Programming for Bioinformatics | BIOL 7200

Exercise 10

Submission specifications

1. Your module ***does*** need to be generalizable to any SAM input files.
2. You may only use Python standard library modules
3. Your submission should consist of a module file or the contents of a package directory including a __init__.py file as in previous assignments.

Background

Assemblies

Until this point our magnum opus has been focused on extracting information from assemblies. Assemblies are hypotheses about the structure and order of DNA or RNA present in an organism based on some kind of sequencing data (structure here referring to things like the number of chromosomes and other genetic elements). There are different ways to generate these hypotheses about how nucleotides are organized inside an organism (e.g., *de novo* or reference-based assembly), and the sequencing data used can also differ (e.g., Illumina short reads, Oxford Nanopore, Pacbio long reads, Sanger sequencing, etc.). However, all assemblies represent an attempt to **summarize** what can be inferred about the order and organization of nucleotides based on the sequencing data available.

The use of assemblies is not always appropriate or necessary. For example, as assemblies summarize sequencing data, they contain less information than the raw sequencing data upon which they are based. Many analyses require the additional information contained in raw sequence data. Furthermore, assemblies are usually an attempt to represent the entire genome of an organism. To test a hypothesis about a specific region of the genome, it is not necessary to spend the time and computational resources to assemble the whole genome. instead, raw reads can be used without the need to generate an assembly.

Finally, assemblies are often not accurate representations of the sequence or organization of a genome. Assemblies represent a hypothesis of what the DNA you sequenced looked like. However, assembly software can make mistakes, omit data, or otherwise misrepresent the true sequence and structure of the sequenced organism. This is not to say that assemblies have no value. Depending on the quality of the input data, assemblies can be very close to the true genome. However, it is generally necessary to examine the raw sequencing data in addition to assembled sequences for any situation where accuracy is important to your needs.

Sequencing (Illumina short reads)

We are going to write a new entrypoint for our magnum opus to accept Illumina short reads. Illumina is one of several companies which sells a technology for nucleotide sequencing. During the "next-generation" sequencing period of the early 2010s, Illumina established itself as the sequencing technology of choice, and is still the most commonly used sequencing technology (although long-read sequencing is becoming more common as its price drops).

The process by which Illumina sequencing is performed is shown in the video linked here. For this assignment it is not necessary that you are familiar with how Illumina sequencing works except for a few features that impact the structure of the data produced. The key elements of Illumina sequencing for our purposes are as follows. We'll focus on the approach to sequence a pure sample of only one strain or genotype here, but mixed samples can also be sequenced in the same way - the subsequent analyses are just more complicated.

1. Illumina sequencing typically starts with many copies of the same DNA. For example, the sample being sequenced might be a tissue sample composed of millions of cells from the same patient, or could be a colony of bacteria grown on a plate (bacteria reproduce asexually so all members of the colony will be identical clones).
2. The DNA is broken up into small fragments (ideally ~ 500bp or so). Because we started with lots of copies of the same sequence and broke them all up randomly, many of the fragments will contain overlapping stretches of sequence.
3. Each fragment is then sequenced from both ends, resulting in two paired reads. A pair in this context indicates that both reads were produced by sequencing the same fragment of DNA, with each read containing the sequence of one of the two DNA strands in the fragment.
4. In addition to recording the sequence of the read, Illumina sequencers also record a quality score. This score is a representation of how confident the sequencer in that the base it called. We'll discuss this score a bit more below.

FASTQ format

After performing a sequencing run, you will get out a large amount of sequencing data which is typically stored in FASTQ format. FASTQ is similar to the FASTA format we have been working with. The key difference is that, in addition to the DNA sequence of the read, a quality score for each base is also included (i.e., the "Q" in FASTQ). FASTQ format looks like this

```
@Sequence_header
ATCGATCG
+
:;A@?AA!
```

Each entry in FASTQ format is made up of four lines:

1. The first line begins with an @ symbol and the rest of the line is the sequence header. This line often contains lots of information about the sequencing run. However, you will also see very simple headers.
2. Next is the sequence. This is just like the sequence you have been working with in FASTA format, except that in FASTQ format all of each sequence must be on a single line.
3. The third line is simply a spacer. This line begins with a "+" and sometimes also has the header text again. The repetition of the header on this line is optional.
4. The fourth line contains quality information about the sequence. Each character of the quality line indicates the quality of the corresponding character in the sequence line. These quality characters encode information according to a scoring system called "Phred scores". Note that in the above example there are @ symbols in the quality score line. While in FASTA format the header character > is only found on header lines, that is not true for FASTQ format.

Phred scores

When Illumina sequencers are taking photos of their flow cells and making calls about which DNA strands have a flash, there is a level of uncertainty about exactly which strand of DNA the flash came from. There are many factors that can impact the ability of the sequencer to make confident base calls and the level of confidence is reported as a percent probability that each call is wrong.

These probabilities are important data to consider when analyzing sequencing data. For example, if you were to find that some of your sequencing reads contain a mutation in the same place relative to your reference, but the quality scores for the mutant base calls are low, you might be less confident that the mutation is real.

Each base in the sequence line has an associated quality score. These scores are encoded in such a way that each quality score is only one character long. This design choice makes fastq data much easier to work than if the quality scores were each a 3-digit (or longer) percent score.

The encoding used in FASTQ format is called a Phred score (after the software in which it was first implemented). Phred scores are calculated using the following equation

$Q = -10 \log_{10}P$

Where P is a percent probabilty expressed as a number between 0 and 1. Q is always rounded to the nearest integer.

Conversely, quality scores can be converted to the corresponding probability using the formula

$P = 10^{-Q/10}$

So, a 100% chance that the base call is wrong corresponds to a quality score of 0. A 10% chance corresponds to a score of 10, a 1% chance to a score of 20, and so on.

In order to express these quality scores using a single character, Phred scores take advantage of ASCII character codes. When text is encoded in a computer, it is stored as a sequence of numbers. Different text encodings use different numbers to represent the characters they encode. In ASCII encoding, the numbers 1-31 encode non-printable characters, while 32 encodes a space character. 33-127 encode printable characters such as letters, numbers, and special characters like :;!?() etc. Because ASCII uses the first 32 numbers to encode non-printable characters and space, the Phred scale uses the characters encoded using 33 and up to convert Q scores into printable characters. When quality scores are encoded using ASCII characters in this way they are often called "Phred scores".

You can find a handy lookup table to convert between Q, P, and Phred scores at the MUSCLE developer's website. You can also convert from characters to their ASCII codes in python using the built-in function `ord()` and from number to character again using `chr()`. Phred scores are simply the ASCII code minus 33.

read mapping

There are broadly two approaches to the problem of how to take the short sequence fragments that are generated during DNA sequencing and piece them back together into a representation of the larger DNA molecules from which they were derived: *de-novo* assembly, and read mapping.

de-novo assembly (where *de-novo* is Latin meaning "from new" or "from the beginning") attempts to fit together all the sequencing reads without any prior information about the DNA from which they were derived. This process is computationally expensive and algorithmically complex.

Read mapping involves identifying where in a known reference sequence each read might align. Read mapping broadly involves using some sort of approach to identify candidate locations where each read might align, and then perform a pairwise local alignment between the read and reference sequence, similar to the approach used by BLAST.

There are lots of read mapping tools available, and some may perform better under different situations. I often use Minimap2, which is a read mapping tool that works well for both short and long read data. You can read the full description of how it works in the associated publication, or read a short description of the algorithm is uses on the Github page. Most mapping software will produce a .sam file as an output in order to describe how reads align to the reference sequence.

SAM format

Sequence Alignment and Mapping format (SAM format) is a file format for encoding alignment information between a query and reference sequence. You can find a short description in the associated publication and a much longer and more detailed description in the format specification documentation. SAM format was developed as a tool to standardize how sequence mapping information was stored and represented. The format supports the storage of lots of useful information about how reads align to reference sequences.

SAM files have up to two sections:

1. An optional header section composed of lines beginning with "@" that describe characteristics of the whole dataset such as the sequence headers in the reference and the command used to map reads.
2. The alignment section. This section contains the details of read mappings. Each separate mapping is a single line, where each read may have multiple mappings (more about multiple mappings later). The alignment section of the SAM file is typically the part you care most about as that's the section that actually describes the read mapping information.

Below is a slightly modified version of the first few lines of the example SAM file which I provided for this assignment to illustrate the key components of SAM format.

```
@HD     VN:1.6  SO:coordinate
@SQ     SN:Fusibacter_paucivorans       LN:1525
@PG     ID:minimap2     PN:minimap2     VN:2.28-r1209   CL:minimap2 -ax sr -k10 -B0
refs/Fusibacter_paucivorans_16S.fna reads/sub_ERR11767307_1.fastq reads/sub_ERR11767307_2.fastq
ERR11767307.541398      163     Fusibacter_paucivorans  1       60      68S83M  =       314     464
CAAGAAACAAACCATAAAGCCAGATATTTTGATAACAATAGTATCTGAGCCTGATAAACTTTTATTTGAGAGTTTGATCCTGGCTCAGGATGAACGCTGGCG
GCGTGCCTAACACATGCAAGTTGAGCGATTTACTTCGGTAAAGAGCGGC
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF   NM:i:15 ms:i:136        AS:i:136        nn:i:0
tp:A:P  cm:i:8  s1:i:105        s2:i:0  de:f:0.1807     rl:i:0
ERR11767307.1723569     163     Fusibacter_paucivorans  1       60      60S91M  =       376     529
AAACCATAAAGCCAGATATTTTGATAACAATAGTATCTGAGCCTGATAAACTTTTATTTGAGAGTTTGATCCTGGCTCAGGATGAACGCTGGCGGCGTGCCT
AACACATGCAAGTTGAGCGATTTACTTCGGTAAAGAGCGGCGGACGGGT      F-
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5FFFFFFFFFFFFF   NM:i:18 ms:i:146        AS:i:146        nn:i:0
tp:A:P  cm:i:9  s1:i:117        s2:i:0  de:f:0.1978     rl:i:0
```

There are 11 mandatory fields in each line of a SAM file alignment section (also described in the documentation):

1. QNAME - the name of the query sequence (e.g., read sequence header)
2. FLAG - an integer that encodes bitwise flags describing properties of the mapping (more later)
3. RNAME - the name of the reference sequence to which the read mapped (i.e., FASTA header of reference sequence)
4. POS - the 1-base leftmost position to which a base in the read is mapped (leftmost in the reference sequence)
5. MAPQ - a mapping quality score describing how well supported this alignment is
6. CIGAR - a sequence of operators and lengths describing the details of how bases in the query and reference line up (more later)
7. RNEXT - sequence ID of the reference to which the mate of this read (i.e., the other member of the pair from the same DNA fragment) mapped
8. PNEXT - the POS of the mate read
9. TLEN - called the "observed template length", this value is a bit complex so I'll describe it below even though we don't need it for this assignment.
10. SEQ - the sequence of the read that was mapped
11. QUAL - the Phred scores of the bases in SEQ

The full TLEN description: this value takes into account both of a pair of reads and is simply the number of bases from the POS of the leftmost-mapped read to the rightmost mapped base of the rightmost-mapped read. The value is given as a positive integer for the forward-mapped read and a negative integer for the reverse-mapped read. The "observed template length" part describes how, based on the mapping to the reference, that distance is the length of the originally sequenced DNA fragment for which we have evidence. If you watched the Illumina sequencing video linked above, you will have seen how each fragment of DNA is sequenced from the ends towards the middle. If the fragment of DNA is longer than the sum of the two read lengths produced from it, the reads won't meet in the middle and will have no overlap. In that case, based on just the pair of reads we have no idea how long the DNA fragment that was sequenced was. When you map those reads to a reference sequence, you can assess how close together the two reads map. You can then hypothesize how big the DNA fragment might have been based on the distance between the two ends of the mapped pair. That's what TLEN is - a hypothesis of the original fragment length.

SAM FLAG field

The SAM format FLAG field contains an integer encoding various properties of the read mapping described by that entry (i.e., line) in the SAM file. These properties are encoded in the binary representation of the integer. If you are unfamiliar with binary, recall the description provided all the way back when we covered permissions in Bash. There are three permission types (read, write, execute) which can be either set or unset (two states = binary). We discussed how we can encode any combination of those three permissions using an integer from 0 to 7, where 0 is all unset and 7 is all set.

SAM format defines 12 flags which means that the FLAG value is composed of a binary string with 12 bits. If you convert the value in the FLAG column to a binary string and assess which bits are set, then you can see which properties the read mapping has.

You can convert integers to their binary representation in Python using the built-in function `bin()`. Let's use that to explore the properties indicated by the commonly seen flag value, 99. Below I'm showing an interactive Python session where ">>>" is like the

"$" prompt in a Bash shell and lines without ">>>" are outputs.

```
>>> bin(99)
'0b1100011'
```

Binary strings returned by the `bin()` function begin with "0b", but that is not part of the binary. You can ignore the "0b" here.

The first thing to note about the binary in the output shown above is that there are not 12 bits shown. That's because Python is only showing up until the most significant bit that is set. The most significant bit is simply the bit that corresponds to the number with the greatest magnitude. If you think about the binary string shown above and look at how much each bit adds to the total value, the bit on the right adds 1 to the total value, while the bit on the left adds 64 to the total value. The left-most bit is therefore the most-significant - whether it is 0 or 1 has the biggest impact on the overall value. See the below table for an illustration of bit values and their significance in a 12-bit binary string

Significance	**Most**											**Least**
Bit	11	10	9	8	7	6	5	4	3	2	1	0
Value	2^{11}=2048	2^{10}=1024	2^{9}=512	2^{8}=256	2^{7}=128	2^{6}=64	2^{5}=32	2^{4}=16	2^{3}=8	2^{2}=4	2^{1}=2	2^{0}=1

Unless you tell Python how many bits to print, it has no way of knowing which bits to show that are more significant, but unset (i.e., 0). If you really want to always show the same number of bits in a binary string, you can use string formatting to achieve it using f-strings (consult the string format mini-language for more f-string magic!).

```
>>> f"{99:012b}" # 012b means 0-pad to 12 bits in binary
'000001100011'
```

SAM flag meanings

SAM flags are a set of 12 true or false statements. Each statement is assigned a bit in a 12-bit binary string and each statement's value is encoded as a 1 if it is true or a 0 if it is false.

Each bit in a SAM flag indicates a different property of the associated read mapping. The first flag corresponds to the right-most (least significant) bit, so you would read a binary string right to left to correspond bits to flags. This website provides a useful interactive tool to see which bits are set in a given FLAG value and provides a short description of what each flag indicates. FLAG meanings are also described in the SAM format documentation. I'll also describe them here with a bit of extra description:

Bit	**Decimal**	**Name**	**Meaning**
1	1	PAIRED	This read has a mate in your sequencing data
2	2	PROPER_PAIR	Both mates in the pair are mapped in the expected orientation and spacing
3	4	UNMAP	This read was not mapped
4	8	MUNMAP	The mate of this read was not mapped
5	16	REVERSE	This read was mapped to the reverse strand of the reference sequence
6	32	MREVERSE	The mate of this read was mapped to the reverse strand of the reference sequence
7	64	READ1	This read is mapped further left in the reference than its mate
8	128	READ2	This read is mapped further right in the reference than its mate
9	256	SECONDARY	This is not the best mapping of this read, but is still good enough to report
10	512	QCFAIL	This read failed quality control checks
11	1024	DUP	This read is identical to one or more other read in your dataset due to technical issues during sequencing

Bit	Decimal	Name	Meaning
12	2048	SUPPLEMENTARY	A different, larger part of this read mapped elsewhere, but this portion maps here instead

SAM flags provide a simple way to quickly assess valuable information about a read mapping in SAM format. One of the most common ways you will use the FLAG field is to filter reads to include or exclude read alignments with certain flags. For example, you might filter a SAM file to exclude lines with the 4 bit set, which corresponds to the read being unmapped. After doing so you would be left with only the reads which have mapping information.

To restate what I said above about these flags being true or false statements, the flag with a decimal value of 4 corresponds to the read not being mapped. That means if there is a 1 in this bit, it is true that the read was not mapped. If there is a 0 then it is false that the read is unmapped (i.e., the read *was* mapped). This can be a bit confusing as some of the flags are positive-sounding attributes such as "paired", while others are negative-sounding attributes like "unmapped" and "QC fail". Crucially the 0 or 1 in a bit doesn't indicate a good or bad thing, it's just whether the condition associated with that flag is true or false.

SAM CIGAR strings

For reads which do map to the reference sequence, there are several fields which report information about the mapping. The most complicated of these is the CIGAR string. The CIGAR string includes details of how each base in the read aligned to each base in the reference. This information is encoded as a sequence of number-letter pairs. In each pair, the letter indicates the way in which some bases align, and the number indicates how many bases are being described. The types of alignment feature described in CIGAR strings are as follows:

Character	Meaning
M	Bases in read align to bases in reference without gaps
I	Insertion in read relative to the reference
D	Deletion in read relative to the reference
S	Soft clipping
H	Hard clipping

You may encounter other characters in CIGAR strings, but you don't need to know about them for this assignment. For example, some software splits M into = and X for matching and mismatching bases, respectively.

Clipping indicates that an end of the read does not align with the reference. Soft clipping usually indicates a sequence mismatch or that the read maps to the beginning or end of the reference (and so there is no reference sequence to map to). Hard clipping is often due to low sequence quality (i.e., poor quality sequence on one end of a read is dealt with by clipping off the low-quality bases and ignoring them), but can also be added to mask unwanted parts of reads during analysis of mapping data. Both kinds of clipping indicate that the clipped portion of the read is not part of the alignment, but they differ in an important way: soft clipped bases are still included in the SEQ and QUAL columns, while hard clipped bases are not.

An important thing to note about the M character in CIGAR strings is that it does not indicate that the read and reference sequences were necessarily the same. M only indicates that each base in the read maps to a single base in the reference so there are no gaps in this portion of the alignment.

For example, what would the CIGAR string 61S10M5I79M3D25M mean about a read mapping?

- 61S means the first 61 bases of the read were not mapped to the reference and were soft clipped
- 10M means the next 10 bases aligned to the reference without gaps
- 5I means there is then a 5 base insertion in the read
- 79M means the next 79 bases of the read align with the reference
- 3D means there is a 3 base deletion in the read
- 25M means the last 25 bases in the read align to the reference.

As there are no more characters in the cigar string after the final M, that means that the end of the read aligned to the reference. If the end of the read had not aligned, we would expect to see soft clipping at the end of the CIGAR string too.

Getting the hang of what CIGAR strings tell you about read mappings takes time. I have provided an additional handout (Exploring_read_mapping_with_IGV) that walks through mapping reads to some example reference sequences to see how modifying the reference sequences changes the CIGAR strings. Unless you are intimately familiar with read mapping, I recommend you read that document and spend some time experimenting with mapping reads to contrived sequences to become comfortable with what each component of a cigar string indicates about the alignment between read and reference.

16S

Over the remaining assignments, we are going to specialize our magnum opus tool to classify prokaryotic organisms using their 16S ribosomal RNA (rRNA) sequence. The 16S sequence is commonly used to differentiate prokaryotic organisms at the species level. Sequences used to classify organisms generally need to satisfy 3 criteria:

1. They must be present in all target organisms
2. They must differ enough between each organism in order to tell the organisms apart
3. They must not differ so much that the sequences cannot be aligned confidently or so that many positions in the sequence have mutated more than once (a base changing and then changing again looks like 1 difference when it is actually 2 so you under count sequence distance)

The 16S can be used to differentiate highly divergent organisms including Archaea and Bacteria because the last common ancestor of those organisms had a 16S ribosomal RNA gene and, because the gene is essential for life, the sequence is sufficiently conserved to allow useful comparisons. However, the same features that make the 16S well-suited to distinguish Archaea and Bacteria make it unsuitable for distinguishing strains of the same species of bacteria. Mutations accumulate very slowly in the 16S gene so it can only be used to reliably distinguish organisms which diverged long ago.

Assignment details

Your assignment over the next two weeks will be to use Python classes to build up the functionality to use read mapping information to extract the sequence of 16S genes in Illumina sequencing data. This will serve as a second entry-point into your magnumopus package. That means that given either assemblies or sequencing reads, your magnumopus will be able to extract 16S sequences and create an alignment.

The process we are going to take to extract sequence information from reads will be as follows:

1. Use an external tool (e.g., minimap2) to align reads with a reference sequence
2. For each sequencing read, use the mapping information in the SAM file to determine which bases this read indicates are in the sample
3. Assess the base calls in all of the reads to determine what the sequence is at each position in your sample

Step one in this process will be handled by an external tool. Steps 2 and 3 are what we will be coding ourselves over the next two exercises because they offer an excellent opportunity to use Python classes. Classes are well suited to storing data and for storing methods for interacting with those data. Over the next two weeks we are going to write two related classes to process sequence data.

In the first assignment we will be writing a class to store an entry in a SAM file and which will provide methods for interacting with that SAM entry. This first class will allow us to extract information about individual read mappings. In the second assignment we will write a class to contain all of the individual SAM entries. This second class will both store instances of our first class and provide functionality that will allow us to extract information about the whole SAM file.

Our goal in writing these two classes is to be able to read in a SAM file and to output the consensus sequence. With that goal in mind, we will design our first class so that it allows a user to identify key features of the read mapping and to lookup what base the read indicates is at a certain position in the reference. Then we will design our second class so that it stores instances of the first class in a usable form and provides the ability to extract base calls from all the reads and then summarize them.

For this assignment and next weeks assignment, your task is to add classes to your magnumopus package which satisfy the listed requirements. I have provided you the requirements of what your classes should do in two forms: a written description of the functionality, and unit tests which assess whether your code performs as desired. Unit tests are described below. The provided unit tests are essentially what the autograder runs to assess your code. The benefit of providing them to you with this assignment is twofold: most importantly, it allows you to see if your code is working more quickly than uploading and waiting for the autograder to run. It also provides an introduction to tests for any students who have not previously encountered them before.

Assignment steps:

1. Create a module called "sam" within your existing magnumopus package.
2. Within the new sam module, write a `Read` class that stores each field of the SAM entry in named attributes and includes the following methods and attributes (You may implement any other methods that you wish in support of the below methods. However, only the named methods below will be assessed):
    1. Write an `__init__` method to construct an instance of your `Read` class from a (non-header) line in a SAM file
    2. Use the "flag" field of the SAM entry to return booleans corresponding to the following mapping attributes:
        1. `is_mapped` (did the read map to the reference sequence?)
        2. `is_forward` (did the read map in the forward direction?)
        3. `is_reverse` (did the read map in the reverse direction?)
        4. `is_primary` (is this SAM entry the primary mapping of this read?)
    3. Write a `base_at_pos`method to take a 1-base position in the reference sequence and return the base mapped to specific position in the reference. The base should be the same strand as the reference. You will need to use POS value and CIGAR string information for this. If the alignment does not include the requested base (i.e., the positions is before or after this mapping, are deleted in the read, or were clipped from this alignment), return an empty string. This method should take an `int` specifying the 1-base position in the reference for which an aligned base should be returned as a `str`. e.g., `def self.base_at_pos(self, pos: int) -> str:`
    4. Write a `qual_at_pos` method to return the quality score of a base mapped to a specific position in the reference (same as base_at_pos but for the QUAL field). This method should take an `int` specifying the 1-base position in the reference for which an aligned base's quality score should be returned as a `str`. e.g., `def self.qual_at_pos(self, pos: int) -> str:`
    5. Write a `mapped_seq` method which returns the mapped portion of the read that corresponds to the reference sequence (i.e., no clipped bases). Deletions relative to the reference should be represented using dashes of length equal to the deletion length ("-"). Insertions relative to the reference should also be included. This method should take no input and return a `str` of the mapped portion of this alignments. e.g., `def self.mapped_seq(self) -> str:`

Hint: When tackling the `base_at_pos` and `qual_at_pos` methods remember that there are two crucial data:

1. The POS value of the alignment in order to determine where you should start counting within the sequence positions.
2. The CIGAR string. Specifically, for each CIGAR operation until you reach the requested position, does the operation consume (i.e., add to the alignment) bases in the query, the reference, or both. The idea of an operation "consuming" a base here is similar to the direction of traceback in Needleman-Wunsch: a vertical move in NW traceback adds the base along the left side of the matrix to the alignment. A horizonal move add the base along the top to the alignment. A diagonal move adds both bases to the alignment. If you have trouble with this be sure to go through the "Exploring_read_mapping_with_IGV" document and ask for help if you are stuck.

I have provided you with an example SAM file and the reference to which it corresponds in the "data.tar.gz" file. In addition, the "Exploring_read_mapping_with_IGV" document describes how you can generate read mapping data yourself in order to get access to real SAM alignment data that you can use when developing your code.

Unit testing

In some of the previous assignments I have provided you with one or more expected outputs. What I expected (and hoped) that you would do in those cases is to write some code, run the code and compare its outputs to the expected outputs, and then repeat that process until the outputs match.

The use of the sorts of test cases that I have been providing you is something that you can continue to use in future development. Ideally, when sitting down to code something, you should have a concrete idea in your head of exactly what your code should do. What I mean by that is that you should have a sense of what the inputs to your code will look like, and what the outputs should be given those inputs. If you were to attempt to start a coding project without a clear idea of those things you would have difficulty both in designing your code and in confirming that it works properly. So, testing is a crucial step in any development process.

Testing your code by running whole scripts on a set of input files and comparing each output against expected outputs works fine on a small scale, but it can be quite laborious if you want to check inputs with lots of different features. Furthermore, if you change a lot of parts of your code and find that some of your outputs no longer match the expected output it can be difficult to figure out exactly where the mistake is now occurring. Fortunately, there is a better option: unit tests. Unit tests are what the autograders you have been interacting with use to assess your code.

What is unit testing

Unit tests are simply code that assess the performance of some other portion of code. You can define the "unit" of code that is tested on any scale. Typically, a unit test is written to assess whether a single function or method is working correctly.

A unit test has two key components:

1. Run the unit of code on an input
2. Assert that the output is equal to an expected value

If the output is equal to the expected value, the test is passed. If the values differ, the test fails.

For example, if you have a function that adds two numbers together

```
def add(num1, num2):
    return num1 + num2
```

You could write a test like this

```
x = add(3, 4)
assert x == 7
```

The keyword `assert` in Python checks a condition and throws an `AssertionError` exception if the condition is false. The above test therefore runs the `add()` function with some input for which we know what the answer should be and then compares the output to our expectation. If we modified the function to instead subtract, then the existing test would now fail.

By writing clearly defined tests, which are stored in a script, we have two things: a permanent record of the conditions that we have thought of, and a way to quickly and easily check those conditions by running the test script.

How are unit tests used

There are a few ways that unit tests can be used. Perhaps the most important is to help you to not introduce bugs into your code, but they can also be used in the initial writing of the code too.

A common issue when writing code to perform a complex function is that you can't generally think of every edge case that you might encounter. You might therefore find that, after having written something that works for the cases you can think of, you encounter a weird situation that hadn't occurred to you for which your code fails. You would then set about changing your code so that it works for the newly discovered situation. This might happen more than once. The difficulty then becomes how to make sure you haven't made a change that makes your code not work for a situation in which it had previously worked.

Using unit tests in a situation like that removes the risk of accidentally introducing new bugs into your code so that it doesn't work in cases it used to. Here's how you might use unit testing in such a situation:

1. Write your code to handle the situations you are aware of
2. Write a unit test for each of the situations you have handled and confirm all the tests pass
3. Encounter a new situation in which your code doesn't work
4. Change your code so that it works for the new situation
5. Re-run all of your existing tests to confirm your code still works for the previously known situations
6. Add a new test that tests the newly found situation

If you use an approach like this, you will be much less likely to publish code in which you have introduced new bugs. If you did make a change that broke functionality for something that worked before, and you had a test that assessed it, you would know immediately and could fix the issue. Well-written tests are therefore extremely valuable, especially when developing complicated code or when working as part of a team where you may not be intimately familiar with the code you are changing.

In addition to tests being useful after you have written your code, they can also be helpful before you have written anything. The process of writing tests before code is called "test-driven development".

The idea of test-driven development is that before you start writing your code you already know what the code will do. You know what inputs it will take and what output it should produce. You can therefore write a test that defines the expected behavior. At first, the test will fail if you run it as you haven't written any code. However, you would next write enough code to perform the tested behavior. Then the test will pass. Next you write a new test that assesses the next step your code should perform and then repeat this process until you are finished.

While the formal process described above might sound needlessly laborious, you might find that you've already been taking that approach in your development. If you've ever written a bit of code and then run the script to see if the output looks how you want yet, then you made more changes, you have engaged in a form of test-driven development.

Test-driven development is especially valuable because the exercise of defining exactly what your code should do before you start writing it encourages you to be targeted and precise in the code you write. After all, it doesn't make much sense to start writing before you have a good sense of what you are doing.

Unit tests in this assignment

I am providing you with unit tests for this assignment as a way of showing you how they work and how they can be used. Python has a number of frameworks to allow the use of unit testing. For this assignment I have provided tests written for pytest. You do not need to understand how these tests work for this assignment as you are not being asked to write your own tests. However, the documentation on the pytest website provides an excellent overview of how to write and use tests using the pytest software.

The basic functionality of pytest (and other python testing software) is to read one or more scripts which define tests and to then run those tests and provide some sort of informative output describing whether the tests passed or failed. Different testing software may offer additional features or simpler syntax, but at their core, they all pretty much just evaluate assert expressions like `assert x == y`. I have opted to use pytest for the tests I am providing here as the output printed to your terminal is clearer than alternative such as the testing framework that is distributed as part of a Python installation, unittest. Note that I use unittest for the autograder so the tests I've provided are not identical (though they are very similar).

Installing and running pytest

You can install pytest using mamba (`mamba install pytest`) or pip (`pip install -U pytest`).

After installing pytest, you can run tests by invoking pytest in any directory containing properly configured tests. Pytest describes two options for setting up tests for a package. You can experiment with both setups if you like. Otherwise, you can just organize your directories as I did. I wrote the tests with the following directory layout:

```
$ tree
.
├── magnumopus
│   ├── __init__.py
│   └── sam.py
└── tests
    ├── __init__.py
    ├── helper_functions.py
    ├── test_data.py
    ├── test_imports.py
    └── test_read.py
```

using the above layout, running `pytest` runs all of the tests I defined in the tests directory. If all of the tests pass, you will see something like the following

```
$ pytest
=============================== test session starts ===============================
platform linux -- Python 3.12.3, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/alan/read_mapping_assignment
collected 43 items

tests/test_imports.py ...                                                    [  6%]
tests/test_read.py ........................................                  [100%]

=============================== 43 passed in 0.23s ================================
```

The first time you run these tests, it is likely they will fail for one reason or another. I have written the provided tests to print (hopefully) informative messages describing why a test failed. I did this using the `pytest.fail()` function to print custom failure messages and using functions I wrote in the helper_functions.py file.

Let's take a look at two different outputs you will see if your tests don't fail. The first example we'll discuss is a less than ideal error. The second is the sort of output that pytest is supposed to produce.

First, the less than ideal error: if you try running **all** the provided tests without there being a magnumopus directory for the test script to import from, you will see the following:

```
$ pytest
=============================== test session starts ===============================
platform linux -- Python 3.12.3, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/alan/read_mapping_assignment
collected 3 items / 1 error

===================================== ERRORS ======================================
_______________________ ERROR collecting tests/test_read.py _______________________
tests/test_read.py:16: in <module>
    from magnumopus.sam import Read
E   ModuleNotFoundError: No module named 'magnumopus'

During handling of the above exception, another exception occurred:
tests/test_read.py:18: in <module>
    pytest.fail(f"Can't perform remaining tests as imports failed")
E   Failed: Can't perform remaining tests as imports failed
============================= short test summary info =============================
ERROR tests/test_read.py - Failed: Can't perform remaining tests as imports failed
!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!!
================================ 1 error in 0.23s =================================
```

This error actually reflects a failure of pytest to run. I am not aware of test software that allows testing whether you are able to import packages that are also required for other tests. Therefore, I've contrived a somewhat workable message that quits the tests if the import fails. If you see this error message, then you should start with running just the import tests in tests/test_imports.py. You can run only the tests in a specific file with `pytest <filename>`. If you run just the tests in that file you should get some nice messages describing what can't be imported.

The second output you will see is the sort of output pytest is supposed to produce. Python tests are really supposed to assess whether your code does what it is expected to. If your `Read` class does not behave in any of the ways covered by the tests, you will see an output like this:

```
$ pytest
=============================== test session starts ===============================
platform linux -- Python 3.12.3, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/alan/read_mapping_assignment
collected 43 items

tests/test_imports.py ...                                                   [  6%]
tests/test_read.py ..................F.....................                 [100%]

==================================== FAILURES =====================================
_____________ TestReadAttrsBehavior.test_is_primary_with_supplemental _____________

E   AssertionError
All traceback entries are hidden. Pass `--full-trace` to see hidden and internal frames.

During handling of the above exception, another exception occurred:

self = <tests.test_read.TestReadAttrsBehavior object at 0x7fd02b9cbe60>

    def test_is_primary_with_supplemental(self):
        """Is the is_primary attribute set properly"""
>       check_attribute_value(
            instance=self.supplemental_read_instance,
            attr="is_primary",
            expected=False,
            tested_data="supplemental read mapping"
        )
E       Failed: Your Read.is_primary contains True for a supplemental read mapping, when it should
have contained False.

tests/test_read.py:154: Failed
============================= short test summary info =============================
FAILED tests/test_read.py::TestReadAttrsBehavior::test_is_primary_with_supplemental - Failed: Your
Read.is_primary contains True for a supplemental read mapping, wh...
========================== 1 failed, 42 passed in 0.26s ===========================
```

Let's walk through that output bit by bit.

First, there is a summary of your test environment and the tests performed. This section is indicated by the "test session starts" header. The important information here is how many dots you see (which indicate a test passed) and how many Fs you see (which indicate a test failed). The percent on the right just indicates the cumulative progress through all your tests that was the case after running the tests in each file. For example, test_imports.py contains 3 tests out of the 43 total. That's 6.9% of the tests. pytest then rounds this number down to 6.

Next is the failures section. This section shows the code executed in each failing test. It also includes a printed message describing what went wrong. In the above example, that message is "Failed: Your Read.is_primary contains True for a supplemental read mapping, when it should have contained False." If there was more than one test failure, you will see quite a lot of these traceback and failure messages. This can be a bit overwhelming so see below for a way to control this printout.

Finally, there is a short test summary info section. This section lists the failed tests as well as the printed error message. Note that if your terminal is not wide enough to fit the error message then it will clip as shown in the example. This section is a good place to look to get an overview of which tests are failing. You can then scroll up to see the full failure description in the failures section.

Especially when you first start writing, you may find that you have too many tests failing to easily deduce why each has failed. You have two options in this case. The best option is probably to comment out all the tests in the files I gave you and then uncomment and address them one by one (starting with the test_imports.py file and then working top to bottom in the test_read.py file). Alternatively, you can control the information in the pytest printout. For example, you can tell pytest to only print one line per failing test using `--tb=line`, where `--tb` controls how the traceback is printed. If we do that for the above example we get this much more manageable printout:

```
$ pytest --tb=line
=============================== test session starts ===============================
platform linux -- Python 3.12.3, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/alan/read_mapping_assignment
collected 43 items

tests/test_imports.py ...                                                   [  6%]
tests/test_read.py ..................F.....................                  [100%]

==================================== FAILURES =====================================
/home/alan/read_mapping_assignment/tests/test_read.py:154: Failed: Your Read.is_primary contains True
for a supplemental read mapping, when it should have contained False.
============================= short test summary info =============================
FAILED tests/test_read.py::TestReadAttrsBehavior::test_is_primary_with_supplemental - Failed: Your
Read.is_primary contains True for a supplemental read mapping, wh...
========================== 1 failed, 42 passed in 0.22s ============================
```

You can also suppress the header section with `-q` if you like.

If you only want to see the first N failures, you can set that with

```
pytest -x           # stop after first failure
pytest --maxfail=2  # stop after two failures
```

If you would like to see what the tests are doing you should explore the test scripts I provided. To understand the core approach of testing you may want to read the pytest Anatomy of a test primer.