Continuous Integration 1 with tests

Admin

- Assignment from last slide set and new one for now,
  - Read chapter 1 in pragmatic programmer
  - Listen to “The Programming Podcast” podcast (linked on the class web site) episode from Dec 4, 2025 (three links below)
    - https://www.youtube.com/watch?v=ap9kVWOs-fk
    - https://podcasts.apple.com/us/podcast/the-job-search-crisis-why-3-3-million-people-are/id1778885184?i=1000739722249
    - https://open.spotify.com/episode/5JxdklEjKVqbi1aFsmlH18
  - Get me that github ID
  - Install your tools as per email

Continuous Integration

- What do we mean by continuous integration?
  - Lucky volunteer?

Continuous Integration

- What do we mean by continuous integration?
  - Every time we commit code to version control, the entire project is built and tested.
  - Compare to previous approaches
    - Group might work on its piece of the project, maybe a library, and build and test it in isolation except for occasional “gold master” style builds
  - Now, since automated tests run for every commit/push/pull request,
    - you are either fairly confident that the new changes don’t break the existing project
    - Or find out about the breaks right away.

Previous CI experience

- Has anyone worked with Continuous Integration before?
  - What sorts?
    - Jenkins
    - TravisCI
    - CircleCI
    - Azure devOps
    - CodeShip
    - Bamboo
    - etc

Continuous Integration

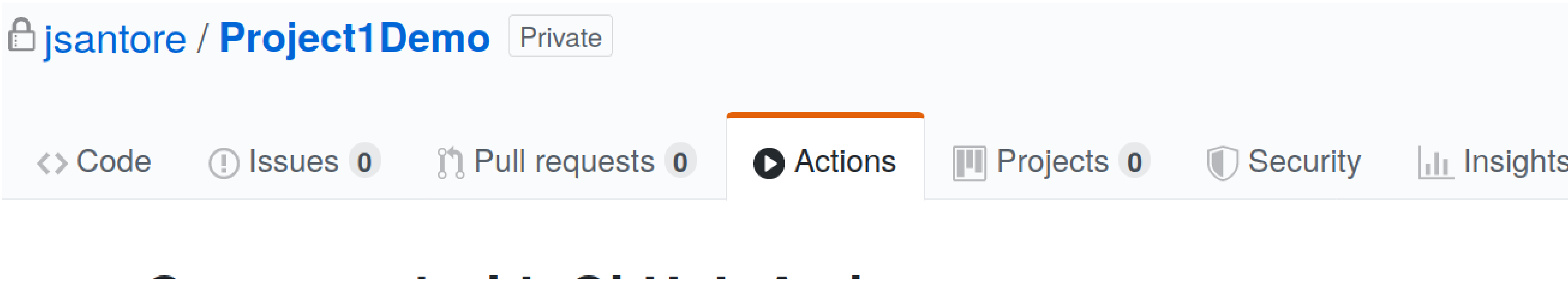
- Today the top two cloud based git servers provide CI services too
  - Gitlab has had CI for years
    - Solid, powerful experience
    - Jetbrains has good integration with gitlab for the last couple of years
  - Github introduced github actions about a few years ago
    - And made them free for everyone after the Microsoft takeover.
    - We will use github for this class
      - Since the jetbrains integration with github is also really good.
      - And it is ‘free’ if you let Microsoft datamine your every behavior
  - Remind me about usage here?
-

Trying out actions

- I’ll use a python example first,
  - but check out what github provides for your language when you press the ‘actions’ tab the first time
- Lets have a look at my example python “production code”
  - And add github actions to run flake8 on that code and automatically run the automated tests everytime you push to the branch.
  - Then look at some actions yaml syntax.

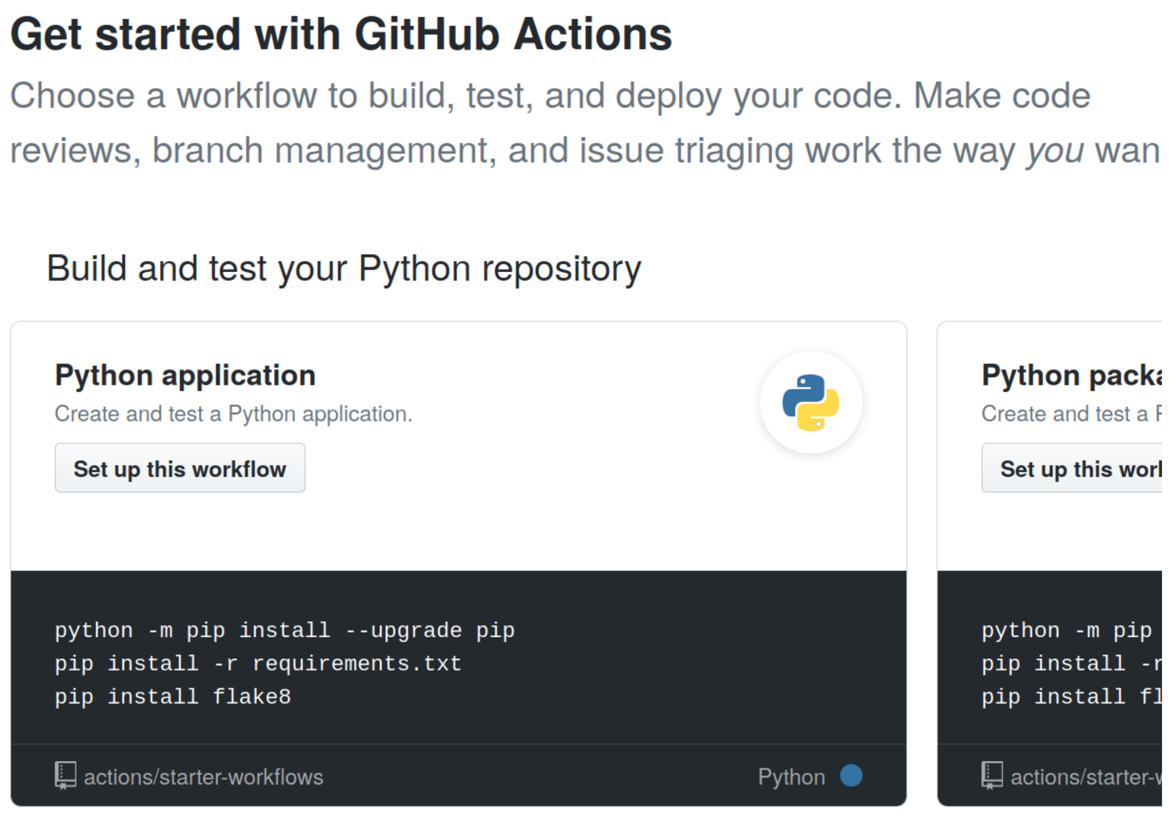
Adding Actions

- First click the actions tab
  - Before you add an actions script, will prompt you to add one
- Will bring up a few suggestions based on the dominant language in your project



Python

- Your project one will be an application, not a library
- For python projects choose python application



Get started with GitHub Actions

Choose a workflow to build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way *you* wan

Build and test your Python repository

Python application

Create and test a Python application.

```
python -m pip install --upgrade pip
pip install -r requirements.txt
pip install flake8
```

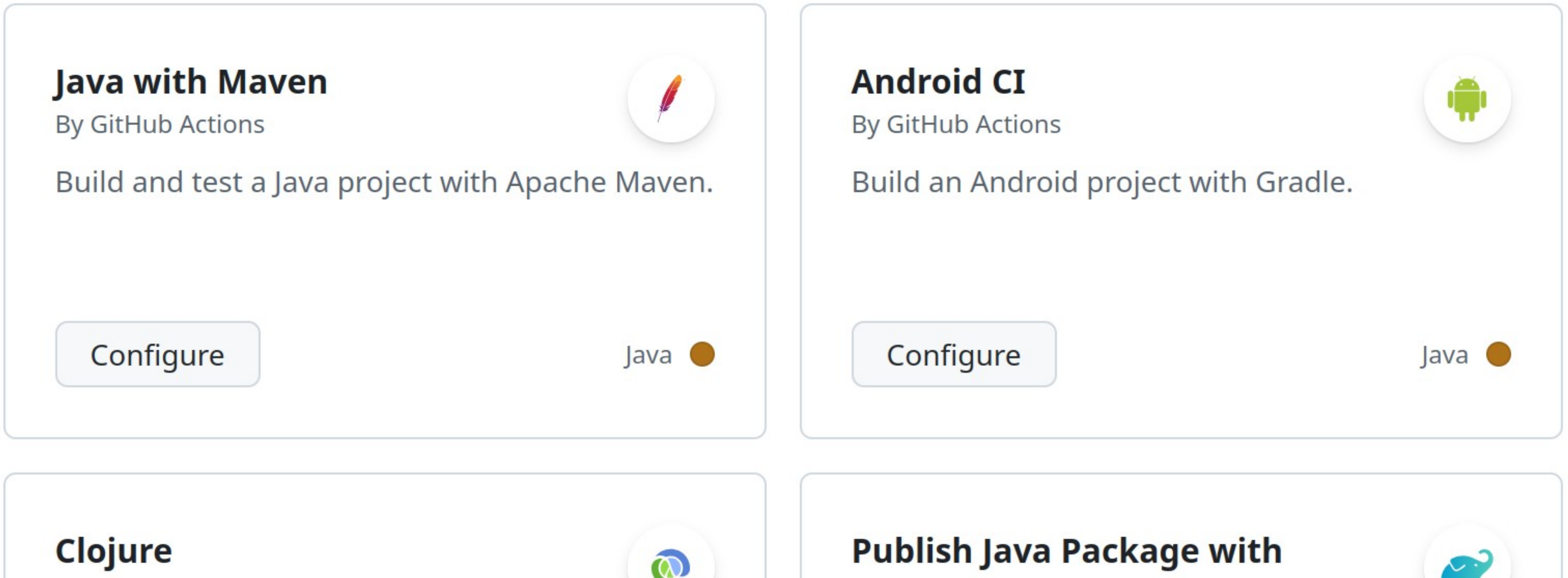
Python packa

Create and test a P

```
python -m pip
pip install -r
pip install fl
```

Java

- If you are using java – I suggest starting with the maven action
  - There are more options for java and I haven't explored them all


Java with Maven
By GitHub Actions
Build and test a Java project with Apache Maven.
Configure
Java ●

Android CI
By GitHub Actions
Build an Android project with Gradle.
Configure
Java ●

Clojure

Publish Java Package with

Github actions scripts

- Github actions scripts are yaml
  - Yet another markup language
  - (or YAML ain't markup language)
  - Yaml uses space-indenting as syntactic structure
    - Much like python
  - Uses dash character ‘-’ to denote the beginning of a step
    - Rest of step is at same indent level or indented more
  -

- Github actions need to be in your projects main folder in a sub-folder called
  - .github/workflows/
- You can have more than on action script in your project
  - Github will check each to see if it should be run.

Synatx Example:Top of default python application action

```
name: Python application
on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]
permissions:
  contents: read
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
```

- Give it a unique name
- Next on section
  - Defines when this script will be run
  - Here for any push or pull request on default branch, we will run this action
- For now, always go read only
- Next the jobs section
  - Can have multiple subsections indented
  - Build subsection is only one here

Synatx Example:Top of default python application action

```
name: Python application
on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]
permissions:
  contents: read
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
```

- Lets look at build section
  - First runs-on
    - Select a docker container w/ flavor of Linux
  - Steps:
    - Can be quite long
    - Starts with '- uses:'
      - Specify a prebuilt actions script
      - That dash is important
    - Next a series of
      - `name : <something>`
        `uses: <some other action script>`
      - Or
      - name: <something>
        run: |
          <a series of Linux command line commands>

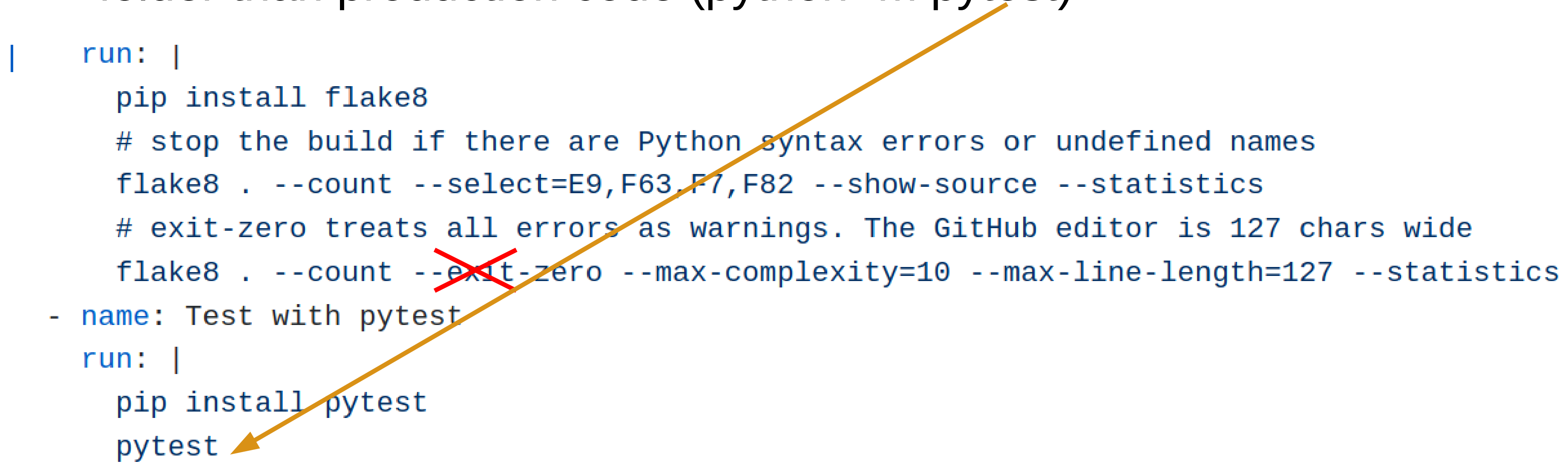
Syntax example continued, the rest of the file

```
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest
```

- Comments
  - Yaml uses same ‘#’ comment character as python
- If slide messes w/ formatting, all lines beginning with dash are at same indent.
- Run is one indent further
  - And each step in one indent under run
  - Each step is a command line linux command run on docker container

If using the python default

- Once you have the default python action
  - Change flake8 to actually fail on format errors
  - And maybe change the way pytest is run to deal with tests in different folder than production code (python -m pytest)



```
|    run: |
       pip install flake8
       # stop the build if there are Python syntax errors or undefined names
       flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
       # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
       flake8 . --count --~~exit~~-zero --max-complexity=10 --max-line-length=127 --statistics
   - name: Test with pytest
     run: |
       pip install pytest
       pytest
```

Linux for actions

- You might need more linux command line commands for actions
  - Of course running programming tools and similar (maven’s mvn command etc)
  - apt (formerly apt-get) is the ubuntu command line package manager – use it to install system packages
  - curl is a command line download tool – use it to do something like getting a model file from the internet for your actions
  - wget is another command line download tool as an alternative to curl
  - unzip is the command line unzip program to expand a zip file and extract its contents.
- Unknown unknowns to known unknowns

What is your experience with Automated Tests?

- How many have written them?
-

What is your experience with Automated Tests?

- How many have written them?
  - You are supposed to do them in comp152 and comp390
- Lucky volunteer, tell me about what they do and what they are for

Automated Tests

- Pretty much every serious software project uses Automated Tests today
  - Code that evaluates the "production code" and run automatically by the CI system
    - And should be run the the programmer on their local machine first.
    - And github actions (or similar) before code is accepted
  - May or may not exercise the entire code base, but does test/exercise at least part of it.
- Not everyone believes in TDD
  - But yes to automated tests.

Kinds of Automated Tests

- There are several ways to classify tests
- One Categorization that is used fairly commonly
  - Unit tests
  - Functional tests
  - Acceptance tests
- What are each of these? What do they do?

Kinds of Automated Tests

- Unit tests
  - Item by item – function by function tests
  - Officially “tests smallest testable unit”
    - Class? Function? Other?
- Functional tests
  - Does the app do what it is supposed to do?
- Acceptance tests
  - Does the app do what the client thinks it is supposed to do?

Why?

- So what are the tests supposed to do for us in Test Driven Development or other methods of using automated tests?
  - Why has Testing (TDD?) become so accepted in the last 10-15 years?
    - Going from something more avant-garde that many managers resisted to "table stakes" at most software development places?
  - Well actually some people still call it TDD but ‘automated tests’ might be a better term
  - What does Automated testing buy us? (especially with CI)

Assignment for new testers

- For those of you new to automated testing
  - Read a couple of introductions
  - https://katalon.com/resources-center/blog/what-is-automation-testing
  - https://medium.com/tenable-techblog/automation-testing-with-pytest-444c8b34ead2
- And a quick look at doing some of this in pytest (we'll look at some examples later)
  - https://bas.codes/posts/python-pytest-introduction
- For those of you who have done automation tests before let's move on

Unit Tests

- First an easiest tests to understand/automate are Unit Tests
-
- Testing Smallest Testable part of application
  - Functions, methods, etc
  - Sometimes the entire public interface to a class
  - Extend compiler's error checking capability.
- Traditionally each unit test should be done in isolation
  - Even if your class relies on a database, mock database and test class
  - Recently lots of conference talks pushing back against mocks, tests on each unit will include its dependencies
  - It seems to be gaining a lot of traction

Unit and Automated Test packages

- There are libraries/packages to support automated tests in nearly every important language
  - Java : JUnit (the granddaddy of all)/Mockito/cucumber
  - Python : pytest (and older unittest and nose)
  - C++ : Catch 2, google-test, unittest++
  - C# : Mstest
  - Kotlin: kotlin-test (standard lib)
- Newer language like Go and Rust:
  - Tools are built in to the language tooling, no library or framerwork required

Pytest: the current preferred python test framework

- pip install pytest
  - I suggest through pycharm unless you have a linux distro with a package manager.
  - <file><settings> menu (or <pycharm><preferences> or Mac)
  - Then choose the python item from the left list at the top
    - And the project interpreter
    - Then push the '+' icon to add a package
    - From there select pytest and install it.

Best Practices

- For best practices,
  - Have a separate test directory
  - Create a new directory as a subdirectory in your project
  - Used by nearly every language (go and rust do it differently)
- Lets call it tests.

What sorts of tests?

- What sorts of tests should we write?
  - Many people suggest at least as much test code as production code
  - AI and tests debate

What sorts of tests?

- What sorts of tests should we write?
  - Many people suggest at least as much test code as production code
  - again many people suggest at least as much test code as production code
  - Want ‘happy path’ tests
    - When all data is as expected
  - Want bad data tests
    - When we enter junk
    - c.f little bobby tables
  - Especially want to check unusual values
    - Like the (in)famous $0 billing statements
  - Eventually want to try restricting resources
    - Simulate network outage for example.

Let’s try some

- The first/easiest automated tests
- Test a single function that computes a value
  - Usual starting demo online
  - Let's write a couple of automated tests for the simpler functions
  - that automated test should find ‘error’

- Lets take a look at the TestingDemo project that I have on github
- https://github.com/jsantore/TestingDemo
  - Recently updated so should be near top

Another Test

- So the first happy path tries some easy wins
  - 3,4,5 triangle
  - Then we add in floating point answers
  - But floating point has precision and rounding issues for repeating decimals and irrational decimals
    - you’ve heard this since CS1
  - Now we run into it with these tests
  - For floating point numbers in pytest use
    - Pytest.approx(<expected number>, <acceptable tolerance>)
    - Eg
      - assert pretendProductionCode.simple_distance(0, 0, 6, 5) == pytest.approx(7.81024967590, .000001)

Junit has one too

- JUnit provides an equivalent
- public static void assertEquals(double expected,
- double actual,
- double delta)
  - Version without delta is deprecated

- Example:
  - double myPi = 22.0d / 7.0d; //Don't use this in real life!
  - assertEquals(3.14159, myPi, 0.001);
- From: https://stackoverflow.com/questions/5939788/junit-assertequalsdouble-expected-double-actual-double-epsilon
-

The save function

- Let's try to test the output function
- Let's look at the two options
- And then test the one that can be tested.

The Podcast

- Now lets talk about ‘The Programming Podcast” episode:
  - **The Job Search Crisis: Why 3.3 Million People Are Failing (And How To Fix It)**
  -