D

USING GIT FOR VERSION CONTROL

Version control software allows you to take snapshots of a project whenever it's in a working state. When you make changes to a project—for example, when you implement a new feature—you can revert back to a previous working state if the project's current state isn't functioning well.

Using version control software gives you the freedom to work on improvements and make mistakes without worrying about ruining your project. This is especially critical in large projects, but can also be helpful in smaller projects, even when you're working on programs contained in a single file.

In this appendix, you'll learn to install Git and use it for version control in the programs you're working on now. Git is the most popular version control software in use today. Many of its advanced tools help teams collaborate on large projects, but its most basic features also work well for solo developers. Git implements version control by tracking the changes made to every file in a project; if you make a mistake, you can just return to a previously saved state.

Installing Git

Git runs on all operating systems, but there are different approaches to installing it on each system. The following sections provide specific instructions for each operating system.

Installing Git on Windows

You can download an installer for Git at *https://git-scm.com/*. You should see a download link for an installer that's appropriate for your system.

Installing Git on macOS

Git might already be installed on your system, so try issuing the command `git --version`. If you see output listing a specific version number, Git is installed on your system. If you see a message prompting you to install or update Git, simply follow the onscreen directions.

You can also visit *https://git-scm.com/*, where you should see a download link for an appropriate installer for your system.

Installing Git on Linux

To install Git on Linux, enter the following command:

```
$ sudo apt install git-all
```

That's it. You can now use Git in your projects.

Configuring Git

Git keeps track of who makes changes to a project, even when only one person is working on the project. To do this, Git needs to know your username and email. You must provide a username, but you can make up a fake email address:

```
$ git config --global user.name "username"
$ git config --global user.email "username@example.com"
```

If you forget this step, Git will prompt you for this information when you make your first commit.

Making a Project

Let's make a project to work with. Create a folder somewhere on your system called *git_practice*. Inside the folder, make a simple Python program:

hello_git.py

```
print("Hello Git world!")
```

We'll use this program to explore Git's basic functionality.

Ignoring Files

Files with the extension *.pyc* are automatically generated from *.py* files, so we don't need Git to keep track of them. These files are stored in a directory called *__pycache__*. To tell Git to ignore this directory, make a special file called *.gitignore*—with a dot at the beginning of the filename and no file extension—and add the following line to it:

.gitignore

```
__pycache__/
```

This file tells Git to ignore any file in the *__pycache__* directory. Using a *.gitignore* file will keep your project clutter free and easier to work with.

You might need to modify your text editor's settings so it will show hidden files in order to open *.gitignore*. Some editors are set to ignore filenames that begin with a dot.

Initializing a Repository

Now that you have a directory containing a Python file and a *.gitignore* file, you can initialize a Git repository. Open a terminal, navigate to the *git_practice* folder, and run the following command:

```
git_practice$ git init
Initialized empty Git repository in git_practice/.git/
git_practice$
```

The output shows that Git has initialized an empty repository in *git_practice*. A *repository* is the set of files in a program that Git is actively tracking. All the files Git uses to manage the repository are located in the hidden directory *.git*, which you won't need to work with at all. Just don't delete that directory, or you'll lose your project's history.

Checking the Status

Before doing anything else, let's look at the project's status:

```
git_practice$ git status
❶ On branch master

No commits yet

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

  .gitignore
  hello_git.py

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

In Git, a *branch* is a version of the project you're working on; here you can see that we're on a branch named `master` ❶. Each time you check your project's status, it should show that you're on the branch `master`. You then see that we're about to make the initial commit. A *commit* is a snapshot of the project at a particular point in time.

Git informs us that untracked files are in the project ❷, because we haven't told it which files to track yet. Then we're told that there's nothing added to the current commit, but untracked files are present that we might want to add to the repository ❸.

Adding Files to the Repository

Let's add the two files to the repository, and check the status again:

```
❶ git_practice$ git add .
❷ git_practice$ git status
  On branch master

  No commits yet

  Changes to be committed:
    (use "git rm --cached <file>..." to unstage)

❸   new file:   .gitignore
    new file:   hello_git.py

  git_practice$
```

The command `git add .` adds all files within a project that aren't already being tracked to the repository ❶. It doesn't commit the files; it just tells Git to start paying attention to them. When we check the status of the project now, we can see that Git recognizes some changes that need to be committed ❷. The label *new file* means these files were newly added to the repository ❸.

Making a Commit

Let's make the first commit:

```
❶ git_practice$ git commit -m "Started project."
❷ [master (root-commit) ee76419] Started project.
❸  2 files changed, 4 insertions(+)
    create mode 100644 .gitignore
    create mode 100644 hello_git.py
❹ git_practice$ git status
  On branch master
```

```
  nothing to commit, working tree clean
git_practice$
```

We issue the command `git commit -m "message"` ❶ to take a snapshot of the project. The `-m` flag tells Git to record the message that follows (`"Started project."`) in the project's log. The output shows that we're on the `master` branch ❷ and that two files have changed ❸.

When we check the status now, we can see that we're on the `master` branch, and we have a clean working tree ❹. This is the message you want to see each time you commit a working state of your project. If you get a different message, read it carefully; it's likely you forgot to add a file before making a commit.

Checking the Log

Git keeps a log of all commits made to the project. Let's check the log:

```
git_practice$ git log
commit a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f (HEAD -> master)
Author: Eric Matthes <eric@example.com>
Date:   Mon Jan 21 21:24:28 2019 -0900

  Started project.
git_practice$
```

Each time you make a commit, Git generates a unique, 40-character reference ID. It records who made the commit, when it was made, and the message recorded. You won't always need all of this information, so Git provides an option to print a simpler version of the log entries:

```
git_practice$ git log --pretty=oneline
ee76419954379819f3f2cacafd15103ea900ecb2 (HEAD -> master) Started
project.
git_practice$
```

The `--pretty=oneline` flag provides the two most important pieces of information: the reference ID of the commit and the message recorded for the commit.

The Second Commit

To see the real power of version control, we need to make a change to the project and commit that change. Here we'll just add another line to *hello_git.py*:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")
```

When we check the status of the project, we'll see that Git has noticed the file that changed:

```
git_practice$ git status
❶ On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

❷  modified:   hello_git.py

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

We see the branch we're working on ❶, the name of the file that was modified ❷, and that no changes have been committed ❸. Let's commit the change and check the status again:

```
❶ git_practice$ git commit -am "Extended greeting."
[master 51f0fe5] Extended greeting.
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)
❷ git_practice$ git status
 On branch master
 nothing to commit, working tree clean
❸ git_practice$ git log --pretty=oneline
 51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master)
Extended greeting.
 ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
 git_practice$
```

We make a new commit, passing the `-am` flags when we use the command `git commit` ❶. The `-a` flag tells Git to add all modified files in the repository to the current commit. (If you create any new files between commits, simply reissue the `git add .` command to include the new files in the repository.) The `-m` flag tells Git to record a message in the log for this commit.

When we check the project's status, we see that we once again have a clean working directory ❷. Finally, we see the two commits in the log ❸.

Reverting a Change

Now let's look at how to abandon a change and revert back to the previous working state. First, add a new line to *hello_git.py*:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")

print("Oh no, I broke the project!")
```

Save and run this file.

We check the status and see that Git notices this change:

```
git_practice$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

❶    modified:   hello_git.py

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

Git sees that we modified *hello_git.py* ❶, and we can commit the change if we want to. But this time, instead of committing the change, we'll revert back to the last commit when we knew our project was working. We won't do anything to *hello_git.py*: we won't delete the line or use the Undo feature in the text editor. Instead, enter the following commands in your terminal session:

```
git_practice$ git checkout .
git_practice$ git status
On branch master
nothing to commit, working tree clean
git_practice$
```

The command `git checkout` allows you to work with any previous commit. The command `git checkout .` abandons any changes made since the last commit and restores the project to the last committed state.

When you return to your text editor, you'll see that *hello_git.py* has changed back to this:

```
print("Hello Git world!")
print("Hello everyone.")
```

Although going back to a previous state might seem trivial in this simple project, if we were working on a large project with dozens of modified

files, all the files that had changed since the last commit would be reverted. This feature is incredibly useful: you can make as many changes as you want when implementing a new feature, and if they don't work, you can discard them without affecting the project. You don't have to remember those changes and manually undo them. Git does all of that for you.

NOTE

You might have to refresh the file in your editor to see the previous version.

Checking Out Previous Commits

You can check out any commit in your log, not just the most recent, by including the first six characters of the reference ID instead of a dot. By checking out and reviewing an earlier commit, you can then return to the latest commit or abandon your recent work and pick up development from the earlier commit:

```
git_practice$ git log --pretty=oneline
51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master)
Extended greeting.
ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
git_practice$ git checkout ee7641
Note: checking out 'ee7641'.

❶ 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 performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again.
```

Example:

```
git checkout -b <new-branch-name>

HEAD is now at ee7641... Started project.
git_practice$
```

When you check out a previous commit, you leave the master branch and enter what Git refers to as a *detached HEAD* state ❶. *HEAD* is the current committed state of the project; you're *detached* because you've left a named branch (`master`, in this case).

To get back to the `master` branch, you check it out:

```
git_practice$ git checkout master
Previous HEAD position was ee76419 Started project.
Switched to branch 'master'
git_practice$
```

This command brings you back to the `master` branch. Unless you want to work with some more advanced features of Git, it's best not to make any changes to your project when you've checked out an old commit. However, if you're the only one working on a project and you want to discard all of the more recent commits and go back to a previous state, you can reset the project to a previous commit. Working from the `master` branch, enter the following:

```
❶ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❷ git_practice$ git log --pretty=oneline
  51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master)
Extended greeting.
  ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
❸ git_practice$ git reset --hard ee76419
  HEAD is now at ee76419 Started project.
```

```
❹ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❺ git_practice$ git log --pretty=oneline
  ee76419954379819f3f2cacafd15103ea900ecb2 (HEAD -> master) Started
project.
  git_practice$
```

We first check the status to make sure we're on the `master` branch ❶. When we look at the log, we see both commits ❷. We then issue the `git reset --hard` command with the first six characters of the reference ID of the commit we want to revert to permanently ❸. We check the status again and see we're on the `master` branch with nothing to commit ❹. When we look at the log again, we see that we're at the commit we wanted to start over from ❺.

Deleting the Repository

Sometimes you'll mess up your repository's history and won't know how to recover it. If this happens, first consider asking for help using the methods discussed in Appendix C. If you can't fix it and you're working on a solo project, you can continue working with the files but get rid of the project's history by deleting the *.git* directory. This won't affect the current state of any of the files, but it will delete all commits, so you won't be able to check out any other states of the project.

To do this, either open a file browser and delete the *.git* repository or delete it from the command line. Afterwards, you'll need to start over with a fresh repository to start tracking your changes again. Here's what this entire process looks like in a terminal session:

```
❶ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❷ git_practice$ rm -rf .git
❸ git_practice$ git status
```

```
  fatal: Not a git repository (or any of the parent directories): .git
❹ git_practice$ git init
  Initialized empty Git repository in git_practice/.git/
❺ git_practice$ git status
  On branch master

  No commits yet

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

    .gitignore
    hello_git.py

  nothing added to commit but untracked files present (use "git add" to
track)
❻ git_practice$ git add .
  git_practice$ git commit -m "Starting over."
  [master (root-commit) 6baf231] Starting over.
   2 files changed, 4 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 hello_git.py
❼ git_practice$ git status
  On branch master
  nothing to commit, working tree clean
  git_practice$
```

We first check the status and see that we have a clean working tree ❶. Then we use the command `rm -rf .git` to delete the *.git* directory (`rmdir /s .git` on Windows) ❷. When we check the status after deleting the *.git* folder, we're told that this is not a Git repository ❸. All the information Git uses to track a repository is stored in the *.git* folder, so removing it deletes the entire repository.

We're then free to use `git init` to start a fresh repository ❹. Checking the status shows that we're back at the initial stage, awaiting the first

commit ❺. We add the files and make the first commit ❻. Checking the status now shows us that we're on the new `master` branch with nothing to commit ❼.

Using version control takes a bit of practice, but once you start using it you'll never want to work without it again.