Hamming Distance

The Hamming Distance is the number of edits you need to make in "string1" to change it to "string2," which here would be 1. It's named for Richard Hamming (https://en.wikipedia.org/wiki/Richard_Hamming) who was a brilliant mathmetician and early computer programmer. Along the way, we'll be solving http://rosalind.info/problems/hamm/.

You will need to write a Python program called `hamm.py` which will be given two files with the same number of "words" which each may differ from each other. (You do not have to worry about the files having different numbers of words. You only have to compare the first words in each file, then the seconds words, and so on.) You will need to count all the Hamming distances between all the words and print a sum. If you are given the same files as both arguments, the answer will be "0." The "american.txt" and "british.txt" files differ by 28 edits, e.g., "behavior" vs "behaviour."

This will be the first `test.py` that will `import` your code in order to specifically test a function that is expected to be called `dist` which should take two strings and return an integer value which is the Hamming distance. It should be defined like so:

```
def dist(s1, s2):
```

That is, the function must be called `dist` and should take two arguments but they do not have to be called `s1` and `s2`. Given `foo` and `faa`, the function should return `2`. Given `foo` and `fool`, it should return 1. Given `foo` and `foo`, it should return 0. This assignment does not require alignment, so you can assume that the strings begin the same and may differ only in length and content.

NB: You should probably most definitely use `new_py.py -a` to start this one as it will create a file that can be `import`ed into the test suite without problems.

The two file inputs should be specified as positional arguments with `nargs=2` to indicate exactly two arguments. The program can take an optional *flag* called `-d|--debug` which turns on low-level debugging statements. Without this flag, only *critical* errors will be logged. The program will use the `logging` module to log debugging statements to the file `.log` (so it will be hidden). Here is the code to make that happen (put this into `main`):

```
logging.basicConfig(
    filename='.log',
    filemode='w',
    level=logging.DEBUG if args.debug else logging.CRITICAL
)
```

If a file argument is not a valid file, die with the message ' "XXX" is not a file.'

Expected Behavior

```
$ ./hamm.py
usage: hamm.py [-h] [-d] FILE FILE
hamm.py: error: the following arguments are required: FILE

$ ./hamm.py -h
usage: hamm.py [-h] [-d] FILE FILE

Hamming distance

positional arguments:
  FILE         File inputs

optional arguments:
  -h, --help   show this help message and exit
  -d, --debug  Debug (default: False)

$ ./hamm.py fox.txt
usage: hamm.py [-h] [-d] FILE FILE
hamm.py: error: the following arguments are required: FILE

$ ./hamm.py fox.txt foo
"foo" is not a file

$ ./hamm.py fox.txt fox.txt
0

$ ./hamm.py american.txt british.txt
28

$ ./hamm.py sample1.fa sample2.fa
6
```

Note in the last case that the inputs are sequences.

```
$ cat sample1.fa
AAATAAA
AAATTTT
TTTTCCC
AAATCCC
GGGTGGG
$ cat sample2.fa
AAATAAC
ACATTTT
TTTTGCC
```

```
TAATCCC
GGGTGTT
```

The Hamming distance is synonymous with single-nucleotide polymorphisms/variations (SNPs or SNVs).

Logging

An important aspect of this assignment is for you to learn how to use the `logging` module. To that end, the test suite will be looking for *debug* messages for the names of the two files given as input to the program and additionally a message in the `dist` function that report on the two words given as arguments and their Hamming distance, e.g.:

```
$ ./hamm.py american.txt british.txt -d
28
$ head .log
DEBUG:root:file1 = american.txt, file2 = british.txt
DEBUG:root:s1 = I, s2 = I, d = 0
DEBUG:root:s1 = went, s2 = went, d = 0
DEBUG:root:s1 = to, s2 = to, d = 0
DEBUG:root:s1 = the, s2 = the, d = 0
DEBUG:root:s1 = theater, s2 = theatre, d = 2
DEBUG:root:s1 = last, s2 = last, d = 0
DEBUG:root:s1 = night, s2 = night, d = 0
DEBUG:root:s1 = with, s2 = with, d = 0
DEBUG:root:s1 = my, s2 = my, d = 0
```

Test Suite

A passing test suite looks like this:

```
$ make test
python3 -m pytest -v test.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/worked_examples/hamm, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 5 items

test.py::test_usage PASSED                                               [ 20%]
test.py::test_bad_input PASSED                                           [ 40%]
test.py::test_dist PASSED                                                [ 60%]
```

```
test.py::test_runs_ok PASSED                                             [ 80%]
test.py::test_log PASSED                                                 [100%]

========================== 5 passed in 0.67 seconds ==========================
```

A note about the testing: If you were to put this function directly into your `hamm.py`:

```
def test_dist():
    """dist ok"""
    tests = [('foo', 'boo', 1), ('foo', 'faa', 2), ('foo', 'foobar', 3),
             ('TAGGGCAATCATCCGAG', 'ACCGTCAGTAATGCTAC',
              9), ('TAGGGCAATCATCCGG', 'ACCGTCAGTAATGCTAC', 10)]

    for s1, s2, n in tests:
        d = dist(s1, s2)
        assert d == n
```

Then you could run PyTest directly on the program:

```
$ pytest -v hamm.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/worked_examples/hamm, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 1 item

hamm.py::test_dist PASSED                                                [100%]

========================== 1 passed in 0.02 seconds ==========================
```

You can write as many `test_*` functions into your programs as you like to ensure you functions do what they are supposed to do, then run `pytest` on the program to verify that changes to the program have not accidentally introduced bugs in code that previously worked (cf. "unit tests" and "regression tests").