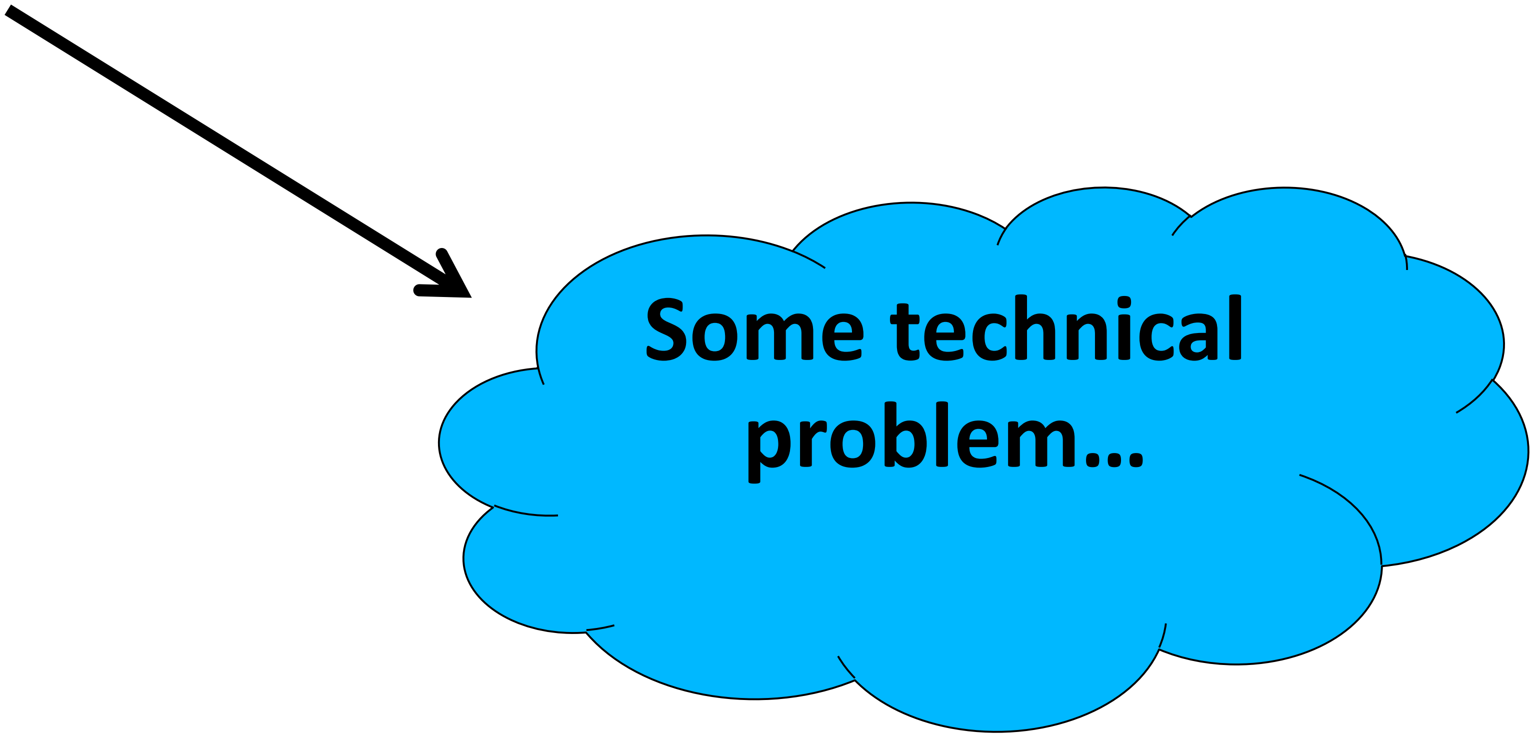
The Unix Shell

Advanced Shell Tricks

"How should I do this?"



In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters

```
$ wc -l *.pdb | sort | head -1
```

In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files

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

In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
```

In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

Very powerful when used together

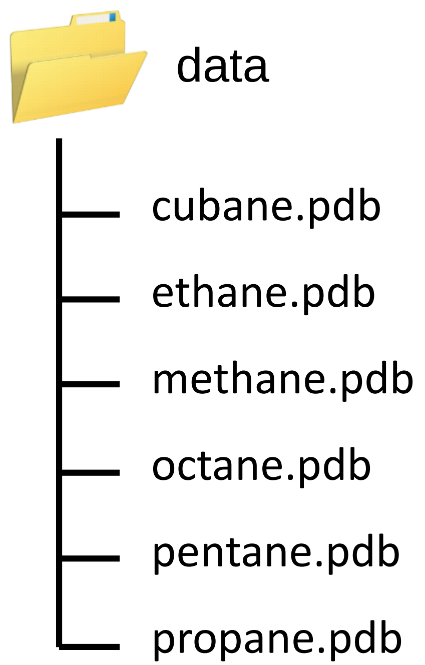
In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

Very powerful when used together

But there are other useful things we can do with these – let's take a look…

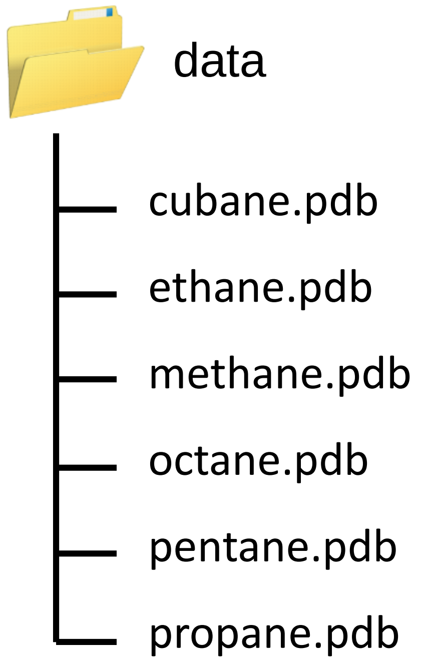
First, let's revisit redirection…



First, let's revisit redirection...

```
$ ls *.pdb > files
```

list all pdb files
redirect to a file

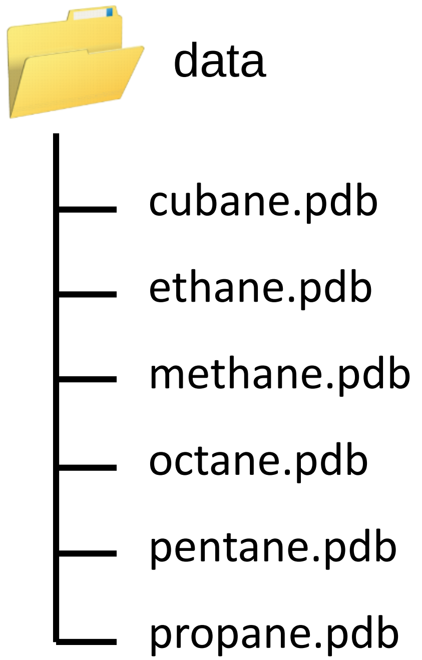


First, let's revisit redirection...

```
$ ls *.pdb > files
```

list all pdb files redirect to a file

The 'redirection' operator

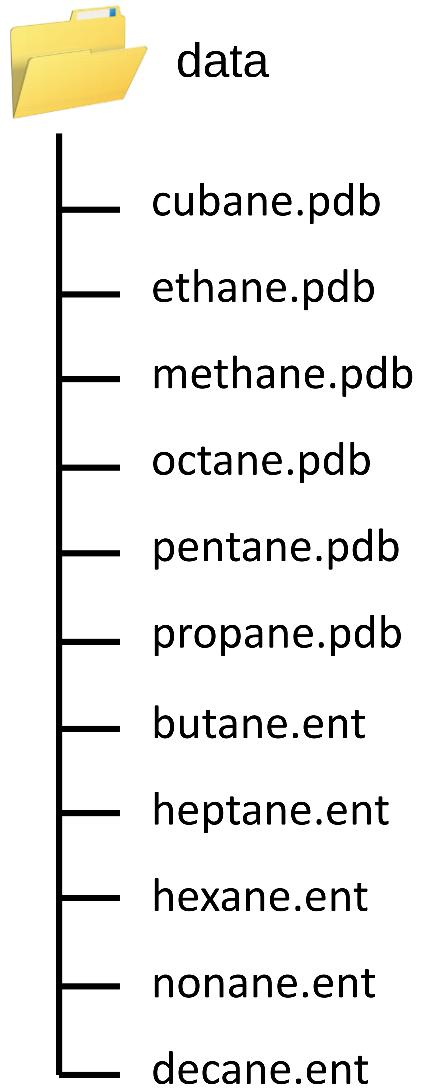


First, let's revisit redirection…

```
$ ls *.pdb > files
```

list all pdb files
redirect to a file

But what about adding this together with other results generated later?



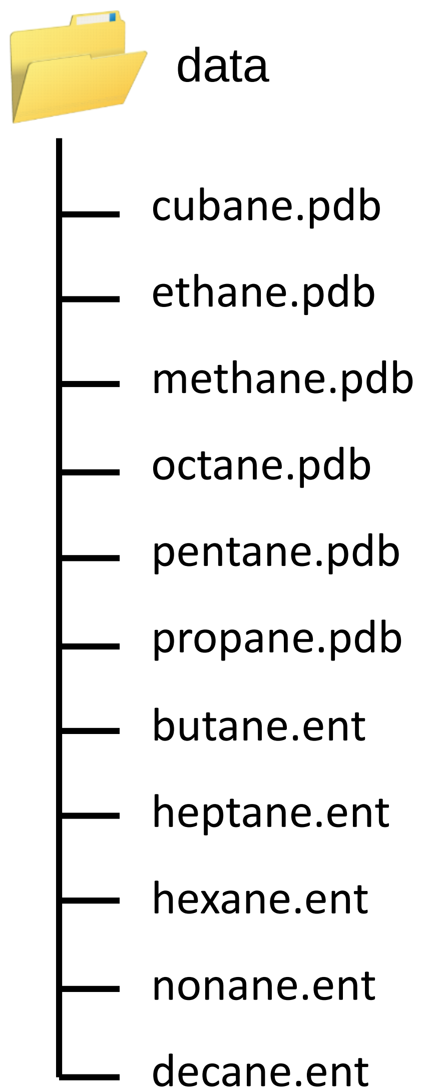
First, let's revisit redirection...

```
$ ls *.pdb > files
```

← list all pdb files
redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
```



First, let's revisit redirection...

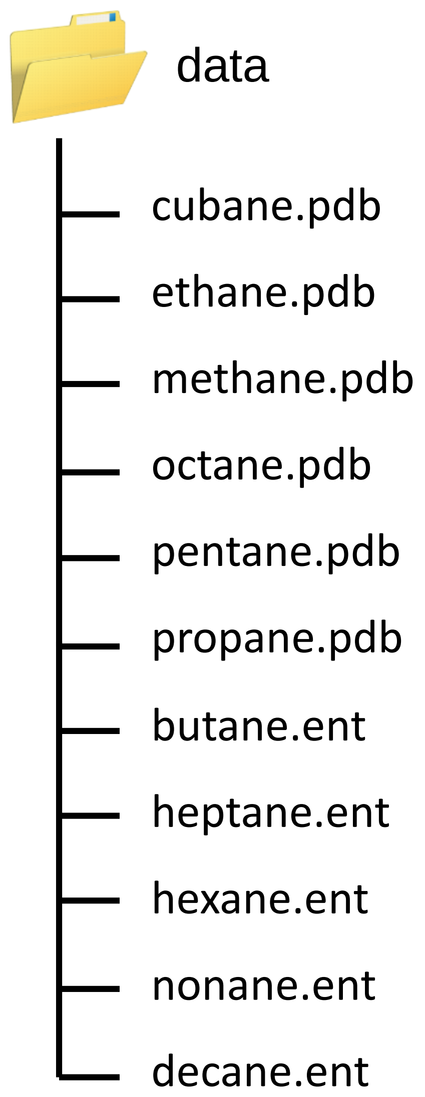
```
$ ls *.pdb > files
```

list all pdf files
redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
```

We just want the ent files



First, let's revisit redirection...

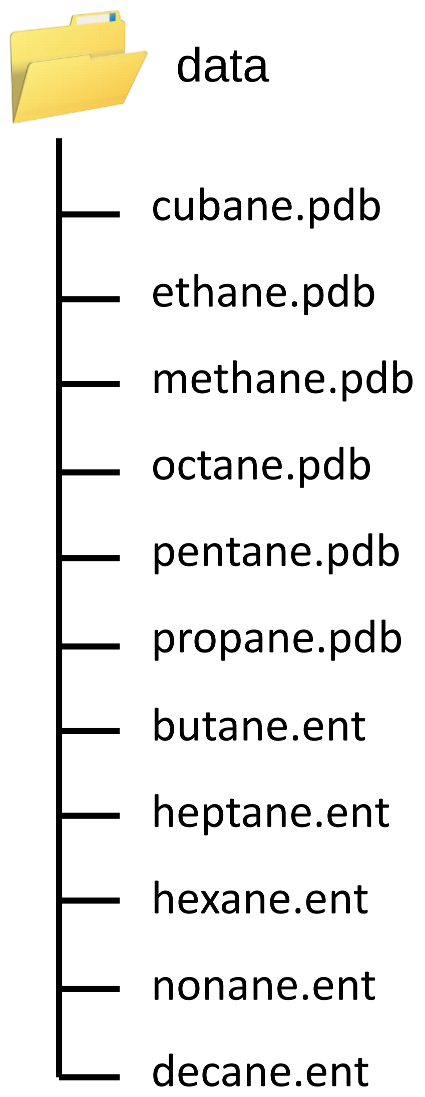
```
$ ls *.pdb > files
```

list all pdb files
redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

append files
into a single
new file



First, let's revisit redirection...

```
$ ls *.pdb > files
```

list all pdb files redirect to a file

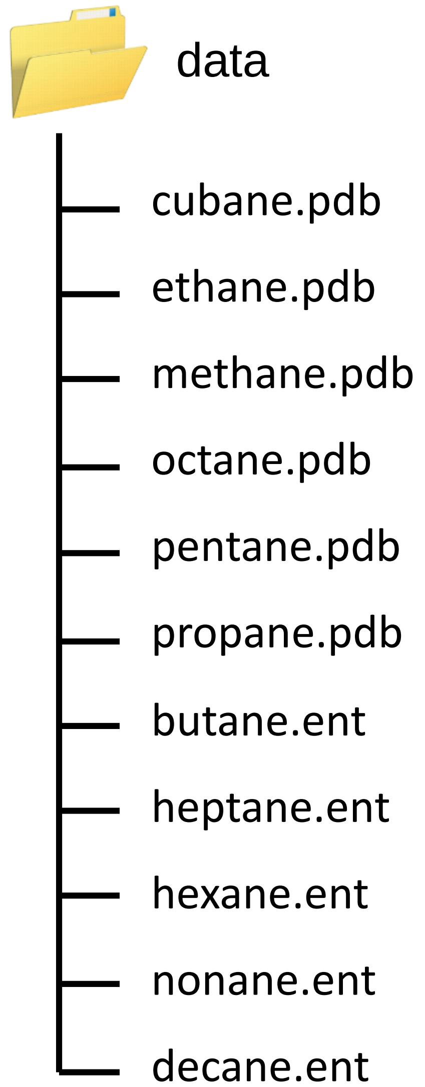
But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

Instead, we can do...

```
$ ls *.ent >> files
```

append files into a single new file



First, let's revisit redirection…

```
$ ls *.pdb > files
```

list all pdb files redirect to a file

But what about adding this together with other results generated later?

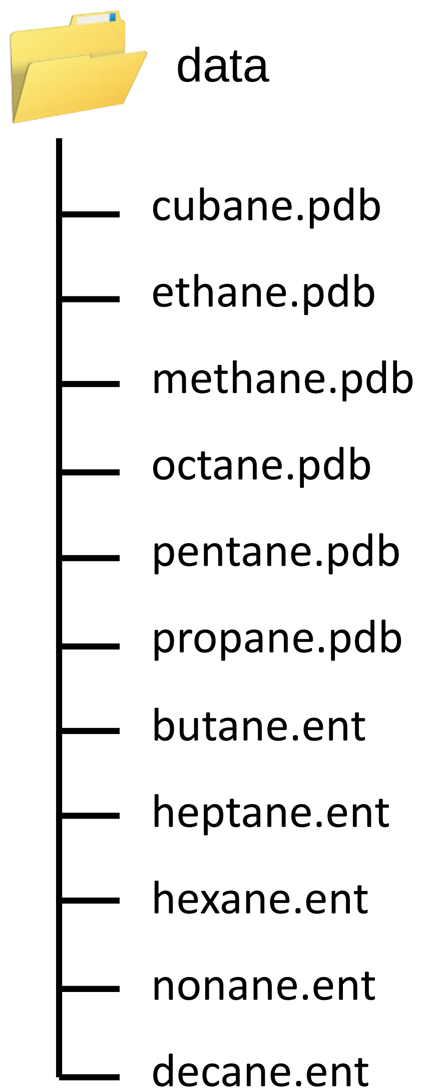
```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

append files into a single new file

Instead, we can do…

```
$ ls *.ent >> files
```

Note the double >'s – the append' operator

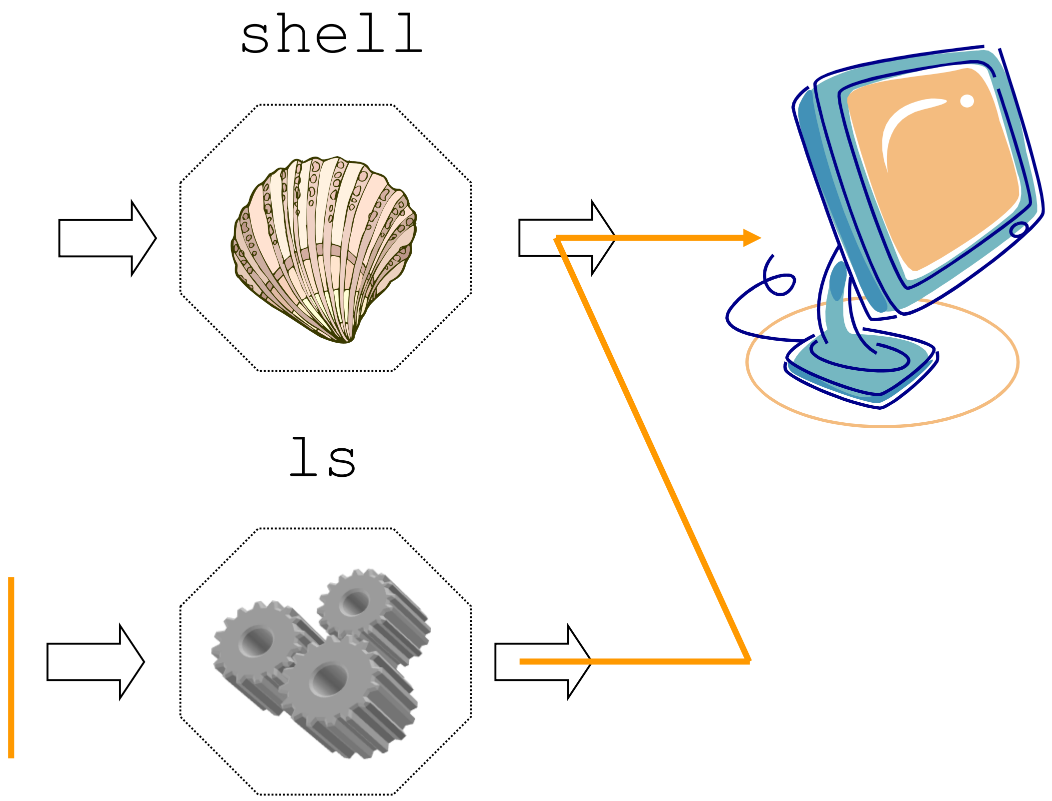


We know that…

Normally, standard output is directed to a display:

We know that…

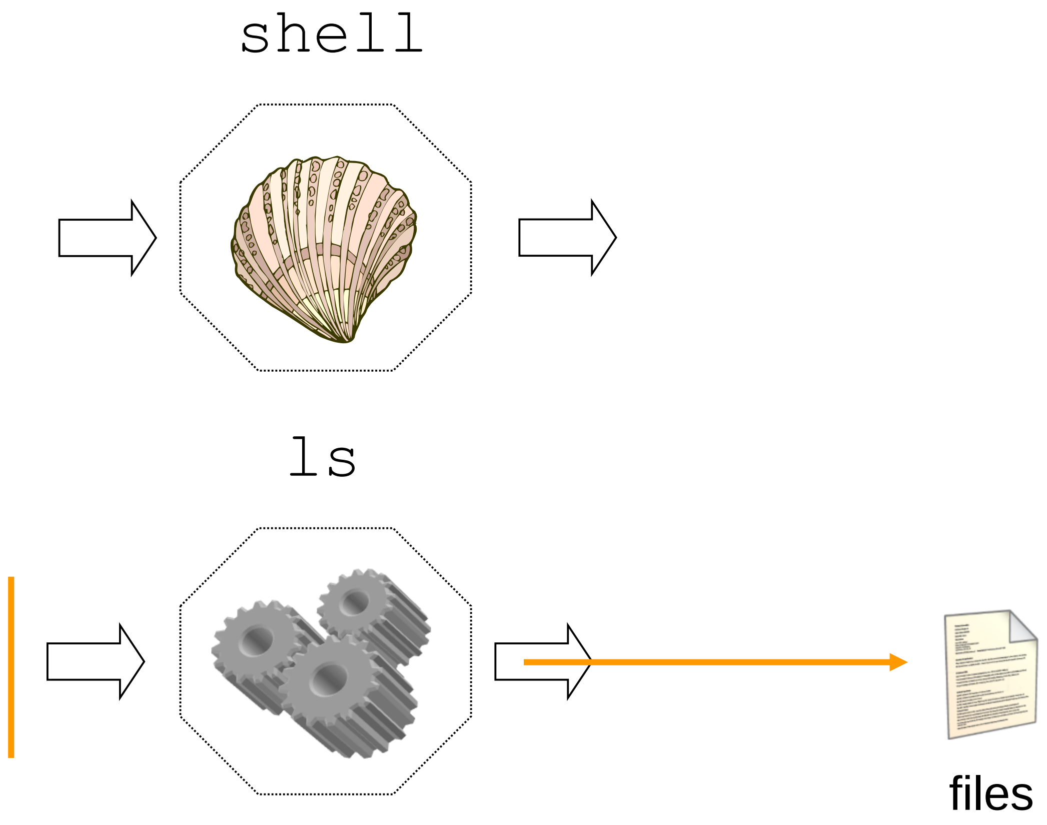
Normally, standard output is directed to a display:



We know that...

Normally, standard output is directed to a display:

But we have redirected it to a file instead:



But what happens with error messages?

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or
directory
```

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or
directory
```

No files are listed in *files*, as you might expect.

But what happens with error messages?

For example...

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or
directory
```

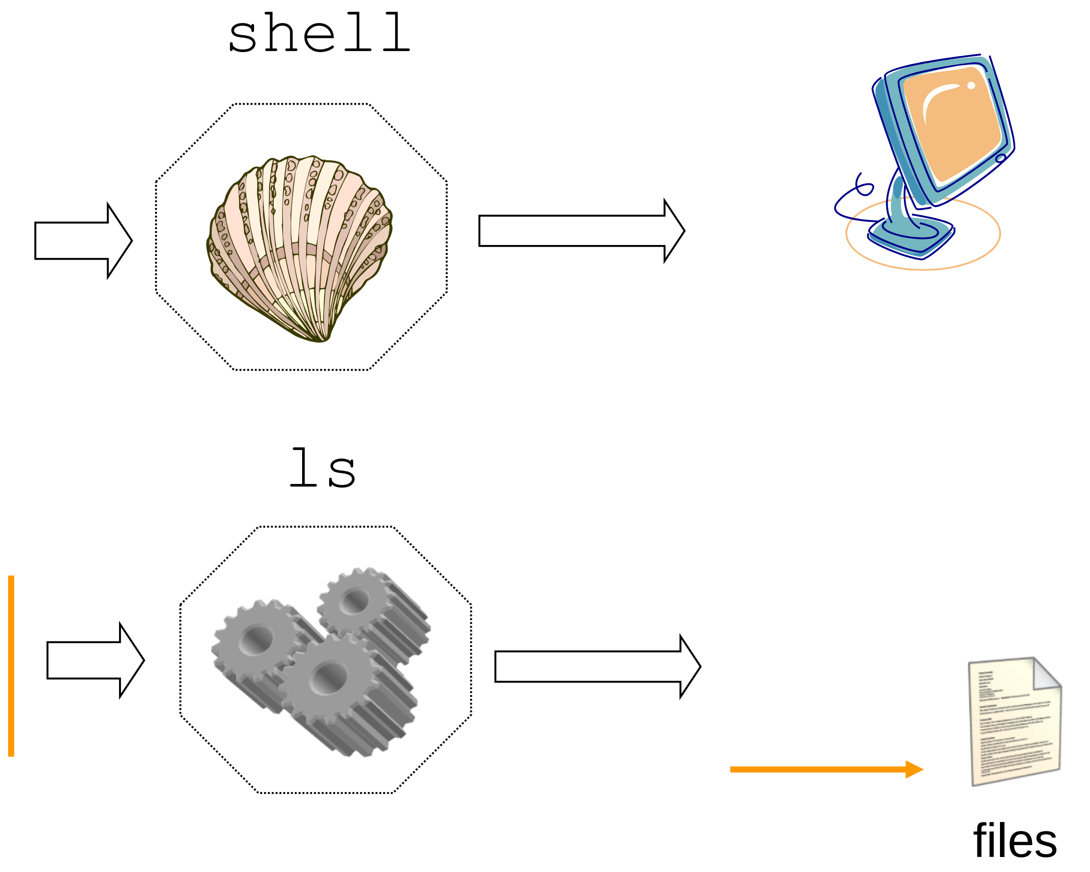
No files are listed in *files*, as you might expect.

But why isn't the error message in *files*?

This is because error messages are sent to the *standard error* (stderr), separate to stdout

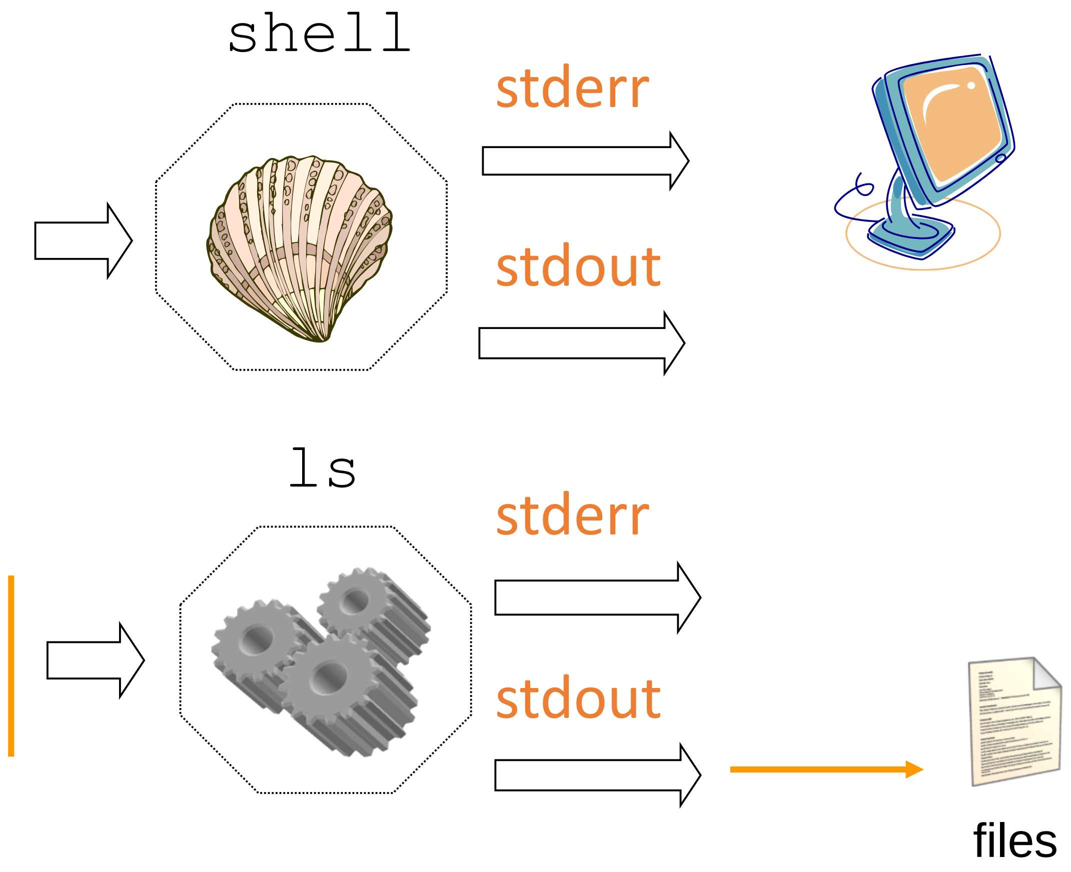
This is because error messages are sent to the *standard error* (stderr), separate to stdout

So what was happening with the previous example?



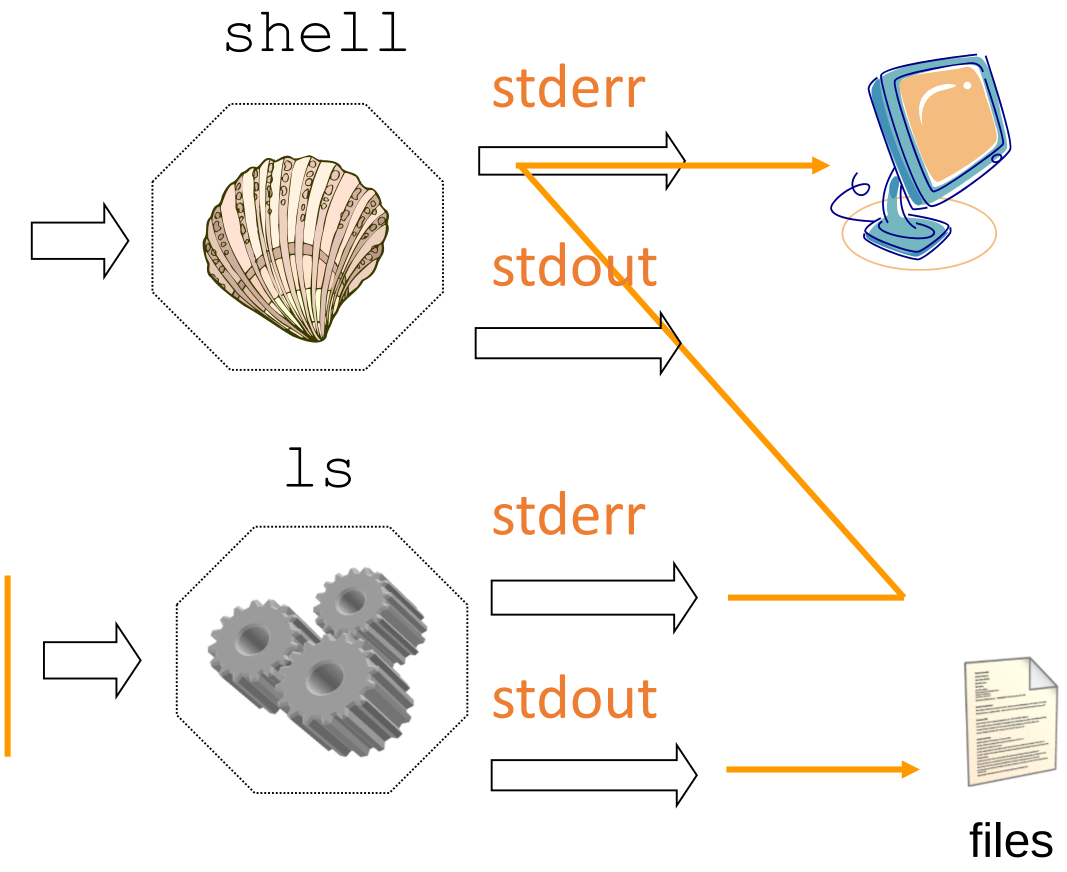
This is because error messages are sent to the *standard error* (stderr), separate to stdout

So what was happening with the previous example?



This is because error messages are sent to the *standard error* (stderr), separate to stdout

So what was happening with the previous example?



We can capture standard error as well as standard output

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Redirect as before, but with a slightly different operator

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2> error-log
```

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2> error-log
```

We can use both stdout and stderr redirection – at the same time

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2> error-log
```

Which would give us contents of */usr* in *files* as well.

So why a '2' before the '>' ?

So why a '2' before the '>' ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2> error-log
```

So why a '2' before the '>' ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2> error-log
```

Refers to stdout

Refers to stderr

So why a '2' before the '>' ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2> error-log
```

To just redirect both to the same file we can also do:

```
$ ls /usr /some/nonexistent/path &> everything
```

With '&' denoting both stdout and stderr

So why a '2' before the '>' ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2> error-log
```

To just redirect both to the same file we can also do:

```
$ ls /usr /some/nonexistent/path &> everything
```

With '&' denoting both stdout and stderr

We can also use append for each of these too:

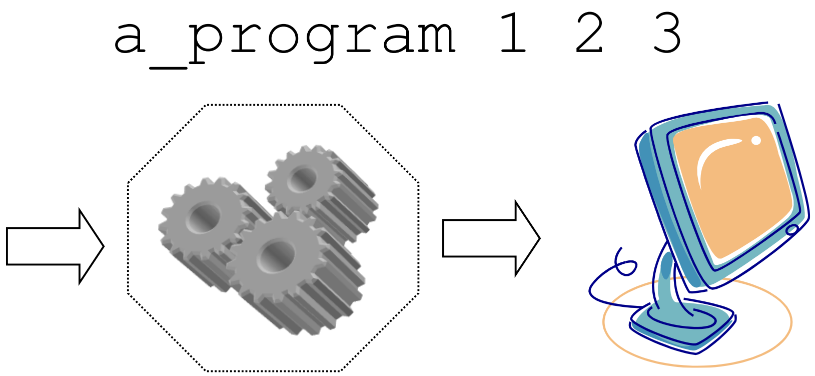
```
$ ls /usr /some/nonexistent/path 1>> files 2>> error-log
```

`>`	`1>`	Redirect stdout to a file
	`2>`	Redirect stderr to a file
	`&>`	Redirect both stdout and stderr to the same file

`>`	`1>`	Redirect stdout to a file
	`2>`	Redirect stderr to a file
	`&>`	Redirect both stdout and stderr to the same file
`>>`	`1>>`	Redirect and append stdout to a file
	`2>>`	Redirect and append stderr to a file
	`&>>`	Redirect and append both stdout and stderr to a file

We've seen how pipes and filters work with using a single program on some input data…

We've seen how pipes and filters work with using a single program on some input data...

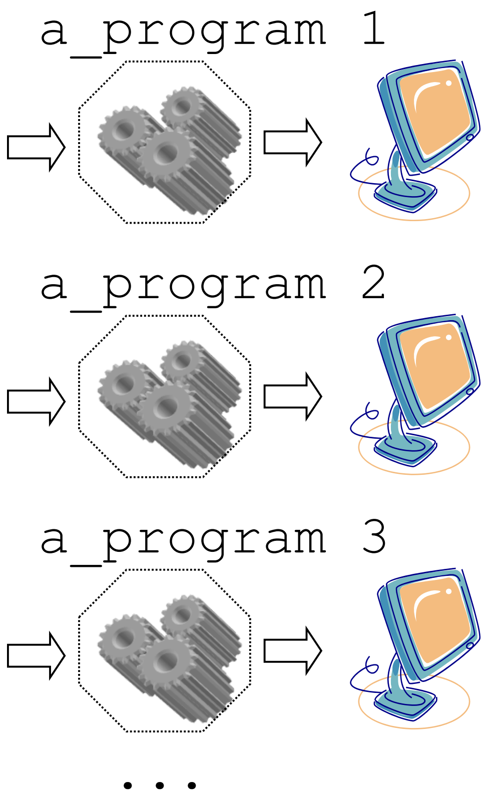


We've seen how pipes and filters work with using a single program on some input data…

But what about running the same program *separately*, for each input?

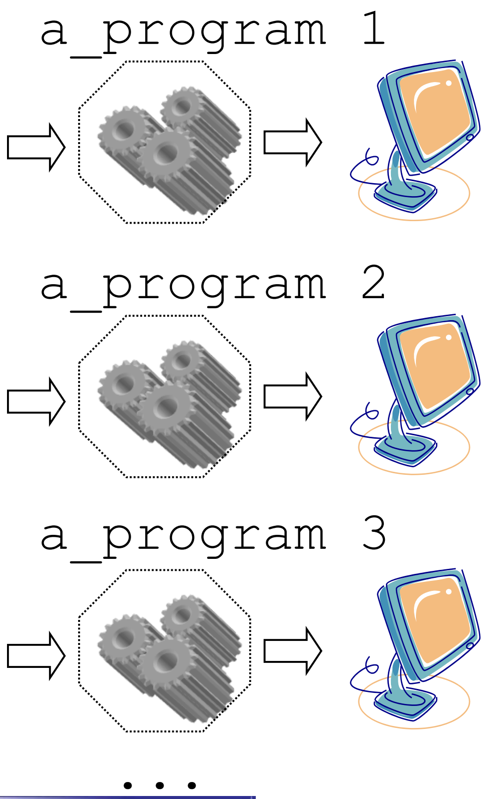
We've seen how pipes and filters work with using a single program on some input data…

But what about running the same program *separately*, for each input?



We've seen how pipes and filters work with using a single program on some input data...

But what about running the same program *separately*, for each input?

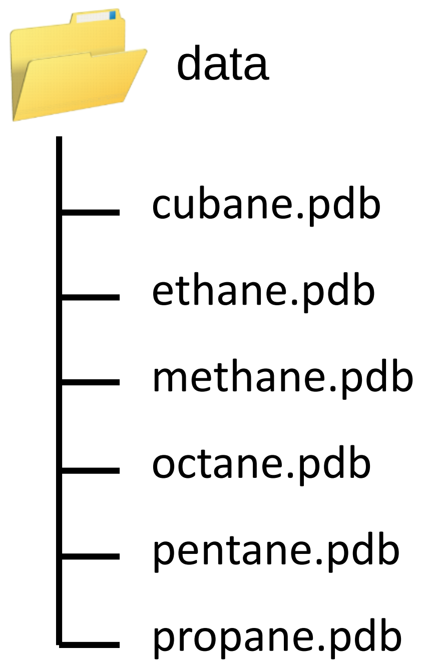


We can use *loops* for this...

So what can we do with loops?

So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

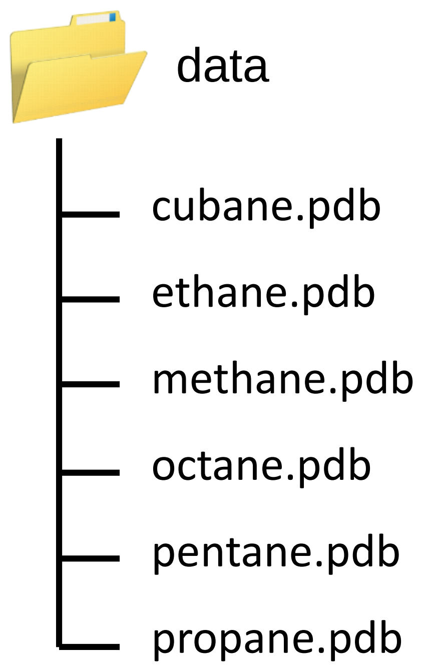


So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```



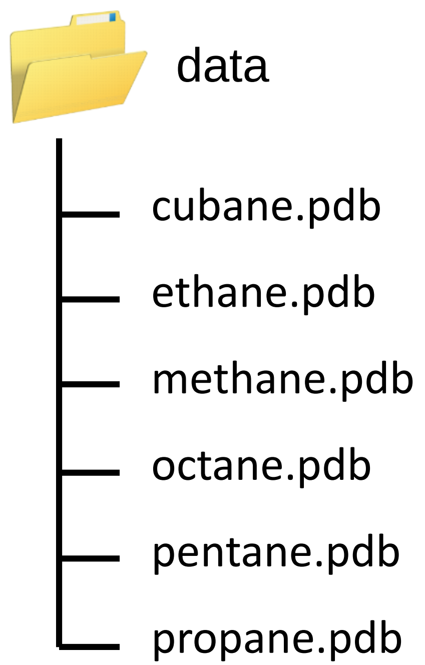
So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

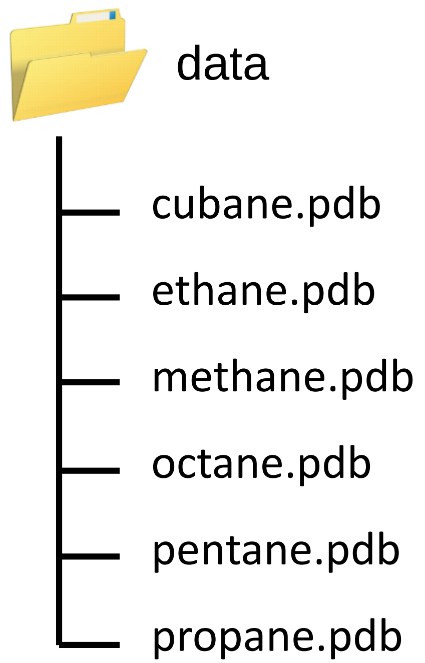
typical output from the zip command



So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:



```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

The zip file we wish to create

typical output from the zip command

So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

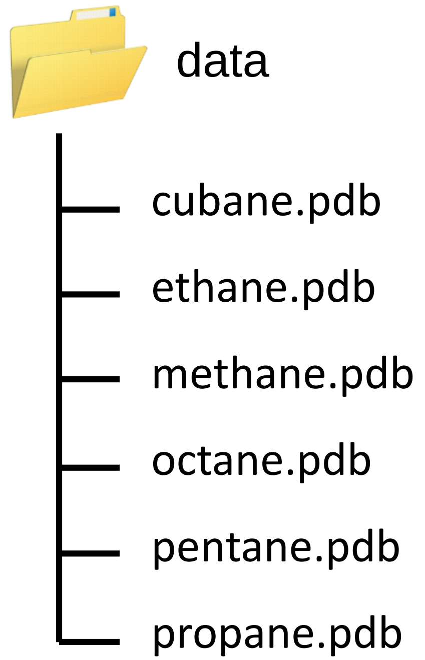
We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

The zip file we wish to create

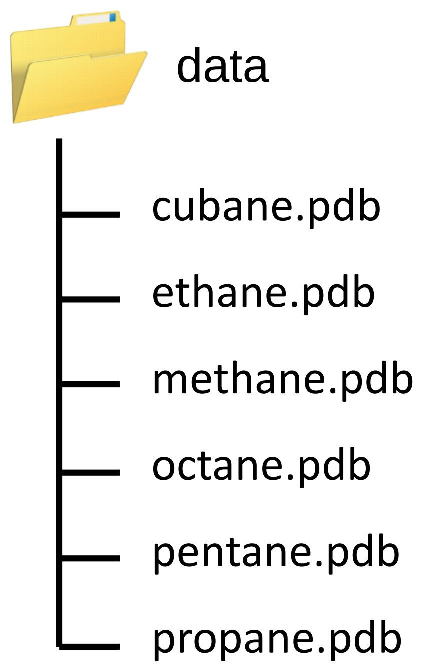
The file(s) we wish to add to the zip file

typical output from the zip command



So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them



We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

The zip file we wish to create

The file(s) we wish to add to the zip file

typical output from the zip command

Not efficient for many files

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

For each pdb file in this directory…

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

Run this command

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

This is the end of the loop

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

The semicolons separate each part of the loop construct

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

This expands to a list of every pdb file

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

This variable holds the next pdb file in the list

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

We reference the 'file' variable, and use '.' to add the zip extension to the filename

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

We reference the 'file' variable again

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
  adding: methane.pdb (deflated 66%)
  adding: octane.pdb (deflated 75%)
  adding: pentane.pdb (deflated 74%)
  adding: propane.pdb (deflated 71%)
```

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
  ...
```

In one line, we've ended up with all files zipped

Using a loop, we can iterate over each file,
and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
  ...
```

In one line, we've ended up with all files zipped

```
$ ls *.zip
cubane.pdb.zip    methane.pdb.zippentane.pdb.zip
ethane.pdb.zip    octane.pdb.zip    propane.pdb.zip
```

Now instead, what if we wanted to output the first line of each pdb file?

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

```
==> cubane.pdb <==
COMPND      CUBANE

==> ethane.pdb <==
COMPND      ETHANE

==> methane.pdb <==
COMPND      METHANE
…
```

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

head produces this
(it's not in the file)

```
==> cubane.pdb <==
COMPND      CUBANE

==> ethane.pdb <==
COMPND      ETHANE

==> methane.pdb <==
COMPND      METHANE
…
```

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

```
==> cubane.pdb <==
COMPND      CUBANE

==> ethane.pdb <==
COMPND      ETHANE

==> methane.pdb <==
COMPND      METHANE
…
```

head produces this (it's not in the file)

this is actually the first line in this file!

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

```
==> cubane.pdb <==
COMPND       CUBANE

==> ethane.pdb <==
COMPND       ETHANE

==> methane.pdb <==
COMPND       METHANE
…
```

Perhaps we only want the actual first lines…

However, using a loop:

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
```

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
```

We use $file as we did before, but this time with the head command

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
COMPND      CUBANE
COMPND      ETHANE
COMPND      METHANE
COMPND      OCTANE
COMPND      PENTANE
COMPND      PROPANE
```

What if we wanted this list sorted in reverse afterwards?

What if we wanted this list sorted in reverse afterwards?

Simple!

```
$ (for file in ls *.pdb; do head -1 $file; done) | sort -r
```

What if we wanted this list sorted in reverse afterwards?

Simple!

```
$ (for file in ls *.pdb; do head -1 $file; done) | sort -r
```

Using a pipe, we can just add this on the end

What if we wanted this list sorted in reverse afterwards?

Simple!

```
$ (for file in ls *.pdb; do head -1 $file; done) | sort -r
COMPND        CUBANE
COMPND        ETHANE
COMPND        METHANE
COMPND        OCTANE
COMPND        PENTANE
COMPND        PROPANE
```

`zip`	Create a compressed zip file with other files in it
`for …; do … done;`	Loop over a list of data and run a command once for each element in the list

created by

Steve Crouch

July 2011

All this typing is giving me RSI

- Didn't we use a file to store the commands in earlier examples?

All this typing is giving me RSI

- Didn't we use a file to store the commands in earlier examples?

- - Yes we did. Let's have a look at how that works…

- If you put commands in a file and chmod the file to be executable then you can run the file as a command.

```
$ cat > y.sh
for i in *
do
echo == $i ==
head -1 $i
tail -1 $i
done
$ chmod 700 y.sh
$ ./y.sh
== a.txt ==
Hi there
xxx
== b.txt ==
File B
File B
== y.sh ==
for i in *
done
$
```

What happens when I try to run a file that is not a script?

```
$ more a.txt
Hi there
this
is
file a
Xxx
$
```

Unintended consequences

```
$ chmod 700 a.txt
$ ./a.txt
./a.txt: line 1: Hi: command not found
./a.txt: line 2: this: command not found
./a.txt: line 3: is: command not found
a: cannot open 'a' (No such file or
directory)
./a.txt: line 5: xxx: command not found
$
```

```
Hi there
this
is
file a
Xxx
```

How to make sure your script is a script

1) Only use the execute (x) permission if you are going to execute it.
2) Use an interpreter header as the first line of the file. (You can use the `which bash` to find your bash programs location.)

```
#!/bin/bash
for j in *

...
```

What about other control structures

- `if test -e myfile; then ...; fi`
- `case ...; esac`
- `while ...`
- I'm not going to tell you because

1) You can find this out on your own (`man bash`)
2) If you find yourself using if and case then you should probably switch to a programming language like python. It's safer.