Creating synthetic DNA/RNA sequences

In this exercise, you will write a Python program called `moog.py` [1] that will generate a FASTA-formatted [2] file of synthetic DNA or RNA. The program will accept the following optional arguments:

- `-o|--outfile`: The output file to write the sequences (default "out.fa")
- `-t|--seqtype`: The sequence type, either `dna` or `rna` (`str`, default "dna")
- `-n|--numseqs`: The number of sequences to generate (`int`, default 10)
- `-m|--minlen`: The minimum length for any sequence (`int`, default 50)
- `-x|--maxlen`: The maximum length for any sequence (`int`, default 75)
- `-p|--pctgc`: The average percentage of GC content for a sequence (`float`, default 0.5 or 50%)
- `-s|--seed`: An integer value to use for the random seed (`int`, default `None`) so that the random choices of the program can be repeated under testing conditions.

Here is the usage the program should generate:

```
$ ./moog.py -h
usage: moog.py [-h] [-o str] [-t str] [-n int] [-m int] [-x int] [-p float]
               [-s int]

Create synthetic sequences

optional arguments:
  -h, --help            show this help message and exit
  -o str, --outfile str
                        Output filename (default: out.fa)
  -t str, --seqtype str
                        DNA or RNA (default: dna)
  -n int, --numseqs int
                        Number of sequences to create (default: 10)
  -m int, --minlen int  Minimum length (default: 50)
  -x int, --maxlen int  Maximum length (default: 75)
  -p float, --pctgc float
                        Percent GC (default: 0.5)
  -s int, --seed int    Random seed (default: None)
```

For instance, I can run it to create 3 sequences with the default values, and the program will tell me how many of what kind of sequences were placed into which output file:

```
$ ./moog.py -n 3 -s 1
Done, wrote 3 DNA sequences to "out.fa".
```

The output file should be in FASTA format:

- Each sequence record takes up two lines
- The first line for each sequence record starts with a literal `>` (greater than sign) and is followed by a unique identifier. For this exercise, the ID is not important so numbering the sequences in order is sufficient.
- The second line of a record is the sequence itself. Note that some FASTA formats will limit the length of this line and so may break up the sequence over several lines. This is not necessary. The sequence can be one very long line. If you really want to break the sequence after something like 80 characters, that is fine, too.

Here is what the output for the above should (might) look like:

```
$ cat out.fa
>1
ATTTGCATAGGAGCAGGACAAAGGGCTCGACTCTTCCGCGCCATGTTGTATCAGAACA
>2
CCCTTGATCGGCCCGGGGGTACGCATACCGTACAAGCTGGTTAATTACTAAAAATTACTGAAACGGAATGC
>3
TTCTGTGGGAGTCAGAGACCTATGAAGATTCTAATAGCAGACGCCAAGATCCGCAGCACAT
```

Writing `moog.py`

The first challenge is to define all your arguments correctly. Onerous as this is, perhaps 30% of the program is in accepting the arguments correctly!

Some tips:

- Be sure to use `choices` for the `seqtype` argument
- Consider using `type=argparse.FileType('wt')` for the `--outfile`. You don’t have to do this, but, if you do, then `args.outfile` will be an open, writable file handle!
- You should also manually verify that `pctgc` is between 0 and 1 which can be done with a compound comparison like so:

```
args = parser.parse_args()

if not 0 < args.pctgc < 1:
    parser.error(f'--pctgc "{args.pctgc}" must be between 0 and 1')
```

You should be sure to set the random seed immediately after accepting the arguments, *before* you do anything using the `random` module. Then your program will need to get a "pool" of bases for creating the sequences:

```
def main():
    args = get_args()
    random.seed(args.seed)
    pool = create_pool(args.pctgc, args.maxlen, args.seqtype)
```

You can use the following `create_pool` function. The function accepts three positional arguments:

1. `pctgc`: The average percentage of GC content for each sequence
2. `max_len`: The maximum length for any sequence
3. `seq_type`: The sequence type, either "rna" or "dna"

```
def create_pool(pctgc, max_len, seq_type):
    """ Create the pool of bases """

    t_or_u = 'T' if seq_type == 'dna' else 'U' ①
    num_gc = int((pctgc / 2) * max_len)        ②
    num_at = int(((1 - pctgc) / 2) * max_len)  ③
    pool = 'A' * num_at + 'C' * num_gc + 'G' * num_gc + t_or_u * num_at ④

    for _ in range(max_len - len(pool)):       ⑤
        pool += random.choice(pool)

    return ''.join(sorted(pool))               ⑥
```

① Choose either "T" if `seq_type` is "dna" or choose "U" for "rna"

② The number of G or C bases is the `pctgc` divided by 2 times the number of bases.

③ The number of A or T/U bases is the `1 - pctgc` divided by 2 times the number of bases.

④ The `pool` of bases will be each base in "ACG[TU]" repeated the correct number of times.

⑤ Because of rounding issues, we may not actually have enough bases, so pad the pool with random choices from the existing pool (in the hopes this essentially keeps the GC content the same).

⑥ Return a sorted string of the bases.

Here is the test for the function. Notice how the test also gives us a very clear understanding of how we'll pass in and receive values:

```
def test_create_pool():
    """ Test create_pool """

    state = random.getstate() ①
    random.seed(1)            ②
    assert create_pool(.5, 10, 'dna') == 'AAACCCGGTT'
    assert create_pool(.6, 11, 'rna') == 'AACCCCGGGUU'
    assert create_pool(.7, 12, 'dna') == 'ACCCCCGGGGGT'
    assert create_pool(.7, 20, 'rna') == 'AAACCCCCCCGGGGGGGUUU'
    assert create_pool(.4, 15, 'dna') == 'AAAACCCGGGTTTTT'
    random.setstate(state)   ③
```

① The state of the `random` module is *global* to the program. Any change we make here could affect unknown parts of the program, so we save our current state.

② Set the random seed to a known value. This is a *global change* to our program. Any other calls to `random` after this line are affected, even if they are in another function or module!

③ Reset the global state to the original value.

With the above functions, your program is essentially left to fill in this:

```
def main():
    args = get_args()
    random.seed(args.seed)
    pool = create_pool(args.pctgc, args.maxlen, args.seqtype)

    for ...:                    ①
        seq_len = ...           ②
        seq = ...               ③
        args.outfile.write(...)) ④

    print(...)                  ⑤
```

① You need to do this `args.numseqs` times

② Use `random.randint` to select a number between `args.minlen` and `arg.maxlen`

③ Use `random.sample` to select `seq_len` number of bases from the `pool` and make a new `str` for your sequence.

④ Write the new `seq` to the output file in the FASTA format.

⑤ Print the output message.

Using the `random` functions

As noted in the `abuse` chapter, we can use `random.seed` to control the pseudo-random generator in Python. This allows us to test that our random functions are *reproducible*! You should set this value before calling any functions in the `random` module. The default value for your `--seed` parameter should be `None`. If you set the seed to `None`, it is the same as not setting it at all. The seed can be a `str` or an int, but stick with using an int for this exercise.

For each sequence, you will use random.randint to select a length for the sequence between the min/max values:

```
>>> import random
>>> min_len = 5
>>> max_len = 15
>>> seq_len = random.randint(min_len, max_len)
>>> seq_len
12
```

You will then use this value to select the bases for your new sequence:

```
>>> random.sample(pool, seq_len)
['A', 'T', 'A', 'T', 'C', 'C', 'G', 'C', 'G', 'A', 'G', 'T']
```

The output should be written to the output file like so (assuming this is the first sequence):

```
>1
ATATCCGCGAGT
```

Testing

You may need to install BioPython and numpy in order to run the tests:

```
$ python3 -m pip install biopython numpy
```

I would recommend you study the test.py as it uses the BioPython module to parse the output file your program creates so as to check:

- that the output file is parsable as FASTA format
- that the output file has the correct number of sequences
- that the sequences lie in the range of min/max lengths
- that the bases are correct for the given sequence type
- that the average GC content of the sequences is close to the value indicated

Note that BioPython will emit a deprecation warning under pytest, so I have added an additional flag --disable-pytest-warnings to the make test target that you should use:

```
$ make test
pytest -xv --disable-pytest-warnings test.py
============================= test session starts ==============================
...
collected 6 items

test.py::test_exists PASSED                                              [ 16%]
test.py::test_usage PASSED                                               [ 33%]
test.py::test_bad_seqtype PASSED                                         [ 50%]
test.py::test_bad_pctgc PASSED                                           [ 66%]
test.py::test_defaults PASSED                                            [ 83%]
test.py::test_options PASSED                                             [100%]

========================= 6 passed, 1 warning in 0.87s =========================
```

[1] Why "moog"?

[2] https://en.wikipedia.org/wiki/FASTA_format