Command-line interfaces

Eric Franzosa (franzosa@hsph.harvard.edu)

http://franzosa.net/bst273

Overview

- Announcements
- Command-line interfaces
  - Brief intro to Object-Oriented Programming (OOP)
- The `argparse` module
- Live-coding

Command-line interfaces

Review: the sys module

script.py (*open in Atom*)

```python
import sys

print( sys.argv )
```

The sys module gave us access to arguments pased at the command line via **sys.argv**

(*a terminal*)

```
$ python script.py x y z

  [“script.py”, “x”, “y”, “z”]
```

Review: the sys module

`script.py` (*open in Atom*)

```
import sys

print( sys.argv )

fh = open( sys.argv[1] )
for line in fh:
    print( line.strip( ) )
fh.close( )
```

We used this to pass file paths into our scripts (instead of hard-coding)

(*a terminal*)

```
$ python script.py data.txt

  [“script.py”, “data.txt”]

  Come gather 'round people
  Wherever you roam
  And admit that the waters
  Around you have grown
  And accept it that soon
  You'll be drenched to the bone.
  If your time to you
  Is worth savin'
  Then you better start swimmin'
  ...
```

Command-line interfaces

- We’d like these interfaces to be more like those of command-line tools
  - For example, `ls` or `grep`
- We’d like our programs to fail gracefully if given the wrong arguments
  - For example, “Sorry, I expected a file and a pattern”
- We’d like to use flags to provide arguments “out of order”
  - For example, `python grep.py --pattern “Eric” --input data.txt`
- We’d like more helpful guidance on the meaning of arguments
  - For example, `grep --help` or `man grep`

One possible idea…

grep.py (*open in Atom*)

```
import sys

print( sys.argv )

# store flag arguments in a dictionary
args_dict = {}
for i in range( len( sys.argv ) ):
  if “--” in sys.argv[i]:
    args_dict[sys.argv[i]] = sys.argv[i+1]

# use those arguments
fh = open( args_dict[“--input”] )
for line in fh:
  if args_dict[“--pattern”] in line:
    print( line.strip( ) )
fh.close( )
```

(*a terminal*)

```
$ python grep.py --pattern A --input data.txt

 [“script.py”, “--pattern”, “A”, “--input”, “data.txt”]

 And admit that the waters
 Around you have grown
 And accept it that soon
 And keep your eyes wide
 And don't speak too soon
 And there's no tellin' who
 ...
```

Don’t reinvent the wheel!

- The idea we just spelled out is nicely implemented (along with many other useful features) in Python’s **`argparse`** module
  - [`https://docs.python.org/3/library/argparse.html`](https://docs.python.org/3/library/argparse.html)
  - Part of today’s reading assignment
- This module is centered on a helpful `class` called **`ArgumentParser`**
  - Let’s talk about what a `class` is briefly…

What is a `class`?

Using modules

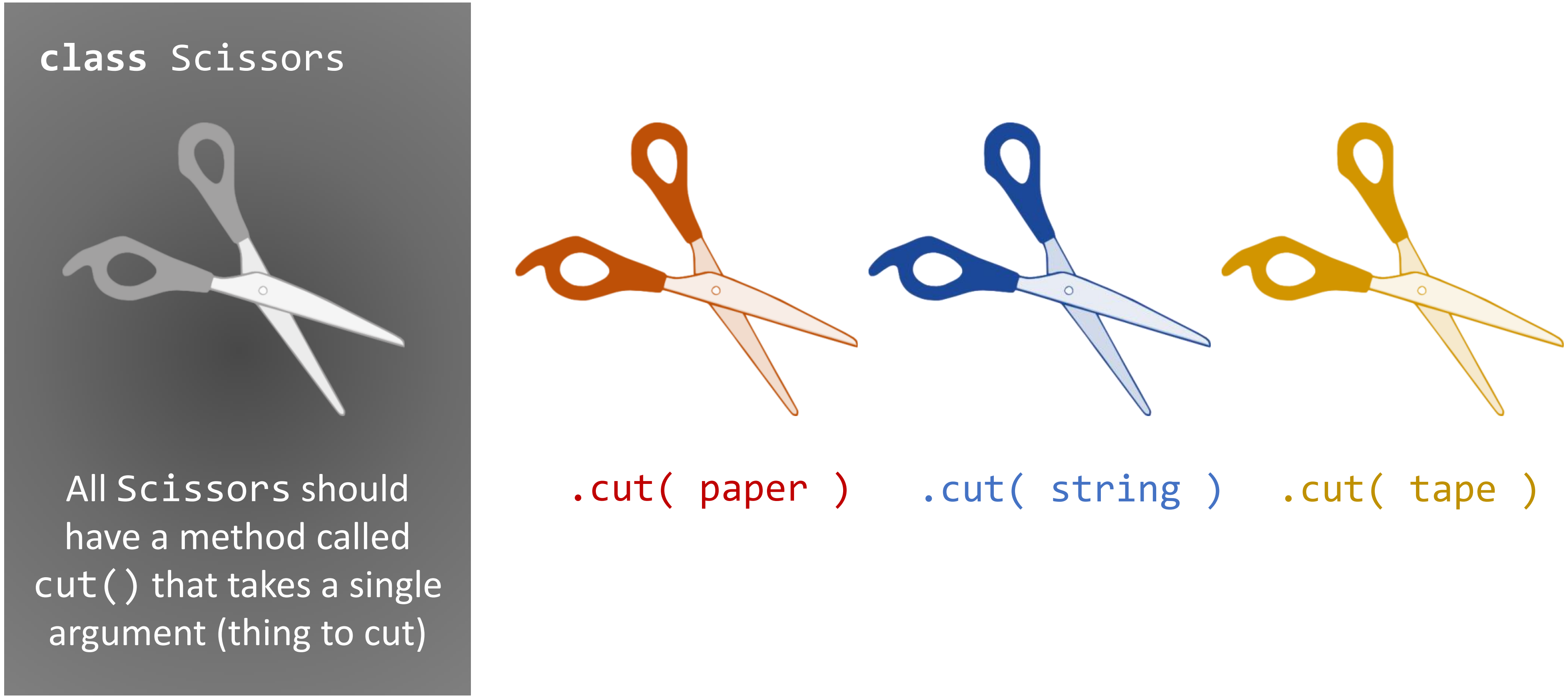
- Modules contain the same sort of elements as other Python code
- Modules can contain variables
  - `math.pi` *contains the value of pi (to many decimal places)*
  - `string.uppercase` *contains the uppercase English alphabet*
- Modules can contain functions
  - `time.sleep` *pauses computation for N seconds*
  - `math.sqrt` *returns the square root of a number*
- Modules can contain `classes` defining other data types
  - `collections.Counter` *a special dictionary for counting*
  - `Bio.Seq` *special strings for representing biological sequences*

What is a `class`?

- `classes` are a key aspect of "Object-Oriented Programming (OOP)"
- An "object" is a structure that contains some data + related functions
- For example, Python dictionaries are objects
  - They contain some data: *keys and values*
  - They contain some relevant functions (methods): e.g. `dict.items( )`
- A `class` is a definition for a particular kind (or `type`) of object
  - It's the Python code that dictates what kind of data lives inside a certain type of object, and what sort of methods the object can do
  - We'll talk about the syntax for that code later in the course
  - For now…

What is a `class`?

- Think of a `class` as being like a mold (or factory) for making a certain type of real-world object:



`class Scissors`

All Scissors should have a method called `cut()` that takes a single argument (thing to cut)

`.cut( paper )` `.cut( string )` `.cut( tape )`

What is a `class`?

- We call a class like a function to `return` a new “instance” of the class:
  - `Scissors( )`
- As usual, we want to put the returned value in a variable for later use:
  - `my_scissors = Scissors( )`
  - `my_scissors.cut( paper )`
- You’ve actually been doing this already (maybe without realizing it):
  - `my_list = []`
  - `my_list.append( 1 )`
- The fundamental built-in types (including `lists`) are also objects, but we usually create them using the above shortcuts for convenience
  - `my_list = list( )` also works

What is a `class`?

- Functions/methods have "verb" names and are usually pothole-case:
  - `add_argument`
- `classes` have "noun" names and are usually camel-case:
  - `ArgumentParser`
  - The built-in types (`str`, `float`, `list`, `dict`, etc.) are exceptions to this trend
- These are just conventions though
  - Not required for making valid Python code
  - Not everyone will follow them

What is a `class`?

- Many Python programs don’t need to define any new `class`es
- Many Python programs are fine using just the built-in Python types
- Additional `class`es become useful when working with complex data that aren’t handled well by the built-in Python types
  - Example 1: A family tree with parent-child relationships
  - Example 2: A character in a video game
  - *Example 3: A command-line interface*
- OOP / `class`es have other advantages that we will return to later

`argparse`

for building command-line interfaces

Using argparse: first steps

script.py (*open in Atom*)

```python
import argparse
```

We import argparse like any other module

(*a terminal*)

```
$ python script.py
```

Using argparse: first steps

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
args = parser.parse_args( )
```

These two lines of code make an ArgumentParser object (stored in parser) then call its method parse_args to gather command-line arguments and store them in args

(*a terminal*)

```
$ python script.py
```

Using argparse: adding an argument

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “my_string” )
args = parser.parse_args( )
```

We tell the parser what sort of arguments to look for by defining them with add_argument

(*a terminal*)

```
$ python script.py

  usage: script.py [-h] my_string
  script.py: error: too few arguments
```

If we try to use the script without the right arguments, we get a nice warning message

Using argparse: using an argument value

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “my_string” )
args = parser.parse_args( )

print( args.my_string )
```

Parameter values are stored in args in variables of the same name (accessed with . syntax)

(*a terminal*)

```
$ python script.py “Hello, World!”

  ‘Hello, World!’
```

Using argparse: using an argument value

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “input_file” )
args = parser.parse_args( )

fh = open( args.input_file )
for line in fh:
    print( line.strip( ) )
fh.close( )
```

We can naturally use this to define file paths

(*a terminal*)

```
$ python script.py data.txt

  Come gather 'round people
  Wherever you roam
  And admit that the waters
  Around you have grown
  And accept it that soon
  You'll be drenched to the bone.
  If your time to you
  Is worth savin'
  Then you better start swimmin'
  ...
```

Using argparse: multiple arguments

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “str1” )
parser.add_argument( “str2” )
args = parser.parse_args( )

print( args.str1 + args.str2 )
```

Use multiple calls to **add_argument** to describe multiple arguments

(*a terminal*)

```
$ python script.py “banana” “rama”

  ‘bananarama’
```

Using argparse: positional arguments

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “str1” )
parser.add_argument( “str2” )
args = parser.parse_args( )

print( args.str1 + args.str2 )
```

By default, arguments are positional (like positional arguments to functions)

(*a terminal*)

```
$ python script.py “rama” “banana”

  ‘ramabanana’
```

Using argparse: keyword arguments

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “--str1” )
parser.add_argument( “--str2” )
args = parser.parse_args( )

print( args.str1 + args.str2 )
```

We can define keyword arguments (flags) with the prefix “--”

(*a terminal*)

```
$ python script.py --str2 “rama” --str1 “banana”

  ‘bananarama’
```

Using argparse: arguments types

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “--int1” )
parser.add_argument( “--int2” )
args = parser.parse_args( )

print( args.int1 + args.int2 )
```

By default, argument values are parsed as string data

(*a terminal*)

```
$ python script.py --int1 11 --int2 9

  ‘119’
```

Using argparse: arguments types

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument( “--int1”, type=int )
parser.add_argument( “--int2”, type=int )
args = parser.parse_args( )

print( args.int1 + args.int2 )
```

We can tell the parser to automatically convert the arguments to other types

(*a terminal*)

```
$ python script.py --int1 11 --int2 9

  20
```

Using argparse

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument(
  “--int1”,
  type=int,
)
parser.add_argument(
  “--int2”,
  type=int,
)
args = parser.parse_args( )

print( args.int1 + args.int2 )
```

(*a terminal*)

```
$ python script.py --int1 11 --int2 9

  20
```

In fact, we can add a lot of useful information for each argument. I like to “wrap” mine to make them more readable.

Using argparse: default values

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument(
  “--int1”,
  type=int,
  default=0,
)
parser.add_argument(
  “--int2”,
  type=int,
  default=0,
)
args = parser.parse_args( )

print( args.int1 + args.int2 )
```

(*a terminal*)

```
$ python script.py --int1 11

  11
```

We can add a default value for any keyword parameter.

(*Another similarity to Python function definitions.*)

Using argparse: providing help

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument(
  “--int1”,
  type=int,
  help=“1st of two integers to add”,
)
parser.add_argument(
  “--int2”,
  type=int,
  help=“2nd of two integers to add”,
)
args = parser.parse_args( )

print( args.int1 + args.int2 )
```

(*a terminal*)

```
$ python script.py --help

  usage: script.py [-h] [--int1 INT1] [--int2 INT2]

  optional arguments:
    -h, --help   show this help message and exit
    --int1 INT1  1st of two integers to add
    --int2 INT2  2nd of two integers to add
```

We can add a description of the parameter to help users understand how to use our script (*doubles as documentation in our code*).

Using argparse: providing help

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser(
  description=“a program to add two integers”,
)
parser.add_argument(
  “--int1”,
  type=int,
  help=“1st of two integers to add”,
)
parser.add_argument(
  “--int2”,
  type=int,
  help=“2nd of two integers to add”,
)
args = parser.parse_args( )

print( args.int1 + args.int2 )
```

(*a terminal*)

```
$ python script.py --help

  usage: script.py [-h] [--int1 INT1] [--int2 INT2]

  a program to add two integers

  optional arguments:
    -h, --help   show this help message and exit
    --int1 INT1  1st of two numbers to add
    --int2 INT2  2nd of two numbers to add
```

We can add a description of the parameter to help users understand how to use our script

Using argparse: logical flags

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument(
  “--verbose”,
  action=“store_true”,
)
args = parser.parse_args( )

for i in range( 1000000 ):
    if args.verbose:
        print( i )
```

(*a terminal*)

```
$ python script.py --verbose

  0
  1
  2
  3
  4
  5
  …
```

We can set flags to act as Boolean values for our program (*similar to how* -v *behaves when inverting the search results from* grep).

Using argparse: logical flags

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument(
  “--write-those-numbers”,
  action=“store_true”,
)
args = parser.parse_args( )

for i in range( 1000000 ):
    if args.write_those_numbers:
        print( i )
```

(*a terminal*)

```
$ python script.py --write-those-numbers

  0
  1
  2
  3
  4
  5
  …
```

Note that “-”s (hyphens) in parameter names become “_”s (underscores) in **args** variables.

Any guesses why?

Using argparse: making choices

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument(
  “--mode”,
  choices=[“fast”, “faster”, “fastest”],
)
args = parser.parse_args( )

import time
if args.mode == “fastest”:
   time.sleep( 1 )
elif args.mode == “faster”:
   time.sleep( 5 )
elif args.mode == “fast”:
   time.sleep( 10 )
```

(*a terminal*)

```
$ python script.py --mode “slow”

  error: argument --mode: invalid choice: ‘slow'
  (choose from 'fast', 'faster', 'fastest')
```

We can also require a parameter value to belong to a set of pre-specified choices.

(Great for flow of control!)

Using argparse: lists of arguments

script.py (*open in Atom*)

```
import argparse

parser = argparse.ArgumentParser( )
parser.add_argument(
  “--fruits”,
  # lots of options for nargs
  nargs=“+”,
)
args = parser.parse_args( )

print( args.fruits )
```

(*a terminal*)

```
$ python script.py --fruits “apple” “pear”

  [‘apple’, ‘pair’]
```

By default, each argument expects one value. We can change this with nargs. Here, `--fruits` will gather any number of given fruits as a list.

Using `argparse`

- The preceding examples cover *most* of the things you'll want to with a command-line interface (CLI).
- Almost everything else is probably still possible using other options to `add_argument` that we didn't cover.
- There are also additional options for `ArgumentParser` itself that like you change the appearance of the CLI help menu.
  - For example, showing default values in help messages
- Consult the documentation as needed.

Using `argparse`

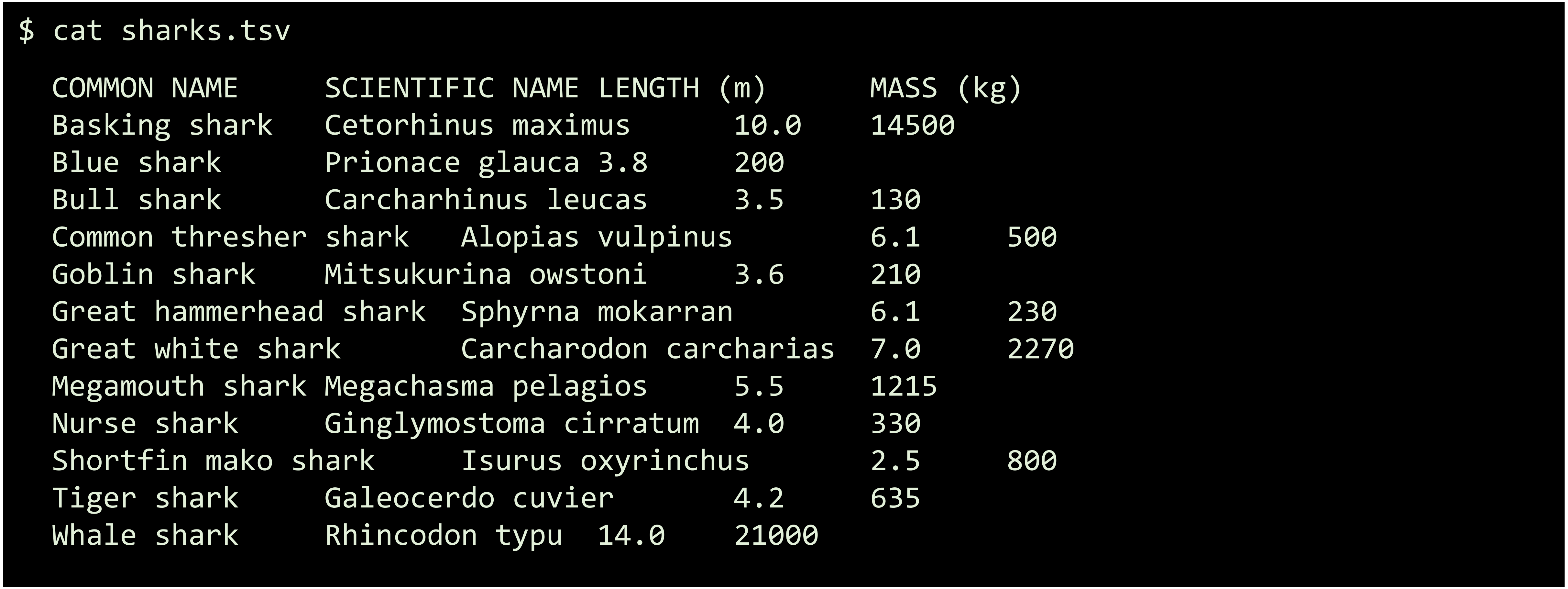
- Command-line interfaces are an example of “boilerplate” code
- Once you have written one nice one, copy/paste and modify for future use
- The `add_argument` syntax makes it easy to add/remove/tweak arguments

columns.py

a live-coding exercise

Has this ever happened to you?

(*a terminal*)



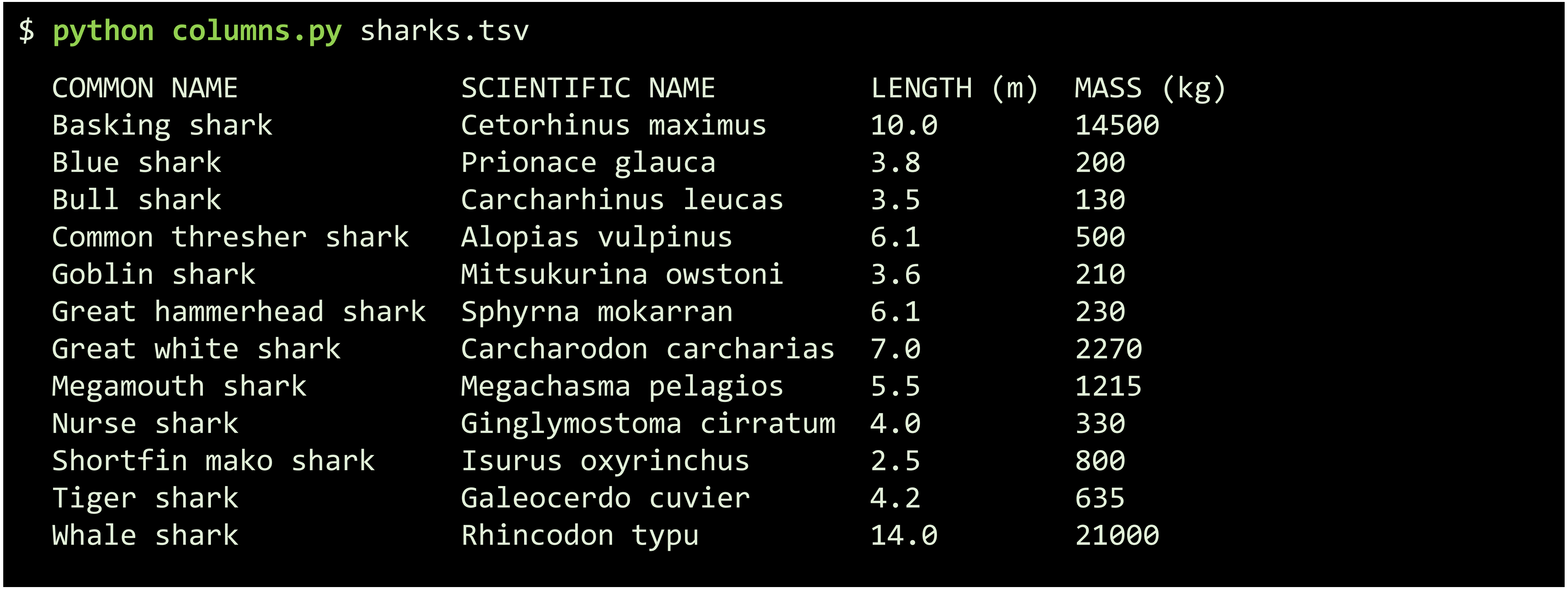
```
$ cat sharks.tsv

  COMMON NAME     SCIENTIFIC NAME LENGTH (m)      MASS (kg)
  Basking shark   Cetorhinus maximus      10.0    14500
  Blue shark      Prionace glauca 3.8     200
  Bull shark      Carcharhinus leucas     3.5     130
  Common thresher shark   Alopias vulpinus        6.1     500
  Goblin shark    Mitsukurina owstoni     3.6     210
  Great hammerhead shark  Sphyrna mokarran        6.1     230
  Great white shark       Carcharodon carcharias  7.0     2270
  Megamouth shark Megachasma pelagios     5.5     1215
  Nurse shark     Ginglymostoma cirratum  4.0     330
  Shortfin mako shark     Isurus oxyrinchus       2.5     800
  Tiger shark     Galeocerdo cuvier       4.2     635
  Whale shark     Rhincodon typu  14.0    21000
```

**Columns of data aren’t nicely lined up, making it hard to scan through their values*

columns.py

(a terminal)



```
$ python columns.py sharks.tsv

  COMMON NAME             SCIENTIFIC NAME         LENGTH (m)  MASS (kg)
  Basking shark           Cetorhinus maximus      10.0        14500
  Blue shark              Prionace glauca         3.8         200
  Bull shark              Carcharhinus leucas     3.5         130
  Common thresher shark   Alopias vulpinus        6.1         500
  Goblin shark            Mitsukurina owstoni     3.6         210
  Great hammerhead shark  Sphyrna mokarran        6.1         230
  Great white shark       Carcharodon carcharias  7.0         2270
  Megamouth shark         Megachasma pelagios     5.5         1215
  Nurse shark             Ginglymostoma cirratum  4.0         330
  Shortfin mako shark     Isurus oxyrinchus       2.5         800
  Tiger shark             Galeocerdo cuvier       4.2         635
  Whale shark             Rhincodon typu          14.0        21000
```

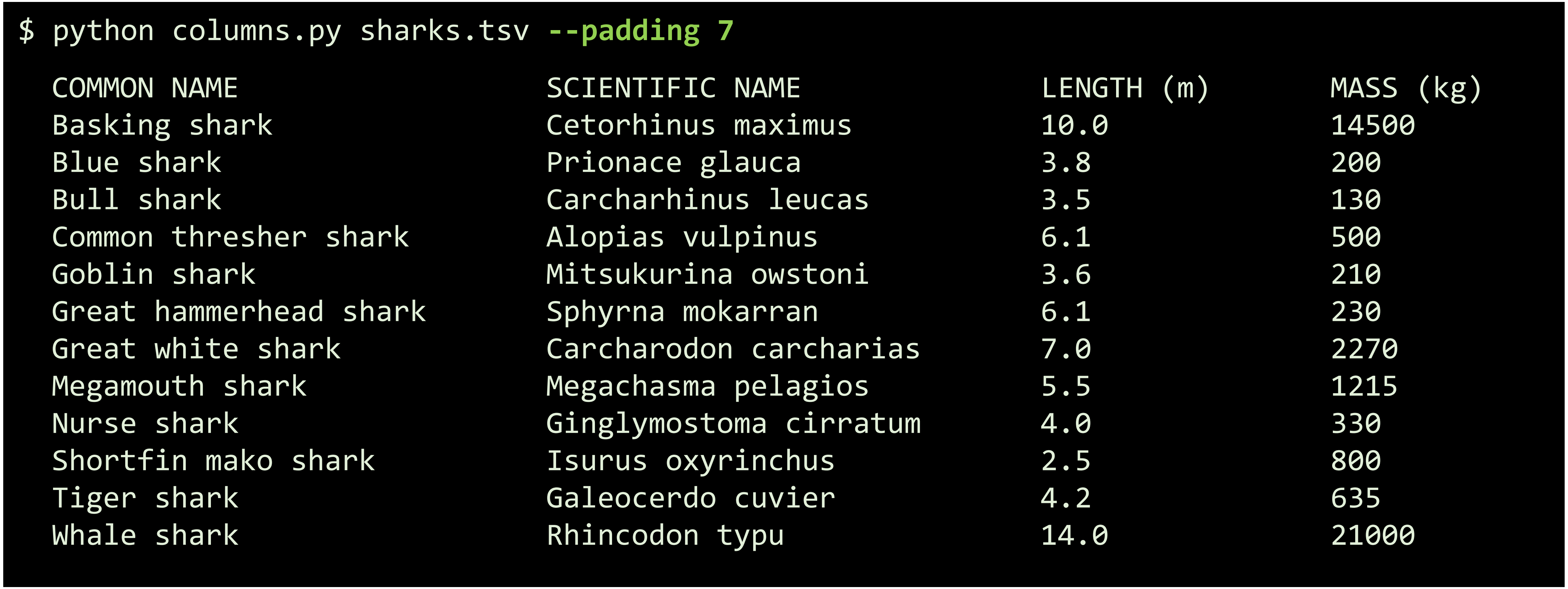
Let’s design a Python script (`columns.py`) to solve this problem + a few extras

What’s on Canvas?

- **`columns.py`**
  - **“Stub” script with just the command-line interface implemented**
- `columns_basic.py`
  - Script with basic functionality implemented
- `columns_extra.py`
  - Script with extra functionality implemented
- `columns_module.py`
  - Modularized version of the “extra” script
- **`sharks.tsv`**
  - **Data file, tab-separated**
- `sharks.csv`
  - Same data file, comma separated

`columns.py`: "extra padding" option

(a terminal)



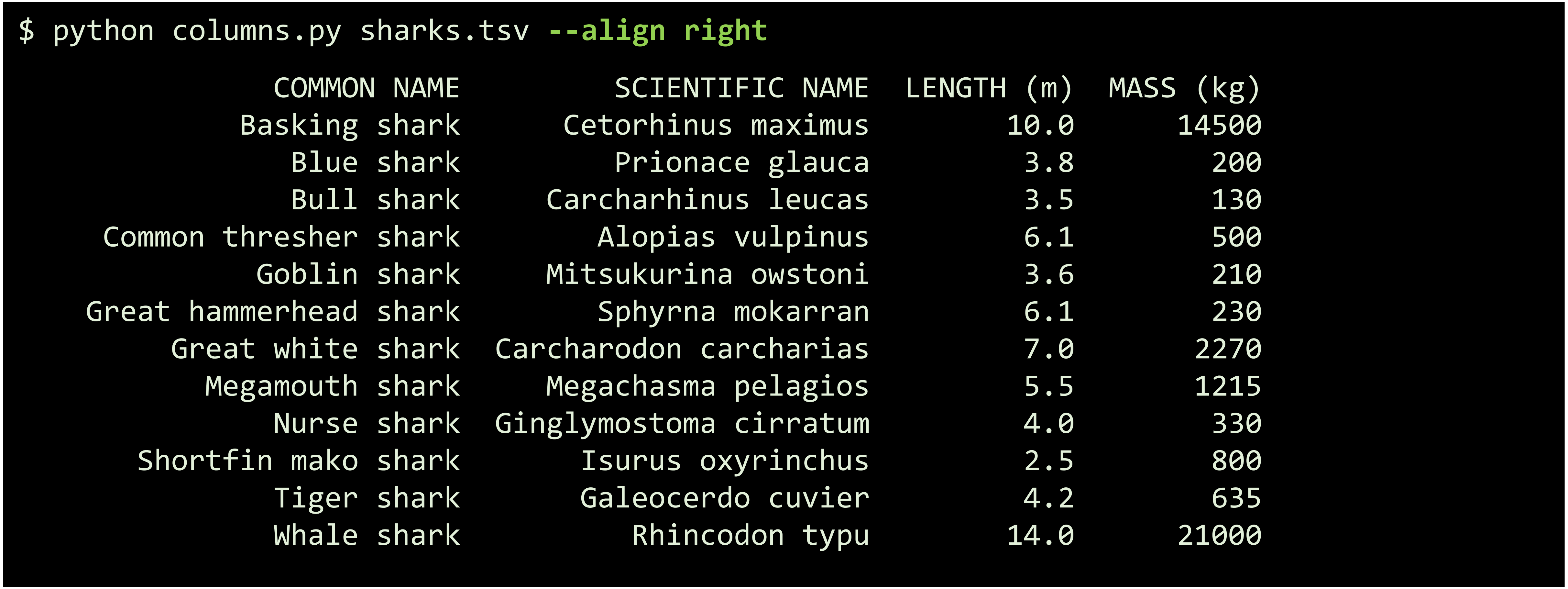
```
$ python columns.py sharks.tsv --padding 7

  COMMON NAME                  SCIENTIFIC NAME              LENGTH (m)       MASS (kg)
  Basking shark                Cetorhinus maximus           10.0             14500
  Blue shark                   Prionace glauca              3.8              200
  Bull shark                   Carcharhinus leucas          3.5              130
  Common thresher shark        Alopias vulpinus             6.1              500
  Goblin shark                 Mitsukurina owstoni          3.6              210
  Great hammerhead shark       Sphyrna mokarran             6.1              230
  Great white shark            Carcharodon carcharias       7.0              2270
  Megamouth shark              Megachasma pelagios          5.5              1215
  Nurse shark                  Ginglymostoma cirratum       4.0              330
  Shortfin mako shark          Isurus oxyrinchus            2.5              800
  Tiger shark                  Galeocerdo cuvier            4.2              635
  Whale shark                  Rhincodon typu               14.0             21000
```

Implement the `--padding` *option to add extra space between columns*
(A very small change to the code)

`columns.py`: “align-right” option

(a terminal)



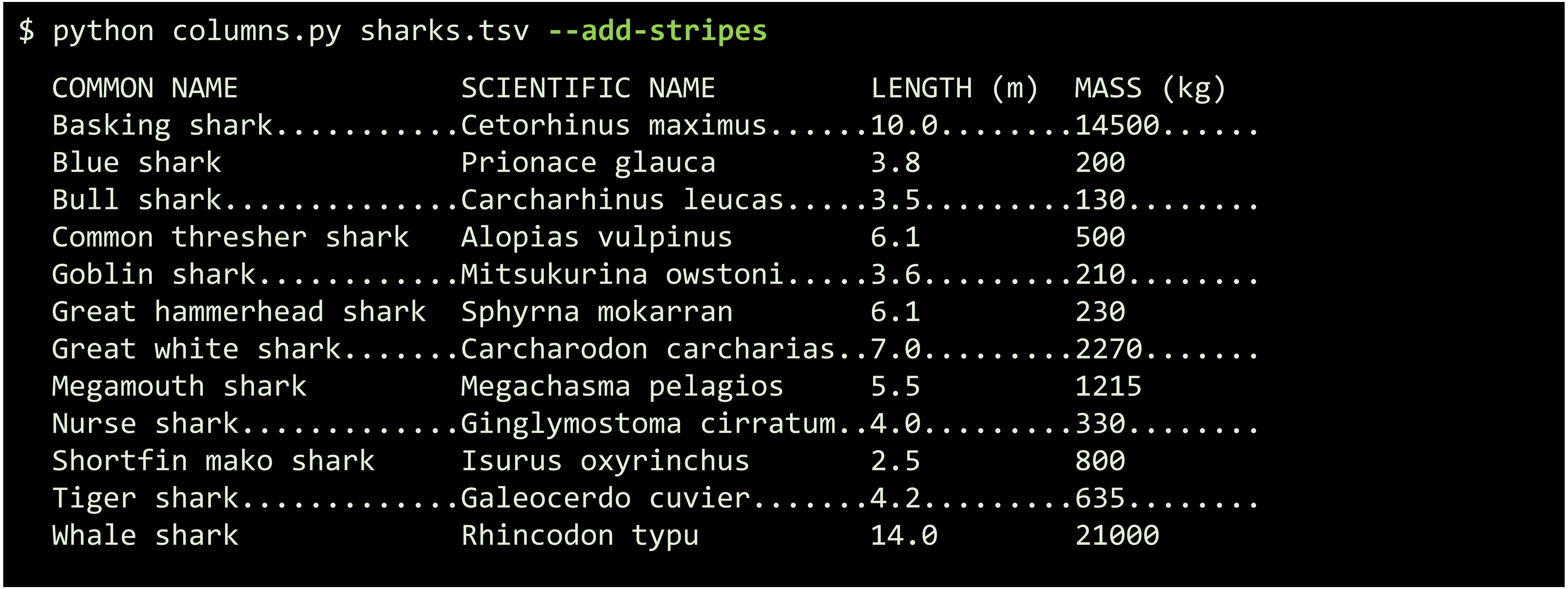
```
$ python columns.py sharks.tsv --align right

               COMMON NAME         SCIENTIFIC NAME  LENGTH (m)  MASS (kg)
             Basking shark      Cetorhinus maximus        10.0      14500
                Blue shark         Prionace glauca         3.8        200
                Bull shark     Carcharhinus leucas         3.5        130
     Common thresher shark        Alopias vulpinus         6.1        500
              Goblin shark     Mitsukurina owstoni         3.6        210
    Great hammerhead shark        Sphyrna mokarran         6.1        230
         Great white shark  Carcharodon carcharias         7.0       2270
           Megamouth shark     Megachasma pelagios         5.5       1215
               Nurse shark  Ginglymostoma cirratum         4.0        330
       Shortfin mako shark       Isurus oxyrinchus         2.5        800
               Tiger shark       Galeocerdo cuvier         4.2        635
               Whale shark          Rhincodon typu        14.0      21000
```

Implement the `--align` *option align right instead of left when requested*
(HINT: this requires the addition of an `if/elif` *block)*

columns.py: “add stripes” option

(a terminal)



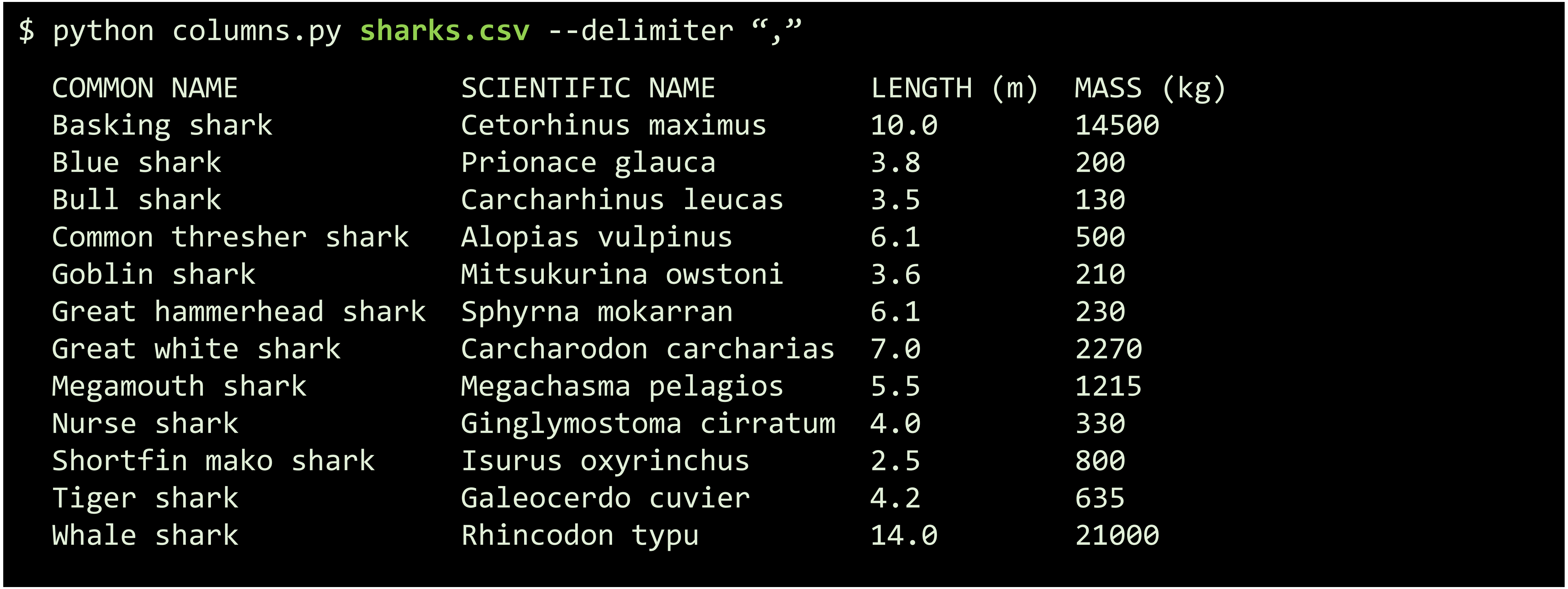
```
$ python columns.py sharks.tsv --add-stripes

  COMMON NAME             SCIENTIFIC NAME         LENGTH (m)  MASS (kg)
  Basking shark...........Cetorhinus maximus......10.0........14500......
  Blue shark              Prionace glauca         3.8         200
  Bull shark..............Carcharhinus leucas.....3.5.........130........
  Common thresher shark   Alopias vulpinus        6.1         500
  Goblin shark............Mitsukurina owstoni.....3.6.........210........
  Great hammerhead shark  Sphyrna mokarran        6.1         230
  Great white shark.......Carcharodon carcharias..7.0.........2270.......
  Megamouth shark         Megachasma pelagios     5.5         1215
  Nurse shark.............Ginglymostoma cirratum..4.0.........330........
  Shortfin mako shark     Isurus oxyrinchus       2.5         800
  Tiger shark.............Galeocerdo cuvier.......4.2.........635........
  Whale shark             Rhincodon typu          14.0        21000
```

Implement the `--add-stripes` option to use dots to pad every-other row
(HINT: count the line number and test if it is even or odd)

columns.py: delimiter option

(a terminal)



```
$ python columns.py sharks.csv --delimiter “,”

  COMMON NAME             SCIENTIFIC NAME         LENGTH (m)  MASS (kg)
  Basking shark           Cetorhinus maximus      10.0        14500
  Blue shark              Prionace glauca         3.8         200
  Bull shark              Carcharhinus leucas     3.5         130
  Common thresher shark   Alopias vulpinus        6.1         500
  Goblin shark            Mitsukurina owstoni     3.6         210
  Great hammerhead shark  Sphyrna mokarran        6.1         230
  Great white shark       Carcharodon carcharias  7.0         2270
  Megamouth shark         Megachasma pelagios     5.5         1215
  Nurse shark             Ginglymostoma cirratum  4.0         330
  Shortfin mako shark     Isurus oxyrinchus       2.5         800
  Tiger shark             Galeocerdo cuvier       4.2         635
  Whale shark             Rhincodon typu          14.0        21000
```

Implement the `--delimiter` option to parse CSV files in addition to TSV

(A very small change to the code)