Three steps to readable, documented code

Documentation is a skill that is learned over the course of a career, but here's an exercise that I often have my students do. Using a framework like this can make documenting your code less daunting if you've no idea where to start.

Step one: make sure your variable and function names are meaningful

Programmers are fond of talking about self-documenting code – i.e. code that doesn't require external documentation to be understood. A large part of this is using meaningful variable names. Examples of bad variable and function/method names include:

- Single-letter names e.g. `a`, `b`, `f` (with the exception of variable names that follow common conventions such as `x` and `y` for co-ordinates or `i` for an index)
- Names that describe the type of data rather than the contents e.g. `my_list`, `dict`
- Names that are extremely generic e.g. `process_file()`, `do_stuff()`, `my_data`
- Names that come in multiples e.g. `file1`, `file2`
- Names that are excessively shortened e.g. `gen_ref_seq_uc`
- Multiple names that are only distinguished by case or punctuation e.g. `input_file` and `inputfile`, `DNA_seq` and `dna_seq`
- Names that are misspelled – the computer does not care about spelling but your readers might

Go through your code and look for any instances of the above, and replace them with good names. Good variable names tell us the job of the variable or function. This is also a good opportunity to replace so-called magic numbers – constants that appear in the code with no explanation – with meaningful variable names e.g. 64 might be replaced by `number_of_codons`.

Example: we want to define two variables which hold the DNA sequence for a contig and a frame, then pass them to a method which will carry out protein translation and store the result. Here's how not to do it, even though the code is perfectly valid Python:

```
1a = 2
2b = 'ATGCGATTGGA'
3c = do_stuff(a, b)
```

This is much better:

```
1frame = 2
2contig_dna_seq = 'ATGCGATTGGA'
3contig_protein_seq = translate(frame, contig_dna_seq)
```

Step two: write brief comments explaining the reasoning behind particularly important or complex statements

For most programs, it's probably true to say that the complexity lies in a very small proportion of the code. There tends to be a lot of straightforward code concerned with parsing command-line options, opening files, getting user input, etc. The same applies to functions and methods: there are likely many statements that do things like unpacking tuples, iterating over lists, and concatenating strings. These lines of code, if you've followed step one above, are self-documenting – they don't require any additional commentary to understand, so there's no need to write comments for them.

This allows you to concentrate your documentation efforts on the few lines of code that are harder to understand – those whose purpose is not clear, or which are inherently difficult to understand.

Here's one such example – this is the start of a function for processing a DNA sequence codon-by-codon (e.g. for producing a protein translation):

```
for codon_start in range(0, len(dna)-2, 3):
codon_stop = codon_start+3
codon = dna[codon_start:codon_stop]
    ...
```

The first line is not trivial to understand, so we want to write a comment explaining it. Here's an example of how **not** to do it:

```
# iterate over numbers from zero to the length of
# the dna sequence minus two in steps of three
for codon_start in range(0, len(dna)-2, 3):
...
```

The reason that this is a bad comment is that it simply restates **what the code does** – it doesn't tell us **why**. Reading the comment leaves us no better off in knowing why the last start position is the length of the DNA sequence minus two. This is much better:

```
# get the start position for each codon
# the final codon starts two bases before the end of the sequence
# so we don't get an incomplete codon if the length isn't a
multiple of three
for codon_start in range(0, len(dna)-2, 3):
 ...
```

Now we can see from reading the comment that the reason for the -2 is to ensure that we don't end up processing a codon which is only one or two bases long in the event that there are incomplete codons at the end of the DNA sequence.

Go through your code and look for lines whose function isn't obvious just from reading them, and add explanations

Step three: add docstrings to your functions/methods/classes/modules

Functions and methods are the way that we break up our code into discrete, logical units, so it makes sense that we should also document them as discrete, logical units. Everything in this section also applies to methods, classes and modules, but it keep things readable I'll just refer to functions below.

Python has a very straightforward convention for documenting functions: we add a triple-quoted string at the start of the function which holds the documentation e.g.

```
def get_at_content(dna):
  """return the AT content of a DNA string.
     The string must be in upper case.
     The AT content is returned as a float"""
  length = len(dna)
  a_count = dna.count('A')
  t_count = dna.count('T')
  at_content = float(a_count + t_count) / length
  return at_content
```

This triple-quoted line is called a docstring. The advantage of including function documentation in this way as opposed to in a comment is that, because it uses a standard format, the docstring can be extracted automatically. This allows us to do useful things like automatically generate API documentation from docstrings, or provide interactive help when running the Python interpreter in a

shell (take a look at the chapter on testing and documentation in Advanced Python for Biologists for an in-depth look at how this works).

There are various different conventions for writing docstrings. As a rule, useful docstrings need to describe the order and types of the function arguments and the description and type of the return value. It's also helpful to mention any restrictions on the argument (for instance, as above, that the DNA string must be in upper case). The example above is written in a very unstructured way, but because triple-quoted strings can span multiple lines, we could also adopt a more structured approach:

```
def get_at_content(dna):
  """return the AT content of a DNA string.

     Arguments: a string containing a DNA sequence.
                The string must be in upper case.

     Returns: the AT content as a float"""
  ...
```

If you think it's helpful, you can also give examples of how to use the function in the docstring. Notice that we're **not** saying anything in the docstring about **how the function works**. The whole point of encapsulating code into functions is that we can change the implementation without worrying about how it will affect the calling code!