# GC content

Coding regions of DNA often have a greater proportion of G and C bases. We can use the relative frequency of these bases to identify short sequences that might lie in coding regions.

For this exercise, we'll assume we're reading sequences that occur separately on each line of an input text file. Note that bases may be in lower- or uppercase:

```
$ cat inputs/sample1.txt
ACgt
```

There may also be blank lines of input you should detect because you will use the length of the line as the denominator in determining the percentage of GC content, and you don't want to divide by zero!

```
$ cat inputs/sample2.txt
ATTTACAATAATTTAATAAAATTAACTAGAAATAAAATATTGTATGAAAATATGTTAAAT
AATGAAAGTTTTTCAGATCGTTTAATAATATTTTTCTTCCATTTTGCTTTTTTCTAAAAT
TGTTCAAAAACAAACTTCAAAGGAAAATCTTCAAAATTTACATGATTTTATATTTAAACA
AATAGAGTTAAGTATAAGAGAAATTGGATATGGTGATGCTTCAATAAATAAAAAAATGAA

AGAGTATGTCAATGTGATGTACGCAATAATTGACAAAGTTGATTCATGGGAAAATCTTGA
TTTATCTACAAAAACTAAATTCTTTTCTGAATTTATTAATGTCGATAAGGAATCTACATT
```

# Iterating lines in a file, characters in a string

In our first example, we'll look at an entire program complete with command-line argument parsing. Later examples will only cover the differences in the `main()` function.

```
#!/usr/bin/env python3
""" ①
Purpose: Calculate GC content
Author : Ken Youens-Clark <kyclark@gmail.com>
"""

import argparse ②


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

    parser = argparse.ArgumentParser(
        description='Calculate GC content',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', ③
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        help='Input sequence file')

    return parser.parse_args()


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

    args = get_args()       ④

    for line in args.file:  ⑤
        seq = line.rstrip() ⑥
        if seq:             ⑦
            # Iterate each base and compare to G or C, add 1 to counter
            gc = 0          ⑧
            for base in seq.lower(): ⑨
                if base == 'g' or base == 'c': ⑩
                    gc += 1                    ⑪
            pct = int((gc / len(seq)) * 100)   ⑫
            print(f'{pct:3}%: {seq}')          ⑬


# --------------------------------------------------
if __name__ == '__main__':
    main() ⑭
```

① A docstring for the program.

② Import the `argparse` module.

③ Define a "file" parameter which must be a readable text file.

④ Get the command-line arguments.

⑤ Use a "for" loop to iterate over each line in "args.file" which is an open file handle.

⑥ Remove trailing whitespace from the "line" and assign this to "seq" (sequence).

⑦ Check if there is a sequence. In a Boolean context, the empty string "" is consider "falsey," and no any non-empty string will be considered "truthy."

⑧ Initialize a counter variable to count the number of G/C bases seen.

⑨ Use a "for" loop to iterate over each base in the lowercased version of the sequence.

⑩ Check if the base is a "g" or a "c". Remember that we only need to check lowercase.

⑪ Increment the "gc" counter by 1.

⑫ Calculate the percentage of bases that are Gs or Cs, truncate to a whole number using the `int()` function.

⑬ Print the GC content and the original sequence.

# Using str.index() and str.count()

There is a `str.index()` function that will return the index of a character in a string. Remember that Python starts indexing at 0, so 1 is actually the second character:

```
>>> 'ACAC'.index('C')
1
```

This is an *unsafe* method as it will throw an exception when the given character is not present:

```
>>> 'ACAC'.index('G')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

The `str.count()` function is a safe function that will return the number of times a string is found inside another string:

```
>>> 'GATTACA'.count('A')
3
```

Note that it will not throw an exception when the character is missing:

```
>>> 'GATTACA'.count('X')
0
```

Here is a solution that uses `str.count()`:

```
def main():
    args = get_args()
    for seq in map(str.rstrip, args.file):
        if seq:
            # Use str.count() to safely count G/C
            gc = seq.lower().count('g') + seq.lower().count('c') ①
            pct = int((gc / len(seq)) * 100) ②
            print(f'{pct:3}%: {seq}')
```

① Convert the `seq` to lowercase and count the number of times "g" and "c" occur.

② The `gc` variable is an integer value containing the sum of the GC counts.

# Using a list comprehension and guard

The pattern of establishing some variable to aggregate some data like the counts of G/C above then using a `for` loop to iterate through some sequence and modifying that variable is almost always better expressed using a list comprehension. For instance, we can use the sequence to create a new list of only the characters which are G/C like so:

```
>>> seq = 'GATTACA'
>>> [base for base in seq.lower() if base == 'g' or base == 'c']
['g', 'c']
```

Here is a solution that uses two list comprehensions. The first is used to strip the whitespace off each line of the input file. The second creates a list of only the G/C bases:

```
def main():
    args = get_args()
    for seq in [line.rstrip() for line in args.file]: ①
        if seq:
            # Use a list comprehension with guard to select only G/C
            gc = [base for base in seq.lower() if base == 'g' or base == 'c'] ②
            pct = int((len(gc) / len(seq)) * 100) ③
            print(f'{pct:3}%: {seq}')
```

① Use a list comprehension to call `str.rstrip()` on each line of the input file.

② Create a new list with only the bases that are "g" or "c."

③ The `gc` variable is a `list`, so we can use the `len()` function to find how many G/Cs were found.

# Using map() and filter()

In the previous solution, we used two list comprehensions:

- The first was used to transform each line of the input file by removing the trainling newlines.
- The second added a guard clause to isolate only the characters of the sequence that were "g" or "c."

We can use a pair of *higher-order functions* called `map()` and `filter()` to rewrite each of these, respectively. Both functions take *another function* as the first argument and some *iterable* — something like a string or a list or a file handle of ordered elements that can be traversed from beginning to end.

A `map()` will apply the given function to each element of the iterable and will return a new list of the transformed elements. For instance, we can apply the `str.rstrip` function to each line of the input file. Here the newlines `\n` are removed from each string in a list. Note that we must use the `list()` function to coerce the lazy `map()` in the REPL:

```
>>> list(map(lambda s: s.rstrip(), ['foo\n', 'bar\n', 'baz\n']))
['foo', 'bar', 'baz']
```

To understand the `lambda`, imagine we create a function called `stripper()` that will call the `str.rstrip()` function on the given value:

```
>>> def stripper(s):
...     return s.rstrip()
...
>>> stripper('foo\n')
'foo'
```

We can use `lambda` to create the function and assign it to the `stripper` variable and the results will be the same:

```
>>> stripper = lambda s: s.rstrip()
>>> stripper('foo\n')
'foo'
```

The `lambda` actually creates an anonymous function, so it can be used as the argument to `map()`:

```
>>> list(map(lambda s: s.rstrip(), ['foo\n', 'bar\n', 'baz\n']))
['foo', 'bar', 'baz']
```

Because the `stripper` function from before accepts a single parameter, we could use it:

```
>>> list(map(stripper, ['foo\n', 'bar\n', 'baz\n']))
['foo', 'bar', 'baz']
```

Inside the function, we're calling code like so:

```
>>> 'foo\n'.rstrip()
'foo'
```

This is exactly the same as calling the `str.rstrip()` function and passing the string as an argument:

```
>>> str.rstrip('foo\n')
'foo'
```

Which means we can actually just use this method directly:

```
>>> list(map(str.rstrip, ['foo\n', 'bar\n', 'baz\n']))
['foo', 'bar', 'baz']
```

A `filter()` function will apply the given function to each element of the iterable and will only return those elements for which the function returns a "truthy" value. For instance, this will produce the same list of G/C as shown above:

```
>>> seq = 'GATTACA'
>>> list(filter(lambda base: base == 'g' or base == 'c', seq.lower()))
['g', 'c']
```

Here are the `map()` and `filter()` functions in context:

```
def main():
    args = get_args()
    # Use a map()/lambda to strip newlines
    for seq in map(str.rstrip, args.file): ①
        if seq:
            # Use filter() to select only G/C
            gc = list(filter(lambda base: base == 'g' or base == 'c', seq.lower())) ②
            pct = int((len(gc) / len(seq)) * 100) ③
            print(f'{pct:3}%: {seq}')
```

① Use `map()` to apply the `str.rstrip()` function to each line in the file handle.

② Use `filter()` to take only those bases that are "g" or "c".

③ The `gc` variable is a list, so use the `len()` function to find how many Gs and Cs where found.

# List membership

Individually comparing each base to "g" and "c" is tedious. We can use the `x in y` expression to find if the value `x` (here a single character) is in the collection `y` (here a longer string):

```
>>> 'G' in 'GATTACA'
True
>>> 'Z' in 'GATTACA'
False
```

This expression can also be used on other data structures like lists:

```
>>> 'bar' in ['foo', 'bar', 'baz']
True
```

Or dictionaries:

```
>>> 'baz' in {'foo': 1, 'bar': 2, 'baz': 3}
True
```

We can shorten the `lambda` by using this expression:

```
def main():
    args = get_args()
    for seq in map(str.rstrip, args.file):
        if seq:
            # Shorter way to write the filter lambda
            gc = list(filter(lambda char: char in 'gc', seq.lower()))
            pct = int((len(gc) / len(seq)) * 100)
            print(f'{pct:3}%: {seq}')
```

# Reducing a list using sum()

Instead of using `filter()` to create a list of the G/C bases, we could instead use `map()` to create a list with the value 1 for any G/C base and 0 otherwise:

```
>>> list(map(lambda base: 1 if base.lower() in 'gc' else 0, 'GATTACA'))
[1, 0, 0, 0, 0, 1, 0]
```

Then we can use the `sum()` function to *reduce* this list of integers to a single value:

```
>>> sum(map(lambda base: 1 if base.lower() in 'gc' else 0, 'GATTACA'))
2
```

Here is the code:

```
def main():
    args = get_args()
    for seq in map(str.rstrip, args.file):
        if seq:
            # Use map() to create a list of 1s for G/C and 0s otherwise, sum()
            gc = map(lambda char: 1 if char in 'gc' else 0, seq.lower()) ①
            pct = int((sum(gc) / len(seq)) * 100) ②
            print(f'{pct:3}%: {seq}')
```

① Create a list with 1 for G/C or 0 otherwise.

② The `gc` variable will be a list of 1s and 0s. Use `sum()` to find the total number of G/C bases.

# Using regular expressions

Regular expressions (AKA "regexes") are a way to describe *patterns of text*. Here we are looking for the literal characters "g" and "c," case-insensitive.

To use regexes, we must import the `re` (regular expression) module:

```
>>> import re
```

There are two functions for finding a pattern inside some text. The `re.match()` function will always start searching at the beginning of a string:

```
>>> re.match('G', 'GATTACA')
<re.Match object; span=(0, 1), match='G'>
```

This will fail for any pattern not found at the beginning:

```
>>> re.match('C', 'GATTACA')
```

This does not appear to return anything, but in truth it returns the special value `None` which has the `type()` of `NoneType`:

```
>>> type(re.match('C', 'GATTACA'))
<class 'NoneType'>
```

The `re.search()` function will search for the pattern *anywhere in the text*, which is what we want:

```
>>> re.search('C', 'GATTACA')
<re.Match object; span=(5, 6), match='C'>
```

Note that it, too, will return `None` when a pattern cannot be found:

```
>>> type(re.search('X', 'GATTACA'))
<class 'NoneType'>
```

Rather than searching for each character "G" and "C," we can create a *character class* to represent the upper- and lowercase versions of these:

```
>>> re.search('[GgCc]', 'GATTACA')
<re.Match object; span=(0, 1), match='G'>
```

That only finds the first occurrence of the pattern, so we can instead use the `re.findall()` function to find every place the pattern is found in the text:

```
>>> re.findall('[GgCc]', 'GATTACA')
['G', 'C']
```

This is the same list we found using the list comprehension with a guard and the `filter()` function, only this time we *described a pattern* and the regular expression engine did the grunt work of iterating through the sequence and find those bases that matched.

Here is the code in context:

```
def main():
    args = get_args()
    for seq in map(str.rstrip, args.file):
        if seq:
            # Use a regular expression to find G/Cs upper/lowercase
            gc = re.findall('[GgCc]', seq) ①
            pct = int((len(gc) / len(seq)) * 100) ②
            print(f'{pct:3}%: {seq}')
```

① The `re.findall()` function will find any character matching "G," "g," "C," or "c".

② The `gc` variable will be a list containing just those characters matching our regex.

# Using case-insensitive regular expressions

Rather than enumerating both the upper- and lowercase bases in our character class like `[GgCc]`, we could have used `[gc]` and lowercased the sequence:

```
>>> seq = 'GATTACA'
>>> re.findall('[gc]', seq.lower())
['g', 'c']
```

Another option is to use the flag `re.IGNORECASE` to tell the regex engine to disregard case:

```
>>> re.findall('[gc]', seq, re.IGNORECASE)
['G', 'C']
```

This can be shortened to the `re.I` flag:

```
>>> re.findall('[gc]', seq, re.I)
['G', 'C']
```

See `help(re)` in the REPL to read the documentation about these flags. Here is the code in context:

```
def main():
    args = get_args()
    for seq in map(str.rstrip, args.file):
        if seq:
            # Use case-insensitive searching
            gc = re.findall('[gc]', seq, re.IGNORECASE) ①
            pct = int((len(gc) / len(seq)) * 100) ②
            print(f'{pct:3}%: {seq}')
```

① Use the `re.IGNORECASE` flag to perform case-insensitive matching.

② The `gc` variable will again be a list of those bases matching the pattern.

# Author

Ken Youens-Clark <kyclark@gmail.com>