

B1 Topic 1: Introduction to the Shell

It's time to get started with the Shell. To open a shell prompt (where you can type commands), you first need a terminal. Your device probably shipped with one installed, or you can install one fairly easily. Once our terminal is set up, we can get started with some basic commands.

See below for an introduction and instructions for installing a terminal.

Getting started with the shell

Watch this video for a quick overview to get started in Bash.

Opening a Shell


For this course, you need to be using a Unix shell like Bash or ZSH.

MacOS and **Linux** users should already have a Unix shell application installed; search for ***Terminal*** in your list of applications (usually under Utilities or System).

Windows does not have a Unix shell installed by default. We recommend installing Bash in your Anaconda Environment.

You can do this by:

- Selecting the Environments tab in Anaconda Navigator
- From the packages drop-down menu, select All (instead of Installed)
- Install the following packages: `m2-base`, `m2-bash`, `m2-bash-completion`, `m2-git`, `m2-tar`, `m2-zip`, `m2-unzip`
- You can now open an Anaconda Command Prompt, and run the `bash` command to start Bash.

Alternately, if you wish to install a Unix shell in your Windows environment you can use [Windows Subsystem for Linux](https://docs.microsoft.com/en-us/windows/wsl/)  (<https://docs.microsoft.com/en-us/windows/wsl/>) or a Linux virtual machine to use Unix-style command-line tools.

>_ Using the shell

When you launch your terminal, you will see a *prompt* that often looks a little like this:

```
user@system:~$
```

This is the main textual interface to the shell. It tells you that:

- You are logged in as user `user` on the machine `system`
- That your "current working directory", or where you currently are, is `~` (short for "home").
- The `$` tells you that you are not the root user (more on that later).
- Sometimes this prompt may exclude the username or only contain the `$`

At this prompt you can type a *command*, which will then be interpreted by the shell. The most basic command is to execute a program:

```
user@system:~$ date
Fri 10 Jan 2020 11:49:31 AM EST
user@system:~$
```

Here, we executed the `date` program, which (perhaps unsurprisingly) prints the current date and time.

The shell then gives us another *prompt* so we can type another command to execute. We can also execute a command with *arguments*:

```
user@system:~$ echo hello
hello
```

In this case, we told the shell to execute the program `echo` with the argument `hello`. The `echo` program simply prints out its arguments. The shell parses the command by splitting it by whitespace, and then runs the program indicated by the first word, supplying each subsequent word as an argument that the program can access.

If you want to provide an argument that contains spaces or other special characters (e.g., a directory named "My Photos"), you can either quote the argument with `'` or `"` (`"My Photos"`), or escape just the relevant characters with `\` (`My\ Photos`).

But how does the shell know how to find the `date` or `echo` programs? Well, the shell is a programming environment, just like Python, and so it has variables, conditionals, loops, and functions (more on that later). When you run commands in your shell, you are really writing a small bit of code that your shell interprets. If the shell is asked to execute a command that doesn't match one of its programming keywords, it consults an *environment variable* called `$PATH` that lists which directories the shell should search for programs when it is given a command:

```
user@system:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
user@system:~$ which echo
/bin/echo
user@system:~$ /bin/echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

When we run the `echo` command, the shell sees that it should execute the program `echo`, and then searches through the `:`-separated list of directories in `$PATH` for a file by that name. When it finds it, it runs it (assuming the file is *executable*; more on that later). We can find out which file is executed for a given program name using the `which` program. We can also bypass `$PATH` entirely by giving the full location or *path* to the file/program we want to run.

? Shell vs Terminal

Are you still wondering at the differences between a shell and a terminal?



executing scripts.

The image to the left is a *terminal* for a mainframe computer. Terminals were (and are) input/output devices to display text and receive keyboard input. In a desktop environment, terminals fulfill the same function - simply a graphical window to interact textually with the command line. In contrast, a *shell* is a program executing inside the terminal window that allows the user to access the operating system's services, such as the file system, by typing commands in the terminal program or



Navigating the shell

A path on the shell is a list of directories/folders; separated by `/` on Linux and macOS and `\` on Windows. On Linux and macOS, the path `/` is the "root" of the file system, under which all directories and files lie, whereas on Windows there is one root for each disk partition (e.g., `C:\`).

We will generally assume that you are using a Linux filesystem in this class. A path that starts with `/` is called an *absolute* path. Any other path is a *relative* path. Relative paths are relative to the current working directory, which we can see with the `pwd` command and change with the `cd` command. In a path, `.` refers to the current directory, and `..` to its parent directory:

```
user@system:~$ pwd
/home/user
user@system:~$ cd /home
user@system:/home$ pwd
/home
user@system:/home$ cd ..
user@system:/$ pwd
/
user@system:/$ cd ./home
user@system:/home$ pwd
/home
user@system:/home$ cd user
user@system:~$ pwd
/home/user
user@system:~$ ../../bin/echo hello
hello
```

Notice that our shell prompt kept us informed about what our current working directory was.

In general, when we run a program, it will operate in the current directory unless we tell it otherwise. For example, it will usually search for files there, and create new files there if it needs to.

To see what files and folders are in a given directory, we use the `ls` command:

```
user@system:~$ ls
user@system:~$ cd ..
user@system:/home$ ls
user
user@system:/home$ cd ..
user@system:/$ ls
bin
boot
dev
etc
home
...
```

Unless a directory is given as its first argument, `ls` will print the contents of the current directory. Most commands accept flags and options (flags with values) that start with `-` to modify their behavior. Usually, running a program with the `-h` or `--help` flag will print some help text that tells you what flags and options are available. For example, `ls --help` tells us:

```
-l      use a long listing format
```

```
user@system:~$ ls -l /home
drwxr-xr-x 1 user  users  4096 Jan 01  2022 user
```

This gives us a bunch more information about each file or directory present. First, the `d` at the beginning of the line tells us that `user` is a directory. Then follow three groups of three characters (`rwX`). These indicate what permissions the owner of the file (`user`), the owning group (`users`), and

everyone else respectively have on the relevant item. A `-` indicates that the given principal does not have the given permission. Above, only the owner is allowed to modify (`w`) the `user` directory (i.e., add/remove files in it). To enter a directory, a user must have "search" (represented by "execute": `x`) permissions on that directory (and its parents). To list its contents, a user must have read (`r`) permissions on that directory. For files, the permissions are as you would expect. Notice that programs usually have the `x` permission set for the last group, "everyone else", so that anyone can execute them.



Useful basic commands

There are hundreds of command line programs and utilities that perform various different tasks. Some other handy programs to know about at this point are:

Create an empty file

The `touch` command modifies file properties.

It can be used to quickly create files:

```
user@system:~$ touch foo
user@system:~$ ls
foo
```

Display the contents of a (plain text) file

The `cat` command displays the contents of a file.

It takes the path to a file as an argument and outputs that file to the shell. This works best with small text-based files:

```
user@system:~$ cat foo
The quick brown fox jumps over the lazy dog
user@system:~$
```

Copy a file

The `cp` command copies a file.

It takes 2 arguments; the name/path of the file to copy, and the name/path of the destination:

```
user@system:~$ ls
foo
user@system:~$ cp foo bar
user@system:~$ ls
bar  foo
```

Use the `-r` flag to copy a folder and its contents; see `--help` for more details on this and other options.

Move a file

The `mv` command moves a file.

It takes 2 arguments; the name/path of the file to move, and the name/path of the destination:

```
user@system:~$ ls
foo
user@system:~$ mv foo bar
user@system:~$ ls
bar
```

Rename a file

Wait, didn't we just do that?

There is no command only for renaming files; we rename files by moving them within the same directory, but with a different file name (just as we did above)

Delete a file

The `rm` command deletes files.

It takes 1 argument; the name/path of the file to delete:

```
user@system:~$ ls
foo
user@system:~$ rm foo
user@system:~$ ls
```

Use the `-r` flag to delete a directory and its contents; see `--help` for more details on this and other options.

Create a directory/folder

The `mkdir` command creates a directory.

It takes the name name/path of the directory to create as an argument:

```
user@system:~$ mkdir baz
mkdir: created directory 'baz'
```

Use the `--parents` flag to create a subfolder and its parent folders if those don't already exist; see `--help` for more details on this and other options.

Get more information about a program

If you ever want *more* information about a program's arguments, inputs, outputs, or how it works in general, give the `man` program a try. It takes as an argument the name of a program, and shows you its *manual page*. Press `q` to exit.

```
user@system:~$ man ls
```



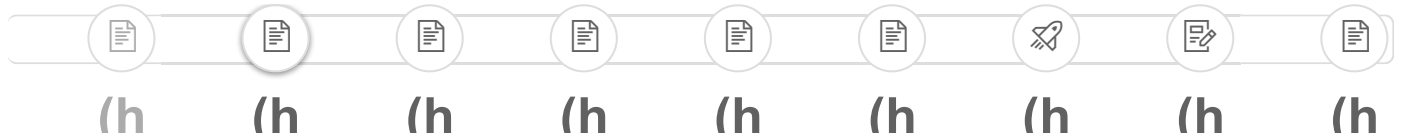
Next Steps

At this point you know your way around a shell enough to accomplish basic tasks. You should be able to navigate around to find files of interest and use the basic functionality of most programs.

Next, we will look at how to perform and automate more complex tasks using the shell and the many handy command-line programs out there.

This page includes content from [MIT's Missing Semester](https://missing.csail.mit.edu/) [\(https://missing.csail.mit.edu/\)](https://missing.csail.mit.edu/). Text content on this page is licensed under Creative Commons

[\(https://creativecommons.org/licenses/by-nc-sa/4.0/\)](https://creativecommons.org/licenses/by-nc-sa/4.0/).



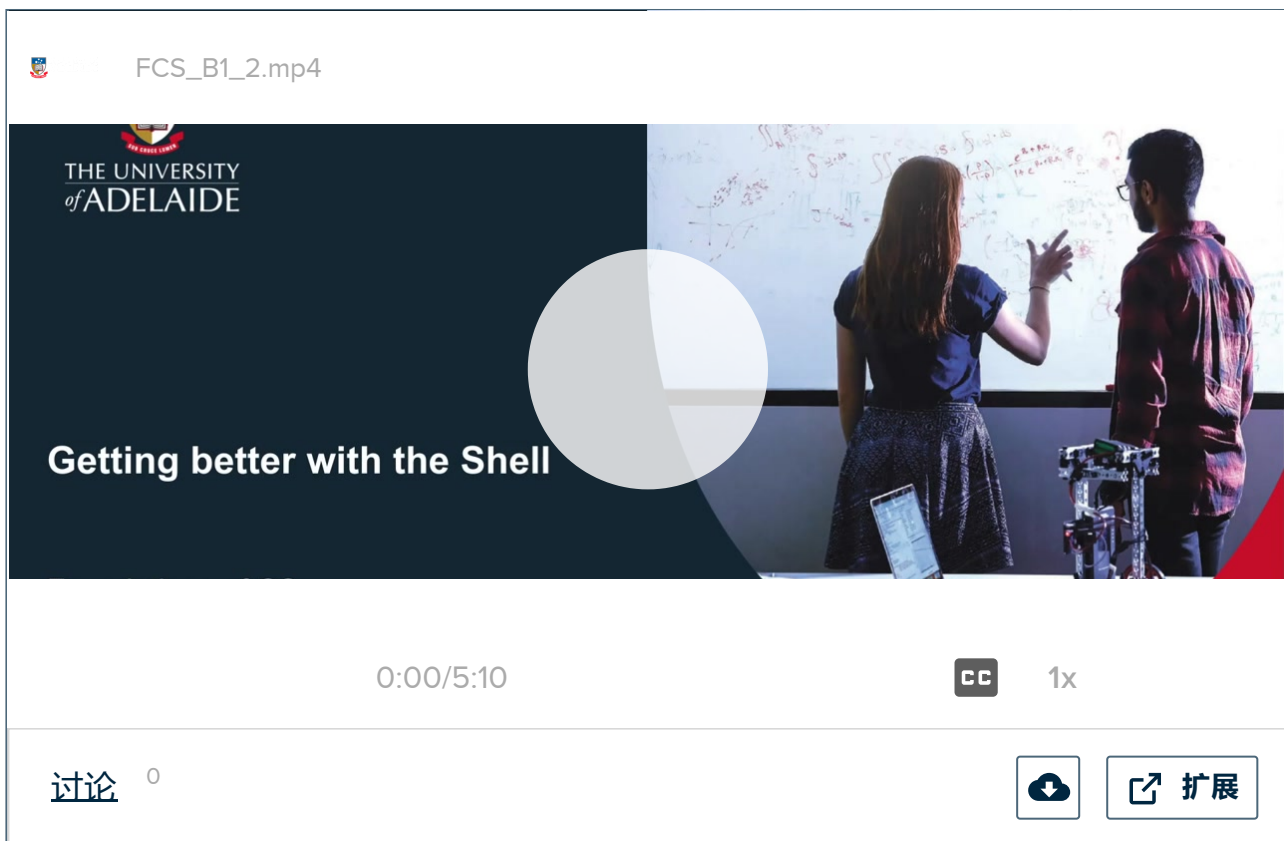
B1 Topic 1: Shell Techniques & Tools

So far we're familiar with some of the basics of the shell, but have only just scratched the surface of what this tool can do.

To make the most of our shell, we will be looking at techniques for improving the way that we work with the shell, some more advanced commands, and how we can package commands together into scripts that we can reuse.

Getting better with the Shell

Watch this video for a quick overview of useful shell techniques:



Speeding things up

Typing out shell commands can seem unwieldy and slow at first, especially when we make mistakes or need to re-enter long commands.

Thankfully, shells like Bash provide a variety of shortcuts to help us quickly perform basic and tedious tasks:

Stop a running command

Sometimes you will need to cancel a running command, and some commands run forever, until they are manually cancelled.

A running command can be stopped with **Ctrl+C** (think **C**ancel or **C**lose)

This is also useful for deleting a partially typed command; if you're part way through entering a command and need to reset/delete it to enter a different one, **Ctrl+C** will cancel what you've typed and give you a fresh prompt:

```
user@system:~$ halfway through typi^C
user@system:~$
```

Notice the **^C** indicating when Ctrl+C was pressed.

Copy/paste

You may have realised that **Ctrl+C** is also the key used on many systems to Copy. How do we copy/paste in a terminal then?

For MacOS users, this is not a problem; **⌘+C** / **Command+C** still works fine (and now you know why MacOS doesn't use the Control/Ctrl key in desktop keyboard shortcuts).

For Windows/Linux users it will depend on your Terminal program, but it's usually **Ctrl+Shift+C** to Copy, and **Ctrl+Shift+V** to Paste.

You can usually also right-click for a pop-up menu that will contain these options.

Repeat previous commands

Often you'll want to use commands that you've previously used. Thankfully terminals like Bash contain a command history that you can access.

You can cycle through previously used commands with the up and down arrow keys **↑** **↓**

The up key will show the previous command, while then down key will show the next command (if you're looking at a previous one)

You can also search through your past commands.

Ctrl+R will give you a search prompt that you can use to find previous commands:

```
(reverse-i-search)`':
```

Simply begin to type the command at the search prompt and the search will suggest the most recent matching command. For example, with **ls**:

```
(reverse-i-search)`ls': ls -l /home
```

You can cycle through suggestions by pressing **Ctrl+R** again, and once you've found the command you're looking for, you can run it directly with **Enter**/**Return**, or edit the command before running by moving the cursor (with the arrow keys)

Auto-completion

Typing file names and full paths can be time-consuming. Thankfully our terminal can help us with this!

The **Tab** key searches file, directory, and program names and will auto-complete them if it can do so. This is known as *Tab-completion*.

Where multiple options are available, it will partially complete the name as much as possible.

For example, given 3 files **foo123**, **bar123**, and **baz123**, if we want to display the contents of **foo123** using the **cat** command, we could type:

```
user@system:~$ cat f
```

Then pressing **Tab**:

```
user@system:~$ cat foo123
```

Notice what happens when we try the same with **bar123**:

```
user@system:~$ cat ba
```

Since **bar123** and **baz123** both being with **ba**, our shell auto-completes the common part of the name.

We can now either type the rest of the name, or type enough to separate our desired file name from the other files:

```
user@system:~$ cat bar
```

Then pressing **Tab**:

```
user@system:~$ cat bar123
```

When multiple options are available, we can also review the options by double-pressing **Tab**:

```
user@system:~$ cat ba  
bar123  baz123
```

Tab-completion also works for full file paths. For example, navigating the **/usr** directory:

```
user@system:/usr$ cd
bin/      games/  include/ lib/      lib32/  libexec/ local/  sbin/   share/  src/
user@system:/usr$ cd local/
bin/      etc/     games/  include/ lib/      man/     sbin/   share/  src/
user@system:/usr$ cd local/share/
ca-certificates/ emacs/          fonts/          jupyter/          man/          texmf/

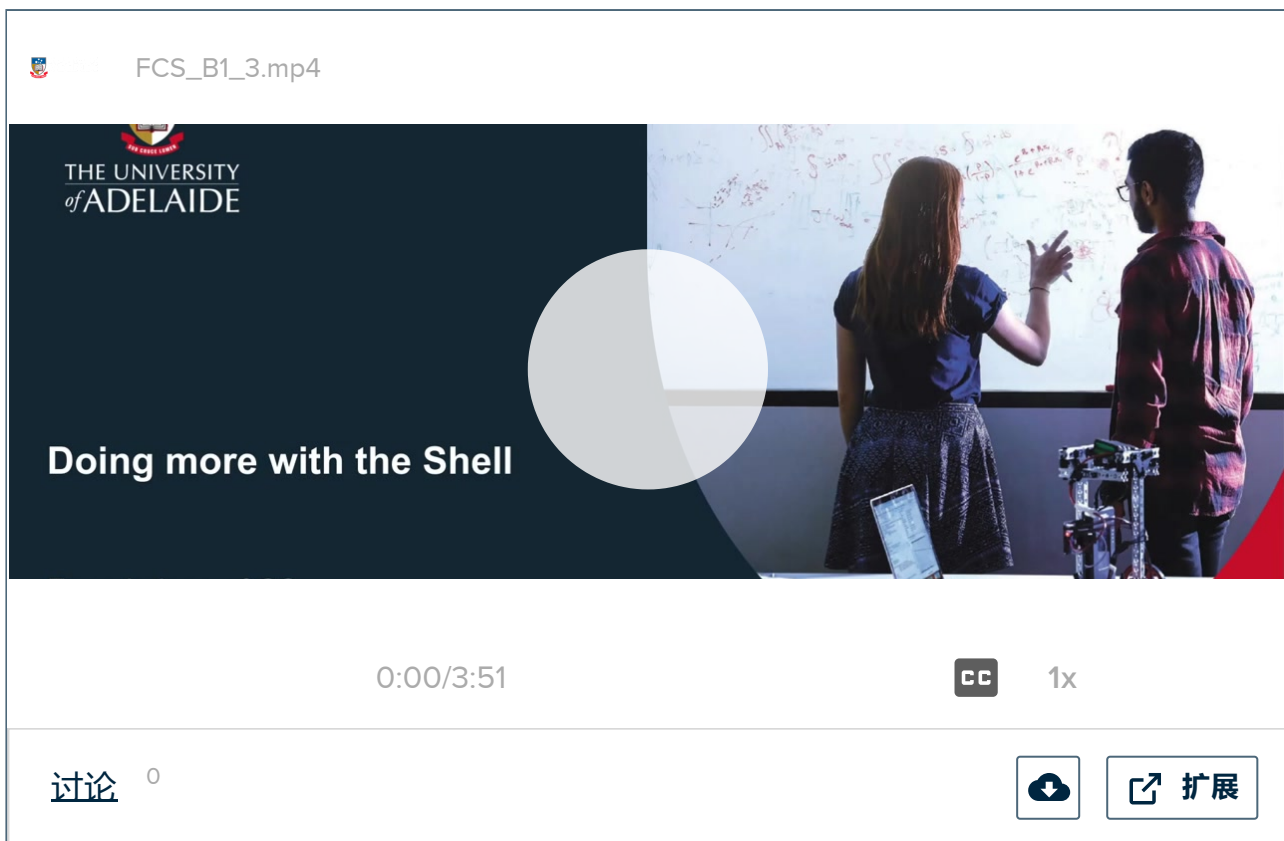
user@system:/usr$ cd local/share/jupyter/
kernels/      nbextensions/
user@system:/usr$ cd local/share/jupyter/kernels/python
```

Highlighted in pink are the inputs typed, and highlighted green is the tab-completed text. You can also see the suggestions from double-pressing tab.

Try this yourself!

Doing more with the Shell


Watch this video for a quick overview of more useful shell commands:



Additional useful commands

We've already seen quite a few basic commands. Here are a few more handy programs:

Finding files (by name)

One of the most common repetitive tasks that every programmer faces is finding files or directories. All UNIX-like systems come packaged with `find`  (<https://www.man7.org/linux/man-pages/man1/find.1.html>), a great shell tool to find files. `find` will recursively search for files matching some criteria. Some examples:

```
# Find all directories named src
find . -name src -type d

# Find all python files that have a folder named test in their path
find . -path '*/test/*.py' -type f

# Find all files modified in the last day
find . -mtime -1

# Find all zip files with size in range 500k to 10M
find . -size +500k -size -10M -name '*.tar.gz'
```

Beyond listing files, `find` can also perform actions over files that match your query. This property can be incredibly helpful to simplify what could be fairly monotonous tasks.

```
# Delete all files with .tmp extension
find . -name '*.tmp' -exec rm {} \;

# Find all PNG files and convert them to JPG
find . -name '*.png' -exec convert {} {}.jpg \;
```

Despite `find`'s ubiquitousness, its syntax can sometimes be tricky to remember.

Finding content in files

The `grep` command searches file contents.

It takes 2 arguments; the name/path of the file to copy, and the name/path of the destination:

```
user@system:~$ grep "thing to search for" path/to/file
This is the thing to search for in this file
```

Use the `-R` flag to search recursively (all subfolders and their files)

Arrange the content of files

The `sort` command arranges the lines of a file in alphabetical order and outputs to the Shell:

```
user@system:~$ cat foo
Some line
Another line
Last line
user@system:~$ sort foo
Another line
Last line
Some line
```

It has a variety of options/arguments that can be used to among other things, change the order used when sorting, ignore case, and skip duplicates.

Output only the start/end of a file

The `head` and `tail` commands can be used to output only the start/end of a file respectively:

```
user@system:~$ cat foo
Some line
Another line
Last line
user@system:~$ head -n 2 foo
Some line
Another line
user@system:~$ tail -n 2 foo
Another line
Last line
```

They also have a variety of options/arguments that can be used to specify how much of the file to show.

A common option (as shown above) is `-n`, which specifies the number of lines to show.

Compare files

The `diff` command compares files.

It takes 2 arguments; the name/path of each file to compare, then generates an output that shows all the places where the files differ:

```
user@system:~$ cat foo
Some line
Another line
Last line
user@system:~$ cat bar
First line
Second line
Last line
user@system:~$ diff foo bar
1,2c1,2
< Some line
< Another line
---
> First line
> Second line
```

If the files have the same contents, no output is produced:

```
user@system:~$ diff foo foo
```

Some common options are `-B` which ignores blank lines, `-i` which ignores case-sensitivity when comparing, and `-w` which ignores all differences in whitespace.

> Redirecting Input & Output

In the shell, programs have two primary "streams" associated with them: their input stream and their output stream. When the program tries to read input, it reads from the input stream, and when it prints something, it prints to its output stream. Normally, a program's input and output are both your terminal. That is, your keyboard as input and your screen as output. However, we can also rewire those streams!

The simplest form of redirection is `< file` and `> file`. These let you rewire the input and output streams of a program to a file respectively:

```
user@system:~$ echo hello > hello.txt
user@system:~$ cat hello.txt
hello
user@system:~$ cat < hello.txt
hello
user@system:~$ cat < hello.txt > hello2.txt
user@system:~$ cat hello2.txt
hello
```

Demonstrated in the example above, `cat` is a program that concatenates files. When given file names as arguments, it prints the contents of each of the files in sequence to its output stream. But when `cat` is not given any arguments, it prints contents from its input stream to its output stream (like in the third example above).

You can also use `>>` to append to a file. Normally, the shell will happily overwrite any existing file when you redirect output to that filename. In contrast, to append content to an existing file, use `>>` like:

```
user@system:~$ echo 'hello' > hello.txt
user@system:~$ echo 'and hello again!' >> hello.txt
user@system:~$ cat hello.txt
hello
and hello again!
```



Pipes

Where this kind of input/output redirection really shines is in the use of *pipes*. The `|` operator lets you "chain" programs such that the output of one is the input of another:

```
user@system:~$ ls -l / | tail -n1
drwxr-xr-x 1 root root 4096 Jun 20 2019 var
user@system:~$ curl --head --silent google.com | grep --ignore-case content-length | cut --delimiter
=' ' -f2
219
```

Pipes are the "killer feature" of the [Unix design philosophy](https://en.wikipedia.org/wiki/Unix_philosophy) (https://en.wikipedia.org/wiki/Unix_philosophy), which favours small, cohesive and highly functional commands of limited individual scope that take on powerful functionality when chained together using pipes such that the output of one command is passed to the next.

In the example above, the command `ls -l` is "piped" to the `tail -n1` command to show only the last file listed in that directory. This exhibits the above mentioned philosophy by allowing the "ls" command to simply perfect listing files while allowing filtering of the output to be left to a subsequent, piped command. This approach of chaining together commands via pipes can be used with almost any command line program to create pipes of great length - building sophisticated behaviour on the fly from commands of seemingly limited individual utility.



Globbering

When launching scripts, you will often want to provide arguments that are similar. Bash has ways of making this easier, expanding expressions by carrying out filename expansion. These techniques are often referred to as shell *globbing*.

- Wildcards - Whenever you want to perform some sort of wildcard matching, you can use `?` and `*` to match one or any amount of characters respectively. For instance, given files `foo`, `foo1`, `foo2`, `foo10` and `bar`, the command `rm foo?` will delete `foo1` and `foo2` whereas `rm foo*` will delete all but `bar`.
- Curly braces `{}` - Whenever you have a common substring in a series of commands, you can use curly braces for bash to expand this automatically. This comes in very handy when moving or converting files.

```
convert image.{png,jpg}
# Will expand to
convert image.png image.jpg

cp /path/to/project/{foo,bar,baz}.sh /newpath
# Will expand to
cp /path/to/project/foo.sh /path/to/project/bar.sh /path/to/project/baz.sh /newpath

# Globbing techniques can also be combined
mv *.py,.sh} folder
# Will move all *.py and *.sh files

mkdir foo bar
# This creates files foo/a, foo/b, ... foo/h, bar/a, bar/b, ... bar/h
touch {foo,bar}/{a..h}
touch foo/x bar/y
# Show differences between files in foo and bar
diff <(ls foo) <(ls bar)
# Outputs
# < x
# ---
# > y
```



Become a Superuser!

On most Unix-like systems, one user is special: the "root" user. You may have seen it in the file listings above. The root user is above (almost) all access restrictions, and can create, read,

B1 Topic 1: Basic Shell Scripting

It's time to dive deeper!


Hopefully you're getting proficient at quickly using the shell, but some of the more common tasks you do might need still seem tedious. Here, we'll look at how we can use scripting techniques to gain mastery and automate development.

>_ Shell Scripting

So far we have seen how to execute commands in the shell and pipe them together. However, in many scenarios, you will want to perform a series of commands and make use of control flow expressions like conditionals ("if this, then that") or loops ("do this thing n times").

A *shell script* is a small file containing a series of commands similar to those you would ordinarily execute from the command line. However, shell scripts are the next step in complexity and have their own *scripting language* with variables, control flow and its own syntax.

What makes shell scripting different from other scripting programming languages is that it is optimized for performing shell-related tasks. Thus, creating command pipelines, saving results into files, and reading from standard input are primitives in shell scripting, which makes it easier to use than general purpose scripting languages for such tasks.

Writing `bash` scripts can be tricky and unintuitive. There are tools like [shellcheck](https://github.com/koalaman/shellcheck)  (<https://github.com/koalaman/shellcheck>) that will help you find errors in your sh/bash scripts.

\$ Variables

A variable is symbolic name associated with a value. For example, we could use the variable name `street_number` to contain a value "40". In reading and writing a code to manipulate a street address, using the variable `street_number` makes our code readable and allows us to reassign a house number when changing addresses.

To assign variables in bash, use the syntax `street_number=40` and access the value of the variable with `$street_number`. In general, in shell scripts the space character will perform argument splitting. This behavior can be confusing to use at first, so always check for that. So `street_number = 40` will not work since it is interpreted as calling the `street_number` program with arguments `=` and `40`.

Strings in bash can be defined with `'` and `"` delimiters, but they are not equivalent.

If we wish to create a variable, `uni`, with the name of the institution as a string, we would write:


```
_(mailto:user@system:~$)_user@system:~$ uni='University of Adelaide'
```

```
user@system:~$ echo "$uni"  
University of Adelaide
```

In contrast, we can use strings delimited with `"` to evaluate the contents of the string programmatically. Thus:

```
user@system:~$ echo "$uni"  
University of Adelaide
```

while:

```
user@system:~$ echo '$uni'  
$uni
```

treats the string `'$uni'` in its literal form - as a simple string.

Already we can see the complexity of shell scripting - but don't despair: the content of this course is designed to provide you with the basic tools that you will need to prosper while embarked on the course and build a foundation for deeper learning throughout your studies.



Environment Variables

While the variables discussed above are local only to the running shell, we can use *environment variables* to make variables *global* to all of the commands invoked from the shell - in fact when the shell starts, a number of environmental variables are declared and made available to our scripts. You can, for example, display for yourself the following examples:

```
user@system:~$ echo $USER  
user  
  
user@system:~$ echo $HOME  
/Users/user  
  
user@system:~$ echo $SHELL  
/bin/bash  
  
user@system:~$ env  
# shows all the environment variables seen by this process
```

You can also define your own environment variables using the builtin bash command `export`. For example:

```
user@system:~$ export EXPECTED_SCORE='HD'  
  
user@system:~$ echo "$EXPECTED_SCORE"  
HD
```



Your Turn: Executing a Script & Shebang

Let's write and execute a first bash script.

Using a text editor, copy and paste the following text (covering the commands above) into a file `hello.bash` and save:

```
echo 'Hello!'
echo $USER
echo $HOME
echo $SHELL
```

Now, using the command line, navigate to the directory where your first script is stored and run the script. There are numerous ways to run the script but first we will try:

```
user@system:~$ bash hello.bash
Hello!
user
/Users/user
/bin/bash
```

Congratulations... your first shell script!

However, to the shell, the script looks like a simple text file - so we have had to explain how to run the script with `bash hello.bash`. We can embed the knowledge of what to do with the script by including a command at the top of the script that explains to the shell which interpreter to use when the script is invoked.

The directive to the shell has the name "*shebang*" or "*hash-bang*" from its identifying characters `#!` - the name coming from an inexact contraction of "sharp" or "hash" (`#`) and exclamation mark or "bang" (`!`).

Let's add a shebang line to the top of your script:

```
#!/bin/bash

echo 'Hello!'
echo $USER
echo $HOME
echo $SHELL
```

We are almost there! To execute the script from the command line, we call the script with its *path*. Since the current directory of our shell is the same directory as the script, our path is denoted by the character `.` - so execution looks like:

```
user@system:~$ ./hello.bash
-bash: ./hello.bash: Permission denied
```

Ah no! One more thing, we have to give ourselves permission to run the script like:

```
# change file mode to add user execute permission
user@system:~$ chmod u+x hello.bash


user@system:~$ ./hello.bash
Hello!
matt
/Users/matt
/bin/bash
```


Don't worry too much about the `chmod` command, but note that to execute a script using a shebang line, we need to use this incantation `chmod u+x "$filename"` to allow ourselves to execute a file.

Note that scripts need not necessarily be written in bash to be called from the terminal. For instance, here's a simple Python script:

```
#!/usr/local/bin/python

print ("hello!")
```

The kernel knows to execute this script with a python interpreter instead of a shell command because we included a [shebang](https://en.wikipedia.org/wiki/Shebang_(Unix))  [_line](https://en.wikipedia.org/wiki/Shebang_(Unix)) at the top of the script.

It is good practice to write shebang lines using the `env`  [_command](https://www.man7.org/linux/man-pages/man1/env.1.html) that will resolve to wherever the command lives in the system, increasing the portability of your scripts. To resolve the location, `env` will make use of the `PATH` environment variable we introduced in the first lecture. For this example the shebang line would look like `#!/usr/bin/env python`.



Arguments & Return Codes

Unlike other scripting languages, bash uses a variety of special variables to refer to arguments passed to the script, error codes that indicate whether the script ran successfully or how it failed, and other relevant variables.

An argument is, most simply, a space separated list of the values following the name of the script when invoked from the command line like: `./hello.bash Angelina` where "Angelina" is an argument value passed into the script for its use during execution.

Below is a list of some of these special variables. A more comprehensive list can be found [here](https://tldp.org/LDP/abs/html/special-chars.html) [_.](https://tldp.org/LDP/abs/html/special-chars.html)

- `$0` - Name of the script
- `$1` to `$9` - Arguments to the script. `$1` is the first argument and so on.
- `$@` - All the arguments
- `##` - Number of arguments
- `$?` - Return code of the previous command

- `$$` - Process identification number (PID) for the current script
- `!!` - Entire last command, including arguments. A common pattern is to execute a command only for it to fail due to missing permissions; you can quickly re-execute the command with `sudo` by doing `sudo !!`
- `_` - Last argument from the last command. If you are in an interactive shell, you can also quickly get this value by typing `Esc` followed by `.` or `Alt+.`

`$0` - `$9` are called *positional parameters*. In a command to the shell to invoke a script like:

```
jingle-bells.sh Dasher Dancer Prancer Vixen Comet Cupid Blitzen Rudolph
```

`$0` contains the name of the script `jingle-bells.sh`, while `$1` through `$8` contain the names of eight of Santa's reindeers in the order they were provided as arguments to the script.

Commands will often return output using `STDOUT`, errors through `STDERR`, and a Return Code to report errors in a more script-friendly manner. The return code or exit status is the way scripts/commands have to communicate how execution went. A value of 0 usually means everything went OK; anything different from 0 means an error occurred.

The `true` program will always have a 0 return code and the `false` command will always have a 1 return code.

Let's modify our script in a text editor to use these special variables and use an argument:

```
#!/bin/bash

echo $USER
echo $HOME
echo $SHELL

echo $0      # Name of the script
echo $@      # All of the arguments

printf 'Hello %s\n' $1
```

Ignoring the syntax of the `printf` command for now, if we invoke our script with several arguments, we can see how the passed values can be used inside the script:

```
user@system:~$ ./hello.bash Angelina Jasmin

user
/Users/user
/bin/bash
./hello.bash
Angelina Jasmin
Hello Angelina
```

Our use of the `printf` command prints a formatted string where the value `%s` is replaced by the following argument, in this case `$1`.



Conditional Statements & Iteration

As mentioned above, bash and other shells, are fleshed out with detailed scripting and builtin command features that facilitate automation of many tasks that we'd generally conduct in a command line shell. It's not within the scope of this course to provide a detailed insight into anything more than basic shell programming (although see the Resources section below if you are interested in pursuing a detailed understanding).



Resources

Shell scripting is a rich and detailed topic of investigation. If you are interested in enhancing your basic Unix skills, these texts are recommended:







1. Ryder, T (2018), [Bash Quick Start Guide: Get up and Running with Shell Scripting with Bash](https://librarysearch.adelaide.edu.au/permalink/61ADELAIDE_INST/1eubam4/cdi_proquest_ebookcentral_EBC5532285) (https://librarysearch.adelaide.edu.au/permalink/61ADELAIDE_INST/1eubam4/cdi_proquest_ebookcentral_EBC5532285), (https://librarysearch.adelaide.edu.au/permalink/61ADELAIDE_INST/1eubam4/cdi_proquest_ebookcentral_EBC5532285) 1st edition, Packt Publishing, Limited, Birmingham.
2. Ebrahim, M & Mallett, A (2018), [Mastering Linux Shell Scripting: A Practical Guide to Linux Command-Line, Bash Scripting, and Shell Programming](https://librarysearch.adelaide.edu.au/permalink/61ADELAIDE_INST/1eubam4/cdi_proquest_ebookcentral_EBC5371676), (https://librarysearch.adelaide.edu.au/permalink/61ADELAIDE_INST/1eubam4/cdi_proquest_ebookcentral_EBC5371676) 2nd Edition, Packt Publishing, Limited, Birmingham.



Next Steps

Hopefully this has been a useful introduction into the basics of writing and using a bash shell script. You should have all you need to complete the Quiz on practicing shell features and scripts. Working with the shell will be a skill you practice regularly as a programmer, and one you will be constantly learning and refining. Good luck!

Next week we'll be looking at how we can manage, backup, and collaborate on our code using version control and Git.

This page includes content from [MIT's Missing Semester](https://missing.csail.mit.edu/)  (<https://missing.csail.mit.edu/>). Text content on this page is licensed under Creative Commons      (<https://creativecommons.org/licenses/by-nc-sa/4.0/>).



• B1 Topic 2: Introduction to Version Control & Git Basics



Getting Started with Version Control & Git

Watch this video for a quick overview to get started with Version Control & Git.



What is Version Control and Why do we use it?

When working on a large project, how do you manage your files?

You might need to keep different versions of the work to refer to later as you make changes.

Perhaps you're working with

others and need to keep track of everyone's changes; it's still common practice in many places to do this via email, sending copies of documents and revisions back and forth, using the inbox or folders and file names to keep track which revision of a file is current.

What about keeping backups?



final-5



final-2



final-1



final-0

Have you ever had 4 different "final" versions of a file?

Programmers have long-since recognised this problem and sought to develop solutions. Version control systems are the result.

While there are many different version control systems, most of them share a few common features:


- Ways to optimise the storage of files across many changes.
- The user chooses when to store the next revision.
- Add a note/message whenever a revision is saved, allowing us to quickly understand what was changed in a given revision.
- Ability to easily restore specific revisions.
- Tools to easily compare revisions.
- Tools to easily combine multiple sets of independent changes.

You might recognise some of these features from online document and backup systems like Google Docs or Dropbox. Indeed, many of these have been influenced by the version control systems that programmers use.

In this course, we'll be using the Git version control system, currently the most widely used version control system, managing billions of lines of code across many software projects including some of the largest software projects under development!

Getting Set up

To get started, you'll need to have Git installed on your system.

- **CS50 Sandbox** users should already have Git installed.
- **MacOS, Linux, and Windows** users can follow the guide here <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>  (<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>)

Once Git is installed, open a shell and navigate to the folder that you want to manage with Git. While there are graphical clients for Git, we'll be using the shell client in this course.

1. A basic Git work-flow

To get started with Git, consider the following basic work-flow:

1. Initialise a new local **repository**, or **clone** an existing remote one

If you're starting a new project or don't yet have a repository set up, navigate your shell to the directory you want to use, and run:

```
git init
```

2. **Stage** files

If you have existing files, or as you create new ones, you can stage the files by running:

```
git add file1.py file2.py
```

where `file1.py` and `file2.py` are the files you want staged.

3. **Commit** changes

Once you've staged all your files, you can now save a revision:

```
git commit -m "A message about what's changed in this revision"
```

This makes a new revision containing all of the staged files.

4. (optional) **Push** changes

If your repository is also stored on a remote server, you can upload your committed revisions:

```
git push
```

5. Get the **Status** of your repository

```
git status
```

Don't worry if you don't understand some of these concepts just yet; we're about to go over them in more detail.



Key Concepts

Repository

A collection of files and folders (in Git referred to as "blobs" and "trees"), and revisions of those files and folders.

Usually a repository is used to manage a single project, with additional projects each getting their own repositories.

Working Copy

When you initialise or clone a repository, a copy of that whole repository resides locally on your system. This is known as a Working Copy.

Staging

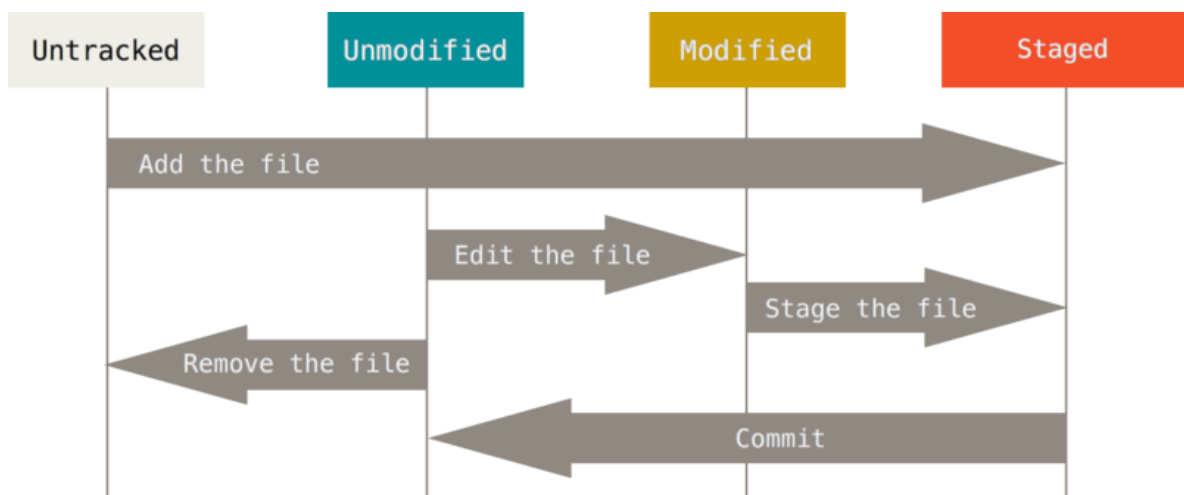
Files in a working copy can be marked for inclusion in the next revision using the `git add` command. These files are considered "staged".

Staged files can be modified and un-staged as needed, up until the next revision is committed.

Commit

Committing saves a revision of the currently staged files. When committing you'll need to include a message/comment about what has changed. This allows easy review of your work later.

Once a new revision has been committed, the staged files are reset/marked as unmodified



The different states of files in Git

Source: <https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>. [↗ \(https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository\)](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository)

Now, let's revisit the commands from the basic work-flow earlier...

>_ Your Turn: Create a Repository/Working Copy

If you're starting a new project or don't yet have a repository set up, navigate your shell to the directory you want to use, and run:

```
mkdir myrepo
cd myrepo
git init
```

This will initialise a new git repository in that directory.

If you already have a repository setup on a remote server (such as Github), you can clone that

repository to make a local **working copy** of that repository ready to work on instead of initialising a new repository:

```
git clone https://url-of-remote.com/repository.git
```

Cloning a repository will make a working copy of that repository in a new directory. You generally don't want to clone a repository inside of a repository. Cloning a repository grabs from a remote server all of the files that you need to create your own working copy and looks something like this:

```
user@system $ git clone https://github.com/libgit2/libgit2

Cloning into 'libgit2'...

remote: Enumerating objects: 117162, done.
remote: Counting objects: 100% (117162/117162), done.
remote: Compressing objects: 100% (32120/32120), done.
remote: Total 117162 (delta 83068), reused 117159 (delta 83065), pack-reused 0
Receiving objects: 100% (117162/117162), 59.30 MiB | 6.01 MiB/s, done.
Resolving deltas: 100% (83068/83068), done.
Updating files: 79% (9085/11500)
```

You can see here an in-progress snapshot of cloning the git repository of... git. About 50 MBytes of open source code.



Staging Changes

If you have existing files, or as you create new ones, you can stage the files by running:

```
git add file1.py file2.py
```

where `file1.py` and `file2.py` are the files you want staged. You could create empty copies of these files with the command:

```
touch file1.py file2.py
```

Each time you commit your work, the staged files are reset, that is, the files are no longer staged.

This means that whenever you make changes to files that you want included in your next revision, you'll also need to add those files to re-stage them for inclusion.

While this can seem a bit cumbersome at first, allows a level of control not available in other version control systems.

That said, there are a few ways we can speed this up:

- We can use wildcards to add multiple files:

```
git add file*.py
```

which would add all files that begin with file and end in .py
or

```
git add *
```

which would add **all** files in the current folder. Be careful with this one.

- We can also add whole folders in the same way:

```
git add my_directory/
```

which would add my_directory and all the files contained therein. *Note: empty folders cannot be added until they contain at least 1 file.*

- We can use the -a flag when committing to automatically add all modified files that have been previously added (see *Committing* below)

At this point, the files are ready to be included in your next revision and are considered under version control.



Reviewing Changes & Current Status

We can check on the status of our repository with:

```
git status
```

Which tells us the current status of files in our repository, for example:

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   file1.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:   file3.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file2.py
    file4.py
```

In this example, there are 4 files:

- file1.py is new and staged to be added in the next commit.
- file3.txt has been modified, but is not currently staged.
- file2.py and file4.py have not been added.



Staging other changes

We also need to stage changes when we rename, move, or delete files.

This can be done with `git add`:

```
# Delete a file, then stage the deletion
rm file1.py
git add file1.py

# Move/rename a file, then stage the move
mv file1.py file5.py
git add file1.py
git add file5.py
```

... but can get a bit unwieldy given how common these operations are.

Thankfully Git provides its own version of these commands that stages the changes automatically:

```
# Delete a file and stage the deletion
git rm file1.py

# Move/rename a file, then stage the move
git mv file1.py file5.py
```



Committing Changes Locally

Once you've staged all your files, you can now save a revision:

```
git commit -m "A message about what's changed in this revision"
```

This makes a new revision containing all of the staged files. A revision is an identified set of the state of all files in the repository that we can reference later - for example, returning to an earlier revision of a file if recent changes have added some unwanted behaviour.

Files that were added in previous revisions can also be automatically included if they were changed with the `a` flag:

```
git commit -am "A message about what's changed in this revision"
```



Saving Changes Remotely

If your repository is also stored on a remote server, you can upload your committed revisions:

```
git push
```

We'll look more into working with remote repositories later this week.

Your turn for some practice

Let's put our script files from Week 1 under version control.

After each of the following steps, run `git status` to see what has changed.


If you get stuck on a command, try `git --help` for more information.

1. Create a new repository for your script files from Week 1
2. Now add your script files.
3. Create a new file named `git.txt` using the touch command
4. Commit your changes with a comment.
5. Add and commit your git.txt file.
6. Edit git.txt to contain some text.
7. Stage and commit your changes.
8. Rename or delete git.txt
9. Stage and commit your changes; try using both methods.

Readings

- Chapters 1 & 2 from Pro Git:

Chacon, S., & Straub, B. (2014). [Pro Git](https://git-scm.com/book/en/v2/) , Springer, London.

- Another great and easy to read overview is the [Atlassian tutorial](https://www.atlassian.com/git/tutorials/setting-up-a-repository)  [_on basic git commands.](https://www.atlassian.com/git/tutorials/setting-up-a-repository)

You can also access all the readings for this course from the 'Course Readings' menu item on the left of your screen.

Next Steps

At this point you understand a bit about what version control is, and the basics of how to use the Git version control system.

Next, we will look at a more advanced workflow and how to work with remote content.

This page includes content from [Pro Git](https://git-scm.com/book/en/v2/) , used by permission under Creative Commons      [_nc-sa/4.0\).](https://creativecommons.org/licenses/by-nc-sa/4.0/)

B1 Topic 2: Git Branching and Merging

i Multi-tasking in our repository

Large software projects often have multiple components being worked on at the same time that need to be managed separately from each other.

Consider what could happen if you and a colleague are working on two separate features of a software system; they might make some changes that break the code you're currently working on next time you update. Similarly, when a new change or feature is accepted and added to the main codebase, how do we keep our working copy up to date with the accepted changes without pulling in all of the other in-progress changes that others are working on?

Many version control systems like Git support **branching**; a way to split from the main codebase and make changes, implement new features, or just experiment without risking breaking main codebase.

You can easily switch between branches, and when ready, a branch can be merged back into the main line of development. In the diagram below, the main line of development (often called the "master") is in black with each dot representing a commit. Where the blue line diverges, we are said to have "branched". When development on the branch is complete, the work is merged back to the master branch as indicated by the rightmost blue arrow.



A branch and merge in a GitHub repository.

→ A Branching Git Workflow

Consider the following workflow that takes advantage of branches - we'll explain the highlighted words in more detail below:

1. Initialise a new local repository, or clone an existing remote one.
2. Create a **branch** for the feature/changes you want to work on and **checkout** that branch.

```
git branch new_feature  
git checkout new_feature
```

3. Develop your feature/changes in the new branch, staging and committing changes as needed.

4. Test your feature/changes.
5. Once everything is verified working, **merge** the branch into your main development branch.

```
git checkout main  
git merge new_feature
```

6. Resolve any **merge conflicts** that arise.
7. (optional) Delete the merged branch if no-longer needed.
8. (optional) At any stage you can checkout/switch-to a different branch. If you have uncommitted changes, you can **stash** your progress.

Following this process, the code in the main branch is always kept in a ready/working state, with changes only being added when they're complete. You can also work on multiple sets of changes at once, and more importantly, multiple users can work on separate branches simultaneously.

At any stage, when working on your branch, you can safely add and commit changes without disturbing the main branch while ensuring that you keep a versioned history of your work to accommodate situations where you need to roll back to a previous version.



Key Concepts

Branch

An independent line of development within the same repository.

Effectively similar to having separate copy of a repository where your changes can be developed, with its own commit history and staging area.

checkout

With `git checkout <branch>`, we can set our working copy to match the named branch. This is typical when we wish to start working on a branch and need to update the working copy before starting to modify or merge. During checkout, modified local files are not overwritten.

main/master

The default branch in a Git repository is usually named "main" or "master". This branch is created automatically when initialising the repository.

HEAD

The most recent commit in any branch is referred to as HEAD for that branch.

Merge

Combining two branches into one.

This is most commonly used to combine changes/bug fixes/new features from a development branch into the main branch.

Merge Conflict

An error that can occur when a merge is performed.

Git will attempt to automatically combine the changes from one branch to another, but if both branches have overlapping changes, it may not be able to do so. In this case you'll need to review the files that cannot be merged and determine the correct changes manually.

Stash

Temporarily storing uncommitted changes for restoration later.

Stashing is commonly used if you need to switch branches but have uncommitted changes. You should always commit your changes regularly to avoid needing to stash them.



Your Turn: Creating and Switching to a Branch

Using an existing repository from earlier work this week, create a **branch** for the feature/changes you want to work on and **checkout** that branch:

```
git branch new_feature  
git checkout new_feature
```

Any work that you do, files that you add, and commits that you make in this branch will be completely independent of your main branch. Make some changes to your working copy before adding and committing any changes. Recall that the last commit on your branch is called the HEAD of the branch.

If your work is committed, you can switch between branches at any time by checking out the branch you want to switch to. To see the list of all available branches, run:

```
git branch
```

To switch branches run:

```
git checkout branch_to_switch_to
```


A common use for this is switching back to the main/master branch.

If your work isn't committed, you may need to stash your changes (we'll revisit this in a moment).

Since creating a new branch and switching to it is such a common task, there's also a shortcut for this:

```
git checkout -b new_feature
```



Merging Branches

Once you've made your changes or created your new feature, you're probably ready to merge it back into your main development branch.

First, checkout your main/master branch:

```
git checkout main
```

This sets the working copy to the HEAD of the main branch but does not overwrite changes that you have made - these will be merged as follows:

```
git merge new_feature
```

Git will attempt to merge your changes from the new_feature branch to your main/master branch.

In most cases, Git is able to successfully make this merge, but sometimes errors can arise:

```
Auto-merging file3.txt
CONFLICT (content): Merge conflict in file3.txt
Automatic merge failed; fix conflicts and then commit the result.
```

In this case, the contents of `file3.txt` have changed in both the `main` and `new_feature` branches since that branch was checked out, so Git doesn't know which to use.

To resolve this, Git has updated the contents of the file to show both versions side-by-side:

```
<<<<<< HEAD
Contents of file in main
=====
Contents of file in new_feature branch
>>>>>> new_feature
```

Any places in a file where a conflict occurs are shown in the above format.

To resolve the conflict:

1. Update the affected files manually to the correct contents.
2. Stage the affected files.
3. Commit our fixes.

At this point the conflict is resolved and the merge is complete.

Once the branch is merged, you can switch back to the branch and continue to make changes, or you can delete the branch if it's no-longer needed.



Deleting a Branch

Deleting a branch is simple:

```
git branch -d new_feature
```

Things to note:

- You cannot delete a branch that you're currently working in; checkout your main/master branch before deleting another branch.
- If a branch has commits that haven't been merged to the main/master branch, Git will warn you.
In this case, you'll need to use `-D` instead of `-d` to forcibly delete the branch if you still want to.

Deleting branches can generally be avoided in most situations. Because git stores files so efficiently and transparently, having an extra branch or two is mostly of little consequence. In a long running development, branches may be pruned to make the codebase comprehensible and manageable.



Switching Branches under Development

When changing branches, we usually commit our changes before checking out the branch that we're switching to. Sometimes however, we may have temporary changes in place that we do not want to commit.

Changing branches at this point gives us an error:

```
git checkout main
error: Your local changes to the following files would be overwritten by checkout:
    file3.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

In this example, the changes we've made to file3.txt would be lost by checking out our main branch.

To get around this, we can temporarily shelve our changes with the **stash** command:

```
git stash
```

Saved working directory and index state WIP on new: 369c18b t1

Now we can checkout our other branches without issue.

When we're ready to return to our stashed work, we can pop the most recent stash to apply it to the current branch:

```
git checkout branch_with_stashed_changes
git stash pop
On branch new
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:   file3.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file2.py
        file4.py

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (940d882a3e7d9993afa712d77fd4e0ddd5153aec)
```

Be aware, stashes apply across all branches in your repository; this means:

1. Popping a stash restores the most recently made stash **from any branch, not just the current one**.
2. If you pop a stash on the wrong branch, you're going to have a bad time.
3. Having multiple stashes across multiple branches can get messy very quickly because of (1) & (2)

In this case you'll need to select the specific stash you want to restore. Refer to this week's readings for more detail on how to do this.

This is one of the reasons stashing changes is avoided unless absolutely necessary.



Your turn for some practice

If you haven't already, let's put our script files from Week 1 under version control.

After each of the following steps, run `git status` to see what has changed.

If you get stuck on a command, try `git --help` for more information.

1. Open your repository from the previous section
2. Create, add, and commit 2 new files, `file1.txt` and `file2.txt`
3. Create and checkout a new branch `new_1`.
4. Make and commit some changes to `file1.txt` in branch `new_1`.
5. Checkout your main/master branch.
6. Create and checkout a second new branch, `new_2`.
7. Make and commit some changes to `file2.txt` in branch `new_2`.
8. Checkout your main/master branch.

9. Make and commit some different changes to file2.txt in the main/master branch.
10. Merge branch `new_1`.
11. Merge branch `new_2`.
12. Resolve any merge conflicts.
13. Delete branch `new_2`.



Readings

- Chapter 3 from Pro Git: Chacon, S., & Straub, B. (2014). [Pro Git](https://git-scm.com/book/en/v2) [_\(https://git-scm.com/book/en/v2\)_](https://git-scm.com/book/en/v2), Springer, London.
- [This page](https://www.atlassian.com/git/tutorials/using-branches) [_\(https://www.atlassian.com/git/tutorials/using-branches\)_](https://www.atlassian.com/git/tutorials/using-branches) from Atlassian also has a great and easily readable overview of branching and merging.
- Git does not enforce a particular branching workflow upon teams - allowing adoption of an approach that suits any particular circumstance. [This page](https://www.atlassian.com/git/tutorials/comparing-workflows) [_\(https://www.atlassian.com/git/tutorials/comparing-workflows\)_](https://www.atlassian.com/git/tutorials/comparing-workflows), outlines the industry standard approaches to git workflows with the [feature branching workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) [_\(https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow\)_](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) being described in the course material above.

You can also access all the readings for this course from the 'Course Readings' menu item on the left of your screen.



Next Steps

At this point you got some practice using Git and understand a the basics of branching with Git.

Next, we will look at how to work with remote content.

This page includes content from [Pro Git](https://git-scm.com/book/en/v2) [_\(https://git-scm.com/book/en/v2\)_](https://git-scm.com/book/en/v2), used by permission under Creative Commons [_\(https://creativecommons.org/licenses/by-nc-sa/4.0\)_](https://creativecommons.org/licenses/by-nc-sa/4.0).



(h



(h



(h



(h



(h



(h



(h



(h



(h

B1 Topic 2: Git Remote Repositories & GitHub

Working with remote repositories

While Git provides a great way to manage code history, revisions, and features on our local systems, most developers store their code in a remote repository.

These remote repositories typically exist either on a central server such as the Git server that our university hosts, or in the cloud via services like GitHub or GitLab. This not only helps with data integrity and redundancy, but also provides a way for multiple developers to work on the same code base.

In turn, Git provides us with tools to link to a remote repository, **pull** updates from that repository, and **push** our changes to that repository (something you've already seen an example of).

A Git work-flow for a remote repository

Consider the following work-flow for interacting with a remote repository:

1. Create a new remote repository, or find an existing one.
2. **Clone** the remote repository
3. Develop your feature/changes, branching, staging and committing changes as needed.
4. **Pull** from the remote repository to merge any updates.
5. **Push** your changes to the remote repository

Following this process, the code in the main branch is always kept in a ready/working state, with changes only being added when they're complete. You can also work on multiple sets of changes at once, and more importantly, multiple users can work on separate branches simultaneously.

Key Concepts

Clone

Fetches a working copy of a remote repository.

remote

A remote repository connected to your working copy. A working copy can have multiple remotes, each with their own name.

origin

The name of the default remote for a cloned repository.

Push

Updates the remote with your working copy of a specific branch.

Pull

Fetches the latest changes from the remote, and merges them into your working copy. Merge conflicts can occur when pulling.



Setting up a remote repository

In this course we'll be using GitHub to store and submit our work.

Start by [signing up for a GitHub personal account](https://github.com/signup)  (<https://github.com/signup>) if you don't already have one.

- You'll also need to create a [Personal Access Token](https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token) 
(<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>)

Creating a new repository:

Repositories



New

Find a repository...

then


Create a new repository

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


[Import a repository.](#)

Owner *

Repository name *

 github-user ▾



 /

new-repository 

Great repository names are short and memorable. Need inspiration? How about **super-bassoon**?

Description (optional)

My new repository

- ☐  **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:

Skip this step if you're importing an existing repository.

☒ **Add a README file**
This is where you can write a long description for your project. [Learn more.](#)

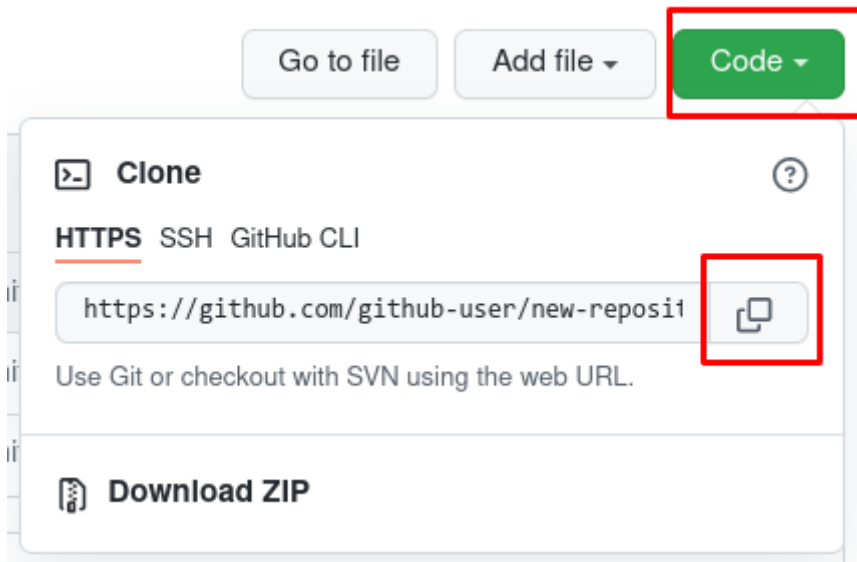
☐ **Add .gitignore**
Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**
A license tells others what they can and can't do with your code. [Learn more.](#)

This will set  **main** as the default branch. Change the default name in your [settings](#).

Create repository

Once the repository is created, you can copy its url then clone it.



```
git clone https://github.com/github-user/new-repository.git
```

If you are prompted for a username and password, use your Personal Access token as the password.

i Receiving updates from the remote

If you're working across multiple computers, or collaborating with others, you'll need to ensure that at least your main branch, if not all branches are kept up-to-date with the remote repository.

Any time you commence work on a branch that might have been updated then, it's a good idea to update that branch from the remote by pulling the latest changes:

```
git pull
```

If you are prompted for a username and password, use your Personal Access token as the password.

If there are changes on the remote, Git will attempt to merge those changes, creating a merge conflict if any issues arise.

Especially if you work across multiple devices, it's good to get into the habit of pulling each time you resume work.

i Sending changes to the remote

If you're working across multiple computers, or collaborating with others, you'll need to ensure that at least your main branch, if not all branches are kept up-to-date with the remote repository.

Any time you commence work on a branch that might have been updated then, it's a good idea to update that branch from the remote by pulling the latest changes:

```
git push
```

If you are prompted for a username and password, use your Personal Access token as the password.

If there are changes on the remote, Git will attempt to merge those changes, creating a merge conflict if any issues arise.

Especially if you work across multiple devices, it's good to get into the habit of pulling each time you resume work.



Your turn for some practice

Let's put our script files from Week 1 under version control.

After each of the following steps, run `git status` to see what has changed.

If you get stuck on a command, try `git --help` for more information.

1. Create a new repository on GitHub
2. Clone the repository
3. Checkout your main/master branch.
4. Create and checkout a second new branch, `new_2`.
5. Make and commit some changes to file2.txt in branch `new_2`.
6. Checkout your main/master branch.
7. Make and commit some different changes to file2.txt in the main/master branch.
8. Merge branch `new_1`.
9. Merge branch `new_2`.
10. Resolve any merge conflicts.
11. Delete branch `new_2`.



Readings

Chapter 2.5 from Pro Git:

Chacon, S., & Straub, B. (2014). [Pro Git](https://git-scm.com/book/en/v2)  (<https://git-scm.com/book/en/v2>), Springer, London.

GitHub Getting Started Guide:

<https://docs.github.com/en/get-started>  (<https://docs.github.com/en/get-started>)

B1 Topic 3: Debugging

So, what are “bugs”, anyway? A bug is simply unintended behaviour in a program. That is *not* to say the computer made a mistake. In fact, in almost all cases the computer is doing exactly what you told it, just the computer is not doing what *meant* to tell it. The severity of bugs can range from harmless annoyances to significant security vulnerabilities. Therefore, it is important for *all* programmers to be able to identify and resolve bugs.

The best time to learn how to debug code is at the beginning of your programming career. In this module you will build on the techniques you have already seen by introducing more powerful methods of debugging. The skills that you learn in this page will remain relevant regardless of which programming languages you use in the future and will scale with your abilities to resolve increasingly complex bugs.

Advanced Debugging – what are debuggers?

You have already been introduced to some forms of debugging. Firstly, you have seen how hand tracing can help you understand the state of a program line-by-line. Secondly, you have seen how print statements and logging can be used to gain information while a program is running to find where code is and the state of variables. Finally, you have learnt how to read the Stack Trace message that occur when the program crashes.

Now these are decent techniques for debugging, but they have their limits. For starters, printing is time consuming if you do not know where your program is having the issues as you have to insert prints throughout your code. You will also need to insert/delete them after each time that you want to use the technique to find a bug. Even if you use conditional statements to control when you print (so you can enable/disable all prints with a single variable), you increase the file size and slow down your program. These problems become exacerbated when your program is thousands or tens of thousands of lines of code line and spans over multiple files.

Logging and Stack Trace are not as messy as print debugging, but each have their limitations. If you do not have a hypothesis of where your bugs are and what is causing them, then logs can be cumbersome to work with. Finally, through what you know so far, Stack Trace will only appear when the program encounters an error and crashes. This of course means it is useless to find some bugs.

Enter debuggers.

A debugger is a powerful program that allows you to run your program in a controlled environment and gives you significant control over your program's flow. With a debugger, you can: run your program line-by-line, run your program until it hits a specific line or condition, check information

about the state of your program (such as the value of variables), etcetera. As you are using Python in this course, you will be learning about two Python debuggers: pdb and ipdb.



How do run the Python Debugger?

The Python Debugger (pdb) is the standard debugger for Python. It is easy to use and requires no additional set-up. There are two ways you can have your Python code run through this debugger.

1. Incorporate the debugger directly into your code through a module and call the debugger's `set_trace`. To do this, simply add the lines `import pdb` and `pdb.set_trace()` to your code. Any code above the line `pdb.set_trace()` will run as normal, but all code after it will run through the debugger.
2. You can call the debugger as a script through the terminal. But you can also use it inside any Shell scripts you have written. Instead of making changes inside your code, you will run your Python program using `python3 -m pdb [file name]`.



Using commands to aid your debugging

When you run your code through the debugger, the program will run as normal until it reaches its first **breakpoint**. Once it reaches this point the program execution will break. If you are running a debugger as a script, a breakpoint is effectively inserted on the first line. If you used the module import method, the a breakpoint will effectively be the line after the `set_trace` function is executed

Once the program hits this breakpoint, the debugger will give you show a Stack Trace and give you a prompt indicated by (Pdb). While the prompt is on screen, you will be able to run **commands** to control what the debugger does next. Below is a list of some of the most useful commands, and what they do. You can run a command by either using the bold letter below, or the entire word. These commands are all case sensitive.

Let us break them down into some helpful categories to explain them more clearly.

The first set of commands allow you manage the program's execution. They allow you to run your code line-by-line or stop at specific lines.

- **s**(tep) – Execute the current line. Stops at the first possible occasion (either the next line or a function call).
- **n**(ext) – Execute the current line. Stops only at the next line.
- **r**(eturn) – Execute the code until the current function returns.
- **c**(ontinue) – Execute the code until a breakpoint is hit.
- **b**(reak) – Set a breakpoint. A breakpoint is a line of code that has been marked by the debugger to stop at. Break is a powerful command, but for most cases in this course you will only have to worry about giving 1 argument to break, which will simply be the line number


you want the program to stop at. In more complicated programs, you may need to specify which file you are referring to when you specify a line number.

The next few commands allow you to gain information about where your code is and what it is doing.

- **l(ist)** – Displays 11 lines around the current line. If you enter this command again, without continuing code execution, it will continue the previous listing.
- **p** – Runs Python's `print`. You can use `p [variable name]` to easily check the current state of the variables in your program. Because this is running Python's print function, print is an invalid command, and if you want to write it out in full, you will have to format it as proper Python code. There's also **pp** to display using `pprint` instead.
- **w(here)** – Displays the Stack Trace. This is not normally necessary, as proceeding to a new line will automatically print the Stack Trace, but it is useful if you have run a few commands and want to remind yourself where the program currently is.

Finally, here are some other useful commands.

- **h(elp)** – Prints the list of commands. If you use this command followed by the name for a second command, it will give you more information on using that second command.
- **q(uit)** – Quit the debugger.

A full list of commands can be found on the pdb page of Python's documentation: [here](https://docs.python.org/3/library/pdb.html)  (<https://docs.python.org/3/library/pdb.html>).

Because Python is an interpreted language, you can also run any line of valid Python code from the debugger.



IPython Debugger – how and when to use it?

The IPython Debugger (ipdb) is an improved debugger for Python that uses the same interface as pdb. This means all the above commands for pdb also apply for ipdb. ipdb has some extra quality of life (such as colour) and extra features which make it a bit more useful to experienced programmers. In reality, you will not need the extra features of ipdb in this course, instead you likely will choose to use ipdb because it is usable inside Jupyter Notebook. To run ipdb inside Jupyter Notebook you will need to add the ipdb module by adding the line `from IPython.core.debugger import set_trace` the call the `set_trace()` function where you want the first breakpoint.

If you are using ipdb inside Jupyter Notebook, when you run commands like step you might notice the program jumping to some code you did not expect, to get around this you should avoid running ipdb on code outside of functions, or stop only at breakpoints by using the `continue` command.



Your turn for some practice: IPython Debugger

The best way to learn how to debug using a debugger is to actually use it! In this exercise you will resolve a bug that many new programmers run into.

```
In [1]: # here a function is defined, do not worry about  
# how this works if you have not learnt functions yet.  
def diff(num1, num2):  
  
    # [insert a breakpoint here]  
  
    # a difference should not be negative, so if the first  
# number is smaller than the second, swap the numbers around  
    if (num1 < num2):  
        num1 = num2  
        num2 = num1  
  
    # find the difference between the two numbers using subtraction  
    difference = num1 - num2  
    print(difference)  
  
    # run the diff function to find the difference between 10 and 15;  
    # we expect this to be 5  
    diff(10, 15)
```

1. Copy-paste the above code either into a new Python source file or your Jupyter Notebook. Run the file.
2. Since the output is not what we expect the difference to be, it's time to start debugging. Begin by including the pdb module to the code. Next, add a breakpoint to the location marked by the comment.
3. Run the program again. You should notice that the program has not completed running and the debugger has given you a prompt.
 - I. Get more information about where the program is in the code using by using the list. Try using it multiple times to get more information. Pay attention to the line numbers at this step.
 - II. Next try adding some break points. Start by placing a breakpoint at line of the if statement. Afterwards, use the continue, next, or step commands until you reach the breakpoint.
 - III. Try printing the contents of both num1 and num2 to predict if the condition will be true or false.
 - IV. Insert three new breakpoints. The first two should be the two lines that update num1 and num2. The third should be the line that the difference is calculated. You may want to use the list command to remind yourself of the line numbers or where you are in code.
 - V. As you hit each breakpoint, check the values of the variables. From this information, does the breakpoint stop before the line is run or after it?

Did you discover what the bug was?

In most languages, swapping a variable takes three steps, not two. Here the program copies the value of `num2` into `num1`, before moving the value of `num1` into `num2`. This means that both variables contain the value that `num2` started as. To do this in Python, we can use the power of tuples to ease the pain and write a successful, readable swap:

Python 3 REPL

```
>>> a1 = 10
>>> a2 = 15
>>> a1, a2 = a2, a1 # here we swap the variables using tuple syntax
>>> print (f"a1 = {a1}, a2 = {a2}")

a1 = 15, a2 = 10
```

Bugs involving swaps are actually a fairly common mistake that new programmers make – though usually not in this content.

→ Next Steps

Next, we you will learn what the exceptions you encounter during debugging are and how to handle them, as well as how static analysis can help you identify potential issues before runtime.



B1 Topic 3: Exceptions & Static Analysis

So far this week you have seen how to debug your code using a debugger. Even with these tools debugging can still be challenging. So, what other tools and techniques can be used to make debugging more manageable?

i Exceptions

In Python A, you learnt about assertions and how they can help a programmer debug code by aborting execution and reporting an error, specifically an `AssertionError`, if a specified condition was met. Assertions are not the only way a programmer can insert code to help them debug problems. You can also use **exceptions**.

When you execute a program, you expect that as long as a set of conditions are met, your program should behave in a predictable way and allow it to terminate normally. An exception (a runtime error) is **raised** (or **thrown**) if the program cannot terminate normally. You likely have already noticed these as you encountered bugs in your code. For example, when your program crashes, you might see a message such as `NameError: name 'x' is not defined`. In this case, you can deduce that the program was working under the assumption that you would pass it a valid variable name. Since the program was not able to terminate normally, an exception (in this case `NameError`) was raised.

i Working with exceptions in Python: The *try* statement

Normally, when you encounter an exception your program will crash, but you can use special blocks of code to *try* code. In Python you can achieve this using a **try statement**.

A try statement is comprised of at least two components. Firstly, the **try** block. When the program encounters a try, it will *attempt* to run the code inside the block. If the program encounters an exception, the program will stop running code inside the try block and will begin executing the **except** block. If the program did not encounter an exception, the program will skip the except block after the try block has completed.

Here is a simple Python program that takes in a number and prints its inverse as a float. If you have tried to write a program that does calculations on user input, you may have already noticed that if the user enters something unexpected, the program might crash.

```
In [1]: usr_input = input("Enter an integer: ")
        inverse = 1/int(usr_input)
        print(inverse)
```

In the above code, the two most likely exceptions are `ValueError` which occurs when the user enters something that cannot be cast to an integer (such as "Hello world") and `ZeroDivisionError`, which occurs when the user enters a 0 as their input. If you run this block and enter 0, the program crashes and you get an output like this.

Out[1]:

```
Enter an integer: 0
-----
ZeroDivisionError                                Traceback (most recent call last)
/tmp/ipykernel_2553/3442874333.py in
      1 usr_input = input("Enter an integer: ")
----> 2 inverse = 1/int(usr_input)
      3 print(inverse)

ZeroDivisionError: division by zero
```

As you saw, the program crashed and reported it encountered the `ZeroDivisionError` exception. You might decide to introduce a try catch to this program, to allow the program to continue running even if the exception occurs.

In [2]:

```
try:
    usr_input = input("Enter an integer: ")
    inverse = 1/int(usr_input)
    print(inverse)
except:
    print("An error occurred.")
```

Now if you run this code using the same input, the output will be different. Instead of showing the Traceback and exception type, the output is now the result of the except block's print. This can be useful to help you debug as it allows you to customise error messages to have more or less information as required.

Out[2]:

```
Enter an integer: 0
An error occurred.
```

You are not limited to one except block, you can have as many as you want. This is useful for when you want to run different blocks of code for different errors. In this case, the following code might be more helpful to the programmer.

In [3]:

```
try:
    usr_input = input("Enter a non-zero number: ")
    inverse = 1/int(usr_input)
    print(inverse)
except ZeroDivisionError:
    print("The try block attempted to divide by 0")
except ValueError:
    print("The try block was unable to case user input to an int")
except:
    print("Some other exception occurred")
```

There are two other types of blocks that can be used with a try statement. The **else** block will run if there are no errors, that is, it runs if the except block does not run. The **finally** block executes regardless of the result of the try.

The following block combines all of the four blocks together.

In [4]:

```
try:
    usr_input = input("Enter an integer: ")
    inverse = 1/int(usr_input)
    print(inverse)
except:
    print("An error occurred.")
else:
    print("No error occurred.")
finally:
    print("block completed.")
```

The else and finally blocks become more useful as you build more complex programs. Another useful feature of try statements is the ability to nest blocks inside each other. Which can be particularly useful when working with networks or files where many things can go wrong.

Raising your own exceptions

When you are developing more complex code, you might want to define your own exceptions so that any programmer using your functions or methods can get more relevant information. In Python you can do this using the raise keyword. Consider the following example where the program will terminate while raising an error if a is not between 1 and 10.

In [5]:

```
a = 0

if a < 1 or a > 10:
    raise Exception("variable a must be between 1 and 10")
```

Out[5]:

```
-----
Exception                                 Traceback (most recent call last)
/tmp/ipykernel_2553/1300123875.py in
      2
      3 if a < 1 or a > 10:
----> 4     raise Exception("variable a must be between 1 and 10")

Exception: variable a must be between 1 and 10
```

This works like the AssertErrors you have already been using. You can also replace the `Exception` with the specific exception types, such as `TypeError` to further customise the exception.

Static Analysis

There is an old proverb that says “prevention is better than cure”. In the context of programming, this means putting in effort to prevent bugs from arising is a better approach than trying to fix them. To achieve this, programmers need to learn to write higher quality code with fewer bugs. That is easier said than done of course, but there are tools and techniques that allow programmers to do this.

Analysing code is a very useful way to identify issues in your code. This can be done dynamically or statically. Static analysis means the code is analysed without being run, while dynamic analysis involves analysing code while the code is running. Static analysis involves looking over source code to identify problems, this can be done manually by programmers or automatically using specific programs.

Identifying issues in your code before running it is very important as it will save you time trying to debug the program and can help you optimise your code. Performing static analysis is particularly important in Python as it is an interpreted language. In Python, if you attempt to use an undeclared variable, the program will crash, while in a language that requires a compiler (like C++) the compiler will report this issue before runtime.



Your turn for some practise: Static analysis

Pyflakes is an easy-to-use tool that checks for errors in code. It works by parsing over the provided source file and reporting errors it encounters. In this exercise you are going to practice performing static analysis using Pyflakes to analyse the following code.

```
In [6]: import random
import pdb

def sum_two_numbers(x, y):
    sum = x + y
    print("The sum of", x, "and", y, "is", sum)
    return sum

a = random.randint(0, 10)
b = random.randint(0, 10)
sum_two_numbers = a + d
```

1. If you do not have Pyflakes on your machine, install it using `pip install --upgrade pyflakes` in the terminal. You can check if you have it installed by running `pyflakes --version`.
2. Create a new file and copy-paste the above code into it.
3. Run Pyflakes using `pyflakes [file name]` in the terminal.
4. Fix the code such that running Pyflakes does not print any error messages, while maintaining the intended purpose.



Linting

Linting is a subset of static analysis that specifically looks for stylistic issues and mistakes in code. Some can also flag possible memory leaks and insecure code design. Most good text editors and integrated development environments (IDEs) have some basic linting functionality, either built in or available by downloading an extension, where they highlight easily identifiable errors and warnings. A tool used specifically for linting is called a **linter**.

Python has comprehensive design documents called Python Enhancement Proposals or **PEPs** for short. PEP8 describes the set of style rules that Python programmers should practice. PEP8 was written under the idea that code will be read more times than it is written, and this idea drives some of the design choices. Stylistic linters such as **Pylint** and **Pycodestyle** (formally named pep8) can both be used to ensure good programming practices are being followed.

We encourage you to use Pylint as your linter for this course, as it is designed to be a more general-purpose linter, whereas Pycodestyle is designed specifically enforce PEP8. You may choose to use both if you would prefer.

When you run a linter, it checks your code and outputs a report based on its findings. Pylint will output a line for each problem it encounters. Below is an example of a line that might be reported by Pylint.

```
lintingExample.py:3:0: C0325: Unnecessary parens after 'while' keyword (superfluous-parens)
```

The first part of this line describes where the problem was observed. Here you can see the problem exists in a file called lintingExample.py and is located at the start of line 3. It is also reporting a code "C0325", along with a brief description of the problem.

The code and description are a pair, with the description acting as a brief overview of the error and the code being usable to check the documentation for more information. The code will begin with one of 3 letters.

- **C** – your line does not meet the standard coding conventions
- **W** – your line has a potential problem that requires your attention
- **E** – your line has a severe problem that requires your attention.

Pylint also rates your code out of 10.




Your turn for some practise: Linting Code

The best way to understand how to use a linter and why it's useful is to actually use one. In this exercise you will practice linting a very small block of code using Pylint.

```
In [7]: def count_to_ten():
        count = 0
        while(count<10):
            count += 1
            prin(count)

        count_to_ten()
```

1. Check that you have Pylint already installed on your machine by running `pylint --version` in the terminal. If you do not have it, follow the appropriate instructions for your machine at <https://pylint.org/#install>  (<https://pylint.org/#install>).
2. Create a new file and copy-paste the above code into it.
3. Run Pylint using `pylint [file name]` in the terminal.

B1 Topic 3: Testing Basics

Testing is one of the most important parts of programming, regardless of language, as it gives you the confidence that your code works as intended. Insufficient testing leads to buggy code and security vulnerabilities which in turn reflects badly on both you as a programmer and damages the reputation of company or project you represent.

Testing is a massive topic, so this week we are only going to cover the basics of testing. As you develop your abilities and build more advanced programs, your testing skills will need to develop with you.

i Testing using assertions

When you run your code one of two things can happen, it either works as intended or does not. As such, asserts become very useful when testing.

Consider this snippet of code. This code defines a function called `sum`, that takes two inputs `num1` and `num2` and returns the sum of those two inputs as its output.

If you are not yet familiar with functions in Python, it is not a problem as you do not need to understand how they work here. For now all you need to know about functions is that we can call functions we build ourselves, just like how we can call built-in functions like `print()` or `abs()`.

```
In [1]: def sum(num1, num2):  
        sum = num1 + num2  
        return sum  
  
x = sum(num1, num2)  
print('The sum of your two numbers is', x)
```

We expect the output of this function to be consistent with our knowledge of arithmetic. For example, the sum of 5 and 10 is 15. In order to test that the sum function is working correctly, we can use an assert like `assert sum(5, 10) == 15`.

Also, based on the variable names and the print message, this function should only sum numbers together. What happens when `num1` and `num2` are instead strings? Currently the code has no check for this, so it would be a good idea to have the code extended to check if `num1` and `num2` are numbers (int, floats, complex).

```
In [2]: def sum(num1, num2):  
        assert (type(num1) == int or type(num1) == float or type(num1) == complex), \  
            "num1 was not a number (int, float or complex)"  
        assert (type(num2) == int or type(num2) == float or type(num2) == complex), \  
            "num2 was not a number (int, float or complex)"  
        sum = num1 + num2  
        return sum
```

```
x = sum(12, 24)
print('The sum of your two numbers is', x)
```

Once the code has been modified in this way we can test that exceptions, in this case as `AssertionErrors`, are generated instead when expected the incorrect types are given.

Unit testing

An application, such as Google Chrome or Microsoft Word, is massive program made up of numerous working parts. It is extremely challenging to test such a large application as a whole. One way programmers address this is using **unit testing**. Unit testing is just as applicable to smaller programs.

With unit testing, we test each individual component, called a **unit**, independently of the other units. What defines a unit depends on context of what you are working on. Typically a unit is an individual algorithm, function, or method, but it could be any block of code. If the unit behaves as expected in a reliable manner, the unit testing is complete and we consider it ready to be connected to other units to form the program.

Black box Testing

In computer science, a **block box** a system that is only perceived through its input and output. That is you have no knowledge of how it works internally.

When you write your own code, you know exactly how the code works. In contrast, when you use code you have not written, such as in-built functions, you likely do not know how they work. For example, the built-in `abs()` function is used to find the absolute value of a number, but you do not know, nor do you need to know, how that function is implemented.

Black-box testing builds on the concept of the black box. In this type of testing, you test an application as if you did not know how it was written. This type of testing can be applied on all levels of testing, including unit testing.

To write black-box tests, the tester needs to consider only their input and the output of the black box. The tester might also have access to documentation to know how the black box is supposed to be used. The tester can try both valid and invalid inputs to test if the behaviour is what it is supposed to be.

Here are some examples of black-box testing against the aforementioned `abs()` function. To test this function we might want to have a quick look its documentation to check how this function is supposed to work. Here it makes sense to use `assert` for the first three tests, as we expect these to be true. The latter, we do not know if it will work or not, so you might consider using a `try except`.

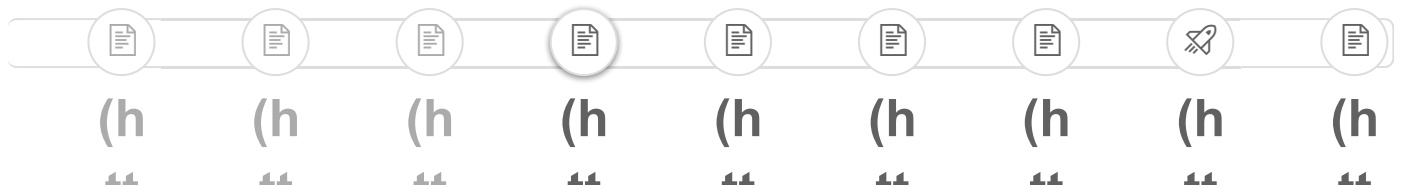
In [3]:

```
assert abs(16) == 16
assert abs(-4) == 4
assert abs(3j) == 3
try:
    abs('hello')
    print('test: abs(\'hello\') passed')
except:
    print('test: abs(\'hello\') failed')
```

→ Next Steps

So far, you have covered how assertions and exceptions can be useful for testing as well as been introduced to some testing styles.

On the next page, you will learn how to make good unit tests and be introduced to the idea of test driven development.



B1 Topic 3: Testing Methodology

i Writing effective test cases

On the last page you were introduced to the idea of unit testing, but how do you perform unit testing and how do you perform it well?

As for building unit tests, all the testing examples on the last page were actually individual unit tests! The challenge in building unit tests is almost always in the designing of the test, not writing the code for the test.

What should you be testing?

It is often infeasible to test a unit under every possible case, so when we need to be smart when performing unit testing. In most cases a unit's behaviour (or output) will differ depending on its input this is the main driver of unit testing.

There are three broad categories to test when unit testing.

1. **Normal usage:** These tests are designed to test if the program performs as expected under normal usage.
2. **Edge cases:** These are the extremes or boundaries that your code should operate under. They may also be special values that you need to manually handle. If your code works at all edges, we can assume it works on all the cases between them. With practice you will become more comfortable with identifying edge cases.
 - When two or more edge cases are tested in a single test, it is called a **corner case**.
3. **Unintended usage:** These tests are designed to try and break your code. For example, if you had some code or a function that attempted to sum two numbers together, does it behave in the way you intend if you pass it two strings?

If you can test every possibility, do so!

An example of deciding on unit tests.

Let's look at an example of how to decide what to test by looking at the following code.

```
In [1]: usr_input = int(input('Enter the patient\'s age in years: '))
        if usr_input <= 1:
            print('You cannot enter a negative age!')
        age = usr_input
        print('Success! The patient is', age, 'years old.')
```

Firstly, this is a small amount of code at only 5 lines, so this is definitely small enough to be a single unit.

In this code, the programmer wants to ensure the user inputs a non-negative number. So, they have an if condition to check a user's input before updating the patient's age. Unfortunately the programmer has made some mistakes and the code does not behave in the way they intended it to. What kinds of tests would be useful for this code?

- Normal usage: Since this code should set the age variable if the user inputs a positive number, the tester should make sure these work.
 - **1**: This is the number written in the code.
 - **0** and **2**: These numbers are adjacent to one on the number line. Zero is also usually a good test case regardless.
 - **54**: This is just some valid number.
- Edge cases:
 - **1000000**: An extremely large positive number. If the code works for the normal behaviour tests and works for this number, it will very likely work for all positive numbers.
- Unintended usage:
 - **-1** and **-1000000**: Negative ages do not make sense. While the code has an if condition that intends to catch negative inputs, the programmer must check that it works. If it works for both negative one and negative one-million, then it probably works for any negative number.
 - Numbers with decimal points: There is a function to cast the user input to an integer, do we observe the expected behaviour if they attempt to enter a float?
 - A string: Same idea as attempting to use a float.
 - Nothing: What happens if we just hit enter without adding a number.

If they do this testing, the programmer will find the following problems:

- A. The program updates the patient's age, even if a negative number is entered.
- B. **0** and **1** are interpreted as a negative numbers and the program will they will declare them as invalid.

While the mistakes above might have seemed obvious to you when looking at the code, such mistakes are made by veteran programmers all the time. Reading your own code is a very difficult experience to reading someone else's. When you read your own code it is easy to read the code as what you expect it to be, not what is actually written there. When looking at your own code it can be helpful to say aloud what the code is doing, in fact rubber duck debugging is a fairly popular method of debugging for many programmers, where you explain the code aloud to yourself, a friend, or even an inanimate object.



Test Driven Development

So far, we've looked at testing as a way of validating your existing code. You develop code first, then verify its correctness with testing.

We can however also take the reverse approach, where we begin with coming up with the tests

Test driven development (TDD) is a philosophy of testing. It is the idea that the development of your program should be directly influenced by your testing. This involves you building all the tests that your eventual code will need to pass, before any of your code's functionality is implemented. This is a slightly unintuitive process, but it goes as follows:

1. Plan your code as a black box, considering only the inputs and the expected behaviour. At this stage, the only code you can build is an interface. For example, if you were writing a function, you can have the function defined but it must return immediately without doing anything inside. A similar approach could be taken with variables; define the initial values of the variables used to input and output from your code.
2. Think about what your code is supposed to do. What **should** the outputs be for different inputs?
Be sure to consider both normal behaviour, as well as edge cases, and invalid inputs.
3. Write your tests, these should test the inputs from the previous step.
4. Run your tests. Since you haven't written the actual code yet almost, if not all, of these tests should fail.
5. Then 'fix' the code by building the functionality into your code until all the tests pass.
6. Keep testing. As you further develop your code and make changes, your test cases should continue to pass.

If a test case fails, then either your changes have broken something, or you need to update your testing to reflect a change in the program's expected behaviour.

Although we won't be focussing further on TDD at this stage, it's a great way to practice developing tests and also helps ensure robust code. We will revisit the philosophy and practice of Test Driven Development in [Section B2, Topic 3](#)

(<https://myuni.adelaide.edu.au/courses/86136/pages/c-%7C-week-3-summary>).



Next Steps

Aaand that's a wrap! By now you should have a basic understanding of Debugging and Testing principles.

The module test is coming up next week. Make sure you can do the exercises and practice material for this week, and take a look at the sample module test (these will be available shortly).

