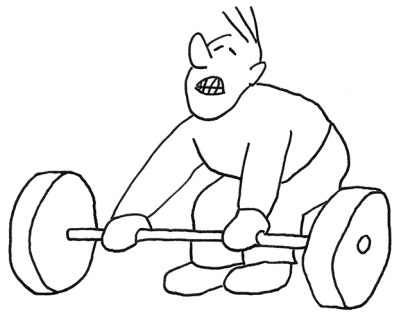
19 Workout of the Day: Parsing CSV files, creating text table output



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 push-ups that day, I just get them done no matter how long it takes.[1]

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
```

[1] See "More Isn't Always Better" by Barry Schwartz (https://hbr.org/2006/06/more-isnt-always-better). He notes that increasing the number of choices given to people actually creates more distress and feelings of dissatisfaction, whatever choice is made. Imagine an ice cream shop with three flavors: chocolate, vanilla, and strawberry. If you choose chocolate, you'll likely be happy with that choice. Now imagine that the shop has 60 flavors of ice cream, including 20 different fruit creams and sorbets and 12 different chocolate varieties from Rocky Road to Fudgetastic Caramel Tiramisu Ripple. Now when you choose a "chocolate" variety, you may leave with remorse about the 11 other kinds you could have chosen. Sometimes having no choice at all provides a sense of calm. Call it fatalism or whatnot.

```
Plank                   38
Situps                  99
Hand-stand pushups       5
```

> **NOTE** 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 sit-ups!

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 inputs/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!

19.1 Writing wod.py

You will be creating a program called wod.py in the 19_wod directory. Let's start by taking a look at the usage that should print when it's run with `-h` or `--help`. Modify your program's parameters until it produces this:

```
$ ./wod.py -h
usage: wod.py [-h] [-f FILE] [-s seed] [-n exercises] [-e]

Create Workout Of (the) Day (WOD)

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

Our program will read an input `-f` or `--file`, which should be a readable text file (default, inputs/exercises.csv). The output will be some `-n` or `--num` number of exercises

(default, `4`). There might be an `-e` or `--easy` flag to indicate that the repetitions of each exercise should be cut in half. Since we'll be using the `random` module to choose the exercises, we'll need to accept an `-s` or `--seed` option (`int` with a default of `None`) to pass to `random.seed()` for testing purposes.

19.1.1 Reading delimited text files

We're going to use the `csv` module to parse the input file. This is a standard module that should already be installed on your system. You can verify that by opening a `python3` REPL and trying to import it. If this works, you're all set:

```
>>> import csv
```

We'll also look at two other modules that you probably will need to install:

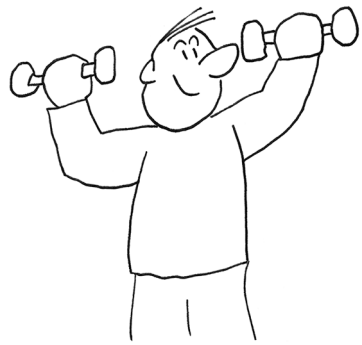
- Tools from the `csvkit` module to look at the input file on the command line
- The `tabulate` module to format the output table

Run this command to install these modules:

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

There is also a requirements.txt file, which is a common way to document the dependencies for a program. Instead of the previous command, you can install all the modules with this one:

```
$ python3 -m pip install -r requirements.txt
```



Despite having "csv" in the name, the `csvkit` module 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 in its documentation (https://csvkit.readthedocs.io/en/1.0.3/). I've included several delimited files in the 19_wod/inputs directory that you can use to test your program.

After installing `csvkit`, you should be able to use `csvlook` to parse the inputs/exercises.csv file into a table structure showing the columns:

```
$ csvlook --max-rows 3 inputs/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
----------  ------
Pushups         32
Situps          71
Crunches        27
```

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

```
$ ./wod.py --seed 1 --num 3 --easy
Exercise      Reps
----------  ------
Pushups         16
Situps          35
Crunches        13
```

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

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

Figure 19.1 shows our trusty string diagram to help you think about it.

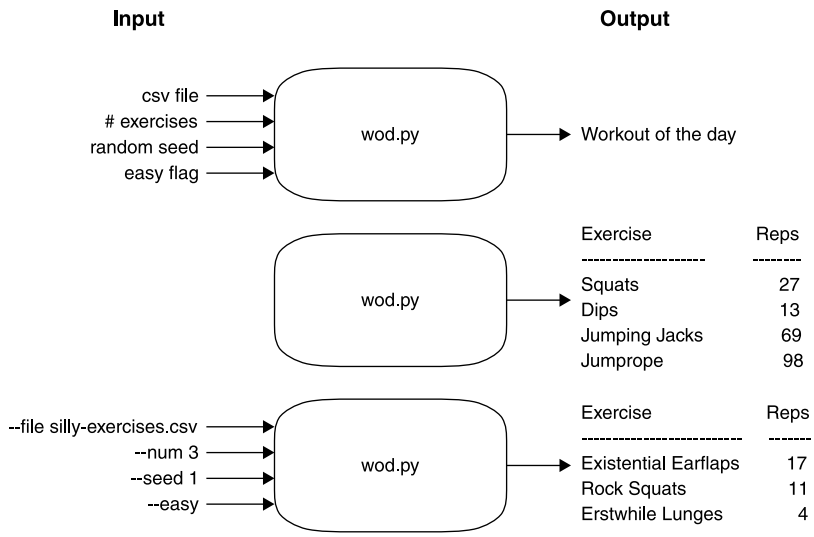


Figure 19.1 The WOD program will randomly select exercises and reps from a CSV file to create a table listing the workout of the day.

19.1.2 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, and then I'll 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 from which we'll randomly select the number of reps. Finally, we'll randomly select some exercises along with their reps to make a workout. Whew, just describing that was a workout!

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

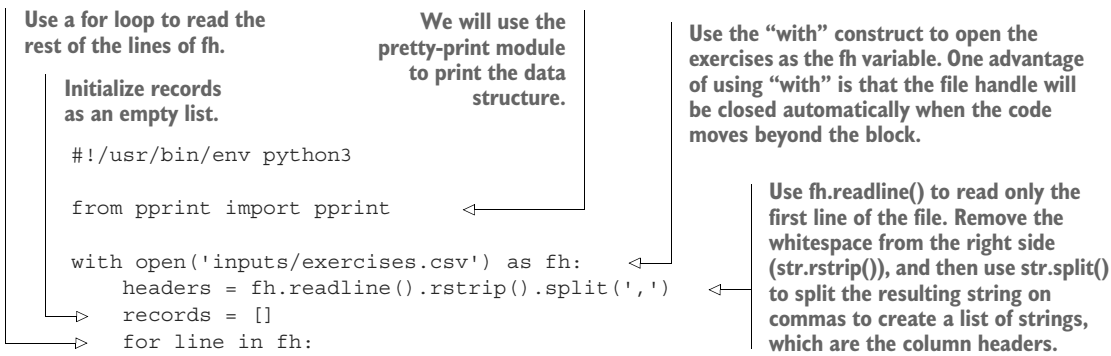
```
$ head -3 inputs/exercises.csv
exercise,reps
Burpees,20-50
Situps,40-100
```

It would be convenient to read this as a list of dictionaries where the column names in the first line are 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 records that contain just two columns, but I regularly deal with records that contain dozens to *hundreds* of columns, and then field names are essential. A dictionary is really the only sane way to handle most delimited text files, so it's good to learn with a small example like this.

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



```
        rec = dict(zip(headers, line.rstrip().split(',')))
        records.append(rec)

pprint(records)
```

Append the resulting dictionary to the records.

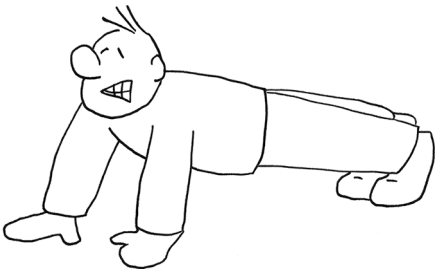
Pretty-print the records.

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 paired with each of the values. Use the dict() function to turn this list of tuples into a dictionary.

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 `str.rstrip()` function to remove that:



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

NOTE Note that I need to keep reopening this file for this demonstration, or each subsequent call to `fh.readline()` would read the next line of text.

Now let's use `str.split()` to 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 likewise read the next `line` of the file to get a `list` of the field values:

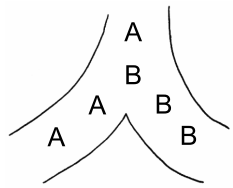
```
>>> line = fh.readline().rstrip().split(',')
>>> line
['Burpees', '20-50']
```

Next we use the `zip()` function to merge the two lists into one list where the elements of each list have been mated with their counterparts in the same positions. That might seem complicated, but think about the end of a wedding ceremony when the bride and groom turn around to face the assembled crowd. Usually they will hold hands and start walking down the aisle to leave the ceremony. Imagine three groomsmen (`'G'`) and three bridesmaids (`'B'`) left standing on their respective sides facing each other:

```
>>> groomsmen = 'G' * 3
>>> bridesmaids = 'B' * 3
```

If there are two lines each containing three people, then we end up with a single line containing three pairs:

```
>>> pairs = list(zip(groomsmen, bridesmaids))
>>> pairs
[('G', 'B'), ('G', 'B'), ('G', 'B')]
>>> len(pairs)
3
```



Or 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 the lists into tuples, grouping all the elements in the first position together, then the second position, and so on, as shown in figure 19.2. 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')]
```

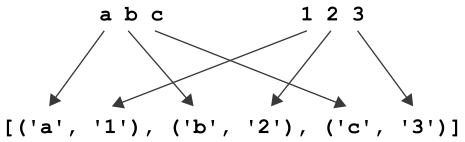


Figure 19.2 Zipping two lists creates a new list with pairs of elements.

The `zip()` function can handle more than two lists. Note that it will only create groupings for the shortest list. In the following example, the first two lists have four elements ("abcd" and "1234"), but the last has only three ("xyz"), so only three tuples are created:

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

In our data, `zip()` will combine the header "exercise" with the value "Burpees" and then the header "reps" with the value "20–50" (see figure 19.3):

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

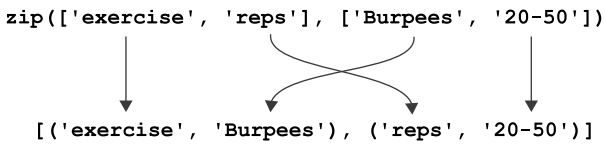


Figure 19.3 Zipping the headers and values together to create a list of tuples

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

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

Recall 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 interchangeable:

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

We still need to break out the headers separately by reading the first line.

This combines the three lines of the for loop into a single list comprehension.

We can use `map()` to write equivalent code:

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

Flake8 will complain about assigning this lambda expression. I generally write my code so as to produce no warnings, but I do tend to disagree with this suggestion. I quite like writing one-line functions using a lambda assignment.

In the next section, I'm going to show you how to use the `csv` module to handle much of this code, which may lead you to wonder why I bothered showing you how to handle this yourself. Unfortunately, I often have to handle data that is terribly formatted, such that the first line is not the header, or there are other rows of information between the header row and the actual data. When you've seen as many badly formatted Excel files as I have, you'll come to appreciate that you sometimes have no choice but to parse the file yourself.

19.1.3 Parsing with the csv module

Parsing delimited text files in this way is extremely common, and it would not make sense to write or copy this code every time you needed to parse a file. Luckily, the `csv` module is a standard module installed with Python, and it can handle all of this very gracefully.

Let's look at how our code can change if we use `csv.DictReader()` (see using_csv1.py in the repo):

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

import csv
from pprint import pprint

with open('inputs/exercises.csv') as fh:
    reader = csv.DictReader(fh, delimiter=',')
    records = []
    for rec in reader:
        records.append(rec)

    pprint(records)
```

Import the csv module.

Create a csv.DictReader() that will create a dictionary for each record in the file. It zips the headers in the first line with the data values in the subsequent lines. It uses the delimiter to indicate the string value for splitting the columns of text.

Initialize an empty list to hold the records.

Use a for loop to iterate through each record returned by the reader.

The records will be a dictionary that is appended to the list of records.

The following code creates the same `list` of `dict` values as before, but with far less code. Note that each record is shown as an `OrderedDict`, which is a type of dictionary where the keys are maintained in their insertion order:

```
$ ./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 (in using_csv2.py) will print the same output:

```
with open('inputs/exercises.csv') as fh:
    reader = csv.DictReader(fh, delimiter=',')
    records = list(reader)
    pprint(records)
```

Open the file.

Create a csv.DictReader() to read fh, using the comma for the delimiter.

Use the list() function to coerce all the values from the reader.

Pretty-print the records.

19.1.4 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. Let's start with a placeholder for our function and the `test_read_csv()` definition:

```
def read_csv(fh):
    """Read the CSV input"""
    pass

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

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 chapter 5's program.

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 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 we might use the `tabulate` module to print out the result, so just trust me here. This is a good way to go!

Let's go back to using `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['exercise'], rec['reps']
    low, high = 0, 0 # what goes here?
    exercises.append((name, low, high))
```

You have a couple of tools at your disposal. Imagine `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?

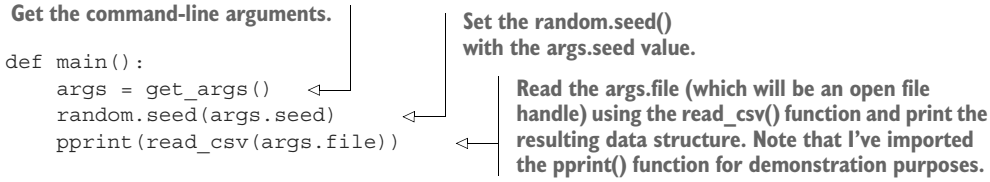
Another way you could go is to use a regular expression. Remember that `\d` will match a digit, so `\d+` will match one or more digits. (Refer back to chapter 15 to refresh your memory on `\d` as a shortcut to the character class of digits.) You can wrap that expression in parentheses to capture the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

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

Can you write a `read_csv()` function that passes the previous `test_read_csv()`?

19.1.5 Selecting the exercises

By this point, I'm hoping you've got `get_args()` straight and your `read_csv()` passes the given test. Now we can start in `main()` with printing out the data structure:

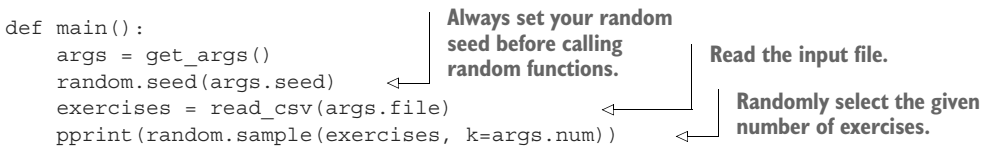



```
def main():
    args = get_args()
    random.seed(args.seed)
    pprint(read_csv(args.file))
```

If you run the preceding 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 match this:




```
def main():
    args = get_args()
    random.seed(args.seed)
    exercises = read_csv(args.file)
    pprint(random.sample(exercises, 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 if you 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 push-ups, and the range is between 25 and 75 reps:

```
>>> import random
>>> random.seed(1)
```

```
>>> random.randint(25, 75)
33
```

If `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
```

19.1.6 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 specify 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.

19.1.7 Handling bad data

None of the tests will give your program bad data, but I have provided several "bad" CSV files in the 19_wod/inputs directory that you might be interested in figuring out how to handle:

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

Once your program passes the given tests, trying running it on 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.

19.1.8 Time to write

OK, enough lollygagging. Time to write this program. You must do 10 push-ups every time you find a bug!

Here are a few hints:

- Use `csv.DictReader()` to parse the input CSV files.
- Break the `reps` field on the `-` character, coerce the low/high values to `int` values, and then use `random.randint()` 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 output into a text table.

19.2 Solution

How did that go for you? Did you manage to modify your program to gracefully handle all the bad input files?

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

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


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

Import the tabulate function we will use to format the output table.

```
    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='FILE',
                        type=argparse.FileType('rt'),
                        default='exercises.csv')

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

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

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

    args = parser.parse_args()

    if args.num < 1:
        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 = []
    exercises = read_csv(args.file)

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

The --file option, if provided, must be a readable text file.

Ensure that args.num is a positive value.

Randomly sample the given 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 and low and high values.

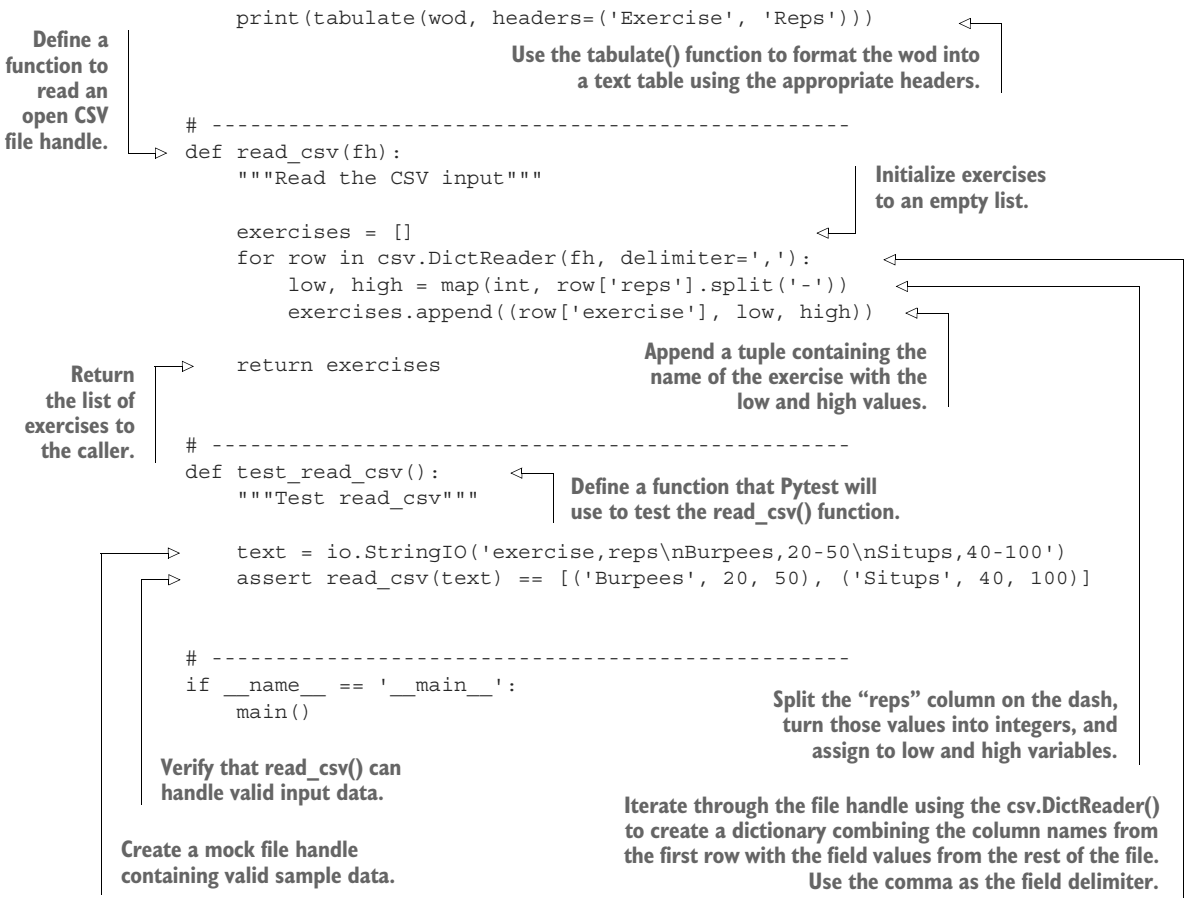
Initialize wod as an empty list.

Read the input file into a list of exercises.

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

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

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



19.3 Discussion

Almost half the lines of the program are found within the `get_args()` function! Even though there's nothing new to discuss, I really want to point out how much work is being done to validate the inputs, provide defaults, create the usage statement, and so forth. Let's dig into the program, starting with the `read_csv()` function.

19.3.1 Reading a CSV file

Earlier in the chapter, I left you with one line where you needed to split the reps column and convert the values to integers. Here is one way:

```
def read_csv(fh):
    exercises = []
    for row in csv.DictReader(fh, delimiter=','):
        low, high = map(int, row['reps'].split('-'))
        exercises.append((row['exercise'], low, high))

    return exercises
```

Split the reps field on the dash, map the values through the int() function, and 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)
```

19.3.2 Potential runtime errors

This code makes many, many assumptions 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]
```

That will cause a *runtime* 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 an 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 is the case when the field names are capitalized?

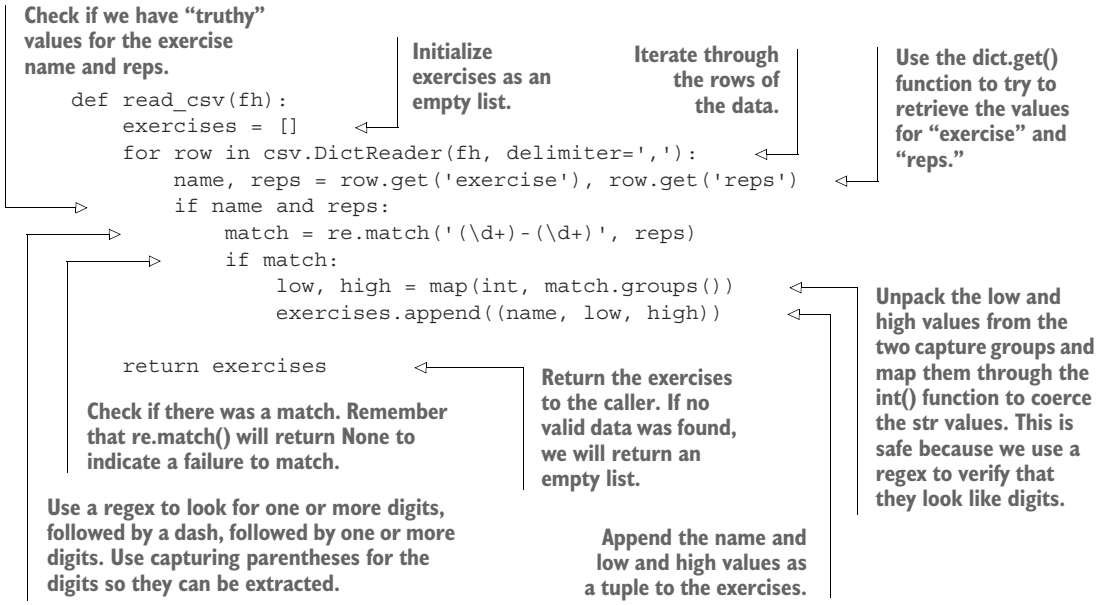
```
>>> rec = {'Exercise': 'Pushups', 'Reps': '20-50'}
```

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'
```

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

Earlier in the chapter, 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()`:



19.3.3 *Using pandas.read_csv() to parse the file*

Many people familiar with statistics and data science will likely know the Python module called `pandas`, which mimics many ideas from the R programming language. I specifically chose the function name `read_csv()` because this is similar to a built-in function in R called `read.csv`, which was in turn used as the model for the `pandas.read_csv()` function. Both R and `pandas` tend to think of the data in delimited/CSV files in terms of a "data frame"—a two-dimensional object that allows you to deal with columns and rows of data.

To run the using_pandas.py version, you'll need to install `pandas` like so:

```
$ python3 -m pip install pandas
```

Now you can try running this program:

```
import pandas as pd

df = pd.read_csv('inputs/exercises.csv')
print(df)
```

You'll see this output:

```
$ ./using_pandas.py
             exercise    reps
0             Burpees   20-50
1              Situps  40-100
2             Pushups   25-75
3              Squats   20-50
4             Pullups   10-30
5  Hand-stand pushups    5-20
6              Lunges   20-40
7               Plank   30-60
8            Crunches   20-30
```

Learning how to use `pandas` is far beyond the scope of this book. Mostly I just want you to be aware that this is a very popular way to parse delimited text files, especially if you intend to run statistical analyses over various columns of the data.

19.3.4 Formatting the table

Let's look at the `main()` function I included in the solution. You may notice a runtime exception waiting to happen:

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

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

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

This line will fail if args.num is greater than the number of elements in exercises, such as if read_csv() returns None or an empty list.

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

```
$ ./wod.py -f inputs/bad-headers-only.csv
Traceback (most recent call last):
  File "./wod.py", line 93, in <module>
    main()
  File "./wod.py", line 62, in main
    for name, low, high in random.sample(exercises, k=args.num):
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/rando
     m.py", line 363, 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 enough data to pass to `random.sample()`. We have a couple of possible errors:

- No usable data was found in the input file.
- We are trying to sample too many records from the file.

Here is a possible way to handle these problems. Remember that calling `sys.exit()` with a string value will cause the program to print the message to `sys.stderr` and exit with a value of `1` (which is an error value):

```
def main():
    """Make a jazz noise here"""

    args = get_args()
    random.seed(args.seed)
    exercises = read_csv(args.file)

    if not exercises:
        sys.exit(f'No usable data in --file "{args.file.name}"')

    num_exercises = len(exercises)
    if args.num > num_exercises:
        sys.exit(f'--num "{args.num}" > exercises "{num_exercises}"')

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

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

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

Check if exercises is "falsey," such as an empty list.

Check if we are trying to sample too many records.

Continue after we verify that we have enough valid data.

The version in solution2.py has these updated functions and gracefully handles all the bad input files. Note that I moved the `test_read_csv()` function to the unit.py file because it became much longer as I tested with various bad inputs.

You can run `pytest -xv unit.py` to run the unit tests. Let's inspect unit.py to see a more rigorous testing scheme:

```
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')
    assert read_csv(good) == [('Burpees', 20, 50), ('Situps', 40, 100)]
```

Remember that you can import your own functions from your own modules into other programs. Here we are bringing in our read_csv() function. If we had instead used import wod, we could call wod.read_csv().

The original, valid input

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

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

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

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

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

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

Testing with no data at all

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

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

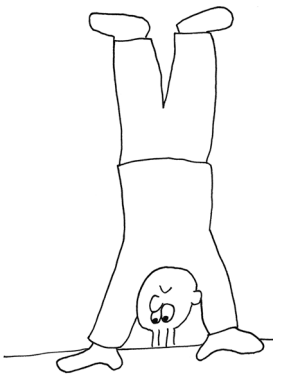
The headers are capitalized, but only lowercase headers are expected.

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

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

19.4 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.



Summary

- The `csv` module is useful for parsing delimited text data such as CSV and tab-delimited files.
- Text values representing numbers must be coerced to numeric values using `int()` or `float()` in order to be used as numbers inside your program.
- 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.