OSS Module 01 - Git Basics

Contents

1: About this training module

Welcome to the GT OSPO VSIP! This training module will introduce you to concepts of Git, especially related to branching and merging. Understanding Git branching/merging will be essential to your success in as code contributor.

How should I use this training module?

You'll get the most out of this module if you follow-along interactively.

You can download this Jupyter notebook and work directly inside it. Make sure you have the `jupyter`, `jupyterlab`, and `bash_kernel` Python modules installed. **Don't forget the second step with `python -m bash_kernel.install`!**

```
$ pip install jupyter jupyterlab bash_kernel
$ python -m bash_kernel.install
```

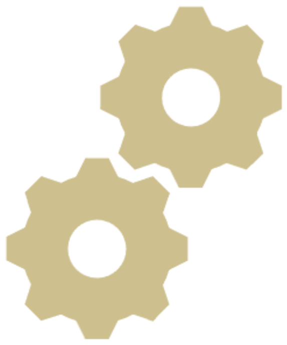
If you are not installing Jupyter or `bash_kernel`, you can also follow-along in any standalone terminal.

Why aren't we learning to use the Git GUI in (VS Code, PyCharm, ...)?

Good question! We love our IDEs, we use Git in our IDEs all the time, and so should you. **But** it really helps to know how to precisely use Git in the command line, especially if your IDE isn't doing exactly what you want. Trust us, it can really help you out of a jam.

Another issue is that not everyone uses the same IDE. All the terms and concepts you learn here will translate directly to whatever IDE you use.

2: Git and GitHub setup



Installing Git

GitHub has excellent instructions for installing Git on macOS, Windows, and Linux: https://github.com/git-guides/install-git

If you are on Windows, be sure include the 'Git Bash' terminal during the installation (it should be selected by default). You can use Git Bash to follow-along with this tutorial. If you are on macOS or Linux, you can use any terminal to follow-along.

First-time Git Setup

After installing Git, you should set your username an email. These will be used to identify changes that you've made to a Git repo. In the terminal run `git config` with your own name email:

```
git config --global user.name 'George Burdell'
git config --global user.email 'gburdell@gmail.com'
```

The `user.name` is arbitrary and is only for the benefit of other humans. However, the `user.email` is quite important when you're working with GitHub, GitLab, or another service. This email is used to link your GitHub account to any commits in a Git repo. It's not a bad idea to use a `user.email` that you'll have access to forever, instead of your GT email. This will ensure that you can always keep your GitHub account linked to commits in Git repos.

For more first-time setup options, see here: https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

Registering for GitHub

If you don't already have a GitHub account, go to https://github.com/ and create one. Again, it's not a bad idea to use an email address that you'll have access to forever, instead of your GT email.

Setting up authentication with GitHub

When using Git, there are several method to authenticate with GitHub. These days, username/password is *not* one of those methods! The simplest method is via SSH. See the GitHub docs section "Connecting to GitHub with SSH" (https://docs.github.com/en/authentication/connecting-to-github-with-ssh), which has instructions for every operating system.

3: About Git

What is Git?

Git is the most widely used **version control system (VCS)** today. Any VCS uses a database to record changes to a set of files over time, such as the source files in a software project. The changes over time are called the history **history**. Using the database, you can retrieve specific **versions** of files from any point in history.

Why is Git?

One of Git's biggest advantages is to make **collaboration** easier. By using a branching workflow (which we'll discuss below), multiple developers can work independently on the same codebase.

Another advantage of any VCS is the ability to **respond to bugs**. If your app is in production, and a bug is discovered, then you can revert the app to a previous stable while you investigate the cause. The VCS also helps you discover which **specific change and developer** instroduced the bug. The intent is not to blame or shame anyone. Rather, the idea is that the developer who introduced the bug is the one who is most likely to know that part of the codebase and respond to that bug quickly.

Parts of a Git Project Directory

The contents of a Git repo are stored on your computer as normal files in a normal directory. In Git terminology, these files are the **working tree**. You can edit these files freely with your usual text editors and IDEs.

On your computer, changes to the files are stored in the **local repository** (or local **database**). Soon, we will learn how to store (or **commit**) new versions of the files; and how to retrieve (or **checkout**) previous versions of the files.

A small project directory is shown below.

- The database and metadata are stored in the hidden `.git/` directory. You can treat this as a black box, and your only interaction with the database and metadata should be via `git` commands.
- `my_program.py` and `README.md` are files you are actively working on. You definitely want to record these in the local repo.

- `__pycache__` contains intermediate bytecode from running your Python program. You probably do *not* want to record these in the local repo...

```
~/my_git_project$ ls -la
total 95
drwxr-xr-x  4 gburdell  staff   6 Jan 25 12:33 .
drwx--x--x 43 gburdell  staff  97 Jan 25 12:33 ..
drwxr-xr-x  7 gburdell  staff  10 Jan 25 12:33 .git
-rw-r--r--  1 gburdell  staff  69 Jan 25 12:32 my_program.py
drwxr-xr-x  2 gburdell  staff   3 Jan 25 12:33 __pycache__
-rw-r--r--  1 gburdell  staff  51 Jan 25 12:33 README.md
```

Tracked vs. Untracked Files

Git does not automatically add files from the working tree to the repo. Instead, the user must explicitly specify which files are stored in the repo (**tracked**) and which are not (**untracked**). It's important to consider which files to *not* track.

In a Python project, you should not track byte code (`*.pyc` and `*.pyo` in `__pycache__` directories) since they will be automatically regenerated whenever someone else runs it with a different operating system or Python version. Tracking them will just take up unnecessary space in the database.

For C/C++ and other compiled languages, you should not track build artifacts (`*.o` files, libraries, executables) for a simlar reason. These artifacts will usually not work on a different operating system.

4: Working with a Git repo

In this section, we will create a local repo, add files, and work remote repo on GitHub

4.1: Creating a repo and committing files

`git init`: Creating a local repo

The `git init` command creates a local repo in the current directory. Let's create a new directory, `~/git-workflow` and initialize a repo inside it.

(You can use a directory with a different name and location, if you like.)

In [1]:
```
mkdir ~/git-workflow
cd ~/git-workflow
git init
```

```
Initialized empty Git repository in /Users/gburdell/git-workflow/.git/
```

You can see the `.git/` directory was created for the database and metadata.

In [2]:
```
ls -lha
```

```
total 0
drwxr-xr-x    3 gburdell  staff    96B May 10 16:05 .
drwxr-xr-x+ 193 gburdell  staff   6.0K May 10 16:05 ..
drwxr-xr-x    9 gburdell  staff   288B May 10 16:05 .git
```

`git status`: Show the state of the working tree

`git status` shows the state of tracked files, untracked files, and the database. Right now, the working tree is empty, and the database is empty, so there's not much to report.

We already have a **branch** called `main`, which was created when the repo was initialized. We'll cover branches in detail soon. For now, you can think of them like branches of a family tree that describe your project's history.

In [3]:
```
git status
```

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Creating new files

To create and edit files in your working tree, you can use any text editor. In `~/git_workflow`, let's create two files.

- First, create a file named `README.md` file with these contents. You might notice a mistake, but don't correct it right now! :)

```
# Git Workflow
```

```
This is an example repo for the GT OSPO VSIP Spring 2024 Program.
```

- Then, let's create a Python source file named `my_abs.py` with these contents. We will be making changes to `my_abs.py` throughout this demo.

```python
def my_abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Untracked files

`git status` reports that the new files are **untracked**. Recall that untracked files are not recorded in repo's database, and that Git does not automatically track new files. Git helpfully suggests that we should track them with the `git add` command, and we'll do that soon. But first let's discuss the possible **states** for files in your working tree. Knowing these states can help you troubleshoot many issues when you're working with your repo.

```
In [5]: git status
```

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
	README.md
	my_abs.py

nothing added to commit but untracked files present (use "git add" to track)
```

The state of files in a working tree

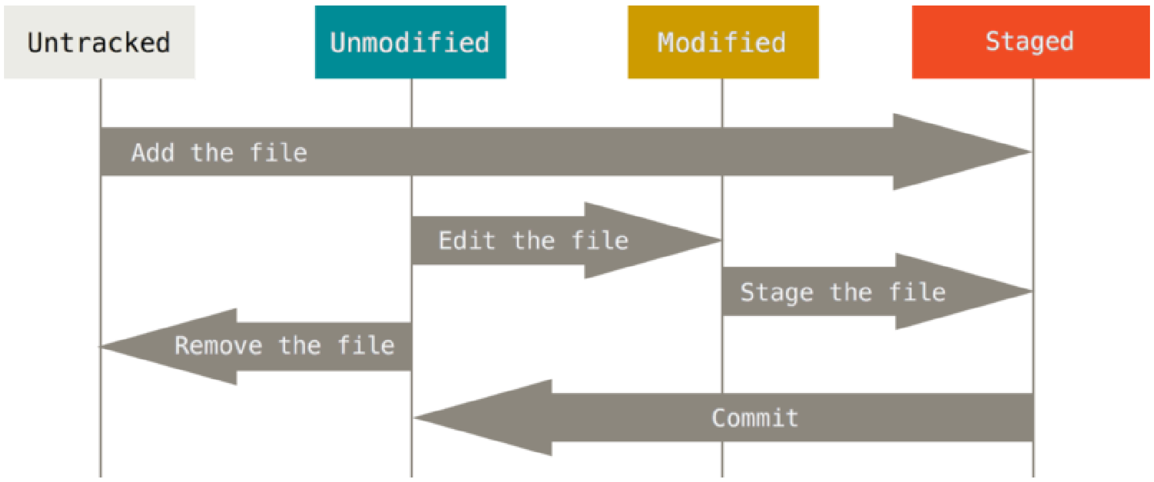
Files in the working tree are either tracked or untracked:

- **Untracked**: No versions of the file are stored in the repo's database
- **Tracked**: One or more versions of the file are in the database

Additionally, changes in tracked files are always in one of three states:

- **Unmodified**: The file in the working is tree are up-to-date with the database. Git sometimes refers to these changes as "up-to-date".
- **Modfied**: The file has changes that have not yet been stored in the database. Furthermore, the user hasn't specified that these changes will be stored in the next database update.

- **Staged**: The file's changes that will be stored in the next database update. Git sometimes refers to these as "to be committed".



(Image credit: Scott Chacon and Ben Straub. Pro Git, Section 2.2)

`git add`: Stage new changes

In our working copy of `git_workflow`, we have two untracked files: `README.md` and `my_abs.py`. To record their changes, we need two steps (as shown in the diagram above):

1. **Add the files:** This changes the files' states from "Untracked" to "Staged". At this point, the changes are *ready* to be stored in the database but *are not yet stored*.
2. **Commit**: This changes their state from "Staged" to "Unmodified". At this point, the changes have actually been stored in the database.

To accomplish Step 1 (Untracked → Staged), we use `git add`. Its usage is:

```
git add <file1> [<file2> ...]
```

In [6]:
```
git add README.md my_abs.py
```

Now `git status` shows the files have changes that are "to be committed". This means the changes are staged for the next database update. But remember that the database has **not** actually been updated.

In [7]:
```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
	new file:   README.md
	new file:   my_abs.py
```

Modifying a staged file

What happens when you've staged a file, but before you commit it, you realize you need to fix something? For example, in `README.md`, we made a typo. We wrote "Spring 2024" instead of "Summer 2024".

Let's correct `README.md` with our text editor, and then look at the `git status`:

In [9]:
```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
	new file:   README.md
	new file:   my_abs.py

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:   README.md
```

Git tells us:

- `README.md` and `my_abs.py` have changes that are staged ("to be committed")
- `README.md` also has changes that are not staged

How can one file have both staged and unstaged changes? The reason is that `git add` stages the state of the file **at the exact moment** you run `git add`. So if you run `git add`, and make additional changes afterwards (like changing "Spring" to "Summer"), then those additional changes are not automatically staged.

Staging new content (again)

To fix this, we'll run `git add` again to stage the new changes ("Spring" to "Summer"). You can see that Git suggests this, too, when it says: 'use "git add ..." to update what will be committed'

In [10]:
```
git add README.md
```

Now we see that all the changes in `README.md` have been staged, since there are no "Changes not staged for commit" anymore.

In [11]:
```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
        new file:   my_abs.py
```

`git commit`: Updating the database

Finally, we'll use `git commit` to record the previously-staged changes to the database.

You can use the `-m` option to specify a **commit message** on the command line. This is a short message that lets other humans know what changes you've made. Many projects have conventions about what info should go into a commit message. Ask your project manager for details.

(If you do not use `-m`, a text editor will pop up and prompt you to enter a message.)

In [12]:
```
git commit -m "First commit of my_abs (no try/except yet)"
```

```
[main (root-commit) 0565d47] First commit of my_abs (no try/except yet)
 2 files changed, 10 insertions(+)
 create mode 100644 README.md
 create mode 100644 my_abs.py
```

`git log`: Showing the repo's history

Now that we actually have information in our repo's database, we can use `git log` to show the history. `git log` has many options to show more or less information about the history. You can run `git help log` to see all the available options.

In [13]:
```
git log
```

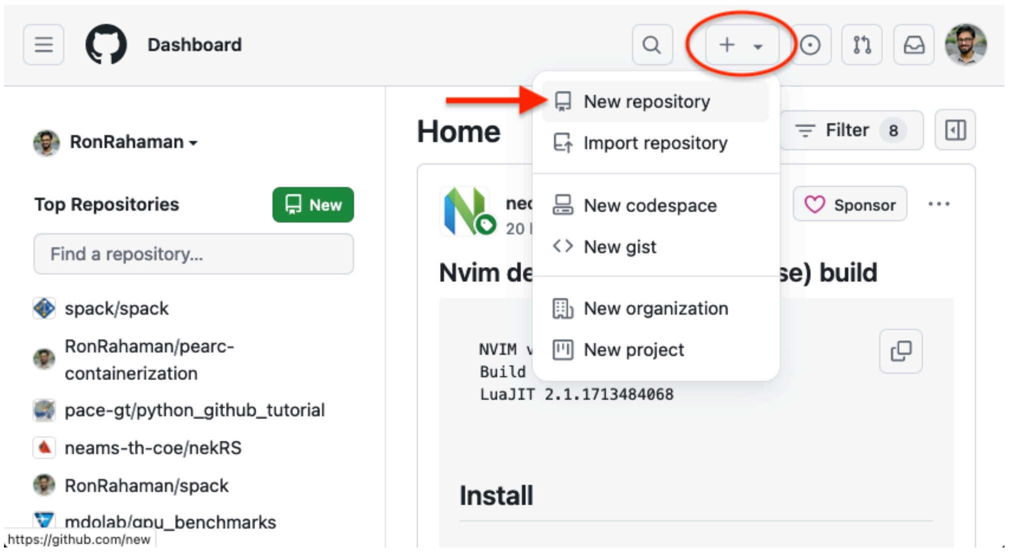
```
commit 0565d4790146c2efaa59041453e7c7c09baa3cc1 (HEAD -> main)
Author: George Burdell <gburdell@gmail.com>
Date:   Fri May 10 16:07:41 2024 -0500

    First commit of my_abs (no try/except yet)
```

4.2: Working with remotes

Up to this point, we've only made changes to the **local** repo on our computer. We haven't touched GitHub at all. Now, we are going to create a **remote** repo on GitHub and upload our local repo to it.

Creating a remote repo

From the front page of GitHub, you can create a new repo by clicking on the the "+" button on the top menu bar. You can also go directly to https://github.com/new

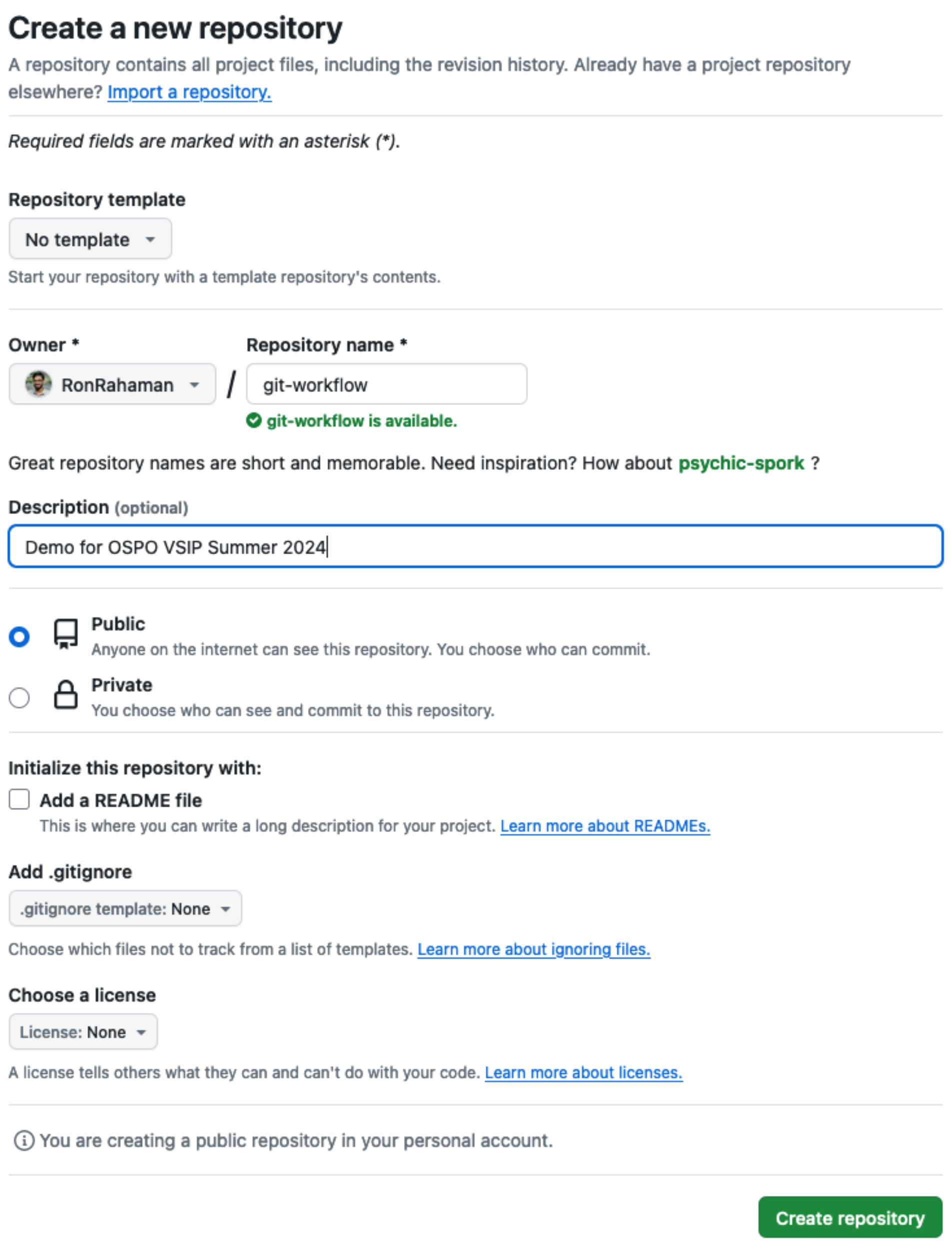


```
NVIM v
Build
LuaJIT 2.1.1713484068
```

You should now see the "Create a new repository" page. Since we are uploading an existing repository (instead of creating a new one), we'll only need a few options.

- **Repository template**: GitHub has starter templates (with directories and some boilerplate code) for common types of projects. We'll select "No template".
- **Owner**: The repo's owner can be an individual account or an organization (a group of accounts). A GitHub organization is a powerful tool for collaborating and managing permissions, and more likely than not, you'll be working in an organization for your VSIP project. For this tutorial, let's use your individual GitHub account.
- **Repository name**: This will be the last part of the repo's URL. When someone clones your repo, it will also be the default name for the project directory. Let's give it the same name as your existing project directory.
- **Visibility**: You can make your repo either Public or Private, whichever you prefer
- **Initialize this repository** and below: Since we are uploading an existing repo, we should not create a `README`, `.gitignore`, or license.

When you're finished, click the "Create repository" button



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Repository template

No template

Start your repository with a template repository's contents.

Owner * RonRahaman / **Repository name *** git-workflow

git-workflow is available.

Great repository names are short and memorable. Need inspiration? How about **psychic-spork** ?

Description (optional)

Demo for OSPO VSIP Summer 2024

[✓ **Public**] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Initialize this repository with:

☐ **Add a README file**
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license

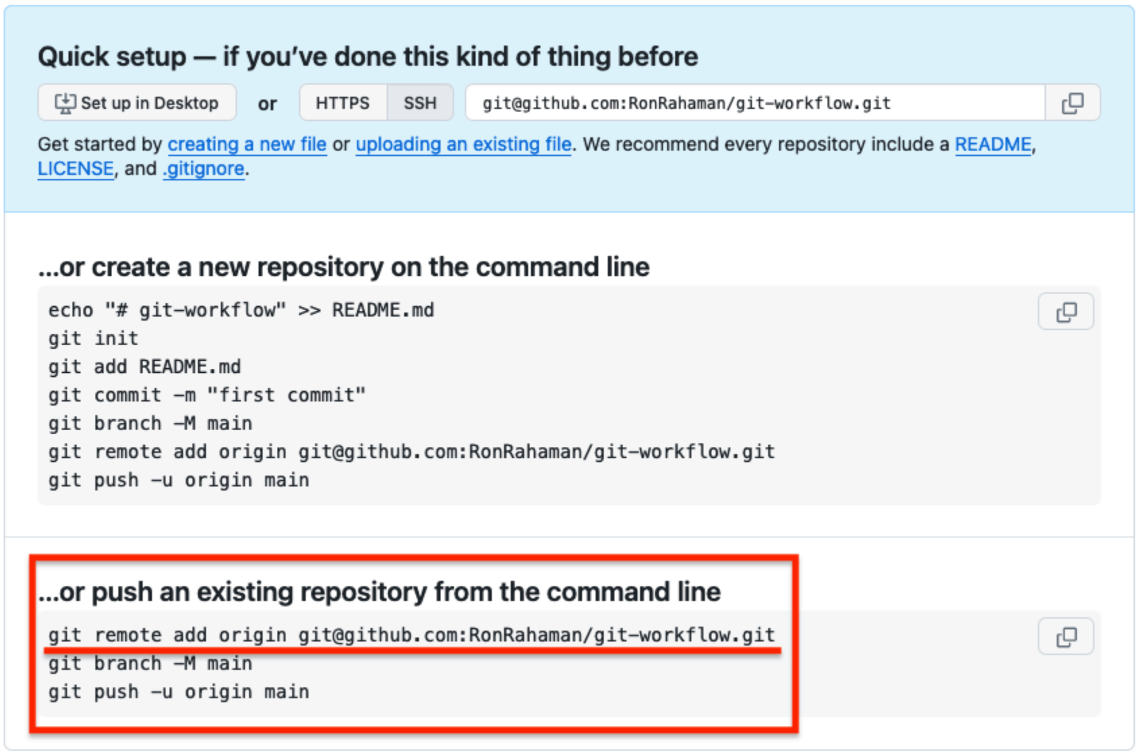
License: None

A license tells others what they can and can't do with your code. Learn more about licenses.

You are creating a public repository in your personal account.

You will now be taken to your repo's webpage, and you should see the following info. We're going to focus on the section "...or push an existing repo from the command line." In particular, we will focus on the `git remote add` command and the `git push` command.

(*The* `git branch -M main` *command is intended for legacy Git repos that would like to change their default branch name. Historically, Git's default branch name was "master". Starting around 2020, the Git community began efforts to change this naming convention to "main". This was widely embraced by stakeholders in the community, such as Git itself, GitHub, GitLab, and BitBucket. Today, in recent versions of Git, the default branch name is "main", so you will not need to explicitly change the default with* `git branch -M main`)


Quick setup — if you've done this kind of thing before

git@github.com:RonRahaman/git-workflow.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# git-workflow" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:RonRahaman/git-workflow.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin git@github.com:RonRahaman/git-workflow.git
git branch -M main
git push -u origin main
```

`git remote`: Add and manage remotes

The `git remote add` command lets you link your local repo to a remote repo. The syntax is:

```
git remote add <name> <url>
```

The `<url>` is the URL of your repo on GitHub. The `<name>` is a shorter identifier for that URL, which makes life easier for humans. By convention, humans use "origin" as the name for the default remote. When you are collaborating with a team and have multiple remotes (which is quite common!), there are naming conventions for other remotes (such as "upstream").

Here, we only have one remote, so we name it "origin". Make sure you use the URL from your own GitHub repo!

```
In [14]: git remote add origin git@github.com:GeorgeBurdell/git-workflow.git
```

Now you can use the `git remote -v` command to see which remotes are defined. You can see that this remote is setup for both fetching (downloading) and pushing (uploading).

```
In [15]: git remote -v
```

```
origin  git@github.com:GeorgeBurdell/git-workflow.git (fetch)
origin  git@github.com:GeorgeBurdell/git-workflow.git (push)
```

`git push`: Upload to the remote repo

The `git push` command is used to upload changes from your local repo to a defined remote repo. The synax is:

```
git push <remote> <branch>
```

A useful first-time option is `-u`. It sets `<remote>` as the default remote for `<branch>`. We'll use that below. Th effect is that, in subsequent pushes, we can use `git push` without any extra arguments to push `main` to `origin`

In [17]:
```
git push -u origin main
```

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 408 bytes | 408.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:GeorgeBurdell/git-workflow.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Back to GitHub

Now you can go back to your repo's webpage on GitHub and see your repo's contents! You might need to click the refresh button in your browser to see the updates.

4.3: Downloading a repo

`git clone`: Download a repo

Now that your GitHub repo is populated, let's try downloading (or **cloning**) it somewhere else. It's common to clone your repo on multiple computers depending on where you're working (like your laptop, a remote server, etc.). For this demo, we'll just create another directory in `~/somewhere-else`

(You can use a different name or location, if you like)

In [18]:
```
mkdir ~/somewhere-else
cd ~/somewhere-else
```

Now we can use the `git clone` command to downlaod the repo. The syntax is:

```
git clone <url>
```

The URL is the same one you used with `git remote add`. This will create a working copy of the local repo in a new directory. The directory's name is the last part of the URL. Afterwards, we can enter the new working copy and see that our files are present. We can also verify that our repo's history is present.

In [19]:
```
git clone git@github.com:GeorgeBurdell/git-workflow.git
```

```
Cloning into 'git-workflow'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

In [20]:
```
cd ~/somewhere-else/git-workflow
```

In [21]:
```
ls -la
```

```
total 16
drwxr-xr-x   5 gburdell  staff  160 May 10 16:09 .
drwxr-xr-x   3 gburdell  staff   96 May 10 16:09 ..
drwxr-xr-x  12 gburdell  staff  384 May 10 16:09 .git
-rw-r--r--   1 gburdell  staff   85 May 10 16:09 README.md
-rw-r--r--   1 gburdell  staff   75 May 10 16:09 my_abs.py
```

In [22]:
```
git log
```

```
commit 0565d4790146c2efaa59041453e7c7c09baa3cc1 (HEAD -> main, origin/main,
origin/HEAD)
Author: George Burdell <gburdell@gmail.com>
Date:   Fri May 10 16:07:41 2024 -0500

    First commit of my_abs (no try/except yet)
```

4.4: Adding more changes

Back to our original working copy

Let's return to our original working copy and make some more changes

In [23]:
```
cd ~/git-workflow
```

Making more changes

Let's add some new features to `my_abs.py`. Copy/paste this into my_abs.py

```
import math

def my_abs(x):
    try:
        if x < 0:
            return -x
        else:
            return x
    except TypeError:
        return math.nan
```

`git diff`: Show changes in modified files

At this point, `my_abs.py` is the in the "modified" state. It has changes that have not yet been staged. It can be very useful to see changes before you stage and commit. `git diff` lets you do exactly that.

The appearance of a "diff" is the same in many places. GitHub and your IDE will use a very similar appearance when showing you differences in files. Hence, understanding diffs is a key skill for understanding your code's history. The meaning is:

- Lines that are green and/or begin with a "+" have been added to the newer version but are not present in the older vesion.
- Lines that are red and/or begin with a "-" are present in the older version but have been omitted in the newer version.
- Lines that are not colored are the same in both versions

In [25]:
```
git diff
```

```
diff --git a/my_abs.py b/my_abs.py
index fc89810..64431e6 100644
--- a/my_abs.py
+++ b/my_abs.py
@@ -1,6 +1,11 @@
+import math
+
 def my_abs(x):
-    if x < 0:
-        return -x
-    else:
-        return x
+    try:
+        if x < 0:
+            return -x
+        else:
+            return x
+    except TypeError:
+        return math.nan
```

Staging and committing

As we mentioned, `my_abs.py` now has unstaged changes. We can stage and commit them in one shot with the `-a` flag. This will stage and commit changes to all currently **tracked** files.

After committing, we can verify that there is a new commit in our history. We'll use some additional options to show a more condensed log. Logs are, by default, shown from most to least recent.

```
In [26]: git status
```

```
On branch main
Your branch is up to date with 'origin/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:   my_abs.py

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

```
In [27]: git commit -am "Added try/except"
```

```
[main faff1d3] Added try/except
 1 file changed, 9 insertions(+), 4 deletions(-)
```

```
In [28]: git log --oneline --graph --branches --remotes
```

```
* faff1d3 (HEAD -> main) Added try/except
* 0565d47 (origin/main) First commit of my_abs (no try/except yet)
```

Pushing to remote

Finally, we'll push our updates to GitHub. Go back to your GitHub webpage and see your updates! (You might have to click refresh again.)

```
In [29]: git push
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 375 bytes | 375.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:GeorgeBurdell/git-workflow.git
   0565d47..faff1d3  main -> main
```

4.5: Getting new changes from the remote

Like most things in Git, you have to be extremely explicit about updating your working copies. If you push changes from one working copy, then other working copies are **not** automatically updated. You must explicitly download the new changes in the other working copies.

What's happening in our other working copy?

Let's go back to other working copy in `~/somewhere-else/git-workflow`. The log shows that the working copy does *not* have the changes that we just pushed. And if we look at `my_abs.py`, we'll see it's still at the older version.

In [30]:
```
cd ~/somewhere-else/git-workflow
```

In [31]:
```
git log
```

```
commit 0565d4790146c2efaa59041453e7c7c09baa3cc1 (HEAD -> main, origin/main,
origin/HEAD)
Author: George Burdell <gburdell@gmail.com>
Date:   Fri May 10 16:07:41 2024 -0500

    First commit of my_abs (no try/except yet)
```

In [32]:
```
cat my_abs.py
```

```
def my_abs(x):
    if x < 0:
        return -x
    else:
        return x
```

`git pull`: Download changes from the remote

We will bring this working copy up-to-date with `git pull`. The syntax is:

```
git pull [<remote> <branch_name>]
```

If the branch has a default remote, then we can omit the extra arguments and just run `git pull`. Now when we look at the log and the contents of `my_abs.py`, we see that everything's up-to-date with the remote.

In [33]:
```
git pull
```

```
hint: Pulling without specifying how to reconcile divergent branches is
hint: discouraged. You can squelch this message by running one of the follow
ing
hint: commands sometime before your next pull:
hint:
hint:   git config pull.rebase false  # merge (the default strategy)
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a defau
lt
hint: preference for all repositories. You can also pass --rebase, --no-reba
se,
hint: or --ff-only on the command line to override the configured default pe
r
hint: invocation.
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 355 bytes | 50.00 KiB/s, done.
From github.com:GeorgeBurdell/git-workflow
   0565d47..faff1d3  main       -> origin/main
Updating 0565d47..faff1d3
Fast-forward
 my_abs.py | 13 +++++++++----
 1 file changed, 9 insertions(+), 4 deletions(-)
```

In [34]:
```
git log
```

```
commit faff1d33243766a5a157d112c9a873db3459caad (HEAD -> main, origin/main,
origin/HEAD)
Author: George Burdell <gburdell@gmail.com>
Date:   Fri May 10 16:10:13 2024 -0500

    Added try/except

commit 0565d4790146c2efaa59041453e7c7c09baa3cc1
Author: George Burdell <gburdell@gmail.com>
Date:   Fri May 10 16:07:41 2024 -0500

    First commit of my_abs (no try/except yet)
```

In [35]:
```
cat my_abs.py
```

```
import math

def my_abs(x):
    try:
        if x < 0:
            return -x
        else:
            return x
    except TypeError:
        return math.nan
```

Back to our first working copy

We'll be working with our first working copy in the next section, so let's make sure to move back:

```
In [36]: cd ~/git-workflow
```

5: Branches

At this point, we've been discussing one branch, "main". Now we will consider additional branches, and we'll see how branches can help you in collaborative workflows.

5.1: About Git branches

Git branches are pointers

It's common to think about Git branches like the branches of a family tree. However, it's more useful to remember that **a Git branch is a pointer to a commit**.

In older VCSs, a "branch" was implemented as a complete copy of the files. This used a large amount of space in the database, and hence, branches were a less common part of workflows.

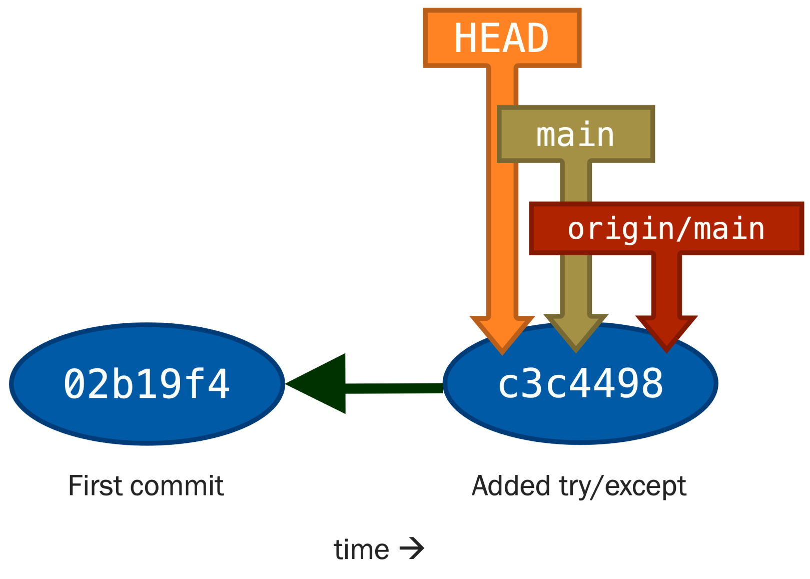
In contrast, Git branches are fast and cheap with very little storage or performance overhead. They are an integral part of any Git workflow.

Our branches

In fact, we already have multiple branches. If you look at `git log` (make sure you've moved back to `~/git-workflow` !) you'll see:

- **HEAD** is a special branch that points to the current state of our working tree.

- **main** is the branch we created when we initialized the repo.
- **origin/main** is a remote-tracking branch. It reflects the state of our remote repo "origin" on GitHub. More on this soon...



5.2: Checking-out changes

`git checkout` : Changing the files in the working tree

Suppose you want to work with the files from a different point in time. We use `git checkout`, which has the syntax:

```
git checkout <branch | commit>
```

First let's checkout the files from our first commit. Every commit is uniquely identified by an alphanumeric hash, and we can use this to specify a commit in `git checkout` and other commands.

In the repo history from this demo, our first commit has the hash ID `0565d4790146c2efaa59041453e7c7c09baa3cc1`. You can use abbreviated hashes (as long as the abbreviation still uniquely identifies a commit), and it's common to use the first 7 characters. So we will use `0565d47`.

(*Your repo will almost certainly have different hashes, so as you're following along, be sure to use your repo's hash.*)

In [37]:
```
git checkout 0565d47
```

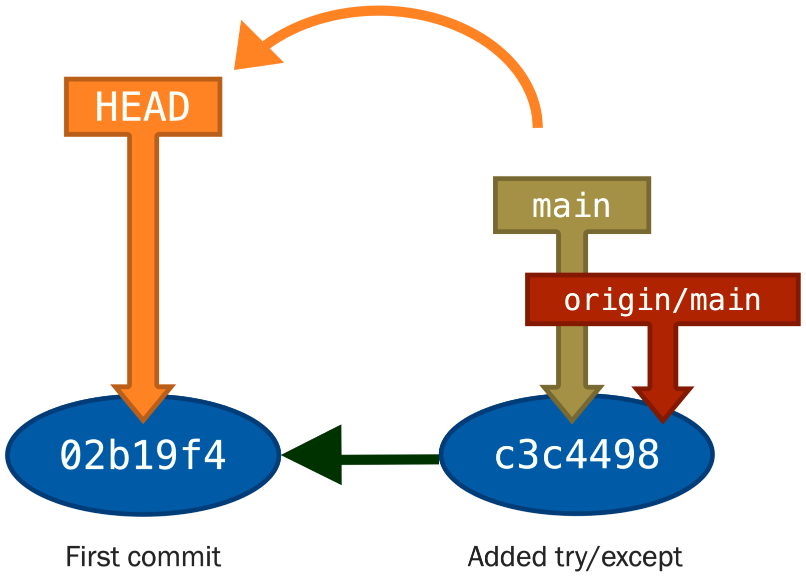
```
Note: switching to '0565d47'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 0565d47 First commit of my_abs (no try/except yet)
```

Results of checkout

When we look at the log, we can see that the HEAD has moved back to "First commit". Likewise, `my_abs.py` in the working tree is at the state of "First commit" -- it does not have the try/except . However, note that the branches "main" and "origin/main" are unchanged.

In [38]:
```
git log --oneline --graph --branches --remotes
```

```
* faff1d3 (origin/main, main) Added try/except
* 0565d47 (HEAD) First commit of my_abs (no try/except yet)
```



In [39]:
```
cat my_abs.py
```

```
def my_abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Detached HEAD

What about the scary message about the detached HEAD? This is perfectly normal and common! All it means is that HEAD is not pointing to the same commit as any other branch. If you're exploring your repo's history, you will encounter this message often. You can suppress this message with:

```
git config --global advice.detachedHead false
```

Or you can learn to ignore it :)

Checking out a branch

You can also specify a branch name to `git commit`. This is probably the more common usage.

Let's return our working tree to the state of "main". You will notice that:

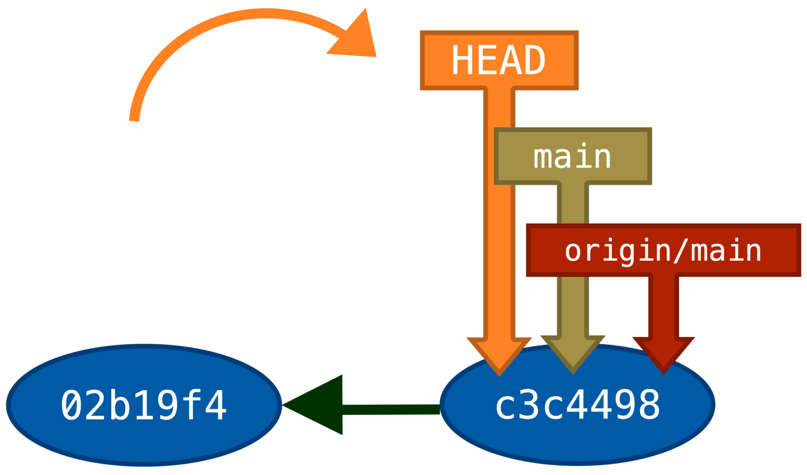
- The HEAD moves back to "main"
- The state of the working tree is also back to "main". Notices that `my_abs.py` has the try/except as expected.

In [40]:
```
git checkout main
```

```
Previous HEAD position was 0565d47 First commit of my_abs (no try/except ye
t)
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

In [41]:
```
git log --oneline --graph --branches --remotes
```

```
* faff1d3 (HEAD -> main, origin/main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



First commit

Added try/except

```
In [42]: cat my_abs.py
```

```
import math

def my_abs(x):
    try:
        if x < 0:
            return -x
        else:
            return x
    except TypeError:
        return math.nan
```

5.3: Using topic branches

The why and what of topic branches

When you are collaborating on a large code base, **topic branches** (or **feature branches**) can be your team's best friend.

The idea is that, when you want to develop a new feature or fix a bug, you create a branch specifically for that purpose (or "topic"). A topic branch has a very specific purpose and (ideally) touches a limited part of the code base. It often has a limited lifetime, too; when its changes are merged into the main branch, the topic branch is usally deleted.

This has a number of technical benefits, but the greatest benefits are organizational:

- Multiple developers can work on the multiple, independent features at the same time without stepping on each others' toes
- Maintainers can more-easily understand and approve changes
- Topic branches can correspond to tasks or cards in a project management system

`git checkout -b`: Create a new branch

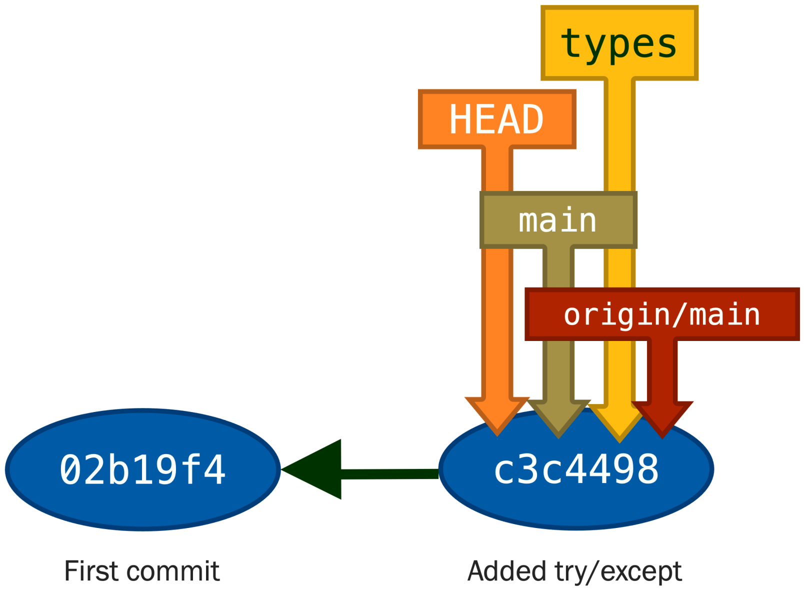
Let's create a topic branch to implement some type-specific behavior in `my_abs`. We will call this branch `types`. We use the command `git checkout -b <branch_name>` to create a new branch at HEAD's current location.

(*Many projects have naming conventions for topic branches. Please consult your project maintainers for details.*)

Notice that by creating the branch, we have just created a new pointer to a commit. No files have been copied or changed, and no new commits have been made.

In [43]:
```
git checkout -b types
```

```
Switched to a new branch 'types'
```



In [44]:
```
git log --oneline --graph --branches --remotes
```

```
* faff1d3 (HEAD -> types, origin/main, main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```

Making new changes

Now we are ready to implement our new feature. Copy this to `my_abs.py`

```
import math
import numbers

def my_abs(x):
    if isinstance(x, numbers.Real):
        if x < 0:
            return -x
        else:
            return x
    elif isinstance(x, numbers.Complex):
        return math.sqrt(
            x.real ** 2 + x.imag ** 2)
    else:
        return math.nan
```

In [45]:
```
git diff
```

```
diff --git a/my_abs.py b/my_abs.py
index 64431e6..bf90a6f 100644
--- a/my_abs.py
+++ b/my_abs.py
@@ -1,11 +1,15 @@
 import math
+import numbers

 def my_abs(x):
-    try:
+    if isinstance(x, numbers.Real):
         if x < 0:
             return -x
         else:
             return x
-    except TypeError:
+    elif isinstance(x, numbers.Complex):
+        return math.sqrt(
+            x.real ** 2 + x.imag ** 2)
+    else:
         return math.nan
```

Results of commit

Now let's commit our changes and observe the results:

- A new commit has been created, and both HEAD and "types" have moved forward.
- "main" and "origin/main" stay put

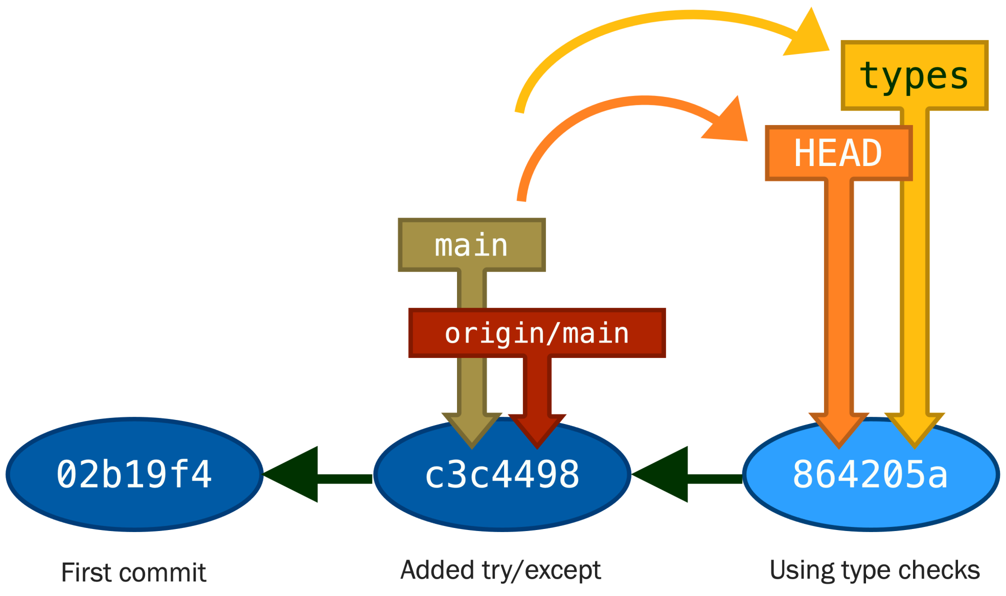
The effect is that your new feature is on "types", while "main" remains as a stable version.

In [46]:
```
git commit -am "Using type checks"
```

```
[types d82a6ba] Using type checks
 1 file changed, 6 insertions(+), 2 deletions(-)
```

In [47]:
```
git log --oneline --graph --branches --remotes
```

```
* d82a6ba (HEAD -> types) Using type checks
* faff1d3 (origin/main, main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



Making another topic branch

Now suppose we want to introduce a separate feature. We're busy people working on multiple things. We might have a lot of ideas, and our projects have a lot of tasks to complete.

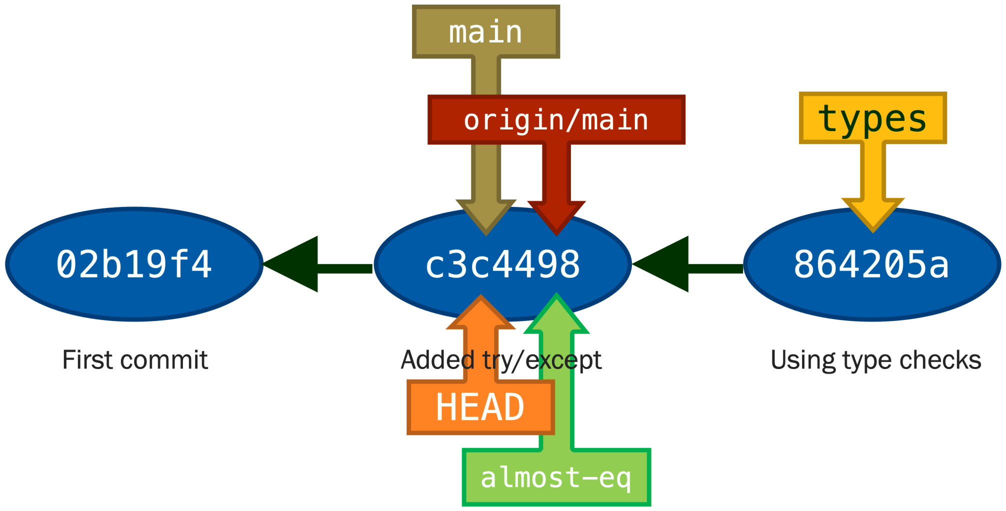
If we're pursuing a separate feature, we will usually create **a separate topic branch**. And in many cases, we will want to create the topic branch **from the stable version in "main"** (instead of another topic branch). This keeps our tasks isolated and makes task management much easier.

Let's suppose our new feature is a function to compare if two numbers are almost equal to each other. We'll call the topic branch "almost-eq". Our steps are:

1. Checkout "main". If we want to branch off of "main", we need to checkout "main" first.
2. Create the topic branch "almost-eq".

In [48]:
```
git checkout main
```
```
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

In [49]:
```
git checkout -b almost-eq
```
```
Switched to a new branch 'almost-eq'
```



Implementing `my_almost_eq`

Our implementation of `my_almost_eq` will use the existing implementation of `my_abs`. Notice that `my_almost_eq` does not depend on the actual implementation of `my_abs`; it just depends on the input and output (the interface) of `my_abs`. This means that someone else can make changes to `my_abs` on another topic branch, and as long as they don't change the interface, then `my_almost_eq` and any other functions that use `my_abs` will still work.

Copy this to `my_abs.py`, and note that `my_abs` is not different from "main".

```
import math

def my_abs(x):
    try:
        if x < 0:
            return -x
        else:
            return x
    except TypeError:
        return math.nan

def my_almost_eq(x, y):
    return my_abs(x - y) < 1e-16
```

In [50]:
```
git diff
```

```
diff --git a/my_abs.py b/my_abs.py
index 64431e6..efa815d 100644
--- a/my_abs.py
+++ b/my_abs.py
@@ -9,3 +9,6 @@ def my_abs(x):
     except TypeError:
         return math.nan

+def my_almost_eq(x, y):
+    return my_abs(x - y) < 1e-16
+
```

Commtting changes to "almost-eq"

Now we are ready to commit our changes. The effects are:

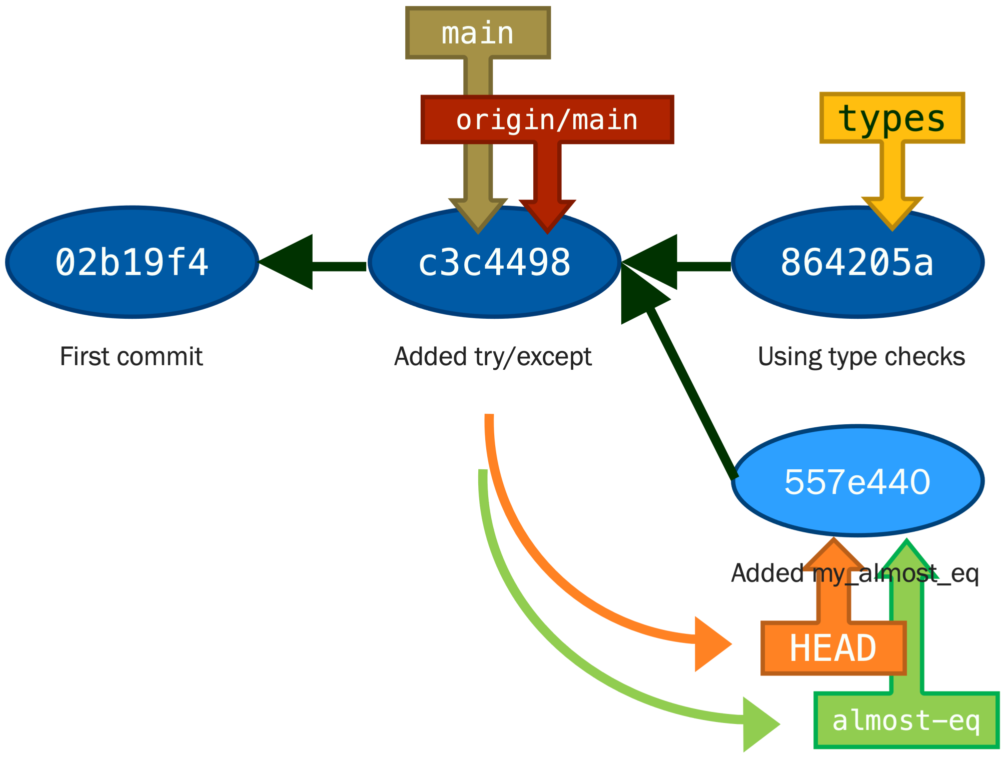
1. A new commit is created
2. HEAD and "almost-eq" move forward
3. "types", "main", and "origin/main" stay put

In [51]:
```
git commit -am "Added my_almost_eq"
```

```
[almost-eq 44a310e] Added my_almost_eq
 1 file changed, 3 insertions(+)
```

In [52]:
```
git log --oneline --graph --branches --remotes
```

```
* 44a310e (HEAD -> almost-eq) Added my_almost_eq
| * d82a6ba (types) Using type checks
|/
* faff1d3 (origin/main, main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



5.4: Merging branches

When a topic branch is ready to be added to the main branch, it needs to **merged** into main.

In a local repo, you would use the `git merge` command. We will use it for this demo and study how merges are resolved.

In most collaborative workflows, merges happen on the remote repo (i.e., GitHub). The main advantage is that GitHub has excellent support for well-defined approval processes through Pull Requests (see upcoming Module 2!). However, on the backend, the Pull Request is ultimately resolved through a `git merge`. And understanding how merges are resolved can be vital for making effective Pull Requests.

Case 1: A fast-forward merge

First, suppose we like the changes in "types" and are ready to merge them into "main". We will use the command:

```
git merge <branch_name>
```

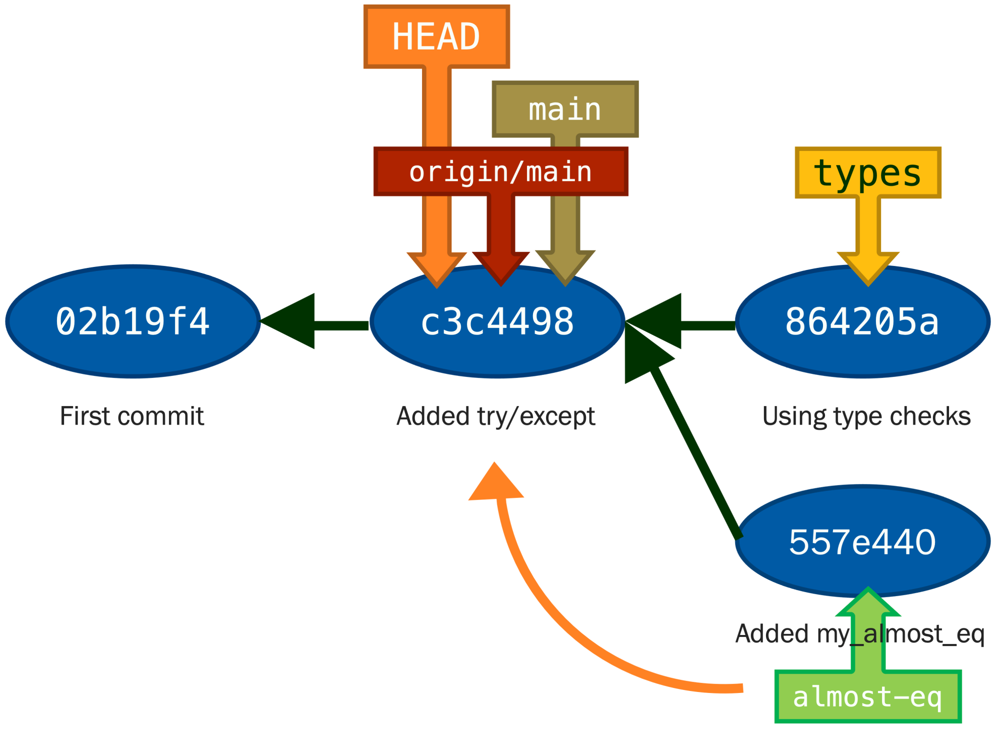
which will merge the specified branch into the currenly checked-out branch. Hence, to merge "types" into "main", we first need to checkout "main". Note how HEAD moves back to "main".

```
In [53]: git checkout main
```

```
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

```
In [54]: git log --oneline --graph --branches --remotes
```

```
* 44a310e (almost-eq) Added my_almost_eq
| * d82a6ba (types) Using type checks
|/
* faff1d3 (HEAD -> main, origin/main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



Now we use `git merge types` to merge "types" into main. In this case, "types" is a direct descendent of "main", so the merge can be resolved simply by moving "main" (and HEAD) forward to "types. This is a case where branches very clearly act like pointers: the merge doesn't create any new commits, and it simply moves the pointer for "main" (and HEAD) forward.

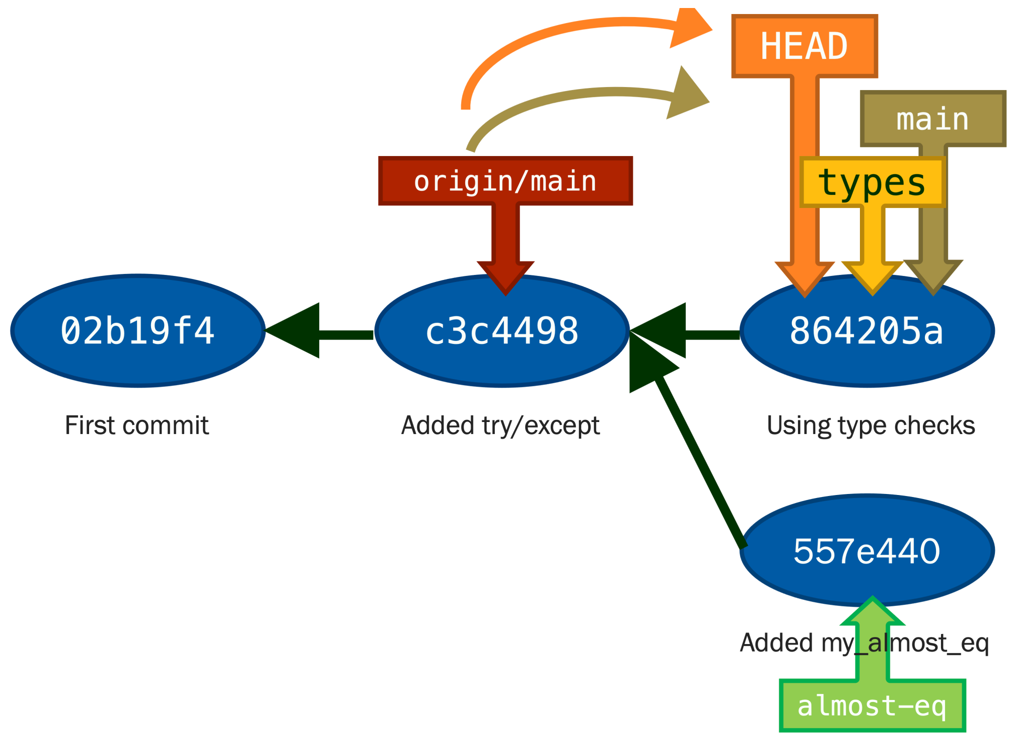
This is called a **fast forward** merge and is the simplest, most unambiguous kind of merge.

In [55]:
```
git merge types
```

```
Updating faff1d3..d82a6ba
Fast-forward
 my_abs.py | 8 ++++++--
 1 file changed, 6 insertions(+), 2 deletions(-)
```

In [56]:
```
git log --oneline --graph --branches --remotes
```

```
* 44a310e (almost-eq) Added my_almost_eq
| * d82a6ba (HEAD -> main, types) Using type checks
|/
* faff1d3 (origin/main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



Case 2: A merge commit

Now let's consider a case that cannot be resolved with a fast-forward merge.

Suppose that we now want to merge "almost-eq" into "main". Unlike before, "almost-eq" is **not** a direct descendant of "main". If Git simply moved the pointer for "main" to the same commit as "almost-eq", then all the changes from the previous merge would be left behind.

Therefore, to resolve this merge, Git will create a **new commit** with changes from both "main" and "almost-eq". The syntax is the same as before (`git merge almost-eq`), and git will analyze the graph of the history to determine whether to do a fast-forward or other merge.

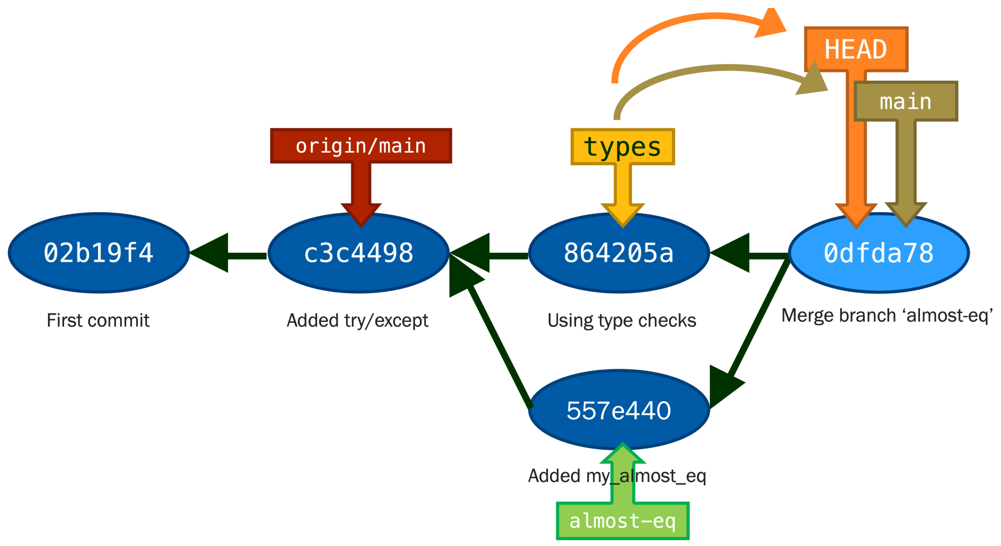
(*In this Jupyter Notebook, I am using the* `--no-edit` *flag to prevent a text editor from popping up. This will use a default message, "Merge branch 'almost-eq'". If you are working directly in the terminal, the* `--no-edit` *flag is optional.*)

In [57]:
```
git merge --no-edit almost-eq
```

```
Auto-merging my_abs.py
Merge made by the 'recursive' strategy.
 my_abs.py | 3 +++
 1 file changed, 3 insertions(+)
```

In [58]:
```
git log --oneline --graph --branches --remotes
```

```
*   113d61f (HEAD -> main) Merge branch 'almost-eq'
|\
| * 44a310e (almost-eq) Added my_almost_eq
* | d82a6ba (types) Using type checks
|/
* faff1d3 (origin/main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



Examining results of the merge

Let's look at the final results. Now "main" contains:

- The implementation of `my_abs` with type-specific behavior (from the branch "types")
- The function `my_almost_eq` (from "almost-eq")

The merges went smoothly because our code was well-modularized and the topic branches did not touch the same parts of the code. This is another reason why well-designed code and coordinated task management can help your collaborations go smoothly

In [59]:
```
cat my_abs.py
```

```
import math
import numbers

def my_abs(x):
    if isinstance(x, numbers.Real):
        if x < 0:
            return -x
        else:
            return x
    elif isinstance(x, numbers.Complex):
        return math.sqrt(
            x.real ** 2 + x.imag ** 2)
    else:
        return math.nan

def my_almost_eq(x, y):
    return my_abs(x - y) < 1e-16
```

5.5: Branches and remotes

Remote-tracking branches

Now it's finally time to explain "origin/main". This is a **remote-tracking branch**, which is a local branch that indicates the state of a remote branch. In this case, "origin/main" indicates where the branch "main" is on the remote "origin". And indeed, if you go to your repo on GitHub, you can confirm that the state of "main" is still at the same place as "origin/main".

Let's do a `git push` and see what happens to both the remote-tracking branch and GitHub. After the push:

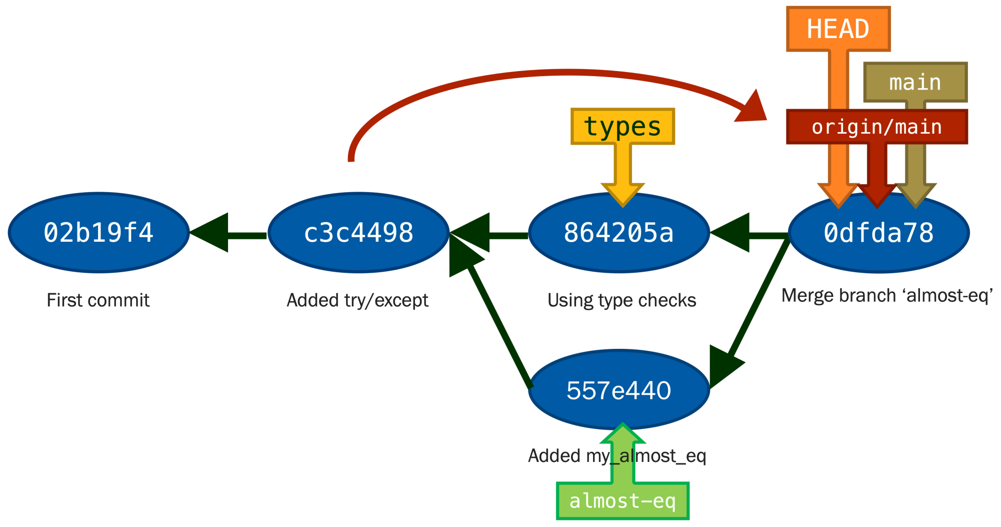
- GitHub has the latest changes from main
- The remote-tracking branch "origin/main" has been moved forward to reflect the new state of the remote.

In [60]:
```
git push
```

```
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 12 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1020 bytes | 510.00 KiB/s, done.
Total 9 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
To github.com:GeorgeBurdell/git-workflow.git
   faff1d3..113d61f  main -> main
```

In [61]:
```
git log --oneline --graph --branches --remotes
```

```
*   113d61f (HEAD -> main, origin/main) Merge branch 'almost-eq'
|\
| * 44a310e (almost-eq) Added my_almost_eq
* | d82a6ba (types) Using type checks
|/
* faff1d3 Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



When the remote-tracking branch seems wrong

The very tricky thing about remote-tracking branches is: **they are not automatically updated**. So, if you push from one working copy, but go to another older working copy, then the remote-tracking branch on the latter will be **out-of-date**. In the older working copy, you must explicitly **fetch** the latest changes from the remote to update the remote-tracking branch.

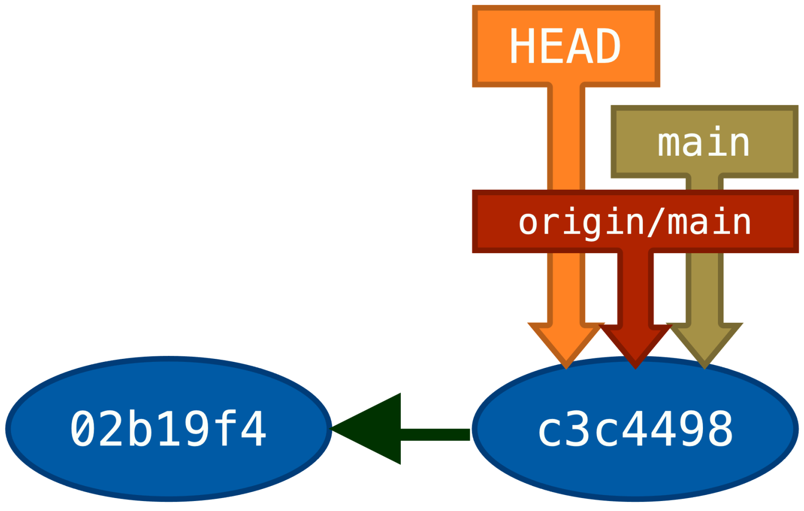
Let's go back to our other working copy in `~/somewhere-else/git-workflow`. If we look at the log, we can see that:

- "main" is not up-to-date
- "origin/main" is not up-to-date either!

In [62]:
```
cd ~/somewhere-else/git-workflow
```

In [63]:
```
git log --oneline --graph --branches --remotes
```

```
* faff1d3 (HEAD -> main, origin/main, origin/HEAD) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



First commit

Added try/except

```
In [64]: cat my_abs.py
```

```
import math

def my_abs(x):
    try:
        if x < 0:
            return -x
        else:
            return x
    except TypeError:
        return math.nan
```

`git fetch`: Update the remote tracking branch

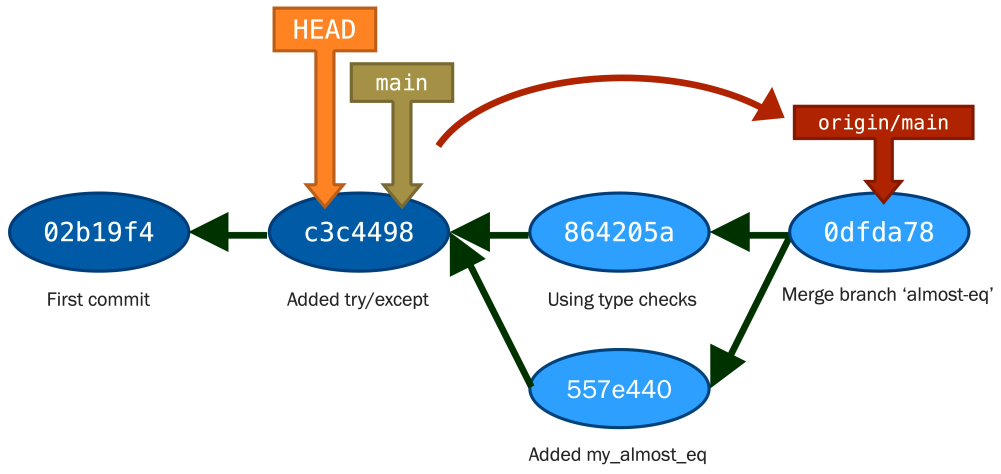
We'll get the new changes in two steps. First, we'll use `git fetch` to update "origin/main". This downloads new commits from the remote and moves the remote-tracking branch forward. It does **not** change the state of the local branch "main".

```
In [65]: git fetch
```

```
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 3), reused 9 (delta 3), pack-reused 0
Unpacking objects: 100% (9/9), 1000 bytes | 55.00 KiB/s, done.
From github.com:GeorgeBurdell/git-workflow
   faff1d3..113d61f  main       -> origin/main
```

```
In [66]: git log --oneline --graph --branches --remotes
```

```
*   113d61f (origin/main, origin/HEAD) Merge branch 'almost-eq'
|\
| * 44a310e Added my_almost_eq
* | d82a6ba Using type checks
|/
* faff1d3 (HEAD -> main) Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



`git merge`: Merge remote-tracking branch into local branch

In many ways, remote tracking branches can be treated like other branches. In particular, we can use them in merges.

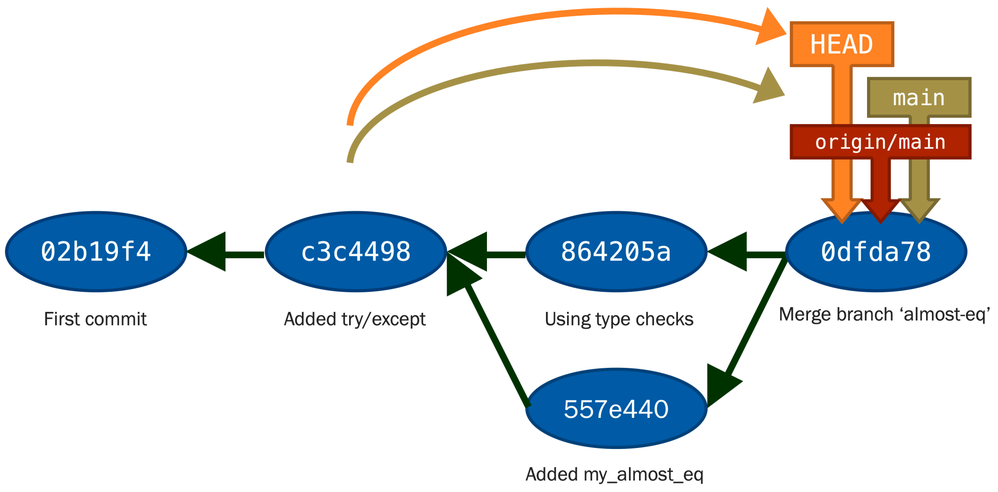
So, to bring "main" up-to-date with "origin/main", we will use `git merge origin/main`. Since "origin/main" is a direct descendent of "main", this will be resolved as a fast-forward merge.

In [67]:
```
git merge origin/main
```

```
Updating faff1d3..113d61f
Fast-forward
 my_abs.py | 11 +++++++++--
 1 file changed, 9 insertions(+), 2 deletions(-)
```

In [68]:
```
git log --oneline --graph --branches --remotes
```

```
*   113d61f (HEAD -> main, origin/main, origin/HEAD) Merge branch 'almost-e
q'
|\
| * 44a310e Added my_almost_eq
* | d82a6ba Using type checks
|/
* faff1d3 Added try/except
* 0565d47 First commit of my_abs (no try/except yet)
```



```
In [69]: cat my_abs.py
```

```
import math
import numbers

def my_abs(x):
    if isinstance(x, numbers.Real):
        if x < 0:
            return -x
        else:
            return x
    elif isinstance(x, numbers.Complex):
        return math.sqrt(
            x.real ** 2 + x.imag ** 2)
    else:
        return math.nan

def my_almost_eq(x, y):
    return my_abs(x - y) < 1e-16
```

`git pull` revisited

When we updated "main" just now, why didn't we use `git pull` like before? We could've! In fact, a `git pull` is just a fetch followed by a merge. This time, I wanted to illustrate what was happening with the remote-tracking branches, so I used two explicit steps.

The vast majority of the time, you'll be doing a `pull` instead of separate `fetch` and `merge`. But sometimes -- like if your local history is messy -- it can be helpful to do a fetch and see what's happening before you merge.