Lab 04: Version Control

CMPT 145

Laboratory 04 Overview

Section 1: Overview ▸ Slide 3

Section 2: Demonstration ▸ Slide 21

Section 3: Multiple Versions ▸ Slide 46

Section 1

Version Control Overview

Version Control

Motivation

- You're writing long essay.
- You save your document every few minutes.
- An hour ago, you decided to delete a paragraph about Batman, and you've written several paragraphs about Shakespeare after that.
- Now you want your Batman paragraph back.
- What do you do?

Motivation

- You’re writing big Python program to analyze some data from a recent experiment.
- You save your script every few minutes.
- An hour ago, you decided to delete some test cases, and you’ve made several changes and revisions to your code after that.
- Now you want your test cases back.
- What do you do?

Do-it-yourself Version Control

You may already be doing version control without any help:

- Using the UNDO/REDO buttons on most applications.
- Using comments to hide code you've written, but are afraid to delete.
- Maintaining your own backup system, with folders containing "old" documents.
- Remembering the changes you made.

These might work for trivial projects, but not larger ones. If only there were software to help!

What is Version Control?

- A Version Control System (VCS) is a software tool that allows content providers to manage their project content.
  - "Content" means articles, essays, programs, scripts, books, papers, audio, video.
  - As the project is edited or revised, the VCS user records every version of the project.
- A version is essentially a backup, which is maintained by the VCS.
- The entire history of all versions is stored carefully.

Using Version Control

- You’re writing long essay.
- You decide to delete a paragraph about Batman.
  - Before the deletion, you use the VCS to record the state (version A) of your essay.
- You write several paragraphs about Shakespeare, and you want your Batman paragraph back. Using the VCS you can:
  - Record the current state (version B) of your essay.
  - Jump back to version A. Copy the paragraph!
  - Jump forward to version B. Paste the paragraph!
  - Record the current state (version C) of your essay.

Using Version Control

- You’re writing big Python program.
- You decided to delete some test cases.
  - Before the deletion, you used the VCS to record the state (version A) of your program.
- You’ve made several changes and revisions to your code after that. Now you want your test cases back. Using the VCS you can:
  - Record the current state (version B) of your program.
  - Jump back to version A. Copy the test cases!
  - Jump forward to version B. Paste the test cases!
  - Record the current state (version C) of your program.

Without version control

- Computer file systems are not your friend.
  - When you save a change to a file, you destroy the previous contents.
- If you decide that your changes made your project worse, you cannot go back.
  - Unless you saved your previous version!
  - That's one of the things version control does.
- You might be able to use the UNDO/REDO keys, but repeatedly using UNDO is a sign you need version control!
- Gamers will recognize the value of having a *saved game*:
  - Save the current game state.
  - Try out several strategies from the saved state.

How Version Control Software Helps

- Backs up your work.
- Allows you to document your back up copies.
- Allows you to
  - Return to a previous version.
  - Advanced: Develop multiple versions in parallel.
  - Advanced: Switch between versions.
  - Advanced: Collaborate with multiple collaborators.
  - Advanced: Work on your project from any computer connected to the internet.

What can Version Control do for me?

- Switch to any stored version at any time.
- Tailor similar content for a specific clients or purposes.
- Multiple versions can be compared, to see differences in the content.
- Teams can collaborate on different parts of the project at the same time.
- Teams can collaborate on different versions of the project at the same time.

Version control for everyone

- Version control was invented by programmers, for software development.
- But version control can be applied to any kind of project you create with a computer. E.g.
  - Microsoft Word has version control software built-in (the Review tab)
  - PyCharm has multiple version control tools available.
  - UNIX systems (Mac, Linux, etc) provide version control software by default.

What is a version?

- A version is
  - The contents of all the files you want included.
  - At a point in time.
  - You decide when to call it a version.
- To make a version:
  - Tell VCS to backup the current state of the project.
  - It's helpful to make versions at short intervals.
- Versions are stored in your project folder.
  - Out of sight; it doesn't clutter your project.

Basic Version Control: a walk-through

- Start a new project, assignment, essay:
  - Initialize version control for that project.
- Begin work on the project.
  - Add files to your new project.
- Complete a task for the project:
  - Commit the current state to make a documented backup.
- Need to work on an alternate version?
  - Identify the version you want, and return to it (checkout).

Version Control Software Packages

There are many systems that implement version control.

- Git
- CVS
- SVN
- Mercurial
- Bazaar
- Darcs
- …and many more

We'll use Git, but the concepts are transferable.

My projects are too trivial...

- Investment: Become familiar with VCS before you actually need it.
- Payoff: Future CMPT courses expect you to use VCS:
    - To submit your work
    - To document your contributions on group projects
    - To download code to be used as a starting point for assignments and projects.
- Payoff: The software you write for your own projects
    - You will adapt and repurpose it over time

VCS on multiple Operating Systems

- Git is a set of tools independent of PyCharm.
- PyCharm provides access to Git through menus, and icons.
- Unfortunately, PyCharm uses slightly different icons on Windows, Linux, and Mac.
- Be sure to become familiar with PyCharm's interface to Git on your preferred platforms.
- Git is famous for cloud-based storage (e.g., gitlab, github, etc).
  - We'll focus on local use of the tools for now.
  - Using the cloud means you can work on your project on any computer with internet access.

Lots to learn!

- You only need to know what CMPT 145 labs teach. Nothing more is expected.
- But there's a lot more to learn, if you want to go beyond CMPT 145.
- A very good Git tutorial is: `http://www-cs-students.stanford.edu/~blynn/gitmagic/index.html`
- Note: most tutorials present Git using command-line tools. We're starting with PyCharm menus and buttons.
- About once per semester, a software developer from a local company gives a Git tutorial for CMPT students.

Section 2

Version Control Demonstration

Version Control in PyCharm

Basic Version Control: a walk-through (recap)

- Start a new project, assignment, essay:
  - Initialize version control for that project.
- Begin work on the project.
  - Add files to your new project.
- Complete a task for the project:
  - Commit the current state to make a documented backup.
- Need to work on an alternate version?
  - Identify the version you want, and return to it (checkout).

Initializing a new project

ACTIVITY:

1. Create a new PyCharm project, called Lab04.
2. Find the VCS menu in PyCharm.
3. Select Enable Version Control Integration...
4. Select Git from the drop-down menu.
5. Click OK

You'll notice PyCharm's interface has changed slightly.

Begin work on your project

ACTIVITY:

1. Create a new Python file, called `fact.py`.
   - PyCharm will ask you about Add files to Git
   - If you don't add files, Git won't know to include them.
   - PyCharm uses coloured fonts in the Project pane to show files that have been added.
   - Keep the defaults, and click Yes
2. At the bottom of your IDE window, a new tab appears called Version Control.
   - Note: The new tab may not appear immediately.
3. Click and explore, but don't do any actions.
4. There may be some new icons on PyCharm icon bar.

Complete a task for the project

ACTIVITY: Add some Python code to `fact.py`

```
# CMPT 145 Lab04: Version Control
#

def factorial(n):
    """
    Purpose:
         Calculate the factorial of a non-negative integer
    Pre-conditions:
         n: non-negative integer
    Return:
        a non-negative integer
    """
    pass
```

It does nothing useful. We call it a stub.

Frequent small versions of your project

- Your script and function don't do anything yet, but it's a start.
- Run your script.
- Be sure there are no errors!
- Let's call this a version!
- We say make a commit to save a version.
    - Making a commit is similar to saving your work from time to time, but better, since it preserves your previous version in a hidden but completely accessible backup.

Commitments

- The word “commit” seems laden with hints of responsibility and serious work.
- But it’s technical jargon for “recording the new information”
- A commit could be made for any of these reasons:
  - You designed a function (doc-string only)
  - You implemented a function.
  - You added a few test cases for your function.
  - You fixed a bug.
  - You revised your function to be more efficient.
  - You added more comments to your code before submitting for grading.

Making a version: commit

ACTIVITY:

1. Find the Commit Changes button ⑂ or VCS or ✓.
   - Click the Commit Changes button.
   - A big window pops up, showing all changes since the previous version.
   - Look for a box called the Commit Message.
2. In Commit Message box, type

```
Added factorial stub function.
```

3. Click Commit at the bottom of the window.

Browsing your changes

ACTIVITY:

1. Find the Log tab in the Version Control panel.
2. You'll see the word master and a few words from your commit message.
   - Git keeps a stack of backup versions.
   - The name master is the default name for the initial branch created by Git.
   - We won't deal with branching yet, but you can have multiple branches, and each one will need a name.

Make a bit more progress

ACTIVITY: Add code to your function:

```
def factorial(n):
    """
    Purpose:
         Calculate the factorial of a non-negative integer
    Pre-conditions:
         n: non-negative integer
    Return:
        a non-negative integer
    """
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Save your changes using PyCharm as normal.

Committing your new version

ACTIVITY:

1. You have made a change to your program.
2. Let's call this a new version!
3. Click the Commit Changes button ⑂ or VCS or ✓.
4. In Commit Message box, type

```
Coded up a recursive implementation of factorial.
```

5. Click Commit at the bottom of the window.

Advice

The better your message is, the more useful it is.

Browsing your changes

1. Find the Log tab in the Version Control panel.
2. The new version is on the top.
3. The master label has moved: it's at the most recent version.
4. You can click on either version, and see the full commit message, and the date it was committed.

Complete a task for your project

ACTIVITY: Recursion is fine, but you want to change to a loop. Change the code as follows:

```python
def factorial(n):
    """
    Purpose:
         Calculate the factorial of a non-negative integer
    Pre-conditions:
         n: non-negative integer
    Return:
        a non-negative integer
    """
    prod = 1
    for i in range(1, n):
        prod = prod * i
    return prod
```

Save your changes using PyCharm as normal.

Committing your new version

ACTIVITY:

1. You have made a change to your program.
2. Let's call this a new version!
3. Click the Commit Changes button ⑂ or VCS↑ or ✓.
4. In Commit Message box, type

```
Replaced the recursion in factorial() with a loop.
```

5. Click Commit at the bottom of the window.

Advice

The better your message is, the more useful it is.

Browsing your changes

Activity:

1. You'll see that the log has updated.
2. Click the top (most recent) version.
3. Find `fact.py` listed under version control (to the right of the log). Click on it.
4. Find the Compare versions button ⇄ or ▣. Click!
   - On Windows, you may need to select both versions, and right-click.
5. A window pops up showing two versions of the file:
   - The left side shows the previous version.
   - The right side shows your current version.
   - You can see exactly what changed!
6. Close the window when you're done.

Returning to a previous version

Advice

This is the important part of the lab.

- Because you have used Git, you have 3 different versions of the function, documented by commit messages.
- After any commit, you can make changes and experiment with ideas.
- If you're happy with your changes, commit ⑂, VCS, or ✓.
- If you're not happy, you can:
  - Discard all changes since the last version
  - Select changes to keep and discard.
  - Revert to any version in the past.

Discarding all changes

- You made some changes, but you're not happy.
- You want to return to the state of your project at your most recent commit.
- Click on `fact.py` in the Project pane.
- Find the VCS menu, select Git, select Revert ↶.
- You're back to the most recent committed version.

Discarding some changes

- You made some changes, but you're not happy.
- You want to return to the state of your project at your most recent commit.
- Click on `fact.py` in the Project pane.
- Open the Compare versions window or .
- Differences are highlighted.
- Look for >> beside line numbers between the panels.
- Clicking >> moves the previous version into your current version.
- When you're done, close the window.
- If you're happy, commit the version VCS ✓.
    - Don't forget to commit!

Revert to a version in the past.

- You made some changes, and some commits, but you're really not happy.
- You want to return to the state of your project some time in the past.
- Click on `fact.py` in the Project pane.
- Open the VCS panel, and click the History tab.
- The history of the file is shown, documented by your commit message.
- Click on any version in the past.
- Click on the Get 📥 button.
- If you are sure you want to keep that version, commit again!

Version Control gives you control

- Remembering to save your work, and back it up, takes practice.
- Giving good commit messages takes practice.
- Using your judgement about when to return to a previous version takes experience.
- If you commit regularly, you can be more experimental with your coding, and debugging.

How to use Version Control for assignments

- Create a new project for your assignment; use a network filesystem if you are on a departmental computer.
- Initialize version control right away, with the empty project.
- Commit your work often, as you progress.
- Commit your work whenever you reach a point where your program does something useful. This should be frequently!
- Always give a good commit message. These will help you find what you look for when you need it.

How to use Version Control for debugging

- Commit your code after every bug you fix.
- Document the bug and the fix in the commit message.
- If you ever discover a new bug, you may find it helpful to compare versions, to see what changed.

ACTIVITY: Working with VCS

- Add some simple code to test the factorial function. (Hint: `factorial(5) == 120`)
- Save and commit your work at various points.
- Using the History of `fact`, show that the recursive version is correct, but the non-recursive version has a bug.
- Correct the bug in the non-recursive version, and commit your changes.

ACTIVITY: Submitting your Git-log

- In PyCharm, find the Terminal tab. Click!
- Type `git --no-pager log` at the prompt, and press enter or return.
- You should see information printed to the console roughly the same as what you see in your VersionControl: Log tab.
- It may have scrolled past the limits of the window.
- Copy/paste the whole log to a text file named `a9q5-log.txt`, and submit to Moodle Lab 04.

Section 3

Multiple Versions

A scenario that calls for multiple versions

Multiple versions are appropriate (1)

- Commercial scenario:
  - You are developing a large application, with lots of features.
  - You want a basic version with limited features, with a low price.
  - You want an advanced version with lots of bells and whistles, for a higher price.
- Many commercial software development projects are like this.

Multiple versions are appropriate (2)

- Development scenario:
  - You are developing a large application, with lots of plans for many features.
  - You plan to start with a simple version with very few features.
  - Each version of your application will add more and more features.
- This is the incremental and iterative development process.

Multiple versions are appropriate (3)

- Research scenario:
  - You have a data analysis script to answer a scientific question.
  - You realize that a slightly modified script could answer different scientific questions about a slightly different dataset.
  - You need multiple versions to address the different datasets.
- Many research projects have scripts created this way.

Example: The MM1 Queueing algorithm

- Did you ever wonder what would happen if the MM1 simulation used LIFO order instead of FIFO?
- Humans value fairness, and queues (FIFO) seem to be fair.
- Just how unfair would it be if customers were served in LIFO order?
- To answer, we could repeat the MM1 simulation using a Stack instead of a Queue.
  - This is an example of needing multiple versions for a research project.

A list of bad ideas

To answer the FIFO vs. LIFO question, we could do one of the following:

- Edit the MM1 program.
- Copy the MM1 program, and edit the copy.
- Edit the Queue ADT.

These are all terrible ideas!

Why editing is bad

- To answer the FIFO vs. LIFO question, we could edit the MM1 program.
- Changing a working program to have different behaviour is fine, as long as the old behaviour is no longer needed.
- In this experiment we want both behaviours:
  - MM1 with a FIFO queue (standard).
  - MM1 with a LIFO stack (experimental).
- Editing back and forth is a waste of programmers' time!

Why copying is bad

- To answer the FIFO vs. LIFO question, we could copy the MM1 program, and change the copy.
- Suppose there are errors you didn't notice before you copied.
  - Copying the program copies the bugs!
  - Twice as much code to fix!
- Suppose we want to add code to the MM1 program, say, to collect more data about wait times.
  - Copying the program forces us to modify both copies the same way.
  - Takes twice as long to make the changes.
- Having two copies of the same program means that you have twice as much code to worry about.

Why changing an ADT is bad

- We could edit the Queue ADT and change the code:
  - Make enqueue behave like Stack's push
  - Make dequeue behave like Stack's pop
- The would affect every program that already uses your Queue ADT!
- You'd have to change it back when you're done.
  - If you forget, scripts that were correct will have faults, and you might not remember why.

Summary of bad ideas for adaptable software

- Copying code is a bad idea.
  - More code means more errors, and more time debugging.
- Editing code repeatedly is a bad idea.
  - Wastes programmer time!
- Modifying an established ADT is a bad idea.
  - Changes the behaviour of every working application that uses it.

Version Control for Multiple Versions

Version Control for Multiple Versions

- Software always changes!
- You may find you need two similar but distinct versions of your current project.
  - No, not in your first year assignments, but maybe in longer term projects (summer research, etc)
- We'll see how to use Version Control to manage multiple versions!

Version control and branching

- So far, we've used version control as sophisticated backup software.
- Git allows us to create multiple versions, called branches.
- When we create a branch, we are creating an exact copy of the whole project.
- When we make changes to a branch, the changes only affect that branch, not all branches.
- We can jump between branches at any time.
- We only see one branch at a time.

Version control to the rescue!

- We'll keep the FIFO queue version as the main version.
    - Git initializes the project creating a branch called master.
- We'll make a new branch, and edit the script to use a LIFO stack.
    - We'll name the new branch StackSim, to reflect its purpose.
- We can jump between these branches when we want to change behaviours.
    - To get the Queue simulation, we'll jump to the master branch.
    - To get the Stack simulation, we'll jump to the StackSim branch.

Version Control Terminology

commit (verb) to save the changes made.

commit (noun) the state of your files when you committed them.

branch (verb) create new copy of the files.

branch (noun) a set of files.

checkout branch-name (verb) to jump to the named branch.

checkout file-name (verb) to retrieve the named file from the most recent commit of the branch you are currently on.

master (noun) the default name for the initial branch created when git initializes a set of files.

Version Control for Multiple Versions: Activities

Activities Overview

This portion of the lab will have the following steps:

1. Create a new project, and initialize version control (Git).
2. Create a new branch for the version using a stack.
3. Become familiar with jumping between branches.
4. Create a new branch for the version using a queue.
5. Working with two different versions.

Activity: GIT Step 1: New Project

- Create a new project, initialize version control, add the MM1 files
    - The original ones from Moodle, not the previous exercise.
- Make sure the script executes properly, using FIFO queues as given.
- Add a new file, called README.txt, with the following text:

```
This version of MM1 uses a Queue. If this were not a lab
exercise, I would write more information about it.
```

- When everything is working, commit the version!

ACTIVITY: GIT Step 2: A branch!

- Create a new branch, named StackSim.
- In PyCharm: VCS menu, GIT, Branches…, New Branch
  - Because it's an exact copy, everything will look exactly the same as before.
  - New Branch will be greyed out if you haven't committed your files yet!
- Change the MM1 script to use Stack instead of Queue.
  - Edit the MM1 script directly! Don't bother with adapters this time.
  - Don't be afraid, your original code is saved under the master branch.
  - Edit the README.txt file (replace Queue with Stack).
- When everything is working, commit the version.

Activity: GIT Step 3: Moving between branches

- To jump between branches, we use “checkout”.
  - In PyCharm: VCS Menu, Git, Branches…, Local branches, master, Checkout.
  - Checkout will swap your files so that they are exactly as they were before you made the StackSim branch.
  - Note: If you have edited but have not committed a change, git will not allow you to complete the checkout.
- View the MM1 script, and see the Queue is being used again.
- Jump between the two versions a couple of times (using checkout as above), to see how git works.

ACTIVITY: GIT Step 4: Another branch

- Checkout the master branch again.
- Make a new branch, called QueueSim.
    - This will be an exact copy of the master branch, with a descriptive name.
    - The name will remind you of its purpose, and will be symmetric with the StackSim branch.
- Now you can jump between 3 branches: master, StackSim, QueueSim.
- You can run either version without editing your code!
- Your project folder is not cluttered with variations.

ACTIVITY: GIT Step 5: Tidying up QueueSim

- Checkout the QueueSim branch.
- It should have the Queue ADT, but does not need the Stack ADT.
- Remove the Stack ADT file from PyCharm project.
  - Don't worry, this will not affect any other branch!
- Before you go on, check that everything is still working.
- Commit!

Activity: GIT Step 6: Tidying up StackSim

- Checkout the StackSim branch.
- It should have the Stack ADT, but does not need the Queue ADT.
- Remove the Queue ADT file, from PyCharm project.
  - Don't worry, this will not affect any other branch!
- Before you go on, check that everything is still working.
- Commit!

Git Summary

- We have two different versions of the MM1 script on separate branches.
- By checking out branches, we can jump between versions.
- In PyCharm, the bottom of the window indicates the branch name, and lets your checkout different branches quickly!

Git helps those who help themselves

- Git won't understand why you need versions, only that you have several.
- When you have a project under version control, it's important to:
  1. Make very good commit messages.
  2. Have external documentation to describe your versions.
- If you do not do these things, you'll forget and regret!

More advanced use of Git

- We used Git to create two branches for two distinct versions.
- Git's branches can be used in other ways.
  1. A feature branch can be created for each new feature you add to the application.
  2. A development branch can be created while you fix a tricky bug.
- To do these tasks, we need to learn a bit more about Git.
- Find reasons to practice branching!