Software Engineering

for Scientists & Engineers

ABOUT ENTHOUGHT

Enthought Quick Facts

FOUNDED
2001

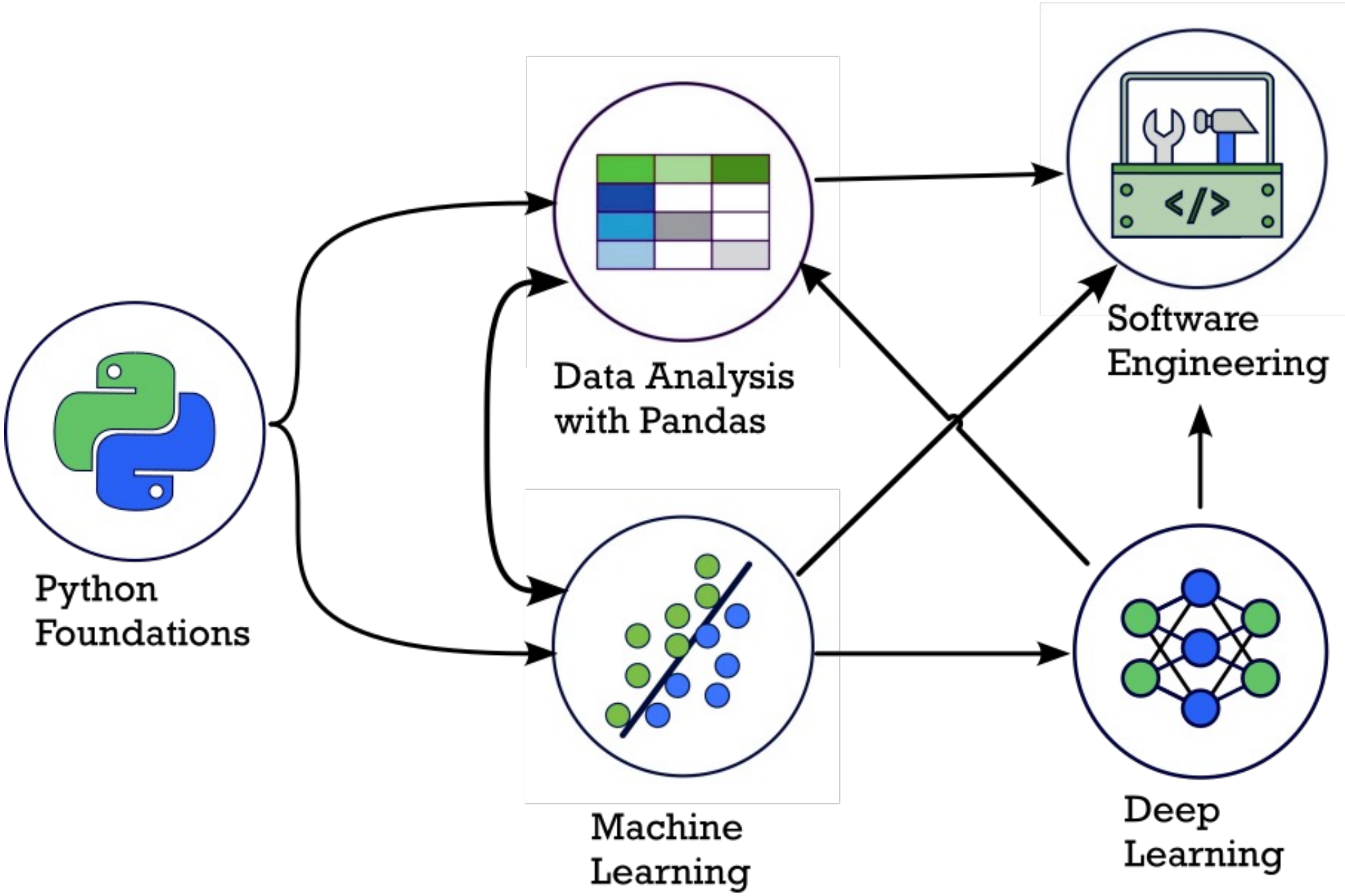
FOCUS
Digital Transformation for Science Companies

TECHNICAL TEAM PROFILE

85% Advanced Degrees

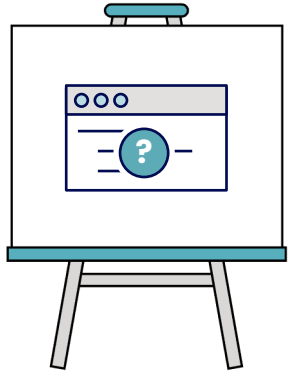
70% PhDs

GLOBAL OFFICES


Python
Foundations
Data Analysis
with Pandas
Machine
Learning
Software
Engineering
Deep
Learning

Syllabus

1. Why Software Engineering?
2. Readable Code
3. Documenting Code
4. Refactoring
5. Monitoring Execution
6. Profiling & Debugging
7. Unit Testing
8. Source Control
9. Effective Code Reviews
10. Development Models

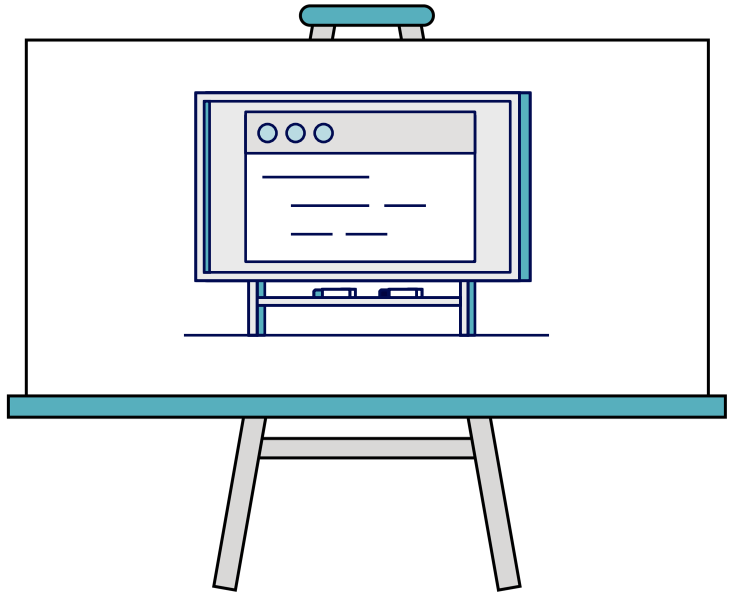


Lecture 01

Why Software Engineering?

Software Engineering for Scientists and Engineers

Table of Contents



Introduction

Lecture 01
Why Software Engineering?

Course Format

Unlike most our other courses, this course is being taught primarily as a **seminar**. While there will be technical components that involve programming in some lectures, they are there to present **ideas**, not to train you with specific tools or what the proper syntax is.

- Please **converse** with the instructor and the rest of the class. (Yes, that means to switch on your microphone and video unless you are specifically prevented from doing so by your work security requirements).
- **Questions** and **discussions** are where the learning will happen.
- Treat all exercises as **demos**. Think about the ideas behind the exercises, not on the technicalities of correct syntax. You can come back to matters of syntax later.

Introductions

After the instructor's introduction, please be ready introduce yourself to your classmates.

1. What is your name?
2. What computer language(s) do you program in?
3. How many years of experience?
4. Do you primarily work alone or with a team?
5. Do you collaborate with others in programming?
6. Do you manage others in programming?
7. Why did you sign up for Software Engineering for Scientists and Engineers?

Discussion

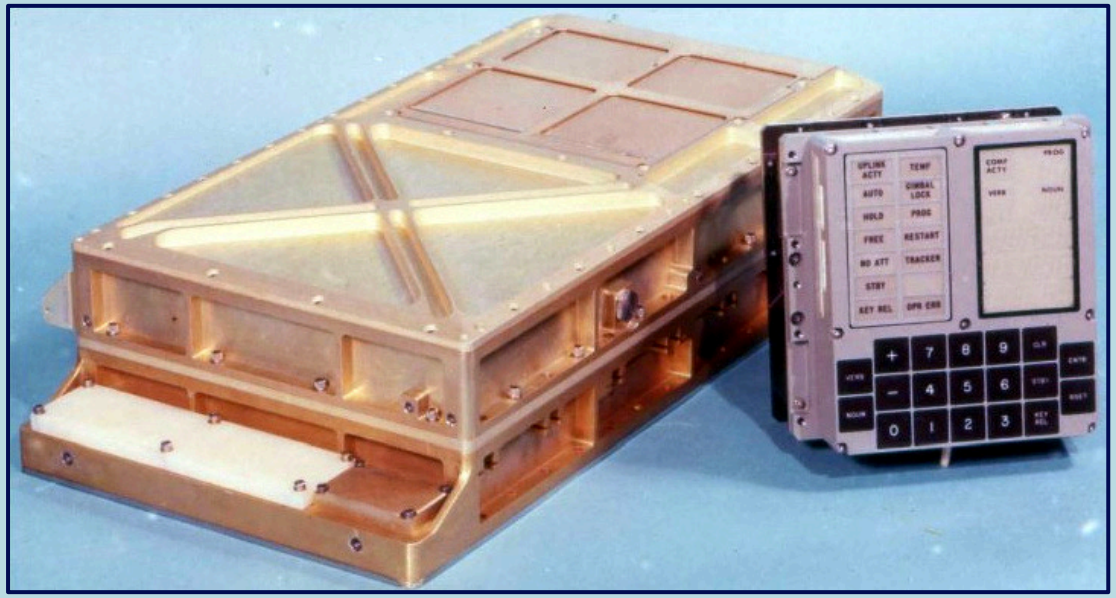
What is Software Engineering?

Apollo Guidance Computer

I fought to bring the software legitimacy so that it—and those building it—would be given its due respect and thus I began to use the term 'software engineering' to distinguish it from hardware and other kinds of engineering, yet treat each type of engineering as part of the overall systems engineering process.

When I first started using this phrase, it was considered to be quite amusing. It was an ongoing joke for a long time. They liked to kid me about my radical ideas. Software eventually and necessarily gained the same respect as any other discipline.

— *Margaret Hamilton, 2014 interview with El País*



Software Engineering Body of Knowledge (SWEBOK)

- Origins in the 1960s.
  - Apollo Guidance Computer
  - NATO Science Committee

- Evolved into SWEBOK
  - Maintained by the IEEE Computer Society
  - Organized as Knowledge Areas (KAs); some shown to the right.

- Now stand-alone discipline in universities.

Requirements

Design

Construction

Maintenance

Quality

Testing

Engineering Management

Configuration Management

Google's Definition

"Software engineering is programming integrated over time."

- Separation of software development and engineering.
  - Programming is the creation of software.
  - Software engineering is the set of policies, practices, and tools that do two things:
    1. make team collaboration possible
    2. keep code useful over its lifetime

https://abseil.io/resources/swe-book/html/ch01.html

Policies

Practices

Tools

An Informal Approach

- Make it work.
- Make it right.
- Make it fast.

Always keeping in mind that:

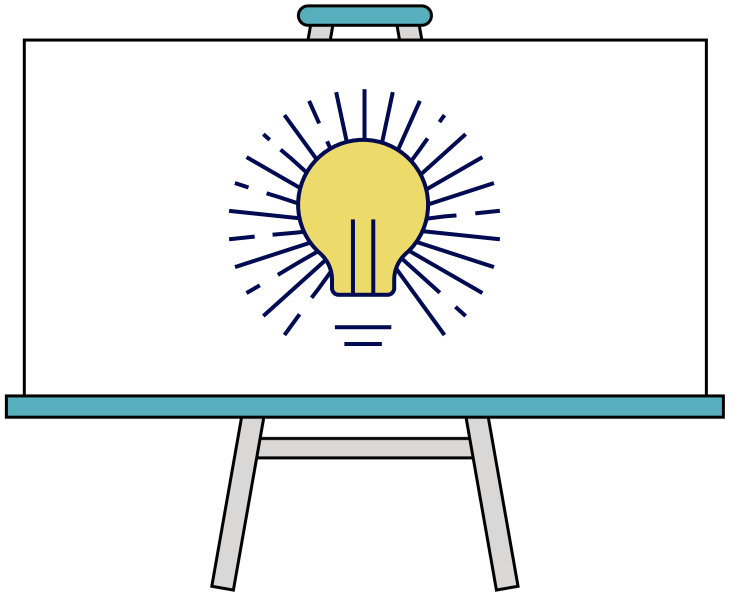
"Engineering is an iterative process."

– Mark Rober, former NASA engineer

A Working Definition

Software engineering is a systematic approach to software development with the following characteristics:

- Draws on well-known patterns (best practices) from past software projects.
- Continuously improves software by learning from past mistakes (has a strong feedback loop).
- Has a standard set of working tools that all practitioners should know and use.



Key Concepts

Lecture 01
Why Software Engineering?

Usable

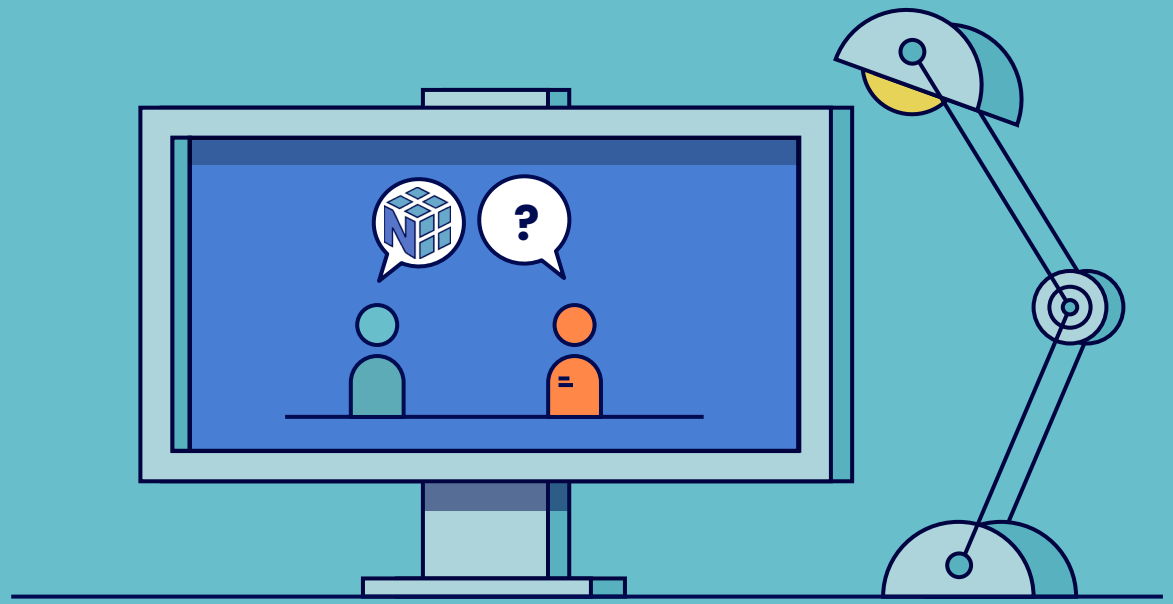
What makes software usable?

- Good Names
- Consistent Design
- Documentation

Sometimes what makes software usable depends on the **audience**.

- Is it written for other programmers to use?
- Is it written for end-users?

Discussion: What makes NumPy usable?



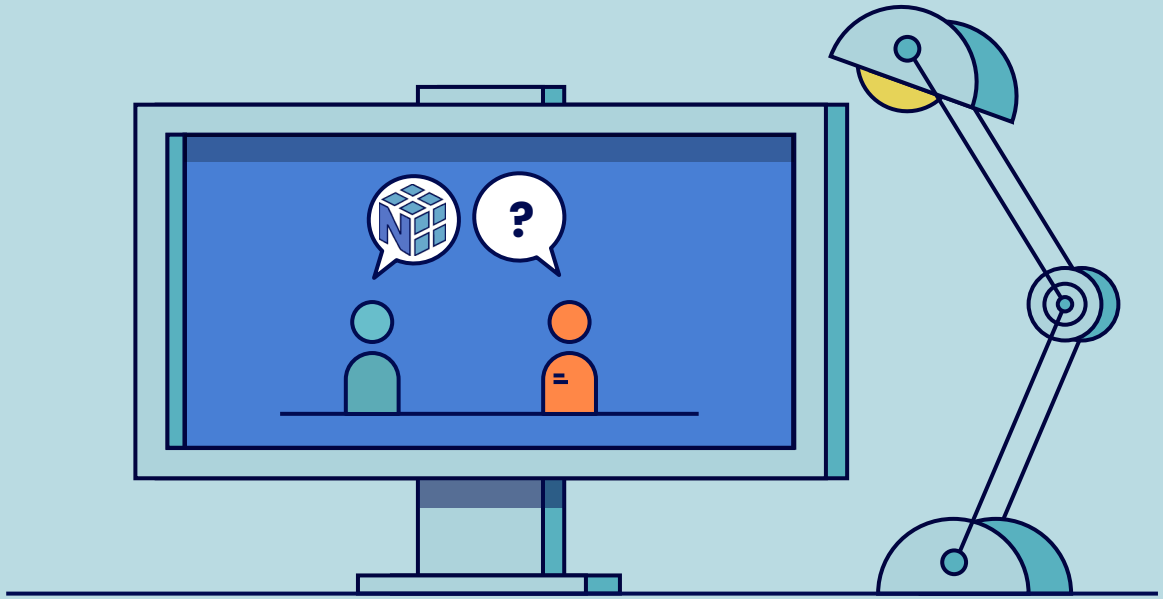
Maintainable

What makes software maintainable?

- Good Names
- Consistent Design
- Self-Documentation (API)
- Meta-Documentation (Architectural Explanations)
- Contained Complexity
- Transparent Dependencies

A lot of the elements from usability apply here. Software that is not usable is usually hard to maintain.

Discussion:
What makes NumPy maintainable?

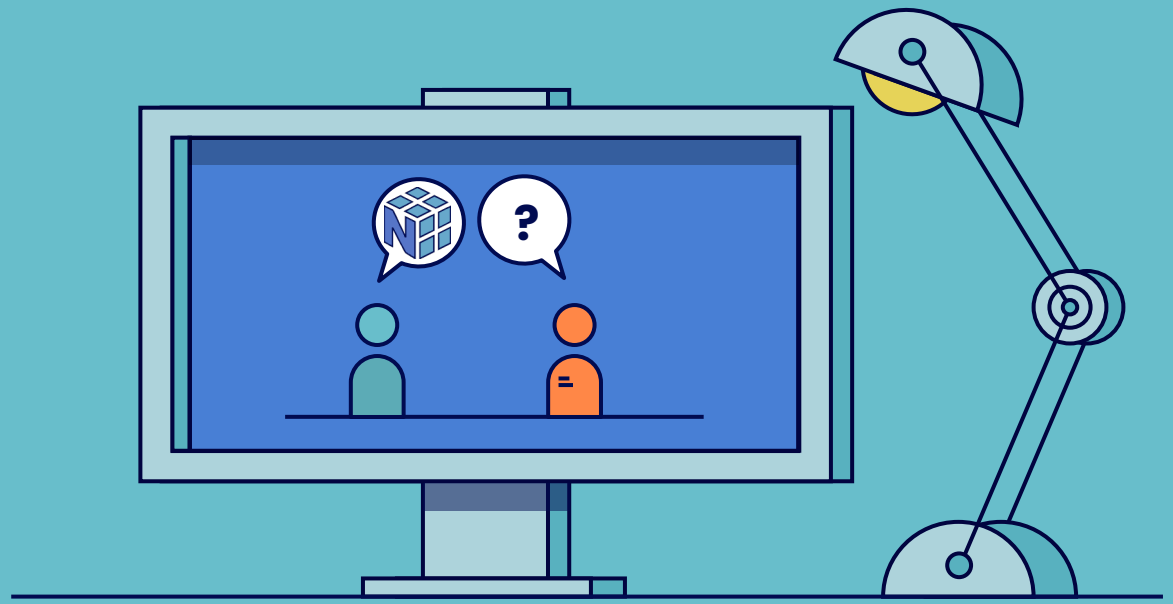


Reliable

What makes software reliable?

- As **simple** as possible; only as **complex** as needed.
  - Built from simple, easily understood components.
  - Complexity comes from orchestrating simple parts.
- Design with **edge cases** in mind
- Testing
- Feedback from **customers**

Discussion:
What makes NumPy reliable?

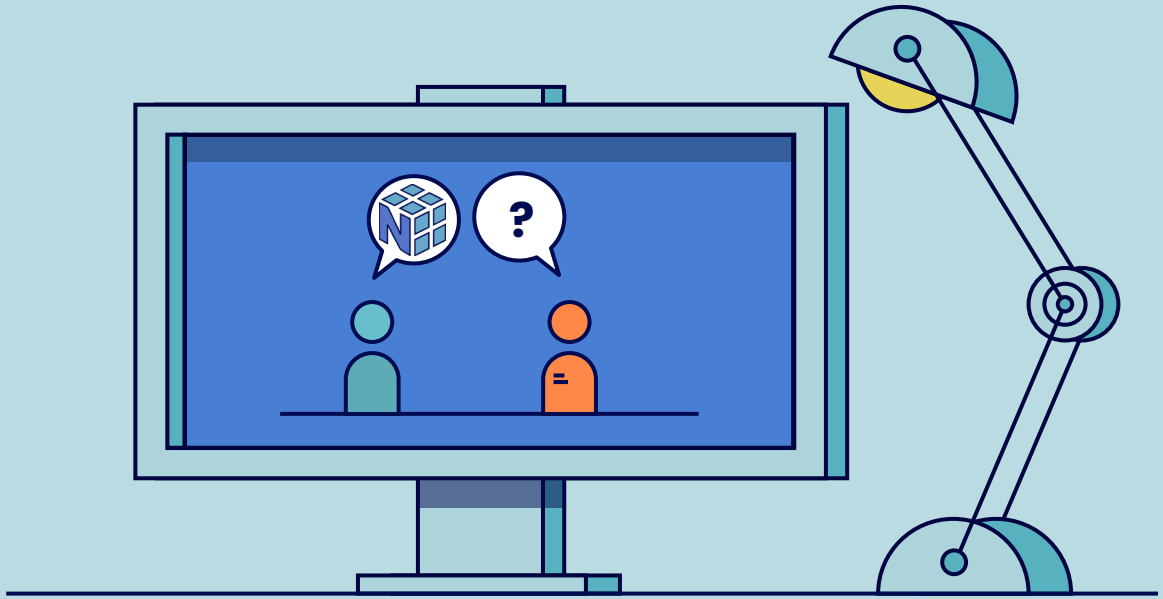


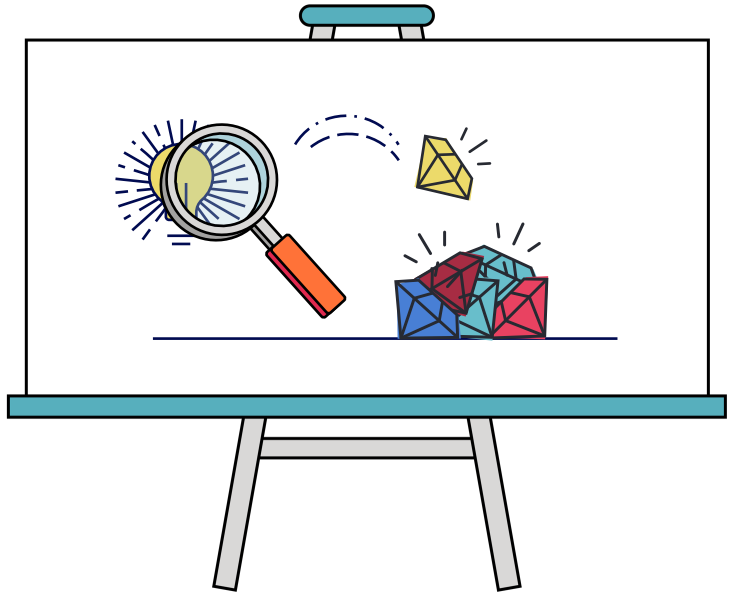
Scalable

What makes software scalable?

- Having all of the attributes of usable, maintainable, and reliable.
- Scalable architectures
  - In practice this often means making the best use of standard specialist components to provide standard services
- Performance testing

Discussion:
What makes NumPy scalable?





The R&D Environment

Lecture 01
Why Software Engineering?

What Makes the R&D Environment Different?

In most R&D settings, software development is a **bottom-up process**.

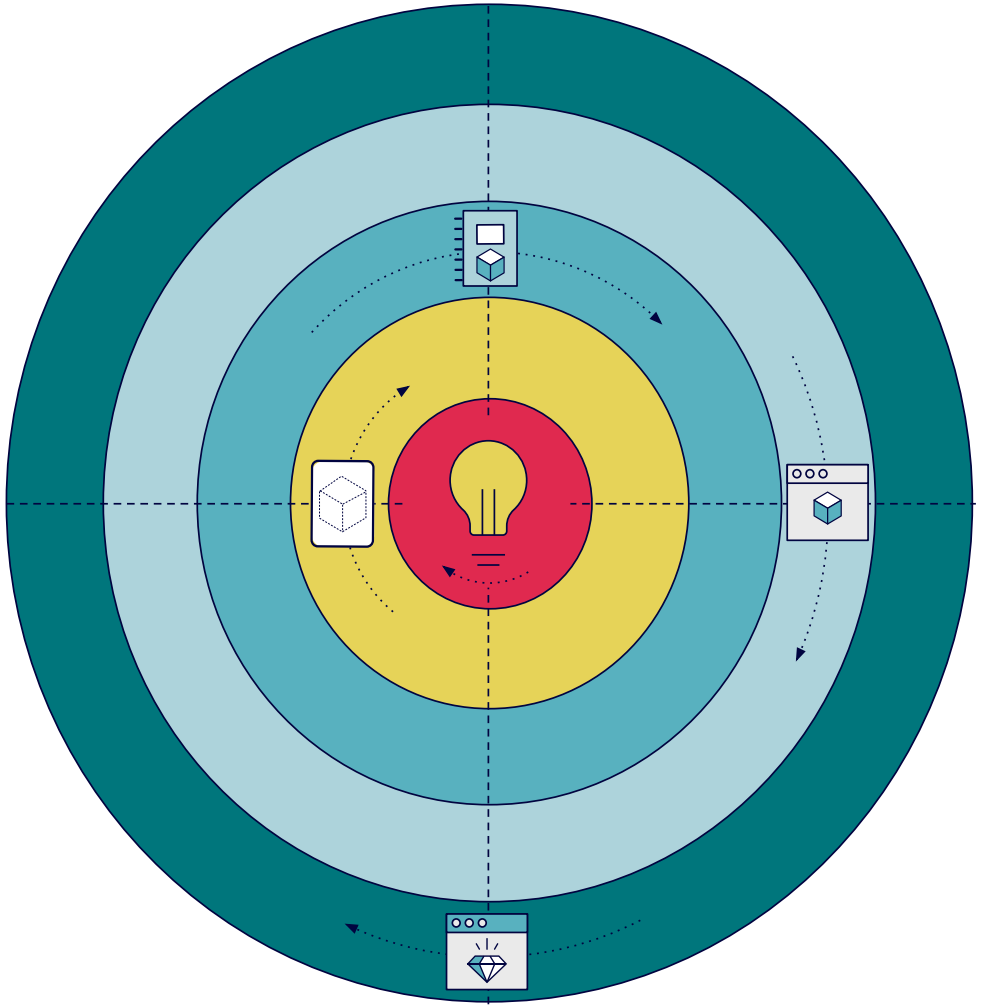
- Software starts at the **individual** researcher level.
- If it proves interesting or useful, it gets developed further and (potentially) **shared** with colleagues.
- Only the most useful ideas survive long-term to grow to complex **applications** used by (potentially) many researchers.

Iterative R&D Software Engineering Cycle

Software Idea → Code Experiment
Code Experiment → Jupyter Notebook
Jupyter Notebook → Script
Script → Application

As each transition takes place:

- complexity increases
- time passes
- potentially more developers touch the code
- potentially more users rely on the code



Order of R&D Focus

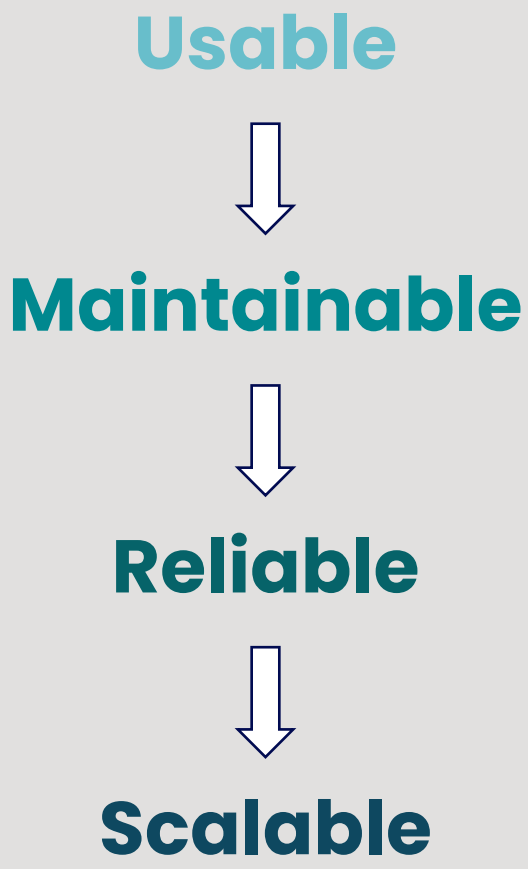
Don't try to do too much at once.

First master **usable**. New programmers and new team members should focus here first. It pays off the most in the long-run.

Next master **maintainable**. The elements of maintainable can be learned; with experience they can become instinctual.

Reliable emerges over time – with experienced team members and understanding of problem being solved.

Focus last on **scalable**. With modern computers, scalability is generally needed only for massive amounts of data or huge numbers of users.



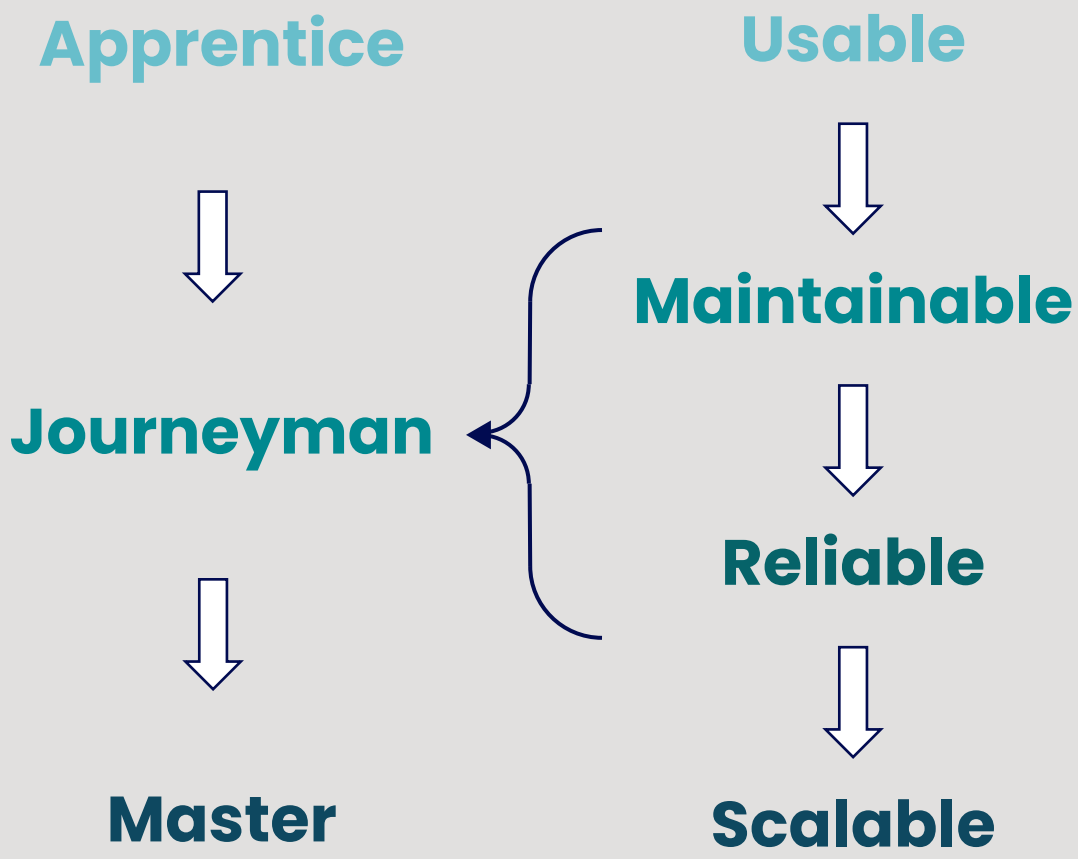
Skill Matures

Programming skill matures through study and experience. Don't expect a new programmer to have all of the right habits and instincts from the get go.

Apprentice programmers (and their managers) should focus first on **usability** as a first milestone in their development work.

Journeyman programmers should be focused on **maintainability** and **reliability**.

Finally, master programmers should be able to think architecturally with **scalability** in mind, but only where needed.



Sources of Complexity

Complexity in the R&D software environment typically comes from the following sources:

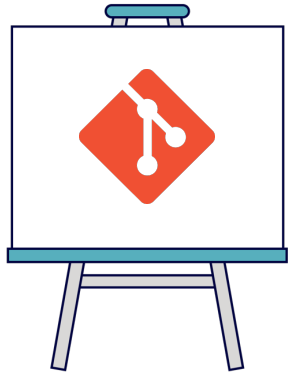
- **Collaboration**: How many people are actively coding a project?
- **Turnover**: Do you expect to have a lot of turnover in project personnel?
- **Time**: How long do you expect a given piece of code to be in use?
- **Downstream Impact**: Is your code foundational to other projects?
- **Backwards Compatibility**: Does someone external to your group rely on your code?
- **Too Many Features**: Trying to do too much with one tool.

Science is the Product

Unless you work in an unusual R&D environment, **science** – not software – **is the final product.**

Software engineering practices should be taken *cum grano salis* – **adopt the ones that make the science more efficient**, ignore the rest.

This course focuses on the software engineering practices that Enthought has found useful in its consulting practice (trying to make R&D workflows more efficient). The ideas and tools presented here are the ones that we have found to be most useful.



Lecture 08

Source Control

Software Engineering for Scientists and Engineers

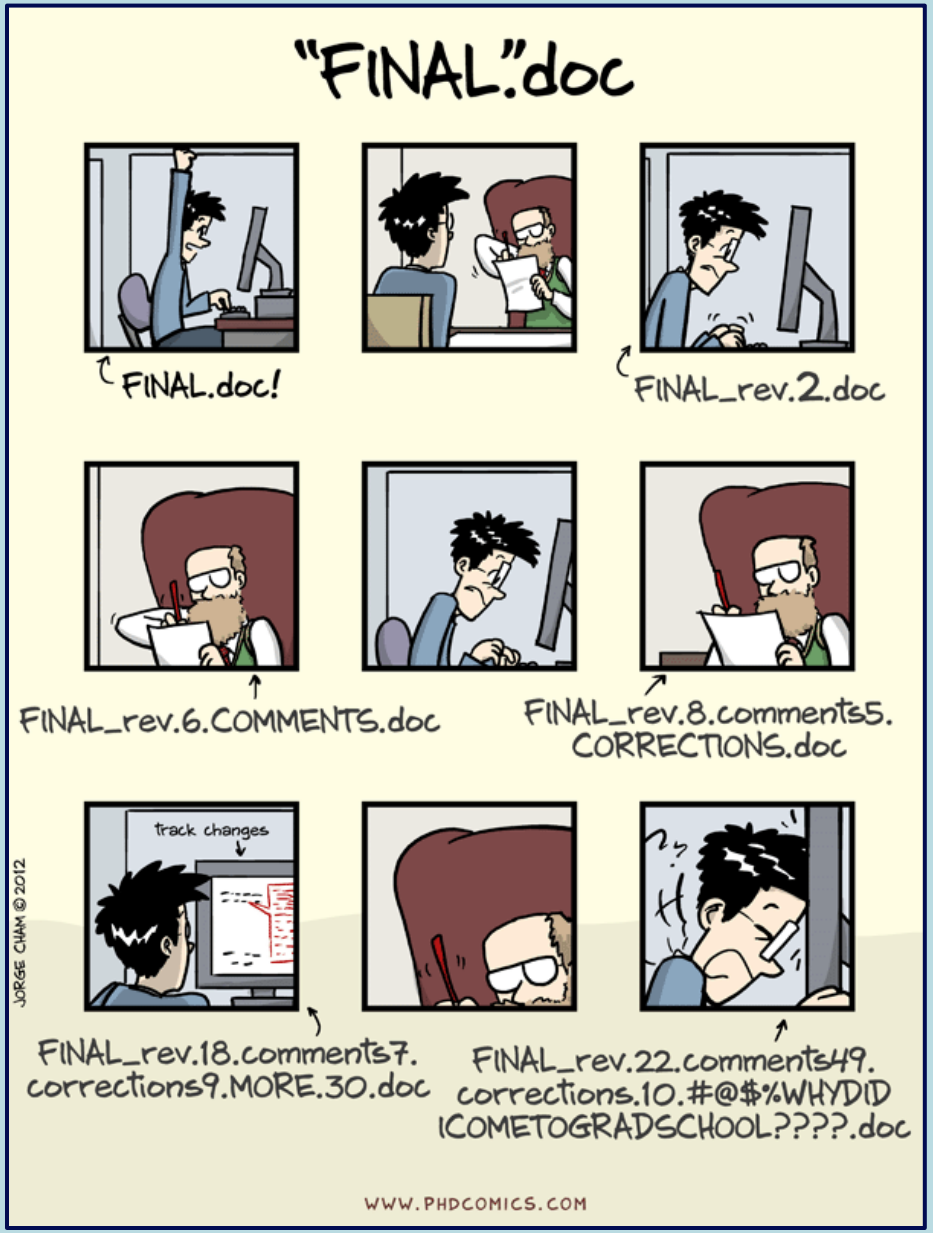
Table of Contents

Introduction

Lecture 08
Source Control

Why Use Version Control?

- Version Control (or Source Control) is a way to track and manage changes to *source code*.
  - See a full history of a code base (who changed what at a given time)
  - Can roll back versions as needed
- Likely that teams are doing this in some form or fashion already
  - Ad-hoc, fragile, error-prone, and time-consuming
- Let a computer do the hard work and mange changes through a Version Control System (VCS)
  - Git
  - Subversion
  - Mercurial
  - ClearCase
  - ...

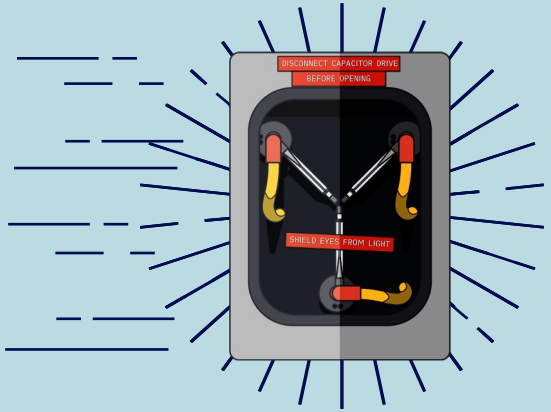


https://phdcomics.com/comics/archive.php?comicid=1531

What is a Version Control System?

- **Version Control Systems**
  - Software for managing the evolution of applications
  - Designed to help coordinate multiple people working on the same codebase
  - Also useful for projects you work on alone
- **Version Control Systems provide the following services:**
  - Tracking the history of changes to files
  - Retrieving previous versions of files
  - Protecting against incompatible changes to a file
- **Ideally, when using a Version Control System (VCS):**
  - You will not lose anything you created
  - If something breaks you can go back to the last working version

Back to the ~~future~~ *Past*



What is Git?

- Git is an *open-source, distributed* Version Control System
  - A personal project of Linus Torvald (the Linux Linus), first released in 2005
  - One of the most popular Version Control Systems (VCS) available today

- Design Goals
  - Fast and lightweight interface
  - Distributed source database
  - Many concurrent users

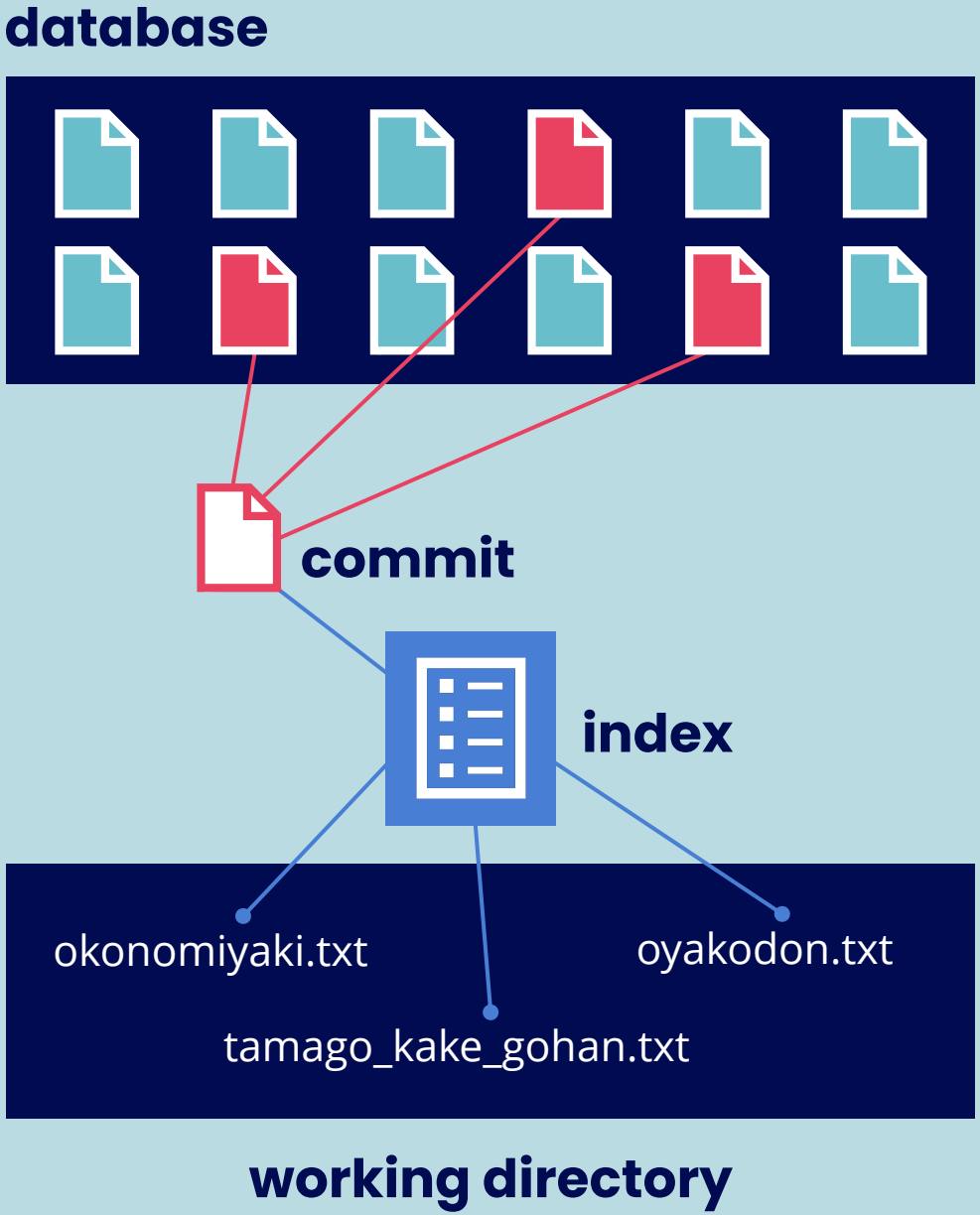
- Downside of being largely nonintuitive even to most experienced users

- Good news is that 90% of the benefit of using Git can be acquired while only understanding 10% of the underlying mechanisms

By Krd (photo)Von Sprat (crop/extraction) - File:LinuxCon Europe Linus Torvalds 03.jpg, CC BY-SA 4.0

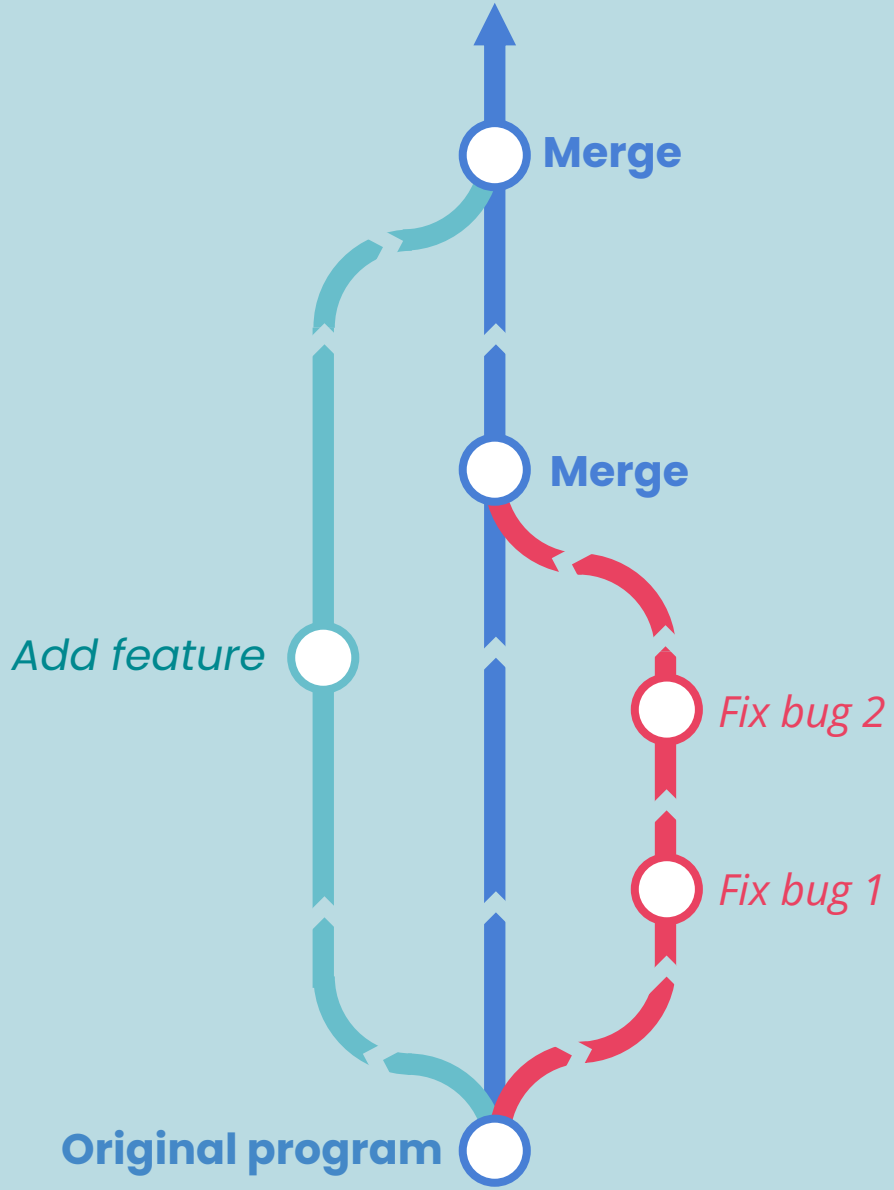
How Does Git Work?

- Git stores an entire file (object) in a database every time a person changes any part of it
- Moments in time (commits) are also stored in this database, and consist of the database location of each file at its last change
- The entire history of a file is recoverable because every moment in time (commit) knows about the moment in time (commit) just before it
- The location of each file in the filesystem (not in Git's database) is stored in a special file (index) and Git will only track changes to files added to the index



Basic Workflow

- Changes to files are made on "branches"
- A "commit" is used to finalize each set of changes
- The initial "commit" is on a branch named "main" (or "master")
- Branches can be merged



Getting Started with Git

- There are three steps to getting started
  - Install Git
  - Configure Git
  - Initialize Git
- On every new computer, Git needs to be *installed* and *configured* (just once)
- For every collection of files that should be kept together, Git needs to be *initialized* in that directory
  - This directory is often referred to as a repository ("repo")

```
# Enter the following
# commands after starting "git bash"
# application that is part of
# Git for Windows

# Step 1: Verify Git Installed
$ git --version
git version 2.39.0

# Step 2: Configure
$ git config --global user.name "Your Name"
$ git config --global user.email "user@company.com"
$ git config --global core.editor "vim"

# Step 3: Initialize Git within a repo
$ cd Desktop/my_repo
$ git init
```

Git allows you to specify a default editor as well. **`git config --global core.editor "code --wait"`** can be used to set VS Code as the editor. **`git config --global core.editor "vim"`** can be used to set the editor to Vim.

Example

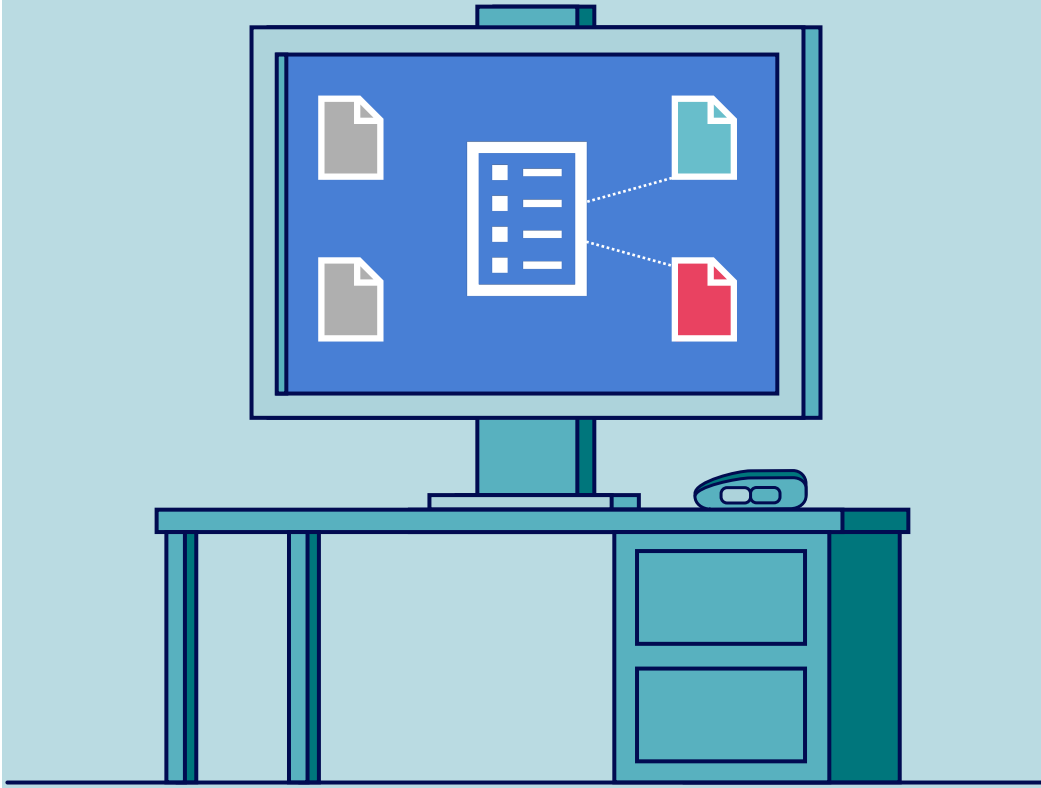
```
$ # Step 1: Verify Git Installed
$ git --version
git version 2.39.0

$ # Step 2: Configure
$ git config --global user.name "Logan Thomas"
$ git config --global user.email "lthomas@e.com"
$ git config --list
user.name=Logan Thomas
user.email=lthomas@e.com
core.editor=vim

$ # Step 3: Create repository called notes
$ cd ~/Desktop
$ mkdir notes
$ cd notes
$ git init
```

Give it a try! Create a Simple Repository

1. Open a terminal or command prompt and verify Git has been installed by using the command **`git --version`**.
2. Configure Git with your name and email. Check that everything is correct using the command **`git config --list`.**
3. Create a folder named "notes"
4. Initialize that folder's Git repository using the command **`git init`.**

Files States

- To have Git track a certain file, it needs to be added to the database index
- To do, use the command
  `git add <filename>`
- To undo this action, use the command
  `git reset`
- To save something that was added (a new file or a modification to an existing file), use the command
  `git commit`
- Commits record
  - A timestamp
  - Name and Email of committer
  - Message that explains what changes were made
  - Snapshot of the files in the index



File States Example

- Create a new To-Do list called **`todos.txt`** and add an item to email co-worker Sheldon
- To see which files are being tracked by Git and which files have been "staged", use the command **`git status`**
  - Notice that the **`todos.txt`** is an Untracked file (haven't told Git to pay attention to it yet)
- Tell Git to track the **`todos.txt`** file by **adding** it to the *database index*
  - Notice that the **`todos.txt`** is now being track
  - This is also called "staging" for a file that is already being tracked
- Tell Git to stop tracking the **`todos.txt`** file with **`git reset`**

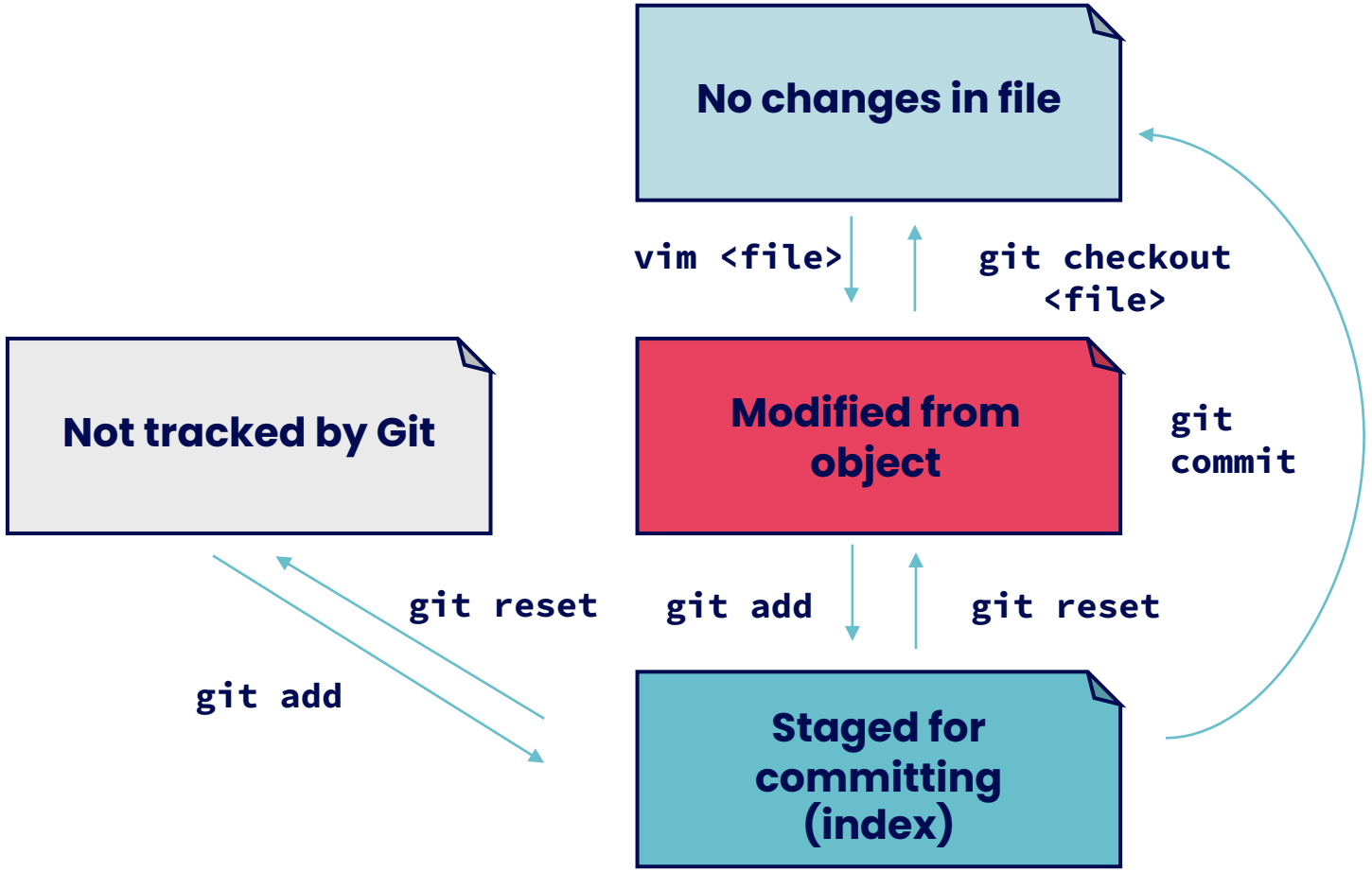
```
$ cd Destkop/notes

$ # Can use any text editor to create and add a line
$ # in new text file titled "todos.txt"
$ echo "- email sheldon about release" > todos.txt

$ git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be
committed)
        todos.txt
nothing added to commit but untracked files present (use "git
add" to track)

$ git add todos.txt

$ git status
On branch main
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   todos.txt

$ git reset
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be
committed)
        todos.txt
nothing added to commit but untracked files present (use "git
add" to track)
```

File States Example

- Ensure Git is tracking the **`todo.txt`** file
- Commit the newly created **`todo.txt`** file using the **`git commit`** command
  - The **`-m`** option allows a message to be attached describing the change
- Update **`todo.txt`** to include a new item on the To-Do list
  - Notice that Git now sees this as a change because it is tracking the file
- To show differences between file states, use the **`git diff`** command
  - If changes have been staged via the git add command, use **`git diff --staged`** to show differences

```
$ git add todos.txt
$ git commit -m "Initial commit"
[main (root-commit) 125b064] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 todos.txt

$ git status
On branch main
nothing to commit, working tree clean

$ # Can use any text editor to add a new line to "todos.txt"
$ echo "- submit issue for new found bug" >> todos.txt

$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   todos.txt

no changes added to commit (use "git add" and/or "git commit
-a")

$ git diff
diff --git a/todos.txt b/todos.txt
index 1635beb..f7d28fd 100644
--- a/todos.txt
+++ b/todos.txt
@@ -1 +1,2 @@
 - email sheldon about release
+- submit issue for new found bug

$ git commit -m "add reminder to submit issue"
[main ef3c112] add reminder to submit issue
 1 file changed, 1 insertion(+)
```

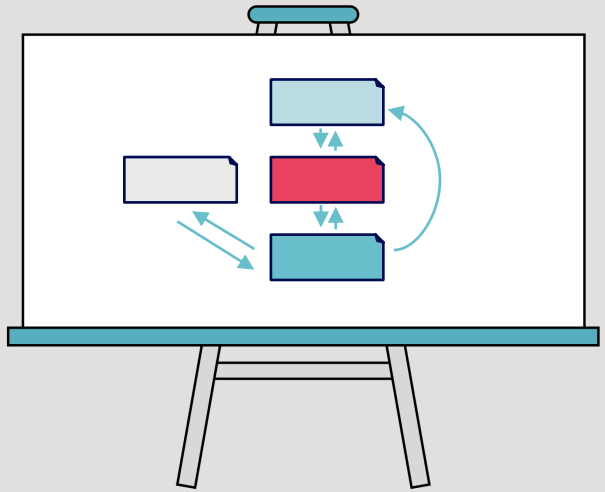
The Possible States of a File



Even though files may be tracked by Git, a commit still needs to be made to "preserve" the changes. **It is not automatic**.

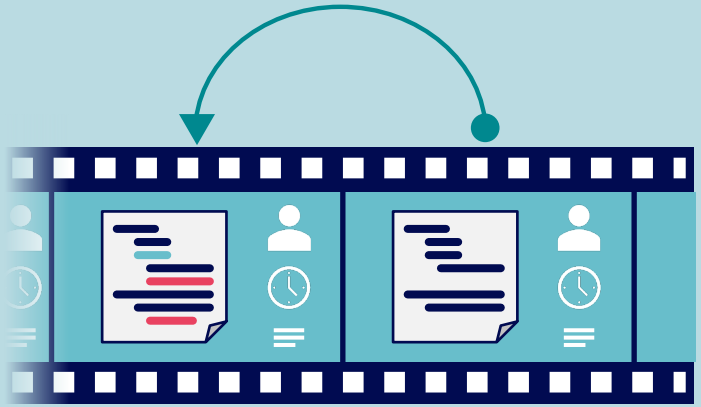
Give it a try! Check File States

1. Using any text editor of choice (or the command line), create a your own **`todos.txt`** file. Add a reminder to schedule a call with Leonard.
2. What is the output of **`git status`**?
3. What is the output of **`git diff`**?
4. What is the output of **`git diff --staged`**?
5. Repeat steps 2 to 4 after executing the command **`git add todos.txt`**
6. Repeat steps 2 to 4 after executing the command **`git commit -m "add reminder to schedule time with leonard"`**



What is a Commit?

- A commit is an object that holds **two** crucial pieces of information
  - Tree: the state of the file(s) at one moment in time
  - Parent: the state of the file(s) before this one moment in time
- **Commits are like a single frame in a movie**
  - A picture (snapshot) of the state of files at a single moment in time (the tree)
  - Every commit knows which one came before it (the parent) which allows traveling backward in time to previous versions of the file(s)
- **Additional metadata is stored in the commit**
  - Identity of person who made the commit
  - Timestamp of the commit
  - A message explaining why the commit was made



Commit Example

- When making a commit, a unique identifier (hash) is reported (**`125b064`** or **`ef3c112`**)
  - This is typically a much longer alpha-numeric string (SHA-1 hash) but typically the first 7 characters are used for referencing
- The *commit* hash identifies the current commit with all its history.
- The *parent* hash is a pointer to the parent of the current commit.
- The *tree* hash captures the state of the whole directory tree of all the files in the repository.

```
# From previous slide
# $ git add todos.txt
# $ git commit -m "Initial commit"
# [main (root-commit) 125b064] Initial commit
#  1 file changed, 1 insertion(+)
#  create mode 100644 todos.txt

$ git cat-file -p 125b064
tree 4384777828909 21cf05a61719966df748b8f172a
author Logan Thomas <lthomas@e.com> 1680889939 -0500
committer Logan Thomas <lthomas@e.com> 1680889939 -
0500

Initial commit

# From previous slide
# $ git commit -m "add reminder to submit issue"
# [main ef3c112] add reminder to submit issue
#  1 file changed, 1 insertion(+)

$ git cat-file -p ef3c112
tree a292413c8533b331ceedb2683278f7671e08e86c
parent 125b064337acc9b62c35c0030032be9251f6a83e
author Logan Thomas <lthomas@e.com> 1680892690 -0500
committer Logan Thomas <lthomas@e.com> 1680892690 -
0500

add reminder to submit issue
```

Viewing History

- Because each commit stores the location of the commit just before it, they can be used to trace back in time
- To see each commit through time use the command **`git log`**
- The **`(HEAD -> main)`** text identifies where the **`HEAD`**, or current state, is on the **`main`** branch
  - **`HEAD`** refers to the currently checked-out branch's latest commit
- Other useful history commands:
  - **`git log --pretty=raw`**
  - **`git log --graph \`**
    `--oneline \`
    `--decorate \`
    `--all`

```
$ git log
commit ef3c112cda2a5427b0a2bfddfccbcdd95f99afb1 (HEAD -> main)
Author: Logan Thomas <lthomas@e.com>
Date:   Fri Apr 7 13:38:10 2023 -0500

    add reminder to submit issue

commit 125b064337acc9b62c35c0030032be9251f6a83e
Author: Logan Thomas <lthomas@e.com>
Date:   Fri Apr 7 12:52:19 2023 -0500

    Initial commit

# Use `--online` for history of commits
# with shortened filename and message
$ git log --oneline
ef3c112 (HEAD -> main) add reminder to submit issue
125b064 Initial commit

$ git log --graph \
$         --abbrev-commit \
$         --decorate \
$         --date=relative \
$         --all
* commit ef3c112 (HEAD -> main)
| Author: Logan Thomas <lthomas@e.com>
| Date:   2 hours ago
|
|     add reminder to submit issue
|
* commit 125b064
  Author: Logan Thomas <lthomas@e.com>
  Date:   2 hours ago

      Initial commit
```

Comparing History

- To see all the changes since a particular commit, use **`git diff <hash>`**
- To see the changes in just one file (useful if there were many files changed), use **`git diff <hash> -- <filename>`**
- To retrieve a file from the history, use the command **`git checkout <hash> -- <filename>`**

```
$ git log --oneline
ef3c112 (HEAD -> main) add reminder to submit issue
125b064 Initial commit

# The current state (HEAD) is at commit ef3c112 so no changes
$ git diff ef3c112

$ git diff 125b064
diff --git a/todos.txt b/todos.txt
index 1635beb..f7d28fd 100644
--- a/todos.txt
+++ b/todos.txt
@@ -1 +1,2 @@
 - email sheldon about release
+- submit issue for new found bug


$ git checkout 125b064 -- todos.txt

$ git diff ef3c112
diff --git a/todos.txt b/todos.txt
index f7d28fd..1635beb 100644
--- a/todos.txt
+++ b/todos.txt
@@ -1,2 +1 @@
 - email sheldon about release
-- submit issue for new found bug

# Remove from staging area
$ git reset
Unstaged changes after reset:
M       todos.txt

# Restore to original state (ef3c112)
$ git restore todos.txt
```

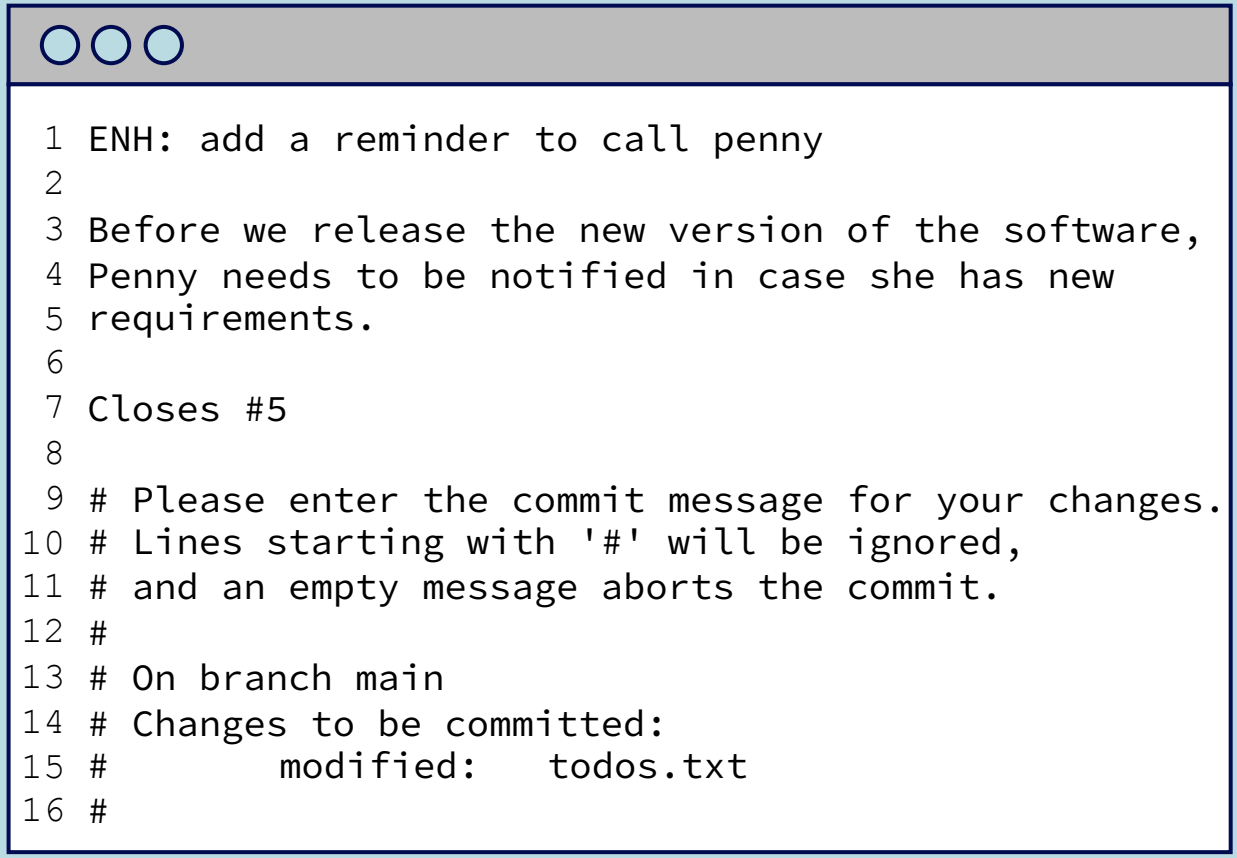
Commit Early, Commit Often

- A logical question to ask is "When to create a commit?"
  - In general, you want to do this when you have completed some logical set of work, like fixing a bug, or adding a test
  - If you can't explain what you changed in one sentence, your commit probably has too much stuff in it
- Remember commits have whole files in them, so it's easy to recover the changes from one whole commit
  - Conversely, it is hard to extract only some changes from one commit, so it helps if your commits are small
- Commits are also a way of persisting work you have done forever, so having lots of small commits prevents you from losing work you have completed
  - Like a save button in a word document with a timestamp
  - Can go back to a previous approach (as long as you "saved" or "committed' it!)

Writing Good Commit Messages

- Commit *messages* have two components – a title and a body
- The **title** should be short (<50 characters)
  - Use imperative mood ("Fix" not "Fixed")
    - "If applied, this commit will _____"
  - Describe what the change does
- Some style guides suggest prepending a three-letter abbreviation of the kind of work:
  - ENH - enhancement
  - BUG - bugfix
  - DOC - documentation
- The **body** should be two to four sentences that describe why the change is necessary and how it is implemented
  - Simple commits (like spelling errors) don't need a body
  - If using GitHub, this is typically where an issue from the issue tracker is referenced if applicable

```
# Instead of the shorthand `-m` option
# Use `git commit` to open a text editor
$ git commit
```



```
ENH: add a reminder to call penny

Before we release the new version of the software,
Penny needs to be notified in case she has new
requirements.

Closes #5

# Please enter the commit message for your changes.
# Lines starting with '#' will be ignored,
# and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#       modified:   todos.txt
#
```

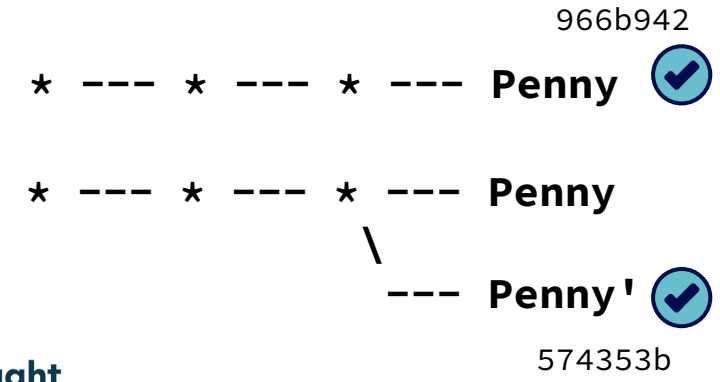
Unstaging Changes

- If a modified file has been placed into the index (via **`git add`**), this can be undone using **`git reset`** to move it back out of the index (i.e. remove from "staging")
- Remember, in Git the index is a critical data structure
  - It serves as the “staging area” between the files on your filesystem and your commit history.
  - When you run **`git add`**, the files from your working directory are hashed and stored as objects in the index, leading them to be “staged changes”.

```
$ cat todos.txt
- email sheldon about release
- submit issue for new found bug

$ echo "- call penny before release 1.2.3" >> todos.txt
$ cat todos.txt
- email sheldon about release
- submit issue for new found bug
- call penny before release 1.2.3

# Add to database index (stage changes)
$ git add todos.txt
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   todos.txt


# Remove from database index (unstage changes)
$ git reset
Unstaged changes after reset:
M       todos.txt

$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   todos.txt
no changes added to commit (use "git add" and/or "git commit
-a")
```

Changing a Commit Message

- To modify the **most recent commit** message, use the command
  git commit --amend
  - This will open a text editor for making updates to the **message** (not the changes)
  - Notice that this creates a new commit object
- It is possible to change the message of older or multiple commit messages (through rebasing), but this is generally frowned upon because it is rewriting history




```
$ cat todos.txt
- email sheldon about release
- submit issue for new found bug
- call penny before release 1.2.3

$ git add todos.txt
$ git commit -m "add a reminder to call penny bfor release
1.2.3"
[main 966b942] add a reminder to call penny bfor release
1.2.3
 1 file changed, 1 insertion(+)

$ git log --oneline
966b942 (HEAD -> main) add a reminder to call penny bfor
release 1.2.3
ef3c112 add reminder to submit issue
125b064 Initial commit

$ git commit --amend
[main 574353b] add reminder to call penny before release
1.2.3
 Date: Mon Apr 10 07:52:41 2023 -0500
 1 file changed, 1 insertion(+)

$ git log --oneline
574353b (HEAD -> main) add reminder to call penny before
release 1.2.3
ef3c112 add reminder to submit issue
125b064 Initial commit
```

Undoing a Commit

- Commits are forever
  - If a file has already been committed, there is not real way to remove the commit
  - Be careful not to push passwords or secrets
- Can move backwards in time to before the last commit with the command
  `git reset HEAD~1`

While Git aims to keep history of files for returning to previous state(s), there are some commands that are irrecoverable. Be careful with **resetting changes** and **renaming files** as history could be completely lost and changes could be unable to restore.

```
$ cat todos.txt
- email sheldon about release
- submit issue for new found bug
- call penny before release 1.2.3

$ git log --oneline
574353b (HEAD -> main) add reminder to call penny before
release 1.2.3
ef3c112 add reminder to submit issue
125b064 Initial commit


$ git reset HEAD~1
Unstaged changes after reset:
M         todos.txt

$ git log --oneline
ef3c112 (HEAD -> main) add reminder to submit issue
125b064 Initial commit

$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
          modified:   todos.txt
no changes added to commit (use "git add" and/or "git commit
-a")

# Restore todos.txt to ef3c112 state
# No more Penny reminder (irrecoverable)
$ git restore
Unstaged changes after reset:
M         todos.txt
```

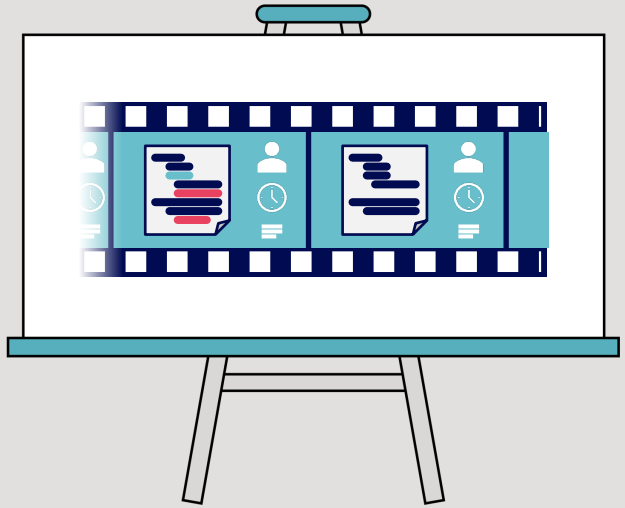
Undoing a Commit

- There is also an option to "revert" changes by using **`git revert <commit hash>`**
- This is the preferred method when wanting to "undo" changes as it keeps a clean history

```
$ cat todos.txt
- email sheldon about release
- submit issue for new found bug

$ git log --oneline
ef3c112 (HEAD -> main) add reminder to submit issue
125b064 Initial commit

$ git revert ef3c112
[main fbcf524] Revert "add reminder to submit issue"
 1 file changed, 1 deletion(-)

$ git status
On branch main
nothing to commit, working tree clean

$ git log --oneline
fbcf524 (HEAD -> main) Revert "add reminder to submit
issue"
ef3c112 add reminder to submit issue
125b064 Initial commit

$ cat todos.txt
- email sheldon about release
```

Give it a try! Committing to a Repo

1. Using any text editor of choice (or the command line), edit your **`todos.txt`** file to include a reminder to "fill out the Enthought class survey".
2. Add your file to the database index (e.g. the "staging area")
3. We want to be more specific – Enthought teaches multiple classes. Undo the change by "unstaging" the change and update to "fill out the Enthought SWNG class survey".
4. Add and commit your new changes.
5. View the history log of your **`todos.txt`** file.



Branching

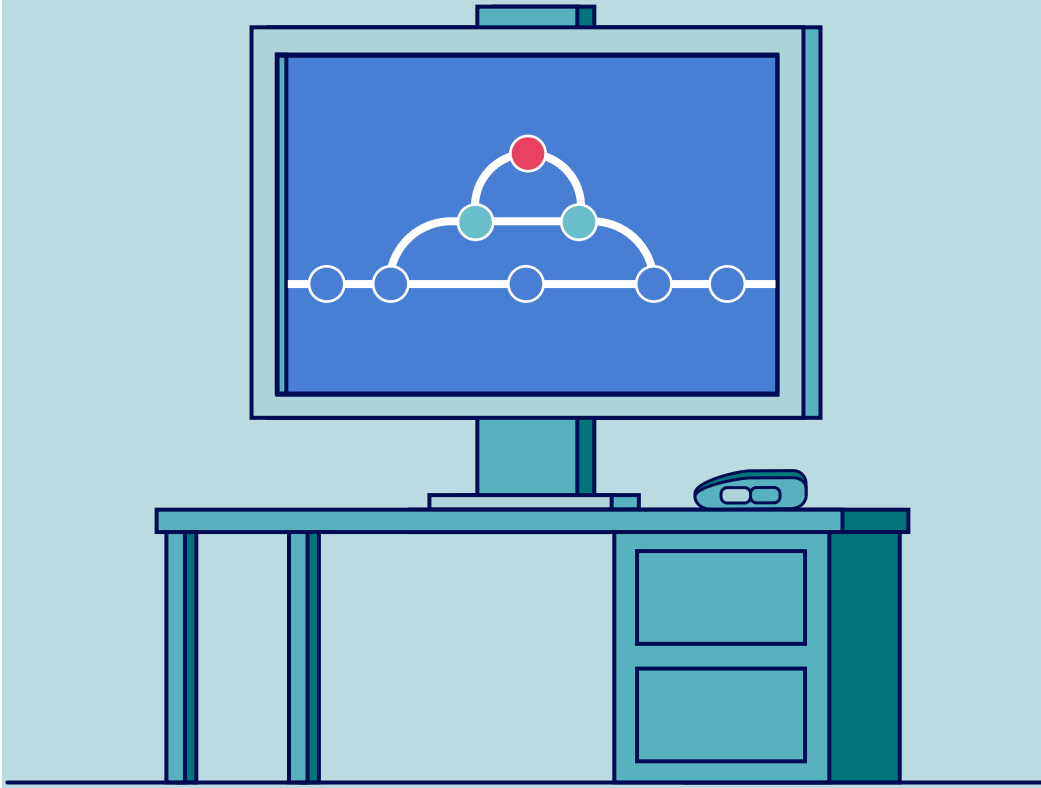
- Up to this point, learned 90% of the commands needed to effectively use Git.
  - Safely track changes in a linear sequence:
    `do work → git add → git commit`
  - Return to a previous state or undo changes:
    `git reset`, **`git checkout`**, **`git revert`**
  - View linear history:
    `git log`
- One more tool to allow multiple people to work on the same code in parallel (or one person to work on multiple tasks simultaneously).

- A **branch** lets users isolate separate histories of changes from each other

- Workflows in Git almost always revolve around the creation, maintenance, and distribution of branches

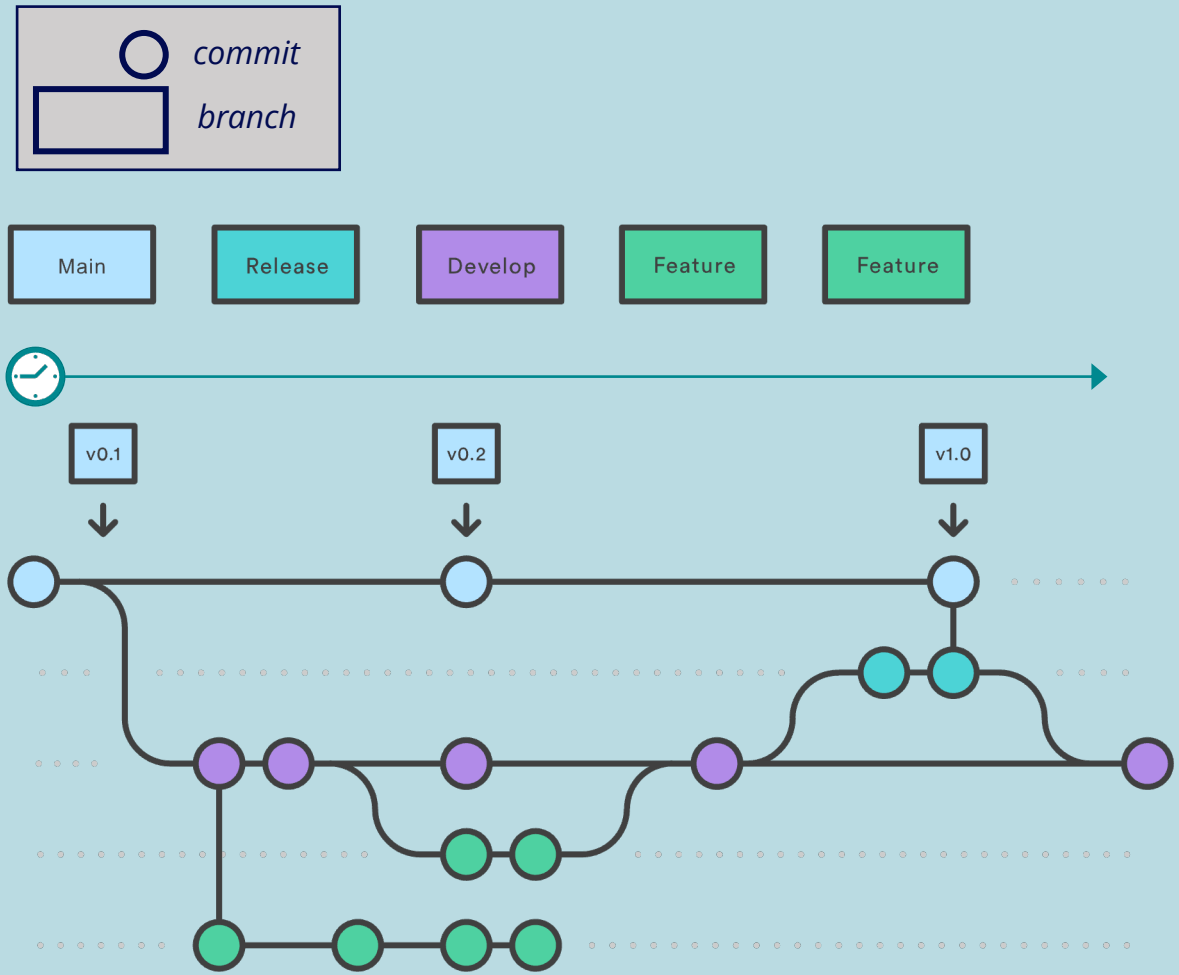
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

https://datasift.github.io/gitflow/IntroducingGitFlow.html



What is a Branch?

- A branch is a name attached to a particular commit
  - Think about them like sticky notes that can be pulled off one commit and placed on another
  - When "on" a branch, commits are made in a separate "workstream" that corresponds to the branch.
  - When a commit is made, the sticky note gets taken off of the parent commit and placed on the child commit
- Sending different commits to different branches is how we can maintain separate change histories in Git.
  - When the command **`git commit`** is executed for the first time, a branch called "main" is automatically created
  - The branch currently "on" gets the name "**HEAD**"
  - Branches are all stored in the **`refs/heads/`** directory in the Git database



Older repositories may have a default branch named "master" instead of "main". In 2020, "main" became the de facto naming convention.

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Managing Branches

- To create a new branch from the current commit, use the command
  `git branch <name of new branch>`
  - Can also create from an existing previous commit (**`git branch <name> <hash>`**) of from a different branch (**`git branch <new> <existing>`**)
  - To create a new branch and switch to it all at once use:
    `git checkout -b <new name>`
- To list all branches use **`git branch`**
  - The active branch is marked with an asterisk ('**`*`**')
- To delete a branch use
  `git branch -d <name>`
  - Cannot delete the current active branch

```
$ git branch dev

$ git cat-file -p dev
tree e914b65fa98817748622b6fcd1a77c602cfb8fc9
parent fbcf524d92dc2a1fdc51180b547b275abae908c0
author Logan Thomas <lthomas@e.com> 1681137778 -0500
committer Logan Thomas <lthomas@e.com> 1681137778 -0500

add reminder to fill out survey

$ git branch
  dev
* main

$ git checkout -b feat1 dev
Switched to a new branch 'feat1'

$ git branch
  dev
* feat1
  main

$ git branch -d feat1
error: Cannot delete branch 'feat1' checked out at
'/Desktop/notes'

$ git checkout dev
Switched to branch 'dev'

$ git branch -d feat1
Deleted branch feat1 (was 707c934).

$ git branch
* dev
  main
```

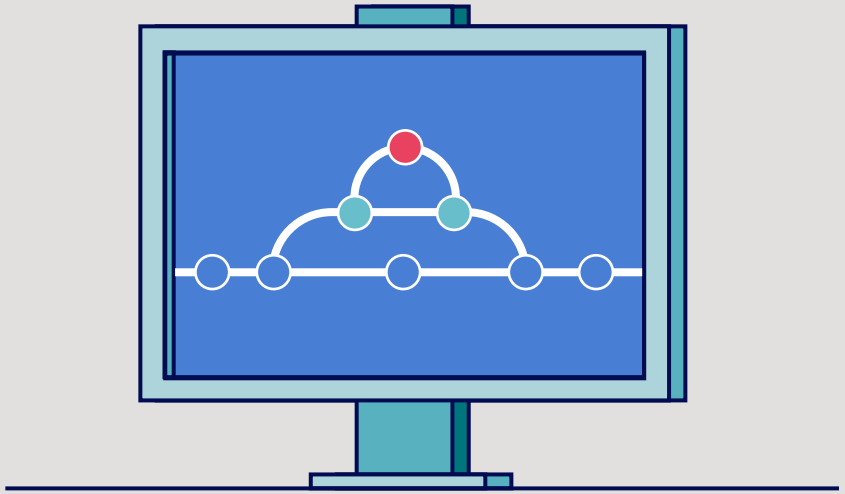
Moving Between Branches

- The current branch is stored in a special file in the database called **HEAD**
  - To move to a new branch (i.e. to move **HEAD**), use the command **`git checkout <branch name>`**
  - Notice this is the same command as before for recovering a past commit (this time using it to retrieve a branch instead of a commit hash)
- When providing a branch name, Git retrieves the correct objects from the database and updates **HEAD** to refer to the proper branch
  - When committing to this branch, only its name moves on to newer commits (**no other branches are changed**)

```
$ git checkout main
Switched to branch 'main'

$ git log --oneline
707c934 (HEAD -> main) add reminder to fill out survey
fbcf524 Revert "add reminder to submit issue"
ef3c112 add reminder to submit issue
125b064 Initial commit

# Use `--all` to see where other branches
# are attached to specific commits
$ git log --oneline --all
14282eb (dev) add note to debug jim datapipeline
707c934 (HEAD -> main) add reminder to fill out survey
fbcf524 Revert "add reminder to submit issue"
ef3c112 add reminder to submit issue
125b064 Initial commit

$ git cat-file -p HEAD
tree e914b65fa98817748622b6fcd1a77c602cfb8fc9
parent fbcf524d92dc2a1fdc51180b547b275abae908c0
author Logan Thomas <lthomas@e.com> 1681137778 -0500
committer Logan Thomas <lthomas@e.com> 1681137778 -0500

add reminder to fill out survey

$ git checkout dev
Switched to branch 'dev'

$ git cat-file -p HEAD
tree a77a9f838bfa0d1cadbebe909ad42372568bd49d
parent 707c934db50d7bae0c515c801eb23ff367b51f58
author Logan Thomas <lthomas@e.com> 1681140632 -0500
committer Logan Thomas <lthomas@e.com> 1681140632 -0500
add note to debug jim datapipeline
```

Give it a try! Create a Branch

1. Create a new branch named **`dev`** in the **`notes`** repository. Navigate to the **`dev`** branch.
2. Inspect the history using **`git log --oneline`**.
3. Add a new reminder to "debug Jim's datapipeline". Commit the change.
4. Inspect the history again – what do you notice about the branch names and **`HEAD`** reference?



Visualizing Branches

- Sometimes it can be helpful to visualize how different commit histories are related
- The **`git log`** command comes with a **`--graph`** flag that will draw out the tree of commits
  - It's a good idea to also use the **`--oneline`** as to not overflow the terminal with output
  - The **`--all`** flag shows all commits (not just those on the current branch)

```
$ git log --all --graph --oneline
* 14282eb (dev) add note to debug jim datapipeline
* 707c934 (HEAD -> main) add reminder to fill out survey
* fbcf524 Revert "add reminder to submit issue"
* ef3c112 add reminder to submit issue
* 125b064 Initial commit

$ echo "- write docs for new database aggregation method" >>
todos.txt

$ git commit -am "add note to document new database agg
method"
[main d908411] add note to document new database agg method
 1 file changed, 1 insertion(+)

$ git log --all --graph --oneline
* d908411 (HEAD -> main) add note to document new database
agg method
| * 14282eb (dev) add note to debug jim datapipeline
|/
* 707c934 add reminder to fill out survey
* fbcf524 Revert "add reminder to submit issue"
* ef3c112 add reminder to submit issue
* 125b064 Initial commit
```

Git Aliases

- You can create aliases for frequently used commands using **`git config`**

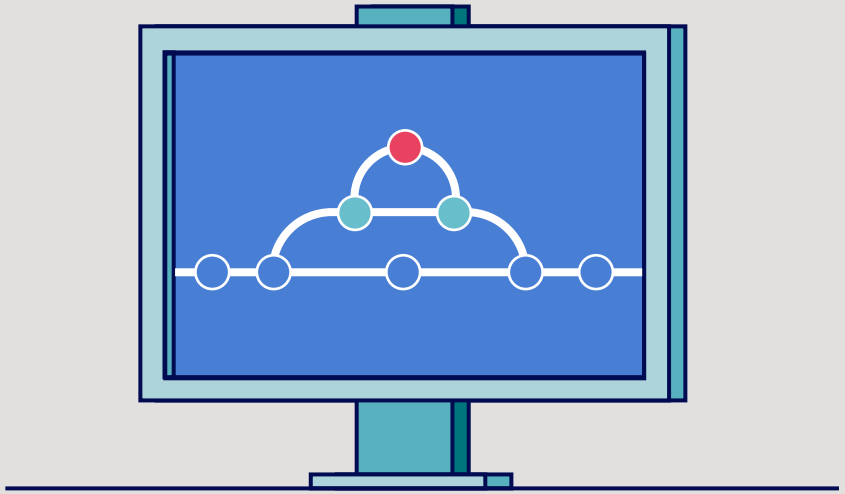
- Aliases created with the --global flag are saved in a file name **`.gitconfig`** in your home directory

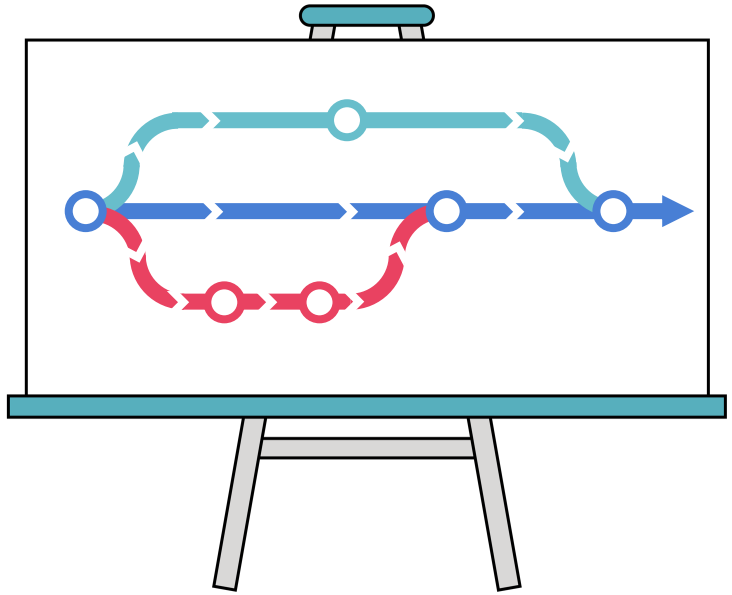
```
$ git config --global alias.unstage 'reset HEAD --'
$ git config --global alias.lga "log  --oneline --graph --all"

$ git lga
*   4ff96a0 (dev) Merge branch 'main' into dev
|\
| * 323d2ef (HEAD -> main) add note to include sort argument
to read_file()
* | ce5fe72 Merge branch 'main' into dev
|\|
| * d908411 add note to document new database agg method
* | 14282eb add note to debug jim datapipeline
|/
* 707c934 add reminder to fill out survey
* fbcf524 Revert "add reminder to submit issue"
* ef3c112 add reminder to submit issue
* 125b064 Initial commit

$ git config --get alias.unstage
reset HEAD
$ git config --get alias.lga
log --oneline --graph --all
```

https://git-scm.com/book/en/v2/Git-Basics-Git-Aliases

Give it a try! Navigate Branches

1. Navigate to the "base" branch of the **`notes`** repository (usually **`main`** or **`master`**).
2. Inspect the history using **`git log --oneline`**.
3. Add a new reminder to "test doug's new sorting feature". Commit the change.
4. Visualize the history of the **`notes`** repository using **`git log`**.



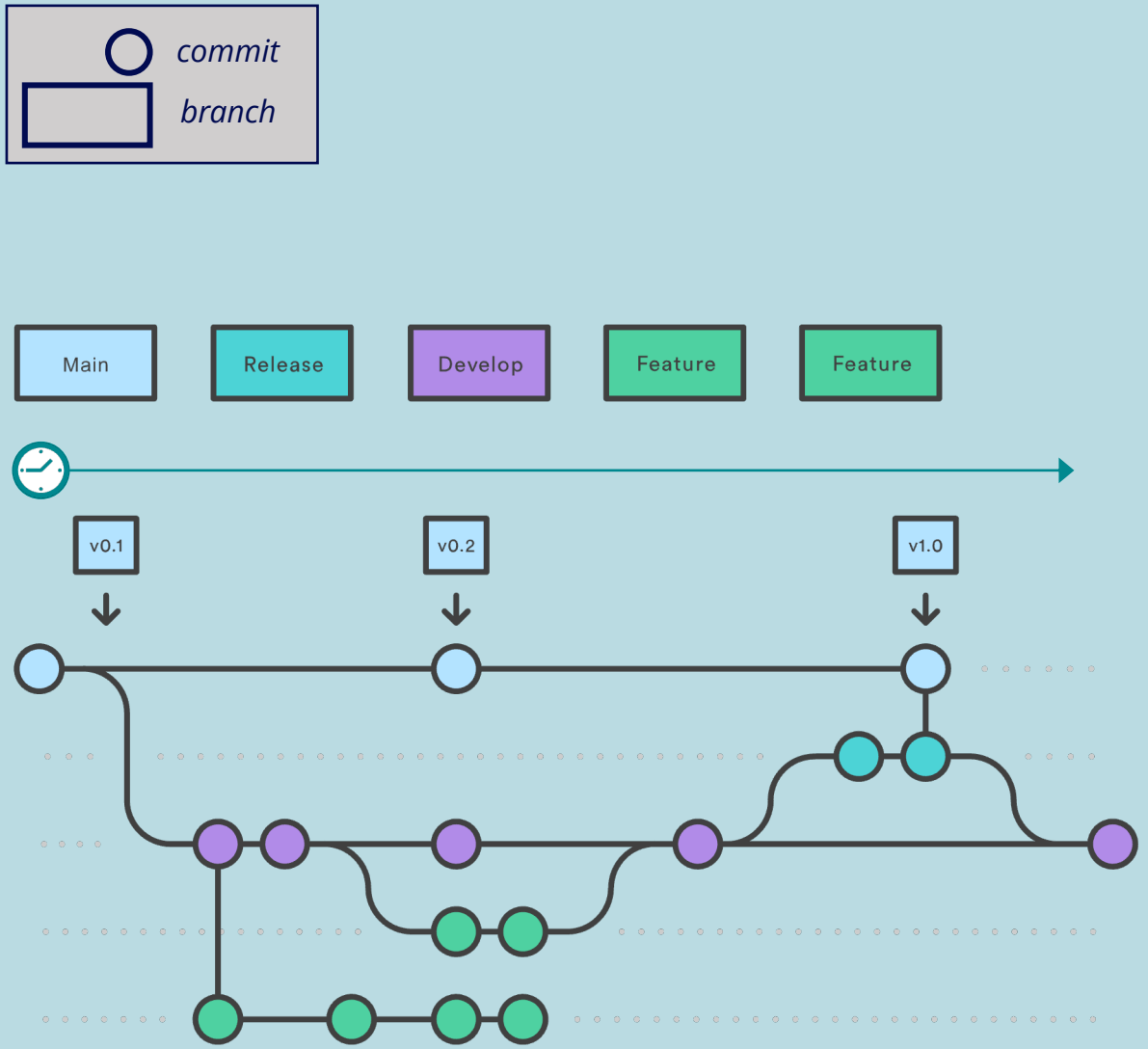


Git Workflow

Lecture 08
Source Control

Joining Branches

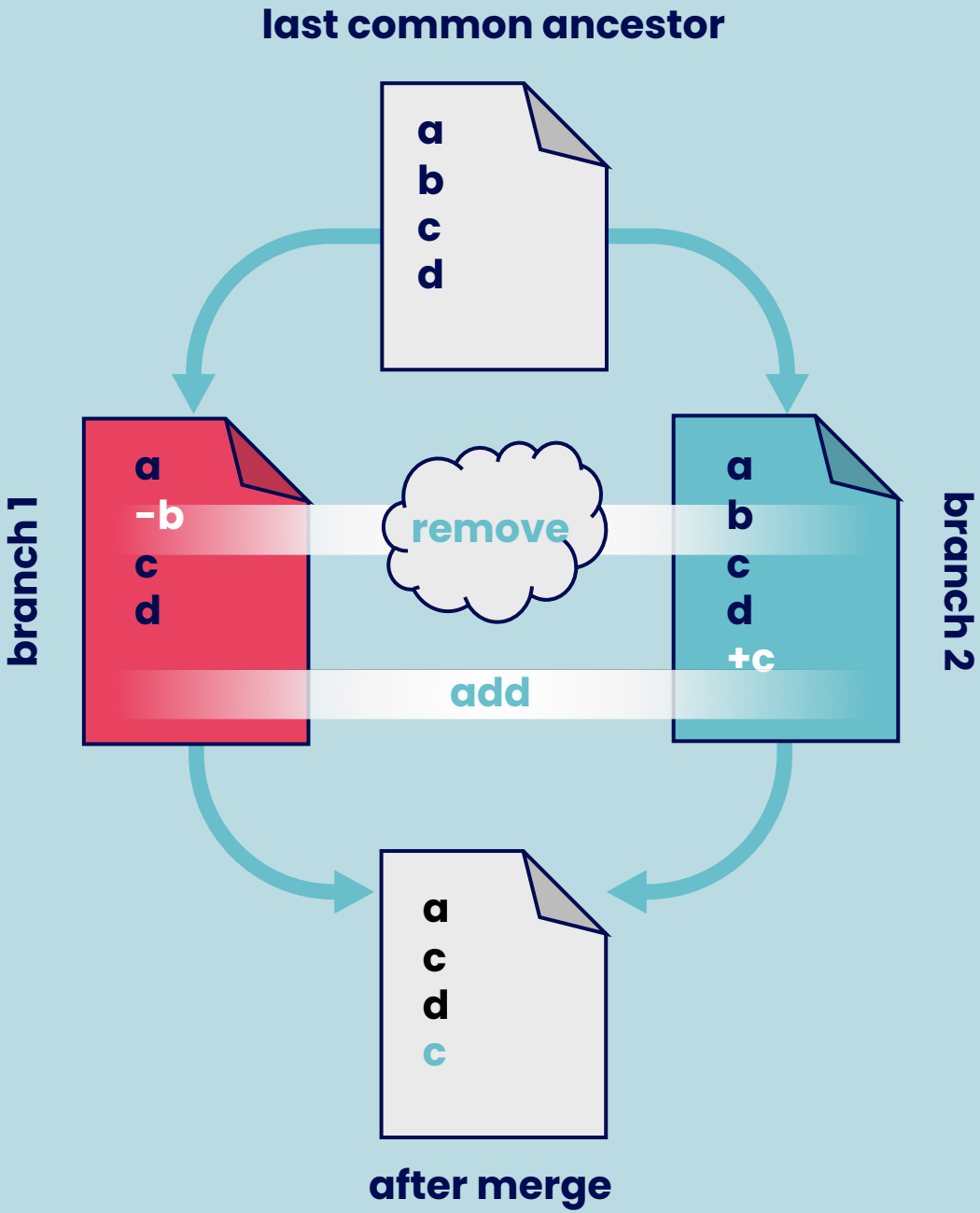
- Branches are a great way to isolate work that adds one feature, or that fixes one bug
- To use branching properly, separate branches need to be re-joined back in the base branch
  - Typically, the base branch is called **`main`** or **`master`** in the repository.
  - Some workflows have several concurrent and long-lived branches
    - Levels of stability (**`main`**, **`dev`**, **`prod`**)
    - Feature sets or releases (**`main`**, **`1.0x`**, **`2.0.x`**)
- Merging branches is based on a combination of history and file "diffing"
  - Errors in either can make it difficult to combine changes to a repository.



https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

What is a Merge?

- The default merging algorithm in Git starts by using the commit history to find the most recent common ancestor of two different versions of a file
- The difference ("diff") between each file and the ancestor is retrieved
  - If a line **hasn't** been changed in either file, it is **copied** verbatim
  - If a line **has been** changed in only one descendent, the change is **copied in place** of the older code
  - If a line is **modified in both** files, it will cause a **merge conflict** and Git will ask you to merge the files by handle



Merging Branches

- To merge another branch into the current branch use **`git merge <other>`**

```
# Currently on main branch
$ cat todos.txt
- email sheldon about release
- fill out the Enthought SWNG class survey
- write docs for new database aggregation method

# Created and navigate to new branch
$ git checkout -b new-branch main
Switched to a new branch 'new-branch'

$ echo "- add sort arg to read file method" >> todos.txt

# Add to index and include message in same command with `-am`
$ git commit -am "add note to include sort argument to
read_file()"
[new-branch 323d2ef] add note to include sort argument to
read_file()
 1 file changed, 1 insertion(+)

$ cat todos.txt
- email sheldon about release
- fill out the Enthought SWNG class survey
- write docs for new database aggregation method
- add sort arg to read file method

$ git checkout main
Switched to branch 'main'

$ git merge new-branch
Updating d908411..323d2ef
Fast-forward
 todos.txt | 1 +
 1 file changed, 1 insertion(+)

$ git branch -d new-branch
Deleted branch new-branch (was 323d2ef).
```

Merge Conflicts I

- If both branches in a merge have modified the same line from its ancestral form, Git does not know which on should persist in the future
  - Git will tell you that there has been a "conflict" and ask you to edit it manually
- Any file with conflicts will have one expanded section per conflict that starts and ends with chevrons (**>>>>>** and **<<<<<**)
  - A string of equal signs (**=====**) will be displayed in the middle of the chevron section
  - **Above the equal signs** are the changes between the **ancestor and the current** branch (usually main); Think "Current State"
  - **Below the equal sign** are the changes between the **ancestor and the branch you are merging in**; Think "Incoming Change"
- Edit the file to have the proper merger of the two source files, noting you may need components of both
- Don't forget to erase the conflict makers!

```
$ git branch
  dev
* main

$ cat todos.txt
- email sheldon about release
- fill out the Enthought SWNG class survey
- write docs for new database aggregation method
- add sort arg to read file method

$ git checkout dev
Switched to branch 'dev'

$ cat todos.txt
- email sheldon about release
- fill out the Enthought SWNG class survey
- debug jim's datapipeline

# Current branch is dev; incoming is main
$ git merge main
Auto-merging todos.txt
CONFLICT (content): Merge conflict in todos.txt
Automatic merge failed; fix conflicts and then commit the
result.

$ cat todos.txt
- email sheldon about release
- fill out the Enthought SWNG class survey
<<<<<<< HEAD
- debug jim's datapipeline
=======
- write docs for new database aggregation method
- add sort arg to read file method
>>>>>>> main
```

Merge Conflicts II

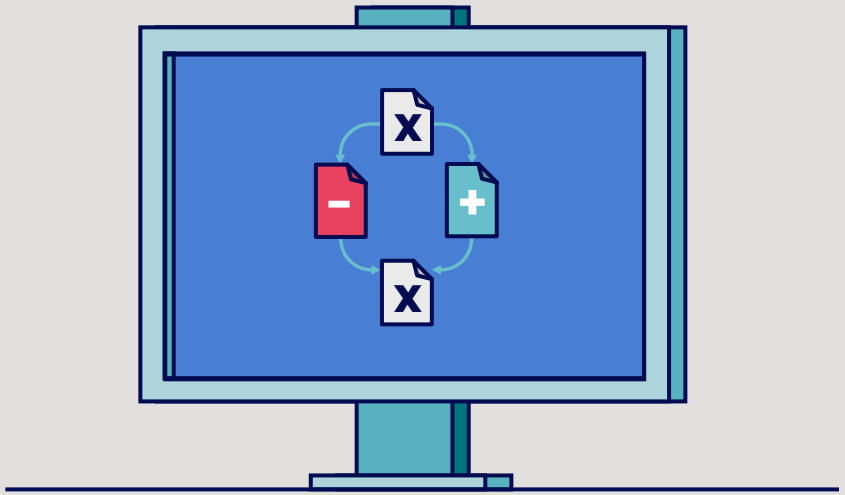
- After editing the files appropriately, they must be added to the index to let Git know merge conflicts have been resolved. This is done via the **`git add <file>`** command
- When all file changes are complete and all conflicts have been resolved, use the **`git merge --continue`** command or simply git commit
- If the merge has unresolvable conflicts (or was a mistake), back out of the attempted merge with the **`git merge --abort`** command

Depending on the ordering of merging the **`new-branch`** and into **`main`** and when the **`main`** branch was merged into **`dev`**, the graph to the right may be different. (**`323d2ef`** was the merge of new-branch)

```
# Open text editor to resolve conflicts
$ vim todos.txt

# Add to the database index
$ git add todos.txt

# Continue merge process; could have used
# git merge -continue
$ git commit
[dev ce5fe72] Merge branch 'main' into dev

$ git checkout main
Switched to branch 'main'

$ git log --oneline --all --graph
*   4ff96a0 (dev) Merge branch 'main' into dev
|\
| * 323d2ef (HEAD -> main) add note to include sort
argument to read_file()
* | ce5fe72 Merge branch 'main' into dev
|\|
| * d908411 add note to document new database agg
method
* | 14282eb add note to debug jim datapipeline
|/
* 707c934 add reminder to fill out survey
* fbcf524 Revert "add reminder to submit issue"
* ef3c112 add reminder to submit issue
* 125b064 Initial commit
```

Give it a try! Handling Merge Conflicts

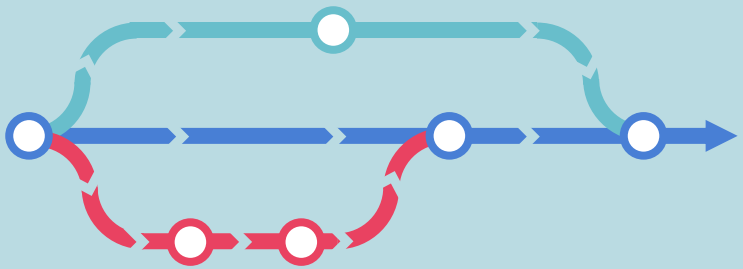
1. Merge the **`dev`** branch into **`main`**.
2. Resolve the merge conflict in your preferred text editor. Include all the to-dos from both branches (assume order doesn't matter).
3. Commit the final merge using **`git commit`**.
4. Visualize the history of the **`notes`** repository using **`git log`**.



Typical Process

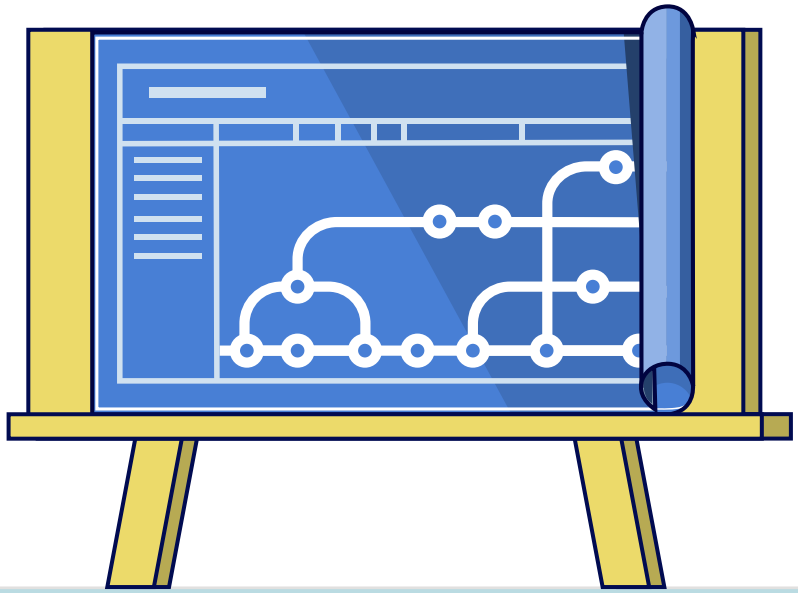
1. Create a branch: **`git branch <branch name>`**
2. Checkout the branch: **`git checkout <branch name>`**
3. Make changes to files
4. Stage the changed files (add to database index): **`git add <filename>`**
5. Commit the changes: **`git commit`**
6. (Periodically merge "main" into branch to include changes from others):
   ```
   git checkout main
   git pull
   git checkout branch name
   git merge main
   ```
7. Once work is complete, checkout the "main" branch: **`git checkout main`**
8. Merge the branch back into "main": **`git merge <branch name>`**

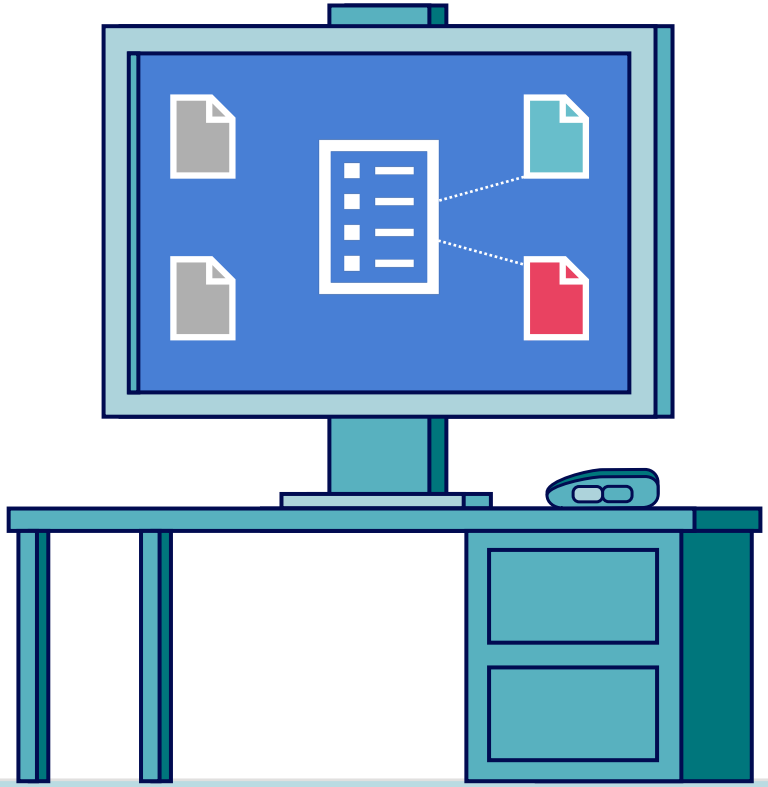
Step 6 is optional if working independently. For multi-developer projects, steps 7 and 8 are usually done via a "Pull Request".



Technical Details (Advanced Usage)

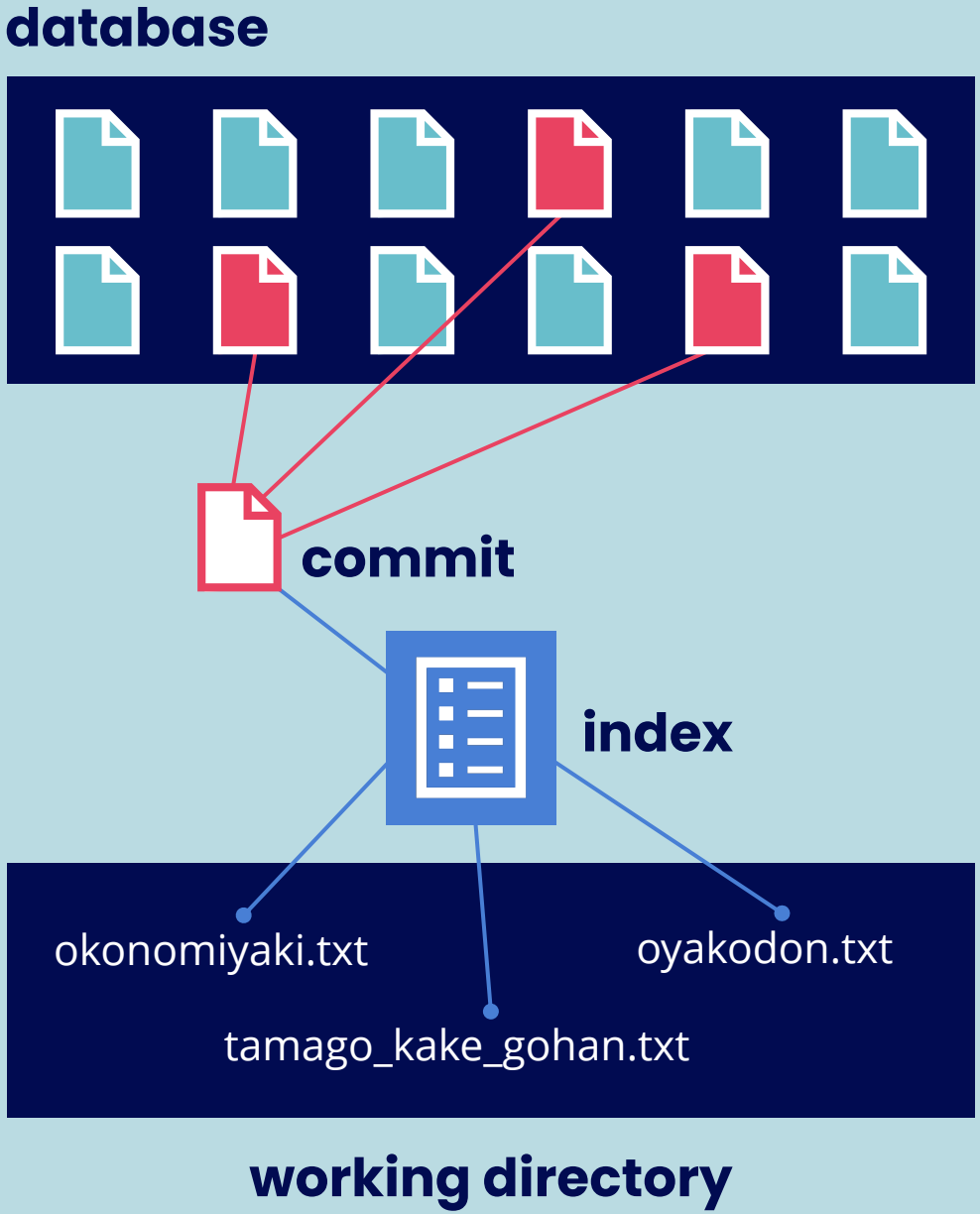
- Remember 90% of the benefit of using Git can be acquired while only understanding 10% of the underlying mechanisms
- Next few slides are for future endeavors as you become more familiar with Git
- Some examples are from retired, previous classes where various food recipes were gathered -- but principles are the same





How Does Git Work?

- Git is a program that stores an entire file (object) in a database every time you change any part of it
- Moments in time (commits) are also stored in this database, and consist of the database locations of each file at its last change
- The entire history of your file is recoverable because every moment in time knows about the moment in time just before it
- The location of each file in your filesystem (not Git's database) is stored in a special file (index), and Git will only track changes to files added to the index



What Happens During a Commit

- If you look in the Git database (.git) you'll see four folders — we'll primarily be interested in the one called objects

- When you ran git add, you created an object in the database to store the file, and an index entry to link it to a place in the file system

- When you ran git commit
  1. An object was created that stores the locations of the current version of every file in the index
  2. Along with the last commit
  3. The message you typed
  4. Your name

- We can use git cat-file to look at the contents of the object

```
$ cd .git
$ ls -ltF
...
-rw-r--r-- ... index
drwxr-xr-x ... info/
drwxr-xr-x ... objects/
drwxr-xr-x ... refs/

$ ls -ltF objects
drwxr-xr-x ... 5f/
drwxr-xr-x ... e0/
drwxr-xr-x ... 55/

$ ls -ltF objects/55
-r--r--r-- ... d7c54988c99f09d41cd09287e41274db3b5a9e

$ git cat-file -p
55d7c54988c99f09d41cd09287e41274db3b5a9e
cabbage
```

The Structure of the Database

- Broadly speaking, there are **three kinds of files** that live in Git's database (the objects)
- **One of these** we have seen already — ***the contents of a file***, where the name is the SHA1 hash of the file contents
- A **second object** is something called a ***tree***, which is an object that stores that locations of other objects
- The purpose of trees is to track collections of files, like the state of a whole repository at one moment in time
- The **third object** is a ***commit***, which stores:
  1. The location of its parent commit
  2. The repo state (as a tree)
  3. Some metadata like a timestamp

```
objects/
├── 48
│   └── 8fa6b235dab51d03a481a541b991f1...
├── 55
│   └── d7c54988c99f09d41cd09287e41274...
├── 5f
│   └── 787ee580886ab09f0f8feb99e181b0...
├── e0
│   └── 949f4ca1132390d22bce021258de02...
├── info
└── pack
```

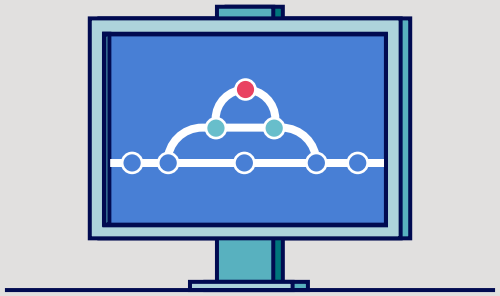
Knowing What Has Changed

- Other VCS store the initial states of files, followed by each incremental change to the file
- This has the downside of making your entire repository unusable if one single commit disappears or is corrupted
- We have already seen that Git stores entire files, not differences between files
- To know what's changed then, Git uses an algorithm to find the difference between two versions of the same file
- This is called **"diff"ing**, which is short for **"difference"ing**
- Git has a handful of diff algorithms, which vary in speed / accuracy
- The output of a **`diff`** is called a **`patch`**

```
diff --git a/okonomiyaki.txt b/okonomiyaki.txt
index 55d7c54..488fa6b 100644
--- a/okonomiyaki.txt
+++ b/okonomiyaki.txt
@@ -1 +1,2 @@
 cabbage
+yam flour
```

Resources

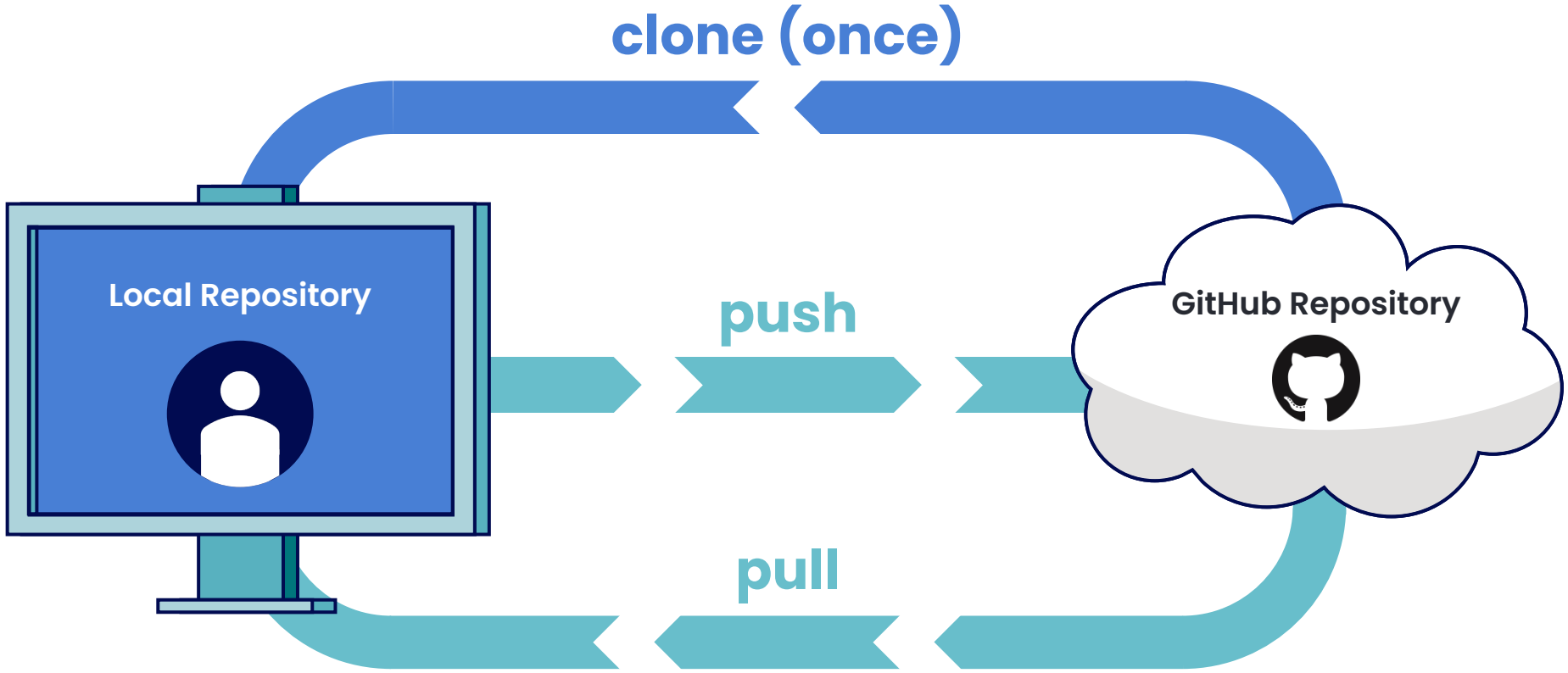
- **Git cheatsheets**
  - https://www.ndpsoftware.com/git-cheatsheet.html#loc=index
  - https://wac-cdn.atlassian.com/dam/jcr:e7e22f25-bba2-4ef1-a197-53f46b6df4a5/SWTM-2088_Atlassian-Git-Cheatsheet.pdf?cdnVersion=99
  - https://education.github.com/git-cheat-sheet-education.pdf
  - https://about.gitlab.com/images/press/git-cheat-sheet.pdf
- **When things go bad**
  - https://sethrobertson.github.io/GitFixUm/fixup.html
  - https://ohshitgit.com/
- **Graphical User Interface**
  - https://desktop.github.com/



Collaboration

Lecture 08
Source Control

GitHub Overview

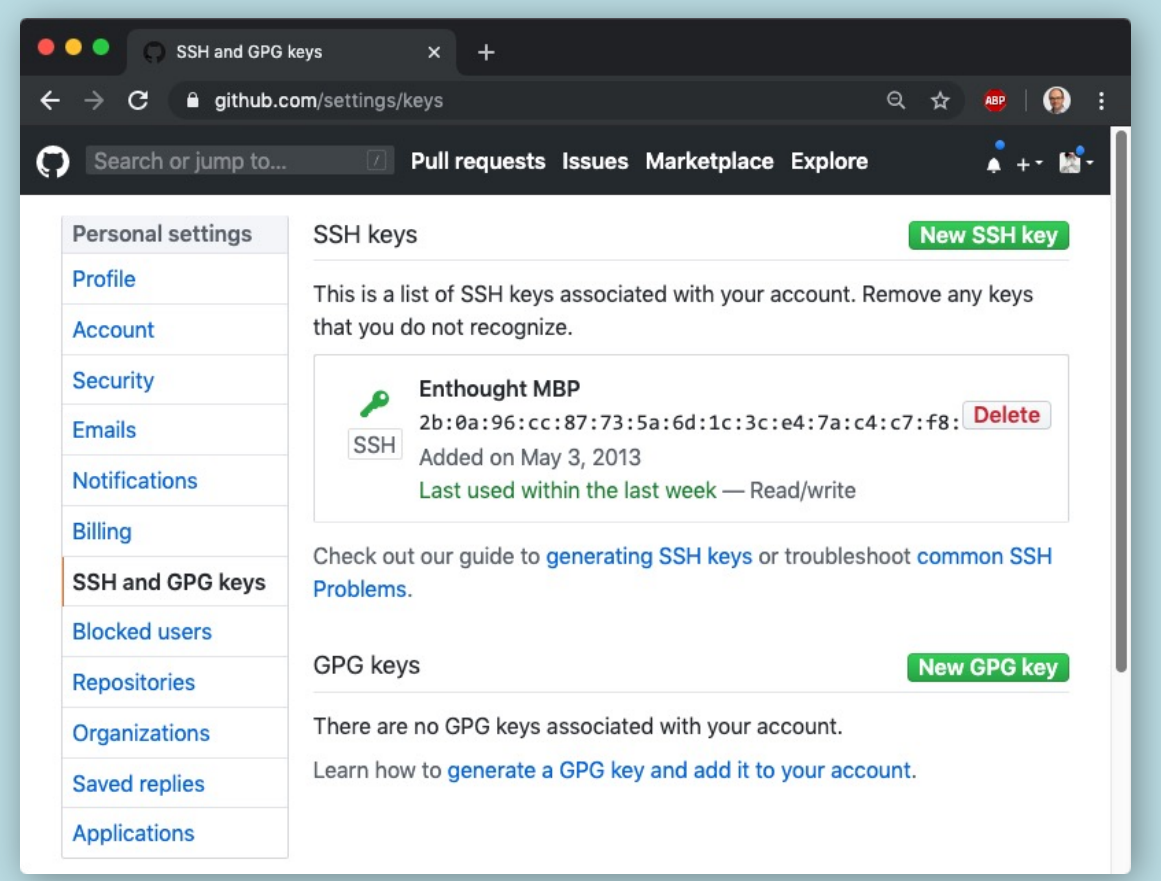


Setting Up SSH Keys

- An ssh key pair should be used to allow secure communication with GitHub.
- **Set up requires three steps:**
  - Generate a key pair.
  - Add the private key to the ssh agent running locally.
  - Add the public key to your GitHub account.

- **Reference:** https://help.github.com/en/articles/connecting-to-github-with-ssh

```
$ ssh-keygen -t rsa -b 4096 -C
"username@e.com"

$ ssh-add ~/.ssh/id_rsa

$ cat ~/.ssh/id_rsa.pub

<copy the output and paste in GitHub>
```

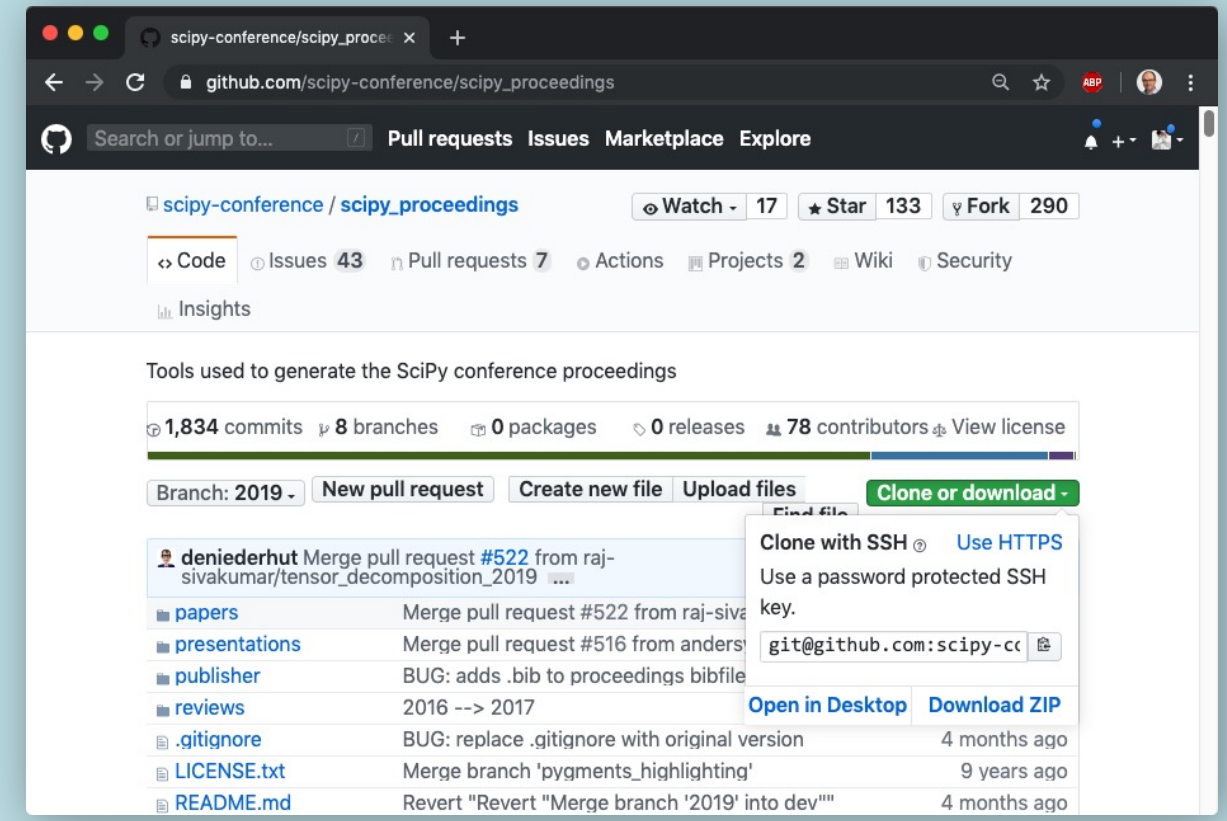


Transferring To and From Remotes

- To copy an existing repository:
  `git clone <location>`
- To update your local repository:
  `git pull`
- Prepare to make changes:
  `git branch [branch]`
  `git checkout [branch]`

  Make actual changes to files, then:
  `git add [file]`
  `git commit`

  "Push" the changes to the GitHub repository:
  `git push -u origin [branch]`




```
$ git clone git@github.com:scipy-
conference/scipy_proceedings.git
$ git remote -v
origin  git@github.com:scipy-conference/scipy_proceedings.git
(fetch)
origin  git@github.com:scipy-conference/scipy_proceedings.git
(push)
```

Example

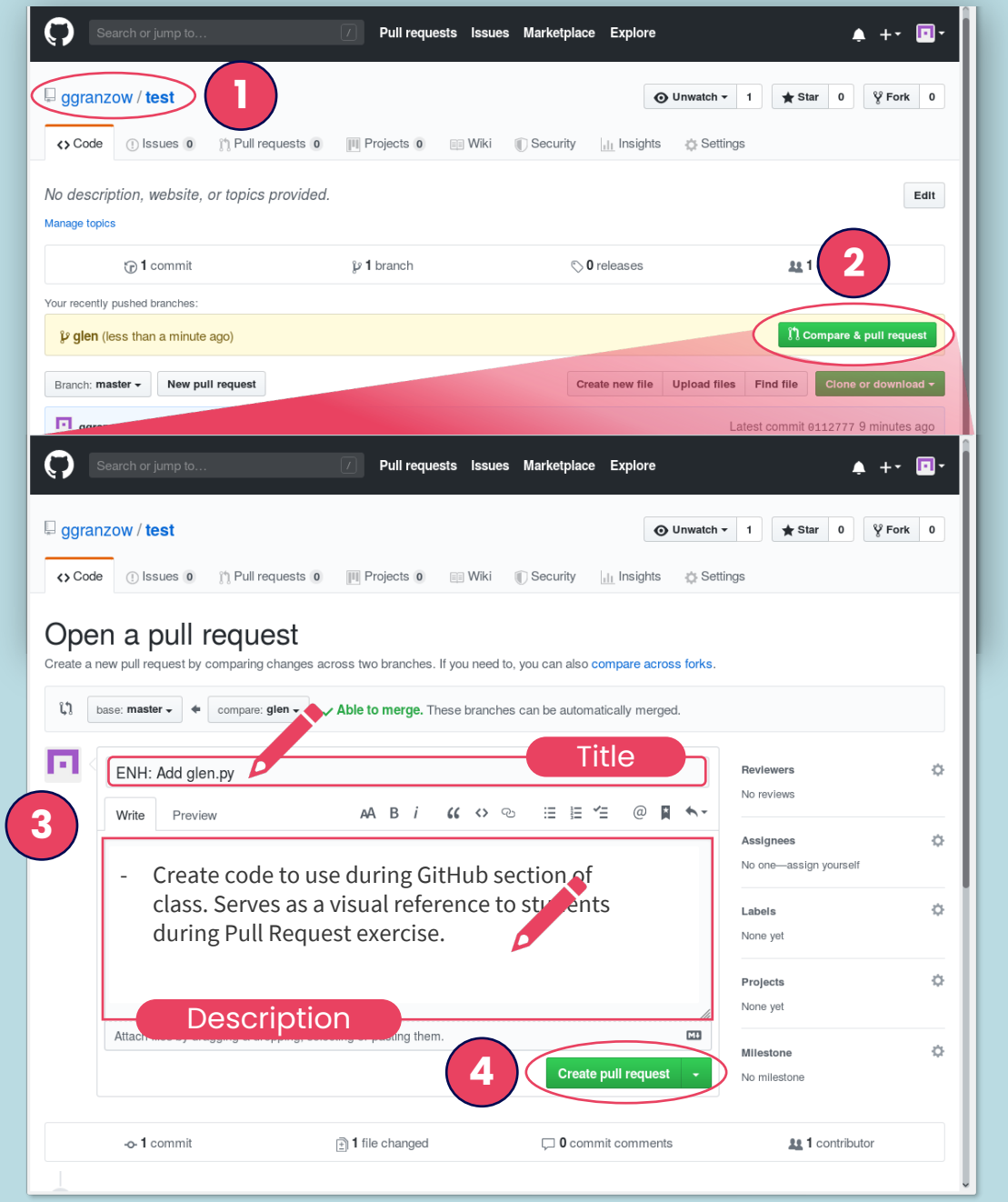
```
$ git clone git@git.dillerdigital.com:DillerDigitalStudents/DD-SWNG-20250825-NM.git
$ cd DD-SWNG-20250825-NM
$ git branch rel/update-students
$ git checkout rel/update-students
$ echo "Hello World" >> students/my_name.txt
$ git add students/my_name.txt
$ git commit -m "REL: add my info"
$ git push -u origin rel/update-students
```

Making a Pull Request (PR)

After you have pushed a branch to GitHub, the next step is to create a “pull request” to merge your branch into the GitHub repository.

This requires three steps:

1. Navigate to the GitHub repository website.
2. Click on “Compare & pull request.”
3. Write a title summarizing the changes and a description justifying why the changes were made.
4. Click on “Create pull request.”

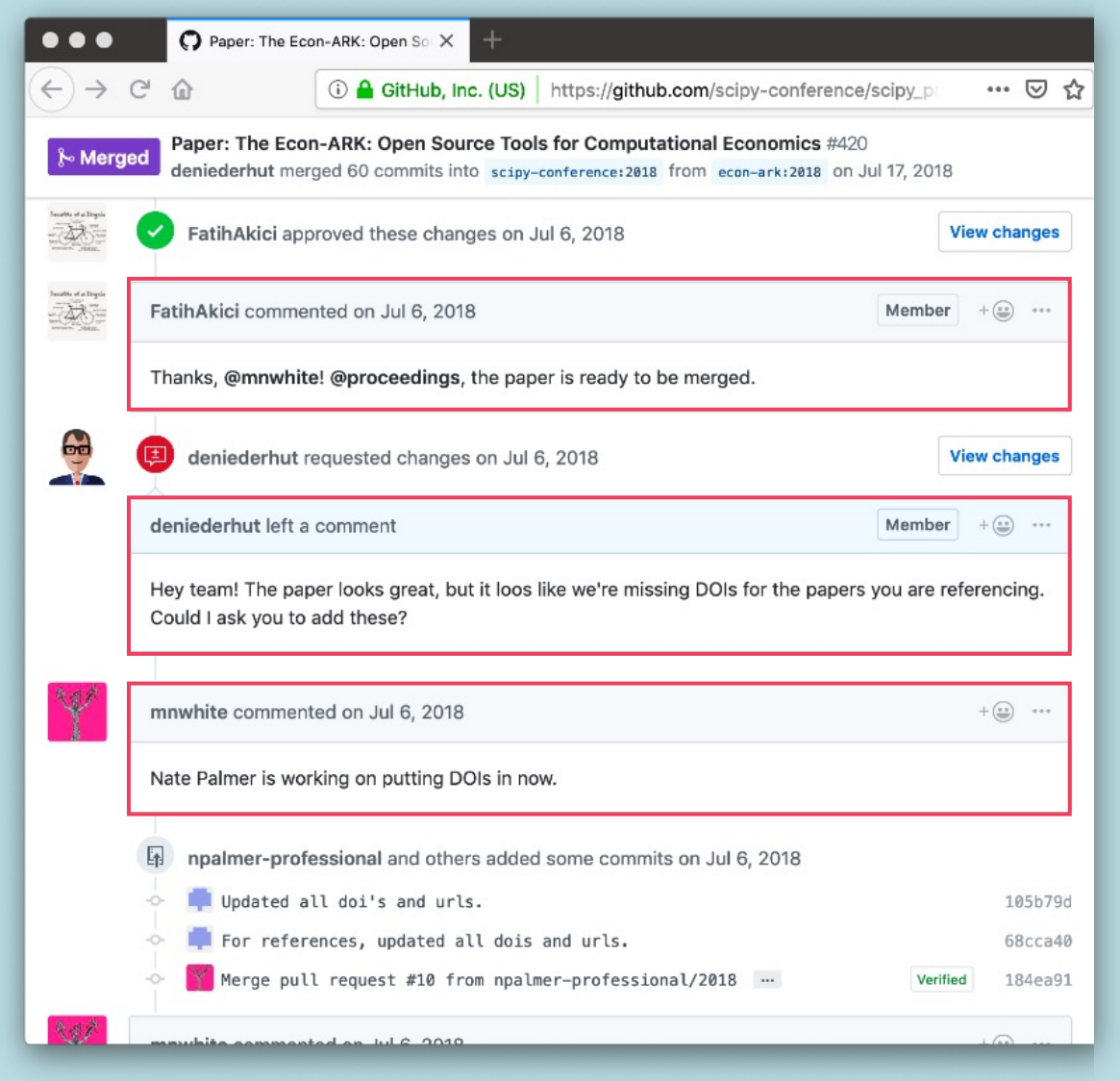


Reviewing a Pull Request (PR)

After you create a pull request a colleague should review it, checking that:

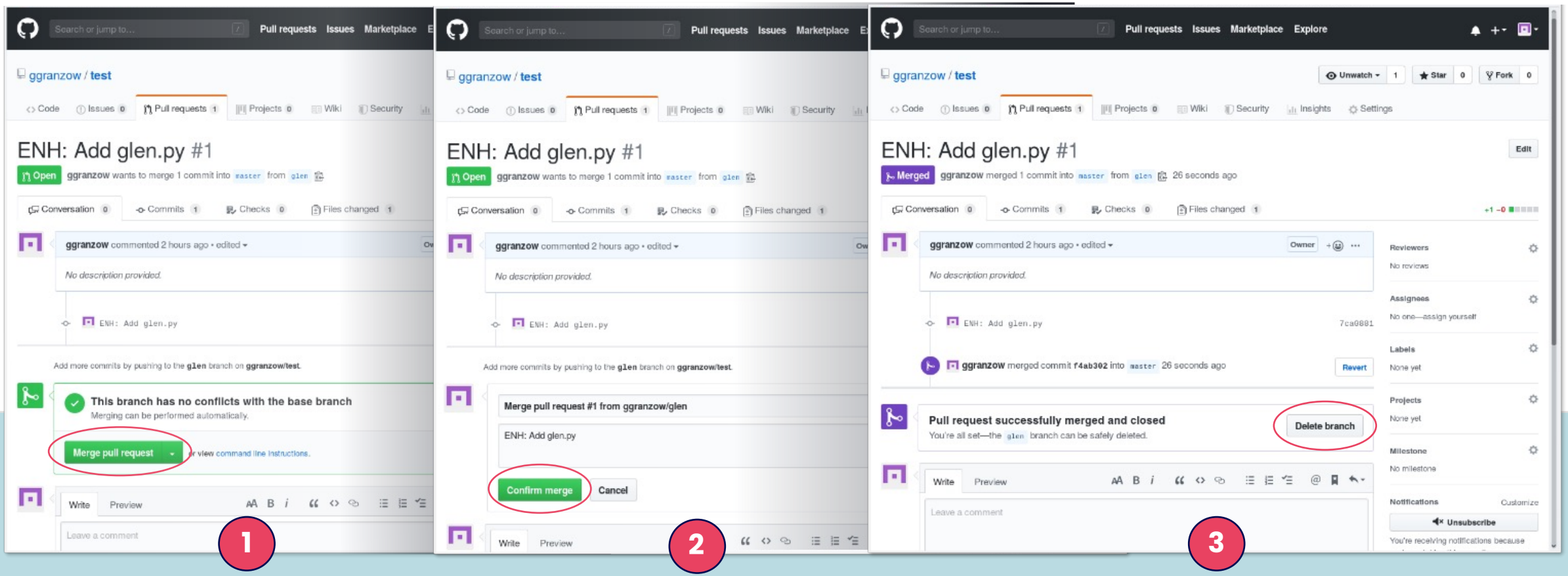
- The program works
- The program is useful
- The program follows style conventions
- The program follows best practices

The reviewer can leave **comments** and/or approve the request.



Merging a Pull Request (PR)

- After a pull request has been reviewed it can be merged into the GitHub Repository
  1. Click on "Merge pull request"
  2. Click on "Confirm merge"
  3. Click on "Delete branch" (usually)



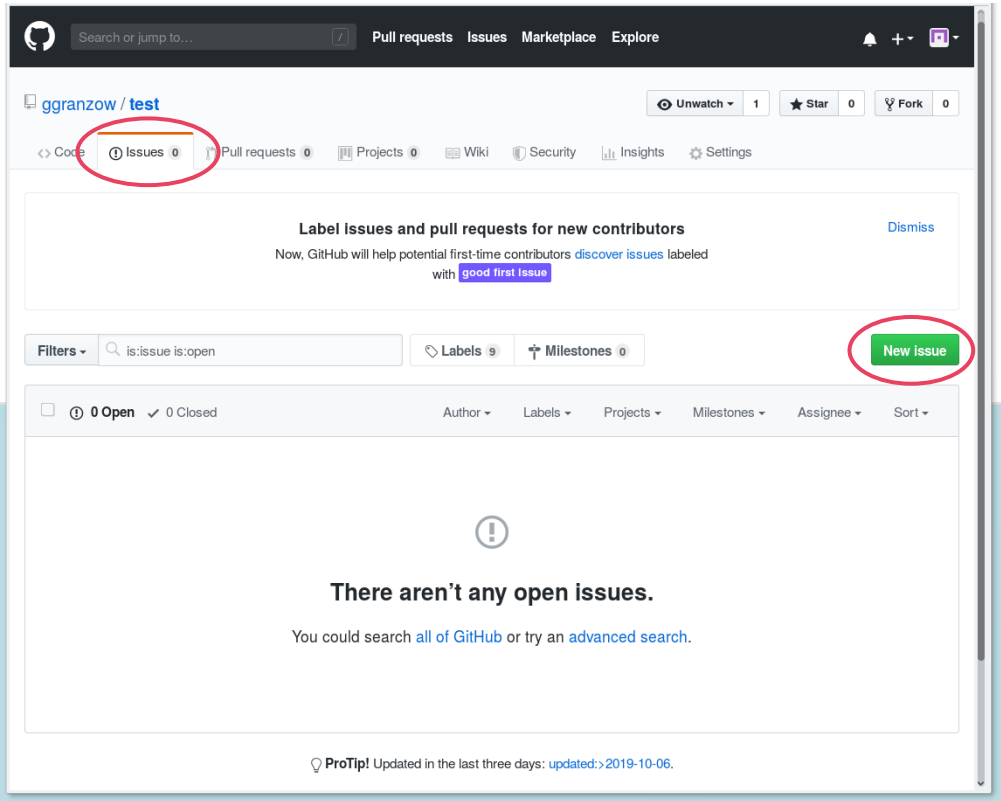
Updating Your Local Repository

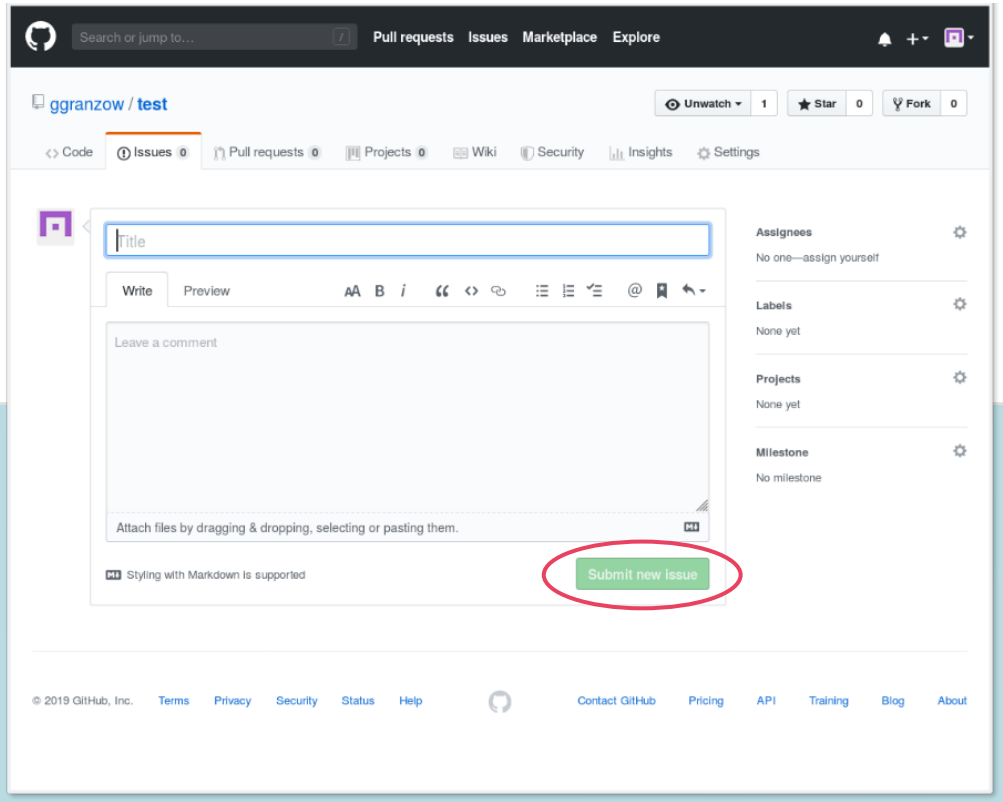
- **To update your local repository**
  1. Checkout the "main" branch: **`git checkout main`**
  2. Pull the most recent commit(s): **`git pull`**

```
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'github/main'.
~/github/class-material/manual_sections (main)
$ git pull
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 10 (delta 2), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), done.
From github.com:enthought/class-material
    e60bf5f7..02cb8583  enh-dcat-search -> github/enh-dcat-search
Already up to date.
~/github/class-material/manual_sections (main)
```

GitHub Issues

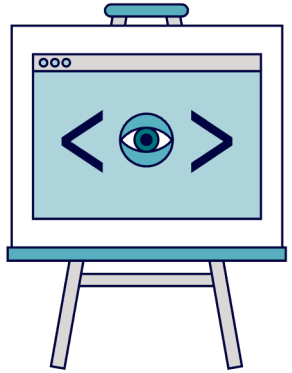
- **Use GitHub's issue feature to**
  - Report bugs
  - Make feature requests
  - Ask questions





GitHub Process

1. Select an issue or feature request to work on
2. Pull the most recent **`main`** commit
3. Make a branch for changes
4. Make and commit changes
5. Push changes to GitHub
6. Make a pull request (PR)



Lecture 02

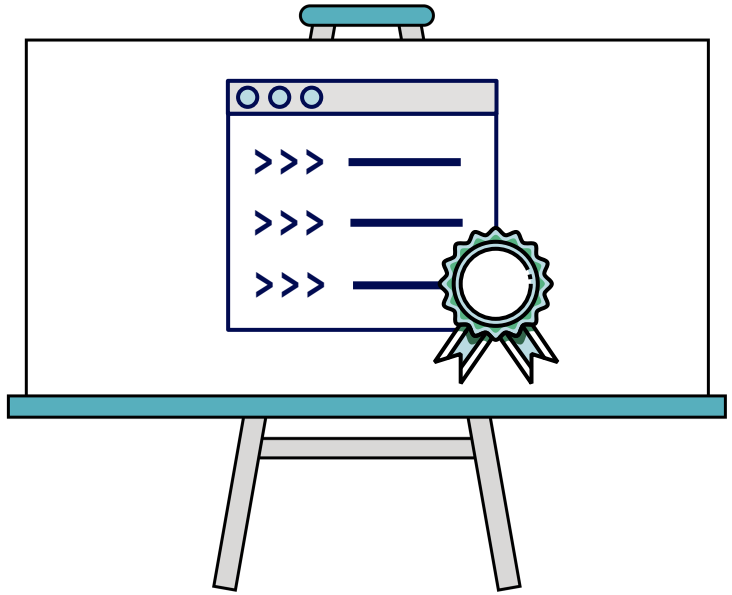
Readable Code

Software Engineering for Scientists and Engineers

Discussion

What makes code readable (for humans)?

Table of Contents

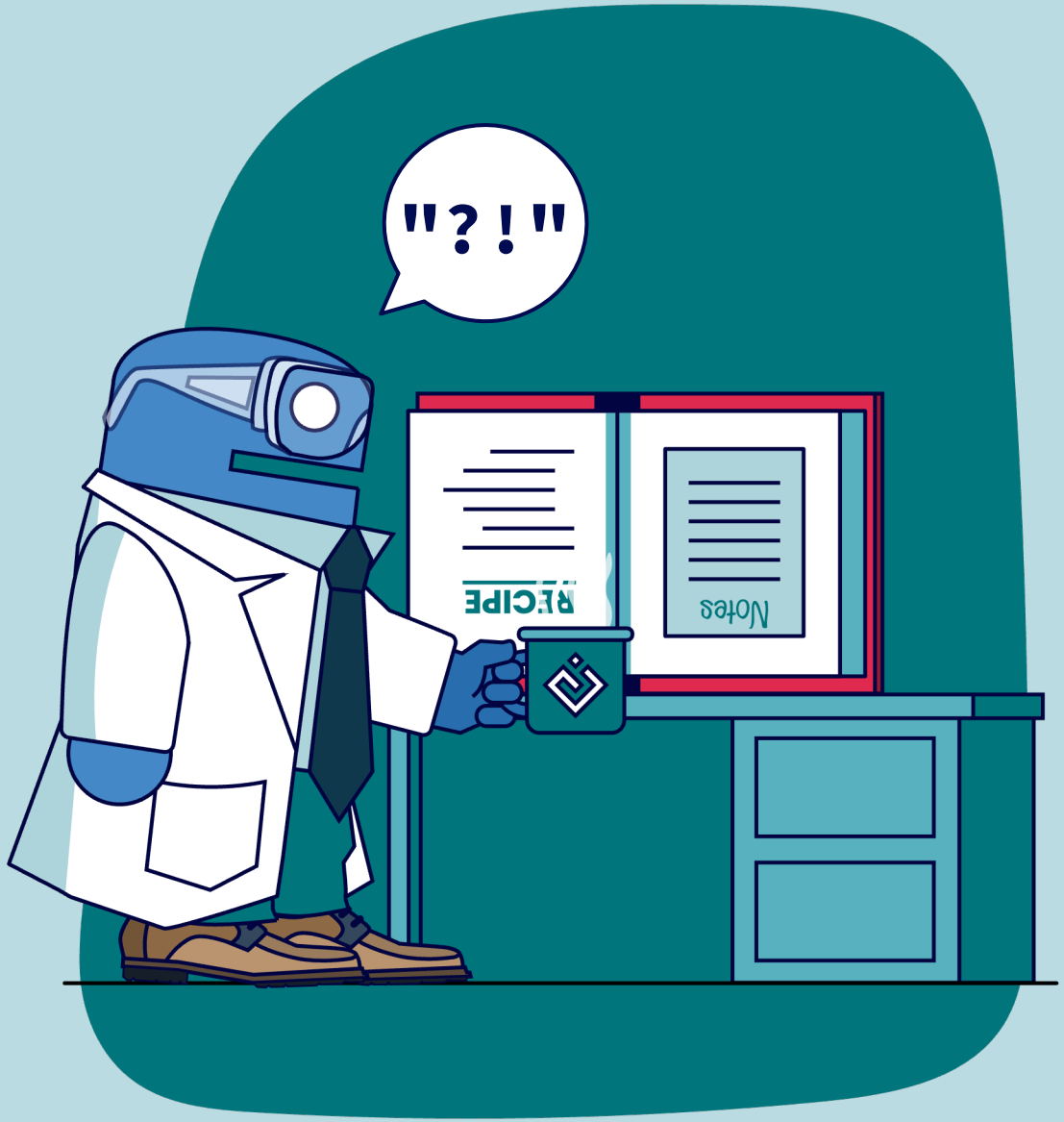


Coding Standards

Lecture 02
Readable Code

Write Readable Code

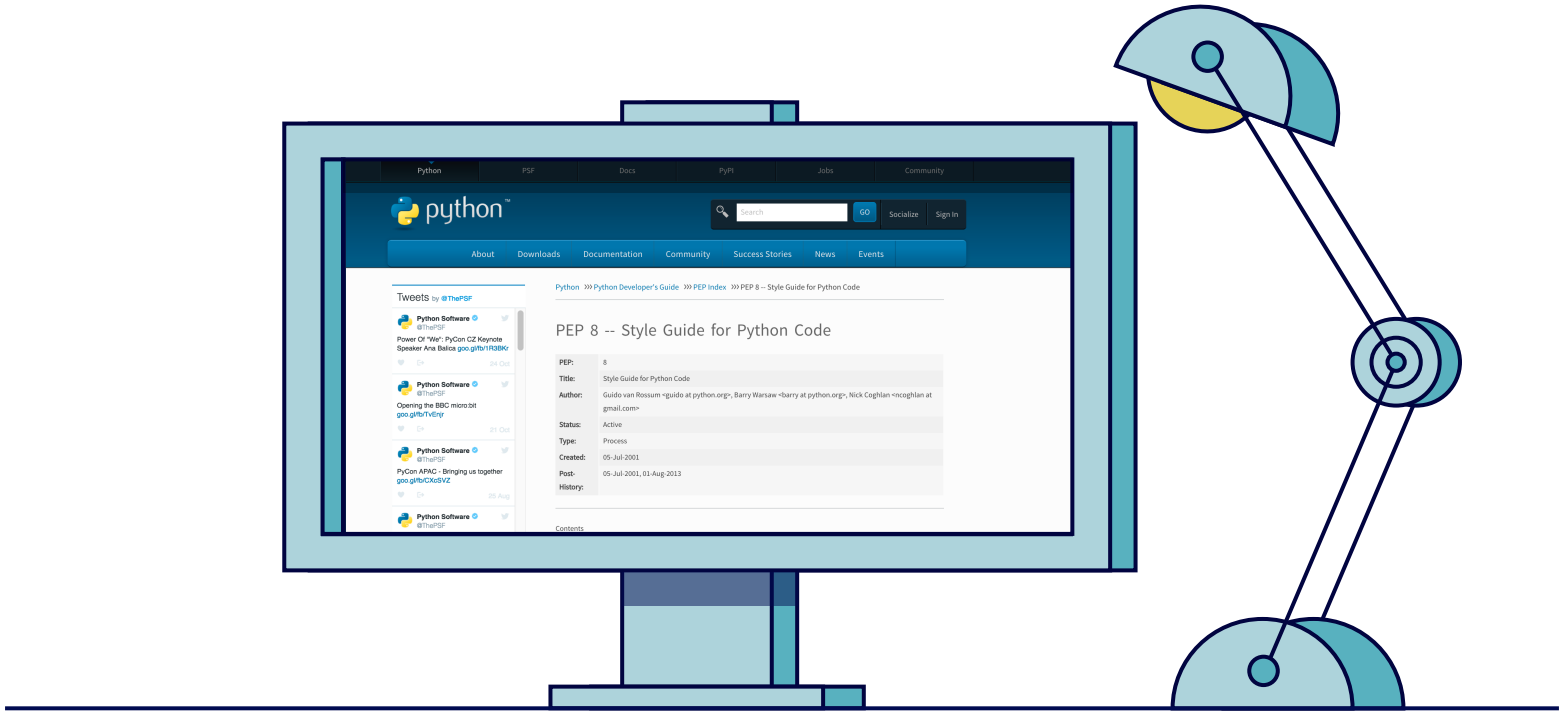
- Code that is **not readable** is **not maintainable**.
- Code that is not maintainable either dies or causes pain for many years.
- To help **make code readable**, maintainable, and shareable, there are coding standards for:
  - code layout (indentation, tabs/spaces, line length, etc.)
  - whitespace
  - comments
  - naming conventions
  - …
- These rules (conventions) **reduce cognitive load** and therefore improve developer productivity.



Python Coding Standard

The Python Coding Standard is defined in Python Enhancement Proposal 8 (PEP-8).

- *http://www.python.org/dev/peps/pep-0008/*



Indentation and Spaces

INDENTATION

Always use 4 spaces to indent code blocks.

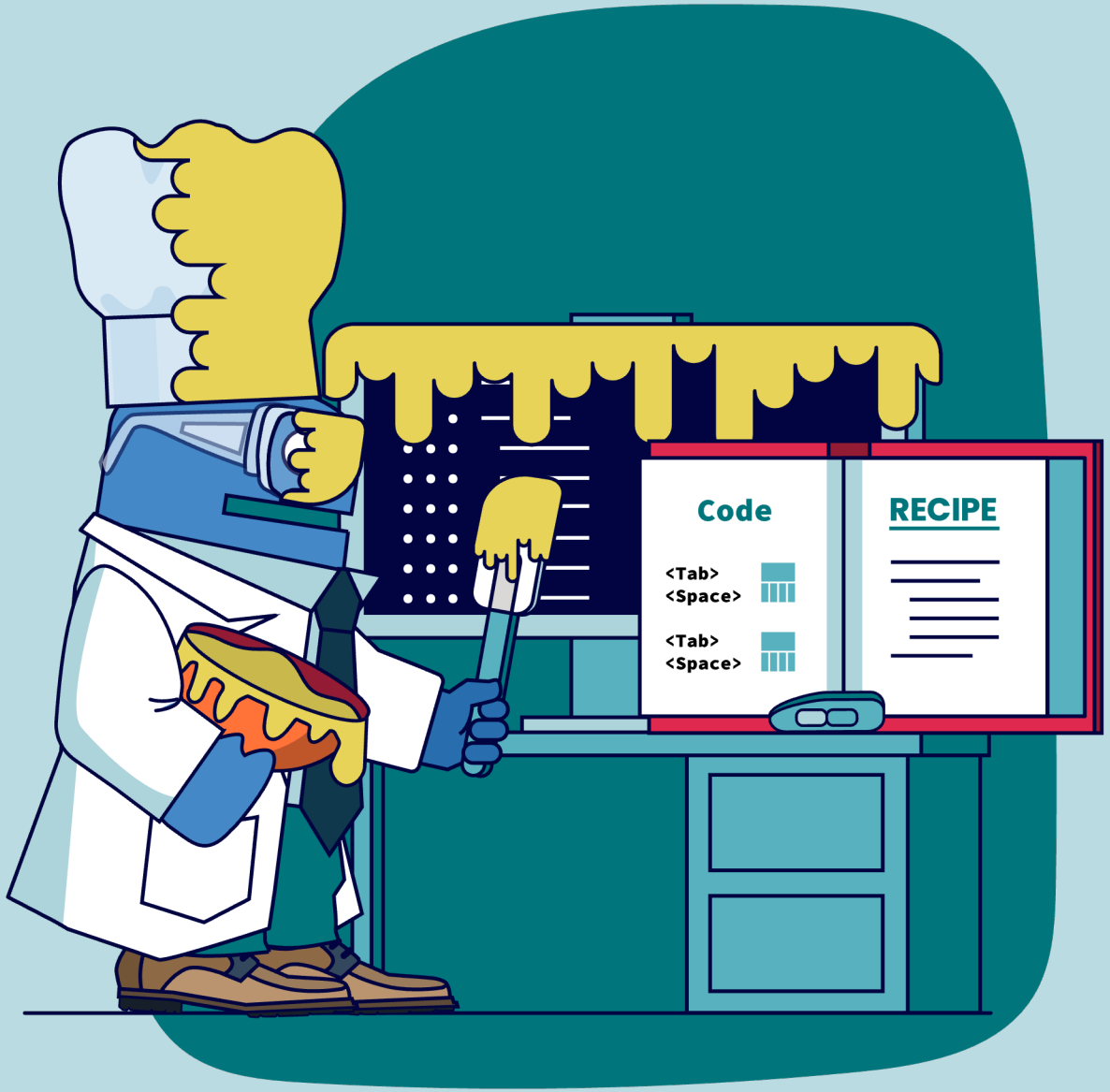
```
def sum(a):
    total = 0
    for value in a:
        total += value

    return total
```

4 spaces each

TABS OR SPACES?

Never mix tabs and spaces in a file.

Most editors (IDEs, vim, emacs, etc.) have a Python mode that will insert 4 spaces when you type <tab>.



Line Lengths and Wrapping

STANDARD LENGTH LINES

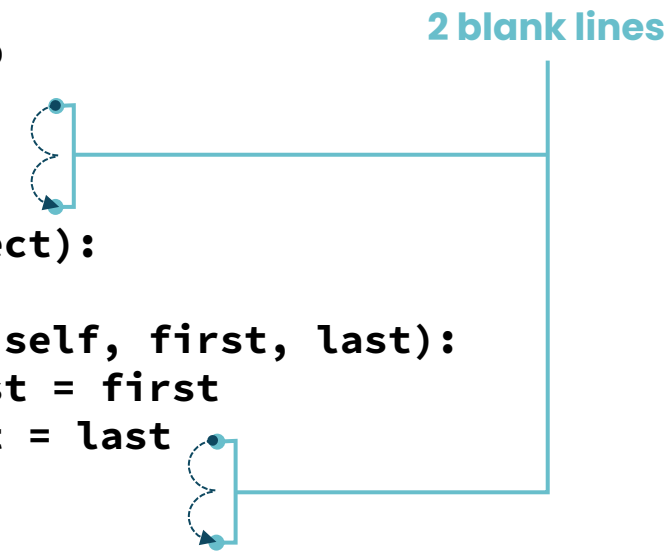
- Aim to limit all lines to a **standard maximum length**. The PEP-8 standard says 79 characters.
- However, if you are exclusively using modern-sized screens, longer lines can improve readability by reducing the need for line wrapping. Set a standard for your team.

```
# Use '\' at the end of lines to continue on the next line.
from numpy import array, sin, cos, allclose, zeros, ones, \
                  uint32, float32

# Code enclosed in () or [] or {} does not need the '\'.
if (instrument_volume > VOLUME_THRESHOLD and
    instrument_price  < PRICE_THRESHOLD):
    <do something>

# Align function arguments with opening delimiter.
a, b = some_long_function_call_name(with_long_var1,
                                    and_long_var2)
```

Line Spacing I

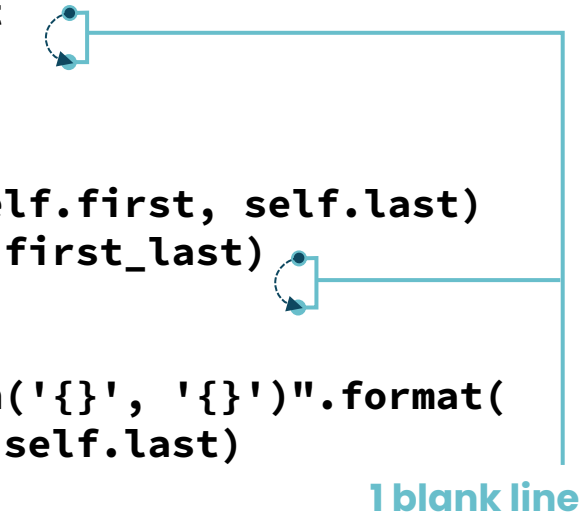
TOP LEVEL FUNCTIONS/CLASSES

```
# Separate top-level functions and classes
# with two blank lines.

def add(a, b):
    total = a + b
    return total


class Person(object):

    def __init__(self, first, last):
        self.first = first
        self.last = last


class City(object):

    def __init__(self, name, state):
        self.name = name
        self.state = state
```



CLASS METHODS

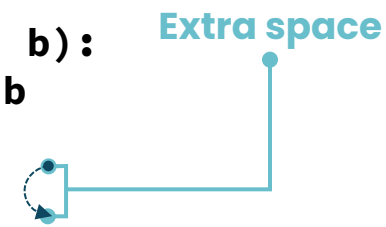
```
# Separate class methods by a single line

class Person(object):

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        first_last = (self.first, self.last)
        return ' '.join(first_last)

    def __repr__(self):
        string = "Person('{}', '{}')".format(
            self.first, self.last)
        return string
```



Line Spacing II

GROUPING FUNCTIONS

```
# Extra blank lines may be used (sparingly) to
# separate groups of related functions.

#--------------------------------
# Math functions
#--------------------------------

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b


#--------------------------------
# Name functions
#--------------------------------

def user_name(first, last):
    return first + last
```



ONE LINER FUNCTIONS

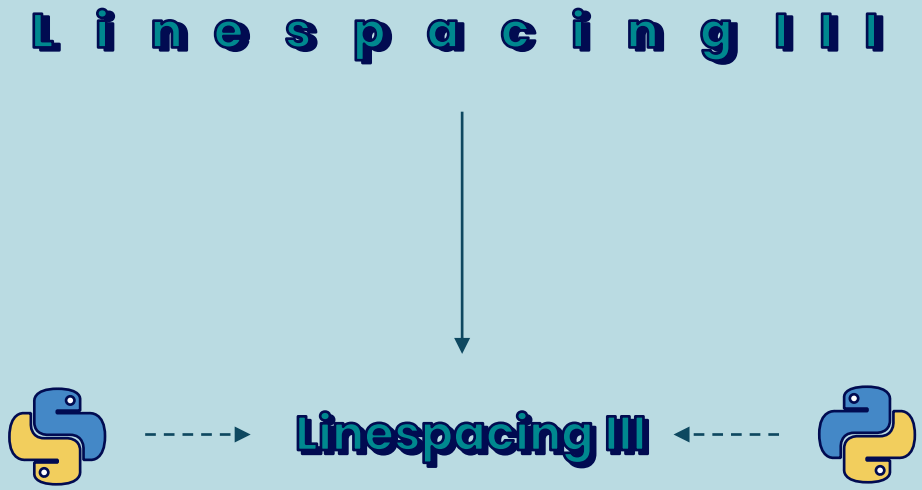
```
# Lines can be omitted between one-liner
implementations (such as dummy functions).

def stub1():
    pass
def stub2():
    pass
def stub3():
    pass
```

Line Spacing III

FUNCTION BODIES

```
# Use blank lines (sparingly) in functions
# to indicate logical sections.

def blend_color_channel(c1, c2, alpha):

    # Validation
    if not 0 <= alpha <= 1.0:
        raise ValueError("bad alpha")

    # Channel blending algorithm.
    new_c = c1 * alpha + \
            c2 * (1 - alpha)

    return new_c
```

Extra space



Whitespace

AROUND BINARY OPERATORS

```
# Binary operators are surrounded by a
# space on either side.

# Yes
if a > 1:
    b = 2 * (a + 1)


# No
if a> 1:
    b= 2 * (a+1)
```

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). For example, `x*x + y*y` and `(a+b) * (a-b)` are not considered a violation. Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

FUNCTION ARGUMENTS

```
# Don't put spaces around '=' in
# keyword arguments.

# Yes
def quad(x, a=0, b=1, c=0):
    y = a * x**2 + b * x + c
    return y

# No
def quad(x, a = 0, b = 1, c = 0):
    y = a * x**2 + b * x + c
    return y
```

Avoid Extraneous Whitespace

INSIDE (), [], OR {}

```
# Yes
spam(ham[1], {eggs: 2})

# No
spam( ham[ 1 ], { eggs: 2 } )
```

BEFORE COMMAS AND COLONS

```
# Yes
if x == 4:
    print x, y

# No
if x == 4 :
    print x , y
```

FUNCTION CALLS

```
# Yes
spam(1)

# No
spam (1)
```

SLICING AND INDEXING

```
# Yes
dict['key'] = list[index]

# No
dict ['key'] = list [index]
```

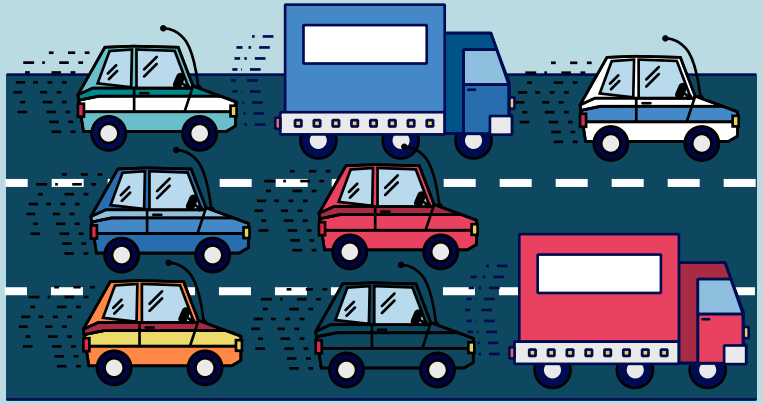
AROUND ASSIGNMENT

```
# PEP-8 encourages the first

# Yes
# No extra space
x = 1
y = 2
long_variable = 3

# No
# Align '=' character
x             = 1
y             = 2
long_variable = 3
```

Compound Statements

AVOID COMPOUND STATEMENTS

```
# Example 1

# Yes
func1()
func2()
func3()

# No
func1(); func2(); func3();

# Example 2

# Yes
if foo == 'blah':
    do_something()

# No (readability reduced)
if foo == 'blah': do_something()
```



flake8 – Tool For Style Guide Enforcement

The Python package **flake8** provides programmers with feedback on whether PEP-8 guidelines are being followed. It also reports code errors like missing imports, undefined variables, syntax errors, and many others.

- **flake8** can be invoked in different ways – on the command-line or as a code linter within an IDE. To invoke from the command line, use one of the following commands:

Command Prompt

```
flake8 <path_to_file_or_dir>
python -m flake8 <path_to_file_or_dir>
```

A narrowed scope can be provided by specifying an exact path to a file or directory of interest.

- **Problems are reported in the following format:**

```
file_name.py:line_number:character_number: Error Code<message>
```

```
code_check.py:3:2: E999 IndentationError: unexpected indent
```

flake8 – Code Checked

code_check.py

```
 1  # indentation problems
 2  a = 5
 3   b = 6
 4
 5  # white space problems
 6  c = a +(b* 5)
 7
 8  # import problems
 9  from math import sqrt
10  import os, sys
11
12
```

Command Prompt

```
(python-class)C:\Users\lthomas\Desktop>flake8 code_check.py
code_check.py:3:2: E999 IndentationError: unexpected indent
code_check.py:3:3: E111 indentation is not a multiple of four
code_check.py:3:3: E113 unexpected indentation
code_check.py:6:8: E225 missing whitespace around operator
code_check.py:9:1: E402 module level import not at top of file
code_check.py:10:1: E402 module level import not at top of file
code_check.py:10:10: E401 multiple imports on one line
code_check.py:11:1: W391 blank line at end of file
```

https://flake8.pycqa.org/en/latest/index.html https://www.flake8rules.com/

Imports I

SEPARATE LINES

```
# Imports should usually be on
# separate lines.

# Like this
import os
import sys

# NOT like this
import os, sys
```

MULTIPLE IMPORTS FROM A MODULE

```
# It is OK (and encouraged) to have
# items imported from a module on the
# same line.

from numpy import array, float32
```

AVOID IMPORT *

```
# Do not use import *

from numpy import *
```

Imports II

IMPORT AT TOP OF FILE

```
# Imports should happen at the top of a
# file so that others can quickly see the
# dependencies for a module.

# foo.py
import os
import sys

def some_function():
    pass

...
```

LOCAL IMPORTS

```
# It is OCCASIONALLY useful to break this
# rule to have "optional" dependencies in a
# module. Be careful about this.

#---------------------------------
# Statistical Functions
#---------------------------------
def mean(a):
    ...

def std(a):
    ...


#---------------------------------
# Statistical Plot Functions
#---------------------------------
def histogram_plot(a):
    from matplotlib.pyplot import plot
    ...
```

Imports III

GROUPING IMPORTS

```
# PEP-8
# Group imports in the following order:
# 1. Standard library imports
# 2. Related third-party imports
# 3. Application specific (local) imports

import os
import sys

import wx

import pricing_models
```

GROUPING IMPORTS (ANOTHER EXAMPLE)

```
# At Enthought, we expand the number of
# groupings and label them.
# 1. Standard library imports
# 2. Math/science libraries
# 3. Related third-party imports
# 4. Enthought specific libraries
# 5. Application specific (local) imports

# Standard Imports
import os

# Math Imports
from numpy import array

# Third-Party Imports
import wx

# Enthought Imports
from traits.api import Int

# Application Imports
import pricing_models
```

Absolute Imports

ALWAYS USE ABSOLUTE IMPORTS

Example directory structure for Python libraries.

```
python_library
├── README.md
├── doc
├── pyproject.toml
├── src
│   └── packagename
│       ├── __init__.py
│       ├── another_module.py
│       └── some_module.py
└── tests
```

Be aware of other conventions:

https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/

```
# PEP-8
# Relative imports of intra-package
# imports are highly discouraged.
# Use absolute package path for
# all imports.
```

another_module.py

```
# File: another_module.py

# Recommended (absolute)
from packagename.some_module import func

# Discouraged (relative)
from some_module import func

...
```

Give it a try! Readable Imports with isort

Explore **isort**, a tool that helps automate making Python imports more readable.

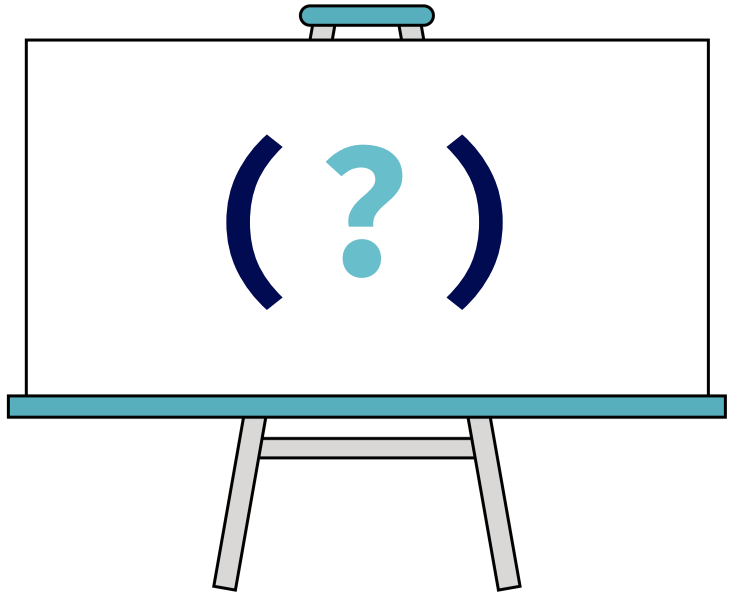
1. Open the file **exercises/tm_software/give_it_a_try/readable_imports.py** in your favorite editor and look at the imports. Review the imports. Think about what you would do to make it easier to see what packages are being imported? What classes are imported from **traitsui.api**?

2. Open a terminal (command prompt) and navigate to **exercises/tm_software/give_it_a_try**.

3. At the command line, run "**`python -m isort -d readable_imports.py`**". Compare the output you get to the original file.

4. At the command line, run "**`python -m isort -d readable_imports.py --float-to-top`**". Compare the output you get to the original file.

isort has many options to fine tune the import sorting scheme. These can be seen by running "**`python -m isort --help`**".



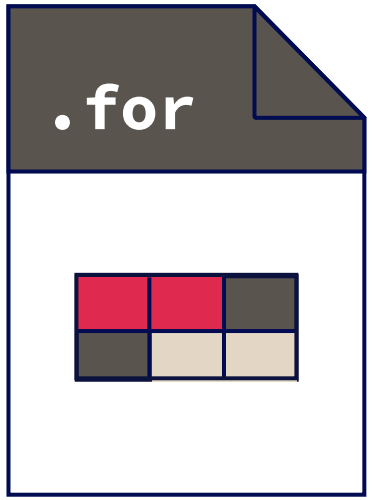
Naming Conventions

Lecture 02
Readable Code

Typical Scientific Naming Example

STRAIGHT FROM FFTPACK

```
SUBROUTINE CFFTB1 (N,C,CH,WA,IFAC)
      DIMENSION CH(*),C(*),WA(*),IFAC(*)
      NF = IFAC(2)
      NA = 0
      L1 = 1
      IW = 1
      DO 116 K1=1,NF
         IP = IFAC(K1+2)
         L2 = IP*L1
         IDO = N/L2
         IDOT = IDO+IDO
         IDL1 = IDOT*L1
         IF (IP .NE. 4) GO TO 103
         IX2 = IW+IDOT
         IX3 = IX2+IDOT
         IF (NA .NE. 0) GO TO 101
         CALL PASSB4 (IDOT,L1,C,CH,WA(IW),WA(IX2),WA(IX3))
         GO TO 102
     <and on and on for 368 lines...>
```



Primary Naming Consideration

Priority 1: A variable name should fully and accurately describe the entity and variable it represents.

POOR NAMING CHOICES

```
# Update Cash Balance after stock trade.
c1 = n * ip
c2 = c1 + compute_tc(ins, n)
b -= c2
```

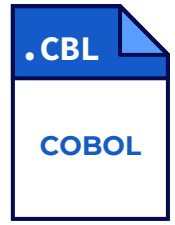
DESCRIPTIVE NAMING CHOICES

```
# Update Cash Balance after stock trade.
instrument_cost = instrument_quantity * instrument_price
trade_cost = instrument_cost + transaction_cost(instrument_name, instrument_quantity)
cash_balance -= trade_cost
```

Optimal Name Lengths

Studies on debugging COBOL indicate:

- 10-16 character names is optimal length.
- 8-20 characters is almost as good.



Gorla, N., A. C. Benander, and B. A. Benander. 1990. "Debugging Effort Estimation Using Software Metrics." IEEE Transactions on Software Engineering SE-16, no. 2 (February): 223-231

TOO LONG	TOO SHORT	JUST RIGHT
`risk_free_interest_rate`	`r, rfir, rate`	`risk_free_rate, rsk_fr_rate`
`length_moving_average_window`	`N, nw`	`win_len, window_length, nwindow, mov_avg_win_len`

Domain vs. Computing Names

Variable names should relate to the **problem space**, not generic programming terminology.

COMPUTING VS DOMAIN

```
# Not so good:  Algorithm-centric names
sum = 0
for record in record_list:
    sum += will_retire(record.name)

# Better:  Domain specific names
gold_watch_orders = 0
for employee in employee_list:
    gold_watch_orders += will_retire(employee.name)
```

It's also a good idea to not use names that exist in Python's builtin namespace. Here, **`sum`** would override the builtin **`sum()`** function which would likely be problematic.

Using *Extremely* Short Names

When dealing with industry standard variables in *small* and *local* contexts, extremely short names work. Here, short names short for three reasons:

1. Because of the both small and local context
2. This is a physics problem, and the variable names use standard physics conventions (i.e., domain names)
3. The docstring that deciphers the variables is right above the short code that uses them

```
def sin_wave(t, a=1, w=2*pi, phi=0):
    """
    Return a sine wave form for time t.

    Inputs
    ------
    t: time in seconds
    a: amplitude scale factor
    w: frequency in radians/second
    phi: phase shift in radians

    Returns
    -------
    y: sin wave output
    """

    y  = a * sin(w*t + phi)
    return y
```

Naming Boolean Variables

GENERIC NAMES

```
# Try not to use generic names as they
# don't provide much information
flag = True
status = True
```

COMMON BOOLEAN NAMES

```
# These common names are more informative.

done = True
found = True
success = True
valid = True
error = True
ok = True

# Using the prefix 'is_' can help identify
# booleans.

is_done = True
```

POSITIVE VALUES

```
# Positive values are easy to interpret

if found:
    employee_of_month = employee

# Negative Booleans take more mental
# gymnastics to read

if not not_found:
    employee_of_month = employee
```

Positive Values Demonstration

Quick! Raise your hand if you are allowed to turn left.

Naming Containers

When naming lists (or sequences) of elements, use the plural of the element name or a suffix that indicates it is a container.

VARIABLE SUFFIX FOR CONTAINERS

```
# Using the _list suffix is descriptive
# and is easily differentiated from a
# single item.

for name in name_list:
    print(name)
```

Using "reverse notation" when naming container elements (**elements**, **elements_active**, **elements_defunct** rather than **elements**, **active_elements**, **defunct_elements**) can help with searching for variables in the namespace and reducing cognitive load for remembering *how* a variable was named (**inactive_***, **defunct_***, **dormant_***, etc.).

PLURAL VARIABLE NAME FOR CONTAINERS

```
# Using the plural version of the variable
# is also a common paradigm.

for name in names:
    print(name)

# For long variables names, the singular
# and plural approach isn't as good.  It is
# too hard to differentiate between the
# two.

close_price = {}
for equity_symbol in equity_symbols:
    close_price[equity_symbol] =
lookup_daily_close(equity_symbol)

# And this is obviously problematic...
for moose in moose:
    moose.bellow()
```

Case Conventions for Variables

VARIABLES

```
# Use lower_case notation without
# capital letters for variables in
# your functions, methods, and
# scripts.

# Yes
lower_case = 3

# No
CamelCase = 3
mixedCase = 3
Mixed_Case = 3
```

GLOBAL VARIABLES

```
# Use UPPER_CASE notation for module
# level "constants" or "static" class
# variables.

# File: foo.py

# Globals

# Yes
UPPER_CASE = 3

# No
CamelCase = 3
mixedCase = 3
Mixed_Case = 3

def some_function(a):
    result = a * UPPER_CASE
    return result
```

Case Conventions for Functions & Classes

FUNCTIONS AND METHODS

```
# Functions and methods should use
# lower_case names with underscores.
#
# Do not use the Java convention of
# mixedCase.

# Yes
def some_function(a):
    pass

# No
def SomeFunction(a):
    pass

def someFunction(a):
    pass
```

CLASSES

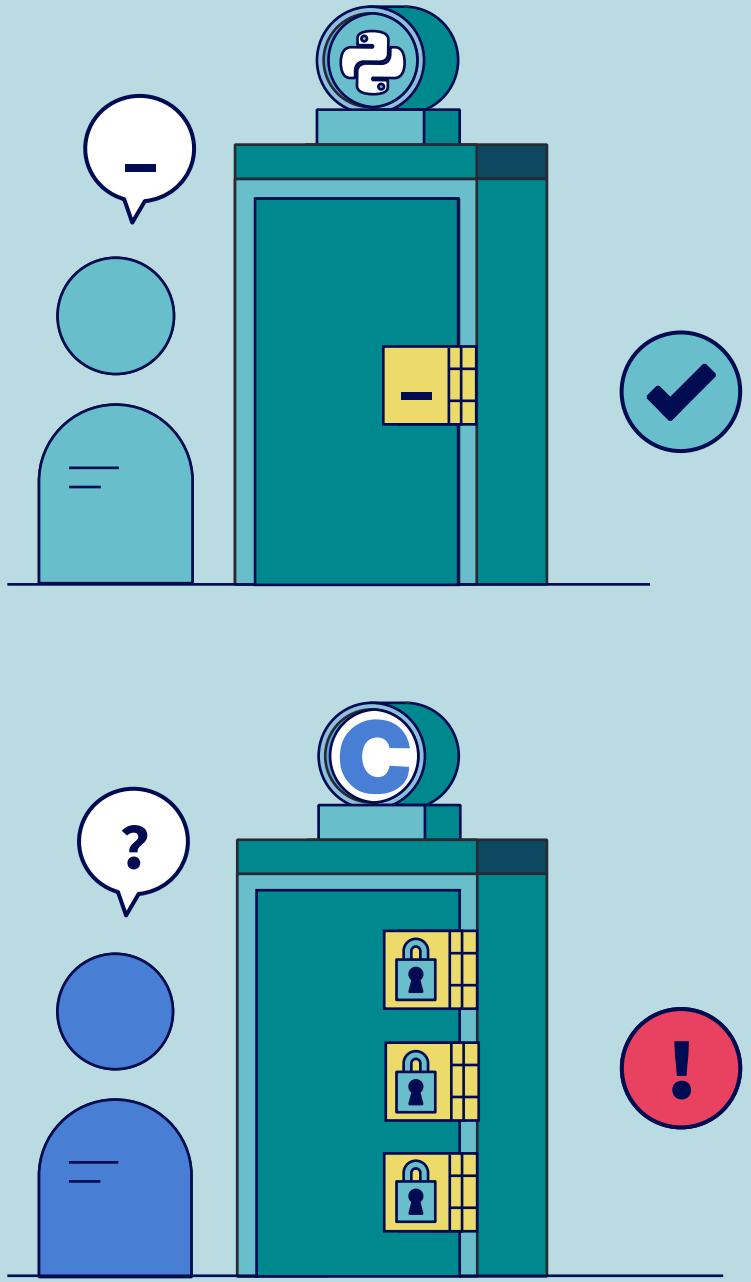
```
# Class names are CamelCase.
#
# If there is one class in a file and
the class is named FooBar, the file
should be named foo_bar.py.

# file: foo_bar.py

# Yes
class FooBar:

    def some_method(self):
        pass

# No
class foo_bar:

    def some_method(self):
        pass
```

Python Function & Class Privacy Conventions

PRIVATE CLASSES AND FUNCTIONS

```
# Names of 'private' classes, methods,
# and functions start with a single
# underscore.
#
# Note that Python does not enforce
# function or class privacy.

def _private_function(a):
    pass

class _PrivateClass(object):
    pass
```

https://docs.python.org/3.8/tutorial/classes.html#private-variables



Naming Packages and Modules

MODULE NAMES

```
# Module names should be lower case
# with underscores.

# Yes
foo_bar.py

# No
FooBar.py
```

PACKAGE NAMES

```
# Package directories should be all
# lower case alpha-numeric characters.
#
# Avoid underscores unless absolutely
# necessary.

# Yes
packagename

# No
PackageName
package_name
```

Recommended Directory Structure

PACKAGE/MODULE/TESTS LAYOUT

Example directory structure for Python libraries.

Command Prompt

```
python_library
├── README.md
├── doc
├── pyproject.toml
├── src
│   └── packagename
│       ├── __init__.py
│       ├── another_module.py
│       └── some_module.py
└── tests
    ├── __init__.py
    ├── test_another_module.py
    └── test_some_module.py
```

File: pyproject.toml

```
[project]
name = "packagename"
version = "0.0.1"
readme = "README.md"


[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

```
> pip install -e .
```

https://packaging.python.org/en/latest/tutorials/packaging-projects/

Foolish Consistency and PEP-0008

https://peps.python.org/pep-0008/#a-foolish-consistency-is-the-hobgoblin-of-little-minds

- "**A style guide is about consistency.** Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important."

- "However, **know when to be inconsistent** – sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best."

Exercise: Readable Code

- Code Check

Lecture 03

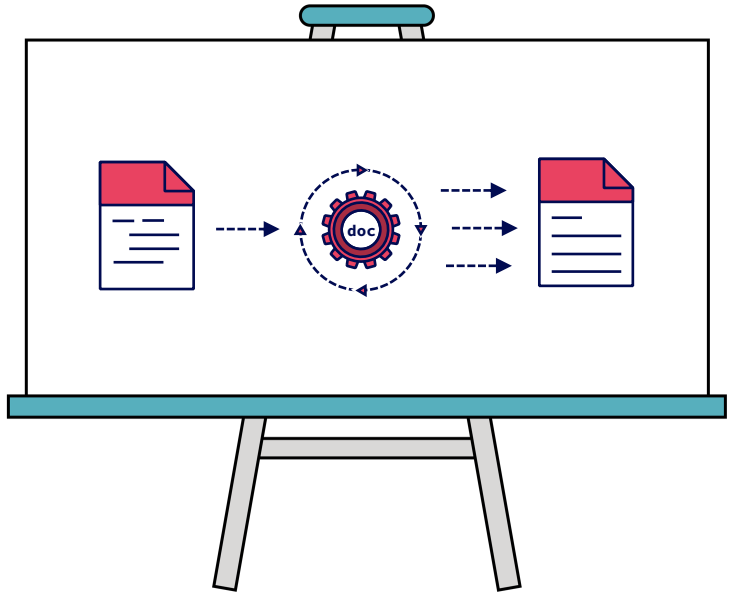
Documenting Code

Software Engineering for Scientists and Engineers

Discussion

Isn't all code self-documenting?

Table of Contents



Self-Documenting Code

Lecture 03
Documenting Code

Self-Documenting Code

Self-documenting code has the following characteristics:

- **Descriptive Variable, Function, and Class Names**
- **Clear Layout**
  - Python encourages this with indentation and other white space conventions.
- **Good Design**
  - Limited levels of indentation.
  - Functions that do one task and do it well.
  - Classes that provide good abstractions.
- **Useful Comments**
  - Describe intent of code (*why* not *what*).
  - Inform readers of algorithm design.
  - Flag unusual usages.

Self-Documenting Code: Example 1

SELF DOCUMENTING QUALITY RUBRIC

- **Names**
  - Are the names in this block informative?
- **Layout**
  - Are logical blocks of code grouped together?
- **Design**
  - How deep is the indentation?
  - Are there overly complex blocks of logic?
  - Are there overly clever snippets of code?
- **Comments**
  - Are comments explanatory?

Does reading this code **obfuscate** or **clarify**?

```
# Snippet out of a draw() method for drawing
# "bulls eye" cross hairs at the mouse
# position on a plot.

plot = self.component
if plot is None:
    return

# sx, sy: mouse position in screen space.
# Returns None if outside the screen space.
sx, sy = plot.map_screen(self.current_position)

if plot.orientation == "h" and sx is not None:
    self._draw_vertical_line(gc, sx)
elif sy is not None:
    self._draw_horizontal_line(gc, sy)

if self.show_marker:
    self._draw_marker(gc, sx, sy)
```

Python Docstrings

API DOCUMENTATION STRINGS

- API documentation provides users of code with information about functions, classes, and modules.
  - For **functions** it provides a description of the function, its arguments, and what it returns.
  - For **classes** it provides a description of what is being modelled.
  - For **modules**, it provides an overview of the contents of the module and their expected use.
- The first string encountered in a Python function, class, or module (before any code) is interpreted as a docstring. Typically, this is a multi-line triple-quoted string.
- The docstring connects code to Python's **help()** function.

API DOCUMENTATION [FROM NUMPY]

```
def interp(x, xp, fp, left=None, right=None):
    """
    One-dimensional linear interpolation.

    Returns the one-dimensional piecewise
    linear interpolant to a function with
    given values at discrete data-points.

    Parameters
    ----------
    x : array_like
        The x-coordinates of the
        interpolated values.
    <snip>
    Returns
    -------
    y : float or complex (for `fp`) or ndarray
        Interpolated values, same shape as `x`.
    <snip>
    """
```

Styles of API Documentation

SIMPLE SPECIFICATION

Use simple docstrings for simple functions.

```
# from Python standard library locale.py

# def str(val):
#     """Convert float to string, taking the
#     locale into account."""
#     return format("%.12g", val)

>>> import locale

>>> locale.setlocale(locale.LC_NUMERIC,
...                  'fr')
>>> locale.str(7.2)
'7,2'
```

FORMAL SPECIFICATION

Use formal specifications for complex functions.

```
def interp(x, xp, fp, left=None, right=None):
    """
    One-dimensional linear interpolation.

    Returns the one-dimensional piecewise
    linear interpolant to a function with
    given values at discrete data-points.

    Parameters
    ----------
    x : array_like
        The x-coordinates of the
        interpolated values.
    <snip>
    Returns
    -------
    y : float or complex (for `fp`) or ndarray
         Interpolated values, same shape as `x`.
    <snip>
    """
```

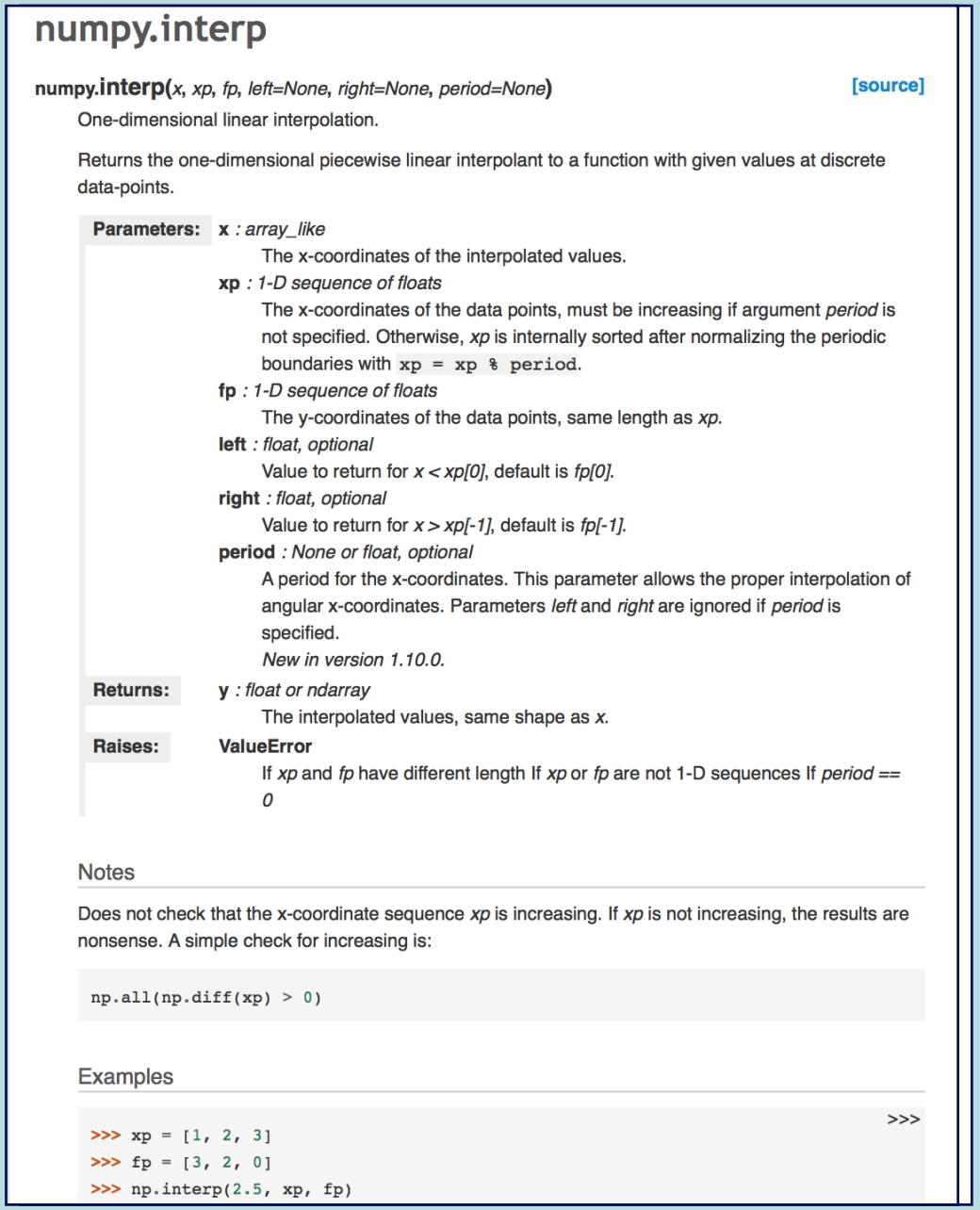
Formal Specifications

BENEFITS

- Formalism helps developers know what to document.
- Strive for a uniform look to documentation.
- Tools like **Sphinx** can generate nice HTML and PDF docs automatically.

DRAWBACKS

- Can add (significant) overhead for creating new functions.
- Overhead can inhibit developers from re-factoring and creating new functions when they should.



numpy.interp

numpy.interp(*x*, *xp*, *fp*, *left=None*, *right=None*, *period=None*) [source]

One-dimensional linear interpolation.

Returns the one-dimensional piecewise linear interpolant to a function with given values at discrete data-points.

Parameters:
x : *array_like*
The x-coordinates of the interpolated values.
xp : *1-D sequence of floats*
The x-coordinates of the data points, must be increasing if argument *period* is not specified. Otherwise, *xp* is internally sorted after normalizing the periodic boundaries with `xp = xp % period`.
fp : *1-D sequence of floats*
The y-coordinates of the data points, same length as *xp*.
left : *float, optional*
Value to return for *x < xp[0]*, default is *fp[0]*.
right : *float, optional*
Value to return for *x > xp[-1]*, default is *fp[-1]*.
period : *None or float, optional*
A period for the x-coordinates. This parameter allows the proper interpolation of angular x-coordinates. Parameters *left* and *right* are ignored if *period* is specified.
New in version 1.10.0.

Returns:
y : *float or ndarray*
The interpolated values, same shape as *x*.

Raises:
ValueError
If *xp* and *fp* have different length If *xp* or *fp* are not 1-D sequences If *period == 0*

Notes

Does not check that the x-coordinate sequence *xp* is increasing. If *xp* is not increasing, the results are nonsense. A simple check for increasing is:

```
np.all(np.diff(xp) > 0)
```

Examples

```
>>>
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
```

Quality Production Python Code

Percentage of Python source lines that are comment or docstring lines:

Project	Comments
`scipy`	39.7%
`ipython`	36.2%
`matplotlib`	35.2%
`numpy`	35.0%
`traits`	34.3%
`chaco`	26.2%
`pandas`	22.3%

* As determined by cloc: http://cloc.sourceforge.net on master versions as of 2021-02-24.

Self-Documenting Code: Example 2

SELF DOCUMENTING QUALITY RUBRIC

- **Names**
  - Are the names in this block informative?
- **Layout**
  - Are logical blocks of code grouped together?
- **Design**
  - How deep is the indentation?
  - Are there overly complex blocks of logic?
  - Are there overly clever snippets of code?
- **Comments**
  - Are comments explanatory?
- **Docstrings**
  - Are there docstrings?

Does reading this code **obfuscate** or **clarify**?

```
# Code snippet out of a Enthought Envisage
# library written by Martin Chilvers.

def unregister_services(self):
    """ Unregister any service offered by the
        plugin.
    """

    # Services must be unregistered in reverse
    # order that they were registered in.
    service_ids = self._service_ids[:]
    service_ids.reverse()

    app = self.application
    for service_id in service_ids:
        app.unregister_service(service_id)

    # Reset services in case the plugin is
    # started again.
    self._service_ids = []
```

Style Guides

Many organizations have Python style guides for self-documenting code that build on PEP-8. This helps them with consistency across the enterprise and generally improves the quality of their code.

You don't have to create your own from scratch. Instead use:

- NumPy : *https://numpydoc.readthedocs.io/en/latest/format.html*
- Google : *https://google.github.io/styleguide/pyguide.html*

If these don't work for you, use them as a baseline.

Give it a try! Self Documenting Code

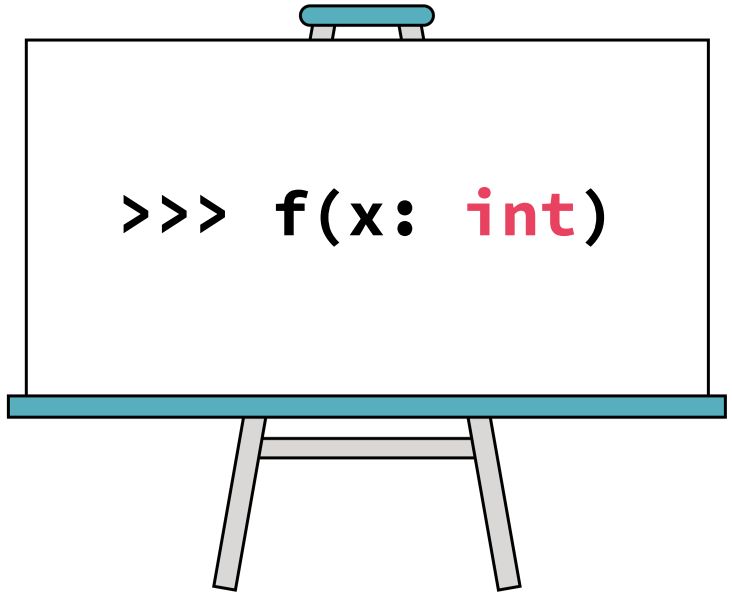
1. Open the file **exercises/tm_software/give_it_a_try/self_documenting_code.py** in your favorite editor and read the code. Specifically look at the **names**, **layout**, **design**, **comments**, and **docstrings**. Do they clarify or obfuscate? What does this code do?

2. Open a terminal (command prompt) and navigate to **exercises/tm_software/give_it_a_try**.

3. Start **IPython** and execute the command **%run self_documenting_code.py**.
   Try running the following statements:

```
help(ext)  # is there a docstring?
ext("upskilling@enthought.com") # can you guess what this function does?
```

4. Use the above investigation to rewrite the function to be self-documenting. Make sure you add a docstring. Run **help()** on the function you rewrote.



Type Annotations

Lecture 03
Documenting Code

What are Type Annotations?

In Python, type annotations indicate the *data types* of **variables**, **arguments** for functions and methods, as well as **return values** for functions and methods.

Used judiciously, type annotations increase code readability, improves its self-documentation, and helps developers understand code.

Type annotations are **optional** in Python. Use them when you find them helpful.

These annotations are useful for writing and maintaining code. However, the Python interpreter will **not** report violations of the types given – Python does not check any of this at runtime. Instead, you use static type-checkers from the command line or in your editor.

Type Annotation Examples

SIMPLE ANNOTATIONS

```
# {variable_name}: {type} = {value}

# Simple integer
x: int = 5

# Dictionary with string keys and integer
# values
items: dict[str, int] = {'apple': 5,
                         'orange': 3}

# Function with arguments
def func(x: float, y: int):
    pass

# Function with arguments and return
def func(x: float, y: int) -> float:
    return x + y
```

ADVANCED ANNOTATIONS

```
from typing import Dict

# Named types
Element = str
AtomicNumber = int
PeriodicTable = Dict[Element, AtomicNumber]

pt: PeriodicTable = dict()

pt['H'] = 1

# [mypy] Invalid index type "int" for
# "Dict[str, int]; expected type "str"
pt[2] = 'He'

# [mypy] Incompatible types in assignment
# (expression has type "str", target has
# type "int")
pt['Li'] = '3'
```

Type Annotation Examples

PYTHON 3.8

```
from typing import Dict, Optional, Union



Element = str
AtomicNumber = int
PeriodicTable = Dict[Element, AtomicNumber]

pt: PeriodicTable = dict()

# Function with arguments
def func(x: Union[float, str], y: Optional[int]):
    pass

# [mypy] Argument 1 to "func" has incompatible
# type "complex"; expected "Union[float, str]"
func(2+1j, y=None)

# [mypy] Argument 2 to "func" has incompatible
# type "str"; expected "Optional[int]"
func(3.5, 'a')
```

ADVANCED ANNOTATIONS (PYTHON 3.9+)

```
# Since Python 3.9: can use built-in types
# in annotations (including dict)
# Since Python 3.10: unions can now be written
# as X | Y (union type expressions)

Element = str
AtomicNumber = int
PeriodicTable = dict[Element, AtomicNumber]

pt: PeriodicTable = dict()

# Function with arguments
def func(x: float | str, y: int | None):
    pass

# [mypy] Argument 1 to "func" has incompatible
# type "complex"; expected "Union[float, str]"
func(2+1j, y=None)

# [mypy] Argument 2 to "func" has incompatible
# type "str"; expected "Optional[int]"
func(3.5, 'a')
```

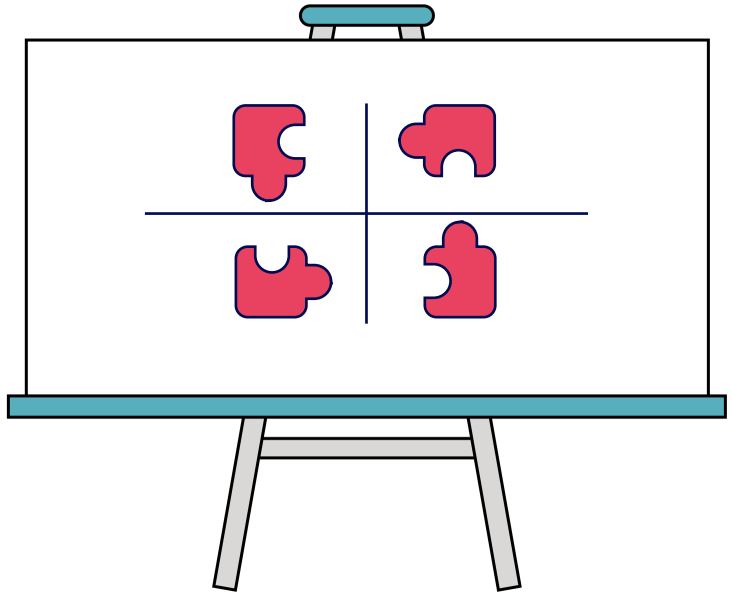
mypy

One type annotation tool that you can use is called **mypy**. This can be used to check the validity of your type annotations and can be integrated into many IDEs.

script.py

```
1  x: int = 5
2  x = 6
3  x = 'hello'
4  def greeting(word: str, count: int) -> str:
5     return word * count
6  greeting('hello', 3)
7  greeting(3, 'hello')
```


```
$ mypy script.py
script.py:3: error: Incompatible types in assignment (expression has type "str", variable
has type "int")
script.py:7: error: Argument 1 to "greeting" has incompatible type "int"; expected "str"
script.py:7: error: Argument 2 to "greeting" has incompatible type "str"; expected "int"
```



Diátaxis

Lecture 03
Documenting Code

Developer Documentation : Four Quadrants

https://diataxis.fr/

Developer Tutorials

Diátaxis: **Learning-Oriented**, **Practical steps**, **Most useful when we're studying**

- Oriented to beginners
  - Generally, a hands-on recipe approach (i.e., first do this, then do this, then ...)
  - Assume little prior knowledge
  - Intended to remove barriers to entry
- For example, Getting started tutorials typically cover:
  - How to install
  - Introduce main concepts
  - Curated examples showing basic usage

Developer How-To Guides

Diátaxis: **Problem-Oriented**, **Practical steps**, **Most useful when we're working**

- Oriented toward solving frequently encountered problems or tasks
  - Generally, a hands-on recipe approach (i.e., first do this, then do this, then ...)
  - Assume basic knowledge of system and encountering common tasks with associated barriers
  - Intended to remove barriers to getting common tasks done

Developer Explanation

Diátaxis: **Understanding-Oriented**, **Theoretical knowledge**, **Most useful when we're studying**

- Oriented to more advanced developers
  - Generally, an architectural approach
  - Focused on concepts, not on details
  - Assume familiarity with the package
  - Intended to deepen understanding of code design and policy choices

Developer (API) Reference

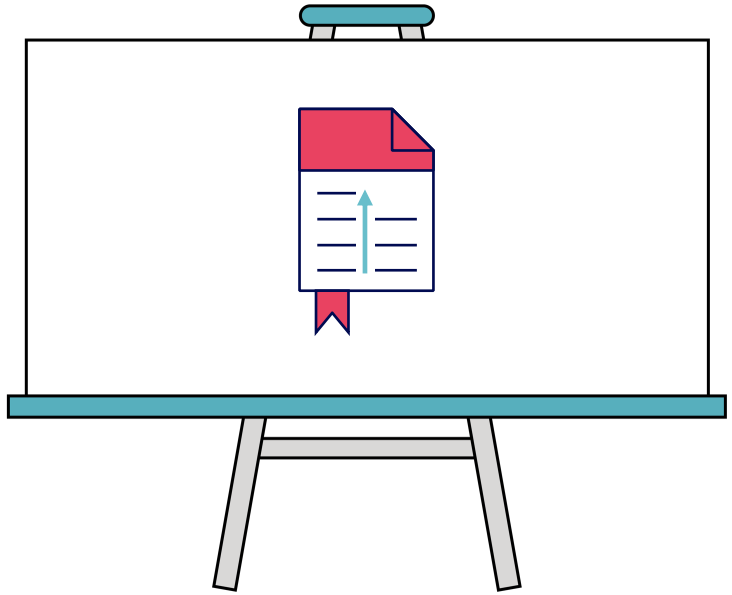
Diátaxis: **Information-Oriented**, **Theoretical knowledge**, **Most useful when we're working**

- Intended primarily for other developers.
- In Python, most reference documentation is done by creating useful docstrings in modules, classes, and functions.
- Ways to access:
  - Python **help()**
  - IPython **?**
  - Python (The Python Standard Library): https://docs.python.org/3/library/index.html
  - 3rd Party Packages
  - Open Source Repository for Package (GitHub, GitLab, etc.)

Application Documentation

If you are working on a complete application that will have **non-developer users**, you will also need to consider the needs of end-users. The same four diátaxis quadrants apply, but with a focus on **end-users**:

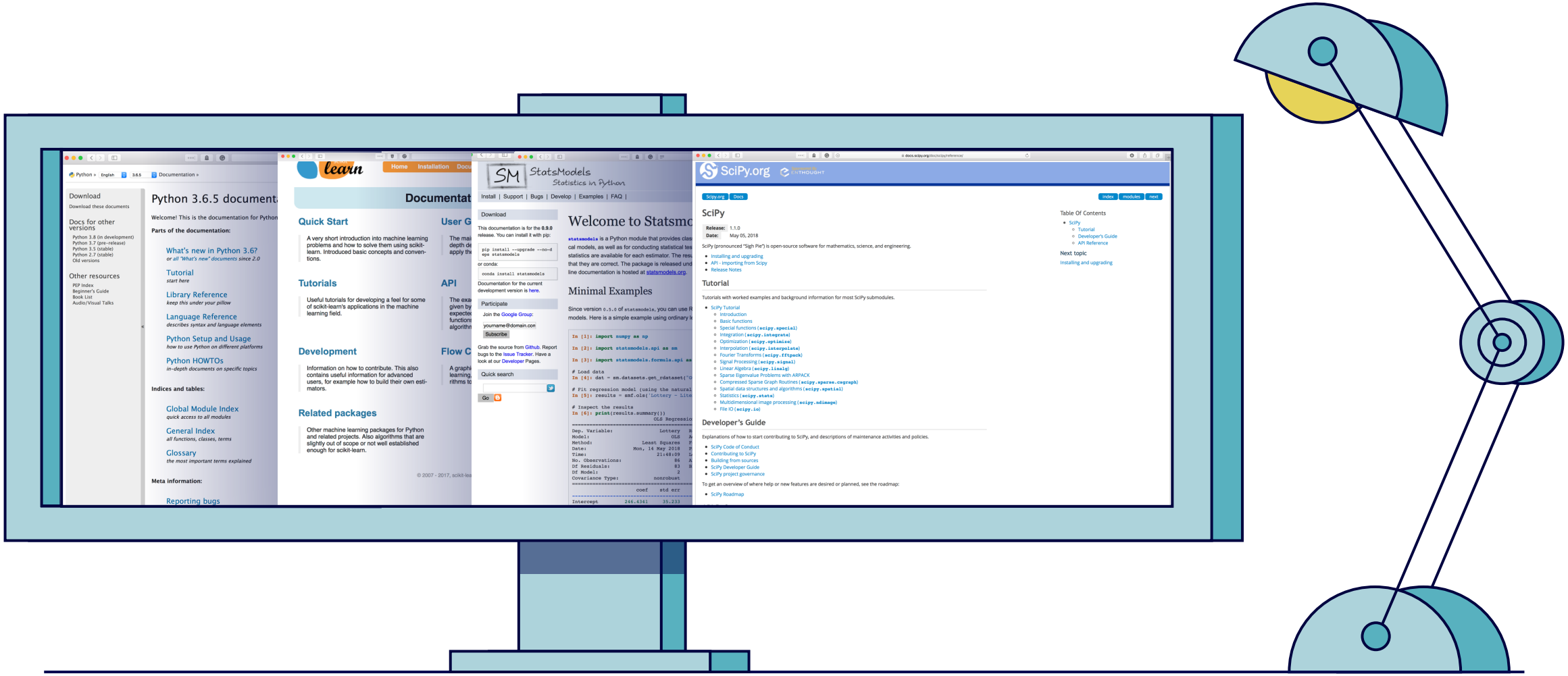
- **Tutorials** : Installation, Running, Basic Tasks
- **How-To Guides** : How to Accomplish Common Tasks, Updating
- **Explanation** : Overview of Application and its Purpose
- **Reference** : For complex applications, may need reference on options, settings, menus, and other elements of the application.



Publishing Documentation

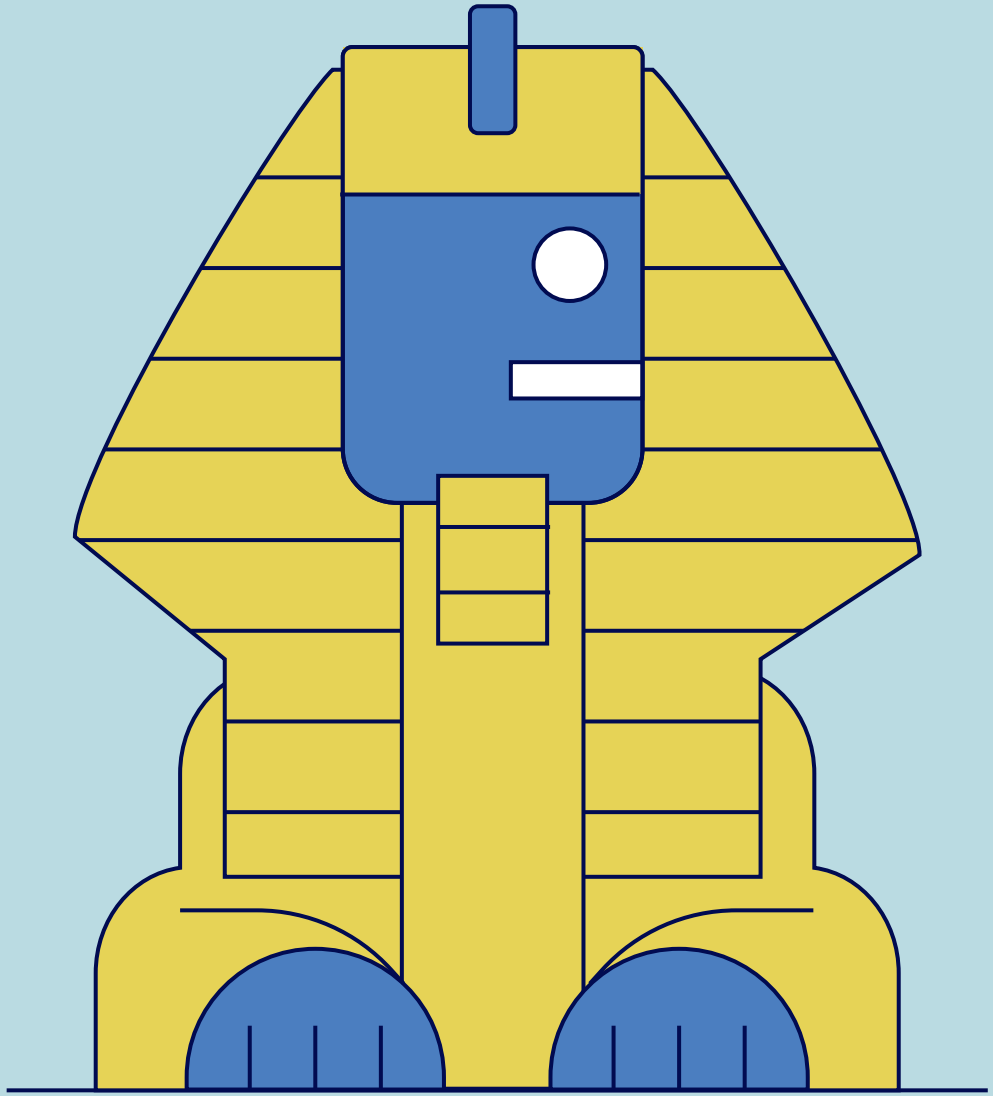
Lecture 03
Documenting Code

Sphinx – A Documentation Generator



Sphinx

- Sphinx is a widely used documentation generator for Python.
- It is used at **`docs.python.org`**, and in many Python projects.
- Output formats include:
  - HTML,
  - PDF (using LaTeX),
  - manual pages
  - and more.
- Documentation markup language is reStructuredText.
- Can generate API documentation automatically.



Sphinx

EXAMPLE

Suppose we have a Python package called **vectorlib**, organized as follows:

```
toplevel/
    pyproject.toml
    vectorlib/
        __init__.py
        vector.py
        util.py
```

- We'd like to create HTML documentation for this package.
- Next, take a look at the code to the right.

The accompanying files for this example can be found in **demo/Software_Engineering/sphinx/demo/**

vector.py

```
"""This module defines the Vector class. """
from math import acos, sqrt

class Vector(object):
    def __init__(self, x, y, z):
        """ Constructor method.
        """
        self.x = x
        self.y = y
        self.z = z

    def dot(self, v):
        """Dot product with Vector *v*."""
        d = self.x * v.x + self.y * v.y + self.z * v.z
        return d

    def abs(self):
        """Returns the magnitude of the vector."""
        m = sqrt(self.x**2 + self.y**2 + self.z**2)
        return m

    def __repr__(self):
        s = ("Vector(x=%s, y=%s, z=%s)" %
             (self.x, self.y, self.z))
        return s
```

Sphinx

util.py

```
"""
Some utilities for working with Vector instances.
"""

from vectorlib.vector import Vector


def print_vectors(veclist):
    """Print a list of vectors.

    *veclist* must be a list of Vector
        instances.
    """
    for vec in veclist:
        print(vec)
```

After executing **sphinx-quickstart**, compare the autogenerated **doc/conf.py** file to the sample **doc/conf.py.sample** file provided. Be sure to check the list of **extensions** and the list of **exclude_patterns**.

sphinx-quickstart

At the command line, navigate to
demo/Software_Engineering/sphinx/demo/

You should see a structure like:

```
├── README.rst
├── doc
│   └── conf.py.sample
├── pyproject.toml
└── vectorlib
    ├── __init__.py
    ├── util.py
    └── vector.py
```

From the **demo** folder install the library so it's importable:

```
pip install -e .
```

The **doc/conf.py.sample** file is an example configuration file. A true configuration file (**doc/conf.py**) will be generated using the below process.

Navigate to the **doc** directory (**cd doc**) and run:

```
sphinx-quickstart
```

This will ask a handful of questions and generate the appropriate directories and files. For this example, use all the default answers.

Sphinx

`make html`

Run the command

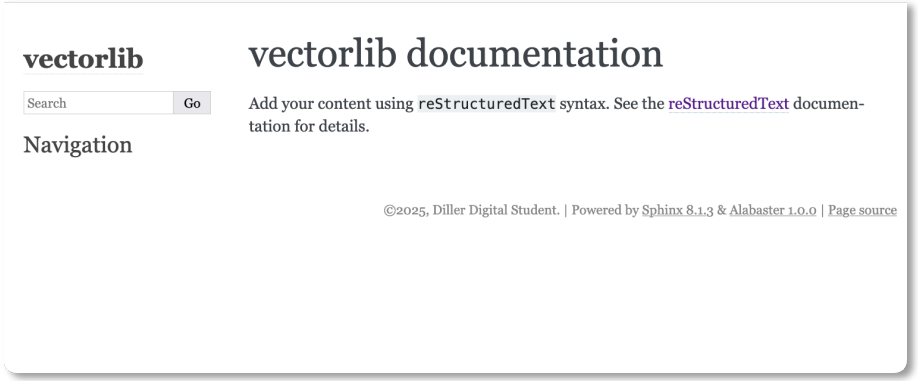
```
make html
```

to generate the (preliminary version) of the docs.

Point your browser to **`index.html`**

```
(Mac) open toplevel/doc/_build/html/index.html
(Windows)  toplevel/doc/_build/html/index.html
```

You should see:



`EDIT index.rst`

The quickstart command created **`index.rst`**.
This is the top level **`.rst`** file for the documentation.

For a less giddy front page, open the **`index.rst`** and change the line

"Welcome to ..."

to, for example,

"vectorlib: a 3D vector library".

Rerun **`make html`**

Sphinx

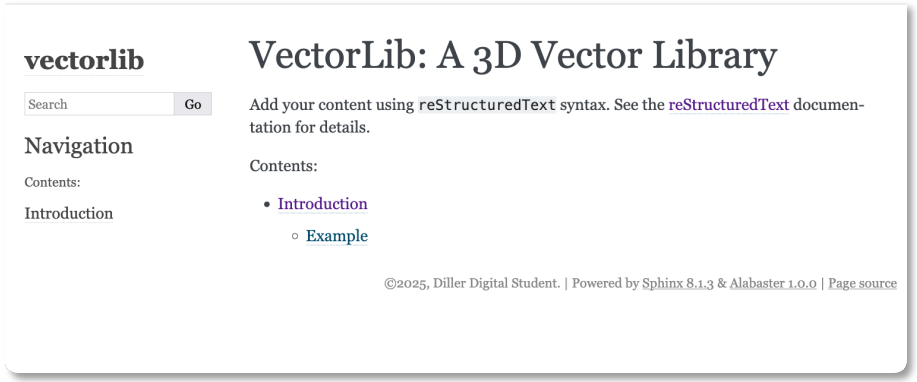
ADD AN INTRODUCTION

Now we can add an introduction to our documentation.

- Add the reStructuredText file **`intro.rst`** (shown to the right) to the doc directory.
- Add the line **`intro`** (the suffix **`.rst`** is implicit) to **`index.rst`**, indented under the **`toctree`** directive, as shown in this snippet:

```
.. toctree::
   :maxdepth: 2

   intro
```

Rerun **`make html`**, and check it out:



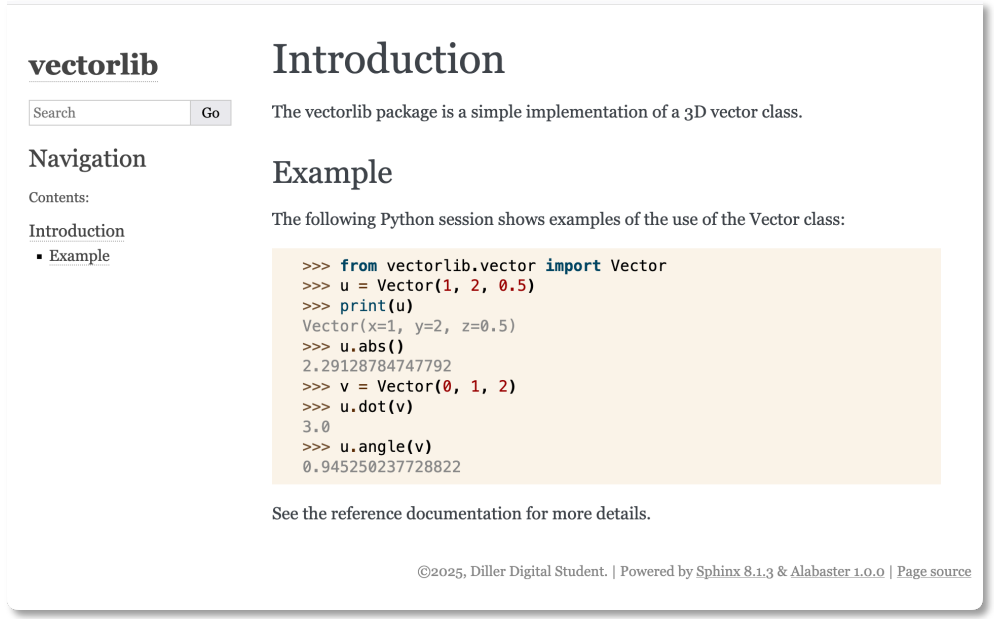
intro.rst

```
Introduction
============

The vectorlib package is a simple implementation
of a 3D vector class.

Example
-------

The following Python session shows examples of
the use of the Vector class::

    >>> from vectorlib.vector import Vector
    >>> u = Vector(1, 2, 0.5)
    >>> print(u)
    Vector(x=1, y=2, z=0.5)
    >>> u.abs()
    2.29128784747792
    >>> v = Vector(0, 1, 2)
    >>> u.dot(v)
    3.0
    >>> u.angle(v)
    0.945250237728822

See the reference documentation for more details.
```

Sphinx

ADD AN INTRODUCTION

Here's the HTML version of the introduction:



For comparison, **`intro.rst`** is still at the right. Sphinx automatically adds syntax highlighting to the sample Python code.

`intro.rst`

```
Introduction
============

The vectorlib package is a simple implementation
of a 3D vector class.

Example
-------

The following Python session shows examples of
the use of the Vector class::

    >>> from vectorlib.vector import Vector
    >>> u = Vector(1, 2, 0.5)
    >>> print(u)
    Vector(x=1, y=2, z=0.5)
    >>> u.abs()
    2.29128784747792
    >>> v = Vector(0, 1, 2)
    >>> u.dot(v)
    3.0
    >>> u.angle(v)
    0.945250237728822

See the reference documentation for more details.
```

Sphinx

API DOCUMENTATION

A quick way to add API documentation is with the Sphinx command **`sphinx-apidoc.`**

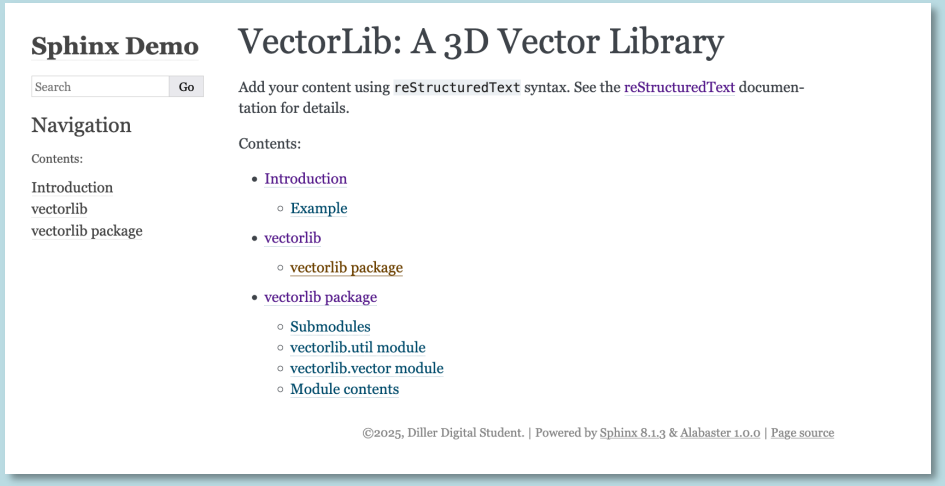
In the **`doc`** directory, run the command **`sphinx-apidoc -o api pathto/vectorlib`** (Modify **`pathto/vectorlib`** to be the correct path.)

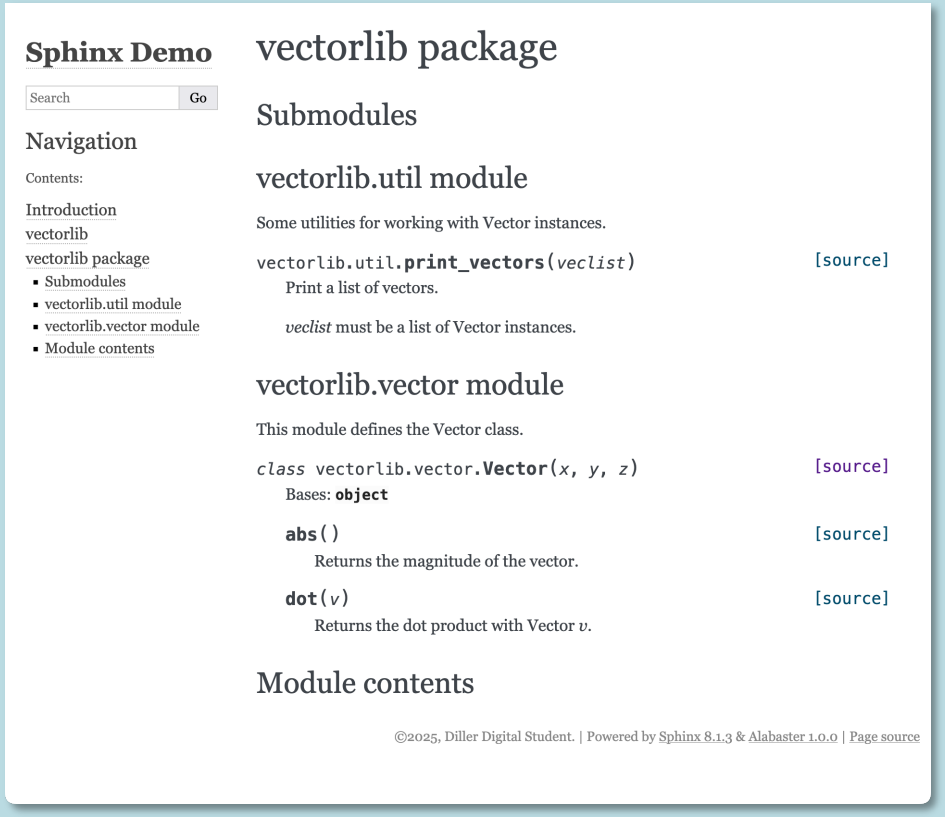
Then add the lines **`api/modules`** and **`api/vectorlib`** after **`intro`** in **`index.rst`**.

```
.. toctree::
   :maxdepth: 2

   intro
   api/modules
   api/vectorlib
```

Rerun **`make html.`**

Compare the HTML on the right to the source code. Note how the docstrings of the modules, functions and class methods are included in the HTML page.



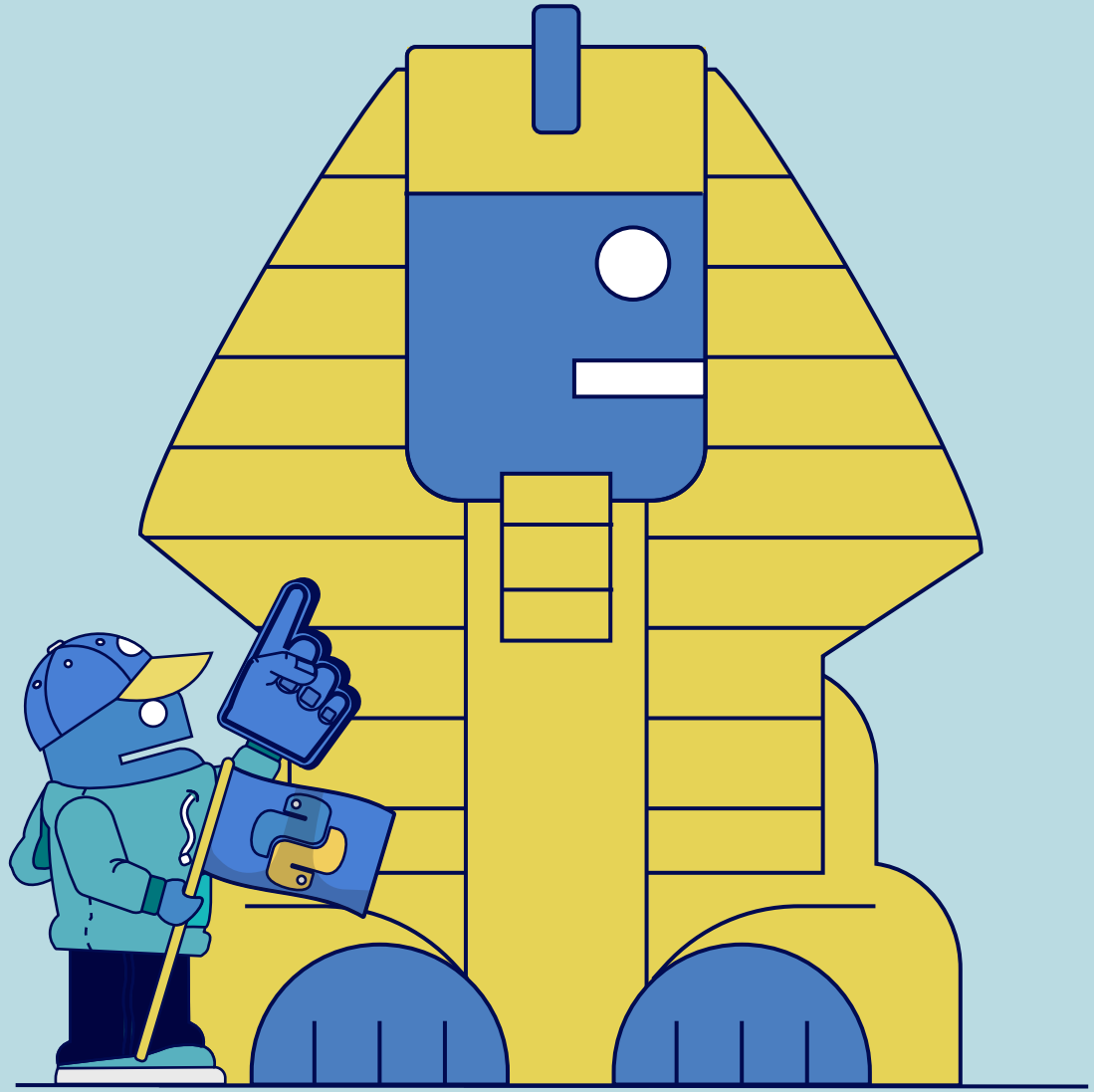


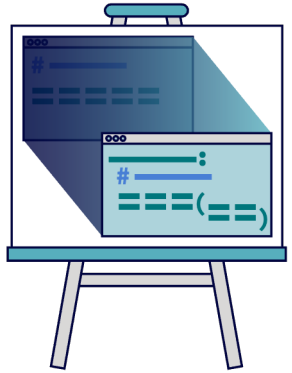
Sphinx

- For finer control, there are many options that can be configured in **`conf.py`**.
- HTML style may be customized by providing your own CSS files.
- API documentation may be fine-tuned by editing the files that were created by **`sphinx-apidoc`**, or by creating your own **`.rst`** files from scratch.
- Sphinx provides a large suite of reStructuredText directives for controlling how the documentation for the:
  - modules,
  - classes,
  - function,
  - etc.

  is generated.

See the **`autodoc`** documentation for complete details





Lecture 04

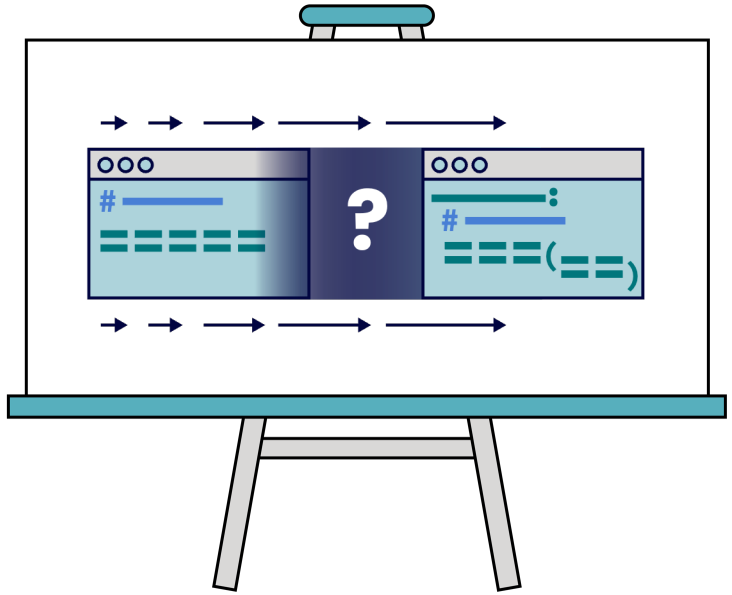
Refactoring Code

Software Engineering for Scientists and Engineers

Discussion

What does refactoring code mean to you?

Table of Contents



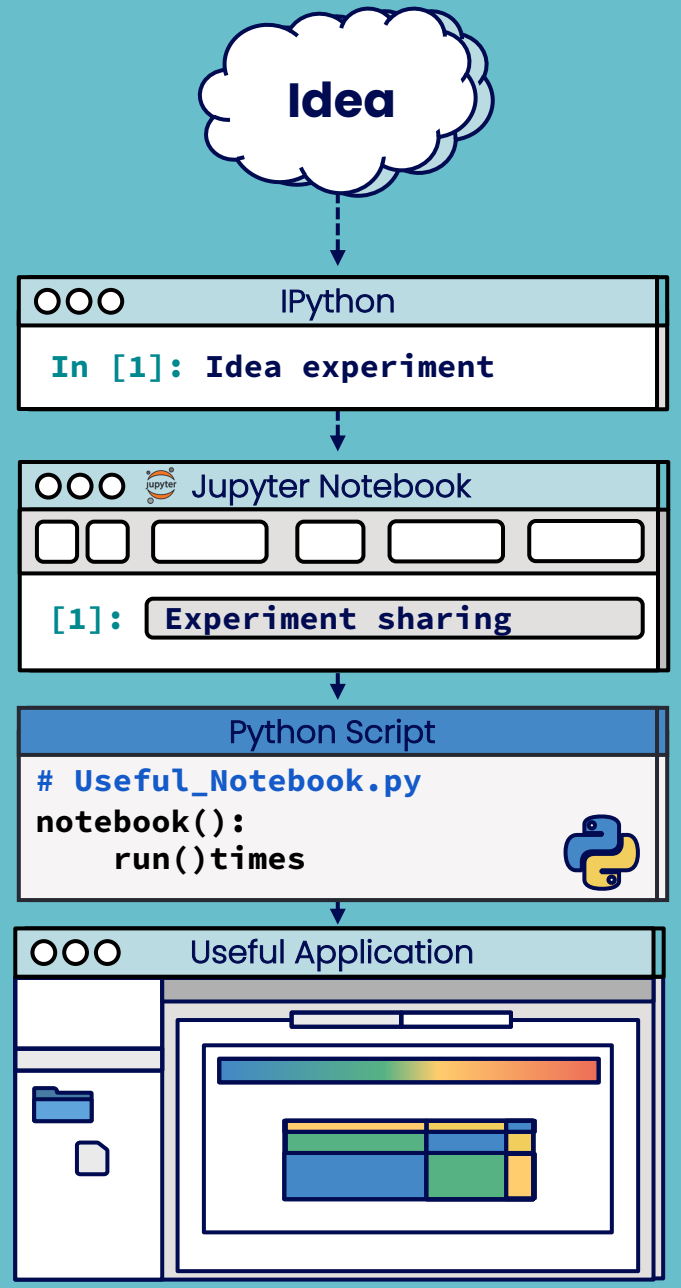
What is Refactoring?

Lecture 04
Refactoring

In the beginning ...

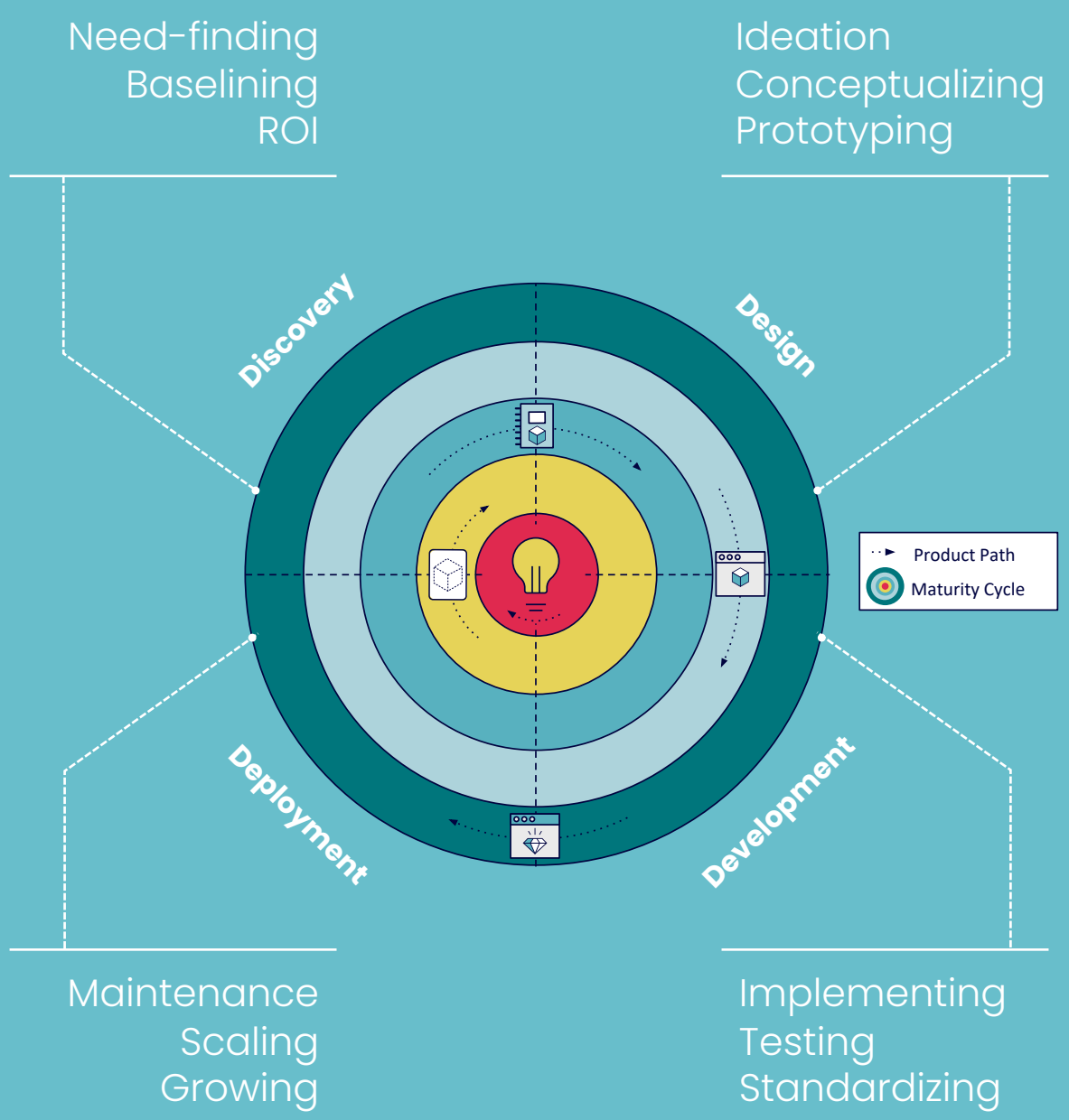
Someone has an **idea**.

- Some **ideas** are worth exploring; in Python these become code **experiments**, perhaps in IPython.
- Some experiments are worth sharing; in Python **Jupyter Notebooks** make sharing easy.
- Some **Jupyter Notebooks** are worth running again and again. If they do something useful for multiple people, convert them into **scripts** to make them runnable from the command line or on a schedule.
- Some **scripts** grow, become more useful, sprout a user-interface, and become **applications**.



- As code evolves from **idea** to **application**, it gets more complex.
- During this evolution, we learn more about the problem we are solving, the data driving it, and the ways we are analyzing and reporting it. Our understanding improves and is refined.
- More people are involved in the code:
  - as developers
  - as users
- As code evolution proceeds, we realize we have problems (old and new) directly related to the code:
  - readability
  - documentation
  - maintainability
  - ...

... it is time to refactor.



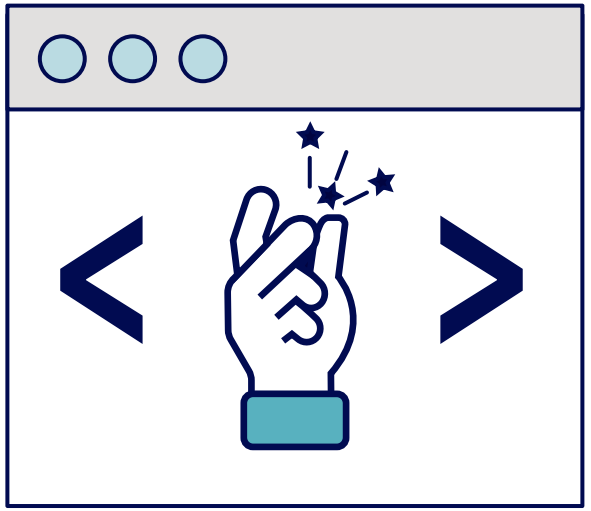
Refactoring Defined

Refactoring is the process of taking existing code and selectively modifying it to improve it for clarity and reduced complexity.

Ideally, we make it possible for someone other than the original author to maintain, extend, and reuse the code.

Use refactoring to make code:

- Easier to understand
- Easier to debug
- Easier to reuse
- Easier to modify
- Easier to test



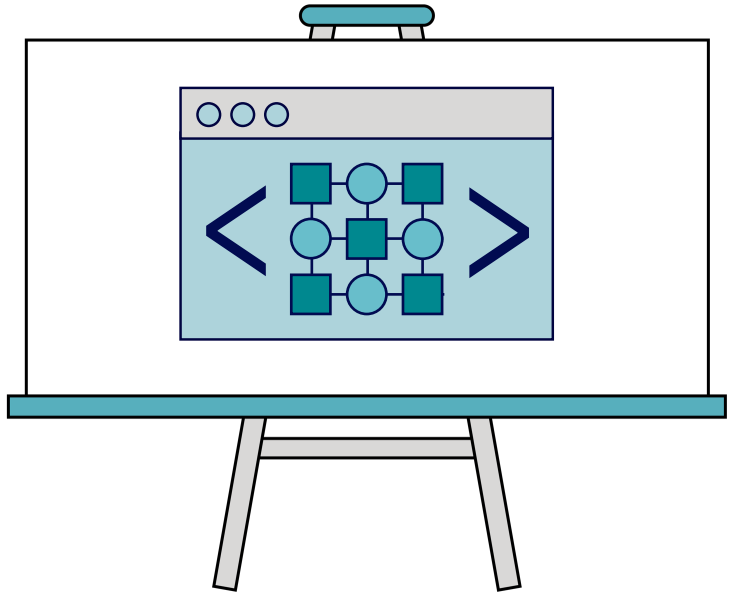
Before You Begin Refactoring

Scan the code you are refactoring and try to understand:

- **Existing Imports** – What modules are imported? Are these standard, installed third-party, or local modules? Do you understand the imported modules?
- **Existing Structure** – How is the code structured? Are the functions long and complex? Are they short? How are they organized? How well documented? Do they have meaningful names?
- **Existing Variables** – Do you understand the local variables? Do they have good names? Are they meaningful to you?

This pre-refactoring scan will help you identify the main areas to focus on and plan your approach.

Remember that the key task in refactoring is to **do no harm**. You want to improve readability and maintainability without breaking anything.



Basic Refactoring Patterns

Lecture 04
Refactoring

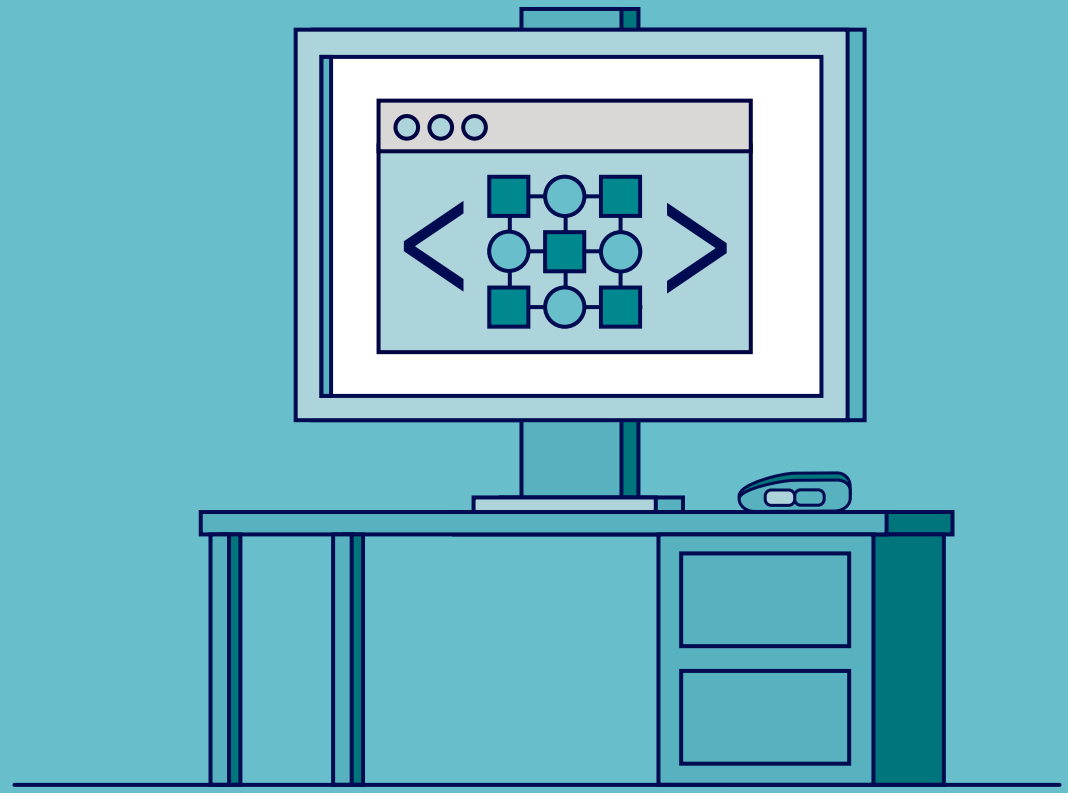
Basic Refactoring Patterns

In this section we will talk about **some basic refactoring patterns.**

These are patterns that are typical of new developer code, but generally work themselves out as the developer gains experience.

Some things to think about:

- Experience and good coding habits will mean these patterns eventually disappear.
- Doing this level of refactoring in a body of code that you are trying to understand is a good way to interact with the code and learn it.
- While debugging production code, look for these refactoring opportunities as part of general code maintenance.



Give Names to Constants

Find hard-coded constants that are used in your code. Replace them with named constants. They become:

- Easier to find and update
- Easier to understand when reading code
- Often useful to define at top of file or function, in a central location

BEFORE

```
percentiles = []
papers = sorted(papers)
percentile_size = 1.0 / 10
for i in range(10 + 1):
        rank = i * percentile_size
        value = get_value(papers, rank)
        percentiles.append((rank, value))

def get_value(papers, rank):
    index = rank * len(papers)
    try:
        return papers[int(index)][0]
    except IndexError:
        return papers[-1][0]
```

AFTER

```
N_PERCENTILES = 10

percentiles = []
papers = sorted(papers)
percentile_size = 1.0 / N_PERCENTILES
for i in range(N_PERCENTILES + 1):
        rank = i * percentile_size
        value = get_value(papers, rank)
        percentiles.append((rank, value))

def get_value(papers, rank):
    index = rank * len(papers)
    try:
        return papers[int(index)][0]
    except IndexError:
        return papers[-1][0]
```

Remove Environmental Dependencies

Find environmental dependencies (usernames, filenames, paths, network addresses, etc.) hard-coded into your code. Replace them with function arguments used to call specialist functions. They become:

- easier to find and update
- easier to understand when reading code

BEFORE

```
# load citations
citations = []
with open('cit-HepTh.txt', 'r',
          encoding='ascii') as f:
    # Skip header
    for _ in zip(range(4), f):
        pass
    ...

# make percentiles table
...
```

AFTER

```
def process_citations(filename):
    # load citations
    citations = []
    with open(filename, 'r',
              encoding='ascii') as f:
        # Skip header
        for _ in zip(range(4), f):
            pass
        ...
    # make percentiles table
    ...

process_citations('cit-HepTh.txt')
```

Convert Code Duplications into Functions

When writing code, it is easy to copy-paste from one section to another, modifying the copy as needed. This is generally a sign that:

- **Commonalities** in the code-block that was copied can be transformed into one or more functions
- **Differences** between the original and its copies could be handled by delegating to a single function

BEFORE

```
percentiles = []
papers = sorted(papers)
percentile_size = 1.0 / N_PERCENTILES
for i in range(N_PERCENTILES + 1):
    rank = i * percentile_size
    # get value at rank
    index = rank * len(papers)
    try:
        percentiles.append(
            (rank, papers[int(index)][0])
        )
    except IndexError:
        percentiles.append(
            (rank, papers[-1][0])
        )
```

AFTER

```
def get_value(papers, rank):
    index = rank * len(papers)
    try:
        return papers[int(index)][0]
    except IndexError:
        return papers[-1][0]

percentiles = []
papers = sorted(papers)
percentile_size = 1.0 / N_PERCENTILES
for i in range(N_PERCENTILES + 1):
        rank = i * percentile_size
        value = get_value(papers, rank)
        percentiles.append((rank, value))
```

Convert Coherent Blocks of Code into Functions

Look for blocks of code with a comment that tells what the block of code does. This is often a clue that this block should be converted into a specialist function. Ideally each function does a single, well-defined task.

- Long blocks of code can often be converted into a series of functions.

BEFORE

```
# load citations
citations = []
with open(filename) as f:
    ...

# build index
index = {}
for paper, cited_paper in citations:
    ...

# get citation counts
papers = [
    (len(citers), paper_id)
    for paper_id, citers
    in index.items()
]
...
```

AFTER

```
def load_citations(filename):
    citations = []
    with open(filename) as f:
        ...

def build_index(citations):
    index = {}
    for paper, cited_paper in citations:
        ...

def get_counts(index):
    papers = [
        (len(citers), paper_id)
        for paper_id, citers
        in index.items()
    ]
...
```

Create a `main()` Function

The top-level sequence of operations (temporal dependencies) in our program should be pulled out and placed in its own function; by convention, this is called **main()**.

- Each function called by **main()** does a single, focused task, returning what is needed for the next task

BEFORE

```
def process_citations(filename):
    # load citations
    citations = []
    with open(filename, 'r',
              encoding='ascii') as f:
        # Skip header
        for _ in zip(range(4), f):
            pass
        ...
    table = '\n'.join(rows)
    # display percentiles table
    print(table)

process_citations('cit-HepTh.txt')
```

AFTER

```
...

def make_table(percentiles):
    rows = ['{:>6.1%} {:>9d}'.format(
        rank, value
    ) for rank, value in percentiles]
    return '\n'.join(rows)

def main(filename):
    citations = load_citations(filename)
    index = build_index(citations)
    percentiles = get_percentiles(index)
    table = make_table(percentiles)
    print(table)

main('cit-HepTh.txt')
```

Separate Definitions from Invocations

Python programs should keep definitions (imports, constants, functions, classes) separate from invocations (assigning variables, calling functions, instantiating classes)

- Can re-use definitions by importing program as a module
- With **if __name__ == "__main__"** can execute invocations running from command line; can avoid invocations when importing as a module

BEFORE

```
def load_citations(filename):
    ...

...

def main(filename):
    citations = load_citations(filename)
    index = build_index(citations)
    percentiles = get_percentiles(index)
    table = make_table(percentiles)
    print(table)

main('cit-HepTh.txt')
```

AFTER

```
def load_citations(filename):
    ...

...

def main(filename):
    citations = load_citations(filename)
    index = build_index(citations)
    percentiles = get_percentiles(index)
    table = make_table(percentiles)
    print(table)

if __name__ == "__main__":
    main('cit-HepTh.txt')
```

Refactoring Processing Pipelines

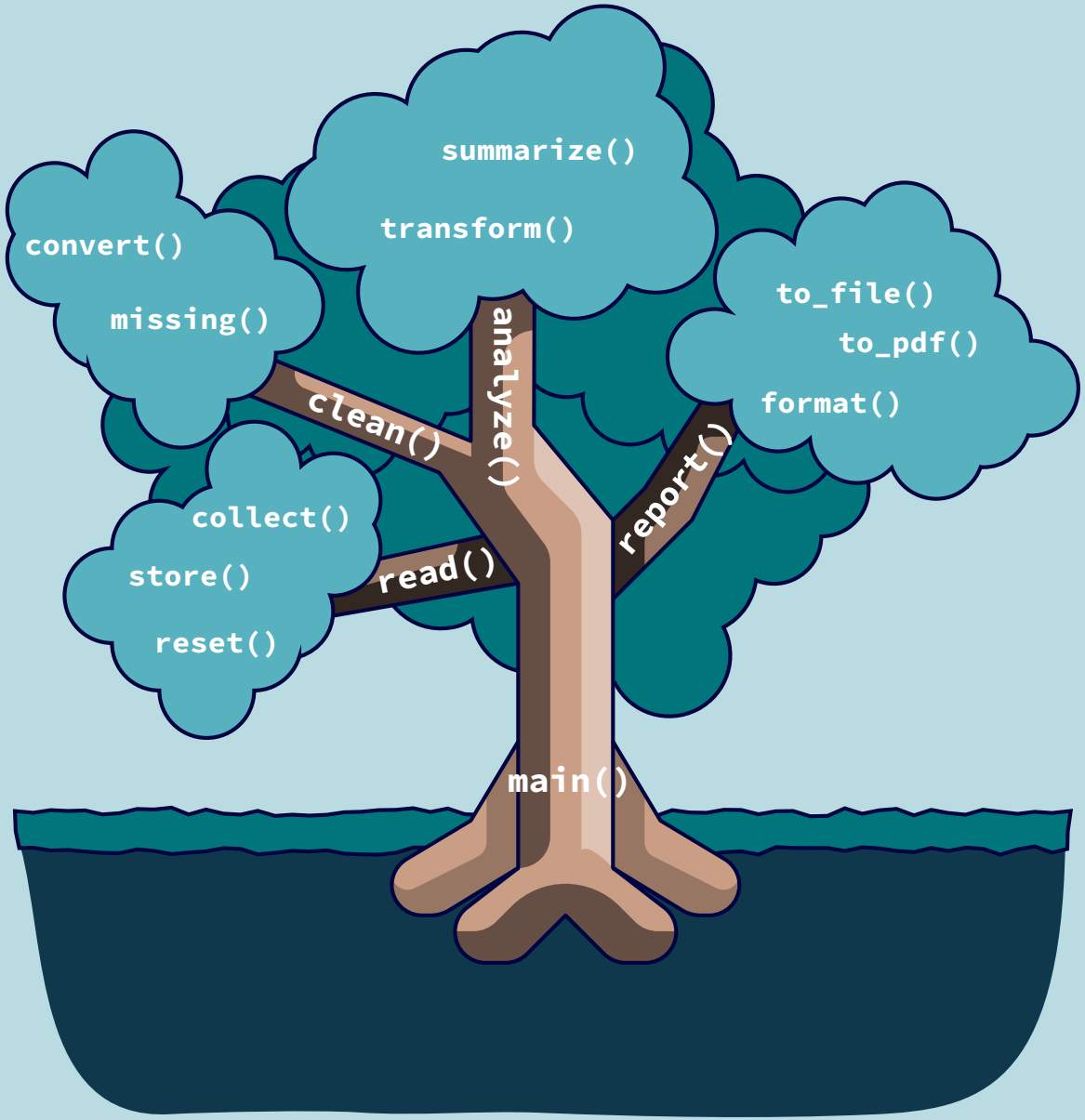
A lot of code is essentially a processing pipeline.

For example, processing scientific data read from a sensor might look something like:

1. Read instrument data.
2. Clean the data (check for missing values, convert sensor readings to meaningful values).
3. Analyze the data.
4. Report.

This code breaks into four natural blocks: **read()**, **clean()**, **analyze()**, and **report()**.
You should have at least these four functions and a **main()** function that calls them in sequence.

If your pipeline functions are complex, they should also be broken into smaller functions.



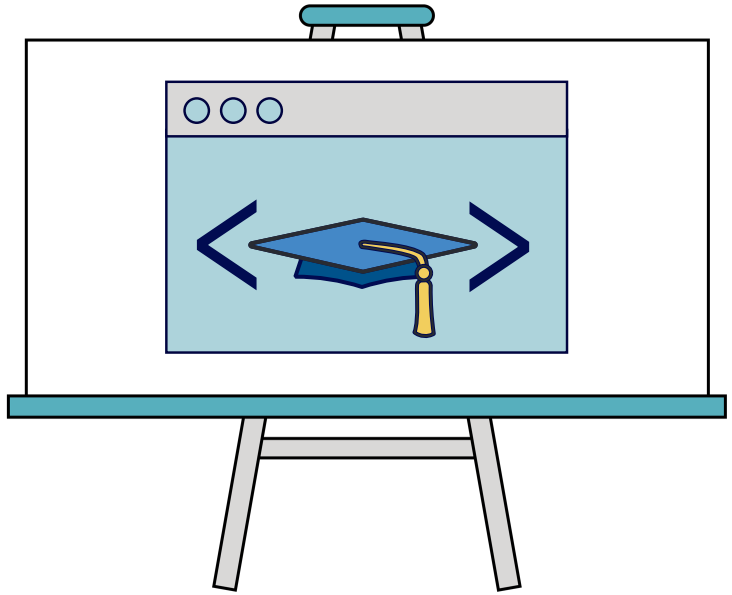
Give it a try! Basic Refactoring

```
masses_oz = [0.5, 0.6, 1.2, 3.4]

# convert to metric
masses_g = [mass_oz * 28.3495 for mass_oz in masses_oz]
masses_kg = [mass_oz * 28.3495 / 1000.0 for mass_oz in masses_oz]
```

1. Give names to constants.

2. Convert code duplications into functions.
   Hint: Both list comprehensions do the same basic operation on a list of values.

3. Create a Python module (.py file) where definitions are separate from invocations.
   Hint: Here the definitions are created in steps 1 and 2 above; the invocations are where you are setting the variables.



Advanced Refactoring Patterns

Lecture 04
Refactoring

Advanced Refactoring Patterns

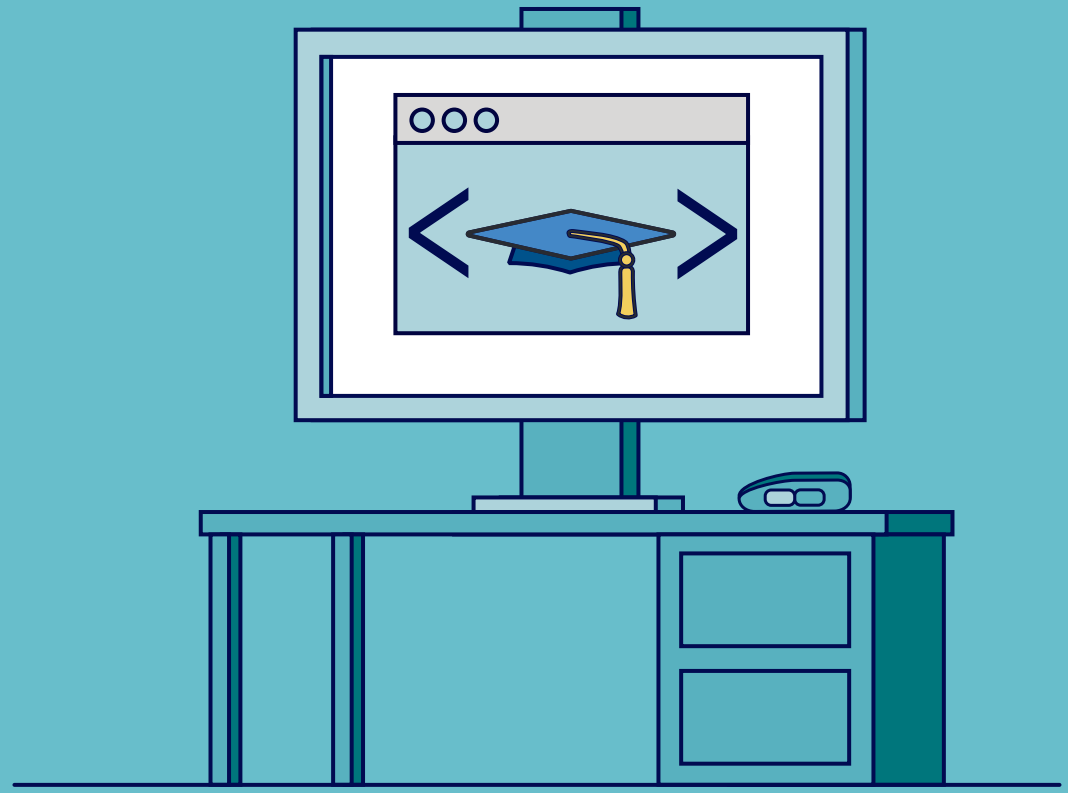
In this section we will talk about some more advanced refactoring patterns.

These are patterns that typically emerge as code evolves, more people get involved, and becomes more complex.

Some things to think about:

- As code evolves, these kinds of patterns will recur.
- These patterns are not necessarily a sign of a lack of coding experience. Instead, they are a sign that the problem being solved is getting better understood.

As this understanding progresses, these will be patterns that you will discover and address.



Flatten Nested Code

Code (if statements, loops, try/except clauses) that is nested more than two levels is hard to read. Flatten them by creating functions or reorganizing the code.

- Nested if statements can often be flattened by combining conditions if a single if.

BEFORE

```
# simple example 1 (combining conditions)

if a < 1:
    if b > 100:
        print("Values out of range.")
```

AFTER

```
# simple example 1 (combining conditions)

if a < 1 and b > 100:
    print("Values out of range.")

# simple example 2 (wrap complex conditions in
# a function)

def allowed_range(lower, upper):
    if lower < 1 and upper > 100:
        return False
    return True

if not allowed_range(a, b):
    print("Values out of range.")
```

Replace Loop Calculations with NumPy

Examine loops used for calculations. It may be possible to eliminate these loops by using NumPy arrays for the calculations.

- Not only will the code be more readable, it will be faster.

BEFORE

```
# zero out values <= 2, then add vectors

a = [1, 2, 3]
b = [4, 5, 6]

c = []
for va, vb in zip(a, b):
    if va <= 2:
        va = 0
    if vb <= 2:
        vb = 0
    c.append(va + vb)
```

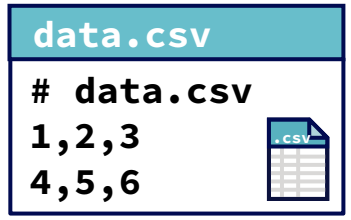
AFTER

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

a_gt2 = a * (a > 2)
b_gt2 = b * (b > 2)

c = a_gt2 + b_gt2
```

Use Established Libraries in Place of Custom Code

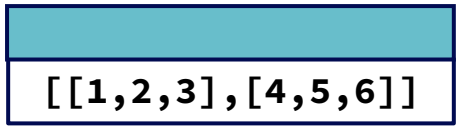
Get to know the Python standard library and the standard scientific libraries.

Instead of re-inventing the wheel, find established, well-vetted code to solve standard problems wherever possible.



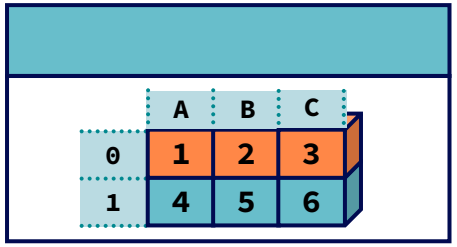
BEFORE

```
filename = 'data.csv'
data = []

with open(filename, encoding='utf-8') as fp:
    for line in fp:
        line = line.strip()
        if line.startswith('#'):
            continue
        d = [int(v) for v in line.split(',')]
        data.append(d)
```



AFTER

```
import pandas as pd

filename = 'data.csv'
data = pd.read_csv(filename,
                   comment='#',
                   names=list('ABC'))
```



Build Classes from Related Functions and Data

Model the important entities from the problem domain you are working in with Python classes. This will enable you to think about your domain instead of generic Python types. Both readability and maintainability will be improved.

BEFORE

```
def kinetic_energy(m, v):
    ...

def momentum(m, v):
    ...

masses = [1.2, 2.4, 2.3, 0.9]
velocities = [0.9, 0.7, 1.1, 0.1]

for m, v in zip(masses, velocities):
    print(kinetic_energy(m, v),
          momentum(m, v))
```

AFTER

```
class Particle:

    def __init__(self, m, v):
        self.m = m
        self.v = v

    def kinetic_energy(self):
        ...

    def momentum(self):
        ...

particles = [Particle(1.2, 0.9),
    Particle(2.4, 0.7), Particle(2.3, 1.1),
    Particle(0.9, 0.1)]

for p in particles:
    print(p.kinetic_energy(), p.momentum())
```

Refactoring and APIs

If you are refactoring code used by other people, be cautious.

Refactoring may improve your own code and its maintainability, **but break other people's code that relies on your code.**

Watch out for:

- Changing names of constants, functions, or classes.
- Changing function signatures.
  - Changing the number or order of arguments.
  - Changing the names of arguments.

Give it a try! Advanced Refactoring

```
OZ_TO_G = 28.3495
G_TO_KG = 1 / 1000

def convert(values, factor):
    return [v * factor for v in values]

if __name__ == '__main__':
    masses_oz = [0.5, 0.6, 1.2, 3.4]
    masses_g = convert(masses_oz, OZ_TO_G)
    masses_kg = convert(masses_g, G_TO_KG)
```

1. **Approach #1 :** Replace Loop Calculations with NumPy.

2. **Approach #2 :** Build Classes from Related Functions and Data. Hint: create a Mass class.

Basic Refactoring Patterns Summary

Refactoring Task	Explanation
Give Names to Constants	Look for constants (int, float, string) that are used in the code (even if only one place) and give them meaningful names. By convention, constant names are often all in caps.
Remove Environmental Dependencies	Find file names, directory paths, network addresses, database names, and other values that tie your code to a specific environment. In the places that use these, create functions that take the values as arguments so that they can be easily updated.
Convert Code Duplications into Functions	Search for places where code has been duplicated (and possibly modified). Commonalities should become functions; differences should be placed in specialist functions.
Convert Coherent Blocks of Code into Functions	Break your program into functions, with each function performing one well-defined task.
Create a main() Function	The first function you call should look like a recipe for solving your problem. It should mostly call other, simpler functions that handle one task at a time.
Separate Definitions from Invocations	Make sure your constants, functions, and classes (your definitions) can be used as an imported module. Any top-level invocations of those functions should not run during import.

Advanced Refactoring Patterns Summary

Refactoring Task	Explanation
Flatten Nested Code	Look for nested code and try to flatten it. This may involve combining boolean conditions, creating functions, or altering code flow.
Replace Loop Calculations with NumPy	In almost every case, running data calculations with loops in Python is a mistake. Use NumPy (or packages built on NumPy) to take advantage of its vectorized operations and masking capabilities.
Use Standard Libraries in Place of Custom Code	Use the functions provided in the Python standard libraries and in the Python scientific ecosystem libraries as much as possible. There is generally no reason to write your own generic functions for tasks like min(), max(), reading CSV files, sorting, etc.
Build Classes from Related Functions and Data	Since Python is an object-oriented language, you can define your own types. This gives you the ability to group functions and data together in ways that will make it easier to use, maintain, and extend your code.
Don't Break APIs	If your code is in use elsewhere, be cautious during refactoring. Changing constants, functions, or classes may break someone else's code.

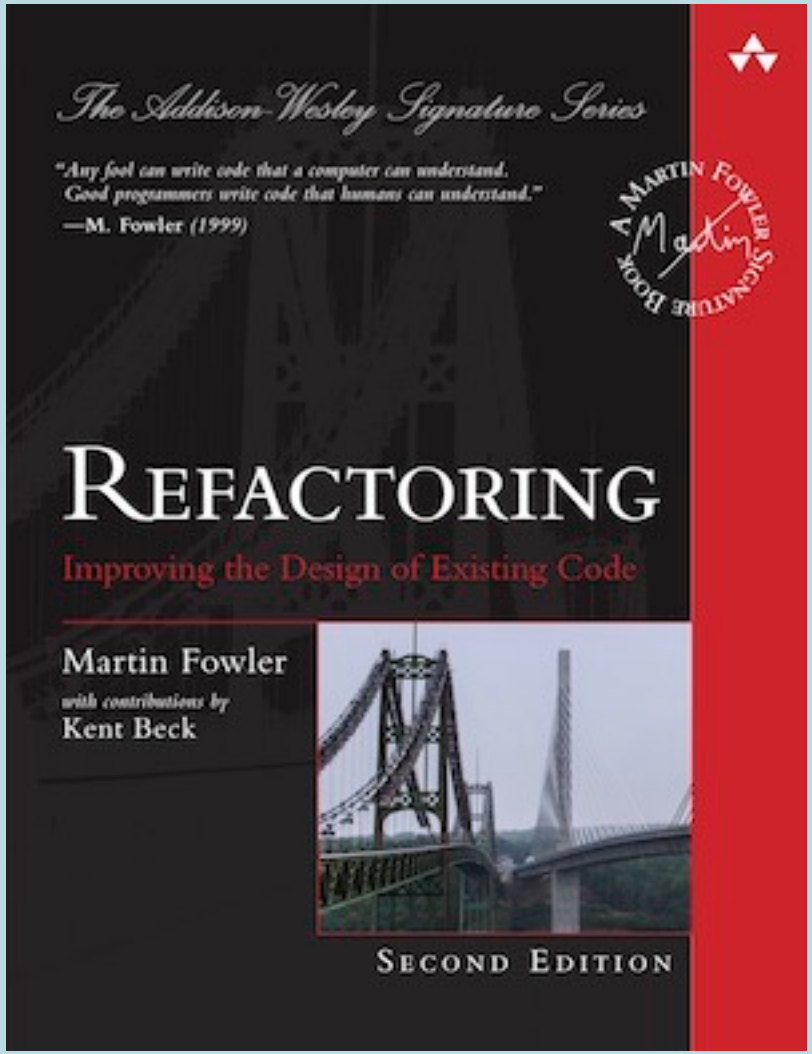
Refactoring Recommendations

Use refactoring to make code:

- Easier to understand
- Easier to debug
- Easier to reuse
- Easier to modify
- Easier to test

Cautions:

- DRY is not necessarily an improvement
- Premature abstraction can make code harder to modify
- Separation into modules can make code harder to understand



https://martinfowler.com/books/refactoring.html

Exercise: Refactoring

- Refactor Inventory

Lecture 05

Monitoring Execution

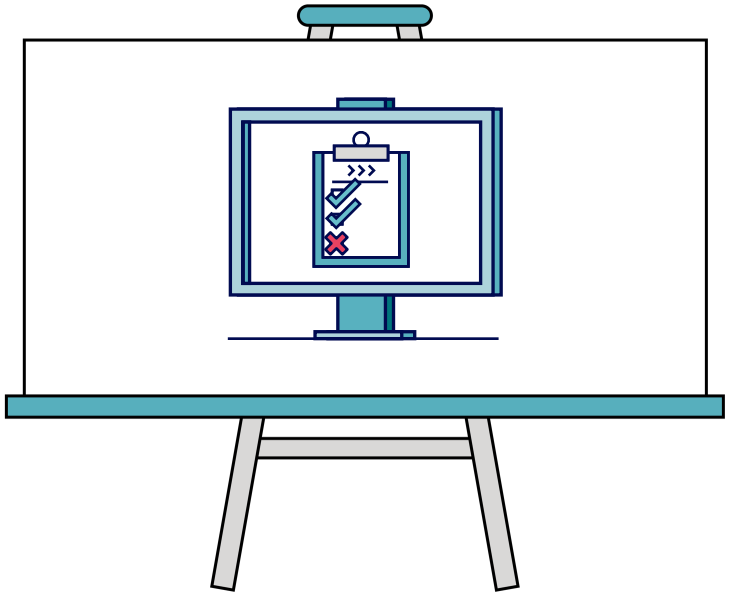
Software Engineering for Scientists and Engineers

Discussion

How do you monitor code execution?

```
%run live_coding/lecture05.py
```

Table of Contents



Basic Logging

Lecture 05
Monitoring Execution

The Print Function

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."

- Brian Kernighan, 1979

- Print statements are your first line of defense – they are fast, easy to use, and require little planning.
- However, they require you to be physically present and paying attention while the program is running.
- The easiest way to make your debugging code more maintainable is by swapping print calls for logging calls.

The Logging Module

- Logging allows you to decide how, when, and where runtime information is persisted, including information from modules you didn't write.
- Logging messages are filtered by a severity level, which represents how important it is that you know something happened.
- By default, Python logs message with **`level >= 30 (WARNING, ERROR, & CRITICAL)`**.

Level	Name	Description
`50`	`CRITICAL`	Program is broken.
`40`	`ERROR`	One thing broke, but program can continue running.
`30`	`WARNING`	Possibly dangerous, but nothing is broken.
`20`	`INFO`	Program is running; here is some information.
`10`	`DEBUG`	Details about configuration and data.
`0`	`NOTSET`	Level not set; inherit level from parent object.

Logging Basic Configuration to Console

BASIC CONFIGURATION

```
>>> import logging
>>> logging.basicConfig()

# What logger was set up and at what level?
>>> logging.getLogger()
<RootLogger root (WARNING)>
```

LOGGING LEVEL METHODS

```
>>> def test_logging():
...     """Function to test logging levels."""
...     print("This always prints.")
...     logging.critical("Critical Message")
...     logging.error("Error Message")
...     logging.warning("Warning Message")
...     logging.info("Info Message")
...     logging.debug("Debug Message")
```

LOGGING FILTERS

```
>>> test_logging()
This always prints.
CRITICAL:root:Critical Message
ERROR:root:Error Message
WARNING:root:Warning Message
```

LOGGING BASIC FORMAT

```
>>> logging.BASIC_FORMAT
'%(levelname)s:%(name)s:%(message)s'
```

CRITICAL:root:Critical Message

levelname : name : message

Changing Basic Logging

CHANGE BASIC LOGGING LEVEL

```
# If already configured, calling basicConfig()
# finds that the root logger is already set up
# and returns without changing anything.
>>> logging.basicConfig(level=logging.INFO)
>>> logging.getLogger()
<RootLogger root (WARNING)>

# Can force a new configuration.
>>> logging.basicConfig(level=logging.INFO,
...                     force=True)
>>> logging.getLogger()
<RootLogger root (INFO)>

# Can also change level through logger object.
>>> logger = logging.getLogger()
>>> logger.setLevel(logging.CRITICAL)
>>> logger
<RootLogger root (CRITICAL)>

>>> test_logging()
This always prints.
CRITICAL:root:Critical Message
```

CHANGE BASIC LOGGING FORMAT

```
# Get handlers from logger object.
>>> handlers = logger.handlers
>>> handlers
[<StreamHandler <stderr> (NOTSET)>]

# Only a stream handler, grab it.
>>> handler = handlers[0]

# Change the format.
>>> fmt = '%(asctime)s:%(funcName)s()'
>>> formatter = logging.Formatter(fmt)
>>> handler.setFormatter(formatter)
>>> test_logging()
This always prints.
2023-03-16 14:19:06,558:test_logging()
```

RESET BASIC LOGGING

```
# Reset basic logging to defaults.
>>> logging.basicConfig(force=True)
```

Logging Formats

Some of the most useful basic logging formatter attributes. More can be found at *https://docs.python.org/3.8/library/logging.html#logrecord-attributes*.

Attribute	Format	Description
`message`	**`%(message)s`**	Logged message.
`levelname`	**`%(levelname)s`**	Text logging level name for the message; e.g., **`'DEBUG'`**, **`'INFO'`**, **`'WARNING'`**, **`'ERROR'`**, or **`'CRITICAL'`**.
`asctime`	**`%(asctime)s`**	Human-readable logging time; e.g., **`"2023-03-16 14:20:22,009"`**
`funcName`	**`%(funcName)s`**	Name of function in which logging call occurred.
`lineno`	**`%(lineno)d`**	Line number in source where the logging occurred.
`name`	**`%(name)s`**	Name of the logger used for logging the message.

Examples:

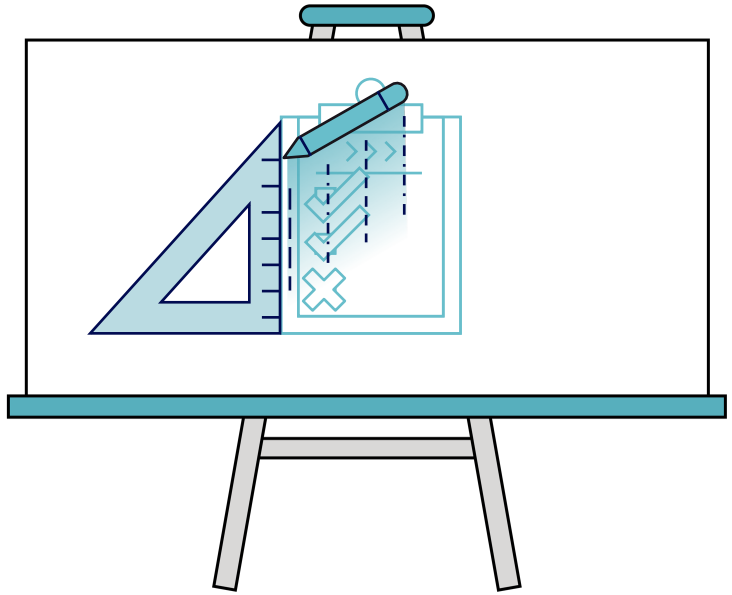
```
%(levelname)s:%(name)s:%(message)s
%(asctime)s:%(levelname).4s:%(message)s
```

Give it a try! Basic Logging

Command Prompt

```
def div(x, y):
    print(f'div(x={x}, y={y})')
    try:
        r = x / y
    except ZeroDivisionError:
        print("Cannot divide by zero.")
        raise
    return r
```

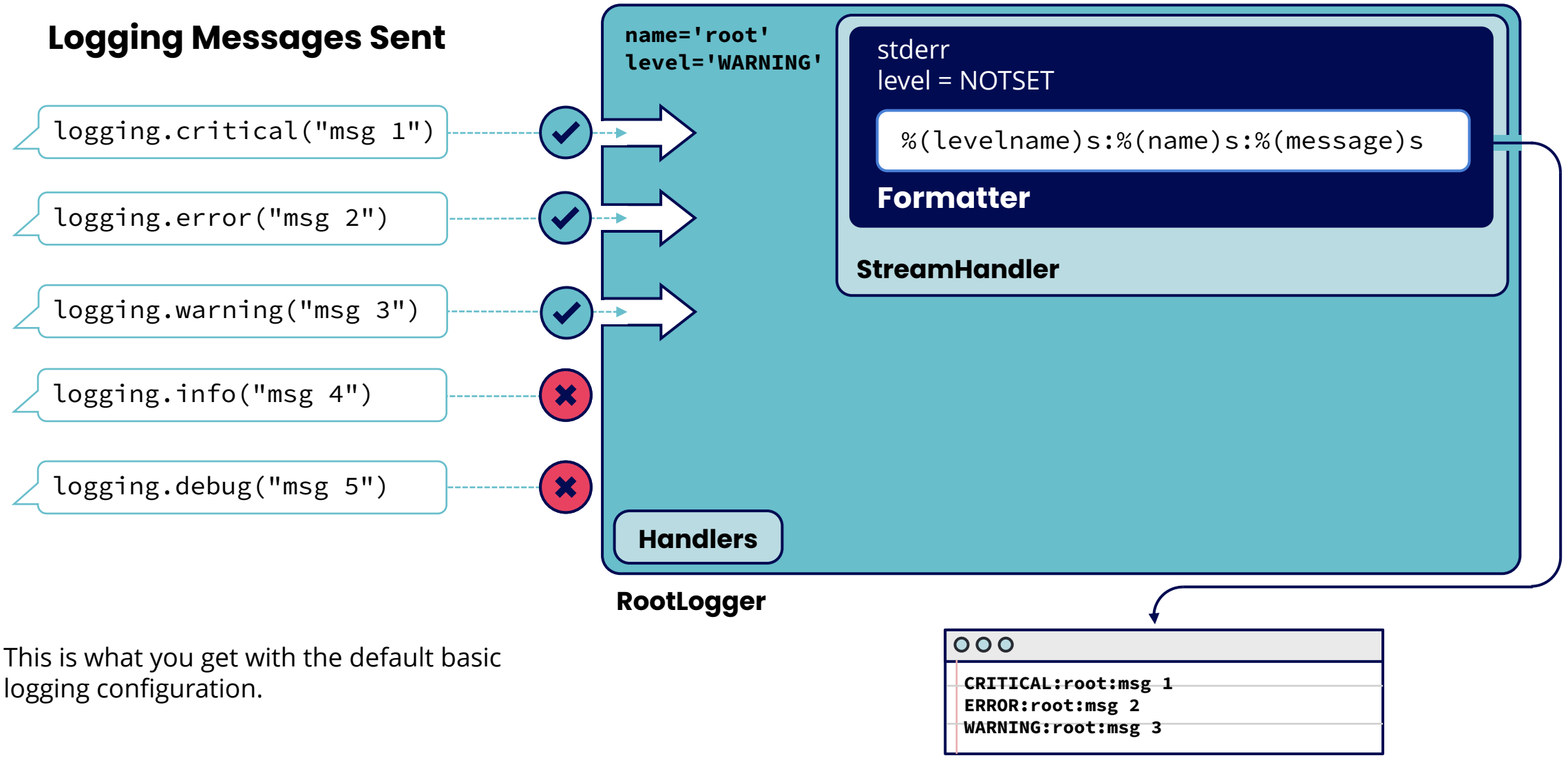
1. Run **`div(2, 3)`** and **`div(2, 0)`**. Observe the output.
2. Import logging and set up the default basic logging configuration.
   Hint: if you set up the logger already, use **`basicConfig()`** with the force keyword.
3. Convert the print functions into logging functions. What level should be used for each one?
4. Run **`div(2, 3)`** and **`div(2, 0)`** again. Observe the output.
5. Change the logging level to **`DEBUG`**, rerun, and observe.



Logging Architecture

Lecture 05
Monitoring Execution

Basic Logging Object Architecture



This is what you get with the default basic logging configuration.

Logging Handlers

Loggers use handlers to format and direct output to the correct destination. Python provides a number of standard handles, including:

Handler	Module	Description
`StreamHandler`	`logging.StreamHandler`	Displays logged messages on console in stderr.
`FileHandler`	`logging.FileHandler`	Sends logged messages to file on disk.
`RotatingFileHandler`	`logging.handlers.RotatingFileHandler`	Sends logged messages to file on disk. Maximum filesize and count can be set to limit logging volume.
`TimedRotatingFileHandler`	`logging.handlers.TimedRotatingFileHandler`	Similar to RotatingFileHandler, but log rotation controlled by time, not size.

- Can choose handler for logger.
- A single logger can have **multiple** handlers, each with a different level and format.

Add Additional Handler to Basic Configuration

BASIC CONFIGURATION (FILEHANDLER)

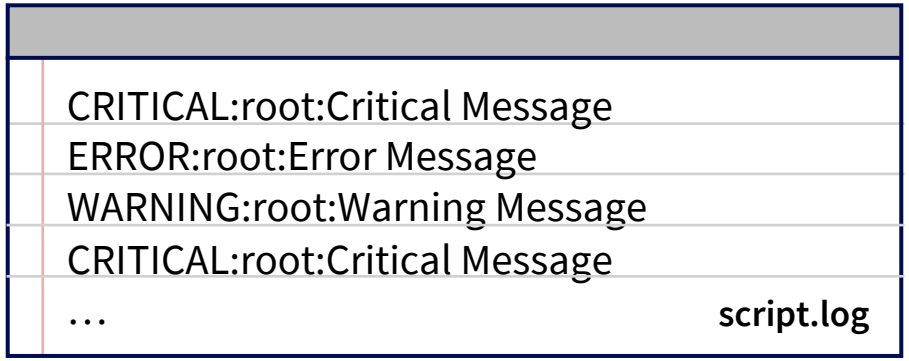
```
# log to file (override previous basic config)
>>> logging.basicConfig(filename='script.log',
...                     force=True)
>>> test_logging()
This always prints.
```

ADD STREAMHANDLER

```
# Grab the logger
>>> logger = logging.getLogger()

# Add new handler
>>> logger.addHandler(logging.StreamHandler())
```

TEST HANDLERS

```
>>> test_logging()
This always prints.
Critical Message
Error Message
Warning Message
```



Handlers Can Have Different Levels and Formats

UPDATE HANDLER LEVEL

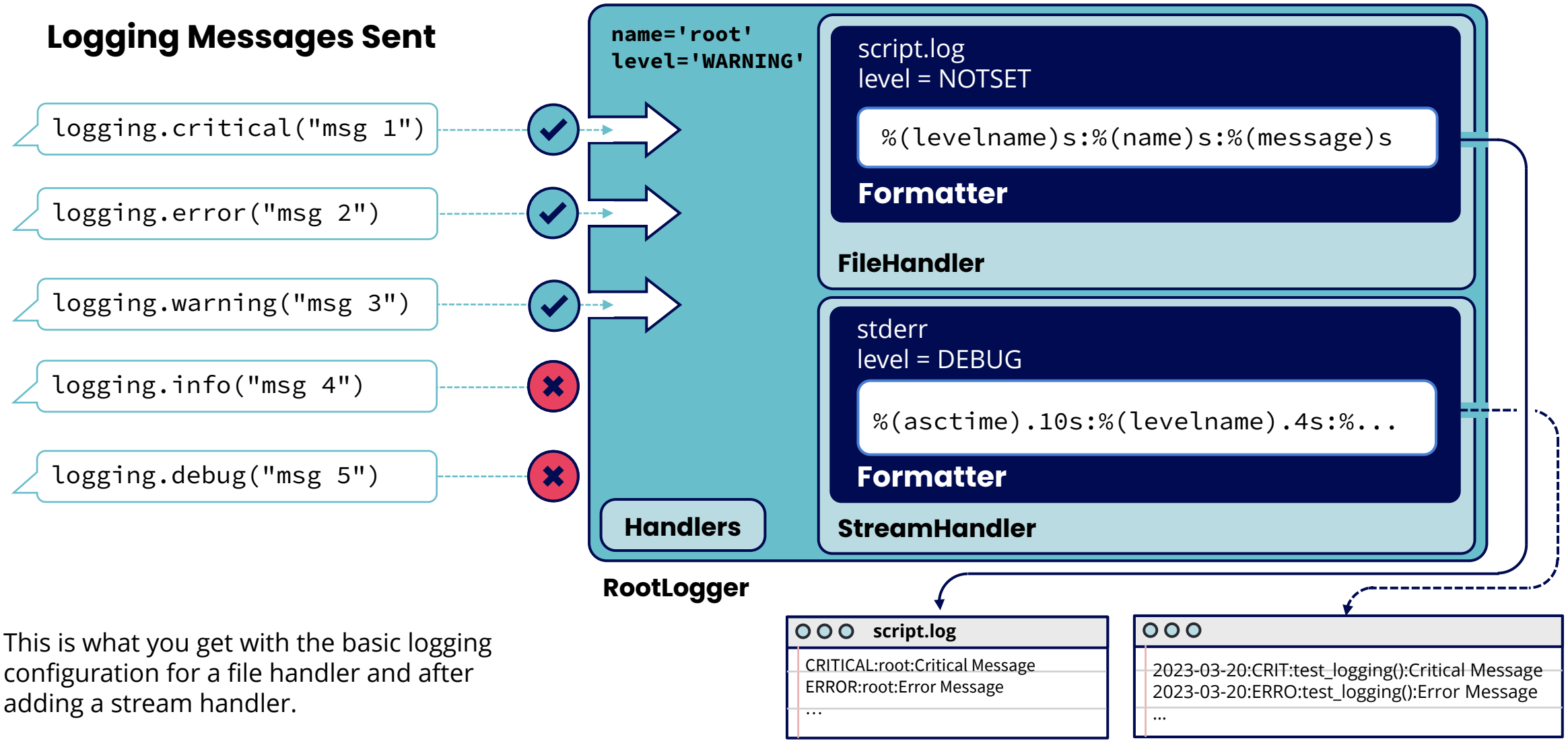
```
>>> logger.handlers
[<FileHandler …\script.log (NOTSET)>,
 <StreamHandler <stderr> (NOTSET)>]>>>
>>> sh = logger.handlers[1]
>>> sh.setLevel(logging.CRITICAL)
>>> test_logging()
This always prints.
Critical Message

# StreamHandler() default format is minimal
# might want to set a better format
>>> fmt = '%(asctime).10s:%(levelname).4s:'
>>> fmt += '%(funcName)s():%(message)s'
>>> formatter = logging.Formatter(fmt)
>>> sh.setFormatter(formatter)
>>> test_logging()
This always prints.
2023-03-20:CRIT:test_logging():Critical Message

# script.log retains original format
```

HANDLER LEVEL < LOGGER LEVEL

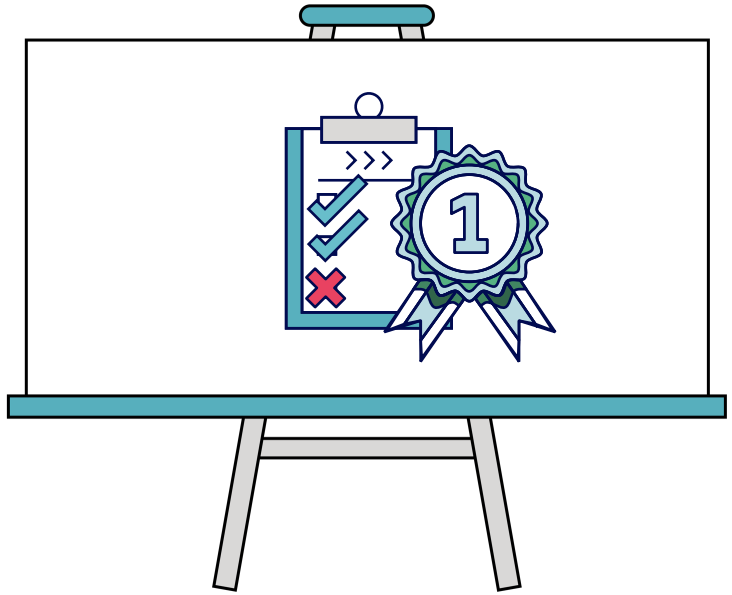
```
# logger level is WARNING (30)
>>> logger.level
30
>>> logger.handlers
[<FileHandler …\script.log (NOTSET)>,
 <StreamHandler <stderr> (CRITICAL)>]>>>
# set StreamHandler level to DEBUG (10)
>>> sh.setLevel(logging.DEBUG)
>>> test_logging()
This always prints.
2023-03-20:CRIT:test_logging():Critical Message
2023-03-20:ERRO:test_logging():Error Message
2023-03-20:WARN:test_logging():Warning Message
```

Note: In this case, the logger filters out the INFO and DEBUG level messages before they ever reach the handler!

Multiple Handlers Logging Object Architecture



This is what you get with the basic logging configuration for a file handler and after adding a stream handler.



Logging Best Practices

Lecture 05
Monitoring Execution

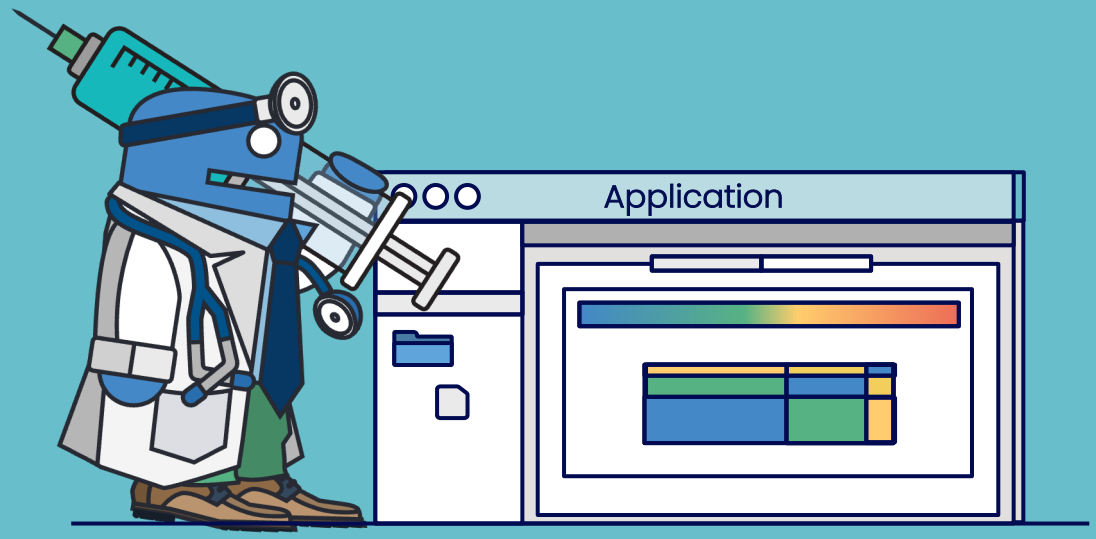
Application Logging

- **Simple Applications:** If you have a simple application (generally everything resides in a single file), use the **`basicConfig()`** as described in the previous slides.

- **Complex Applications:** If your application spans several files (modules), your logging needs may be more complex as well. Some basic best practices:

  - Create a separate logger in each module. Allows you to change logging level by module to reduce noise.

  - Always log something. Logging allows users to know that the application is at least running.



Multi-Module Application Logging Example I

MAIN FILE

In your main file, set up logging for your entire application.

Handle all logging configuration here.

(One possible exception: it is often useful to have a log_setup.py module that your main application calls once to set things up).

```
import logging
from logging.handlers import RotatingFileHandler

# set up root logger
logger = logging.getLogger()
formatter =
logging.Formatter(logging.BASIC_FORMAT)

# log to console and to size-limited log files
console_handler = logging.StreamHandler()
console_handler.setFormatter(formatter)
logger.addHandler(console_handler)

file_handler = RotatingFileHandler("app.log",
                    maxBytes=10e6, backupCount=3)
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

# set the logging level for entire logging system
logger.setLevel(logging.INFO)
```

Multi-Module Application Logging Example II

IMPORTED MODULES

In each of your modules (the files you intend to import), set up a logger that is accessible throughout the module.

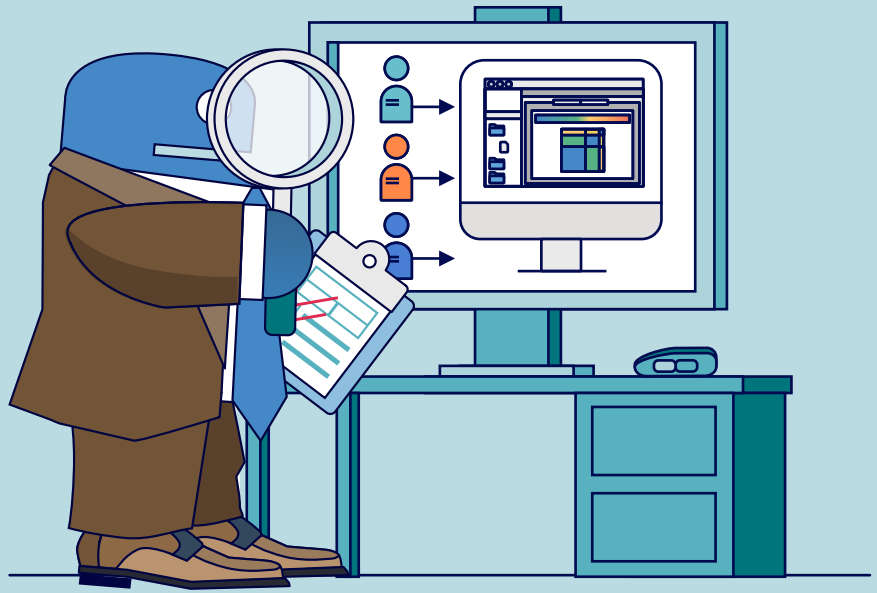
- Do not try to minimize logging.
  Put logging calls in wherever needed or wanted (at all levels).
- Leave this logger **unconfigured** here.
  It will automatically inherit the **root** logger's configuration and propagate its messages there.
- If special configuration is needed, the user who imports the module can grab this module's logger with `getLogger()` in the main program and configure it there.

```
from logging import getLogger

# set up logger for this module
# this will be global to this module
logger = getLogger(__name__)

# throughout code, set up logging
# messages as needed at the appropriate
# level
logger.debug("debug message")
logger.error("error message")
```

Logging for Auditing

The Python logging system is not just for debugging. It can also be used to provide auditing information for an application:

- **Users:** Leave an audit trail of when users logged in and logged out of an application.
  - If your application allows users to change configuration settings and/or data, it can be useful to record who changed what and when.
- **Resources:** If your application accesses external network, database, or other resources, leave an audit trail of connections to those resources (both successful and failed connections).



Logging and the Command Line

When running logging from the command line, it is often useful to set up your command line arguments parser to allow you to turn up or turn down the amount of logging.

ARGPARSE

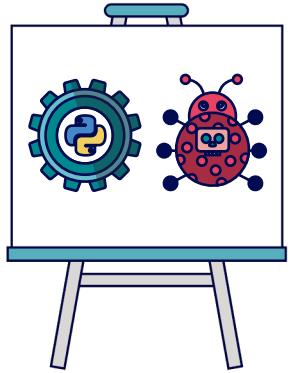
```
parser.add_argument('-v', '--verbose',
                    action='count', default=0)
parser.add_argument('-q', '--quiet',
                    action='count', default=0)

logging_level = logging.WARN
logging_level += 10*args.quiet
logging_level -= 10*args.verbose

# script -vv -> DEBUG
# script -v -> INFO
# script -> WARNING
# script -q -> ERROR
# script -qq -> CRITICAL
# script -qqq -> no logging at all
```

CLICK

```
@click.group(context_settings=CONTEXT_SETTINGS,
             invoke_without_command=True)
@click.option("-v", "--verbose", count=True,
              help="Increase logging level.")
@click.option("-q", "--quiet", count=True,
              help="Decrease logging level.")
def cli(verbose, quiet):
    logging_level = logging.WARN
    logging_level += 10 * quiet
    logging_level -= 10 * verbose
    setup_logger(logging_level)
```

Exercise: Logging

- Logging



Lecture 06

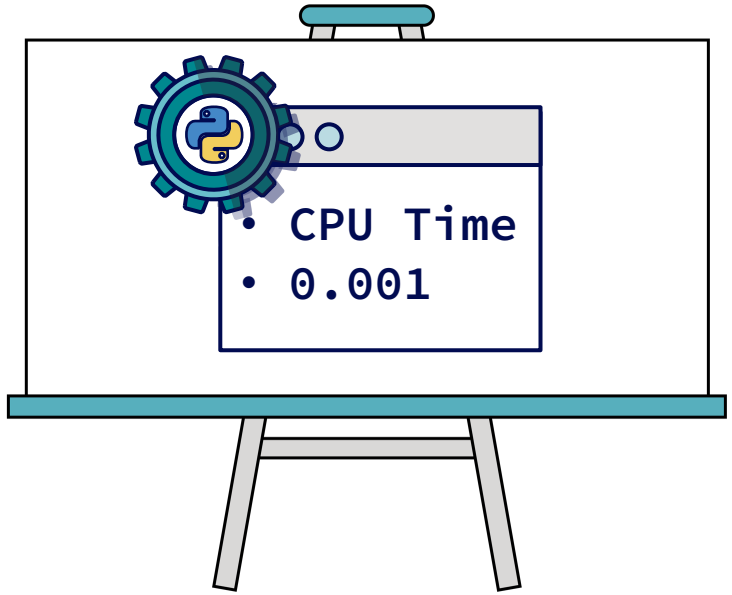
Profiling & Debugging

Software Engineering for Scientists and Engineers

Discussion

When should you think about code performance?

Table of Contents



Profiling

Lecture 06
Profiling & Debugging

Profiling

"Premature optimization is the root of all evil."

- Donald Knuth, 1974

There is a **tradeoff** programmers make between:

- Code that is extensible and debuggable
- Code that is optimized for CPU and/or memory usage

Therefore, one should:

- Limit the scope of optimization to operations that **dominate the time to completion**.
- Use a **profiler** to find the locations of these operations.

cProfile

cProfile (and its pure python version, **profile**) are profiling tools in the Python standard library.

WORKFLOW

The most convenient way to profile a script execution at the function level is to use the **-p** option of **%run** from within IPython.

It is a shortcut for the **cProfile** workflow, which has two main steps:

1. Run the code to be profiled via the cProfile's **run()** (or **runctx()**) function. This counts and times function calls. From this it generates a profiling dataset.

2. Process and display the profile data. In the simplest case (e.g., **cProfile.run('foo()')**), a predefined report is generated and printed. For finer control, you can save the raw data to a file and process it using the **pstats** module.

IPython Magic Run

The bulk of the time (90%) spent in this program is in just two operations. Since **load_citations()** accesses disk (typically slow), let's focus on **build_index()**.

```
In [1]: %run -p citation_network_slow.py

1088860 function calls in 2.392 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.365    1.365    1.402    1.402 citation_network_slow.py:55(build_index)
        1    0.787    0.787    0.914    0.914 citation_network_slow.py:40(load_citations)
   352808    0.094    0.000    0.094    0.000 {method 'split' of 'unicode' objects}
   705633    0.058    0.000    0.058    0.000 {method 'append' of 'list' objects}
        1    0.039    0.039    2.391    2.391 citation_network_solution.py:36(<module>)
        1    0.017    0.017    0.017    0.017 {sorted}
        1    0.011    0.011    0.017    0.017 citation_network_slow.py:75(citation_counts)
     1277    0.011    0.000    0.011    0.000 {_codecs.ascii_decode}
        1    0.005    0.005    0.005    0.005 {method 'items' of 'dict' objects}
        1    0.002    0.002    2.353    2.353 citation_network_slow.py:117(main)
    27782    0.002    0.000    0.002    0.000 {len}
     1277    0.001    0.000    0.012    0.000 ascii.py:25(decode)
```

Fixing the Slow Function

Two things stand out immediately – repeated calls to append, and testing for containment within a list.

BEFORE

```
def build_index(citations, reverse=False):

    index = {}
    for paper, cited_paper in citations:

        if reverse:
            paper, cited_paper = cited_paper, paper

        if paper not in index:
            index[paper] = []

        if cited_paper not in index:
            index[cited_paper] = []

        if cited_paper not in index[paper]:
            index[paper].append(cited_paper)

    return index
```

AFTER

```
def build_index(citations, reverse=False):

    index = {}
    for paper, cited_paper in citations:

        if reverse:
            paper, cited_paper = cited_paper, paper

        if paper not in index:
            index[paper] = set()

        if cited_paper not in index:
            index[cited_paper] = set()

        index[paper].add(cited_paper)


    return index
```

IPython Magic Run

Changing one data structure (list to set) has reduced our time to completion by half.

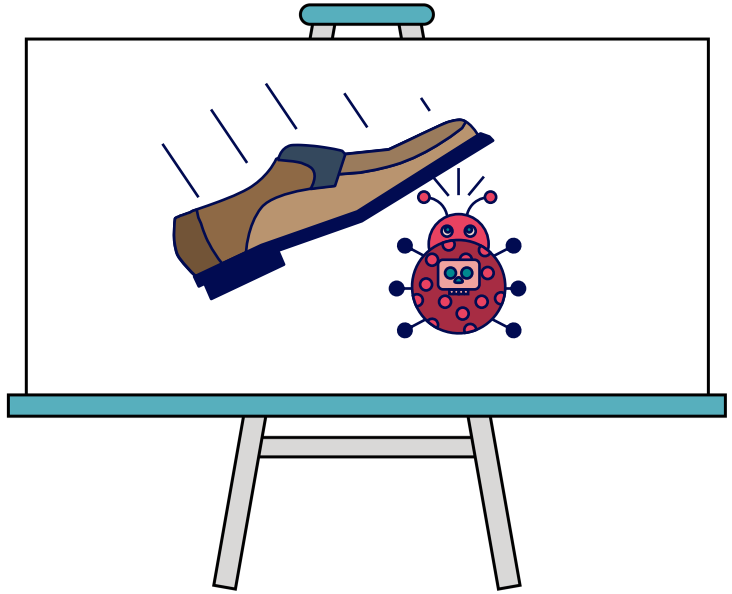
```
In [1]: %run -p citation_network_solution.py

1088860 function calls in 1.347 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.800    0.800    0.928    0.928 citation_network_solution.py:40(load_citations)
        1    0.281    0.281    0.346    0.346 citation_network_solution.py:55(build_index)
   352808    0.095    0.000    0.095    0.000 {method 'split' of 'unicode' objects}
   352807    0.065    0.000    0.065    0.000 {method 'add' of 'set' objects}
        1    0.035    0.035    1.346    1.346 citation_network_solution.py:36(<module>)
   352826    0.021    0.000    0.021    0.000 {method 'append' of 'list' objects}
        1    0.016    0.016    0.016    0.016 {sorted}
        1    0.013    0.013    0.019    0.019 citation_network_solution.py:74(citation_counts)
     1277    0.011    0.000    0.011    0.000 {_codecs.ascii_decode}
        1    0.005    0.005    0.005    0.005 {method 'items' of 'dict' objects}
    27782    0.002    0.000    0.002    0.000 {len}
        1    0.002    0.002    1.311    1.311 citation_network_solution.py:116(main)
     1277    0.001    0.000    0.012    0.000 ascii.py:25(decode)
```

Know Your Data Structures

Element-access times impact performance

	Insert	Remove	Contains	Retrieve
list	linear*	linear*	linear	constant
deque	linear**	linear**	linear	linear**
set / dict	constant	constant	constant	constant

* constant for last item in container
** constant for first and last item in container



Debugging

Lecture 06
Profiling & Debugging

Debugging Code

1. The first step to debugging is to write code that is **easy to understand**.

2. The second step to debugging is to **monitor the execution** of your program to look for aberrant behavior (i.e., logging).

3. The third step to debugging is to check the **documentation**.

4. If the previous three steps are not enough, you may find it helpful to use a **debugger** (a tool that can pause the execution of your buggy program).

What is pdb?

- **pdb** is part of the standard library.
- **pdb**, like Python, is interactive and interpreted, allowing for the execution of arbitrary Python code in the context of any stack frame.
- **pdb** can debug a “post-mortem” condition; it can also be called under program control.
- **ipdb** (not in standard Python library) is similar but includes tab completion and syntax highlighting.

IPython and pdb

```
# ipython can call pdb automatically upon error

In [1]: %pdb
Automatic pdb calling has been turned ON
In [2]: import middle
In [3]: middle.run()
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Z:\projects\Training\pdb\<console>
...

Z:\projects\Training\pdb\middle.py in get_middle(item_list)
      9
     10     if( num_items % 2 ) :
---> 11         return item_list[half]
     12
     13     return item_list[(half - 1):(half + 1)]

IndexError: list index out of range
> z:\projects\training\pdb\middle.py(11)get_middle()
---> 10 return item_list[half]
     11
     12     return item_list[(half - 1):(half + 1)]

ipdb>
```

IPython and pdb

```
# it can also be used to enter into a post-mortem

In [1]: %run citation_network_solution.py
...

/home/mike/class/demo/software_craftsmanship/citation_network/citation_network_solution.py in
load_citations(filename, header_size)
     49
     50         for line in f:
---> 51             citing, cited = [int(item) for item in line.split()]
     52             citations.append((citing, cited))
     53     return citations

ValueError: need more than 0 values to unpack

In [2]: %debug
>/home/mike/class/demo/software_craftsmanship/citation_network/citation_network_solution.py(52)load_citations(
)
     50         for line in f:
     51             citing, cited = [int(item) for item in line.split()]
---> 52             citations.append((citing, cited))
     53     return citations
     54

ipdb>
```

pdb Commands

pdb runs as an interactive session having a specific set of commands. Some of the more common **pdb** commands:

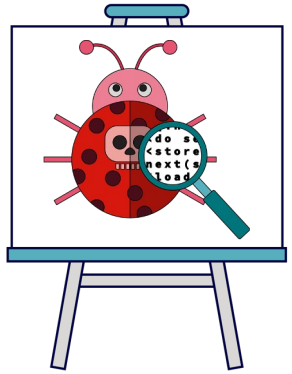
Command	Explanation
`h(elp) [command]`	One of the most important! Lists all the commands available, or help on a specific command.
`u(p) / d(own)`	Pop up or push down the execution stack.
`s(tep) / n(ext)`	Execute the current line only. **step** will push into a function call and **next** will execute the function call and move to the next statement in the current function.
`b(reak) [[filename:]lineno | function[, condition]]`	Set a breakpoint at a specific file/line or function and optionally if a specific condition is met. If no arguments are given, list all the breakpoints & their numbers.
`l(ist) [first[, last]]`	List the source code at the point of execution. **first** and **last** set a range for the number of lines printed.
`p / pp [expression]`	Print or "pretty print" expression in the context of the current frame.
`a(rgs)`	Print the arguments for the current function.

Post-Mortem Debugging

```
# debugging from origin of exception

In [1]: %run citation_network_solution.py
...

ValueError: need more than 0 values to unpack

In [2]: %debug
>/home/mike/class/demo/software_craftsmanship/citation_network/citation_network_solution.py(52)load_ci
tations()
     50         for line in f:
     51             citing, cited = [int(item) for item in line.split()]
---> 52             citations.append((citing, cited))
     53     return citations
     54

ipdb> citing, cited
(1001, 9309097)
ipdb> line
'\n'

In [3]:
```

Exercise: Profiling & Debugging

- Debugging



Lecture 07

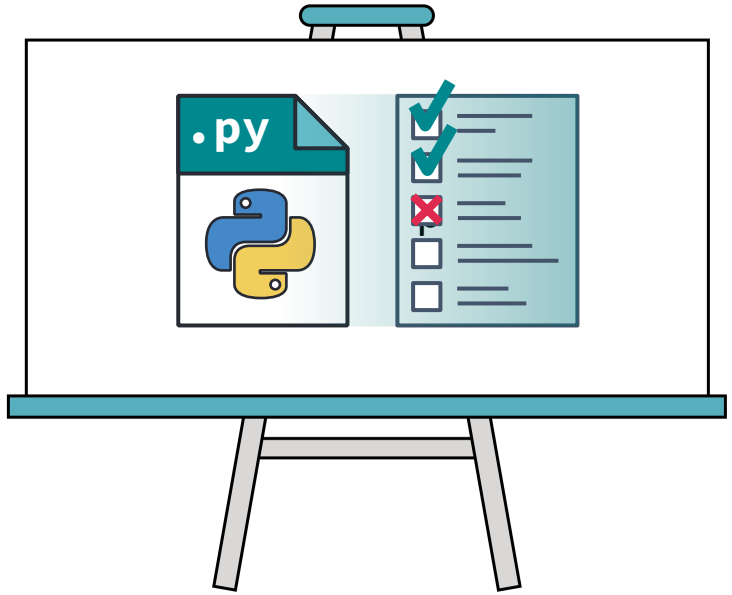
Unit Testing

Software Engineering
for Scientists and Engineers

Discussion

How do you currently test your code?

Table of Contents

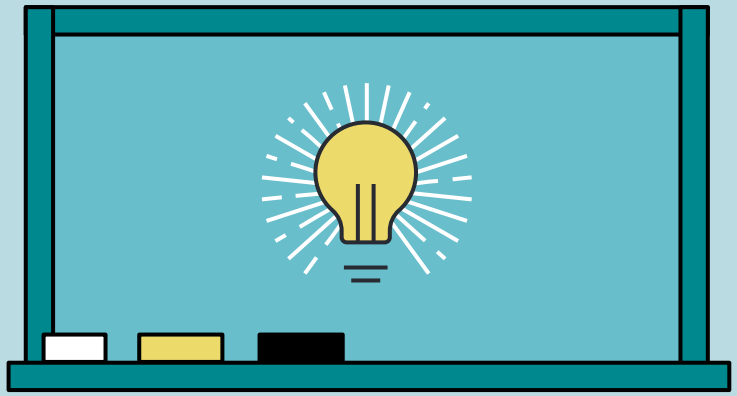


Introduction to Unit Testing

Lecture 07
Unit Testing

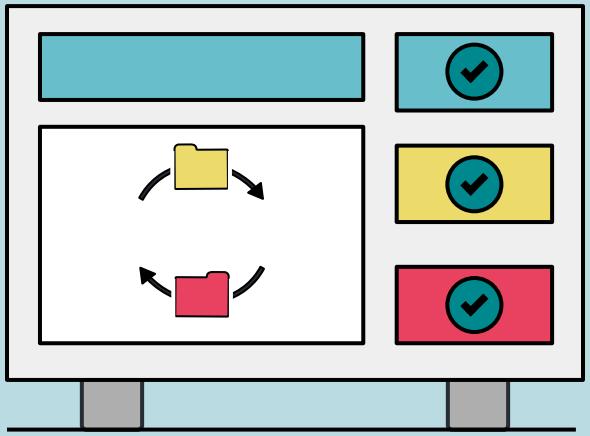
What is Unit Testing?

- Software testing method focused on determining if **units of code** are "fit for use"
  - Units of code refer to sections of a program (a module or an individual function)
  - "Fit for use" means the unit of code meets its design expectations and behaves as intended
  - Typically run automatically and independently
  - Small, unit-sized testing can lead to easier evaluation of complex programs
- Defining a strict set of expected output(s) that must be satisfied given a pre-defined set of input(s).



Why Unit Testing?

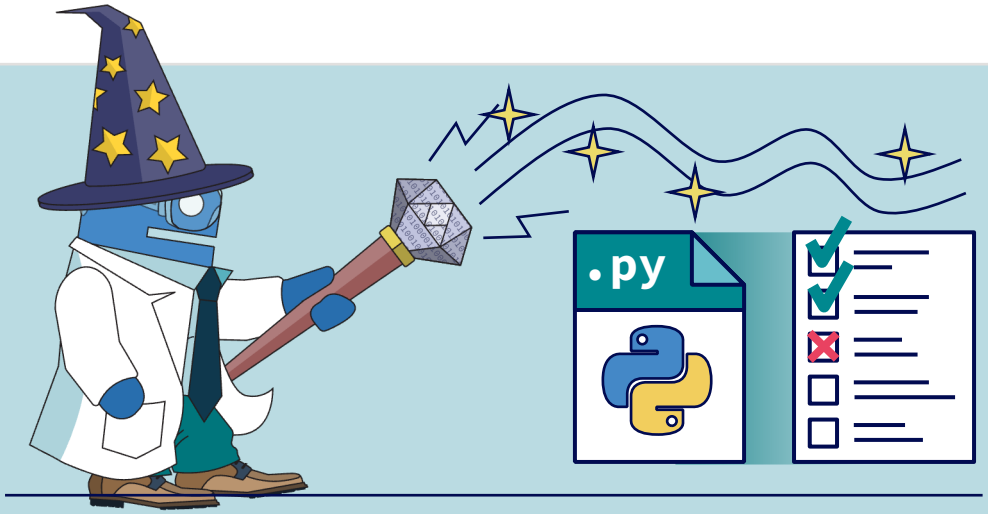
- Isolate individual parts of a program and ensure the single units are correct
  - Define strict set of output(s) that must be satisfied
- Having a test suite facilitates debugging and maintenance
  - Provides ability to refactor code base later while ensuring new updates don't break existing functionality
  - Capturing known use cases and edge cases reveal bugs early on
- Reduce uncertainty in code base
  - Unit tests provide a snapshot of how the system should operate as a whole
  - Can be used to test the system from the bottom up



Unit Tests Dispel Illusion of Code Correctness

Testers don’t like to break things; they like to **dispel the illusion that things work**.

Pettichord, Kaner, Bach.
Lessons Learned in Software Testing. 2001.



Thought Experiment

Does the code to the right behave as expected?

- Notice this is a function (unit of code)
- How would you go about testing this for correctness?
- Do you have use cases in your mind to test?
- Would you run these manually?
- What about edge cases or null/zero values?

All of these are important components of writing unit tests.

```
>>> def simple_div(a, b):
...     """
...     Return quotient of a over b.
...     """
...     return a / b

>>> simple_div(25, 5)
???

>>> simple_div(1, 3)
???

>>> simple_div(1, 0)
???

>>> import numpy as np
>>> simple_div(1, np.nan)
???
```

Thought Experiment

Does the code to the right behave as expected?

- Instead of running each of these mini-validations manually, a test suite can be created to check intended behavior
- This aims to enforce code accuracy, consistency, and behavior

Writing tests are important, but sometimes figuring out what to test can be tricky. Start with 3 things:
(1) the simplest case that should work
(2) different simple case that should work
(3) common edge cases like empty inputs of **`None`**, **`0`**, **`NaN`**, **`''`**.

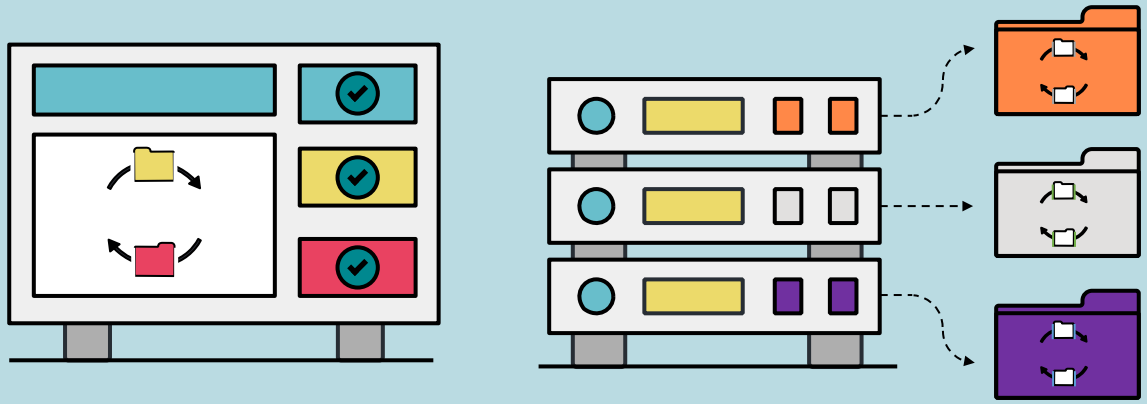
```
>>> def simple_div(a, b):
...     """
...     Return quotient of a over b.
...     """
...     return a / b

# Expecting float?
>>> simple_div(25, 5)
5.0

# Expecting 16 decimal places?
>>> simple_div(1, 3)
0.3333333333333333

# Expecting ZeroDivisionError?
>>> simple_div(1, 0)
ZeroDivisionError: division by zero

# Expecting nan?
>>> import numpy as np
>>> simple_div(1, np.nan)
nan
```

Unit Test Cautions

- Unit tests only test the system where tests have been specified
  - Blind spots may exist where left untested

- Test coverage can provide false sense of confidence
  - 100% coverage (all lines in the code base have a corresponding test) doesn't mean there are no bugs in the system
  - Tests need to be realistic, useful, and non-trivial

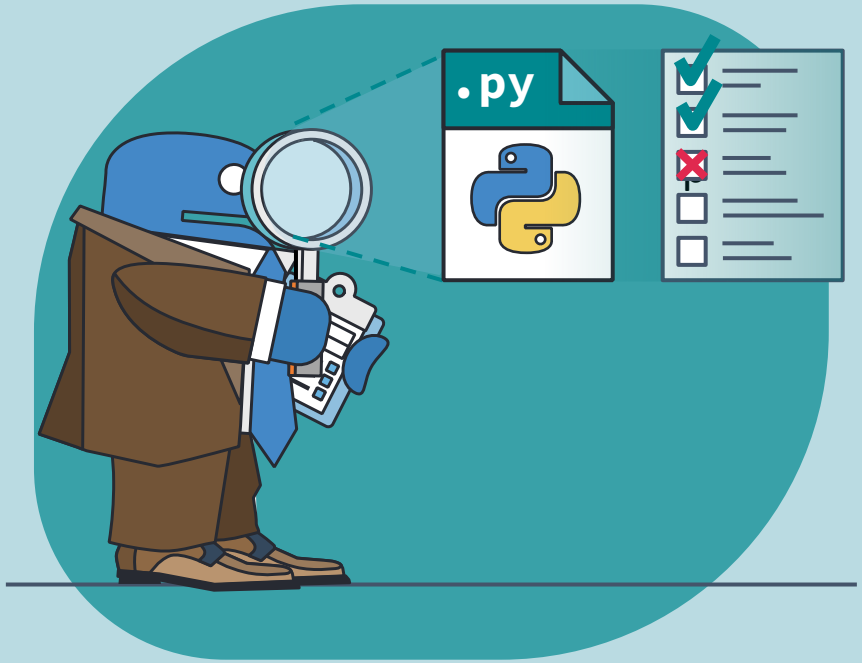
- Unit tests are the smallest pieces of a larger testing puzzle
  - **Unit Testing**: test individual functions and methods
  - **Integration Testing**: test interactions between units
  - **System Testing**: test the complete system or project as a whole
  - **Acceptance Testing**: test usability and adoption for end users



Python's Unit Test Framework I

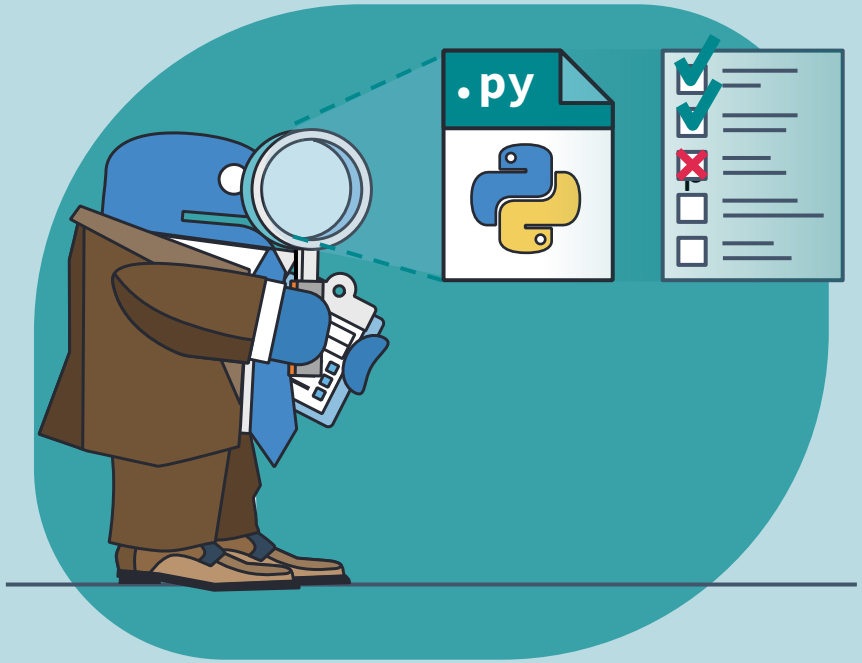
- You *could* write your own test suites by manually using Python's **`assert`** statement
  - Error prone, tedious, and not as easy to automate
  - The **`assert`** statement is a convenient way to insert debugging assertions into a program and is more used for validation than unit testing

- Instead, use Python's built in unit testing framework **`unittest`**
  - Supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework



Python's Unit Test Framework II

- Object-oriented with four key concepts
  - Test case
    - The **individual unit of testing** that checks for a specific response to a particular set of inputs.
    - **`unittest`** provides a base class, **`TestCase`**, which may be used to create new test cases.
  - Test suite
    - Collection of test cases, test suites, or both. It is used to **aggregate tests** that should be executed together.
  - Test runner
    - Component that **orchestrates the execution** of tests and provides the outcome to the user.
  - Test fixture
    - Represents the **preparation** needed to perform **one or more tests**, and any associated **cleanup** actions.
    - This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.



`unittest` Assert Methods I

- Create a class that inherits from **unittest.TestCase**
- Define methods that start with **test_** and use an **assert*()** method
- **Warning**: Python's **`assert`** statement is different than **`unittest`**'s assert **`methods`**

```
import unittest


class TestExampleClass(unittest.TestCase):
    def test_example_equal(self):
        a = 42
        b = 42
        c = 5
        self.assertEqual(a, b)
        self.assertNotEqual(a, c)
```

test_methods.py

`unittest` Assert Methods II

- There are a variety of `assert-like` commands that can be used
- Here, *boolean* assert methods are shown

```
def is_even(number):
    return number % 2 == 0
```

methods.py

```
...
    def test_is_even(self):
        self.assertTrue(is_even(4))
        self.assertFalse(is_even(5))
        self.assertFalse(is_even(2.5))

        # The below would fail the test
        # self.assertFalse(is_even(4))
```

test_methods.py

`unittest` Assert Methods III

- *Membership testing* works with all container types: **`lists`**, **`sets`**, **`dictionaries`** (keys and values), **`arrays`**, etc.
- It is equivalent to **`assert <value> in <container>`**

```
...
    def test_example_membership(self):
        my_set = {1, 2, 3}
        self.assertIn(1, my_set)
        self.assertNotIn(4, my_set)

        # The below would fail the test
        # self.assertIn(4, my_set)
        # self.assertNotIn(1, my_set)
```

`unittest` Assert Methods IV

- *Instance checking* checks that an object is of a particular **`type`**
- It is equivalent to **`assert isinstance(<obj>, <check_type>)`**

```
...
    def test_example_instance(self):
        my_dict = {"a": 1, "b": 2, "c": 3}
        self.assertIsInstance(my_dict, dict)

        # The below would fail the test
        # self.assertNotIsInstance(my_dict, dict)
```

`unittest` Assert Methods V

- To account for *non-exact floating point arithmetic*, assume a certain precision, not equality (default is **7** places)

```
...
    def test_example_almost_equal(self):
        result = 1.4 + 2.3  # 3.6999999999999997

        # checks that round(abs(result - 3.7), 7) == 0
        self.assertAlmostEqual(result, 3.7)

        # The below would fail the test
        # self.assertEqual(result, 3.7)
        # self.assertAlmostEqual(result, 3.7, places=16)
```

`unittest` Assert Methods Conclusion

- For a full list of the assert-like methods, see the **`unittest.TestCase`** docs
  - https://docs.python.org/3/library/unittest.html - unittest.TestCase

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

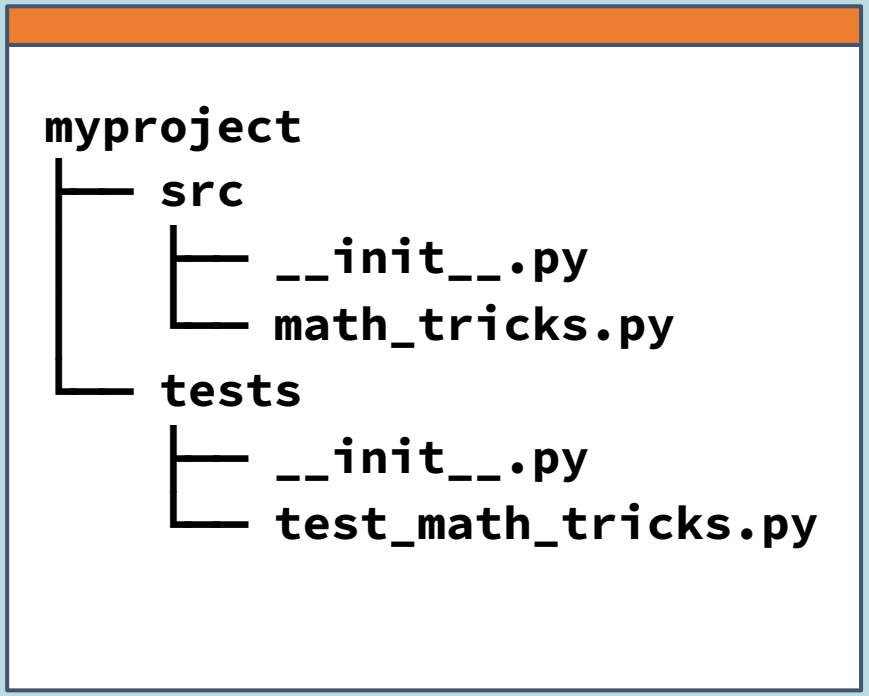
Method	Checks that	New in
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`	
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`	
`assertGreater(a, b)`	`a > b`	3.1
`assertGreaterEqual(a, b)`	`a >= b`	3.1
`assertLess(a, b)`	`a < b`	3.1
`assertLessEqual(a, b)`	`a <= b`	3.1
`assertRegex(s, r)`	`r.search(s)`	3.1
`assertNotRegex(s, r)`	`not r.search(s)`	3.2
`assertCountEqual(a, b)`	*a* and *b* have the same elements in the same number, regardless of their order.	3.2

`unittest` Demo

- Assume that we have a project directory called **`my_project`** as seen on the left that contains a **`math_tricks`** module

- Three functions we need to test:
  - **`gauss_sum(n)`**
  - **`count_permutations(n, k)`**
  - **`count_combinations(n, k)`**

- Tests usually live in their own python files and are separate from the source code they test



Demo: Gaussian Sum

- The **`gauss_sum(n)`** function implements a clever summation strategy to sum all the positive integers less than or equal to a provided integer (**n**)
- It is rumored to have been discovered by Johann Karl Friedrich Gauss when his math teacher challenged the class to add the numbers 1 to 100
- The “trick” works by summing pairs of integers rather than summing consecutive integers together

```
def gauss_sum(n):
    """
    Return the total sum of all positive integers less
    than or equal to n.
    Parameters
    ----------
    n : int
        The highest positive integer value to consider
        in the summation.
    Raises
    ------
    ValueError
        If the number provided is not positive.
    Examples
    --------
    # 5 + 4 + 3 + 2 + 1 = 15
    >>> gauss_sum(5)
    15.0
    # 100 + 99 + 98 ... + 3 + 2 + 1 = 5050
    >>> gauss_sum(100)
    5050.0
    """
    return ((n + 1) * n) / 2
```

math_tricks.py

Demo: Testing Gaussian Sum I

- If we wanted to test the **`gauss_sum()`** function, we could do it manually with some predefined tests scenarios

- This is tedious, error-prone, and does not lend to automation
  - What if we change or update the **`gauss_sum()`** function?
  - Who will run these tests again? How do you know that person will use the same test scenarios?
  - Do you really want to have to do this each time something is changed in your package?

```
>>> from my_project.math_tricks import gauss_sum

# Manual checking
>>> gauss_sum(5)
15.0
>>> gauss_sum(5) == 5 + 4 + 3 + 2 + 1
True

>>> gauss_sum(100)
5050.0
>>> gauss_sum(100) == sum(range(100 + 1))
True

>>> gauss_sum(0)
0.0
>>> gauss_sum(0) == 0
True

# Manual checking with randomness
>>> import random
>>> n = random.randint(1, 1_000_000)
>>> expected = sum(range(n + 1))
>>> print(f"n={n:,}, truth={expected:,}")
n=659,762, truth=217,643,278,203
>>> gauss_sum(n)
217643278203.0
```

Demo: Testing Gaussian Sum II

- A better way to do this is to use the **`unittest`** module to write a test that operates on the **`gauss_sum()`** function
- Follow the *Setup*, *Exercise*, *Verify* pattern when writing unit tests
  - The *Setup* stage **defines any necessary inputs and the expected output** when calling the function (these are the strict input/output pairs that must be satisfied for a successful test run)
  - The *Exercise* stage **calls the function** that you'd like to test (usually saving the output as result)
  - The *Verify* step **uses an assert*() method to ensure that the expected output and the result of calling the function are equivalent**

```python
import math
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    def test_gauss_sum_5(self):
        # Setup
        n = 5
        expected = 15

        # Exercise
        result = math_tricks.gauss_sum(n)

        # Verify
        self.assertEqual(result, expected)
```

test_math_tricks.py

Demo: Testing Gaussian Sum III

- All "manual" tests can be encapsulated into the **`TestGaussSum`** class
- Notice that each unit test follows the *Setup*, *Exercise*, and *Verify* pattern but this time without the comments
- **Required** that each unit test must start with the **`test_`** prefix

```python
class TestGaussSum(unittest.TestCase):
    ...
    def test_gauss_sum_100(self):
        n = 100
        expected = 5050
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_0(self):
        n = 0
        expected = 0
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_random_int(self):
        n = random.randint(1, 1_000_000)
        expected = sum(range(n + 1))
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)
```

test_math_tricks.py

Demo: Executing `TestGaussSum`

- After writing unit tests for **`gauss_sum()`**, **`TestGaussSum`** needs to be run
- Python provides two options to execute tests:
  - Execute the tests via the command line using a call to the **`unittest`** module
  - Execute the test file as a python script
- This slide shows how to make the **`test_math_tricks.py`** file executable as a python script by adding the **`if __name__ == "__main__"`** clause at the bottom of the file

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    def test_gauss_sum_5(self):
        n = 5
        expected = 15
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_100(self):
        n = 100
        expected = 5050
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_0(self):
        n = 0
        expected = 0
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_random_int(self):
        n = random.randint(1, 1_000_000)
        expected = sum(range(n + 1))
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)


if __name__ == "__main__":
    unittest.main()
```

test_math_tricks.py

Demo: `unittest` Output

Executing Tests as Python Script

```
$ pwd
/Desktop/my_project

$ ls
__init__.py  math_tricks.py  tests

$ python tests/test_math_tricks.py
....
----------------------------------------
Ran 4 tests in 0.017s

OK
```

Executing Tests via `unittest` module

```
$ pwd
/Desktop/my_project

$ ls
__init__.py  math_tricks.py  tests

$ python -m unittest
....
----------------------------------------
Ran 4 tests in 0.011s

OK
```

When executed without arguments (**`python -m unittest`**), test discovery started. In order to be compatible with test discovery, all test files must be modules or packages importable from the top-level directory of the project. Make sure all your test methods start with the name **`test_`**.
As a shortcut, **`python -m unittest`** is the equivalent of **`python -m unittest discover`**. If you want to pass arguments to test discovery the discover sub-command must be used explicitly. https://docs.python.org/3/library/unittest.html#test-discovery

Demo: `unittest` Output cont'd

- Verbosity flag (**`-v`**) increases the information reported
- Specific individual tests can be run without executing the entire test suite

```
$ python -m unittest -v
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_100 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_5 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_random_int (tests.test_math_tricks.TestGaussSum) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK

$ python -m unittest tests.test_math_tricks.TestGaussSum.test_gauss_sum_100
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Demo: Gaussian Failure

- One of the nicest features of Python's **`unittest`** module is its ability to generate test failure reports
- Such a report can be demonstrated by *incorrectly updating* the **`gauss_sum()`** function to always return **`42`**
  - This resembles someone refactoring the source code and changing something that will cause an error in the original function
  - If the change shown on this slide were made, expect our test suite to correctly capture this fact and fail the tests currently in place

```
def gauss_sum(n):
    """
    Return the total sum of all positive integers less
    than or equal to n.
    Parameters
    ----------
    n : int
        The highest positive integer value to consider
        in the summation.
    Raises
    ------
    ValueError
        If the number provided is not positive.
    Examples
    --------
    # 5 + 4 + 3 + 2 + 1 = 15
    >>> gauss_sum(5)
    15.0
    # 100 + 99 + 98 ... + 3 + 2 + 1 = 5050
    >>> gauss_sum(100)
    5050.0
    """
    # return ((n + 1) * n) / 2
    return 42
```

math_tricks.py

Demo: Gaussian Failure cont'd

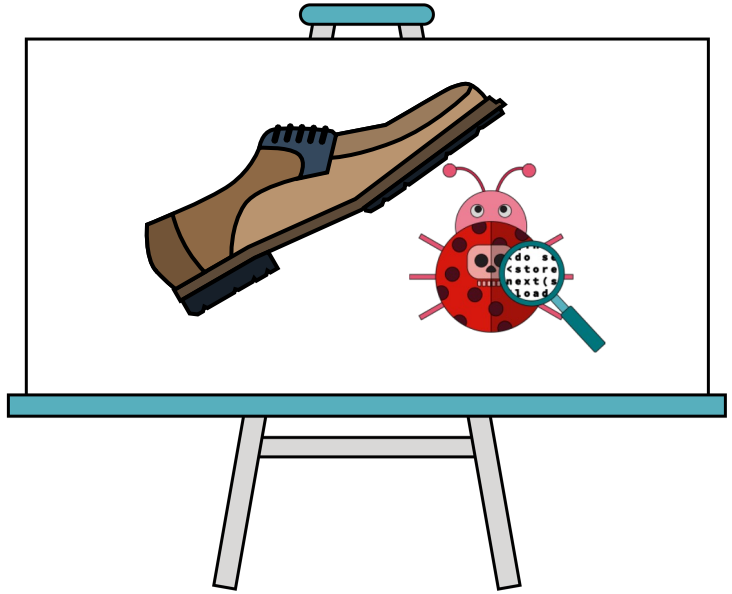
- Test failures are reported as **F**
- Metadata on failures help spot bugs and update source code accordingly

```
$ python -m unittest -v
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... FAIL
test_gauss_sum_100 (tests.test_math_tricks.TestGaussSum) ... FAIL
test_gauss_sum_5 (tests.test_math_tricks.TestGaussSum) ... FAIL
test_gauss_sum_random_int (tests.test_math_tricks.TestGaussSum) ... FAIL

======================================================================
FAIL: test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Desktop/myproject/tests/test_math_tricks.py", line 28, in test_gauss_sum_0
    self.assertEqual(result, expected)
AssertionError: 42 != 0

======================================================================
FAIL: test_gauss_sum_100 (tests.test_math_tricks.TestGaussSum)
...
```

Demo: Gaussian Failure cont'd

- To stop execution as soon as the first failure occurs, use **`--failfast`** flag

```
$ python -m unittest -v --failfast
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... FAIL

======================================================================
FAIL: test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Desktop/myproject/tests/test_math_tricks.py", line 28, in test_gauss_sum_0
    self.assertEqual(result, expected)
AssertionError: 42 != 0

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

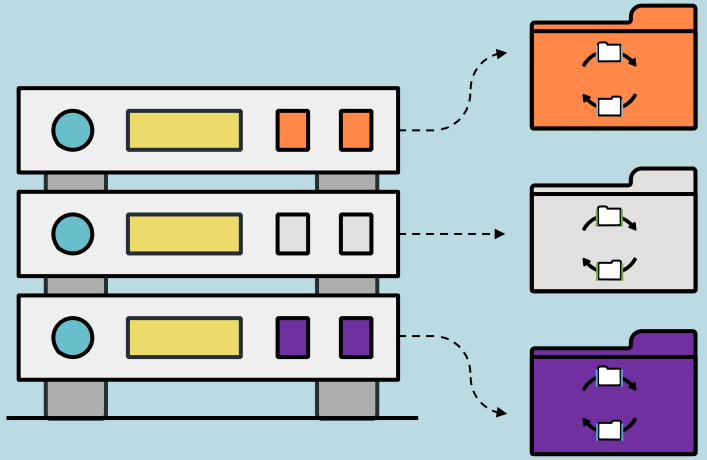


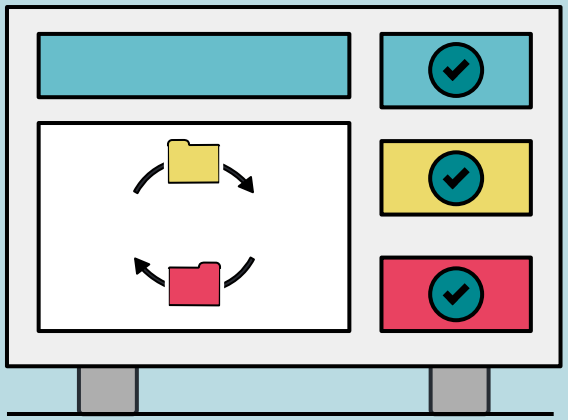
Advanced Unit Testing

Lecture 07
Unit Testing

Simplifying Unit Tests

- Writing unit tests can become a verbose and repetitive process
  - Testing the same function but with slightly different inputs
  - Testing the same flow of information from one unit of code to another, but with the same parent directory and files to load
- Instead of writing the same boilerplate code over and over, the **`unittest`** module provides tools for simplifying unit tests
  - It's often a good idea to keep edge cases as separate, standalone unit tests when simplifying
- If it's hard to write a test for a unit of code, it's a hint that you might need to refactor
  - Functions should do one thing and one thing only
  - This facilitates the writing of unit tests





Simplifying Unit Tests cont'd

- Notice the repeated code in the unit tests
  - The same *Execute* and *Verify* steps are run for each test
  - A similar *Setup* step is run, but defines different inputs and expected outputs

- The three repetitive unit tests to the right could be replaced with:
  - A pre-defined list of inputs (**`5`**, **`100`**, and a random integer)
  - A for loop to iterate over each case
  - A closed form solution **`sum(range(n+1))`**

- Notice the **expected** value is not employing the function to be tested. A different method to calculate the sum (not the gaussian sum trick) is used to define the expected value

The **`test_gauss_sum_0`** unit test is left out to keep edge cases (like testing on **`0`** and **`NaN`**) as separate tests. Often, these should be handled differently than typical integers and floats.

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    def test_gauss_sum_5(self):
        n = 5
        expected = 15
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_100(self):
        n = 100
        expected = 5050
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_0(self):
        n = 0
        expected = 0
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum_random_int(self):
        n = random.randint(1, 1_000_000)
        expected = sum(range(n + 1))
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)


if __name__ == "__main__":
    unittest.main()
```

test_math_tricks.py

Simplifying Unit Tests cont'd

- Notice the repeated code in the unit tests
  - The same *Execute* and *Verify* steps are run for each test
  - A similar *Setup* step is run, but defines different inputs and expected outputs

- The three repetitive unit tests to the right could be replaced with:
  - A pre-defined list of inputs (**`5`**, **`100`**, and a random integer)
  - A for loop to iterate over each case
  - A closed form solution **`sum(range(n+1))`**

- Notice the **expected** value is not employing the function to be tested. A different method to calculate the sum (not the gaussian sum trick) is used to define the expected value

The **`test_gauss_sum_0`** unit test is left out to keep edge cases (like testing on **`0`** and **`NaN`**) as separate tests. Often, these should be handled differently than typical integers and floats.

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    def test_gauss_sum_0(self):
        n = 0
        expected = 0
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]

        for n in inputs:
            expected = sum(range(n + 1))
            result = math_tricks.gauss_sum(n)
            self.assertEqual(result, expected)


if __name__ == "__main__":
    unittest.main()
```

test_math_tricks.py

Simplifying Unit Tests Output

Previously (Four Separate Tests)

```
$ python -m unittest -v
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_100 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_5 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_random_int (tests.test_math_tricks.TestGaussSum) ... ok

----------------------------------------------------------------
Ran 4 tests in 0.022s

OK
```

Simplified Tests (one `test_gauss_sum()`)

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok

----------------------------------------------------------------
Ran 2 tests in 0.011s

OK
```

Simplifying Unit Tests Gotcha

- Saving time and simplifying are often goods ideas when writing unit tests
  - When tests have overlapping/repeat code, it's a good idea to use a predefined set of inputs and loop over those inputs with a single test function

- However, **there is one glaring issue with simplifying unit tests in this way**
  - What happens during a failure?
  - Incorrectly specify the **`expected`** result to be **`42`** to reveal this gotcha

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    def test_gauss_sum_0(self):
        n = 0
        expected = 0
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]

        for n in inputs:
            expected = 42
            result = math_tricks.gauss_sum(n)
            self.assertEqual(result, expected)


if __name__ == "__main__":
    unittest.main()
```

test_math_tricks.py

Simplifying Unit Tests Gotcha cont'd

- Which input in the pre-defined list **`[5, 100, random.randint(1, 1_000_000)]`** is causing the error?
- Without the original input, it's hard to know what caused the code to fail

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... FAIL
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Desktop/myproject/tests/test_math_tricks.py", line 43, in test_gauss_sum
    self.assertEqual(result, expected)
AssertionError: 15.0 != 42

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Sub Tests

- Instead of creating our own tooling to print out each test case, **`unittest`** has a convenient **`subTest`** context manager
- **`unittest.subTest()`** will handle reporting each individual sub test for a single unit test function
  - This is done by specifying which test metadata to report
  - Here, using each input, **`n`**, as the subtest metadata
- Notice what happens when using **`subTest()`** and keeping the erroneous expected value of **`42`**

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    def test_gauss_sum_0(self):
        n = 0
        expected = 0
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]

        for n in inputs:
            with self.subTest(n=n):
                expected = 42
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)


if __name__ == "__main__":
    unittest.main()
```

test_math_tricks.py

Sub Tests cont'd

- Better test report generated with each individual sub-test indicator (**n=___**)

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ...
ok

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Desktop/myproject/tests/test_math_tricks.py", line 50, in test_gauss_sum
    self.assertEqual(result, expected)
AssertionError: 15.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=100)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Desktop/myproject/tests/test_math_tricks.py", line 50, in test_gauss_sum
    self.assertEqual(result, expected)
AssertionError: 5050.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=462592)
----------------------------------------------------------------------
```

Sub Tests cont'd

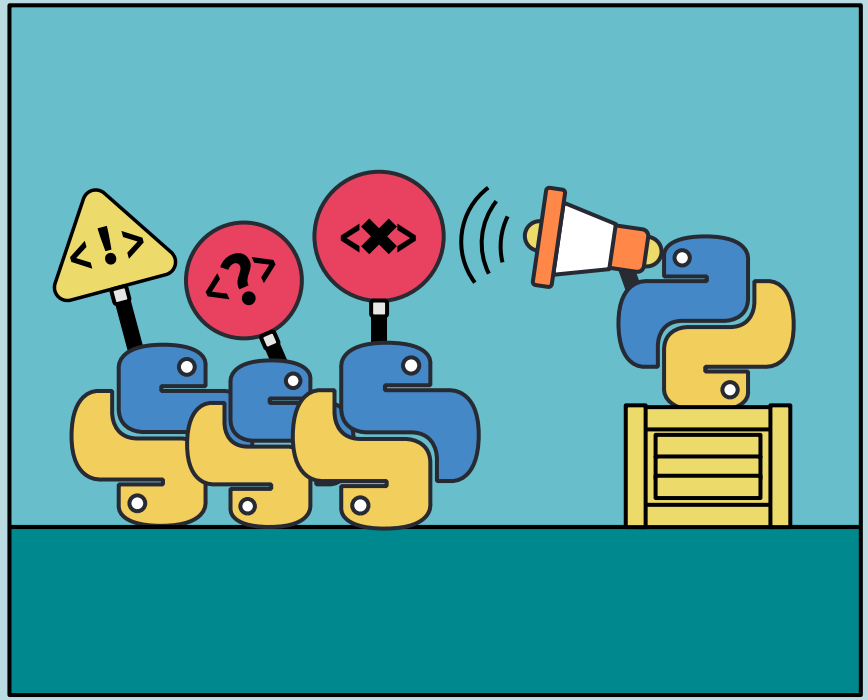
- Correcting the expected value (**`sum(range(n+1))`**) returns tests to passing
- Notice sub-tests are only reported on failures

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.009s

OK
```

Capturing Exceptions in Unit Tests

- What if a piece of code should halt the execution and warn a user about incorrect input?
  - How would a test be written to ensure the correct **`Exception`** is thrown?
  - How can we ensure the program will halt?
  - How do we test the correct error message is provided?

- Python's **`unittest`** module comes with assert methods for raising specific **`Exception`** types



Demo: Capturing Exceptions

- A new edge case has been encountered where users try to provide negative integers to the **`gauss_sum()`** function
- First, write a unit test to see how the current function behaves and whether it will correctly raise an exception

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    ...
    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]

        for n in inputs:
            with self.subTest(n=n):
                expected = sum(range(n + 1))
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)

    # Want to raise an Exception when user provides
    # negative integer
    def test_gauss_sum_negative_int(self):
        n = random.randint(1, 1_000_000) * -1
        expected = "warning!"
        result = math_tricks.gauss_sum(n)
        self.assertEqual(result, expected)

if __name__ == "__main__":
    unittest.main()
```

test_math_tricks.py

Demo: Capturing Exceptions cont'd

- This is a bug in our code!

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) ... FAIL

======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/lthomas/Desktop/repos/class-
material/demo/Software_Engineering/unittest/02_myproject/tests/test_math_tricks.py", line 33, in
test_gauss_sum_negative_int
    self.assertEqual(result, expected)
AssertionError: 226406289241.0 != 'warning!'

----------------------------------------------------------------------
Ran 3 tests in 0.002s

FAILED (failures=1)
```

Refactoring to Raise Exception

- A bug in the **`gauss_sum()`** function was discovered when implementing a unit test for negative integer inputs
- Update the **`gauss_sum()`** function to correctly raise a **`ValueError`** if the input is negative
  - Notice that the error message will specify the value of the incorrect input

```
def gauss_sum(n):
    """
    Return the total sum of all positive integers less
    than or equal to n.
    Parameters
    ----------
    n : int
        The highest positive integer value to consider
        in the summation.
    Raises
    ------
    ValueError
        If the number provided is not positive.
    Examples
    --------
    # 5 + 4 + 3 + 2 + 1 = 15
    >>> gauss_sum(5)
    15.0
    # 100 + 99 + 98 ... + 3 + 2 + 1 = 5050
    >>> gauss_sum(100)
    5050.0
    """
    if n < 0:
        raise ValueError(f"n must be positive but input is n={n}")
    return ((n + 1) * n) / 2
```

math_tricks.py

Refactoring to Raise Exception cont'd

- The source code has been refactored correctly, but now the unit test needs fixing!

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) ... ERROR

======================================================================
ERROR: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Desktop/myproject/tests/test_math_tricks.py", line 32, in test_gauss_sum_negative_int
    result = math_tricks.gauss_sum(n)
  File "/Users/lthomas/Desktop/repos/class-
material/demo/Software_Engineering/unittest/02_myproject/math_tricks.py", line 22, in gauss_sum
    raise ValueError(f"n must be positive but input is n={n}")
ValueError: n must be positive but input is n=-748422

----------------------------------------------------------------------
Ran 3 tests in 0.002s

FAILED (errors=1)
```

Refactoring to Raise Exception

- The previous **`test_gauss_sum_negative_int()`** unit test needs to be refactored to ensure the update to **`gauss_sum()`** correctly raises a **`ValueError`**
- Notice that a check for the raising of a **`ValueError`** can be performed as well as a check for the error message associated with the **`ValueError`** using a context manager and **`str(cm.exception)`**

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    ...
    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]

        for n in inputs:
            with self.subTest(n=n):
                expected = sum(range(n + 1))
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)

    # Post refactoring gauss_sum() to raise Exception
    def test_gauss_sum_negative_int(self):
        n = random.randint(1, 1_000_000) * -1
        expected = f"n must be positive but input is n={n}"

        with self.assertRaises(ValueError) as cm:
            math_tricks.gauss_sum(n)

        self.assertEqual(str(cm.exception), expected)
    ...
```

test_math_tricks.py

Demo: Capturing Exceptions cont'd

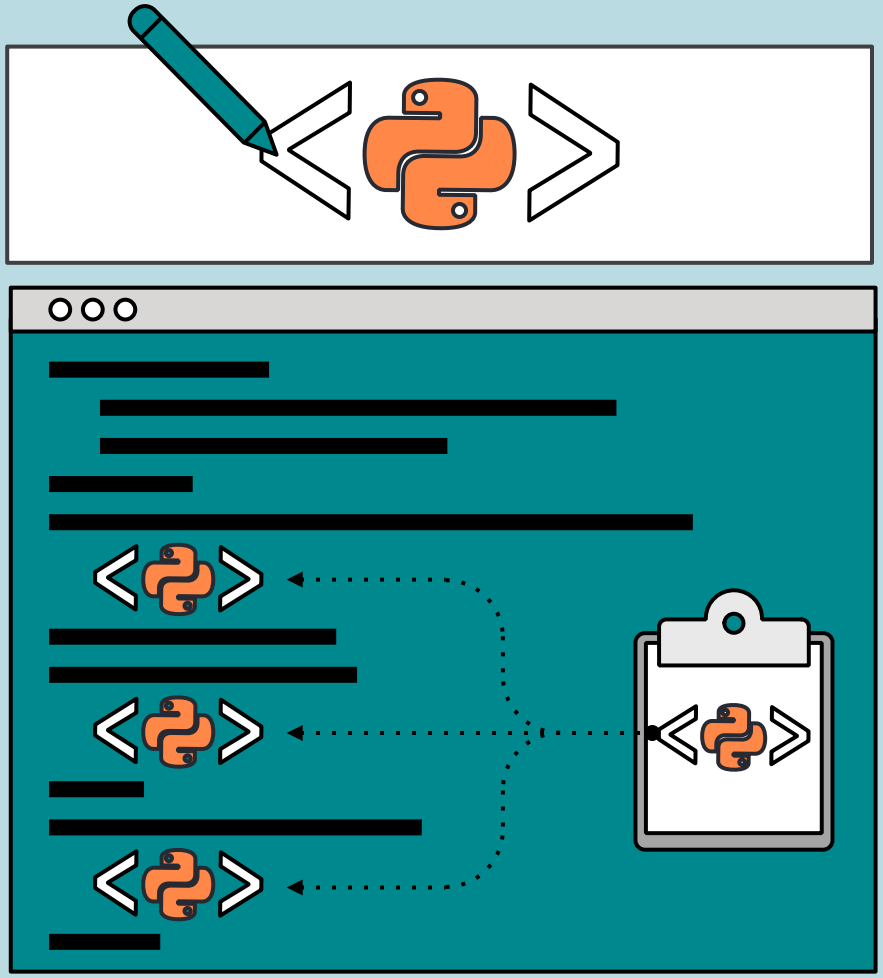
- The bug has been fixed and captured for further testing!

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

Test Fixtures

- A test fixture represents the preparation needed to perform one or more tests, and any associated cleanup actions
  - This may involve creating temporary or proxy databases, directories, or starting a server process

- Fixtures create resources that tests can share and use throughout the test suite
  - This supports the DRY (Don't' Repeat Yourself) principle as it supports defining setup code once inside a **`unittest.TestCase`** class



Test Fixtures: Unit and Class Levels

- Fixtures can exist at the individual unit test level (**called once before *each* test**)
- They can also exist at the class/module level (**called only once before *all* tests** in an individual class/module are run)

UNIT TEST LEVEL

```
class ExampleTestClass(unittest.TestCase):
    # Called before EACH unit test in the class
    def setUp(self):
        self.inputs = ["a", "b", "c"]

    def test_1(self):
        ...
        result = function_under_test(self.inputs)

    def test_2(self):
        ...
```

CLASS LEVEL

```
class ExampleTestClass(unittest.TestCase):
    # Called ONCE before ALL unit tests are executed
    @classmethod
    def setUpClass(cls):
        cls.class_inputs = ["a", "b", "c"]

    def test_1(self):
        ...
        result =
function_under_test(self.class_inputs)

    def test_2(self):
        ...
```

Demo: Test Fixtures

- Suppose the goal for testing negative integers is to use the same set of positive integers used in the **`test_gauss_sum()`** unit test
  - The same **`inputs`** used in **`test_gauss_sum()`** will be used in **`test_gauss_sum_negative_int()`** but negated to ensure they all raise a **`ValueError`**

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    ...
    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]

        for n in inputs:
            with self.subTest(n=n):
                expected = sum(range(n + 1))
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)

    # Post refactoring gauss_sum() to raise Exception
    def test_gauss_sum_negative_int(self):
        n = random.randint(1, 1_000_000) * -1
        expected = f"n must be positive but input is n={n}"

        with self.assertRaises(ValueError) as cm:
            math_tricks.gauss_sum(n)

        self.assertEqual(str(cm.exception), expected)
    ...
```

test_math_tricks.py

Demo:

Test Fixtures cont'd

- If fixtures were not used, the same code to define the inputs would have to be copied into both unit tests
  - What if inputs needed to be updated?
  - Remembering where code was duplicated is error-prone
  - What if the same random value is desired throughout the test suite?

After making these changes, run the test suite via the command **`python -m unittest -v`** to verify the all tests still pass

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    ...
    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]
        for n in inputs:
            with self.subTest(n=n):
                expected = sum(range(n + 1))
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)

    # Post refactoring gauss_sum() to raise Exception
    def test_gauss_sum_negative_int(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]
        for n in inputs:
            n *= -1  # negate original input
            with self.subTest(n=n):
                expected = f"n must be positive but input is n={n}"
                with self.assertRaises(ValueError) as cm:
                    math_tricks.gauss_sum(n)
                self.assertEqual(str(cm.exception), expected)
    ...
```

 test_math_tricks.py

Demo:

Test Fixtures cont'd

- To show how this could affect the test suite, change the **`expected`** values in the unit tests to the right to ensure they fail.

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    ...
    def test_gauss_sum(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]
        for n in inputs:
            with self.subTest(n=n):
                expected = 42
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)

    # Post refactoring gauss_sum() to raise Exception
    def test_gauss_sum_negative_int(self):
        inputs = [5, 100, random.randint(1, 1_000_000)]
        for n in inputs:
            n *= -1  # negate original input
            with self.subTest(n=n):
                expected = "warning!"
                with self.assertRaises(ValueError) as cm:
                    math_tricks.gauss_sum(n)
                self.assertEqual(str(cm.exception), expected)
    ...
```

test_math_tricks.py

Demo: Test Fixtures cont'd

- The same **`inputs`** were used for **`n=5`** and **`n=-5`** as well as **`n=100`** and **`n=-100`**
- The random integer was not carried over because a new random pull was executed for each test (this is not what was intended; the same random integer should be used)

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ...
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ...
test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) ...
======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=5)
----------------------------------------------------------------------
AssertionError: 15.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=100)
----------------------------------------------------------------------
AssertionError: 5050.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=364149)
----------------------------------------------------------------------
AssertionError: 66302429175.0 != 42
```

```
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-5)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-5' != 'warning!'
- n must be positive but input is n=-5
+ warning!
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-100)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-100' != 'warning!'
- n must be positive but input is n=-100
+ warning!
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-580424)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-580424' != 'warning!'
- n must be positive but input is n=-580424
+ warning!
----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=6)
```

Demo: Using `setUp()`

- Instead of repeating the assignment of **`inputs`** for each unit test, use **`setUp()`** to implement a test fixture.
- This will be executed **before each unit test** in the **`TestGaussSum`** class

Note the erroneous expected values are kept to ensure tests will fail. This allows the sub-tests to show the integers used in each test (only shown for test failures)

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    ...
    def setUp(self):
        self.defined_inputs = [5, 100, random.randint(1, 1_000_000)]

    def test_gauss_sum(self):
        # inputs = [5, 100, random.randint(1, 1_000_000)]
        inputs = self.defined_inputs
        for n in inputs:
            with self.subTest(n=n):
                expected = 42
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)

    # Post refactoring gauss_sum() to raise Exception
    def test_gauss_sum_negative_int(self):
        # inputs = [5, 100, random.randint(1, 1_000_000)]
        inputs = self.defined_inputs
        for n in inputs:
            n *= -1  # negate original input
            with self.subTest(n=n):
                ...
```

test_math_tricks.py

Demo: Using `setUp()` cont'd

- After using a test fixture, the **`inputs`** no longer need to be defined in multiple places
- However, the random integer was still not carried over. The same behavior as before is implemented when using **`setUp()`** (a new random pull for each test)

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ...
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ...
test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) ...
======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=5)
----------------------------------------------------------------------
AssertionError: 15.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=100)
----------------------------------------------------------------------
AssertionError: 5050.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=467211)
----------------------------------------------------------------------
AssertionError: 66302429175.0 != 42
```

```
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-5)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-5' != 'warning!'
- n must be positive but input is n=-5
+ warning!
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-100)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-100' != 'warning!'
- n must be positive but input is n=-100
+ warning!
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-60595)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-580424' != 'warning!'
- n must be positive but input is n=-580424
+ warning!
----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=6)
```

Demo: Using `setUpClass()`

- Use a class-level test fixture to ensure the same exact random number is used for the entire **`TestGaussSum`** class.
  - This setup code will be run **once before all the unit tests** in the class
  - Notice the **`@classmethod`** decorator and the use of **`cls`** instead of **`self`** in the definition
  - It's idiomatic in Python to use **`cls`** instead of **`self`** in class methods.
  - **Be careful:** using **`setUpClass`** as it breaks test isolation. Since the fixture is only created once upon instantiation, if one test method mutates the **`defined_inputs`**, that change will leak into proceeding tests (which is probably *not* what you want)

Note the erroneous expected values are kept to ensure tests will fail. This allows the sub-tests to show the integers used in each test (only shown for test failures)

```python
import random
import unittest
import math_tricks


class TestGaussSum(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.defined_inputs = [5, 100, random.randint(1, 1_000_000)]

    def test_gauss_sum(self):
        # inputs = [5, 100, random.randint(1, 1_000_000)]
        inputs = self.defined_inputs
        for n in inputs:
            with self.subTest(n=n):
                expected = 42
                result = math_tricks.gauss_sum(n)
                self.assertEqual(result, expected)

    # Post refactoring gauss_sum() to raise Exception
    def test_gauss_sum_negative_int(self):
        # inputs = [5, 100, random.randint(1, 1_000_000)]
        inputs = self.defined_inputs
        for n in inputs:
            n *= -1  # negate original input
            with self.subTest(n=n):
                ...
```

test_math_tricks.py

Demo: Using `setUp()` cont'd

- The same **`inputs`** were used for **`n=5`** and **`n=-5`** as well as **`n=100`** and **`n=-100`**
- The same random integer was not carried through

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ...
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ...
test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) ...
======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=5)
----------------------------------------------------------------------
AssertionError: 15.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=100)
----------------------------------------------------------------------
AssertionError: 5050.0 != 42

======================================================================
FAIL: test_gauss_sum (tests.test_math_tricks.TestGaussSum) (n=567823)
----------------------------------------------------------------------
AssertionError: 66302429175.0 != 42
```

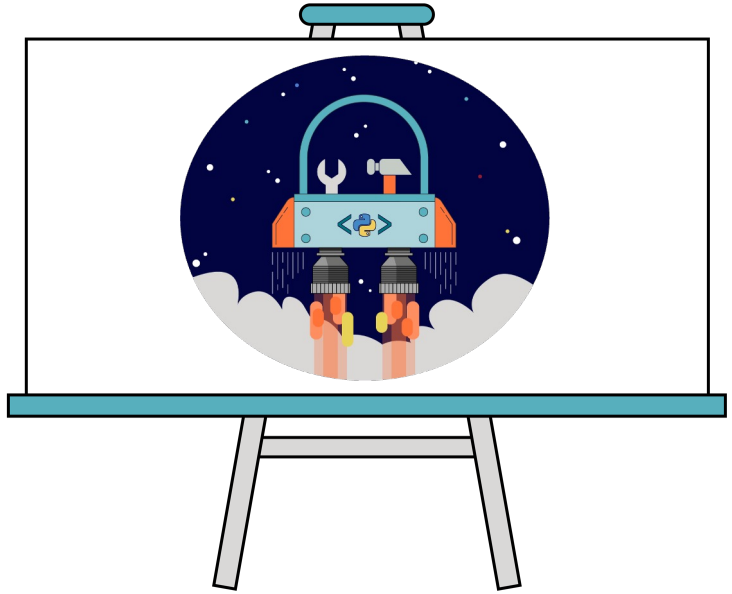
```
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-5)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-5' != 'warning!'
- n must be positive but input is n=-5
+ warning!
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-100)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-100' != 'warning!'
- n must be positive but input is n=-100
+ warning!
======================================================================
FAIL: test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) (n=-567823)
----------------------------------------------------------------------
AssertionError: 'n must be positive but input is n=-580424' != 'warning!'
- n must be positive but input is n=-580424
+ warning!
----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=6)
```

Update Source Code for Passing Tests

- Once we replace the erroneous expected values, all unit tests pass

```
expected = sum(range(n + 1))
expected = f"n must be positive but input is n={n}"
```

```
$ python -m unittest -v
test_gauss_sum (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_0 (tests.test_math_tricks.TestGaussSum) ... ok
test_gauss_sum_negative_int (tests.test_math_tricks.TestGaussSum) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.008s

OK
```

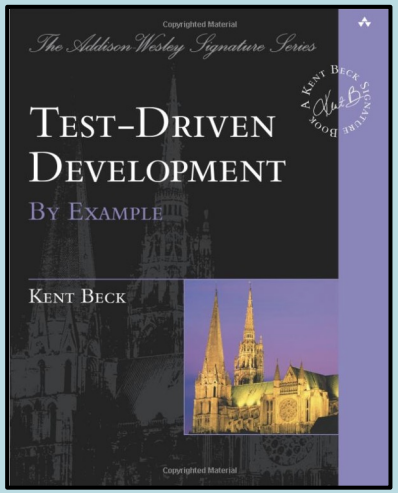


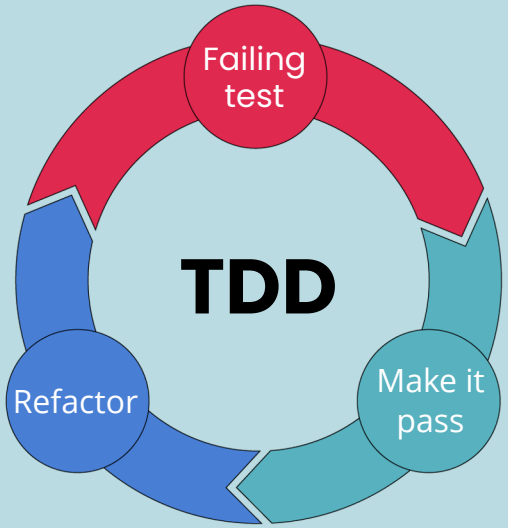
Additional Resources

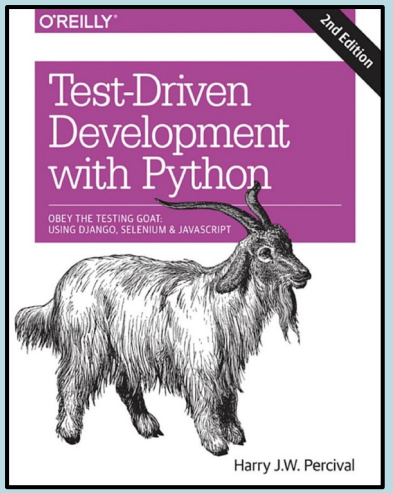
Lecture 07
Unit Testing

Test Driven Development (TDD)

- Approach to writing software where test cases are written *prior* to source code
- Add a test, run all tests (new tests should fail), write simplest code that passes new test, run all tests (all tests should pass), refactor as needed (running tests after each refactor to ensure functionality is preserved), repeat...
- "Red-Green-Refactoring" (red means fail, green means pass)







Test Driven Development (TDD) Example

Write Tests

```python
import unittest
import utils


class TestUtils(unittest.TestCase):
    def test_get_filetype_py(self):
        filepath = 'example.py'
        expected = '.py'
        result = utils.get_filetype(filepath)
        self.assertEqual(result, expected)


if __name__ == "__main__":
    unittest.main()
```

test_utils.py

```
$ python -m unittest
F
```

Write Code

```python
from pathlib import Path


def get_filetype(filepath):
    """Returns the suffix of a given file."""
    filepath = Path(filepath)
    return filepath.suffix
```

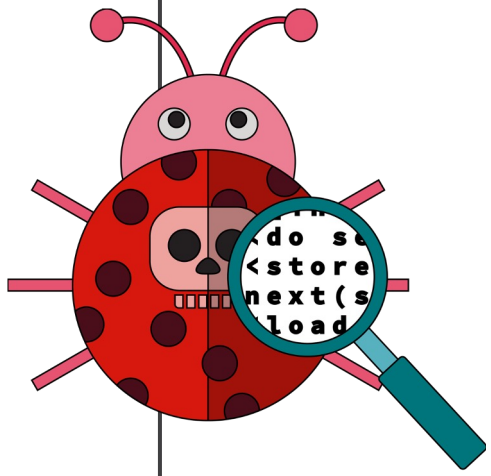
utils.py

```
$ python -m unittest
.
----------------------
Ran 1 test in 0.000s

OK
```

Documentation Testing with `doctest`

- Python has a built-in library for checking module, class, and function **docstrings**
- Searches for pieces of text that look like interactive Python sessions (**>>>**), and then executes those sessions to verify they work exactly as shown

```
def my_function(x):
    """
    Returns the square root of x.

    Examples
    --------
    >>> my_function(25)
    5.0

    >>> my_function(100)
    10.0
    """
    return x ** 0.5
```



utils.py

Documentation Testing with doctest

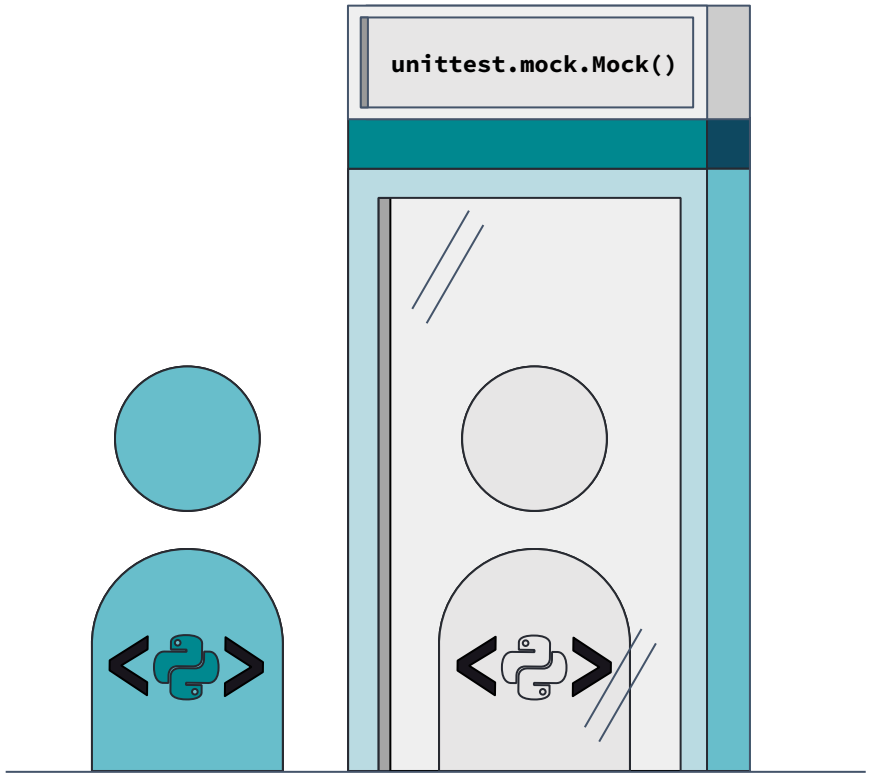
```python
from pathlib import Path

def get_filetype(filepath):
    """
    Returns the suffix of a given file.

    Examples
    --------
    >>> get_filetype('example.py')
    'py'

    >>> get_filetype('presentation.pptx')
    '.pptx'
    """
    filepath = Path(filepath)
    return filepath.suffix
```

utils.py

```
$ python -m doctest -v utils.py
Trying:
    get_filetype('example.py')
Expecting:
    'py'
**********************************************************************
File "/my_project2/utils.py", line 13, in utils.get_filetype
Failed example:
    get_filetype('example.py')
Expected:
    'py'
Got:
    '.py'
Trying:
    get_filetype('presentation.pptx')
Expecting:
    '.pptx'
ok
1 items had no tests:
    utils
**********************************************************************
1 items had failures:
   1 of   2 in utils.get_filetype
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Mocking with Unit Tests

- Python's **`unittest`** library ships with a **`mock`** module
- This supports replacing parts of a program with *mocked* objects
  - Can mime functionality of source code without implementing the *actual* source code function / method
  - Core components are **`unittest.mock.Mock()`** class and **`unittest.mock.patch()`** decorator
  - Helpful when testing production database queries or HTTP GET requests



Mocking Example I

```def get_header(filepath):
    """Returns the first line of a file."""
    with open(filepath, mode="r", encoding="utf-8") as fp:
        header = fp.readline()
    return header```

utils.py

```import unittest
import unittest.mock
import utils


class TestUtils(unittest.TestCase):
    def test_get_header(self):
        expected = "Temp File Header"
        mocked_open = unittest.mock.mock_open(read_data="Temp File Header")

        with unittest.mock.patch(target="builtins.open", new=mocked_open) as m:
            result = utils.get_header("temp_file")

        m.assert_called_once_with("temp_file", mode="r", encoding="utf-8")
        self.assertEqual(result, expected)```

test_utils.py



# Mocking Example II

```python
def google_query(query):
    """
    Performs a google search for provided query string.
    Notes
    -----
    This function does a GET request against google and
    returns a list of (title, link url) pairs.
    """
    url = "https://www.google.com/search"
    params = {"q": query}
    resp = requests.get(url, params=params)

    soup = bs4.BeautifulSoup(resp.content, "html.parser")
    links = soup.find_all("a")

    title_link_pairs = [
        (link.find("h3").getText(),
         link.get("href").split("?q=")[1].split("&sa=U")[0])
        for link in links
        if link.get("href") and link.find("h3")
    ]
    return title_link_pairs
```

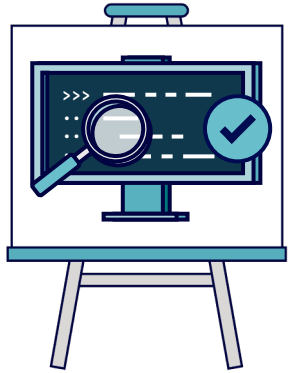
utils.py

```python
import requests
import unittest
import unittest.mock
import utils


class TestUtils(unittest.TestCase):
    ...

    @unittest.mock.patch("requests.get")
    def test_google_query(self, mock_get):
        mock_response = unittest.mock.Mock()
        mock_response.content = (
            '<a href="/url?q=https://www.fakeurl-test.com&sa=Uabcdefg">'
            + "<h3>Test Title from Mock</h3></a>"
        )
        expected = [("Test Title from Mock", "https://www.fakeurl-test.com")]

        mock_get.return_value = mock_response
        result = utils.google_query("Enthought Blog")

        mock_get.assert_called_once()
        self.assertEqual(result, expected)
   ...
```

test_utils.py

Resources

- Simple Smalltalk Testing: With Patterns
  - Kent Beck's original paper on testing frameworks using the pattern shared by unittest.
- pytest
  - Third-party unittest framework with a lighter-weight syntax for writing tests. For example, assert func(10) == 42.
- The Python Testing Tools Taxonomy
  - An extensive list of Python testing tools including functional testing frameworks and mock object libraries.
- Testing in Python Mailing List
  - A special-interest-group for discussion of testing, and testing tools, in Python.



Lecture 09

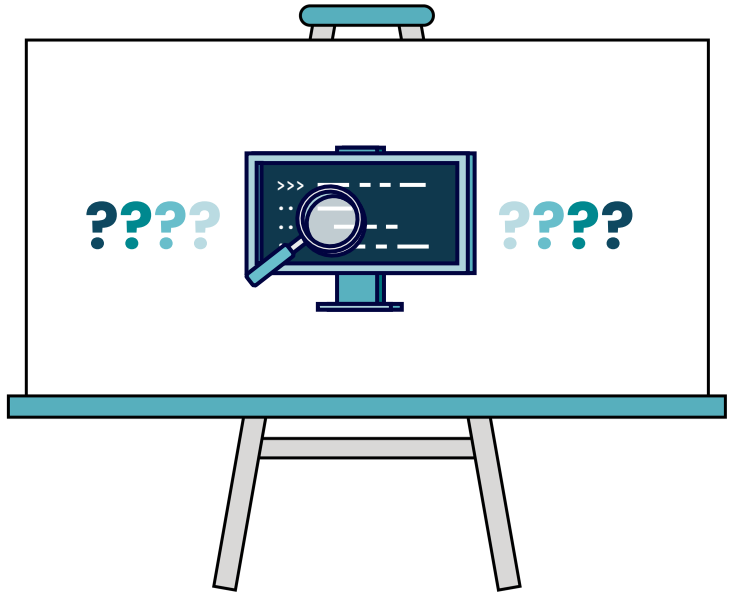
Effective Code Reviews

Software Engineering
for Scientists and Engineers

Discussion

Does your team currently use code reviews?

Table of Contents



Why Code Reviews?

Lecture 09
Effective Code Reviews

Code Quality

When working with a team of developers with a range of experience and skill levels, **code reviews** are one of the best ways to improve code quality.

They do this by:

- Ensuring that more than one person has **thought** about the changes being made. In particular, the second person (the reviewer) has a chance to make sure that someone other than the original developer can understand the changes.

- Ensuring that more than one person has **tested** the changes and found them effective.
  - **Note**: this does not replace systematic and automated unit testing. Unit testing is also code; code reviews add human eyes and brains to the check cycle.

Time and Expense

Code reviews, when done well, add a significant burden of time to the development process. This makes projects more expensive.

However, consider:

- By improving code quality, fewer mistakes are made and code works better. You should have less rework that needs to be done due to a single developer not catching something.
- As a project manager, you have control of the time spent doing code reviews. You can adjust their rigor to fit the project, budget, and numbers of issues coming back.

Tooling to Reduce Code Review Burden

Effective use of tools like **flake8**, **black**, and **isort** (and others) can reduce the burden of code reviews. These integrations generally allow code reviewers to focus on matters of substance, not formatting.

- **Integrated with IDE** : When these kinds of tools are integrated in each developer's IDE with standard configurations, they catch many issues of formatting and style (and even bugs) before checkins occur.
- **Integrated with Source Control** : Integrated these tools with source control adds another layer of enforcement of formatting and style during checkin.

Mentorship

An under-appreciated benefit of code reviews are the mentorship possibilities.

Earlier we divided developers into apprentices, journeymen, and masters.
If you make the effort to pair up developers of different experience levels (and/or background), each code review accelerates learning about development.

For instance:

- **Master reviewing Apprentice**: Apprentice gets feedback on code from a more experienced person.
- **Apprentice reviewing Master**: Apprentice gets to see "good" code, see examples of how issues of greater complexity are handled.

Communication & Collaboration

Code reviews generally increase the level of communication on your team – whether through questions, sharing knowledge, or helping each other find and fix mistakes.

Individual developers who learn to **give** and **receive** effective (and non-combative) code reviews get good experience collaborating with others on teams.

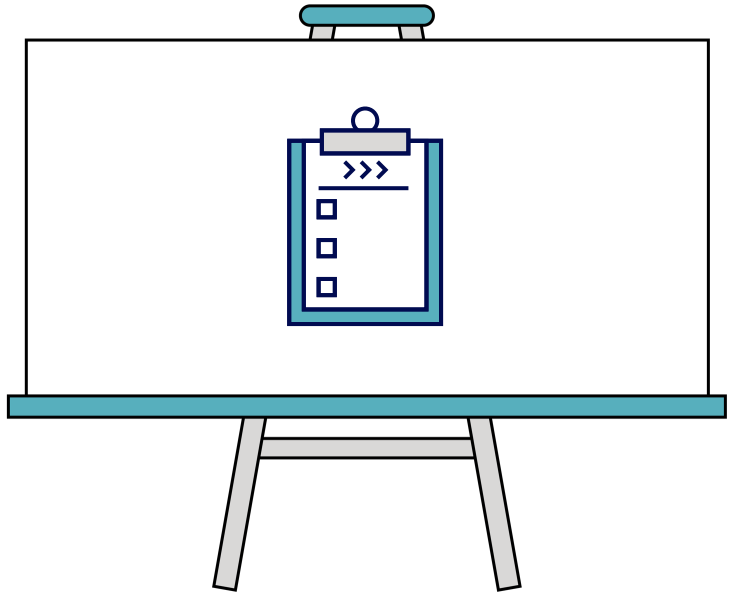
- They write better issues when describing problems.
- They write better design documents when sketching out a system.
- They learn to collaborate by leveraging each other's knowledge and experience.

Do Code Reviews

If you are still not convinced, please take the time to watch:

"How to Shut Down Tolkien"
Brandon Rhodes
PyGotham 2014

https://www.youtube.com/watch?v=qVlqBxpCG24



Code Review Checklists

Lecture 09
Effective Code Reviews

GitHub Context and Vocabulary I

We are going to talk about code reviews in the context of GitHub. For that we need some vocabulary.

- **Issue Tracker**: GitHub provides users with an issue tracker that allows a development team to track issues (bugs), feature requests, and other potential changes to the code.
- **Branch**: In response to an issue, a developer makes a branch and works on the code, checking in changes as needed.
- **Pull Request (PR)**: When the developer is done making changes, a pull request is made for the branch. At this point a reviewer is requested (or assigned).

GitHub Context and Vocabulary II

- **Code Review**:
  - The reviewer checks out the branch on which the PR was developed.
  - The reviewer reads any code changes, tests the code, and generally determines whether the changes fix the issue(s) involved and is ready to go.
  - If not, the reviewer requests changes and the original developer returns to working on the code; when done a check in is made and a new review is requested.

- **Merge**: If the changes are approved, the branch is merged into the main branch (often called 'dev', 'trunk', or 'main') and the development branch is deleted; the PR and issues are closed.

Personal Guidelines for Developers and Reviewers

DEVELOPER GUIDELINES

Make it easy for someone to review your code:

- **Simple PRs** : Make each PR as simple as possible (one change, one PR). Avoid free-rider changes.
- **Use Tools** : Integrate standard linting tools into your IDE with standard setups.
- **Describe Issues** : In PR comments, make sure you discuss any issues or subtleties about your fix. Make it easy for the reviewer to understand what you did and why.

REVIEW GUIDELINES

Make reviews useful, helpful, and nice:

- **Focus on Code** : Review the code, not the developer. Make sure comments are about collaboratively improving the code.
- **Share Knowledge** : If you have a better way, suggest it and explain why it is a better way. Remember that code reviews are about both learning and teaching.
- **Resist LGTM** : For simple issues it is easy to review as Looks Good to Me and approve. Resist this as your default.

General Checklist

- Does the PR have a **meaningful title** and an **adequate description**, so that future visitors can understand what changed?
- Does the PR description **reference related issues**?
- Do I understand the **purpose** of the PR?
- Does the PR **achieve its purpose**? For example, does new code behave as intended in manual testing?
- Do I understand the **mechanism** by which the PR achieves this purpose?

Correctness Checklist

- Are project **style guidelines** being followed?
- Is new code introduced in the PR **consistent** with existing similar code in the codebase?
- If an API has been changed, have all **uses of the API** been updated accordingly?
- Do **external users** of changed functionality have an appropriate **upgrade path** to avoid breakage? Is that upgrade path communicated effectively to users?
- If there are TODO/FIXME/XXX comments introduced, do those comments include a reference to a **corresponding issue** in the issue tracker?

Design Checklist

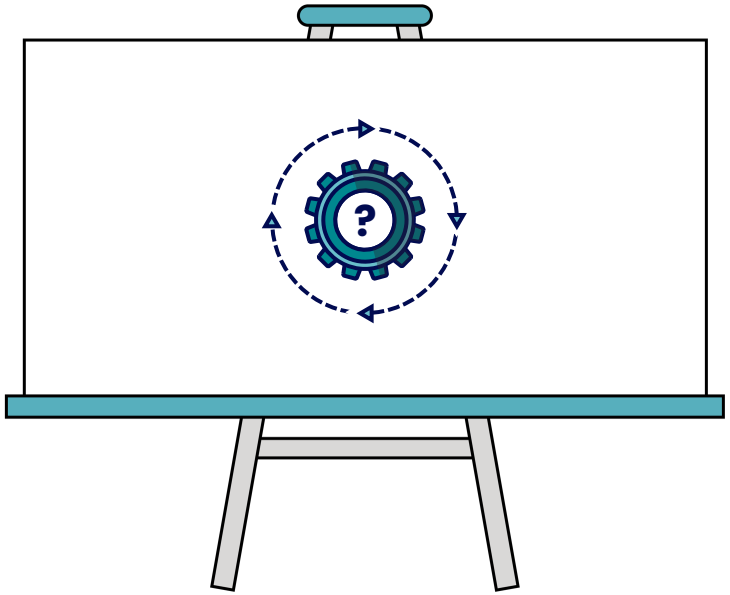
- Are **design decisions** clearly documented?
- If new APIs have been introduced, are those APIs **easy to use** correctly?
- Does the **larger design** make sense? (Hard!)
- Have **larger changes** been subject to the appropriate level of scrutiny? Should the team request input from technology teams or from other experts?
- If the PR introduces user-visible functionality, does the new UI/UX **make sense for users**?
- Should the team **request user feedback** on the proposed changes?

Testing Checklist

- Does the **test suite pass**? (This will typically be answered by the continuous integration setup on the project, but it can be helpful to run the test suite manually as an extra check.)
- Are there **new warnings**, or other **unexpected output**, in the test log?
- Are **bug fixes** accompanied by a regression test? Does that test fail in the expected manner without the bug fix?
- Is any **new functionality** introduced by the PR well covered by the unit tests?
- Are there new **integration tests**, if appropriate?

Documentation Checklist

- If an API has been changed, has documentation (docstrings, developer and user documentation, tutorials, etc.) been **updated** to match?
- Is **new functionality** sufficiently well documented, at all appropriate levels?
- Is any added or changed documentation **clear** and **useful**?
- For major changes, are the **design decisions** leading to those changes adequately documented?
- If there are documentation changes, does the documentation still **build cleanly**?



Also Consider

Lecture 09
Effective Code Reviews

Code Review Standards

Each organization (or project) should have a written code review standard that all team members consult and follow during code reviews.

Build that standard from the **Code Review Checklists** given in the previous slides.
Add in ideas from other sources and your own experience.

Then look at the standard and think about:

- How much time does it take to do a typical review?
- What benefit in quality do I expect to gain?
- What benefit in mentorship and knowledge sharing do I expect to gain?

After assessing the trade-offs, if needed, trim down your code review standards to fit the benefits.

Solo Code Reviews

It is also possible to implement code reviews when working alone on a small, but longer-term project.

- Use source control and the **Issue-Branch-PR-Review-Merge** workflow.
- During the review, use the **Code Review Checklists** (modified to fit your needs) to systematically ensure that everything you want reviewed is.
- It can also be useful to make the PR one day and let a day or more pass before reviewing.

Code Review Resources Online

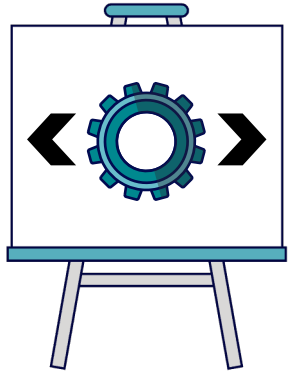
- How to Do Code Reviews Like a Human (Part One), by Mike Lynch: https://mtlynch.io/human-code-reviews-1/
- How to Do Code Reviews Like a Human (Part Two), by Mike Lynch: https://mtlynch.io/human-code-reviews-2/
- How to Make Your Code Reviewer Fall in Love with You, by Mike Lynch: https://mtlynch.io/code-review-love/

Exercise: Refactoring

Earlier you worked on the refactor inventory problem. Revisit it.

Using the **Code Review Checklists**, do a solo code review of your solution.

- Refactor Inventory



Lecture 10

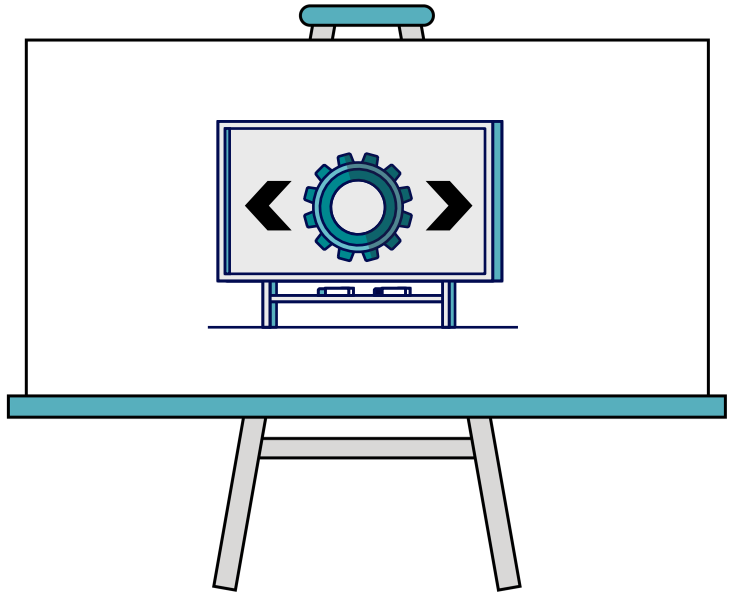
Development Models

Software Engineering
for Scientists and Engineers

Discussion

What is your software planning process?

Table of Contents



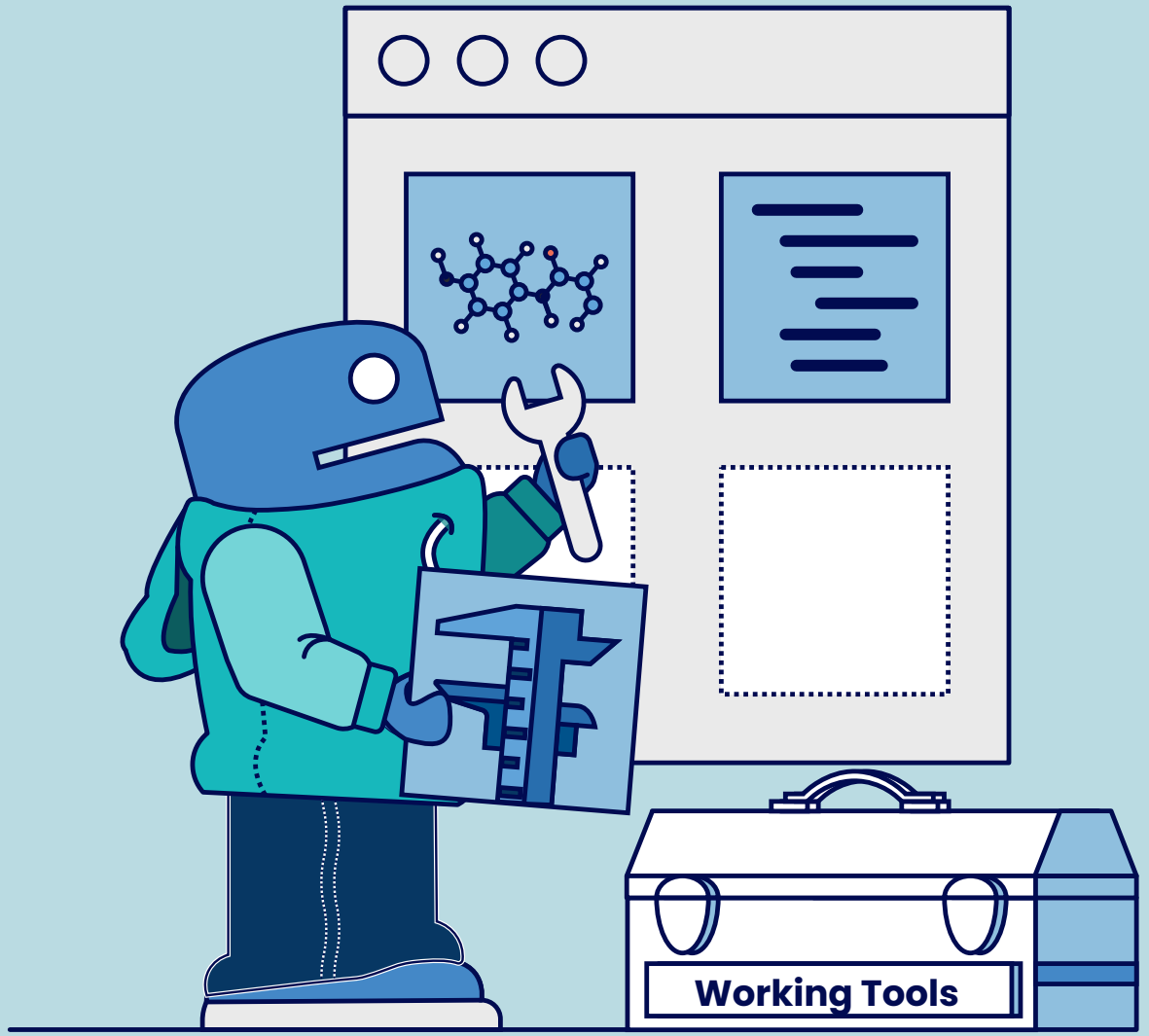
Introduction

Lecture 10
Development Models

Software Engineering: A Working Definition

Software engineering is a systematic approach to software development with the following characteristics:

- Draws on well-known patterns (best practices) from past software projects.
- Continuously improves software by learning from past mistakes (has a strong feedback loop).
- Has a standard set of working tools that all practitioners should know and use.



Planning and Software Engineering

Since software engineering is a "systematic approach" to developing software, it involves planning. Like all engineering disciplines that planning can get very detailed. In the case of software engineering for research projects, the level of planning that you should use depends on your problem domain.

- **Cutting-Edge R&D** – If you are working at the cutting edge, algorithms and systems may be constantly refined. Here, software is more akin to *device engineering*, where you have to design each and every component with few off-the-shelf parts.

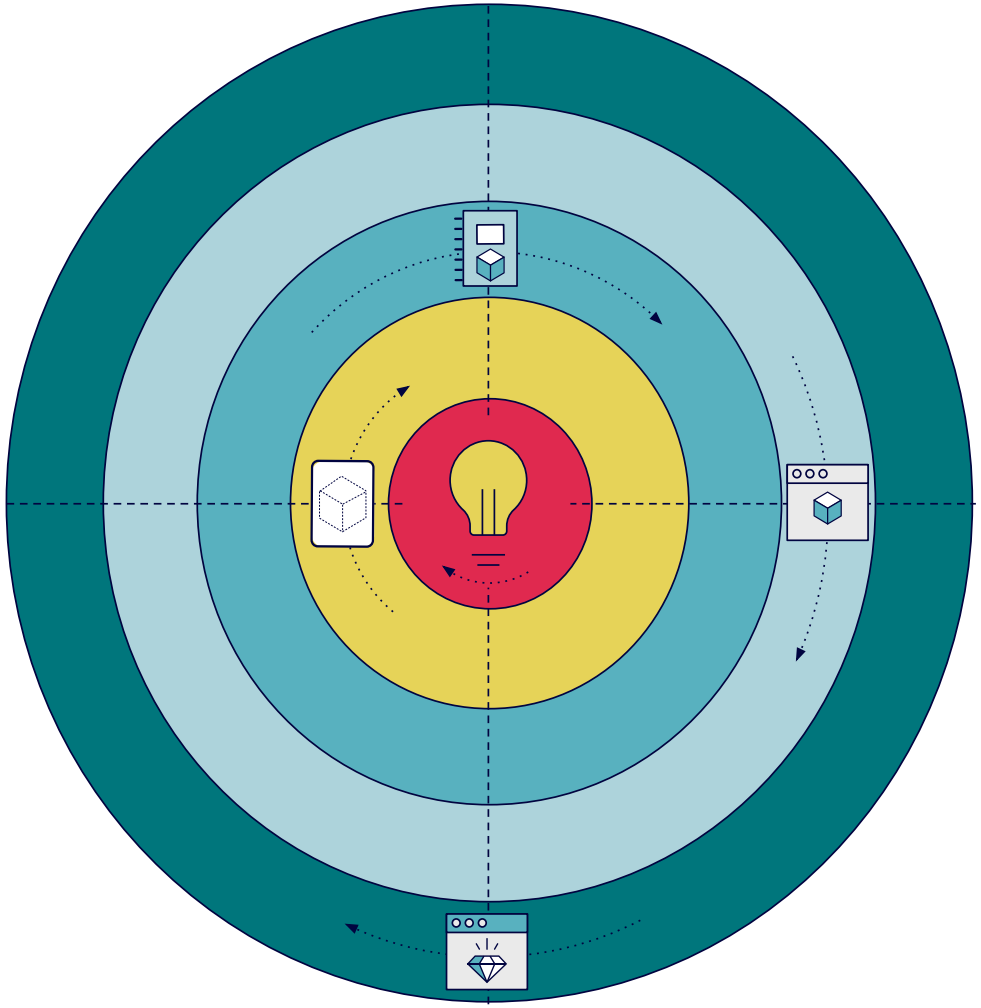
- **Established Science** – If the science (or engineering) that you are working with is well-understood, the algorithms and systems that you will be using are often well-defined. Here, software is more akin to *systems engineering*, where you are mostly assembling standard components in a new way.

Cutting-Edge R&D Software Engineering Cycle

Software Idea → Code Experiment
Code Experiment → Jupyter Notebook
Jupyter Notebook → Script
Script → Application

Since the problems are not well-known, **experimentation** leads the way:

- planning done in on each cycle over short timeframes
- growing personnel requirements
- often too open-ended for many institutions and managers

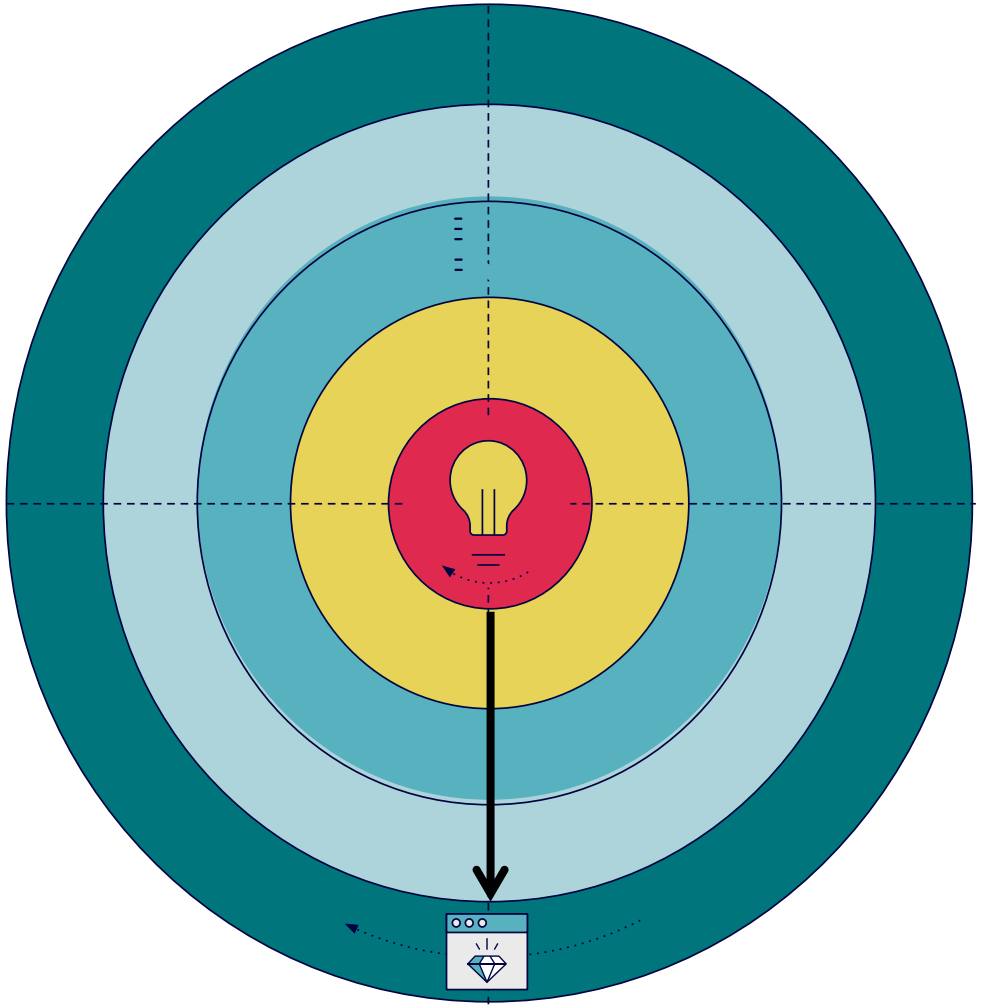


Established Science Software Engineering Cycle

Software Idea → Planning
Planning → Application

Since the problems are well-known, can plan directly from idea to application and skip the many cycles:

- experimentation is replaced by **detailed planning**
- well-defined time and personnel requirements
- well-liked by institutions and managers



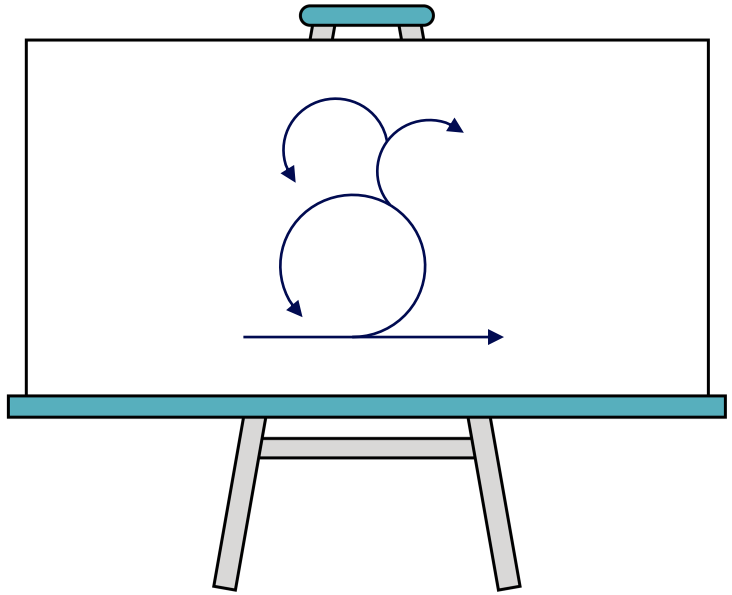
Two Approaches are Opposite Ends of a Spectrum

The two approaches outlined above are at the opposite ends of a spectrum. Most real projects fall somewhere in between, with many R&D software projects falling somewhat closer to the cutting-edge approach.

Both approaches have their merits. Let's examine each one in more detail:

- **Cutting-Edge R&D** – This approach tends toward what software developers call the **Agile** development model.

- **Established Science** – This approach tends toward what software developers call the **Waterfall** development model.



Agile

Lecture 10
Development Models

What is Agile?

Many software development firms describe themselves as – and pride themselves on being – **agile**. What does this mean?

- **Short Development Cycles** – The timescale of development is typically measured in weeks, sometimes as few as two or three. This includes planning, the actual development, testing, and release.
- **Emphasis on Continual Feedback** – Each development cycle ends with a testing phase that often includes the final customer.
- **Emphasis on Adaptability** – Because of the short development cycles and continual feedback, the goals are easy to adapt to changing customer needs.
- **Risk Management** – Manages risk by allowing for change.

An Agile "Sprint"

Each development cycle is typically called a **sprint**.

The sprint is broken up into the following phases:

- **Planning** – Sprint planning is **closely coordinated with customers**. The idea is to choose a set of development tasks to accomplish that the customer sees as the most important to get done.
  - Each feature gets a **user story** that defines what the customer expects. This is written (and diagrammed) in terms that the customer understands.
  - Some features are expected to be complete after one sprint; some may be planned to cover several. Ideally, more complex features are broken up into a group of sprint-sized tasks that can be done individually.
  - Often the **final goal is fuzzy** and only emerges after several sprints are done. For customers and developers this feels like a bottom-up approach.

An Agile "Sprint" cont'd

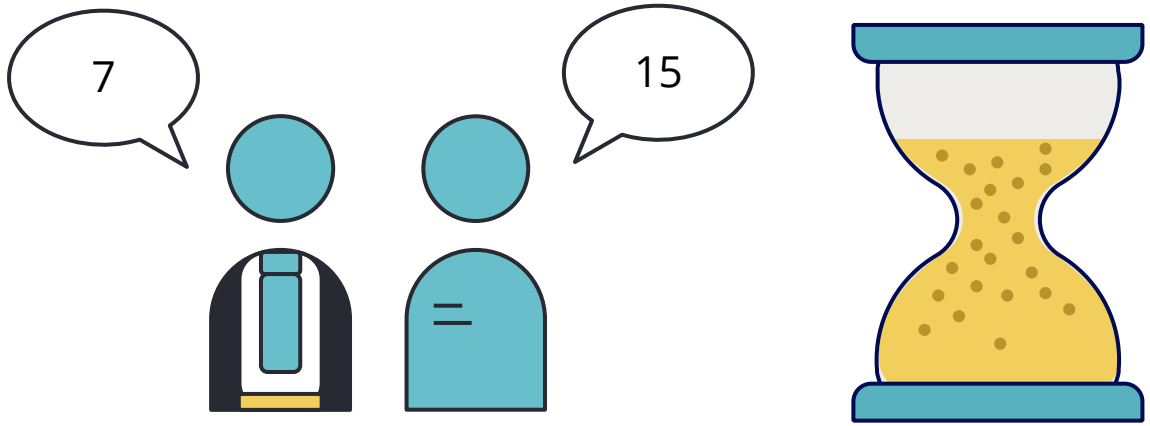
- **Development** – The programmers work on their assigned features during this portion of the sprint. At least weekly, the team working on the sprint meets to discuss progress, resolve roadblocks, and coordinate inter-dependencies.
  - Weekly meetings are typically very **short**. Each developer gets five minutes to state progress. More detailed discussions, if needed happen afterwards or in individual conversations post-meeting.
  - Code reviews are required.

- **Testing** – Throughout development, continuous integration, unit testing, and other forms of testing are run.
  - Don't break existing features and code.
  - Ensure that features to be completed this sprint are working.

Task Estimation in Agile

Estimating effort and value are an important part of Agile development

- **Humans are both bad and good at estimating**
  - Bad at estimating absolute values, large values, unfamiliar things
  - Good at estimating relative values within an order-of-magnitude of something familiar with



https://www.amazon.com/Agile-Estimating-Planning-Mike-Cohn/dp/0131479415

https://www.scruminc.com/calculating-business-value/

Task Estimation in Agile cont'd

Many techniques exist for estimating effort and value

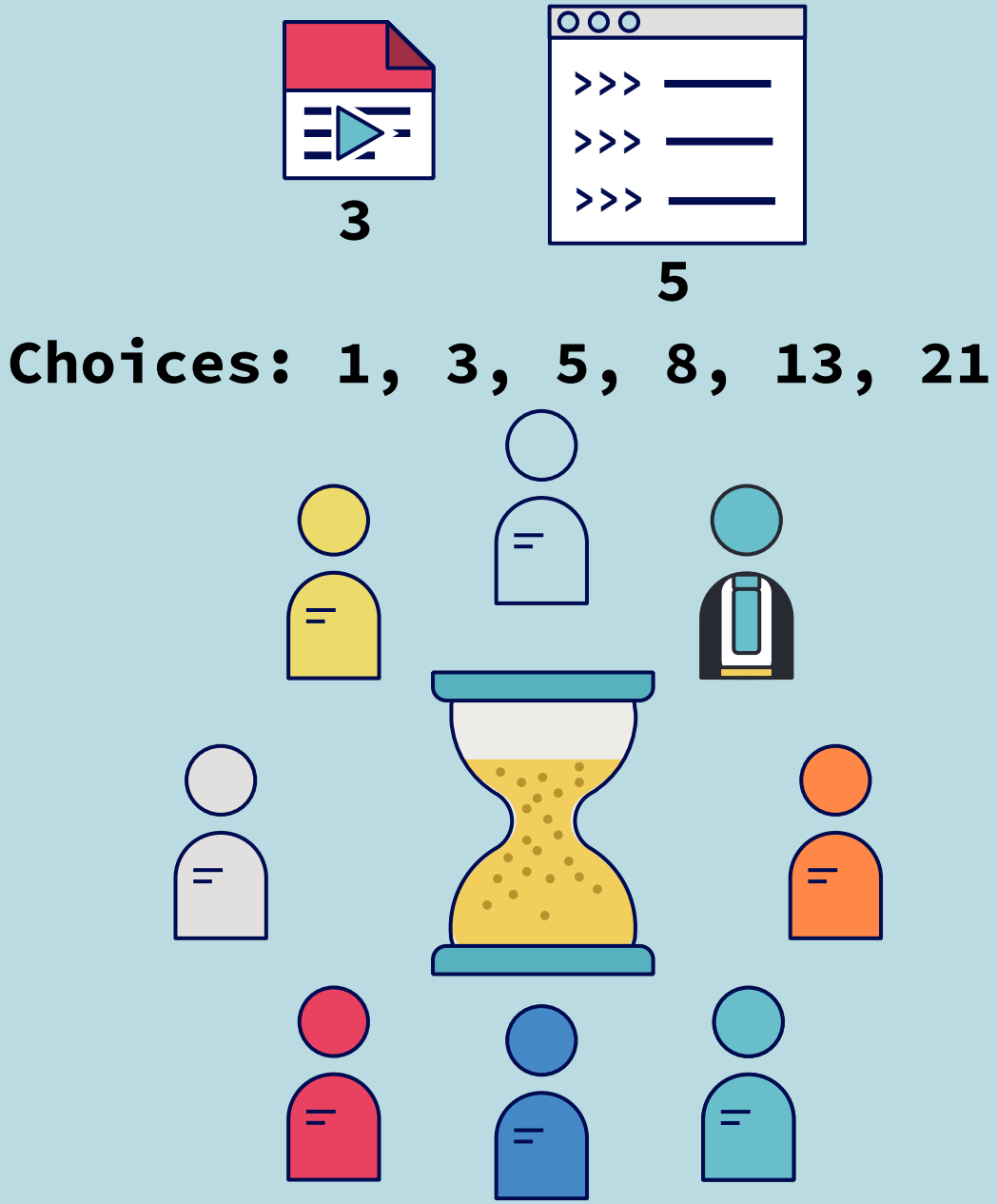
- Estimating on standard scale
- **Voting (Planning Poker)**
- Ordering
- Swim-lanes

Key principles for estimations

- **Relative** to similar, previously completed tasks
- **Collective** by nature (focused on those doing the work)
- **Measured** by effort or ideal time (points system)
- **Small** values for easy comparison
- **Refined** when new information becomes available
- **Tracked for future** using "velocity"

Planning Poker: Set up

- Work to be estimated is described in work units (tasks, features, stories…)
- One work unit is chosen as the baseline (baseline = X points)
  - Can be helpful to triangulate by establishing a baseline of two values (past projects)
- Estimators and work owner all participate
- Every participant has cards of available estimation points (powers of 2, Fibonacci numbers, etc.)



Example Agile Task Estimates

Points	Feature	Maintenance	Bug	Communication	Meta
1	Have the program print "Hello!" when it starts up	Add a copyright header to all files	Fix a typo	Totally clear from the start	A warm summer breeze
2	Add an option for sorting by date	Refactor to pass "timestamp" to each method instead of having a class attribute	Timestamps aren't respecting time zone when sorting	Read issue, maybe post a question	Playing tic-tac-toe
3	Restrict login to accounts created before a certain date	This module is kind of a mess; go through and sort out the interfaces; also fix the tests	Uploading with a tag value containing Japanese characters results in a corrupted filename	Discuss on GitHub, bring it up in the standup meeting, and ask a few questions	Playing checkers
5	Add audit logging to all API methods	Port the file reading code from Python 2 to Python 3. Also, fix the tests for real this time	Long-running file transfers occasional break with "connection reset by peer"	Reach out to product owner to understand the scope of the issue	Playing chess
8	Enable SSL on both the server and the client, figure out compatibility	Switch all the model objects from attributes/properties to Traits; figure out how to use traits futures	The customer updated their "security proxy" and now some of the web dashboard buttons don't work	Discussions on GitHub, decide we need a whiteboarding session	Flying an airplane
13	Add Japanese as a language option across the web UI, desktop, logs, etc.	Audit the SQL layer so it works correctly on both Postgres and MySQL; add a lot of tests for this	Active Directory login works in testing but fails on the customer side with "The parameter is incorrect"	Discuss, prototype, and ask product owner to test a fix and deliver to customer	Flying an airplane, upside down
21	Add ability to log in to the app via Google OAuth and figure out what OAuth is	Get the C++ part of the fluid simulator working on Windows; test for Windows 7 and Vista	On server restart, the Docker volume containing customer database files is empty, sometimes	Both product owner and customer help debug the issue and test a possible fix	Flying an airplane, on fire

Planning Poker: Process

- Moderator picks a work unit
- Owner describes the unit
- Team simultaneously votes (one card; relative to baseline)
- Large estimates require refinement of task
- Outliers defend their estimates
- Team votes again
- Iterate until consensus or decree

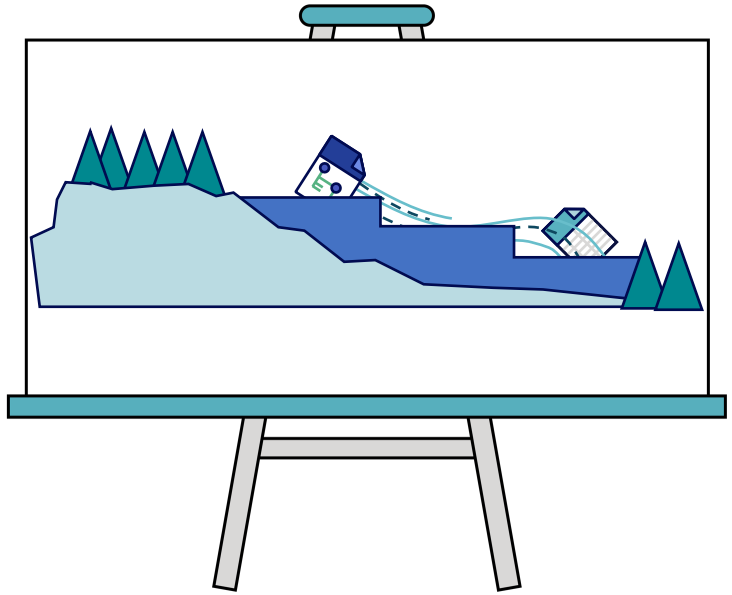
Give it a try! Planning Poker

Use the below work unit to produce your own estimation of effort. Choices are 1, 3, 5, 8, 13, 21.

Do not share your estimate until the moderator prompts you.

> Your team has inherited a relatively small code base (~25 functions total) that has 0 unit tests. A new release has been recently delivered to your clients, so new feature requests are not likely to come in (it's a good time to add tests). **The unit of work to be estimated is adding unit tests to the code base**.
>
> The code base isn't too complex but does require pulling from a production database and submitting a GET request to third-party API. You will likely need to mock these interactions.

1. When prompted, write your estimate in the chat so that votes are simultaneous
2. Large estimates require refinement of task
3. Outliers defend their estimates
4. Team votes again
5. Iterate until consensus or decree



Waterfall

Lecture 10
Development Models

What is Waterfall?

Other software development firms come out of a more engineering background and see agile development as haphazard and unpredictable. These prefer a more fully planned waterfall development style.

- **Long Development Cycles** – The timescale of development is typically measured in months, sometimes years. This includes planning, the actual development, and testing.
- **Emphasis on Planning** – Plans are very detailed, complete with resources required, inter-dependencies, and timelines thought out in advance.
- **Emphasis on Predictability** – A project manager is assigned to ensure that the plan is kept on track, shifting resources and timelines in order to meet the end goal.
- **Risk Management** – Manages risk by planning for all eventualities.

A Waterfall Stage I

Like agile, development is broken into stages of development, with various phases:

- **Planning** – Waterfall stage planning is a major effort that brings together all stake-holders (customers, management, senior developers) to negotiate a detailed plan for development.
  - Each feature being advocated by one group is written up with resource, timelines, and potential impact considered.
  - This often involves several meetings over a pre-defined period of time (weeks or months) as discussions continue.

A Waterfall Stage II

- **Development** – This part of the stage is basically the same as agile, however the features are generally bigger and more complex.
  - There is more time to work on any given feature.
  - Code reviews are still important.

- **Testing** – Testing is generally more involved in the waterfall development model.
  - Continuous integration and unit testing are still useful tools.
  - Additional testing modes like integration testing, user-interface testing, system staging, and user acceptance testing are often added.
  - Sometimes there is a team (QA) devoted solely to testing.

Task Estimation in Waterfall

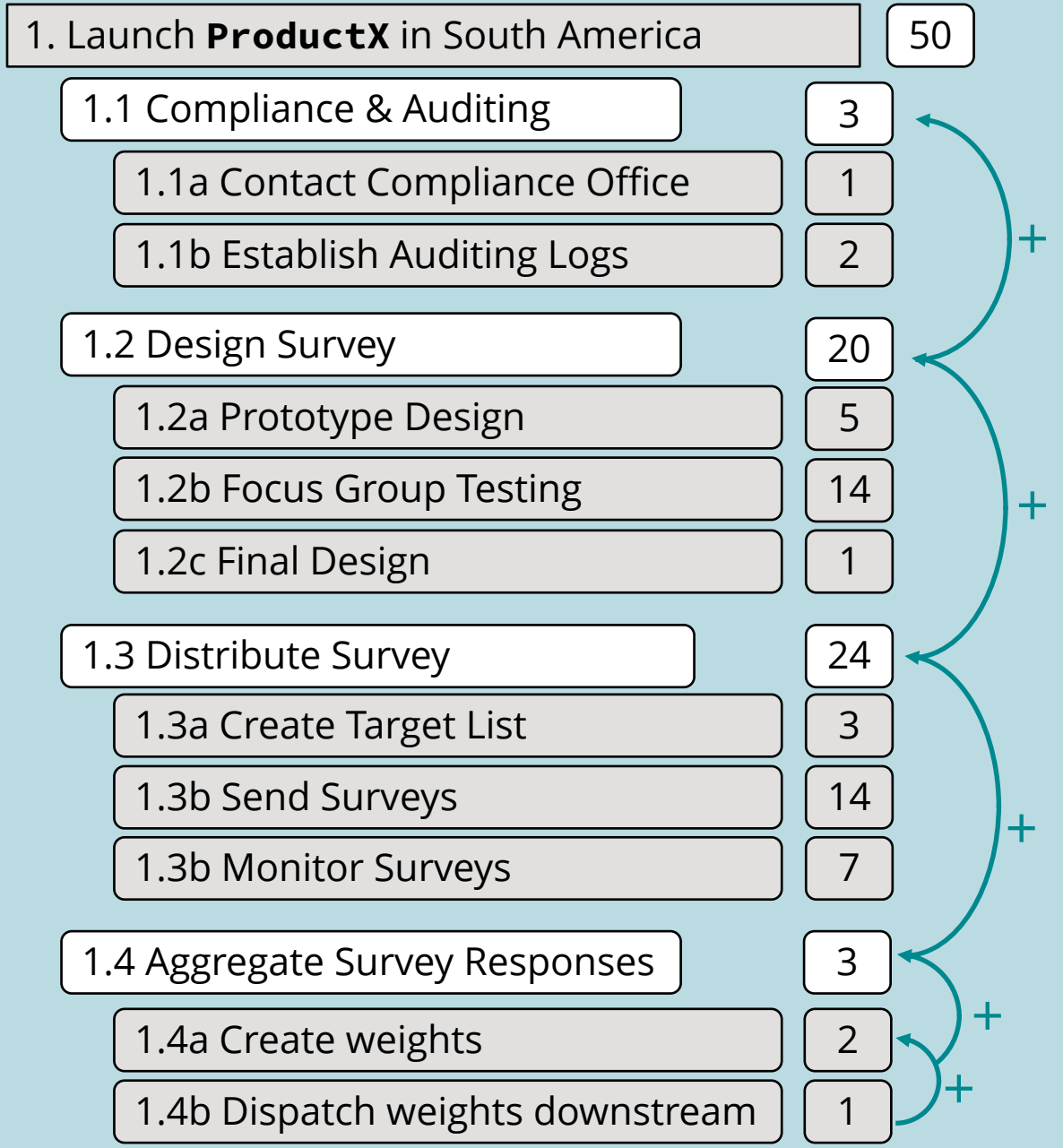
Estimating effort and value involves a more wholistic and pre-defined task view in Waterfall

- Waterfall timelines and budgets are often set from the start
- Client interaction is minimal
- Highest Priority is to deliver the end product that matches initial requirements
- Enabled by extensive upfront estimation and planning

- Two main approaches
  - **Bottom-Up Approach**
  - **Top-Down Approach**

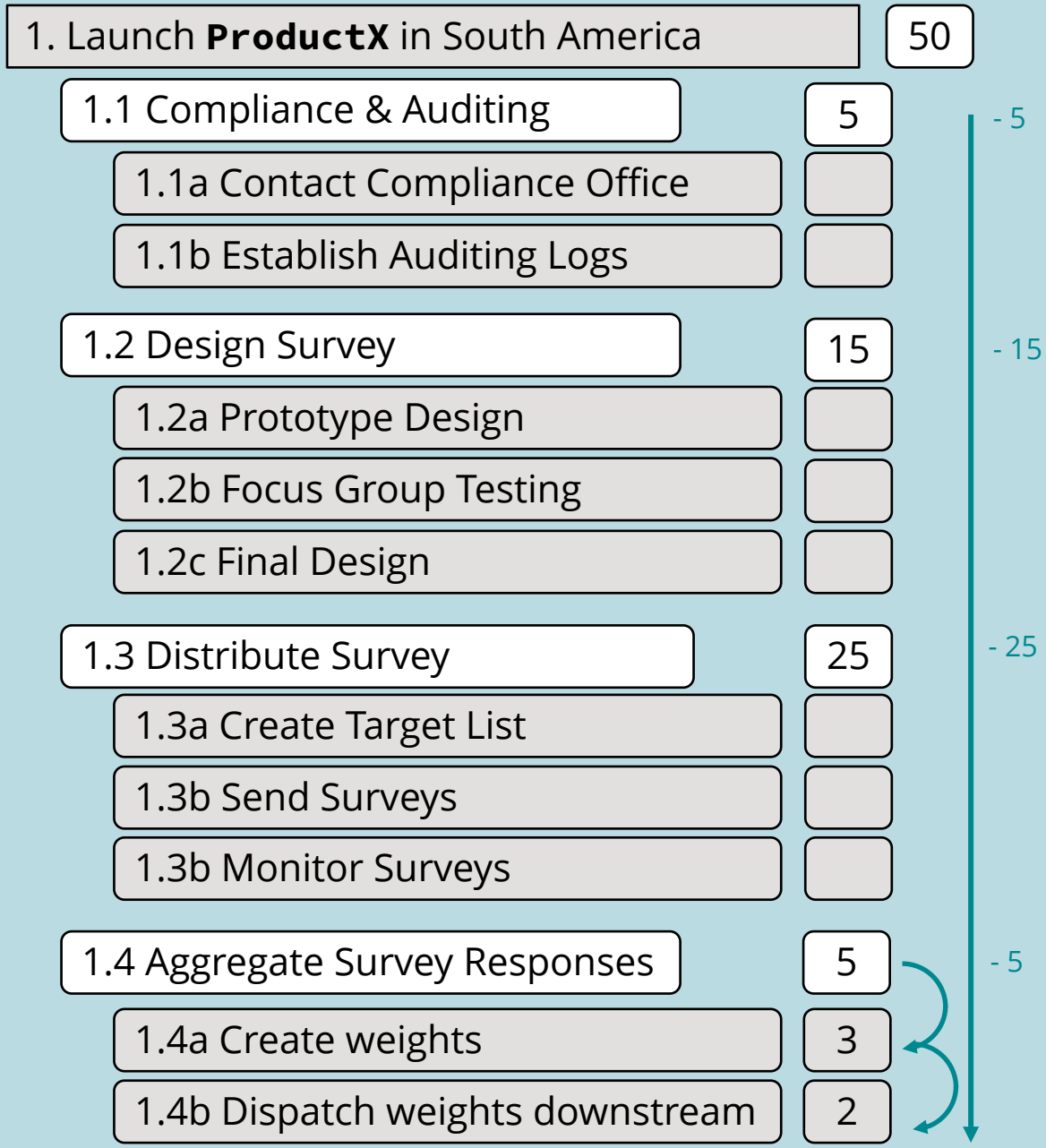
Bottom-Up Approach

- Start with the **lowest-level** tasks
- Aggregate to form comprehensive estimate
- Useful when you need to provide precise estimates to various stakeholders
- Extremely time consuming
  - Need to collect and analyze data from different teams across the company



Top-Down Approach

- Start with **high-level overview** of project and estimate total/universal limit
- Larger tasks are broken down into smaller components based on divvying up total estimation limit
- Useful when limited information or experience is available for the project
- Can be inaccurate as it relies on assumptions that may not reflect actual project scope/complexity



Give it a try! Agile vs Waterfall

Discuss how each development model fits the below hierarchy.

Usable

⇩

Maintainable

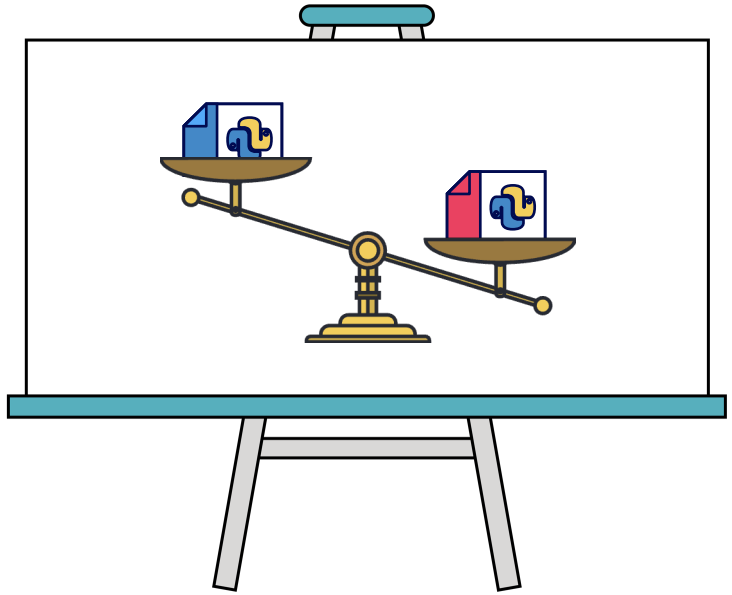
⇩

Reliable

⇩

Scalable

1. How does **agile** work within this hierarchy? What kinds of experience levels (apprentice, journeyman, master) are needed for **agile** to work?
2. How does **waterfall** work within this hierarchy? What kinds of experience levels (apprentice, journeyman, master) are needed for **waterfall** to work?



Pragmatic Development

Lecture 10
Development Models

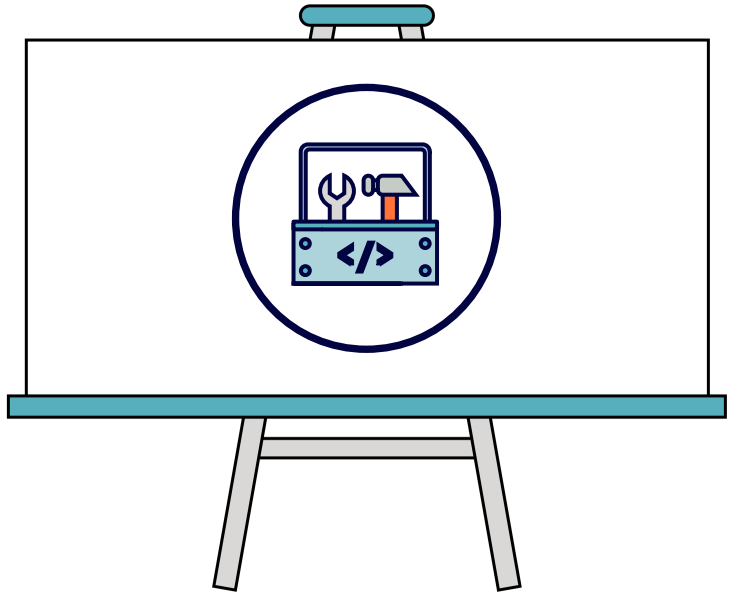
Pragmatic Development

Software developers like to compare and contrast the agile and waterfall models of development. This can sometimes lead to heated discussions about pros and cons and which is right or wrong.

Instead, each has its strengths and weaknesses. **A lot depends on who your customer is:**

- Is it a **big federal agency** spending millions of dollars?
  Probably better to lean toward **waterfall**.
- Is it a **small research project** or a startup? Probably better to lean toward **agile**.

In either case (or any other case), be pragmatic. Pick the pieces of each approach that fit the goals of your project, team, and customer.



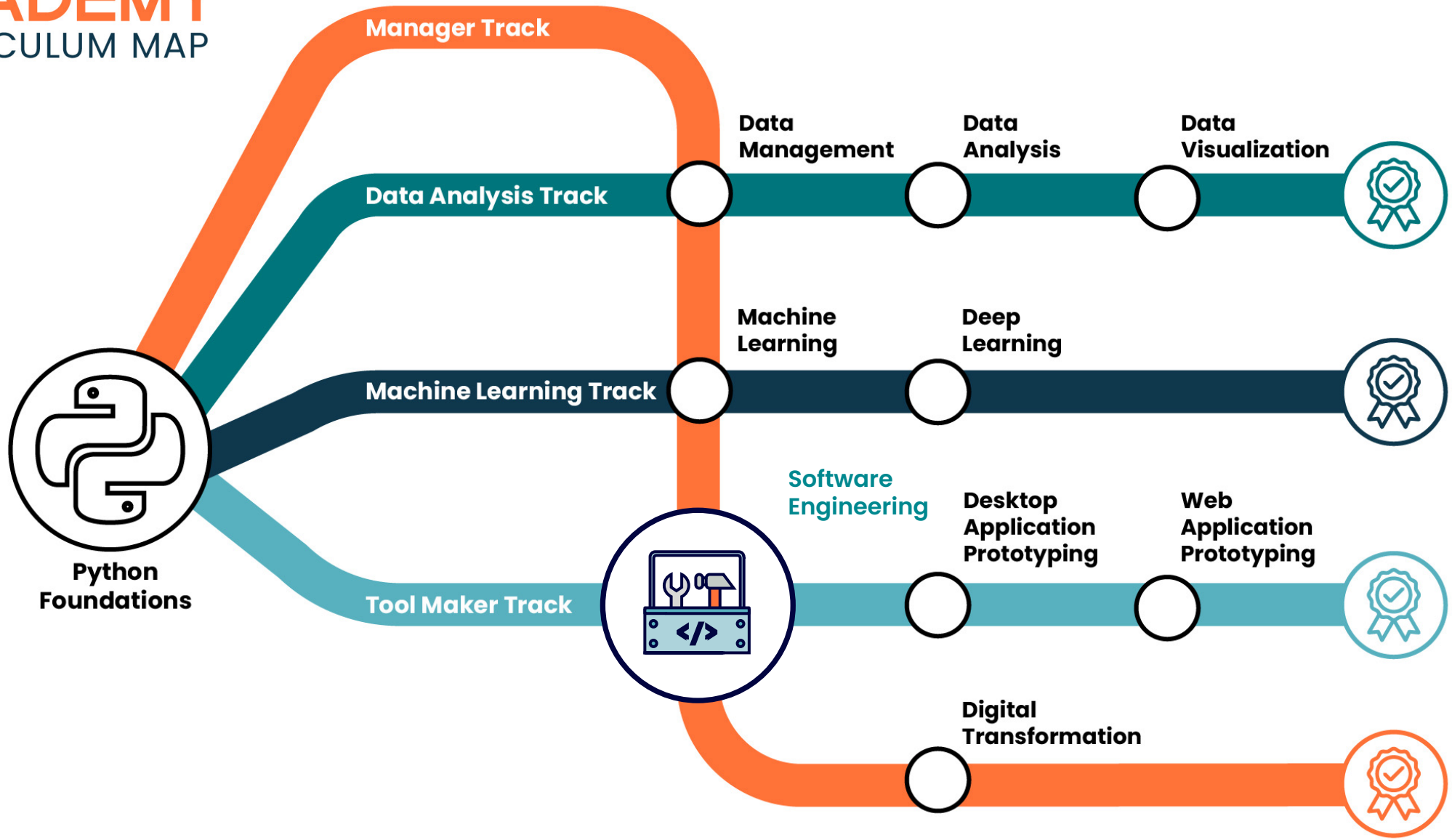
Closing Words

Software Engineering
for Scientists and Engineers

Enthought

ACADEMY

CURRICULUM MAP



Stay in touch!

@enthought

Enthought Media

Enthought

Enthought

SciPy

EuroSciPy

