Getting started with Bash, Emacs, Python and Git

Marc van der Sluys

Nikhef/GRASP, Utrecht University
The Netherlands

May 18, 2022

http://pub.vandersluys.nl

This document was typeset in LaTeX by the author.
The official GNU Bash logo was created by the Free Software Foundation.
The original Emacs logo was created by Luis Fernandes.
The Python logo is a trademark of the Python Software Foundation.
The Git Logo by Jason Long is licensed under the Creative Commons Attribution 3.0 Unported License.

Table of contents

Preface

The (bash) shell, terminal or command line may look daunting at a first, because it looks so Spartan. However, I find that my fingers have often already issued a command before I realise it myself. Typing a command is definitely faster in most cases than moving the hands from the keyboard, finding the mouse and then finding the mouse pointer, let alone directing it wherever it is that you want it to go.

This document is aimed at helping anyone who would like to give the shell a try, but does not know where to start. Despite the title, it does *not* teach you bash in full, neither emacs, Python or git. Instead it tries to give a hands-on example on how to use bash and emacs to write a hello-world Python script, how to make it executable and run it, and how to set up a git repository, add your code and push it into the cloud in a simple workflow. Perhaps you already know how to do this using a mouse, in that case here you will learn how to do the same whilst staying clear from that vermin as much as possible. The document provides references to more introductory material on bash, emacs and git. If you want to learn Python, you will have to look elsewhere.

1 The Bash shell

The **shell** or **command-line interpreter** (**CLI**) is a text-console-based interface through which the user can communicate with the OS. Examples of popular shells are **tcsh**, **bash** and **zsh**. For this course, we recommend the Bash shell. While use of a shell may look Spartan and require more knowledge than the use of a GUI, the shell can be extremely powerful, aided by the use of **hot keys** such as `Ctrl-R` to reuse a command and `Tab` **completion**. An additional advantage for scientists and engineers is that Bash looks and behaves the same on your local machine as **remotely** on a server or headless system through an ssh connection. A further strength of shell commands is that you can put them in a text file with execute permissions, and you have created a **script**. A Bash script is in fact a simple interpreted 'program' that can be written in seconds or minutes and which can handle regular tasks consisting of a fixed set of commands.

Some information in this Chapter has been adapted from ***"Efficient use of the Linux command line in the Bash shell"*** [1], which I will regularly refer to (abbreviated as EUBS) for more details.

1.1 Starting tasks in the shell

1.1.1 The command prompt

The shell is text-based and works by typing commands followed by `Enter`. The location in the terminal where the shell expects a command is called the **command prompt** or simply **prompt**. The command prompt often contains useful information like the current user, the machine name and/or the current working directory. For example, I am currently writing these notes on my laptop (named "think") in a text editor launched from a shell with the prompt

```
[sluys@think BashEmacsCandGit]$
```

User shells are often indicated with a $, whereas root (superuser) shells commonly have a # in their prompt. It is a good idea to add colours to your prompt in order to quickly find where the many screenfulls of output started. You can achieve this (and other customisations) by setting the `PS1` environment variable. See Section 8.4 in EUBS [1], [2] or [3] for examples.

1.1.2 Starting a task

A task or program can be started in the shell by simply typing its name (plus any arguments that may be needed) at the command prompt, followed by Enter. For example, to print the contents of the Python script **hello.py** to the screen, I type

```
$ cat hello.py
```

If I want to start a program that resides in the current directory, in this case my script **hello.py**, I have to prepend **./** to its name:

```
$ ./hello.py
```

Note that **hello.py** must have execute *permissions* in order for this to work (see Sect. 1.2.4).

1.1.3 Foreground and background tasks

In order to open the file **hello.py** with the text-based version of the text editor **emacs** I can type

```
$ emacs -nw hello.py
```

This will open emacs in the **foreground**, *i.e.* it uses the terminal window to run and my Bash prompt is no longer available until I close emacs. Opening the graphical version of **emacs** in the foreground is awkward, as it opens a new graphical window for the editor, while the Bash command prompt does not return and the terminal window becomes useless. Instead, **emacs** can be started in the **background** by appending an ampersand (&):

```
$ emacs hello.py &
```

This will start **emacs** as before, but my Bash prompt returns, so that I can still use the terminal (*e.g.* to execute my script after editing the code).

If you forgot the ampersand, you can stop the foreground job by pressing Ctrl-Z and resume the editor in the background using the **bg** command:

```
$ emacs hello.py
Ctrl-Z
[1]+ Stopped                 emacs
$ bg
[1]+ emacs &
$
```

As you can see, the terminal indicates that the job is in the background using the ampersand, as if you had typed it yourself.

The following commands help to monitor or control (background) processes: **jobs** lists jobs in the current shell, **ps** lists processes on the system, **top** monitors the most CPU-intensive tasks, Ctrl-Z stops a running foreground job, **fg** and **bg** can move a job to the fore- or background and **kill** can send a signal to a process, for example to kill it, but also to suspend or resume it or perform a user-defined action. These commands are explained in more detail in Section 5.2 of EUBS [1].

1.1.4 Starting tasks concurrently

As we have seen, appending an ampersand (&) starts a task in the background and returns the Bash prompt immediately. This can be used to start multiple tasks (*i.e.* programs, scripts) **concurrently**:

```
$ task1 & task2 & task3
```

When `task1` is started, the command prompt returns, sees the command to start `task2` and does so immediately. Hence, all three tasks will be started at (almost) the same time. However, in which order the concurrent tasks will be *executed* depends on the scheduler.[1]

1.1.5 Starting tasks sequentially

As in the C programming language, the semicolon (;) indicates the end of a command line. Normally it is not used, as the `Enter` key has the same function, but it can be used to enter multiple commands on a single line:

```
$ task1 ; task2
```

These two tasks (running in the foreground) will be executed **sequentially**: `task2` will only start once `task1` has finished (just like in Python, where the next code line is only executed after the instructions for the current line have been completed).

Using the logical `and`, represented by `&&`, we can realise **conditional** execution of multiple tasks:

```
$ task1 && task2
```

Again, the two tasks run *sequentially*, but `task2` will only start if `task1` has finished **successfully**. This is useful when developing code, in order to test the current version by running it again and again and checking the output. However, if the code does not run properly, there is no need to check the output:[2]

```
$ ./hello.py > newoutput.txt && diff oldoutput.txt newoutput.txt
```

Finally, executing the next command only if the previous one **failed** can be achieved using the logical `or` (`||`):

```
$ ./hello.py || echo "Script failed."
```

1.1.6 Starting and running tasks nicely

Linux uses **nice values** to set **priorities** for user processes. The higher a process's nice value, the 'nicer' it is for *other* processes and the less run time it may get from the scheduler. Users can assign nice values between 0 (default) and 19, only root (the system administrator) can also use negative values between -20 and -1. Nice values are specified using the **`nice`** command:

```
$ nice -5 task1
# nice --5 task1
```

The first command starts `task1` with a nice value of 5 (the `-` is a *flag*), the second starts it with a nice value of -5 (executed by root; the second `-` is a minus).

To change the nice value of a running task, the **`renice`** command can be used:

```
$ task1 &          # start task1 in the bg with nice=0
$ ps -l            # find the PID <pid> of task1
$ renice 5 <pid>   # task1 set to nice=5
```

See `man nice` and `man renice` for more details.

1.2 Efficient use of the shell

While typing commands every time you need to do something may sound tiring, there are many tricks that allow you to issue commands with a minimum of typing.

[1] As an example, try running `ls & ls -l` a few times in a directory with a number of files (use arrow up) and see how the output changes.

[2] See Sect. 1.3.3 on redirection with the `>` symbol.

1.2.1 Tab completion

Tab completion allows you to type only the beginning of a command or file name and press the Tab key to complete it. For example when I type `libr` and press Tab, the shell will expand it to

```
$ libreoffice
```

If a completion is not unique, the word will be completed as far as possible. Pressing Tab a second time will show the possibilities. This is what happens if I type `lib` and press Tab twice:

```
$ lib
libart2-config libgcrypt-config (...) libreoffice (...) libwmf-fontmap
```

Typing the "r" and pressing Tab again will then give the desired result.

1.2.2 Reusing commands

Since you will often repeat commands or issue commands similar to earlier ones, it is useful (and easy) to reuse and/or adapt earlier commands.

The easiest way to do this is to select a previous command using the **arrow up and down** keys ↑ ↓ to select an earlier command. The key strokes Alt-< and Alt-> jump to the beginning and end of your history respectively. Alternatively, you can **reverse search** an earlier command by pressing Ctrl-R and typing a match string. If the first match is not what you are looking for, narrow it down by typing more and/or press Ctrl-R again. In both cases, pressing Enter will execute it. If you want to edit the line first, press Esc to exit the search mode. This means that searching for a (long) command that is *similar* to what you need is useful.

When pressing Enter, the shell will execute the command after which its history will return to 'now'. If instead you want to execute an earlier command and then the next one that came after it, press Ctrl-O instead of Enter. This will execute the command, and then show the next command line in the shell's history at the prompt, ready for editing or execution.

In many cases you want to **adapt** the earlier command before executing it. After selecting the desired command line, you can navigate using the **arrow left and right** keys ← → to move character by character, use these arrows combined with the Ctrl or Alt key to jump words, or use Ctrl-R from the end of the line to jump to the word you want to start editing. Ctrl-A and Ctrl-E jump to the beginning or end of the line respectively.

Typed text is inserted between existing text, it does not overwrite it. The Delete and Backspace keys work as you expect, Alt-Backspace deletes entire words and Ctrl-U and Ctrl-K remove everything from the cursor to the beginning or end of the line respectively. The text deleted using the last three methods can be yanked back at the cursor using Ctrl-Y.

When fixing typos, Ctrl-T swaps two characters and Alt-T two words. If you made a mistake, Ctrl-/ will undo your last edits. If you did a lot of damage, Alt-R reverts to the original command line.[3]

There is no need to move to the end of the line once you have finished editing it — just press Enter.

[3] Note that many of these key combinations are defined by the *Readline* package [4, 5], which is also used by *e.g.* emacs, so that it uses the same shortcuts.

1.2.3 Using directories

The **root directory** on Linux file systems is called '/'. All other directories (including other (network) drives) are *subdirectories* of this directory. A user's **home directory** is usually located in `/home/<username>`, where the slashes separate the different directory names. Moving closer to the root directory is usually called moving 'up' a directory.

You can change directories using the `cd` command followed by the name of the directory you want to change to. For example, if I want to change to my home directory, I could issue

```
$ cd /home/sluys
```

Here I used the full (absolute) **path**, starting from '/'. Instead, if I am already located in `/home` I could use a relative path:

```
$ cd sluys
```

When you want to 'go up' one directory, you can use

```
$ cd ..
```

The two dots indicate the **parent directory** of the current working directory. I can jump several directories with one command, going up and down. For example, if I am in `/home/sluys/work/GWs` and want to go to `/home/sluys/private/hobby`, I can do one of

```
$ cd /home/sluys/private/hobby   # use the absolute path
$ cd ../../private/hobby         # use the relative path
$ cd ~/private/hobby             # use the path from home
```

The first option uses the absolute path, the second a relative path and the third makes use of the shortcut ~, which stands for *home directory*. In fact, the `cd` command without an argument changes to the user's home directory.

In order to see what is in the current directory, for example files and subdirectories, you can use the `ls` command, which lists the contents. If, after moving around for some time, you forgot where you are, simply print the working directory using `pwd`.

Creating a new directory can be done using `mkdir <dirname>`, removing an empty directory is easy with `rmdir <dirname>`. If you want to remove a directory and all its contents (files, subdirectories) use `rm -r <dirname>`.

1.2.4 Files

You can list the files in the current directory with the `ls` command. The command `ls -l` provides extra information, like file size, date stamp, permissions and the owner. Read the contents of a text file using the `less` command followed by the file name:

```
$ less file.txt
```

You can scroll through the file with ↑/↓ or PgUp/PgDn, hit Space to skip a screenfull, search with / followed by the search string and Enter, find the next hit with n, jump to the beginning or end of the file using g and G respectively, open the file in your favourite editor with v and quit `less` using q. See `man less`[4] for more information.

A file can be copied to another directory, or to a new name in the current directory using the `cp` command. Moving or renaming a file are the same thing and use the command `mv`. Removing a file is done with `rm`.

[4] `man` by default uses `less` to display man pages, so that these key strokes also hold when reading man pages.

```
$ cp file.txt dir/     # copy file.txt into dir
$ mv file.txt dir/     # move file.txt into dir
$ mv file.* dir/       # move everything matching file.* into dir
$ mv file1 file2       # rename file1 to file2
$ cp file1 file2       # duplicate file1 as file2
```

The permissions of a file can be changed with the **chmod** command. This can make a file executable or prevent others from reading it. For example, `chmod u+x script.sh` makes the shell script `script.sh` executable (`x`) for the current user (`u`). See `man chmod` for details.

1.3 Utilities

Utilities, such as **system commands** (`ps, nice, fg, bg, kill`), **text editors** (`emacs, vi, nano`), **test tools** (`cat, less, diff, grep, sed, awk`), **compiler utilities** (`gcc, gdb, gprof, valgrind`), **interpreters** (`bash, python`), **source-control** systems (`git, svn, bzr`) and many more, help to operate your computer. Of special importance is the command `man`.

1.3.1 The man pages

The **man pages** provide documentation for system commands, system calls and more. The information includes the **syntax**, the **header files** that must be included, and often an **example**. The man pages contain about ten sections, of which the first contains user/shell commands and is the most relevant for users of the shell.[5]

The different sections can be called by specifying their number. For example, `man kill` and `man 1 kill` show information on the kill **command**, while `man 2 kill` discusses the kill **system call**. If the section number is omitted, it defaults to the lowest section for which the man page is available; *e.g.* `man kill` will show information on the command.

See Section 7 of EUBS [1] for more information on how to use the man pages and tips how to make them easier to read using colours (as well as [2, 3]). See also the command `man man`.

1.3.2 Version control using git

Currently, git [6] is the state-of-the-art version control system. It is free, open source software (FOSS) developed by Linux Thorvalds for the Linux kernel. I strongly recommend using git for programming. To initialise, you can use the **git** command (after `cd`'ing to the directory containing `hello.py` and/or creating that file):

```
$ git init .          # Create a repository in the current directory
$ git add hello.py    # Add the first version of your program
```

Then, when the latest version of your program works well, do:

```
$ git diff                  # Check the changes you made
$ git commit -m 'Message'   # Commit the current version of your work
$ git log                   # See all your commits
```

We will provide a hands-on example using git in Section 4.5.

1.3.3 Pipes and redirection

In the shell, the symbol "|" ("**pipe**") is used to indicate an (unnamed) pipe. The line

```
$ command1 | command2
```

[5] Sections 2 and 3 contain information on Linux system calls and the standard C library respectively, and are useful for C programmers.

"pipes" the standard output from `command1` to the standard input of `command2`. An example is

```
$ ls | wc -l
```

which takes the output from `ls` and counts the lines using `wc -l`, effectively counting the number of entries in the current directory.

Redirection in the shell can be achieved using the `>` symbol. The following syntax is available:

```
$ command > file    # redirect stdout to file (> is short for 1 >)
$ command 2> file   # redirect stderr to file
$ command &> file   # redirect stdout and stderr to file
$ command 2>&1      # redirect stderr to stdout (e.g. for grep/sed)
$ command >> file   # redirect stdout to file and append (default is
                      overwrite)
$ command < file    # redirect file to command stdin
```

For example, the command

```
$ ls -l > file.txt
```

will store the output of `ls -l` in the file `file.txt`. If the file does not already exist, it will be created; if it does, it will be **overwritten**. You can check the result using *e.g.* `less file.txt` or `cat file.txt`.

1.4 Further reading

More tips on the use of the Bash command line can be found in EUBS [1]. You are encouraged to use the example configuration files at [3] or [2] to enhance your Bash, emacs and git environments with colours, aliases and more. O' Reilly's *Learning the Bash shell* [7] provides an in-depth introduction to the Bash shell and shell scripting. More information on shell scripting can be found in the online document *Advanced Bash-Scripting Guide* [8].

2 Basic use of Emacs

The rationale of recommending emacs[6] is that, while it can take some time to learn, it can easily speed your text editing up with a factor of two or more once learnt. In addition, you need only a single editor for Python, bash, LaTeX C, html and many more languages.[7] If you are going to edit text files for another few decades, it is probably worth the investment. See EUBS [1] for more explanation.

I treat GNU Emacs [10] here, not XEmacs, Aquamacs or any other flavour. I used *Effective Emacs* [11] and *How to learn Emacs* [12] for inspiration and information. See also the official Emacs Reference card and Survival card for more shortcuts [13].

You can open a text file by specifying its name as an argument:

```
$ emacs hello.c
```

If a graphical version of emacs and graphical desktop are available, emacs will open in a new window and you may want to append an ampersand (&) in order to start it in the background. You can force the terminal version of emacs (in the foreground with the option `-nw`.

A default emacs screen consists, from top to bottom, of:

1. The menu (if shown);
2. The main text-edit panel (called *buffer*);

[6] Or vi/vim/gvim. I chose for emacs here because of the gentler learning curve and overlap with bash.
[7] In case you've never heard of Org mode [9] — that alone would be a reason to pick up emacs.

3. A status line;
4. A *minibuffer*, where commands show up (usually one line).

The menu can be accessed (and exited) with F10, and navigated with the arrow keys and Enter. For most options, the keyboard shortcut is shown, so that you can quickly learn it.[8] Once you know the shortcut, you'll probably never use the menu again. Note that only a small (but important) fraction of emacs's functionality is available through the menu!

The most important key strokes to know in order to start learning emacs are:

- F10 Access or exit the **menu**;
- C-/ **Undo** text edit;[9]
- C-g **Cancel** the current command (press more than once if needed);
- Esc Esc Esc **Cancel everything** and go back to edit mode;
- C-x C-s **Save** the current buffer to a file.
- C-x C-c **Exit** emacs, prompting to save changed buffers.

Here, C-/ stands for Ctrl /, *i.e.* holding Ctrl whilst pressing /, then releasing both.[10] Similar combinations exist for Alt, which is written as M-.[11] Most commands have a (long) name as well as a shortcut, which can be typed using M-x, followed by the name of the command and Enter. Tab-completion works while typing the command. For example *undo* can also be achieved by M-x `undo` Enter.

The undo function in emacs is different from its equivalent in many other programs. Consider for example the case where you type `a`, remove it, and then type `b`, then press `undo`, then type `c`. If you then start pressing `undo`, many programs will never recover `b` — that version of the document has been lost forever. Instead, when pressing C-/ repeatedly, emacs will "replay" in reverse order *exactly* what happened before, including undoing previous undo's. While perhaps unorthodox, this means that you will always be able to get back to any previous version by pressing `undo` (within the limits of your undo history).

2.1 Configuration of Emacs

The configuration of your Emacs editor is stored in the *hidden* (note the leading dots) file or directory `.emacs.d/init.el` or `.emacs` in your home directory. An example configuration file can be found at [2] and [3].

3 Getting started with git and GitHub

We will organise our code using git. This is useful *locally* to structure your own work. In order to share your code with others, we will use a *remote* cloud service, in this case GitHub.[12] We will see how to set up git, a local repository and a remote one. Then we'll discuss the basic git work cycle and how to resolve git merge conflicts.

[8] The same is true for gvim.

[9] This was the most important keystroke for me when learning emacs; knowing this I could undo whatever stupid mistake I made.

[10] For C-x C-c you can hold Ctrl while pressing X and then C.

[11] Pressing and releasing Esc has the same effect as pressing and holding Alt.

[12] Although it could have been GitLab, Bitbucket, SourceForge, ...

3.1 Getting started with git

3.1.1 Configuring git

If you haven't used git before (as this user on this system), you need to setup git with some basic options:[13]

```
$ git config --global init.defaultBranch master
$ git config --global pull.rebase false
$ git config --global user.name "<your name>"
$ git config --global user.email "<your@email.addre.ss>"
```

The first two options set the default way of creating a repository and to merge different sets of changes. The last two are your (real) name and email address.[14] Except for the first option (init), you can leave out `--global` if you want to apply the settings only to the current repository (after initialising it, see below).[15]

3.1.2 Creating a local git repository

Ensure that you are in the correct directory for your git repo. Then, create a **git** repository in the current directory (which contains or will contain our source-code files):

```
$ git init .
```

It is good practice to add a source file to the repository as soon as you have a first version. Note that this first version does not have to do anything spectacular yet, but it should compile and run correctly.

Let's add (and then commit) the source code for `hello.py` (assuming we are in `~/exercises/hello/`):

```
$ git add hello.py  # Add file to repo
$ git commit -m "Initial commit of hello.py"  # Commit changes
```

3.2 Setting up a repository on GitHub

GitHub, like Bitbucket and GitLab, currently offers free private (*i.e.*, others can't see it unless you want them to) and public `git` repositories.[16] I use private repositories as backup and/or to share my code with collaborators and public repositories for open-source projects. At the end of a day, GitHub can show me what I achieved that day.

Before you create an account at GitHub, find an email address that you are willing to share with others. Your email address is how GitHub recognises you. When signing up, give your full name (or as much of it as you want to share) and make up a good password — you'll have to type it each time you *push* your commits to GitHub.[17,18]

Once your account has been set up, you are ready to create your first repository. In order to do so, in your home screen, click the **plus** in the top right of the page, choose **New repository**,

[13] If we don't address these things now, we will be bothered by git about them later.

[14] If this is going to be a *local* repository only, you can make something up. However, you may want to *push* your commits to a *remote* repository, for example for backup purposes or to share them with others. In the latter case, you should choose a name and email address that others are allowed to see and can recognise. You email address may have to be the same as the one you used for the remote cloud service.

[15] Global options are stored in `~/.gitconfig`, local options in `.git/config` and can always be changed with a text editor.

[16] See [14] to apply for a student pack with fewer limitations using your school email address. You can do this after creating an account.

[17] Unless you use an ssh key, see *e.g.* https://www.nikhef.nl/pdp/computing-course/work/ssh.html.

[18] As a quick workaround, use the credential helper: `git config --global credential.helper 'cache --timeout=3600'` will remember your credentials for an hour (3600 s).

choose a good (not too long, without spaces) **name**, add a description for your repo, choose a **private** repository, **do not** initialise the repository with a README, **do not** add a `.gitignore` or licence file, and click **Create repository**.[19]

GitHub will now show you how to push an **existing repository** from the command line:

```
$ git remote add Server git@github.com:<user>/<repo>.git
$ git push -vu Server master
```

The command **`git remote add`** adds a remote location called `Server`. I usually use the name `Server` (or `GH` if I intend to push my commits to multiple remote locations) rather than `origin`.[20] The **`git push`** command pushes your master branch to `origin` (or `Server`). You have to specify the name of the branch only once — after this

```
$ git push
```

suffices (though I add the option `-v` to see what happens in more detail).

After pushing your commits to the GitHub server, reload their web page with instructions to see your code.

The opposite of pushing, *i.e. pulling* changes that others made from the remote location to your local repository, is then simply done with

```
$ git pull
```

If you want to share your (private) repo with other GitHub users, go to the page of the desired repository, click **Settings** (top bar, right) / **Collaborators** (left menu), type and select the **user name**, and click on **Add collaborator**. The other user will receive an email and can now **clone** (*i.e.* do an "initial pull") your repo and push their changes. See [15] for detailed help.[21]

3.3 The git workflow

Once a git repository has been set up, we can develop our useful program further. Our workflow will look like:

```
$ git pull    # Get the latest version from the cloud before starting
```

Develop and test your code:

```
$ emacs hello.py    # Edit your source code
$ git diff          # Do the changes make sense?; exit with q
$ ./hello.py        # Code MUST run as expected before we commit
$ git diff          # Check your changes; exit with q
```

Once your new feature has been developed and tested, commit your changes...

```
$ git commit -a -m "Short, informative message"    # Commit all changes
```

...and push them to the cloud:

```
$ git log     # Check your commit history; exit with q
$ git push    # Push our commits to the cloud
```

Then restart the cycle from the top:

```
$ git pull    # Et cetera...
```

[19] Alternatively, if you don't have a local repository or code yet, you can create an empty repo online (or fill it with a README, `.gitignore` or licence and/or other files) before *cloning* it locally using *e.g.* `git clone git@github.com:<user>/<repo>.git`. The *remote* is then set automatically, so after adding and committing files locally you can push and pull right away.

[20] The name `origin` for GitHub doesn't make sense when the repository originates on my laptop.

[21] Note that while anyone can *see* and *clone* a public repo, no-one can *push* to it by default.

If you don't specify a commit message with `-m`, git will open the editor specified in the environment variable `$EDITOR`.[22] Write a single line (or, better, a single line as title, a blank line and below that more details), save it and exit your editor.

If a coworker has pushed their commits before you did, you will need to pull the latest code version from the remote before pushing:

```
$ git push

Pushing to github.com:UserName/RepoName.git
To github.com:UserName/RepoName.git
rejected] master -> master (fetch first)
error:  failed to push some refs to 'github.com:UserName/RepoName.git'
hint:  Updates were rejected because the remote contains work that you do
hint:  not have locally.  This is usually caused by another repository pushing
hint:  to the same ref.  You may want to first integrate the remote changes
hint:  (e.g., 'git pull ...')  before pushing again.
hint:  See the 'Note about fast-forwards' in 'git push --help' for details.
```

Note how git provides useful hints after the error message, including the solution (`git pull`). The last line even provides a reference to more detailed documentation.

3.3.1 Committing selected files only

You may be developing a new feature in your project when you discover an unrelated issue in another file. If you quickly fix that issue, you may want to commit that file only. In that case:

```
$ git add module.py
$ git status  # Which files were modified, which were added (staged)?
$ git commit -m '<message>'
```

Note that we didn't use `-a` to commit *all* changes. Hence, only the added file(s), known as *staged* files will be committed. Hence `git add` has two meanings:

1. Add a new file to the repo that wasn't previously part of it;
2. *Stage* an existing file for commit.

3.3.2 Git rules

When working with files in a git repo, it is important to obey the following rules:

1. Make sure you **always** work in the git directory. **Never *ever*** copy the files elsewhere, edit them there, and copy them back.[23]
2. Git can merge different changes made by different users when doing `git pull`, even to the same files in the repository. However, this only works if you edit different lines within the same file. If you're editing the same lines as your colleague (without full cycles of committing, pushing and pulling for both you and your coworker in between), git receives one set of changes from you and a conflicting set of changes from your colleague and no longer knows what to do. This is known as a **conflict**.
3. Avoid conflicts by pulling, committing and pushing often, even for relatively minor (but always complete) (sets of) changes. Note that the conflict will usually occur on the side of the person that has a slower pull/commit/push cycle.

[22] You can set this variable with `EDITOR=emacs` and check the result with `echo $EDITOR`. However, this variable will be lost when you exit the shell. To fix that, setup your environment properly (see Sect. 4.7).

[23] Unless you want to piss off your coworkers by destroying their work. What is likely to happen at some point is that they make changes, and you copy back into the git directory a version *without* their changes.

4. Another way to avoid conflicts is by using many short lines, separated by hard returns, instead of a few long ones, if possible and/or applicable (*e.g.* in LaTeX). Remember that conflicts arise when multiple people are editing the same lines. Having your content divided over more lines reduces the risk.

3.4 Resolving git conflicts

If you work with git long enough, a git conflict[24] is likely to happen at some point. While this is annoying, this is by no means the end of life on this planet. All it means is that two different developers made a different set of changes (to the same line of code) and git needs human intervention to resolve this. For example, if the original line of code was `A`, you changed it to `B` and your coworker to `C`, git will ask you to make up your minds. Conflicts occur when git merges different versions of the code, which in the cycle described above is when you do a `git pull`.[25,26] Luckily, you all have been pulling, committing and pushing quite often, so the cause of the conflict is recent and the intention of the changes still fresh in your mind. Hence, the conflict will be easy to fix.

3.4.1 Recognising a conflict

Let's assume we have a cool Python program

```
print('Hello world!')
```

but you want to make it even cooler

```
print('Hello world!!!')
```

while at the same time (*i.e.* starting from the same commit/version) your annoyingly quick and uncertain colleague is not that sure

```
print('Hello world??')
```

If your colleague (commits and) pushes first, **you** will run into a conflict when you pull,[27] which looks something like:

```
$ git pull

Auto-merging hello.py
CONFLICT (content):  Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.
```

3.4.2 Resolving a conflict

You can get more details on the conflict with

```
$ git status

<...>
You have unmerged paths.
 (fix conflicts and run "git commit")
 (use "git merge --abort" to abort the merge)
```

[24] Because conflicts happen when git merges versions, it is officially known as a *git merge conflict*

[25] At the command `git pull`, git will fetch a remote copy of the code and (try to) merge it with your local copy. Other situations where git merges two versions are *e.g.* when merging branches, rebasing, cherry picking or popping a stash.

[26] This is by no means a reason to pull less often — on the contrary: you may prevent them this way, and *if* there is a conflict, you'll want to catch it early.

[27] You won't be able to push, as we saw in Section 3.3.

```
Unmerged paths:
 (use "git add <file>..." to mark resolution)
 both modified:  hello.py
```

In other words, in order to resolve the conflict, we need to manually edit `hello.py`, which now looks like

```
<<<<<<< HEAD
print('Hello world!!!')
=======
print('Hello world??')
>>>>>>> 081049eb6c245a4fbf94e1be3056c914342f4006
```

Here, the version between `<<<<<<<` and `=======` is your latest version (known as `HEAD`) and the version between `=======` and `>>>>>>>` colleague's version, where the long number is the hash of her commit. Note that for complex conflicts, there may be several of these blocks in your code.

You may come up with a resolution yourself, for example if there is an obvious error, or you may have to discuss this with with your colleague. Edit the file with your favourite editor[28] and make sure you remove the metadata as well (check that your program runs (and/or compiles) before you commit it):

```
print('Hello world!?!?!')
```

Again, you can see how things are going

```
$ git status

All conflicts fixed but you are still merging.
 (use "git commit" to conclude merge)

Changes to be committed:
 modified:  hello.py
```

so you can

```
$ git commit -a
```

which will show a commit message template with a default message (and more details in the comments)

```
Merge branch 'master' of github.com:UserName/RepoName
```

and finish with a `git log` —to see whether all is well— and a `git push`. Note that your colleague will need to `git pull`.

3.5 Further reading

See *Git for coworkers* [16] for more basic information on the git work cycle and links to more detailed information. Some more details on creating a git conflict can be found here.[29] GitHub provides *git cheat sheets* [17] and a longer introduction into git and version control in *About git* [18]. Introductions to related topics can be found in the GitHub guides [19]. The book *Pro git* is open source and online [20].

[28] Emacs!

[29] Create and resolve a git merge conflict: https://github.com/MarcvdSluys/LIGO-Virgo-notes/blob/master/git/git-merge-conflict.org.

4 Example: Python programming on the command line

As an example, we will write and execute a simple hello-world program in Python on the command line, using the **bash** shell. We will use the **emacs** text editor for writing, **git** for version control, and the **man pages** as a source of information.

4.1 The Bash shell

Most terminals start with a **bash** shell. Open a terminal window, type `ps` and check that it lists `bash`. If not, you can type `bash` to start a new bash shell in your terminal or use the current shell instead.[30] Typing `exit` will exit the current shell and return to the previous one (or close your window if there isn't a previous one).

4.2 Creating directories

Before we start writing our hello-world program, we will first create the necessary directories. Ensure that you are in your *home directory*[31] and use `mkdir` to create a new directory called `mycode`, which will contain our program and `cd` into it:[32]

```
$ cd
$ mkdir mycode
$ cd mycode
```

Here, we will create a subdirectory called `hello` and `cd` into it:[33]

```
$ mkdir hello && cd hello
```

You can now check that you are indeed in the directory `/home/<username>/mycode/hello/` by typing `pwd`.

More information on working with directories is provided in Section 3 of EUBS [1].

4.3 Creating a Python file using emacs

Next, we will use the text editor **emacs** to write a small *hello-world* program called `hello.py`. We open the file in the terminal using

```
$ emacs -nw hello.py
```

Then type (or copy) the program shown in Code listing 1.

Listing 1: hello.py: a *hello-world* script in Python.

```
1 print('Hello world!')
```

Save the buffer to file by pressing C-x C-s. Emacs should display something like *Wrote /home/user/.../hello/hello.py* to the minibuffer in the bottom left. You can then exit emacs by pressing C-x C-c.

Check the contents of the current directory with `ls`:

```
$ ls -l     # Note 1 is a lowercase L
-rw-r--r-- 1 student student 22 Feb 5 17:22 hello.py
```

[30] A different shell will probably behave similarly most of the time, but may give surprising results here and there.

[31] Your home directory is in `/home/<username>/` and has the alias ~. You can check your current directory with `pwd`. You can always jump (back) to your home directory by typing `cd ~` or `cd` without any argument.

[32] Note that $ is Bash's *command prompt*, which you should not type.

[33] Note that in the previous step, you could have done `mkdir -p mycode/hello && cd mycode/hello`.

This shows you the file you just created, its permissions, ownership, size, the timestamp of its last change, and more.

See for more information on editing files with emacs Section 2 and *Getting started with Emacs* [21] and the references in that document.

4.4 Executing a Python script

Since a Python script is simply text, we need to explicitly invoke Python to run it:

```
$ python hello.py
Hello world!
```

However, this becomes a bit tiresome after a while. Instead, we will tell the file that it ought to be run with Python, using emacs to add a few lines to our script:

Listing 2: hello.py: a *hello-world* program in Python.

```
1 #!/bin/env python
2 # -*- coding: utf-8 -*-
3
4 """hello.py: a hello-world program in Python."""
5
6 print('Hello world!')
```

The first line is called the *shebang* and specifies Python as the interpreter, while the second line specifies how the text file is encoded.[34] Line 4 is called a *docstring* because of the pair of triple double quotes bracketing the text and is used in Python to write explanatory documentation that can later be extracted, *e.g.* to generate a manual.[35]

The shebang tells the script what to do if it is executed, but it does not have execute *permissions* yet. We can change that with

```
$ chmod u+x hello.py
```

This adds (`+`) executable (`x`) permissions to the user (`u`) (*i.e.* owner) of the file. We can check that the file is now longer than before and has an extra `x`:

```
$ ls -l
-rwxr--r-- 1 student student 116 Feb 5 17:29 hello.py
```

Our Python script has now been turned into a program that can be executed by prefixing `./` to the name of the program:

```
$ ./hello.py
Hello world!
```

4.5 Adding your code to git

We will now create a git repository and add our Python script `hello.py` to it. This repository can then also contain other code in other subdirectories. Therefore, we will go up one directory:

```
$ cd ..
```

You should check that you are now indeed in the directory `~/mycode/` before you continue. When we create the git repository here, all subdirectories, including `hello/`, will be part of it.

Setting up a repository is described in Section 3.1. Carry out those steps, and then return here. You can now add the file `hello.py` using the **`git`** command by[36]

[34] This is not strictly needed here, but makes your file more portable.
[35] For example using Sphinx [22].
[36] Note that you typically want to add source code or other text files to a git repository, not binary files.

```
$ cd hello
$ git add hello.py
```

or in a single command, with

```
$ git add hello/hello.py
```

We can now commit this version of our great program with

```
$ git commit -a -m 'Initial version of hello/hello.py'
```

If you want to create a remote repository in the cloud, follow Section 3.2 and push your code to GitHub.[37] This will ensure that you have a backup[38] and/or that you can share your code.

4.6 Man pages

While developing our code, we will use the **man pages** to look up function prototypes of system calls and functions from the standard C library, header files that must be included and more.[39] Section 7 of EUBS [1] provides more details on the man pages and how to navigate them. In short, use ↑ ↓ to scroll, Space to skip a screenfull, / to search, n for the next hit, g/G to jump to the top/bottom of the page and q to quit. See Section 8.4 of EUBS or [2, 3] to see how to add colours to your man pages and make them easier to read and navigate. As an exercise, type `man ls` to see what the command `ls -alrt` does.

4.7 Setting up your environment

The configuration of your bash environment is done in ~/.bashrc and ~/.bash_profile, that of emacs in ~/.emacs.d/init.el or ~/.emacs and that of git in ~/.gitconfig.

You are encouraged to use the example configuration files at [2] or [3] to enhance your bash, emacs and git environments with colours, aliases and more. See Section 8 of EUBS [1] for more details. Note that changes in `.bashrc` only take effect when a new shell is started (*e.g.* by typing `bash`).

Appendices

Appendix A References

[1] **van der Sluys, M.** *Efficient use of the Linux command line in the Bash shell.* URL http://pub.vandersluys.nl. Visited 2022-04-19.

[2] ——. *My terminal configuration.* URL https://github.com/MarcvdSluys/MyTerminalConfig. Visited 2022-04-19.

[3] ——. *Environment settings (bash, emacs, git).* URL https://github.com/MarcvdSluys/han-ese-ops-env. Visited 2019-02-11.

[4] **Fox, B. & Ramey, C.** *Readline.* URL https://tiswww.case.edu/php/chet/readline/rltop.html. Visited 2022-04-21.

[37] Alternatively you can use Bitbucket or GitLab.

[38] You would not be the first student who loses her data and has to restart programming from scratch. In particular when you are using a *virtual machine* to run Linux, a corruption of your disc image easily occurs, in which case you may lose *all* your data.

[39] When programming in C, it is often possible to copy a piece of code from a man page to use as a template for (a part of) your program.

[5] **Linux, A.** *Arch Wiki: Readline*. URL https://wiki.archlinux.org/index.php/Readline. Visited 2016-08-02.

[6] **Hamano, J. & al.** *git*. URL https://git-scm.com. Visited 2017-08-29.

[7] **Newham, C. & Rosenblatt, B.** *Learning the Bash Shell: Unix Shell Programming*. O'Reilly Media, 2005.

[8] **Cooper, M.** *Advanced Bash-Scripting Guide*, 2014. URL http://tldp.org/LDP/abs/html/. Visited 2016-08-10.

[9] **Dominick, C.** *Org mode for Emacs*. URL https://orgmode.org. Visited 2022-04-08.

[10] **Stallman, R.** *Emacs*. URL https://www.gnu.org/software/emacs/. Visited 2016-03-20.

[11] **Yegge, S.** *Effective Emacs*. URL https://sites.google.com/site/steveyegge2/effective-emacs. Visited 2017-11-12.

[12] **Röthlisberger, D.** *How to learn Emacs*. URL http://david.rothlis.net/emacs/howtolearn.html. Visited 2017-11-12.

[13] **GNU**. *GNU Emacs reference cards*. URL https://www.gnu.org/software/emacs/refcards/. Visited 2017-11-13.

[14] **GitHub**. *Applying for a student developer pack*. URL https://help.github.com/articles/applying-for-a-student-developer-pack/. Visited 2018-12-18.

[15] ——. *Inviting collaborators to a personal repository*. URL https://help.github.com/articles/inviting-collaborators-to-a-personal-repository/. Visited 2018-12-18.

[16] **AstroFloyd**. *Git for coworkers*. URL https://astrofloyd.wordpress.com/2012/12/16/git-for-coworkers/. Visited 2017-08-29.

[17] **GitHub**. *Git Cheat Sheets*. URL https://training.github.com/. Visited 2022-04-20.

[18] ——. *About git*. URL https://docs.github.com/en/get-started/using-git/about-git. Visited 2022-04-20.

[19] ——. *GitHub guides*. URL https://guides.github.com. Visited 2020-04-10.

[20] **Chacon, S. & Straub, B.** *Pro git*. Springer Nature, 2014. URL http://git-scm.com/book. Visited 2022-04-20.

[21] **van der Sluys, M.** *Getting started with Emacs*. URL http://pub.vandersluys.nl. Visited 2022-04-13.

[22] **Georg Brandl and the Sphinx team**. *Sphinx – Python Documentation Generator*, 2007–2019. URL https://www.sphinx-doc.org. Visited 2020-02-12.

Appendix B Index