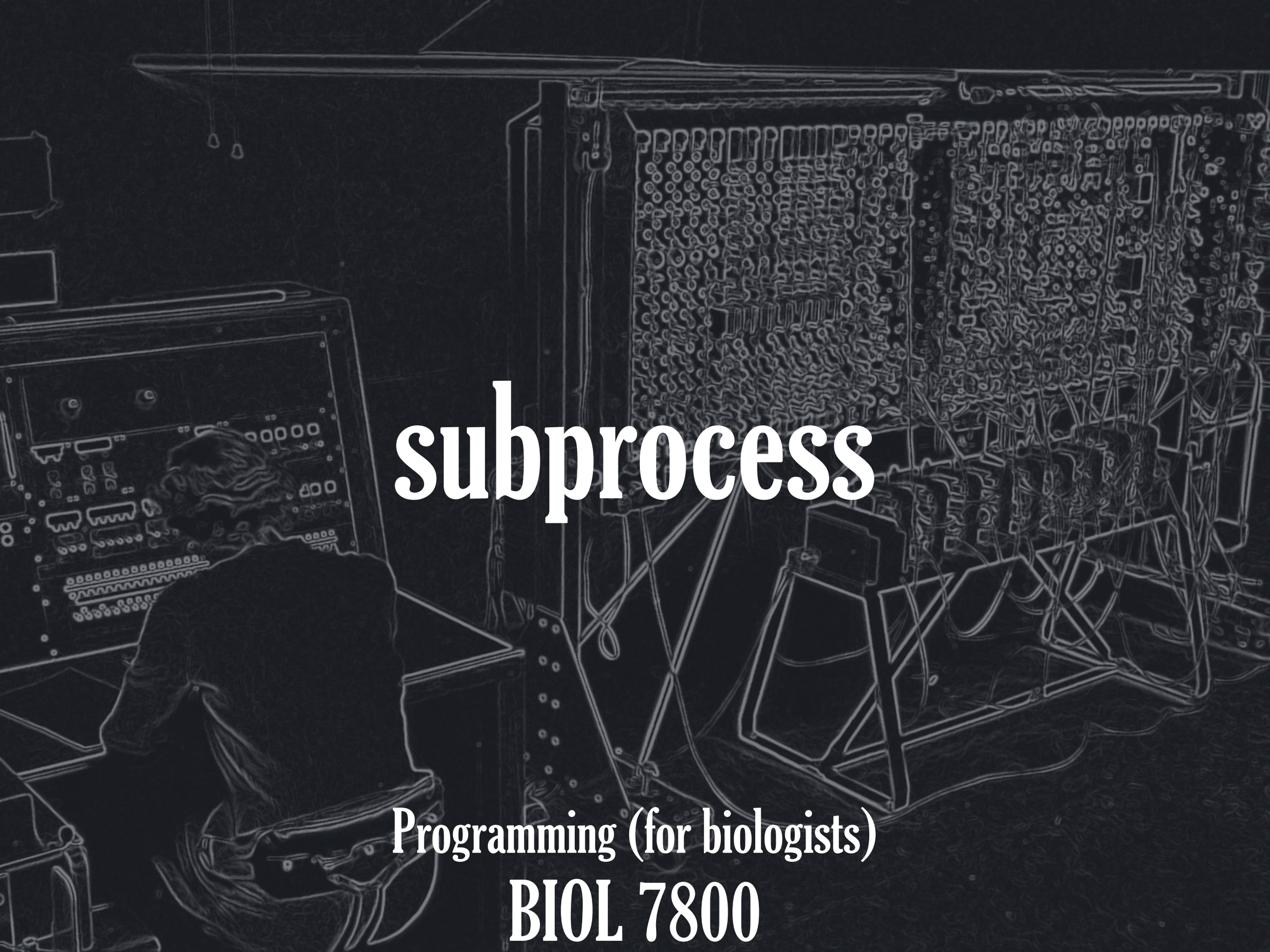


subprocess

Programming (for biologists)

BIOL 7800

code_example.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture20

code_example.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
(c) 2016 Brant Faircloth || http://faircloth-lab.org/
All rights reserved.
This code is distributed under a 3-clause BSD license. Please see
LICENSE.txt for more information.
Created on 17 March 2016 16:01 CDT (-0500)
"""

import re
import argparse
from collections import Counter

# import pdb


def get_args():
    """Get arguments from CLI"""
    parser = argparse.ArgumentParser(
        description="""Reads a file in; write some stuff out"""
    )
    parser.add_argument(
        "--infile",
        required=True,
        type=str,
        help="""The path to the input file"""
    )
    parser.add_argument(
        "--output",
        required=True,
        type=str,
        help="""The path to the output file"""
    )
    return parser.parse_args()


def get_and_clean_file(f):
    with open(f, 'r') as infile:
        text = infile.read()
    return re.findall("\w+['-]*\w*", text.lower())


def pretty_print_counts(cnt, top=20):
    for word, cnt in cnt.most_common(top):
        print("{: <20}{}".format(word, cnt))


def pretty_write_counts(o, cnt):
    with open(o, 'w') as outfile:
        # don't need to sort here - most_common gives us sorted output.
        for word, cnt in cnt.most_common():
            outfile.write("{}\t{}\n".format(word, cnt))


def main():
    args = get_args()
    clean_text = get_and_clean_file(args.infile)
    cnt = Counter(clean_text)
    pretty_print_counts(cnt)
    pretty_write_counts(args.output, cnt)


if __name__ == '__main__':
    main()
```

File 0 Project 0 ✔ No Issues code_example.py 66:11 LF UTF-8 Python

Previously…

You've been writing code to accomplish tasks

New

Classes

Methods

Attributes

Functions

But often…

Programs already exist that do something you want to do

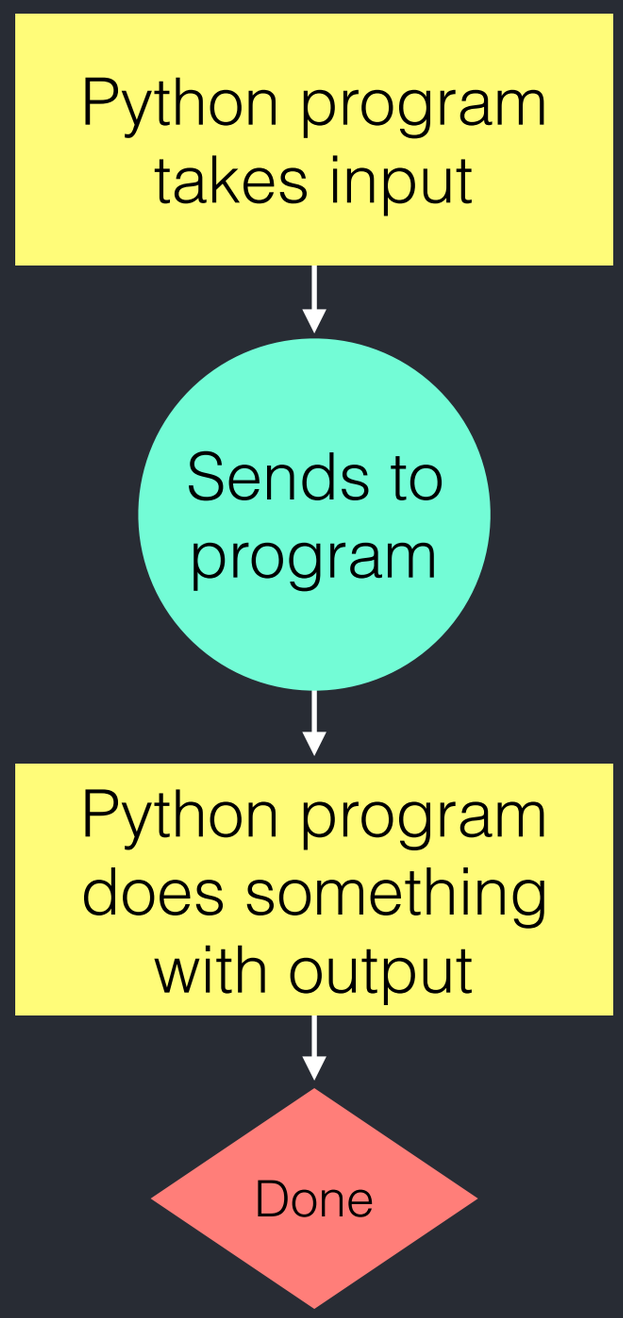
Shell commands	Sequence assembly	Alignment
tail	velvet	bwa
head	trinity	bowtie
wc	abYss	lastz
cat		blast
		mafft
		muscle

Or, any other external program you want to run

(from within your Python code)

AKA Pipelining

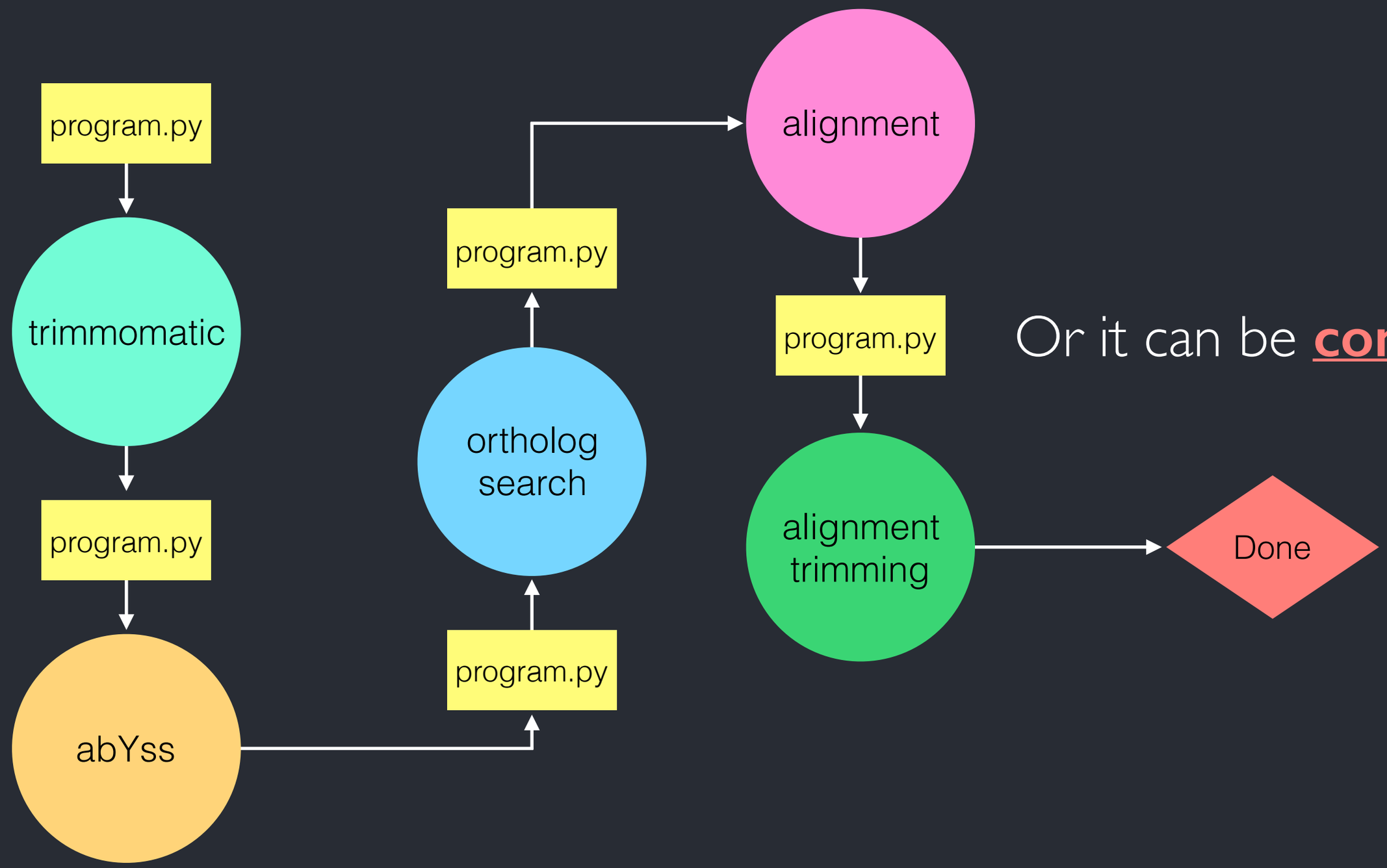
(or building a pipeline)



This can be simple

AKA Pipelining

(or building a pipeline)



Or it can be **complex**

But, how do you do all of this?

subprocess

A python (standard) module that allows you to

"spawn new processes, connect to their input/output/error pipes and obtain their return codes"

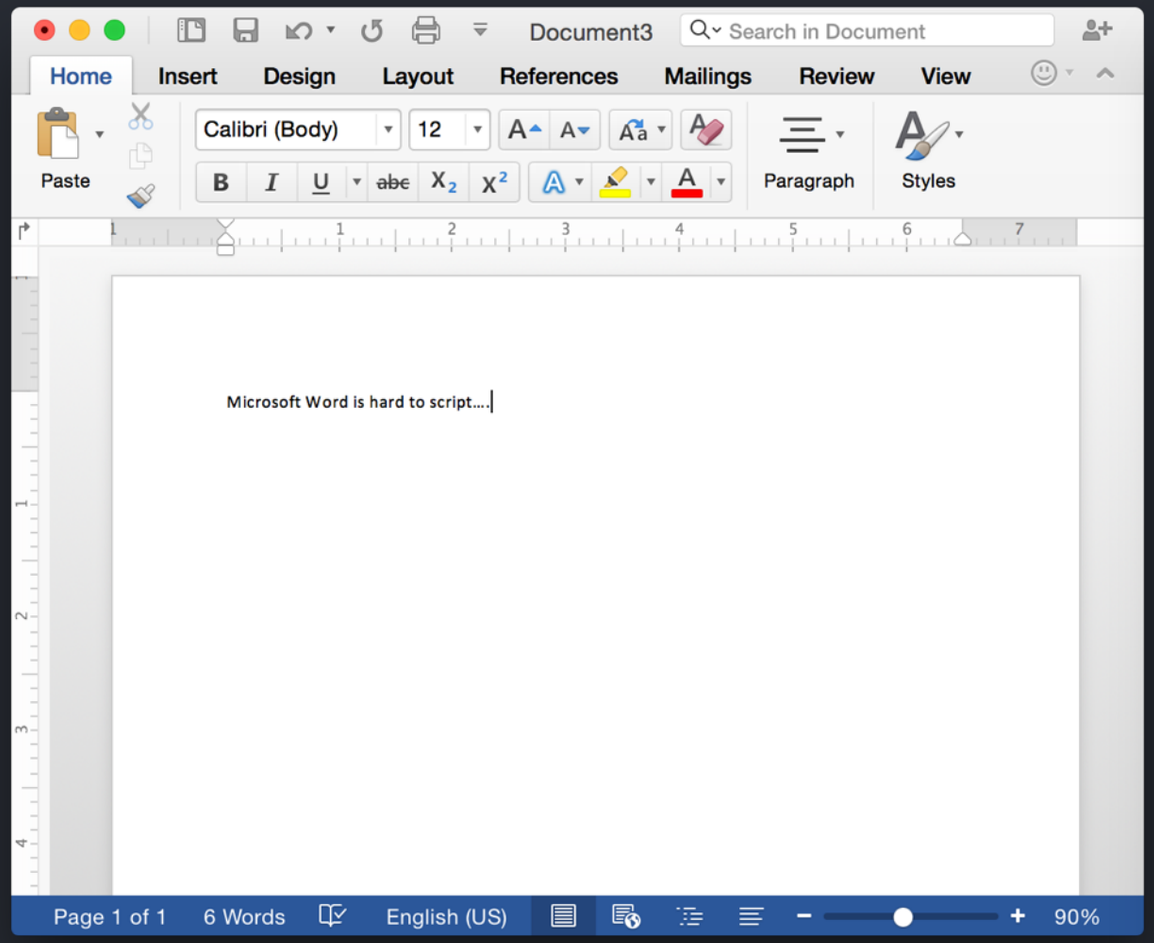
- python 3.5 doc

subprocess

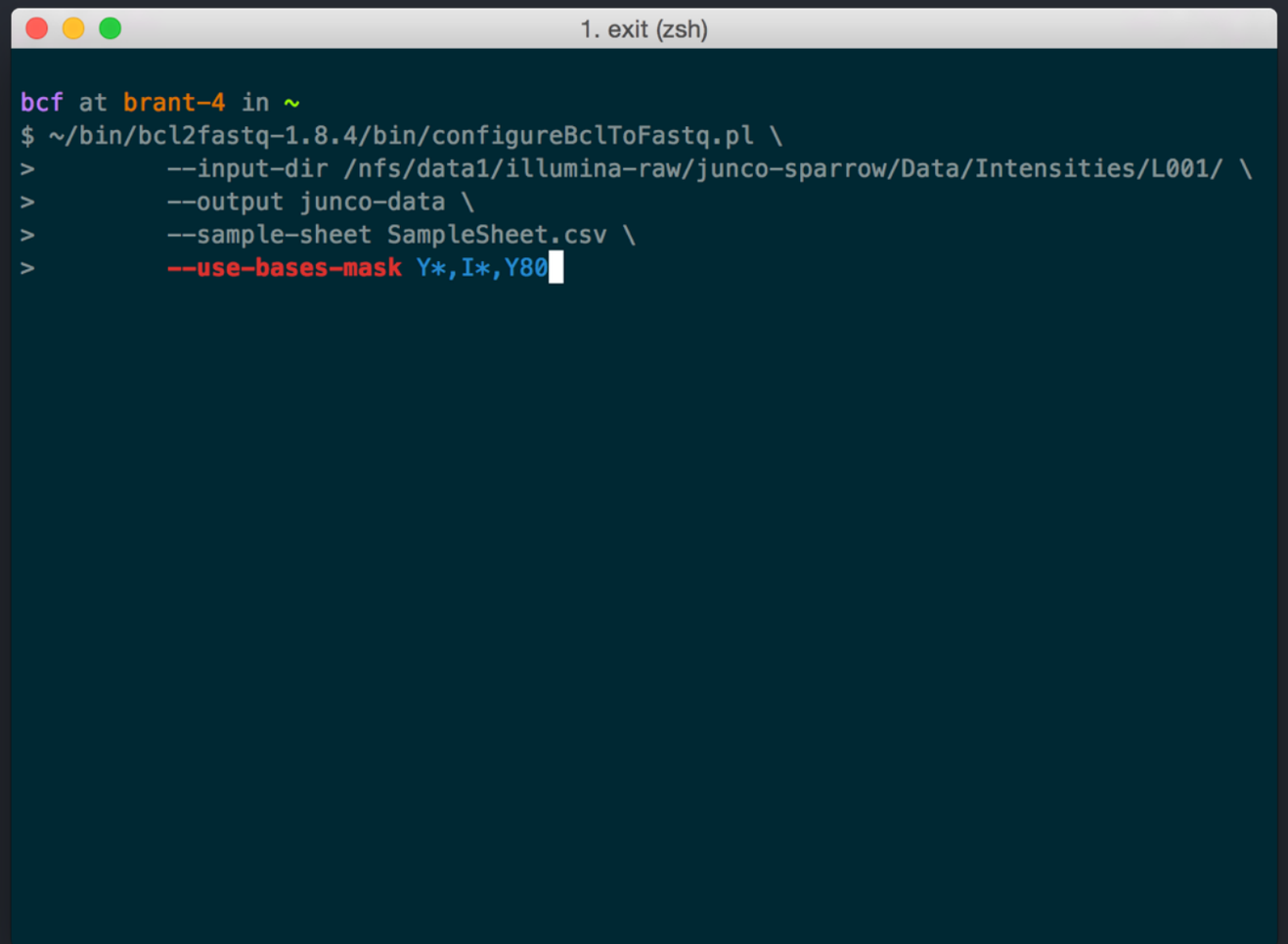
What does **subprocess** require to make this magic happen?

Basically, that you have a "scriptable" program
you want to run from python

Not so much GUI



More like command-line

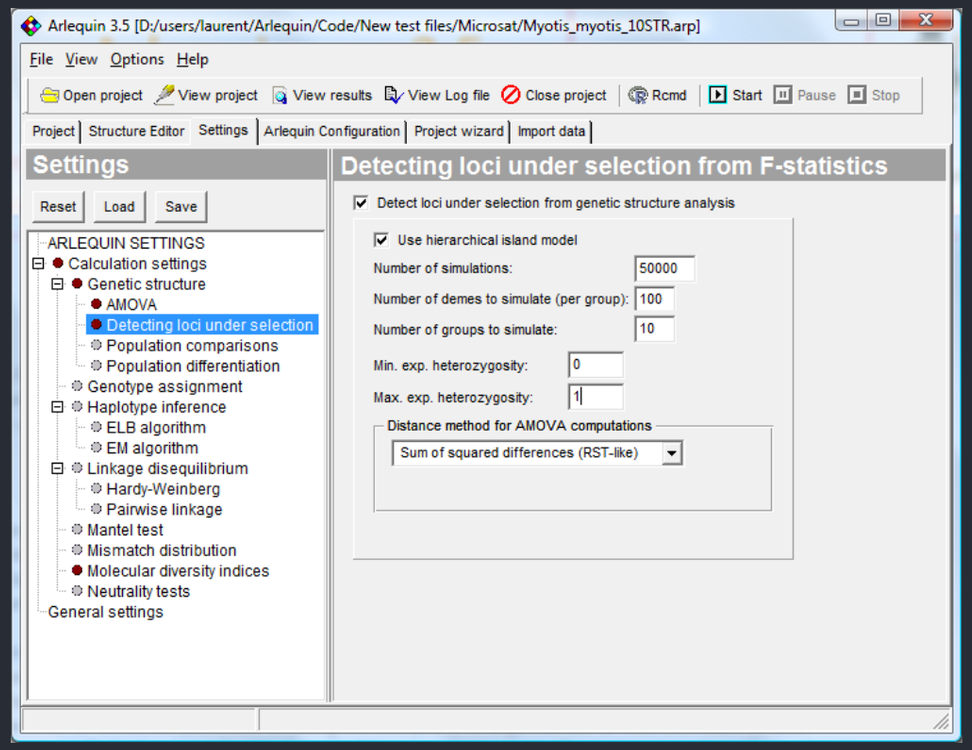


subprocess

What does **subprocess** require to make this magic happen?

Basically, that you have a "scriptable" program
you want to run from python

Generally, that means a command-line program
but *not always*



A good example is Arlequin

Has both GUI and command-line interaction

anatomy of subprocess

In: `import subprocess`

Import subprocess

`run()` method is responsible for running external program

program (and arguments) goes in a `list`

In: `subprocess.run(['ls'])`

Out: `CompletedProcess(args=['ls'], returncode=0)`

Use `run` to run prog

`subprocess` returns a `CompletedProcess` object

The `returncode` is programming lingo for

`success = 0`

`failure = 1`

anatomy of subprocess

```
In:   import subprocess

In:   my_process = subprocess.run(['ls'])
In:   dir(my_process)
Out:
'args',
 'check_returncode',
 'returncode',
 'stderr',
 'stdout'
In:   my_process.args
Out: ['ls']
In:   my_process.check_returncode()
Out:
In:   print(my_process.check_returncode())
Out: None
```

We can store CompletedProcess object of subprocess in a variable

These are the CompletedProcess attributes & methods

We can access the .args attribute

We can use the check_returncode() method to make sure program ran

check_returncode() returns None if program ran successfully

anatomy of subprocess

```
In:  import subprocess
```

These arguments are wrong
(they don't work)

```
In:  my_process = subprocess.run(['ls', '-ZZZ'])
In:  my_process.args
Out: ['ls', '-ZZZ']
```

Use **run** to run prog
(with bad arguments)

```
In:  my_process.returncode
Out: 1
In:  my_process.check_returncode()
Out:
```

Let's look at actual
returncode

Now, let's check it…

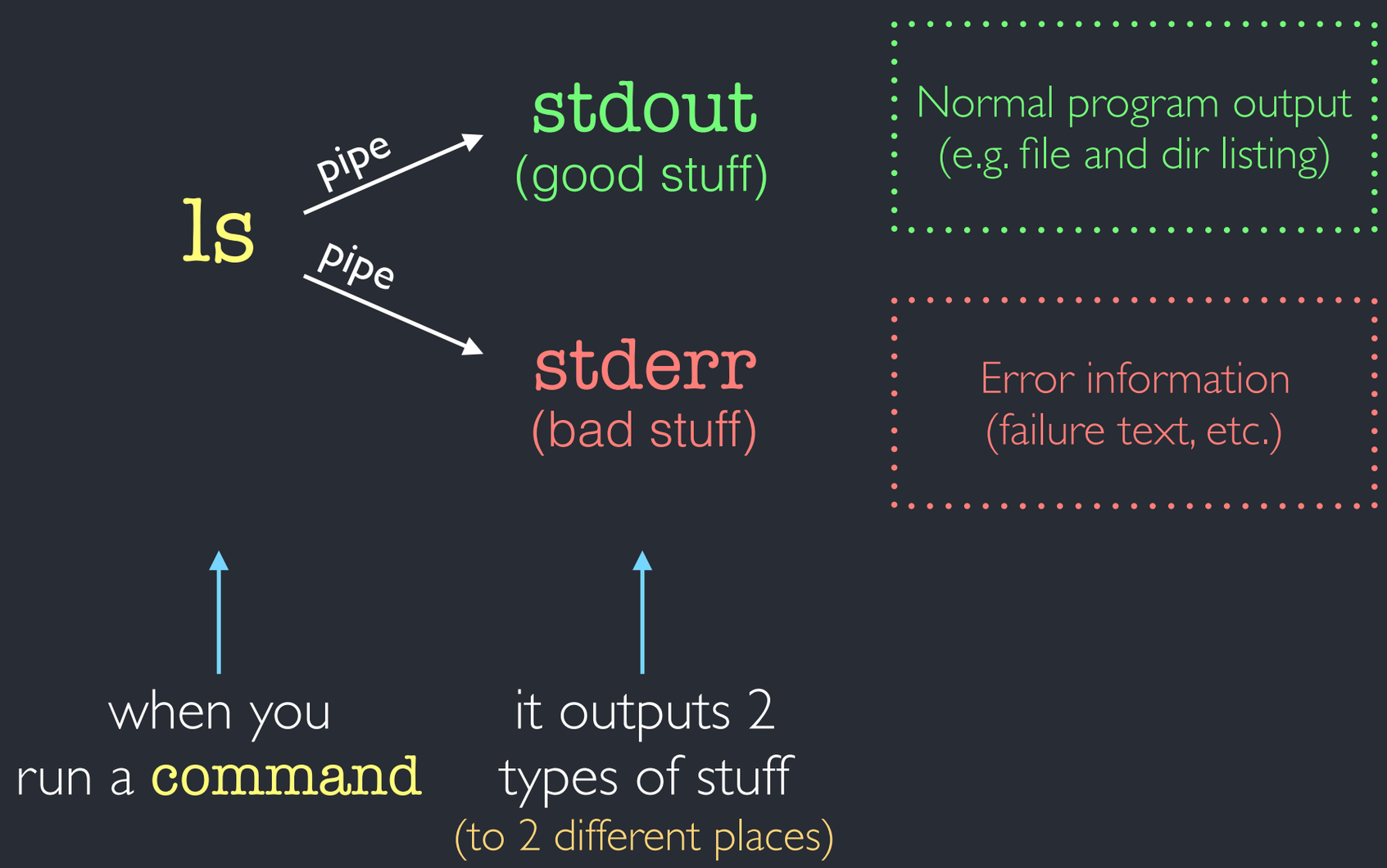
```
CalledProcessError                        Traceback (most recent call last)
<ipython-input-21-c11267dce8c6> in <module>()
----> 1 my_process.check_returncode()

/Users/bcf/anaconda/envs/py35/lib/python3.5/subprocess.py in check_returncode(self)
    660         if self.returncode:
    661             raise CalledProcessError(self.returncode, self.args, self.stdout,
--> 662                                      self.stderr)
    663
    664
CalledProcessError: Command '['ls', '-ZZZ']' returned non-zero exit status 1
```

Error !

anatomy of subprocess

okay, that's great, but where's our output?

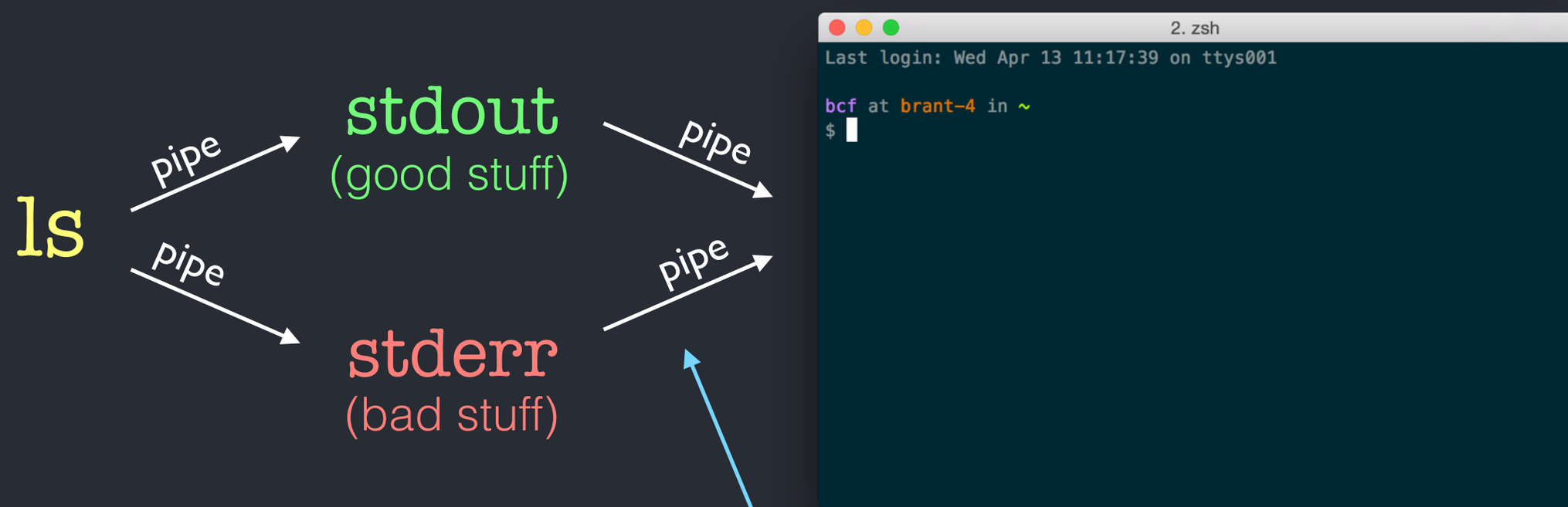
(a quick diversion into the land of pipes)



anatomy of subprocess

okay, that's great, but where's our output?

When you run a command from your shell



when you run a **command**

it outputs 2 types of stuff (to 2 different places)

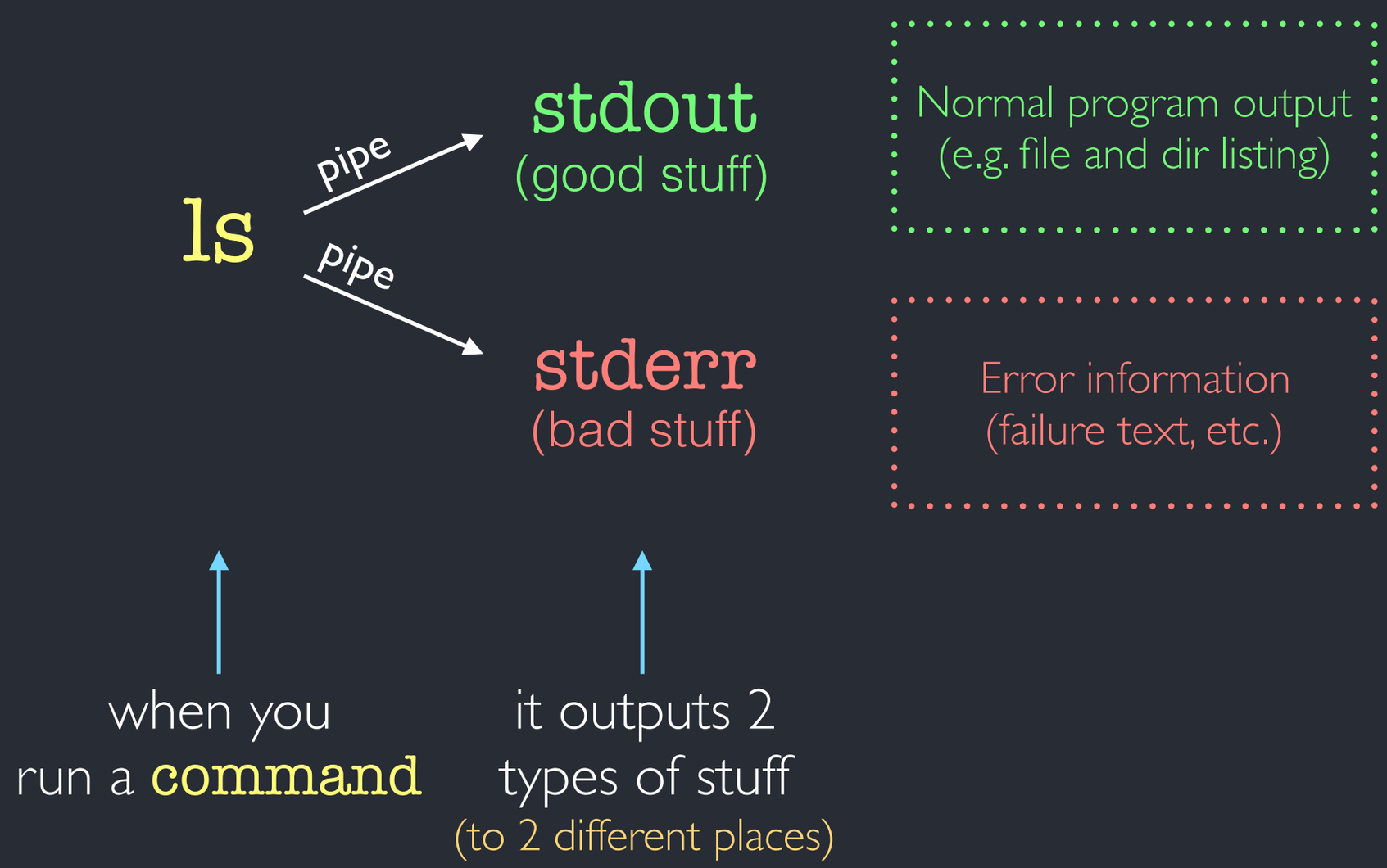
Often, these are merged back together

And sent to your terminal

anatomy of subprocess

When you run a command from subprocess

The ideal situation



anatomy of subprocess

When you run a command from subprocess

But it's not always ideal…

program → & (good stuff) (bad stuff) —pipe→ stdout

Normal program output
(e.g. file and dir listing)

Error information
(failure text, etc.)

some programs take everything

and write it **all** to stdout

anatomy of subprocess

When you run a command from subprocess

But it's not always ideal…

program → & (good stuff) (bad stuff) —pipe→ stderr

Normal program output
(e.g. file and dir listing)

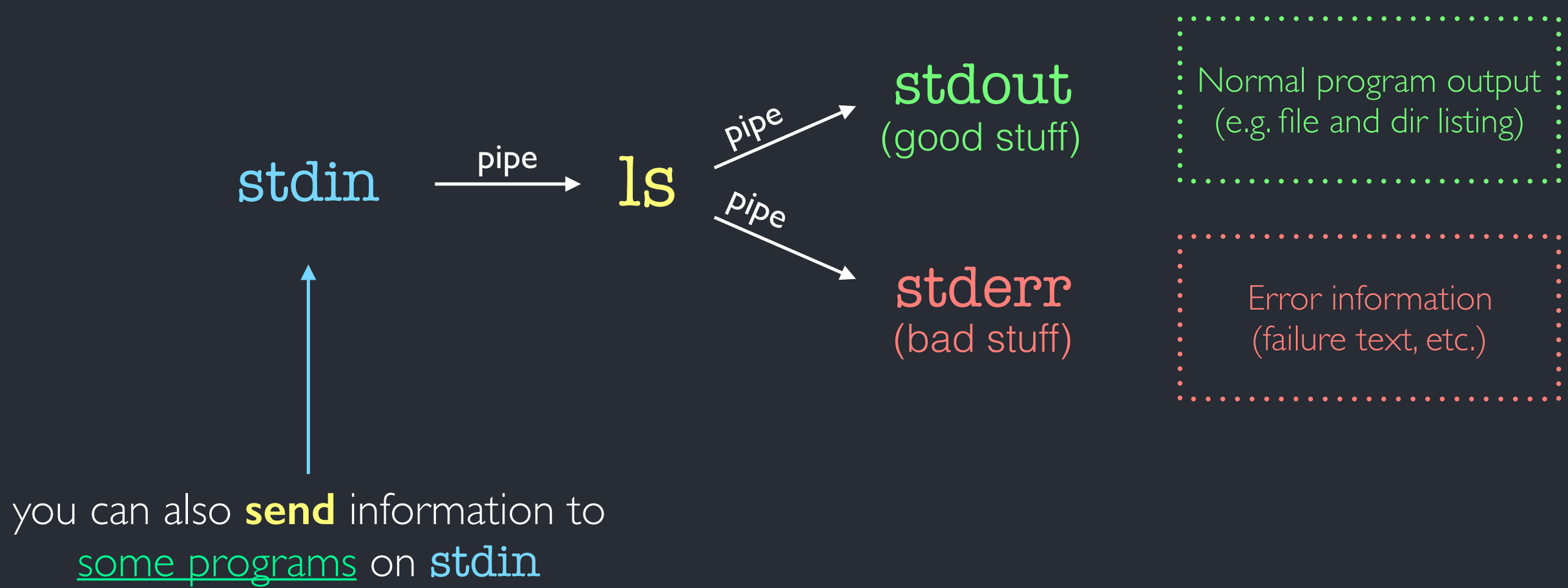
Error information
(failure text, etc.)

some programs take everything

and write it **all** to stderr

anatomy of subprocess

One more little wrinkle…



anatomy of subprocess

So, generally, we have 3 types of things to worry about

And, the stdout is sometimes different from what the program can write to a file

(although we can also store stdout and stdin in a file)

anatomy of subprocess

```
In:  import subprocess
```

program (and arguments) goes in a **list**

set keyword stdout to **subprocess.PIPE**

We're only capturing **stdout**

```
In:  my_proc = subprocess.run(['ls'], stdout=subprocess.PIPE)
In:  my_proc.args
Out: ['ls']

In:  my_proc.returncode
Out: 0

In:  my_proc.stdout
Out: b'Applications\nCreative Cloud Files
\nDesktop\nDocuments\nDownloads\nDropbox
\nDropbox (faircloth-lab)\nLibrary
\nMesquite_Support_Files\nMovies\nMusic
\nPictures\nPublic\nVirtual Machines
\nanaconda\nbin\ngit\ngithub-app\nnotebooks
\nsrc\ntmp\n'
```

See if program ran successfully

stdout is in the **stdout** attribute

stdout is returned as a **bytes** object, which is not the same as a string

(more in a minute)

anatomy of subprocess

```
In:  my_proc = subprocess.run(['ls'],
                              stdout=subprocess.PIPE,
                              stderr=subprocess.PIPE)
```

Capture stdout and stderr

```
In:  my_proc.returncode
Out: 0
```

See if program ran successfully (it did)

```
In:  my_proc.stdout
Out: b'Applications\nCreative (…truncated…)'
```

stdout is in the stdout attribute

```
In:  my_proc.stderr
Out: b''
```

There is nothing in stderr attribute (program ran successfully)

anatomy of subprocess

These arguments are wrong
(they don't work)

```
In:  my_proc = subprocess.run(['ls', '-ZZZ'],
                              stdout=subprocess.PIPE,
                              stderr=subprocess.PIPE)
```

Capture stdout and stderr

```
In:  my_proc.returncode
Out: 1
```

See if program ran successfully
(it didn't)

```
In:  my_proc.stdout
Out: b''
```

There is nothing in stdout
(didn't run)

```
In:  my_proc.stderr
Out: b'ls: illegal option -- Z\nusage: ls [-
ABCFGHLOPRSTUWabcdefghiklmnopqrstuwx1] [file ...]\n'
```

But there is something in stderr
(because the program failed)

anatomy of subprocess

Can we do something with the stdout?

```
In:  my_proc = subprocess.run(['ls'],
                              stdout=subprocess.PIPE,
                              stderr=subprocess.PIPE)
```

Capture stdout and stderr

```
In:  my_proc.stdout
Out: b'Applications\nCreative Cloud Files\nDesktop
\nDocuments\nDownloads\nDropbox\nDropbox (faircloth-lab)
\nLibrary\nMesquite_Support_Files\nMovies\nMusic
\nPictures\nPublic\nVirtual Machines\nanaconda\nbin\ngit
\ngithub-app\nnotebooks\nsrc\ntmp\n'
```

stdout is returned as a bytes object, which is not the same as a string

```
In:  type(my_proc.stdout)
Out: bytes
```

stdout is returned as a bytes object, which is not the same as a string

bytes objects are a primitive way to encode a string

anatomy of subprocess

Can we do something with the stdout?

```
In:   my_proc = subprocess.run(['ls'],
                               stdout=subprocess.PIPE,
                               stderr=subprocess.PIPE)
```

Capture stdout and stderr

```
In:   my_proc.stdout
Out: b'Applications\nCreative Cloud Files\nDesktop
\nDocuments\nDownloads\nDropbox\nDropbox (faircloth-lab)
\nLibrary\nMesquite_Support_Files\nMovies\nMusic
\nPictures\nPublic\nVirtual Machines\nanaconda\nbin\ngit
\ngithub-app\nnotebooks\nsrc\ntmp\n'
```

stdout is returned as a bytes object, which is not the same as a string

```
In:   my_proc.stdout.split("\n")
Out: TypeError: a bytes-like object is required, not 'str'
```

This fails, because my_proc.stdout is a bytes object, not a string

And bytes objects need to be *encoded* to string objects

anatomy of subprocess

Two ways around the problem

Method 1

```
In:   my_proc = subprocess.run(['ls'],
                               stdout=subprocess.PIPE,
                               stderr=subprocess.PIPE)

In:   my_proc.stdout.decode("utf-8").split("\n")
Out:
['Applications',
 'Creative Cloud Files',
 'Desktop',
 'Documents',
 'Downloads',
 'Dropbox',
 'Dropbox (faircloth-lab)',
 'Library',
 'Mesquite_Support_Files',
 'Movies',
 (…truncated…)
 '']
```

Use the bytes object's decode() method to convert it to "utf-8" (or other) encoding

anatomy of subprocess

Two ways around the problem

Method 2

```
In:  my_proc = subprocess.run(['ls'],
                              stdout=subprocess.PIPE,
                              stderr=subprocess.PIPE,
                              universal_newlines=True)
```

← Pass the universal_newlines argument to subprocess.run()

```
In:  type(my_proc.stdout)
Out: str
```

← stdout is now a string

```
In:  my_proc.stdout.split("\n")
Out:
['Applications',
 'Creative Cloud Files',
 'Desktop',
 'Documents',
 'Downloads',
 'Dropbox',
 (…truncated…)
 '']
```

← And, we can easily split a string

anatomy of subprocess

Pipes are nice, but we may want to write stdout/stderr to a file

(particularly when there is a **lot** of it - our pipes can get full)

```
In:   with open('stdout.txt', 'w') as stdout_file:
          with open('stderr.txt', 'w') as stderr_file:
              my_proc = subprocess.run(['ls'],
                                       stdout=stdout_file,
                                       stderr=stderr_file,
                                       universal_newlines=True)
```

Open two files, one each to capture stdout and stderr

```
In:   print(open('stdout.txt', 'r').read())
Out:
Applications
Creative Cloud Files
Desktop
Documents
Downloads
Dropbox
(…truncated…)
```

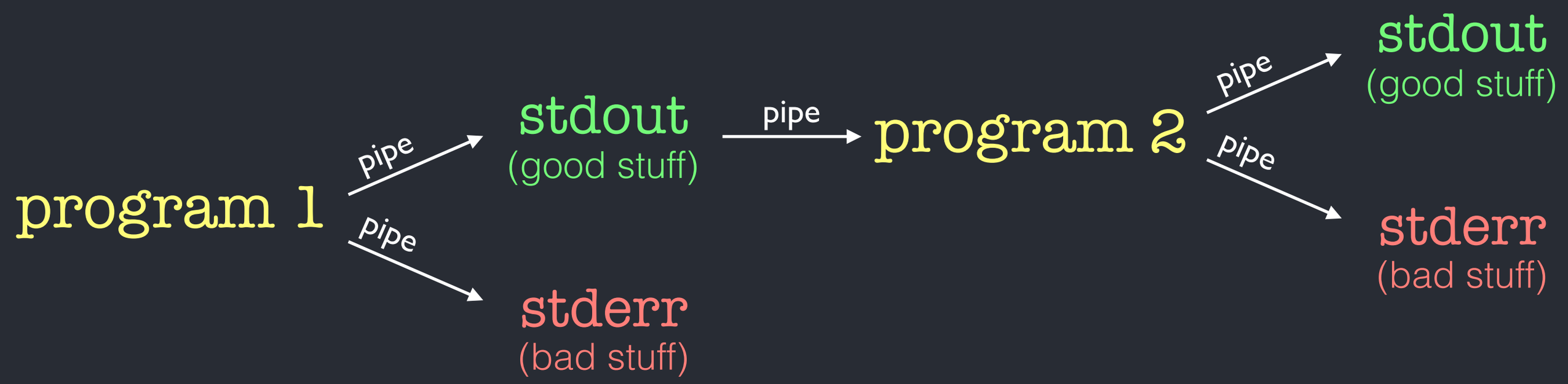
The lines that **were** going to our stdout pipe are now being written to a file

Same for stderr
(but there's nothing there)

We can now do anything we want with these file (parse them, etc.)

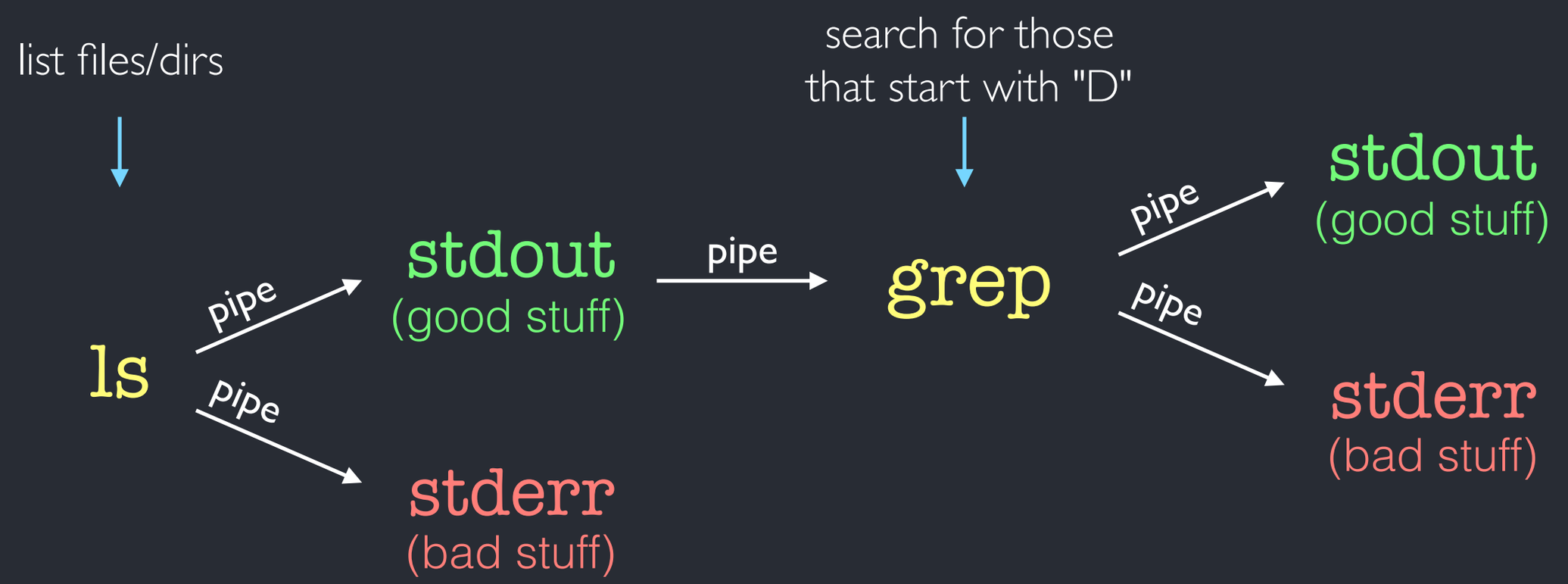
anatomy of subprocess

What if we want to chain several programs together ?



anatomy of subprocess

What if we want to chain several programs together ?



or, as we would type it in the shell

```
ls | grep "^D"
```

anatomy of subprocess

Chaining programs together by pipes

```
In:  my_proc_1 = subprocess.run(['ls'],
                 stdout=subprocess.PIPE,
                 stderr=subprocess.PIPE)
```

Run ls, send the output to stdout

```
     my_proc_2 = subprocess.run(['grep', '^D'],
                 input=my_proc_1.stdout,
                 stdout=subprocess.PIPE,
                 stderr=subprocess.PIPE)
```

Take the my_proc_1.stdout as input, and run grep against it

```
     print(my_proc_2.stdout)
```

```
Out: b'Desktop\nDocuments\nDownloads\nDropbox\nDropbox (faircloth-lab)\n'
```

stdout is returned as a bytes object

anatomy of subprocess

Chaining programs together by pipes (with string output)

In:
```
my_proc_1 = subprocess.run(['ls'],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
                           universal_newlines=True)

my_proc_2 = subprocess.run(['grep', '^D'],
                           input=my_proc_1.stdout,
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
                           universal_newlines=True)

print(my_proc_2.stdout)
```

We need to specify universal_newlines in both processes for string output

Out:
```
Desktop
Documents
Downloads
Dropbox
Dropbox (faircloth-lab)
```

stdout is returned as a string

anatomy of subprocess

What about programs with lots of parameters?

e.g.

```
java -jar /usr/bin/gatk -T UnifiedGenotyper -nt 12 -R genome.fasta -I genome.bam -gt_mode DISCOVERY -glm INDEL
```

You make a really long list of all the parameters…

Each part of a parameter is a list item

Numbers **must** be converted to strings

```
In:   my_proc_1 = subprocess.run(['java', '-jar', '/usr/bin/
gatk', '-T', 'UnifiedGenotyper', '-nt', '12', '—R', 'genome.fasta',
'-I', 'genome.bam', '-gt_mode', 'DISCOVERY', '-glm', 'INDEL'],
                                stdout=subprocess.PIPE,
                                stderr=subprocess.PIPE,
                                universal_newlines=True)
```

anatomy of subprocess

What about programs with lots of parameters?

e.g.

```
java -jar /usr/bin/gatk -T UnifiedGenotyper -nt 12 -R genome.fasta -I genome.bam -gt_mode DISCOVERY -glm INDEL
```

```
In:  command = ['java', '-jar', '/usr/bin/gatk', '-T',
'UnifiedGenotyper', '-nt', '12', '—R', 'genome.fasta', '-I',
'genome.bam', '-gt_mode', 'DISCOVERY', '-glm', 'INDEL']
```

Because this is just a **list**, you can make the list and place it in a variable…

```
In:  my_proc_1 = subprocess.run(
                    command,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE,
                    universal_newlines=True)
```

Then, pass the variable to subprocess.run

anatomy of subprocess

What about programs with lots of parameters?

e.g.

```
java -jar /usr/bin/gatk -T UnifiedGenotyper -nt 12 -R genome.fasta -I genome.bam -gt_mode DISCOVERY -glm INDEL
```

In: `import shlex`

← You can also use the **shlex** module to help you split complex command line statements

In: `shlex.split('java -jar /usr/bin/gatk -T UnifiedGenotyper -nt 12 -R genome.fasta -I genome.bam -gt_mode DISCOVERY -glm INDEL')`

Out:

```
['java',
 '-jar',
 '/usr/bin/gatk',
 '-T',
 'UnifiedGenotyper',
 '-nt',
 '12',
 '-R',
 'genome.fasta',
 '-I',
 'genome.bam',
 '-gt_mode',
 'DISCOVERY',
 '-glm',
 'INDEL']
```

← Then, use the list that **shlex** makes as your input