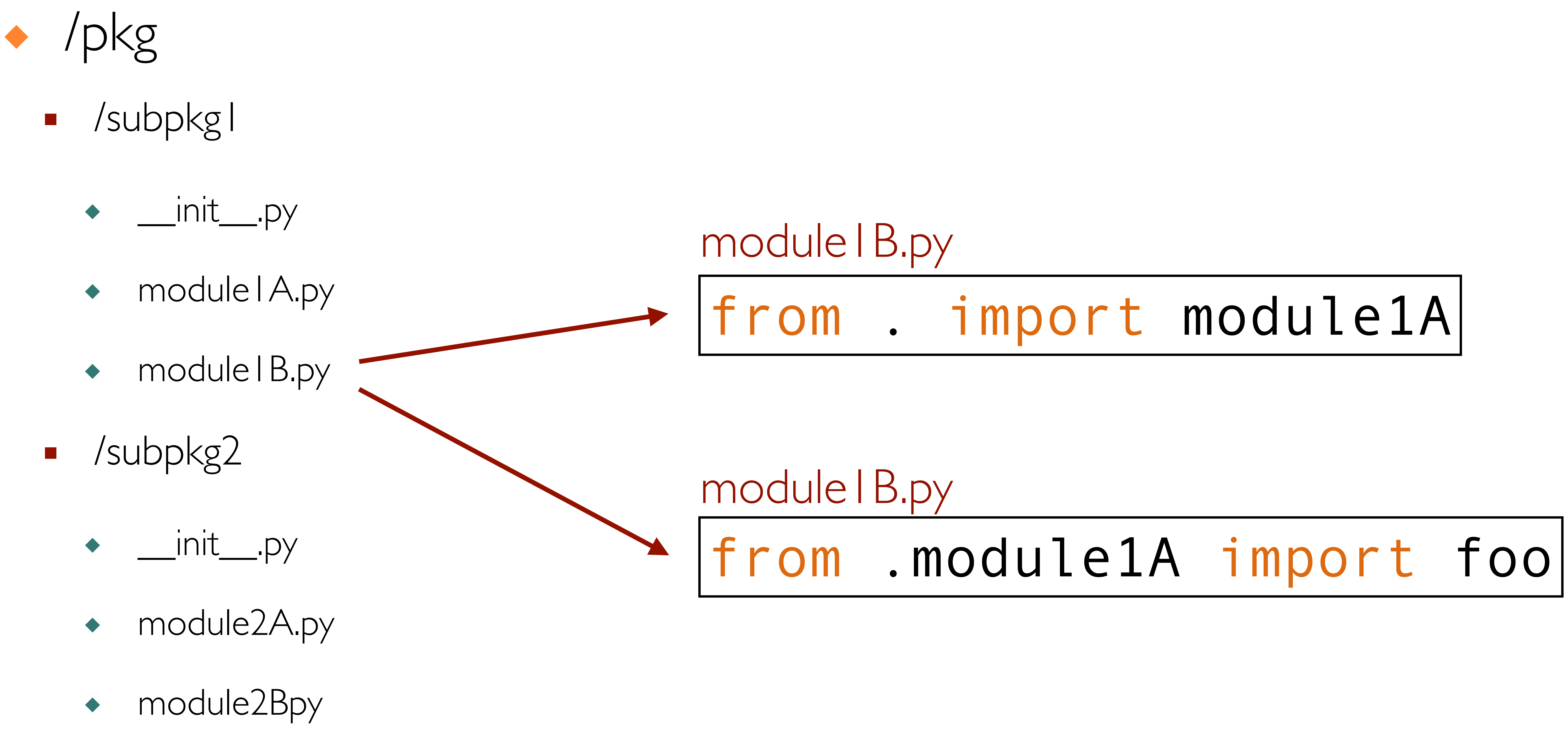
Relative imports

- Relative imports are used within a package
  - Useful to import between subpackages
  - Necessary if module shares name with standard library
- Use dot syntax:

```
from . import module1 # import from this package
from .module1 import foo # import from this package
from .. import module2 # import from sibling  subpackage
```

Importing a sibling module

- Use . to indicate the same subpackage



- /pkg
  - /subpkg1
    - __init__.py
    - module1A.py
    - module1B.py
  - /subpkg2
    - __init__.py
    - module2A.py
    - module2Bpy

module1B.py

```
from . import module1A
```

module1B.py

```
from .module1A import foo
```

Importing from a sibling sub package

- Use .. to indicate the parent package directory
  - /pkg
    - /subpkg1
      - __init__.py
      - module1A.py
      - module1B.py
    - /subpkg2
      - __init__.py
      - module2A.py
      - module2Bpy

module2A.py

```
from ..subpkg1 import module1A
```

Including unit tests

- Include a package directory of unit tests
  - /pkg
    - __init__.py
    - module1.py
    - module1.py
    - tests/
      - __init__.py
      - test_module1.py
      - test_module2.py

test_module1.py

```
from ..module1 import foo
```

EXAMPLE PACKAGE:

https://github.com/kuwisdelu/containers