Data Structures

DNA Manipulation

In molecular biology, a DNA sequence can be represented by a string made up of the characters `A`, `T`, `G` and `C`. The length of a DNA sequence must also be a multiple of three.

1. Write a function `is_dna(string)` which returns `True` if the input is a valid DNA sequence and `False` otherwise.

```
>>> is_dna("AATGAC")
True
>>> is_dna("ACtACT")
False
>>> is_dna("AATC")
False
```

2. The *reverse complement* of a DNA sequence is computed by interchanging `A` with `T`, and `G` with `C`, and reversing the string. For example, the reverse complement of `AATGAC` is `GTCATT`. Write a function `reverse_complement(dna)` which returns the reverse complement of the sequence `dna`. If the sequence is invalid, return `None`.

```
>>> reverse_complement("AATGAC")
'GTCATT'
>>> reverse_complement("GATTACAAA")
'TTTGTAATC'
>>> print(reverse_complement("GAGA"))
None
```

3. A DNA sequence is divided up into blocks of three letters, called *codons*. For example, the sequence `GATTACAAA` contains three codons: `GAT`, `TAC` and `AAA`. Write a function `print_codons(dna)` which one-by-one prints each of the codons of the sequence `dna`. If the sequence is invalid, do nothing.

```
>>> print_codons("GATTACAAA")
GAT
TAC
AAA
>>> print_codons("AATGAC")
AAT
GAC
>>> print_codons("GAGA")
>>>
```

> **Challenge: Returning Codons**
>
> In general, a function that performs a logical task, such as dividing a DNA sequence into codons, should not perform output from within the function. It is better to return the result and let the calling code determine what to do with the result. This makes it easier to reuse the function in other parts of a program. It also means that the code for outputting data is not spread throughout the logic of the program.

Write a `generate_codons(dna)` function that returns a list containing each of the codons of the sequence `dna`. If the sequence is invalid, return `None`.

Modify the `print_codons(codons)` function so that the parameter is now a list containing the codons. Print each codon found in the list on a separate line, like in the previous implementation of `print_codons(dna)`. For the purposes of this exercise, you may assume that the list contains a valid set of codons.

```
>>> generate_codons("GATTACAAA")
['GAT', 'TAC', 'AAA']
>>> result = generate_codons("GA")
>>> type(result)
<class 'Nonetype'>
>>> print_codons(generate_codons("GATTACAAA"))
GAT
TAC
AAA
>>>
```

Aside: Dealing with Bad Input

If we encounter a string which is not an invalid DNA sequence, the optimal situation would be not to 'ignore' the invalid data by doing nothing, instead we should cause an error to occur, by raising an exception:

```
...
if ... :
    raise ValueError("Invalid DNA sequence")
...
```

We will cover raising and handling exceptions later in the course.

Number Extraction

Data mining is the process of discovering and analysing information from raw datasets. Often, most of the data is not relevant, and so the first task in data mining is to extract only the useful information. In this task, we are given a string that contains numbers, and we want to extract the first integer out of the string.

Write a function `get_number(string)` which returns the first positive integer in the string, or `None` if there are no integers in the string. You may find it useful to use the `str.isdigit` method.

```
>>> get_number("zbagy22.17clguba19")
22
>>> get_number("ubyl 014tenvy,1975 ncevy9gu")
14
>>> print(get_number("sylvatpvephf"))
None
```

Challenge: Negative Numbers

Modify this function to handle negative integers.

```
>>> get_number("zl-39u0irepensg")
-39
>>> get_number("vf-shyy23-4bs")
23
>>> get_number("33yf-")
33
```