

Contents

Live demo on command line usage of Git	1
Commonly used Git commands	1
Working with a local repo	2
Initialise a local repo	2
Set the local config	2
Query current repo status	2
Adding a file to the repo	2
Mark file for next commit	3
Make a first commit	3
Modify the committed file	4
Commit most recent changes	4
Review the commit history	4
Amending a commit	4
Locally working with branches	6
Create a feature branch for in-place addition	6
Merge back to master branch	6
Create a feature branch for multiplication and division	7
Add tests for multiplication and division on master	8
Integrating changes from divergent branches	8

Live demo on command line usage of Git

Commonly used Git commands

In our live demo we will use the following commands that are essential for working with Git from the command line:

These commands are used when working with a local repo only:

- `git config`: Set/Query repo options.
- `git init`: Initialise an empty Git repo.
- `git status`: Show the working tree status.
- `git add`: Add file content to the staging area to prepare the next commit.
- `git commit`: Add staged changes to the repo.
- `git merge`: Join two or more development histories together.
- `git rebase`: Reapply commits from one branch to tip of another branch.

Working with a local repo

Navigate to the Demos directory and locate the `Python_complex-class` sub-directory. It contains several other sub-directories which we will use in the following for our live demonstration.

Initialise a local repo

We start by initialising a local repo in some directory on our current system.

```
$ mkdir -vp $HOME/tmp/prothpc-git/local-repo
$ cd $HOME/tmp/prothpc-git/local-repo
$ git init
```

Set the local config

We will configure some information for the *local* repo only. You can set up your global configs later.

```
$ git config --local user.name "Niko Luke"
$ git config --local user.email "niko.luke@uni-kassel.de"
$ git config --list | grep user # check the user settings
```

Query current repo status

At this point we have clean new repo. We can query the repo status with

```
$ git status
On branch master

No commits yet

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

Adding a file to the repo

From the `Demos/Python_complex-class/0_init` directory copy the file `complex_class.py`. Please make sure to adapt the path to your current directory layout.

```
$ cp -av ../Demos/Python_complex-class/0_init/complex_class.py .
```

We again query the status of the local repo with the `git status` command. We see that the file `complex_class.py` is listed as being *untracked*.

```
$ git status
# some lines of output here
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    complex_class.py
# some lines of output here
```

Mark file for next commit

In order to make Git track the file we must first add it and put it into the staging area. After having `git add`ed the file we again query the current status of the repo.

```
$ git add complex_class.py
$ git status
# some lines of output here
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   complex_class.py
# some lines of output here
```

We can see that we have prepared `complex_class.py` to be tracked by Git after the commit.

Make a first commit

We now commit the staged files to make Git track them in the future.

```
$ git commit -m "Skeleton version of custom complex number class in
→ Python."
# some lines of output here
```

The file `complex_class.py` is now successfully committed and safely stored in the local repo.

Modify the committed file

We now will start developing our custom complex number class.

Next, copy the file `Demos/Python_complex-class/1_abs-value/complex_class.py` to the local repo.

```
# Copying the file simulates us changing the file manually.
$ cp -av ../Demos/Python_complex-class/1_abs-value/complex_class.py .
$ git status
# some lines of output here
modified:   complex_class.py
# some lines of output here
```

`complex_class.py` is now in the *modified* stage. If you are interested in the changes we applied to the file, execute the `git diff` command. We have added a class method that returns the absolute value of a complex number.

Commit most recent changes

We now add and commit the file to make a snapshot of the current state of our small software project.

```
$ git add complex_class.py
$ git commit -m "Add method to compute absolute value."
```

Review the commit history

We can now have a look at our recent commits by using the `git log` command:

```
$ git log --oneline # short output, one commit per line
$ git log --grep "absolute value" # use search pattern for filtering
```

Amending a commit

We further develop our complex number class by adding methods `add`/`subtract` values to our complex number objects. For this we must add the `__add__` and `__sub__` magic methods to our Python class.

```
$ cp -av ../Demos/Python_complex-class/2.1_add-sub/complex_class.py .
$ git diff # inspect the changes
$ git add complex_class.py
$ git commit -m "Add methods for adding and subtracting values."
```

But wait a moment?

We can now only add values to if value (e.g. a Python float or int) is on the right hand side of an instance of the complex class (let's call it `complex_value`) and obtain another instance: `complex_value + value`. We *cannot* add `complex_value` to `value` and obtain another instance: `value + complex_value`. We forgot to add the `__radd__` and `__rsub__` methods for reverse addition/subtraction.

```
$ cp -av ../Demos/Python_complex-class/2.2_radd-rsub_amend/complex_class.py
→ .
$ git diff
$ git add complex_class.py
```

Now we could of course make an additional commit stating that we added methods for reverse addition and subtraction. However, we want these changes to be part of the previous commit as well. We will amend our previous commit.

Before we do actually do this, let's recall the hash string of this commit (why will become clear in a moment):

```
$ git log --oneline | grep adding
978861b Add methods for adding and subtracting values.
```

Mind: Your commit hash might look different from mine.

Now let's really apply the changes by amending the previous commit.

```
$ git commit --amend
# An editor will pop up but we will leave the commit message.
$ git log --oneline | grep adding
19d53d2 Add methods for adding and subtracting values.
```

Please note that we have just *changed* the commit history! Since the content of our last commit changed the hash string is different now. That is to say, our new improved commit *replaces* the old commit.

For more details on how to rewrite the commit history of a repo see [here](#).

Locally working with branches

Create a feature branch for in-place addition

Create a new feature branch named `inplace_addition_subtraction` for adding the `__iadd__` and the `__isub__` magic methods for in-place addition and subtraction.

```
$ git checkout -b inplace_addition_subtraction
$ git status
On branch inplace_addition_subtraction
nothing to commit, working tree clean
# git log --oneline
```

We will make two *separate* commits:

1. Add unit tests for `__iadd__` and `__isub__`.

```
$ cp -v ../Demos/Python_complex-class/3.1_iadd-isub_feature-
↳ branch/test_complex_class.py
↳ .
$ git status
$ git add test_complex_class.py
$ git commit -m "Add unit tests for in-place addition and subtraction."
```

2. Add implementation for `__iadd__` and `__isub__` methods.

```
$ cp -v ../Demos/Python_complex-class/3.1_iadd-isub_feature-
↳ branch/complex_class.py
↳ .
$ git status
$ git add complex_class.py
$ git commit -m "Add implementation for in-place addition and subtraction."
```

Note: When creating a branch to add new features to your code, always add a corresponding test as well!

Merge back to master branch

Now switch back to the master branch and merge the feature added on the branch `inplace_addition_subtraction`.

```
# We are still on the 'inplace_addition_subtraction' branch.
$ git status
On branch inplace_addition_subtraction
nothing to commit, working tree clean
$ git switch master
Switched to branch 'master'
$ git merge inplace_addition_subtraction
# some output lines here
Fast-forward # <-- the merge strategy
# some output lines here
```

Finally we delete the `inplace_addition_subtraction` branch since branches should not live too long.

```
$ git branch -d inplace_addition_subtraction
```

Create a feature branch for multiplication and division

From the master branch create a branch named `multiplication_division` for adding `__mul__`, `__rmul__` and `__imul__` as well as `__truediv__`, `__rtruediv__` and `__itruediv__` methods.

```
$ git checkout -b multiplication_division
```

1. Add implementation of `__mul__`, `__rmul__` and `__imul__` methods.

```
$ cp -v ../Demos/Python_complex-class/3.2_mul/complex_class.py .
$ git add complex_class.py
$ git commit -m "Add implementation for multiplication."
```

2. Add implementation of `__truediv__`, `__rtruediv__` and `__itruediv__` methods.

```
$ cp -v ../Demos/Python_complex-class/3.3_div/complex_class.py .
$ git add complex_class.py
$ git commit -m "Add implementation for division."
```

Add tests for multiplication and division on master

We will add the unit tests for multiplication and division on the master branch.

Note: At this point we will *not* merge back the `multiplication_division` branch.

1. Add unit tests for `__mul__`, `__rmul__` and `__imul__`.

```
# Make sure we are on the master branch
$ git switch master
$ cp -v ../Demos/Python_complex-class/3.2_mul/test_complex_class.py .
$ git add test_complex_class.py
$ git commit -m "Add unit tests for multiplication."
```

2. Add unit tests for `__truediv__`, `__rtruediv__` and `__itruediv__`.

```
$ cp -v ../Demos/Python_complex-class/3.3_div/test_complex_class.py .
$ git add test_complex_class.py
$ git commit -m "Add unit tests for division."
```

Integrating changes from divergent branches

We first make a copy of the repo:

```
$ cp -av local-repo local-repo.rebase
```

We now have two options to integrate the development on master and `multiplication_division`.

1. merge the `multiplication_division` branch to the master branch:

```
# Make sure we are on the master branch.
$ git switch master
$ git merge multiplication_division
Merge made by the 'recursive' strategy.
# some output lines here
$ # pytest -v # make sure all tests run without errors
```

Since commits were made on both branches `master` and `multiplication_division` are said to be *diverged*. The merge strategy is now is the *recursive* strategy (instead of a fast-forward merge). Git will create an extra *merge commit* for integrating the changes from both branches.

2. rebase the master on multiplication_division and merge back to master:

Switch to the local-repo.rebase directory and execute the following commands:

```
# Make sure we are on the 'multiplication_division branch
$ git switch multiplication_division
$ git rebase master
Successfully rebased and updated refs/heads/multiplication_division.
$ git switch master
$ git merge multiplication_division # a fast-forward merge
$ # pytest -v # make sure all tests run without errors
```