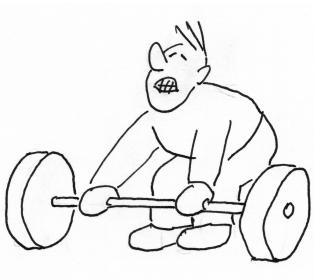
1. Workout Of the Day: Parsing CSV file, creating text table output



Several years ago, I joined a workout group. We meet several times a week in our coach's unpaved driveway. We pick up and drop heavy things and run around trying to keep Death at bay for another day. I'm no paragon of strength and fitness, but it's been a nice way to exercise and visit with friends. One of my favorite parts of going is that our coach will write a "Workout Of (the) Day" or "WOD" on the board. Whatever it says is what I do. It doesn't matter if I actually want to do 200 pushups that day, I just get them done no matter how long it takes.[2]

In that spirit, we'll write a program called `wod.py` to help us create a random daily workout that we have to do, no questions asked:

```
$ ./wod.py
Exercise              Reps
------------------  ------
Pushups                 40
Plank                   38
Situps                  99
Hand-stand pushups       5
```

Each time you run the program, you are required to perform all the exercises *immediately*. Heck, even just *reading* them means you have to do them. Like **NOW**. Sorry, I don't make the rules. Better get going on those situps!

We'll choose from a list of exercises stored in a *delimited text file*. In this case, the "delimiter" is the comma, and it will separate each field value. Data files that use commas as delimiters are often described as "comma-separated values" or CSV files. Usually the first line of the file names the columns, and each subsequent line represents a row in the table:

```
$ head -3 exercises.csv
exercise,reps
Burpees,20-50
Situps,40-100
```

In this exercise, you will:

- Parse delimited text files using the `csv` module
- Coerce text values to numbers
- Print tabular data using the `tabulate` module
- Handle missing and malformed data

This chapter and the next are meant to be a step up in how challenging they are. You will be applying many of the skills you've learned in previous chapters, so get ready!

1.1. Writing `wod.py`

Let's start by taking a look at the usage that should print when run with `-h` or `--help`, Modify your program's parameters until it produces this:

```
$ ./wod.py -h
usage: wod.py [-h] [-f str] [-s int] [-n int] [-e]

Create Workout Of (the) Day (WOD)

optional arguments:
  -h, --help          show this help message and exit
  -f str, --file str  CSV input file of exercises (default: exercises.csv)
  -s int, --seed int  Random seed (default: None)
  -n int, --num int   Number of exercises (default: 4)
  -e, --easy          Halve the reps (default: False)
```

Our program will read an input `-f` or `--file` (default `exercises.csv`) and will produce a `-n` or `--num` of exercises (default `4`). There might be an `-e` or `--easy` flag to indicate that the reps should be cut in half. We'll be using the `random` module to choose the exercises, so we'll need to accept a `-s` or `--seed` option to pass to `random.seed` for testing purposes.

1.1.1. Reading delimited text files

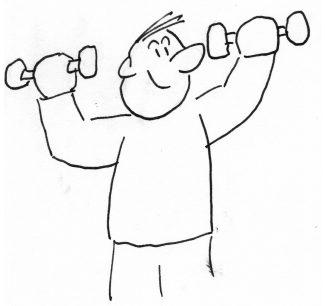
We're going to use three modules that might not be installed on your system already.

1. The `csv` module to parse the input file
2. Tools from the `csvkit` module to look at the input file on the command line
3. The `tabulate` module to format our output table

Run this command to install these modules:

```
$ python3 -m pip install csv csvkit tabulate
```

Despite having "csv" in the name, the `csvkit` can handle just about any delimited text file. For instance, it's typical to use the tab (`\t`) character as a delimiter, too. The module includes many tools that you can read about on their documentation page at https://csvkit.readthedocs.io/en/1.0.3/. After installing `csvkit`, you should be able to use the `csvlook` program to parse the `exercises.csv` file into a table structure showing the columns:



```
$ csvlook --max-rows 3 exercises.csv
| exercise | reps   |
| -------- | ------ |
| Burpees  | 20-50  |
| Situps   | 40-100 |
| Pushups  | 25-75  |
| ...      | ...    |
```

The "reps" column of the input file will have two numbers separated by a dash like `10-20` meaning "from 10 to 20 reps." To select the final value for the "Reps," you will use the `random.randint` function to select an integer value between the low and high values. When run with a seed, your output should exactly match this:

```
$ ./wod.py --seed 1 --num 3
Exercise      Reps
----------  ------
Plank           54
Lunges          35
Crunches        27
```

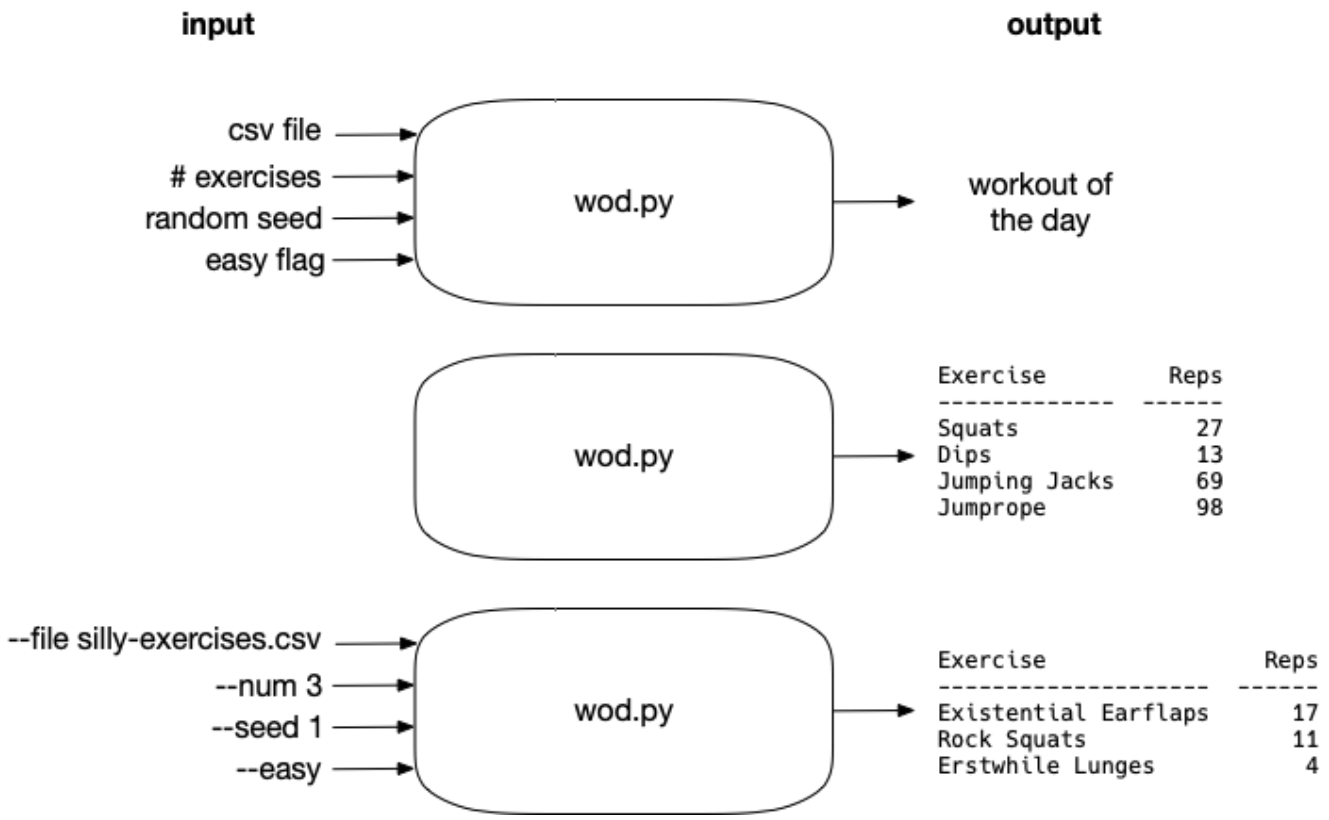
When run with the `--easy` flag, the "reps" should halved:

```
$ ./wod.py --seed 1 --num 3 --easy
Exercise      Reps
----------  ------
Plank           27
Lunges          17
Crunches        13
```

The `--file` option should default to the `exercises.csv` file, or we can indicate different input file:

```
$ ./wod.py --file silly-exercises.csv
Exercise             Reps
-----------------  ------
Hanging Chads          46
Squatting Chinups      46
Rock Squats            38
Red Barchettas         32
```

Here is our trusty string diagram to help you think about it:



1.1.2. Manually reading a CSV file

First I'm going to show you how to manually parse each record from a CSV file into a list of dictionaries, then I'm going to show you how to use the `csv` module to do this more quickly. The reason we want to make a dictionary from each record is so that we can get at the values for each "exercise" and the number of "reps" (repetitions or how many times to repeat a given exercise). We're going to need to split the reps into low and high values so that we can get a range of numbers to randomly select the number of reps. Finally, we're going to randomly select some of the exercises along with their reps to make a workout. Whew, just describing that was a workout!

Notice that the "reps" is given as a range from a low number to a high number, separated by a dash:

```
$ head -3 exercises.csv
exercise,reps
Burpees,20-50
Situps,40-100
```

It would be convenient to read this as a list of dictionaries where each column name in the first line is combined with each line of data like this:

```
$ ./manual1.py
[{'exercise': 'Burpees', 'reps': '20-50'},
 {'exercise': 'Situps', 'reps': '40-100'},
 {'exercise': 'Pushups', 'reps': '25-75'},
 {'exercise': 'Squats', 'reps': '20-50'},
 {'exercise': 'Pullups', 'reps': '10-30'},
 {'exercise': 'Hand-stand pushups', 'reps': '5-20'},
 {'exercise': 'Lunges', 'reps': '20-40'},
 {'exercise': 'Plank', 'reps': '30-60'},
 {'exercise': 'Crunches', 'reps': '20-30'}]
```

It may seem like overkill to use a dictionary for a record that contains just two columns, but I regularly deal with records that contain dozens to *hundreds* of columns! A dictionary is really the only sane way to handle most delimited text files, so it's good to learn using a small example like this.

Let's look at the `manual1.py` code that will do this:

```
 1 #!/usr/bin/env python3
 2
 3 from pprint import pprint                          ①
 4
 5 with open('exercises.csv') as fh:                 ②
 6     headers = fh.readline().rstrip().split(',') ③
 7     records = []                                  ④
 8     for line in fh:                               ⑤
 9         rec = dict(zip(headers, line.rstrip().split(','))) ⑥
10         records.append(rec)                       ⑦
11
12     pprint(records)                               ⑧
```

① We will use the "pretty print" module to print the data structure.

② Use the `with` construct to `open` the exercises as the `fh` variable. One advantage to use `with` is that the file handle will be closed automatically when the code moves beyond the `with` block.

③ Use `fh.readline()` to read only the first line of the file. Remove the whitespace from the right side (`rstrip`), and then `split` the resulting string on commas to create a `list` of strings in `headers`.

④ Initialize `records` as an empty `list`.

⑤ Use a `for` loop to read the rest of the lines of the `fh`.

⑥ Strip and split the `line` of text into a `list` of field values. Use the `zip` function to create a new `list` of tuples containing each of the `headers` with each of the values. Use the `dict` function to turn this list of tuples into a dictionary.

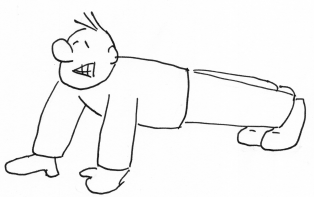
⑦ Append the `rec` dictionary onto the `records`.

⑧ Pretty print the `records`.

Let's break this down a bit more. First we'll `open` the file and read the first line:

```
>>> fh = open('exercises.csv')
>>> fh.readline()
'exercise,reps\n'
```

The line still has a newline stuck to it, so we can use the `rstrip` function to remove that. Note that I need to keep re-opening this file for this demonstration or each subsequent call to `fh.readline()` would read the next line of text:



```
>>> fh = open('exercises.csv')
>>> fh.readline().rstrip()
'exercise,reps'
```

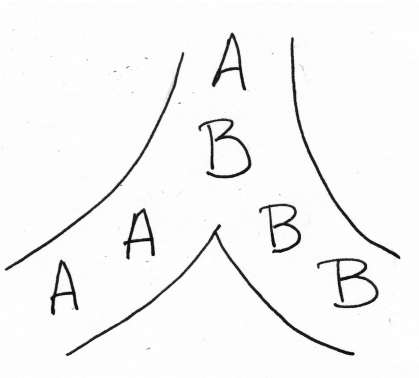
Now let's `split` that line on the comma to get a `list` of strings:

```
>>> fh = open('exercises.csv')
>>> headers = fh.readline().rstrip().split(',')
>>> headers
['exercise', 'reps']
```

We can read the next `line` of the file likewise to get a `list` of the field values:

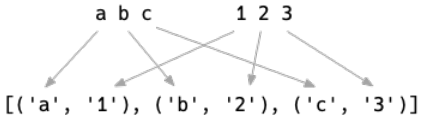
```
>>> line = fh.readline().rstrip().split(',')
>>> line
['Burpees', '20-50']
```

Next I use the `zip` function to merge the two lists into one list. Think of two lines of cars merging to exit a parking lot. It's customary for one car from one lane (say "A") to merge into traffic, then a car from the other lane (say "B"). The cars are combining like the teeth of a zipper, and the result is "A," "B," "A," "B," and so forth.



The `zip` function will group the elements of lists into tuples, grouping all the elements in the first position together, then the second position, etc. Note that this is another *lazy* function, so I will use `list` to coerce this in the REPL:

```
>>> list(zip('abc', '123'))
[('a', '1'), ('b', '2'), ('c', '3')]
```

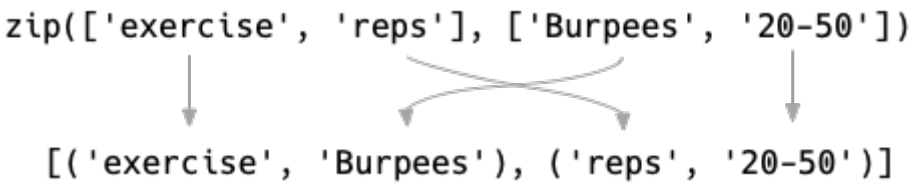


It can work with more than two lists. Note that it will only create groupings for the shortest list. Here, the first two lists have 4 elements ("abcd" and "1234"), but the last has only 3 ("xyz") and so only 3 tuples are created:

```
>>> list(zip('abcd', '1234', 'xyz'))
[('a', '1', 'x'), ('b', '2', 'y'), ('c', '3', 'z')]
```

In our data, `zip` will combines the `headers` and `line` values "exercise" with "Burpees," "reps" with "20-50":

```
>>> list(zip(headers, line))
[('exercise', 'Burpees'), ('reps', '20-50')]
```



That created a `list` of `tuple` values. Instead of `list`, use `dict` to create a dictionary:

```
>>> rec = dict(zip(headers, line))
>>> rec
{'exercise': 'Burpees', 'reps': '20-50'}
```

Remember that the `dict.items()` function will turn a `dict` into a `list` of `tuple` (key/value) pairs, so you can think of these two data structures as being fairly interchangable:

```
>>> rec.items()
dict_items([('exercise', 'Burpees'), ('reps', '20-50')])
```

We can drastically shorten our code by replacing the `for` loop with a list comprehension:

```
with open('exercises.csv') as fh:
    headers = fh.readline().rstrip().split(',')
    records = [dict(zip(headers, line.rstrip().split(','))) for line in fh]
    pprint(records)
```

Or a `map`:

```
with open('exercises.csv') as fh:
    headers = fh.readline().rstrip().split(',')
    records = list(
        map(lambda line: dict(zip(headers,
                                  line.rstrip().split(','))), fh))
    pprint(records)
```

Now we have a really nice data structure (a `list` of `dict` values) that we can use to create our table.

1.1.3. Parsing with the `csv` module

Parsing delimited text files in this way is extremely common. It would not make sense to write or copy this code every time you needed to parse a file. Luckily, some very kind souls have already written some really nice code to do this and have released it as the `csv` module. Let's look at how our code can change if we use the `csv.DictReader` (see the `using_csv1.py` file):

```
#!/usr/bin/env python3

import csv                                      ①
from pprint import pprint

with open('exercises.csv') as fh:
    reader = csv.DictReader(fh, delimiter=',') ②
    records = []                                ③
    for rec in reader:                          ④
        records.append(rec)                     ⑤

    pprint(records)
```

① Import the `csv` module.

② Create a `csv.DictReader` that will create a `dict` for each record in the file that zips the headers in the first line with the data values in the subsequent lines. Use the `delimiter` to indicate the `str` value to `split` each line.

③ Initialize an empty `list` to hold the `records`.

④ Use a `for` loop to iterate through each `rec` (record) in the `reader`.

⑤ The `rec` will be a `dict`. Append that to the `list` of `records`.

This code creates the same `list` of `dict` values as before but with much less code:

```
$ ./using_csv1.py
[OrderedDict([('exercise', 'Burpees'), ('reps', '20-50')]),
 OrderedDict([('exercise', 'Situps'), ('reps', '40-100')]),
 OrderedDict([('exercise', 'Pushups'), ('reps', '25-75')]),
 OrderedDict([('exercise', 'Squats'), ('reps', '20-50')]),
 OrderedDict([('exercise', 'Pullups'), ('reps', '10-30')]),
 OrderedDict([('exercise', 'Hand-stand pushups'), ('reps', '5-20')]),
 OrderedDict([('exercise', 'Lunges'), ('reps', '20-40')]),
 OrderedDict([('exercise', 'Plank'), ('reps', '30-60')]),
 OrderedDict([('exercise', 'Crunches'), ('reps', '20-30')])]
```

We can remove the entire `for` loop and use the `list` function to coerce the `reader` to give us that same `list`. This code will print the same output:

```
1 with open('exercises.csv') as fh:              ①
2     reader = csv.DictReader(fh, delimiter=',') ②
3     records = list(reader)                     ③
4     pprint(records)                            ④
```

① Open the file.

② Create a `csv.DictReader` to read the `fh` using the comma for the `delimiter`.

③ Use the `list` function to coerce all the values from the `reader`.

④ Pretty-print the `records`.

1.1.4. Creating a function to read a CSV file

Let's try to imagine how we could write and test a function we might call `read_csv` to read in our data. I'll start with the `test_read_csv` definition:

```
1 def test_read_csv():
2     text = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,40-100')   ①
3     assert read_csv(text) == [('Burpees', 20, 50), ('Situps', 40, 100)] ②
```

① Use `io.StringIO` to create a mock file handle to wrap around a valid text that we might read from a file. The `\n` represents the newlines that break each line in the input data, and each line uses commas to separate the fields. We previously used `io.StringIO` in the low-memory version of "Howler."

② Affirm that our imaginary `read_csv` file would turn this text into a `list` of `tuple` values with the name of the exercise and the "reps" which have been split into a low and high values. Note that these values have been converted to integers.

Hey, we just did all that work to make a `list` of `dict` values, so why am I suggesting that we now create a `list` of `tuple` values? I'm looking ahead here to how I'll use the `tabulate` module to print out the result, so just trust me here that this is a good way to go!

Let's go back to using the `csv.DictReader` to parse our file and think about how we can break the

"reps" value into `int` values for the low and high:

```
reader = csv.DictReader(fh, delimiter=',')
exercises = []
for rec in reader:
    name, reps = rec['name'], rec['reps']
    low, high = 0, 0 # what goes here?
    exercises.append((name, low, high))
```

You have a couple of tools at your disposal. Imagine your `reps` is this:

```
>>> reps = '20-50'
```

The `str.split` function could break that into two strings, "20" and "50":

```
>>> reps.split('-')
['20', '50']
```

How could you turn each of the `str` values into integers?

Another way you could go is to use a regular expression. Remember that `\d` will match a digit, and so `\d+` will match one or more digits. (Refer back to chapter 15 to refresh your memory on `\d` as a shortcut to the character class of digits.) You can wrap that expression in parentheses to capture the "low" and "high" values:

```
>>> match = re.match('(\d+)-(\d+)', reps)
>>> match.groups()
('20', '50')
```

Can you write a `read_csv` function that passes the above test?

1.1.5. Selecting the exercises

Let's start off by making our `main` print out the data structure from reading an input file, then we'll modify it to actually print our exercise regimem:

```
def main():
    args = get_args()           ①
    random.seed(args.seed)      ②
    pprint(read_csv(args.file)) ③
```

① Get the command-line arguments.

② Set the `random.seed` with the `args.seed` value.

③ Read the `args.file` (which will be an open file handle) using the `read_csv` function and print the

resulting data structure. Note that I've imported the `pprint` function for demonstration purposes.

If you run the above code, you should see this:

```
$ ./wod.py
[('Burpees', 20, 50),
 ('Situps', 40, 100),
 ('Pushups', 25, 75),
 ('Squats', 20, 50),
 ('Pullups', 10, 30),
 ('Hand-stand pushups', 5, 20),
 ('Lunges', 20, 40),
 ('Plank', 30, 60),
 ('Crunches', 20, 30)]
```

We will use the `random.sample` function to select the `--num` of exercises indicated by the user. Add `import random` to your program and modify your `main` to this:

```
1 def main():
2     args = get_args()
3     random.seed(args.seed)
4     pprint(random.sample(read_csv(args.file), k=args.num))
```

Now instead of printing all the exercises, it should print a random sample of the correct number of exercises. In addition, your sampling should exactly match this output because you have set the `random.seed` value:

```
$ ./wod.py -s 1
[('Pushups', 25, 75),
 ('Situps', 40, 100),
 ('Crunches', 20, 30),
 ('Burpees', 20, 50)]
```

We need to iterate through the sample and select a single "reps" value using the `random.randint` function. The first exercise is "Pushups," and the range is between 25 and 75 reps:

```
>>> import random
>>> random.seed(1)
>>> random.randint(25, 75)
33
```

If the `args.easy` is `True`, you will need to halve that value. Unfortunately, we cannot have a fraction of a "rep":

```
>>> 33/2
16.5
```

You can use the `int` function to truncate the number to the integer component:

```
>>> int(33/2)
16
```

1.1.6. Formatting the output

Modify your program until it can reproduce this output:

```
$ ./wod.py -s 1
[('Pushups', 56), ('Situps', 88), ('Crunches', 27), ('Burpees', 35)]
```

We will use the `tabulate` function from the `tabulate` module to format this `list` of `tuple` values into a text table:

```
>>> from tabulate import tabulate
>>> wod = [('Pushups', 56), ('Situps', 88), ('Crunches', 27), ('Burpees', 35)]
>>> print(tabulate(wod))
--------  --
Pushups   56
Situps    88
Crunches  27
Burpees   35
--------  --
```

If you read `help(tabulate)`, you will see that there is a `headers` option where you can give a `list` of strings to use for the headers:

```
>>> print(tabulate(wod, headers=('Exercise', 'Reps')))
Exercise      Reps
----------  ------
Pushups         56
Situps          88
Crunches        27
Burpees         35
```

If you synthesize all these ideas, you should be able to pass the provided tests.

1.1.7. Handling bad data

None of the tests will give your program bad data, but I have provided several "bad" CSV files that

you might be interested in figuring out how to handle:

- The `bad-headers-only.csv` is well-formed but has no data. It only has headers.
- The `bad-empty.csv` file is empty. That is, it is a zero-length file that I created with `touch bad-empty.csv` and has no data at all.
- The `bad-headers.csv` has headers that are capitalized, so "Exercise" instead of "exercise," "Reps" instead of "reps."
- The `bad-delimiter.tab` uses the tab character (`\t`) instead of the comma (`,`) as the field delimiter.
- The `bad-reps.csv` file contains `reps` that are not in format `x-y` or which are not numeric or integer values.

Once your program passes the given tests, trying running it on all the "bad" files to see how your program breaks. What should your program do when there is no usable data? Should your program print error messages when it encounters bad or missing values, or should it quietly ignore errors and only print the usable data? These are all real-world concerns that you will encounter, and it's up to you to decide what your program will do. After the solution, I will show you ways I might deal with these files.

1.1.8. Time to write!

OK, enough lollygagging. Time to write this program. You must do 10 pushups everytime you find a bug!

Hints:

- Use the `csv.DictReader` module's to parse the input CSV files
- Break the `reps` field on the `-` character, coerce the low/high values to `int` values, and then use the `random.randint` module to choose a random integer in that range.
- Use `random.sample` to select the correct number of exercises.
- Use the `tabulate` module to format the WOD.

1.2. Solution

```
#!/usr/bin/env python3
"""Create Workout Of (the) Day (WOD)"""

import argparse
import csv
import io
import random
from tabulate import tabulate                          ①


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Create Workout Of (the) Day (WOD)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-f',
                        '--file',
                        help='CSV input file of exercises',
                        metavar='str',
                        type=argparse.FileType('r'),  ②
                        default='exercises.csv')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    parser.add_argument('-n',
                        '--num',
                        help='Number of exercises',
                        metavar='int',
                        type=int,
                        default=4)

    parser.add_argument('-e',
                        '--easy',
                        help='Halve the reps',
                        action='store_true')

    args = parser.parse_args()

    if args.num < 1:                                   ③
        parser.error(f'--num "{args.num}" must be greater than 0')
```

```
    return args


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    random.seed(args.seed)
    wod = []                                            ④

    for name, low, high in random.sample(read_csv(args.file), k=args.num): ⑤
        reps = random.randint(low, high)                ⑥
        if args.easy:                                   ⑦
            reps = int(reps / 2)
        wod.append((name, reps))                        ⑧

    print(tabulate(wod, headers=('Exercise', 'Reps'))) ⑨


# --------------------------------------------------
def read_csv(fh):                                       ⑩
    """Read the CSV input"""

    exercises = []                                      ⑪
    for row in csv.DictReader(fh, delimiter=','):       ⑫
        low, high = map(int, row['reps'].split('-'))    ⑬
        exercises.append((row['exercise'], low, high)) ⑭

    return exercises                                    ⑮


# --------------------------------------------------
def test_read_csv():                                    ⑯
    """Test read_csv"""

    text = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,40-100')   ⑰
    assert read_csv(text) == [('Burpees', 20, 50), ('Situps', 40, 100)] ⑱


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

① Import the `tabulate` function to format the output table.

② The `--file` option, if provided, must be a file.

③ Ensure that the `args.num` is a positive value.

④ Initialize `wod` as an empty `list`.

⑤ Read the input file and randomly sample the correct number of exercises. The result will be a `list` of tuples that each contain three values which can be unpacked directly into the variables `name`, `low`, and `high`.

⑥ Randomly select a value for the `reps` that is in the provided range.

⑦ If `args.easy` is "truthy," then cut the `reps` in half.

⑧ Append a `tuple` of the name of the exercise and the reps to the `wod`.

⑨ Use the `tabulate` function to format the `wod` into a text table using the appropriate headers.

⑩ Define a function to read an open CSV file handle.

⑪ Initialize `exercises` to an empty `list`.

⑫ Iterate through the file handle using the `csv.DictReader` to create a dictionary combining the column names from the first row with the field values from the rest of the file. Use the comma as the field delimiter.

⑬ Split the "reps" column on the dash, turn those values into integers, and assign to `low` and `high`.

⑭ Append a `tuple` containing the name of the exercise with the low and high values.

⑮ Return the list of `exercises` to the caller.

⑯ Define a function that `pytest` will use to test the `read_csv` function.

⑰ Create a mock file handle containing valid sample data.

⑱ Verify that `read_csv` can handle valid input data.

1.3. Discussion

How did that go for you? Did you manage to modify your program to gracefully handle all the bad input files? Let's dig into the program, starting with the `read_csv` function.

1.3.1. Reading a CSV file

In the introduction, I left you with one line where you needed to split the "reps" column and convert the values to integers. Here is one way:

```
1 def read_csv(fh):
2     exercises = []
3     for row in csv.DictReader(fh, delimiter=','):
4         low, high = map(int, row['reps'].split('-')) ①
5         exercises.append((row['exercise'], low, high))
6
7     return exercises
```

① Split the "reps" field on the dash, `map` the values through `int`, assign to `low` and `high`.

The annotated line works as follows. Assume a "reps" value like so:

```
>>> '20-50'.split('-')
['20', '50']
```

We need to turn each of those into an `int` value, which is what the `int` function will do. We could use a list comprehension:

```
>>> [int(x) for x in '20-50'.split('-')]
[20, 50]
```

But the `map` is much shorter and easier to read, in my opinion:

```
>>> list(map(int, '20-50'.split('-')))
[20, 50]
```

Since that produces exactly two values, we can assign them to two variables:

```
>>> low, high = map(int, '20-50'.split('-'))
>>> low, high
(20, 50)
```

1.3.2. Potentials run-time errors

This code makes many, many assumptions, however, that will cause it to fail miserably when the data doesn't match the expectations. For instance, what happens if the "reps" field contains no dash? It will produce one value:

```
>>> list(map(int, '20'.split('-')))
[20]
```

And that will cause a *run-time* exception when we try to assign one value to two variables:

```
>>> low, high = map(int, '20'.split('-'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 2, got 1)
```

What if one or more of the values cannot be coerced to an `int`? It will cause exception, and, again, you won't discover this until you *run* the program with bad data!

```
>>> list(map(int, 'twenty-thirty'.split('-')))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'twenty'
```

What happens if there is no `reps` field in the record as in the case when the field names are capitalized?

```
>>> rec = {'Exercise': 'Pushups', 'Reps': '20-50'}
```

Then the dictionary access `rec['reps']` will cause an exception:

```
>>> list(map(int, rec['reps'].split('-')))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'reps'
```

While the `read_csv` function seems to work just fine as long as we pass it well-formed data, the real world usually does not always give us clean data sets. An unfortunately large part of my job, in fact, is finding and correcting errors like this.

In the introduction, I suggested you might use a regular expression to extract the low and high values from the "reps" field. A regex has the advantage of inspecting the entire field, ensuring that it looks correct. Here is a more robust way to implement `read_csv`:

```
1  def read_csv(fh):
2      exercises = []                                          ①
3      for row in csv.DictReader(fh, delimiter=','):           ②
4          name, reps = row.get('exercise'), row.get('reps') ③
5          if name and reps:                                   ④
6              match = re.match('(\d+)-(\d+)', reps)           ⑤
7              if match:                                       ⑥
8                  low, high = map(int, match.groups())        ⑦
9                  exercises.append((name, low, high))         ⑧
10
11     return exercises                                        ⑨
```

① Initialize `exercises` as an empty `list`.

② Iterate through the rows of the data.

③ Use the `dict.get` function to try to retrieve the values for "exercise" and "reps."

④ Check if we have "truthy" values for the exercise name and reps.

⑤ Use a regex to look for one or more digits followed by a dash followed by one or more digits. Use capturing parentheses for the digits so they can be extracted.

⑥ Check if there was a `match`. Remember that `re.match` will return `None` to indicate a failure to

match.

⑦ Unpack the `low` and `high` values from the two capture groups, `map` them through the `int` function to coerce the `str` values. This is safe because we use a regex to verify that they look like digits.

⑧ Append the name, low, and high values as a `tuple` to the `exercises`.

⑨ Return the `exercises` to the caller. If no valid data was found, then we will return an empty list.

1.3.3. Formatting the table

Let's look at the `main` I included in the solution and notice a run-time exception waiting to happen:

```
def main():
    args = get_args()
    random.seed(args.seed)
    wod = []

    for name, low, high in random.sample(read_csv(args.file), k=args.num): ①
        reps = random.randint(low, high)
        if args.easy:
            reps = int(reps / 2)
        wod.append((name, reps))

    print(tabulate(wod, headers=('Exercise', 'Reps')))
```

① This line will fail if `read_csv` returns `None` or an empty list.

If you test the given solution with the `bad-headers-only.csv` file, then you would see this error:

```
$ ./wod.py -f bad-headers-only.csv
Traceback (most recent call last):
  File "./wod.py", line 87, in <module>
    main()
  File "./wod.py", line 56, in main
    for name, low, high in random.sample(read_csv(args.file), k=args.num):
  File "/Users/kyclark/anaconda3/lib/python3.7/random.py", line 321, in sample
    raise ValueError("Sample larger than population or is negative")
ValueError: Sample larger than population or is negative
```

A safer way to handle this is to check that `read_csv` returns a usable value. We can print a message that there is no usable data in the input file if it does not:

```
def main():
    args = get_args()
    random.seed(args.seed)
    exercises = read_csv(args.file) ①

    if exercises:                   ②
        for name, low, high in random.sample(exercises, k=args.num): ③
            reps = random.randint(low, high)
            if args.easy:
                reps = int(reps / 2)
            wod.append((name, reps))

        print(tabulate(wod, headers=('Exercise', 'Reps')))
    else:
        print(f'No usable data in --file "{args.file.name}"')
```

① Read the input file into `exercises`. The function should only return a list, possibly empty.

② See if `exercises` is "truthy," that is, a non-empty list.

③ Now that we know there is something to sample, move ahead.

The version in `solution2.py` has these updated functions and gracefully handles all the bad input files.

You may note that the `test_` functions have been moved to the `unit.py` file such that I can run `pytest unit.py`. The `test_csv` function became much longer as I tested with various bad inputs, so it seemed more readable to me to move that code to a separate file:

```
import io
from wod import read_csv


# --------------------------------------------------
def test_read_csv():
    """Test read_csv"""

    good = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,40-100')    ①
    assert read_csv(good) == [('Burpees', 20, 50), ('Situps', 40, 100)]

    no_data = io.StringIO('')                                            ②
    assert read_csv(no_data) == []

    headers_only = io.StringIO('exercise,reps\n')                        ③
    assert read_csv(headers_only) == []

    bad_headers = io.StringIO('Exercise,Reps\nBurpees,20-50\nSitups,40-100')    ④
    assert read_csv(bad_headers) == []

    bad_numbers = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,forty-100') ⑤
    assert read_csv(bad_numbers) == [('Burpees', 20, 50)]

    no_dash = io.StringIO('exercise,reps\nBurpees,20\nSitups,40-100')    ⑥
    assert read_csv(no_dash) == [('Situps', 40, 100)]

    tabs = io.StringIO('exercise\treps\nBurpees\t20-40\nSitups\t40-100') ⑦
    assert read_csv(tabs) == []
```

① The original, valid input.

② Testing with no data at all.

③ Well-formed file (correct headers and delimiter), but no data.

④ The headers are capitalized and only lowercase are expected.

⑤ A string ("forty") that cannot be coerced by `int` to a numeric value.

⑥ A "reps" value ("20") missing a dash.

⑦ Well-formed data with correct headers but using a tab for the delimiter.

1.4. Review

- The `csv` module is useful for parsing text files delimited by commas and tabs.
- Text values representing numbers must be coerce to numeric values using `int` or `float` in order to be used as numbers.
- The `tabulate` module can be used to create text tables to format tabular output.
- Great care must be taken to anticipate and handle bad and missing data values. Tests can help you imagine all the ways in which your code might fail.

1.5. Going Further

- Add an option to use a different delimiter or guess that the delimiter is a tab if the input file extension is ".tab" as in the `bad-delimiter.tab` file.
- The `tabulate` module supports many table formats including plain, simple, grid, pipe, orgtbl, rst, mediawiki, latex, latex_raw and latex_booktabs. Add an option to choose a different `tabulate` format using these as the valid `choices`. Choose a reasonable `default` value.