>>>>>>>>>>>>>>>>>>>> constants.py <<<<<<<<<<<<<<<<<<<<
```
promoter_sequence = "AGGTTGGCAGTCAGTCAGCATCTACTGTTTGCAG"
terminator_sequence = "CGTCTGCTTTTGTCTCTGCTGCTGTCGTTT"
```

>>>>>>>>>>>>>>>>>>>> DNASequence.py <<<<<<<<<<<<<<<<<<<<
```
class DNASequence:
    def __init__(self, sequence: str):
        self._sequence = sequence

    def get_sequence(self) -> str:
        return self._sequence

    def __eq__(self, other):
        return self._sequence == other._sequence

    def get_rc(self) -> "DNASequence":
        """
        :return: a reverse-complemented sequence
        """
        complement = {'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A'}
        comp_sequence = [complement.get(base) for base in self._sequence]
        rev_comp_sequence = reversed(comp_sequence)
        return DNASequence("".join(rev_comp_sequence))

    def shift(self, amount: int):
        """
        Returns a new shifted DNA sequence
        :param amount: Amount of bases to shift
        """
        return DNASequence(self._sequence[amount:])

    def __str__(self):
        return self.get_sequence()

    def __repr__(self):
        return str(self)
```

>>>>>>>>>>>>>>>>>>>> ProteinSequence.py <<<<<<<<<<<<<<<<<<<<
```
from src.DNASequence import DNASequence

class ProteinSequence:
    @staticmethod
    def _translate(dna_sequence: str) -> str:
        translation_table = <SUPPRESSED>
        protein = ""  # TODO: string concatenation in python is inefficient. Find out why and how to improve.
        for i in range(0, len(dna_sequence), 3):
            codon = dna_sequence[i:i + 3]
            is_full_codon = len(codon) == 3
            if is_full_codon:
                protein += translation_table[codon]
        return protein

    def __init__(self, sequence: DNASequence):
        self._sequence = ProteinSequence._translate(sequence.get_sequence())

    def get_sequence(self) -> str:
        return self._sequence

    def get_first_pos_of_aminoacid(self, aminoacid: str) -> int:
        return self._sequence.find(aminoacid)

    def get_last_pos_of_aminoacid(self, aminoacid: str) -> int:
        return self._sequence.rfind(aminoacid)

    def trim(self, first_pos, last_pos):
        self._sequence = self._sequence[first_pos:last_pos]

    def get_count_of_aminoacid(self, aminoacid: str) -> int:
        return self._sequence.count(aminoacid)

    def __len__(self):
        return len(self._sequence)

    def __str__(self):
        return self._sequence

    def __repr__(self):
        return str(self)
```

>>>>>>>>>>>>>>>>>>> FASTAS<<<<<<<<<<<<<<<<<<<
```
from src.DNASequence import DNASequence

class FastaRecord:
    def __init__(self, comment: str, sequence: DNASequence):
        self._comment = comment
        self._sequence = sequence

    def is_plasmid(self) -> bool:
        return "plasmid" in self._comment or "plm" in self._comment

    def get_sequence(self) -> DNASequence:
        return self._sequence

    def get_comment(self) -> str:
        return self._comment

    def __str__(self):
        return f"{self.get_comment()}\n{self.get_sequence()}"

    def __repr__(self):
        return str(self)
```

```
from src.FastaRecord import FastaRecord
from src.DNASequence import DNASequence
from src.Plasmid import Plasmid
from pathlib import Path
from typing import List

# Note: untested, add unit tests
class FastaFile:
    def __init__(self, fasta_filepath: Path):
        # TODO: this class is inefficient as it stores all Fasta records in RAM, it might not be appropriate for real
        # tools. It can be efficiently implemented using context managers (see methods __enter__ and __exit__)
        with open(fasta_filepath) as fasta_file:
            all_lines = fasta_file.readlines()
            all_lines = list(map(str.strip, all_lines))
        self._all_records: List[FastaRecord] = []
        for comment, sequence in zip(all_lines[::2], all_lines[1::2]):
            record = FastaRecord(comment, DNASequence(sequence))
            self._all_records.append(record)

    def get_all_plasmids(self) -> List[Plasmid]:
        all_plasmids = []
        for record in self._all_records:
            if record.is_plasmid():
                plasmid = Plasmid(record)
                all_plasmids.append(plasmid)
        return all_plasmids

    def __repr__(self):
        return f"Fasta file with {len(self._all_records)} records"
```

```
>>>>>>>>>>>>>>> PLASMID <<<<<<<<<<<<<<<<<<<<<<<<<,
from src.FastaRecord import FastaRecord
from src.DNASequence import DNASequence
from src.Plasmid import Plasmid
from pathlib import Path
from typing import List

# Note: untested, add unit tests
class FastaFile:
    def __init__(self, fasta_filepath: Path):
        # TODO: this class is inefficient as it stores all Fasta records in RAM, it might not be appropriate for real
        # tools. It can be efficiently implemented using context managers (see methods __enter__ and __exit__)
        with open(fasta_filepath) as fasta_file:
            all_lines = fasta_file.readlines()
            all_lines = list(map(str.strip, all_lines))
        self._all_records: List[FastaRecord] = []
        for comment, sequence in zip(all_lines[::2], all_lines[1::2]):
            record = FastaRecord(comment, DNASequence(sequence))
            self._all_records.append(record)

    def get_all_plasmids(self) -> List[Plasmid]:
        all_plasmids = []
        for record in self._all_records:
            if record.is_plasmid():
                plasmid = Plasmid(record)
                all_plasmids.append(plasmid)
        return all_plasmids

    def __repr__(self):
        return f"Fasta file with {len(self._all_records)} records"
```

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> GENE <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```
from src.DNASequence import DNASequence
from src.Transcript import Transcript, NotAValidTranscript
from typing import List
from src.constants import promoter_sequence, terminator_sequence


class Gene:
    def __init__(self, plasmid: "Plasmid", start_pos: int, end_pos: int):
        self._plasmid = plasmid
        self._start_pos = start_pos
        self._end_pos = end_pos

    def get_coding_sequence(self) -> DNASequence:
        coding_sequence_start_pos = self._start_pos + len(promoter_sequence)
        coding_sequence_end_pos = self._end_pos - len(terminator_sequence)
        coding_sequence = DNASequence(self._plasmid.get_record().get_sequence().get_sequence()[
                                      coding_sequence_start_pos:coding_sequence_end_pos])

        return coding_sequence

    def __eq__(self, other):
        return (self._plasmid, self._start_pos, self._end_pos) == (other._plasmid, other._start_pos, other._end_pos)

    def __str__(self):
        return f"{self._plasmid}[{self._start_pos}:{self._end_pos}]"

    def __repr__(self):
        return str(self)

    def _get_transcripts(self) -> List[Transcript]:
        transcripts = []
        for reverse in [True, False]:
            for frame in [0, 1, 2]:
                try:
                    transcript = Transcript.build(self, reverse, frame)
                    transcripts.append(transcript)
                except NotAValidTranscript:
                    pass
        return transcripts

    def is_expressed(self) -> bool:
        transcripts = self._get_transcripts()
        return any(not transcript.has_PTC() for transcript in transcripts)
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> TRANSCRIPT <<<<<<<<<<<<<<<<<<<<<<<<<<
from src.ProteinSequence import ProteinSequence

class NotAValidTranscript(Exception):
    pass


class Transcript:
    def __init__(self, gene: "Gene", reverse: bool, frame: int):
        """
        Builds a transcript given a gene and a frame.
        :param gene: the gene itself
        :param reverse: if the transcript being built is from the forward or reverse strand of the gene
        :param frame: the gene frame (0, 1 or 2)
        """
        frame_is_valid = frame in [0, 1, 2]
        assert frame_is_valid, f"Error when building transcript, invalid frame: {frame}"

        if reverse:
            dna_sequence = gene.get_coding_sequence().get_rc()
        else:
            dna_sequence = gene.get_coding_sequence()
        dna_sequence = dna_sequence.shift(frame)
        self._protein_sequence = ProteinSequence(dna_sequence)

        first_start_codon_pos = self._protein_sequence.get_first_pos_of_aminoacid("M")
        last_stop_codon_pos = self._protein_sequence.get_last_pos_of_aminoacid("_")

        if Transcript._start_and_stop_codons_are_invalid(first_start_codon_pos, last_stop_codon_pos):
            raise NotAValidTranscript()

        self._protein_sequence.trim(first_start_codon_pos, last_stop_codon_pos + 1)

    @staticmethod
    def build(gene, reverse: bool, frame: int):
        return Transcript(gene, reverse, frame)

    @staticmethod
    def _start_and_stop_codons_are_invalid(start_codon_pos: int, stop_codon_pos: int) -> bool:
        there_is_no_start_codon = start_codon_pos == -1
        there_is_no_stop_codon = stop_codon_pos == -1
        start_codon_appears_after_stop_codon = start_codon_pos > stop_codon_pos
        return there_is_no_start_codon or there_is_no_stop_codon or start_codon_appears_after_stop_codon

    def has_PTC(self) -> bool:
        return self._protein_sequence.get_count_of_aminoacid("_") > 1

    def __str__(self):
        return str(self._protein_sequence)

    def __repr__(self):
        return str(self)
```

>>>>>>>>>>>>>>>>>>>>>>>>>>>> DNA SEQ TEST <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```
from unittest import TestCase
from src.DNASequence import DNASequence

class TestDNASequence(TestCase):
    def test___constructor___ACGT_sequence___ok(self):
        actual_sequence = DNASequence("ACGT")
        expected_sequence = "ACGT"

        self.assertEqual(expected_sequence, str(actual_sequence))

    def test___get_rc___empty_sequence(self):
        source = DNASequence("")
        actual = source.get_rc()
        expected = DNASequence("")
        self.assertEqual(actual, expected)

    def test___get_rc___one_base(self):
        source = DNASequence("A")
        actual = source.get_rc()
        expected = DNASequence("T")
        self.assertEqual(actual, expected)

    def test___get_rc___long(self):
        source = DNASequence("AGGGGTCACA")
        actual = source.get_rc()
        expected = DNASequence("TGTGACCCCT")
        self.assertEqual(actual, expected)

    def test___shift___zero_shift(self):
        source = DNASequence("ACGTTGCA")
        actual = source.shift(0)
        expected = DNASequence("ACGTTGCA")
        self.assertEqual(actual, expected)

    def test___shift___one_shift(self):
        source = DNASequence("ACGTTGCA")
        actual = source.shift(1)
        expected = DNASequence("CGTTGCA")
        self.assertEqual(actual, expected)

    def test___shift___two_shift(self):
        source = DNASequence("ACGTTGCA")
        actual = source.shift(2)
        expected = DNASequence("GTTGCA")
        self.assertEqual(actual, expected)

    def test___shift___six_shift(self):
        source = DNASequence("ACGTTGCA")
        actual = source.shift(6)
        expected = DNASequence("CA")
        self.assertEqual(actual, expected)
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PROTEIN SEQ TEST <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
from src.DNASequence import DNASequence
from src.ProteinSequence import ProteinSequence
from unittest.mock import patch
from unittest import TestCase

class TestProteinSequence(TestCase):
    def test___constructor___0_nts___0_aa(self):
        actual = ProteinSequence(DNASequence(""))
        expected = ""
        self.assertEqual(str(actual), expected)

    def test___constructor___1_nts___0_aa(self):
        actual = ProteinSequence(DNASequence("A"))
        expected = ""
        self.assertEqual(str(actual), expected)

    def test___constructor___2_nts___0_aa(self):
        actual = ProteinSequence(DNASequence("AC"))
        expected = ""
        self.assertEqual(str(actual), expected)

    def test___constructor___3_nts___1_aa(self):
        actual = ProteinSequence(DNASequence("ACT"))
        expected = "T"
        self.assertEqual(str(actual), expected)

    def test___constructor___4_nts___1_aa(self):
        actual = ProteinSequence(DNASequence("ACTG"))
        expected = "T"
        self.assertEqual(str(actual), expected)

    def test___constructor___5_nts___1_aa(self):
        actual = ProteinSequence(DNASequence("ACTGG"))
        expected = "T"
        self.assertEqual(str(actual), expected)

    def test___constructor___6_nts___2_aa(self):
        actual = ProteinSequence(DNASequence("ACTGGC"))
        expected = "TG"
        self.assertEqual(str(actual), expected)

    def test___constructor___complex(self):
        actual = ProteinSequence(DNASequence("TGGTATCGATCATTATCTAGTCAGTCGTAGTCAGCGATGCAT"))
        expected = "WYRSLSSQS_SAMH"
        self.assertEqual(str(actual), expected)

    @patch.object(ProteinSequence, "_translate", return_value="MPYYRPPP_")
    def test___get_first_pos_of_aminoacid___not_found(self, translate_mock):
        sequence = ProteinSequence(DNASequence("hello!"))
        actual_pos = sequence.get_first_pos_of_aminoacid("I")
        expected_pos = -1
        translate_mock.assert_called_once_with("hello!")
        self.assertEqual(actual_pos, expected_pos)

    @patch.object(ProteinSequence, "_translate", return_value="MPYYRPPP_")
    def test___get_first_pos_of_aminoacid___found(self, translate_mock):
        sequence = ProteinSequence(DNASequence(""))
        actual_pos = sequence.get_first_pos_of_aminoacid("P")
        expected_pos = 1
        self.assertEqual(actual_pos, expected_pos)

    @patch.object(ProteinSequence, "_translate", return_value="MPYYRPPP_")
```

```
def test___get_first_pos_of_aminoacid___last_aminoacid___found(self, translate_mock):
    sequence = ProteinSequence(DNASequence(""))
    actual_pos = sequence.get_first_pos_of_aminoacid("_")
    expected_pos = 8
    self.assertEqual(actual_pos, expected_pos)

@patch.object(ProteinSequence, "_translate", return_value="MPYYRPPP_")
def test___get_last_pos_of_aminoacid___not_found(self, translate_mock):
    sequence = ProteinSequence(DNASequence(""))
    actual_pos = sequence.get_last_pos_of_aminoacid("I")
    expected_pos = -1
    self.assertEqual(actual_pos, expected_pos)

@patch.object(ProteinSequence, "_translate", return_value="MPYYRPPP_")
def test___get_last_pos_of_aminoacid___found(self, translate_mock):
    sequence = ProteinSequence(DNASequence(""))
    actual_pos = sequence.get_last_pos_of_aminoacid("P")
    expected_pos = 7
    self.assertEqual(actual_pos, expected_pos)

@patch.object(ProteinSequence, "_translate", return_value="MPYYRPPP_")
def test___get_last_pos_of_aminoacid___first_aminoacid___found(self, translate_mock):
    sequence = ProteinSequence(DNASequence(""))
    actual_pos = sequence.get_last_pos_of_aminoacid("M")
    expected_pos = 0
    self.assertEqual(actual_pos, expected_pos)
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> GENE TEST <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
from unittest import TestCase
from unittest.mock import patch, call, Mock
from src.Gene import Gene
from src.Transcript import Transcript, NotAValidTranscript

class TestGene(TestCase):
    # could be done without the build method, by mocking Transcript.__new__, Transcript.__init__, and Gene.__eq__
    @patch.object(Transcript, "build", side_effect=["T1", "T2", "T3", "T4", "T5", "T6"])
    def test____get_transcripts___all_six_transcripts_are_valid(self, build_mock):
        dummy_gene = Gene("plasmid", 0, 0)
        transcripts = dummy_gene._get_transcripts()
        build_mock.assert_has_calls([
            call(dummy_gene, True, 0),
            call(dummy_gene, True, 1),
            call(dummy_gene, True, 2),
            call(dummy_gene, False, 0),
            call(dummy_gene, False, 1),
            call(dummy_gene, False, 2),
        ], any_order=True)
        self.assertEqual(transcripts, ["T1", "T2", "T3", "T4", "T5", "T6"])

    @patch.object(Transcript, "build", side_effect=["T1", NotAValidTranscript, NotAValidTranscript, "T4", "T5", NotAValidTranscript])
    def test____get_transcripts___3_valid_3_invalid(self, build_mock):
        dummy_gene = Gene("plasmid", 0, 0)
        transcripts = dummy_gene._get_transcripts()
        build_mock.assert_has_calls([
            call(dummy_gene, True, 0),
            call(dummy_gene, True, 1),
            call(dummy_gene, True, 2),
            call(dummy_gene, False, 0),
            call(dummy_gene, False, 1),
            call(dummy_gene, False, 2),
        ], any_order=True)
        self.assertEqual(transcripts, ["T1", "T4", "T5"])


    @patch.object(Gene, "_get_transcripts", return_value=[])
    def test___is_expressed___no_transcripts_are_valid___not_expressed(self, *uninteresting_mocks):
        dummy_gene = Gene("plasmid", 0, 0)
        self.assertFalse(dummy_gene.is_expressed())

    @patch.object(Gene, "_get_transcripts", return_value=[Mock(has_PTC=Mock(return_value=True)),
                                                          Mock(has_PTC=Mock(return_value=True)),
                                                          Mock(has_PTC=Mock(return_value=True))])
    def test___is_expressed___3_transcripts___all_have_PTC___not_expressed(self, *uninteresting_mocks):
        dummy_gene = Gene("plasmid", 0, 0)
        self.assertFalse(dummy_gene.is_expressed())

    @patch.object(Gene, "_get_transcripts", return_value=[Mock(has_PTC=Mock(return_value=True)),
                                                          Mock(has_PTC=Mock(return_value=True)),
                                                          Mock(has_PTC=Mock(return_value=False))])
    def test___is_expressed___3_transcripts___one_does_not_have_PTC___expressed(self, *uninteresting_mocks):
        dummy_gene = Gene("plasmid", 0, 0)
        self.assertTrue(dummy_gene.is_expressed())
```

>> PLASMID TEST <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```
from unittest import TestCase
from unittest.mock import patch
from src.FastaRecord import FastaRecord
from src.DNASequence import DNASequence
from src.Plasmid import Plasmid
from src.Gene import Gene
from src.constants import terminator_sequence

class TestPlasmid(TestCase):
    def setUp(self) -> None:
        self.record = FastaRecord(">record", DNASequence(

"ACCGTAGGTTGGCAGTCAGTCAGCATCTACTGTTTGCAGTTTTTCGTCTGCTTTTGTCTCTGCTGCTGTCGTTTAAAAA")
        )
        self.plasmid = Plasmid(self.record)

    def test___constructor(self):
        self.assertEqual(self.plasmid._record, self.record)

    def test____find_promoter___just_before_promoter___finds_promoter(self):
        actual_promoter_pos = self.plasmid._find_promoter(4)
        expected_promoter_pos = 5
        self.assertEqual(actual_promoter_pos, expected_promoter_pos)

    def test____find_promoter___just_on_promoter___finds_promoter(self):
        actual_promoter_pos = self.plasmid._find_promoter(5)
        expected_promoter_pos = 5
        self.assertEqual(actual_promoter_pos, expected_promoter_pos)

    def test____find_promoter___just_after_promoter___does_not_find_promoter(self):
        actual_promoter_pos = self.plasmid._find_promoter(6)
        expected_promoter_pos = -1
        self.assertEqual(actual_promoter_pos, expected_promoter_pos)

    def test____find_promoter___just_after_promoter___finds_next_promoter(self):
        record = FastaRecord(">record", DNASequence(

"ACCGTAGGTTGGCAGTCAGTCAGCATCTACTGTTTGCAGTTTTTCGTCTGCTTTTGTCTCTGCTGCTGTCGTTTAAAAAAGGT
TGGCAGTCAGTCAGCATCTACTGTTTGCAG")
                            )
        plasmid = Plasmid(record)
        actual_promoter_pos = plasmid._find_promoter(6)
        expected_promoter_pos = 79
        self.assertEqual(actual_promoter_pos, expected_promoter_pos)

    def test___get_genes___single_gene(self):
        actual_genes = self.plasmid.get_genes()
        expected_genes = [Gene(self.plasmid, 5, 74)]
        self.assertEqual(actual_genes, expected_genes)

    @patch.object(Plasmid, "_find_promoter", side_effect=[15, -1])
    @patch.object(Plasmid, "_find_terminator", side_effect=[500, -1])
    def test___get_genes___1_promoter_1_terminator___1_gene(self, *mocks):
        dummy_plasmid = Plasmid(None)
        actual_genes = dummy_plasmid.get_genes()
        expected_genes = [Gene(dummy_plasmid, 15, 500+len(terminator_sequence))]
        self.assertEqual(actual_genes, expected_genes)

    @patch.object(Plasmid, "_find_promoter", side_effect=[-1])
    @patch.object(Plasmid, "_find_terminator", side_effect=[-1])
    def test___get_genes___0_promoter_0_terminator___no_genes(self, *mocks):
        dummy_plasmid = Plasmid(None)
```

```
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = []
    self.assertEqual(actual_genes, expected_genes)

@patch.object(Plasmid, "_find_promoter", side_effect=[15, -1])
@patch.object(Plasmid, "_find_terminator", side_effect=[-1])
def test___get_genes___1_promoter_0_terminator___no_genes(self, *mocks):
    dummy_plasmid = Plasmid(None)
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = []
    self.assertEqual(actual_genes, expected_genes)

@patch.object(Plasmid, "_find_promoter", side_effect=[15, 100, -1])
@patch.object(Plasmid, "_find_terminator", side_effect=[-1])
def test___get_genes___2_promoter_0_terminator___no_genes(self, *mocks):
    dummy_plasmid = Plasmid(None)
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = []
    self.assertEqual(actual_genes, expected_genes)

@patch.object(Plasmid, "_find_promoter", side_effect=[-1])
@patch.object(Plasmid, "_find_terminator", side_effect=[50, -1])
def test___get_genes___0_promoter_1_terminator___no_genes(self, *mocks):
    dummy_plasmid = Plasmid(None)
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = []
    self.assertEqual(actual_genes, expected_genes)

@patch.object(Plasmid, "_find_promoter", side_effect=[-1])
@patch.object(Plasmid, "_find_terminator", side_effect=[50, 200, -1])
def test___get_genes___0_promoter_2_terminator___no_genes(self, *mocks):
    dummy_plasmid = Plasmid(None)
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = []
    self.assertEqual(actual_genes, expected_genes)

@patch.object(Plasmid, "_find_promoter", side_effect=[15, 50, -1])
@patch.object(Plasmid, "_find_terminator", side_effect=[500, -1])
def test___get_genes___2_promoter_1_terminator___case_1___1_gene(self, *mocks):
    dummy_plasmid = Plasmid(None)
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = [Gene(dummy_plasmid, 15, 500 + len(terminator_sequence))]
    self.assertEqual(actual_genes, expected_genes)

@patch.object(Plasmid, "_find_promoter", side_effect=[15, 1000, -1])
@patch.object(Plasmid, "_find_terminator", side_effect=[500, -1])
def test___get_genes___2_promoter_1_terminator___case_2___1_gene(self, *mocks):
    dummy_plasmid = Plasmid(None)
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = [Gene(dummy_plasmid, 15, 500 + len(terminator_sequence))]
    self.assertEqual(actual_genes, expected_genes)

@patch.object(Plasmid, "_find_promoter", side_effect=[15, -1])
@patch.object(Plasmid, "_find_terminator", side_effect=[500, 1000, -1])
def test___get_genes___1_promoter_2_terminator___case_1___1_gene(self, *mocks):
    dummy_plasmid = Plasmid(None)
    actual_genes = dummy_plasmid.get_genes()
    expected_genes = [Gene(dummy_plasmid, 15, 500 + len(terminator_sequence))]
    self.assertEqual(actual_genes, expected_genes)
```