Natural language text to python code transpiler

Pablo Munoz A01222422, Francisco Castellanos A01261268

November 25, 2018

Contents

1 Introduction

Natural languages are unlike programming languages due to the fact that they cannot be easily characterized with rules. In addition, natural languages contain a lot of ambiguity and nuance. However, us humans have been using natural language to communicate and convey meaning and emotions very effectively for thousands of years. Wouldn't it be nice if we could just talk as we talk to another person and have a computer understand us and do what we tell it? This text documents a python program that does a very simple version of that, it converts human text (with some restrictions) into code for python classes. The problem turned out to be much tougher than we originally thought, and we never thought it would be easy. So while the program might not be anything noteworthy, it does shed some light into the difficulties that any engineer will face when making a bridge between natural and computer languages.

2 Limitations

1. Every paragraph will be a new class. And if in the source file contains the same paragraph there would be two identical classes.
2. Class properties must be declared together in a sentence after the class declaration sentence, separated by commas. The sentence must begin with a pronoun and then the word "can".
3. We discovered that it was really difficult to determine where a sentence ends. Initially we relied on periods, however, periods may appear between numbers or in property accessing expressions like `self.var`. So we now require that method definition sentences must end with the tokens: `, end.`
4. We only support methods with 0 or 1 parameters.
5. We do not support literal strings in the program with commas inside of them, for example "bark, bark" would lead to an incorrect output, the string must be "bark bark".
6. Our transpiler does not check the validity of the program, so referencing a variable that is not declared or other errors of the sort will go through.
7. Only class attributes can be assigned in the function bodies (no globals).
8. The function bodies can only be composed of the actions:
   - Increase a class attribute
   - Decrease a class attribute
   - Set a class attribute to some expression
   - Print some expression
   - Return something
9. The attribute declaration/initialization sentence must end with a period, and if the last value to initialize is a number, a space must lie between the number and the period.
10. The program will transform all attribute and method names to lowercase, except for the class name which will have the first letter capitalized. This does not affect the correctness of the program unless you were having identically named attributes or methods but with different casing.
11. The process of producing code insert spaces between "tokens". This does not affect the correctness of the program as `call(x, y)` is the same as `call ( x , y )` to the interpreter. However, strings are affected as there will be an extra space after the first string quote and before the last string quote.

3 Anatomy of a valid input

The range of valid inputs that our program can parse turns out to be quite limit as they must follow a particular structure.

As mentioned before each class must be defined in one paragraph and one paragraph can only define one class. The first sentence of the paragraph will indicate the class name. The first sentence must end with `is/are/am a class`. The class name will be taken to be the noun closest to the word `class`.

After the class definition sentence, there must be an attribute definition sentence with the form `<pronoun> has <attr> = <value>`. The attributes and their values must be separated by commas and if the last value is a number, there must be a space between it and the following period.

After the attributes definition sentence, there is the attribute declaration sentence, which must have structure `<pronoun> can <method-1-name>, <method-2-name>, ..., and <method-n-name>`.

After the method declaration sentence, there must be one sentence per method (which will define it). The rest of the paragraph must consist only of the method definition sentences. All method definition sentences must begin with `To <method name>`. If the method accepts an argument this must be followed by a `<pronoun> need(s) <arg`

`name>`, then we will have each of the actions that the method will take, separated by commas and culiminating in a `, end.`. The actions can be the following:

To increase an attribute: `<pronoun> <attr name> increase(s) in/by <amount>` To decrease an attribute: `<pronoun> <attr name> decrease(s) in/by <amount>` To print something: `<pronoun> print(s) <value>` To set an attribute: `<pronoun> <attr name> is <value>` To return something: `return <value>`

4 Code

4.1 The `main` program

Our main program will consist of a command line interface that takes as input the path to a text file containing natural language and outputs the program constructed from it to stdout. The program accepts two flags, `-i` or `--input` whose value must be equal to the input file with natural language, this flab is required. You can also give the `-o` or `--output` whose value is a path where the resulting python program will be written to; if you omit this flag, the code will simply be send to stdout.

```
import sys
import argparse
import logging
import os

logging_level = logging.CRITICAL

if os.environ.get('DEBUG'):
    logging_level = logging.DEBUG

logging.basicConfig(level=logging_level)
logger = logging.getLogger(__file__)

from transpiler import Transpiler

if __name__ == '__main__':
    argument_parser = argparse.ArgumentParser()

    argument_parser.add_argument(
        '-i', '--input', type=str,
        help='Path to text file with program description in natural language')
    argument_parser.add_argument(
        '-o', '--output', type=str,
        help='Path of text file to write generated code',
        required=False, default=None
    )

    args = argument_parser.parse_args(sys.argv[1:])

    input_ = args.input
    logger.debug('input: {}'.format(input_))

    with open(input_, 'r') as fhandle:
        text = fhandle.read()

    transpiler = Transpiler()
    code = transpiler.transpile(text)
    if args.output:
        with open(args.output, 'w') as fhandle:
            fhandle.write(code)
    else:
        print(code)
```

4.2 class Transpiler

Although not perfectly accurate for our task, we take the liberty of calling the process of taking a text in natural language and converting it into code **transpilation**.

4.2.1 split_paragraphs(self, text)

One of the constraints of our system is that each class must be defined in a paragraph of natural language text. We can use this to our advantage by splitting the paragraphs into separate strings, this way we can focus on one at a time in other methods.

```
def split_paragraphs(self, text):
    '''
    Returns a list of strings. Each strings is a paragraph in the
    given text.

    Arguments:
    text -- string assumed to contain one or more paragraphs, where a
      paragraph is defined as consecutive lines, i.e. two consecutive
      line breaks demarcate a paragraph.

    Usage:
    >>> two_paragraphs_together = """this is the
    ... first paragraph
    ...
    ... and this is the
    ... second
    ... """
    >>> transpiler = Transpiler()
    >>> separated_paragraphs = transpiler.split_paragraphs(two_paragraphs_together)
    >>> print(separated_paragraphs[0])
    this is the
    first paragraph
    >>> print(separated_paragraphs[1])
    and this is the
    second
    '''
    return list(map(lambda s: s.strip('\n'), paragraph_regex.split(text)))
```

4.2.2 parse_class_paragraph(self, paragraph)

This method takes a paragraph of the natural language program as a string and parses it. By parsing we mean it is transformed into an internal representation (a dictionary in this case), that other methods can use to produce a code representation. For the purposes of this system, the `paragraph` will always be a string that has been generated by the `split_paragraphs` method.

The process of how we parse a paragraph string into its internal dictionary representation can be split into steps:

1. Lowercase paragraph
2. Tokenizing words
3. POS-tagging (part-of-speech tagging)
4. Parsing class name
5. Parsing attributes
6. Parsing methods

```python
def parse_class_paragraph(self, paragraph):
    '''Returns a dictionary that contains an internal representation of
    a python class given as a natural language string *paragraph*.

    Arguments:
    paragraph -- str, Natural language string describing a python class.

    Usage:
    >>> paragraph = """
    ... A dog is a Class. He has mood = "HAPPY", energy = 100,
    ... x = 0, y = 0 . He can Bark, Run, MoveLeft, MoveRight,
    ... MoveForward, Lay and Check. To Run he used MoveForward(2), his energy
    ... decreases in 1, his mood is "PLAY" and return 0, end. To MoveForward he
    ... needs numbersSteps, his y increases in
    ... numbersSteps, his mood is "MOVING", his energy decreases by 1, end. To MoveLeft
    ... he needs numbersSteps, his x decreases in
    ... numbersSteps, his mood is "MOVING", his energy decreases by 1, end. To
    ... MoveRight he needs numbersSteps, his x increases
    ... in numbersSteps, his mood is "MOVING", his energy decreases by 1, end. To Bark
    ... he print "barf barf", his energy decreases by 1, his mood is
    ... "BARKING", end. To Lay he used print "relax", he used print "move the
    ... Booty", his energy increases in 3", end. To Check he print "mood: " +
    ... self.mood, he print "energy: " + str(self.energy), he print "Position" +
    ... str(self.coordinatePosition), end.
    ... """
    >>> transpiler = Transpiler()
    >>> class_metadata = transpiler.parse_class_paragraph(paragraph)
    >>> class_metadata['class_name']
    'dog'
    >>> class_metadata['property_names_and_defaults']
    [('mood', '"happy"'), ('energy', '100'), ('x', '0'), ('y', '0')]
    >>> class_metadata['method_names']
    ['bark', 'run', 'moveleft', 'moveright', 'moveforward', 'lay', 'check']
    '''
    # 1. Lowercase paragraph
    lowercase_paragraph = paragraph.lower()
    # 2. Tokenize words
    word_tokens = self.tokenize_words(lowercase_paragraph)
    # 3. POS-tagging
    tagged_word_tokens = self.part_of_speech_tag(word_tokens)

    class_name = None
    class_keyword_index = word_tokens.index('class')
```

```
first_noun_before_class_keyword = next(filter(
    lambda pair: pair[1] == 'NN',
    tagged_word_tokens[class_keyword_index-1:0:-1]))[0]
class_name = first_noun_before_class_keyword


property_names_and_defaults = []
index_of_first_word_second_sentence = (
    class_keyword_index +
    1 + # Because of the dot that finalizes the class declaration
    1 # the word after the dot
)

i = 0
j = index_of_first_word_second_sentence
while True:
    next_word = word_tokens[j + i]
    if next_word == '.':
        break

    if next_word == '=':
        # The property name is the word before the = symbol
        property_name = word_tokens[j + i - 1]

        # The property default value is the concatenations of all
        # the words/tokens after the = symbol and before the next
        # immediate comma or period.
        property_default_value = ''

        i += 1
        next_word = word_tokens[j + i]

        while next_word not in ('.', ','):
            property_default_value += next_word
            i += 1
            next_word = word_tokens[j + i]

        # Something weird happens where string value, i.e. with double quotes
        # are parsed as beginning with 2 of this symbol: '`'. If that is the
        # case, coerce to using the double quote.
        value_with_funny_double_quote = property_default_value[0] == '`'
        if value_with_funny_double_quote:
            property_default_value = '"{}"'.format(property_default_value[2:-2])

        property_names_and_defaults.append((property_name, property_default_value))

        i -= 1

    else:
        i += 1

    index_of_last_word_second_sentence = j + i
```

```
method_names = []
method_params = {}
method_actions = {}
# Step 1. Parse the method names.
index_of_first_word_third_sentence = index_of_last_word_second_sentence + 1
j = index_of_first_word_third_sentence

# Skip two tokens (pronoun and can)
j += 2

i = 0

while True:
    token = word_tokens[j+i]
    if token == '.':
        break

    if token not in (',', 'and'):
        method_names.append(token)

    i += 1

# Step 2. Parse the method arguments
# Each method has its arguments and actions defined in a single
# sentence that must appear after the method names declaration
# sentence.
# From this point on, all remaining tokens will be method definition
# tokens, and then the paragraph must end.
method_definition_tokens = word_tokens[j+i+1:]
method_definition_sentences = []

beginning_of_sentence = 0
end_of_sentence = method_definition_tokens.index('end', beginning_of_sentence)
while end_of_sentence > 0:
    method_definition_sentences.append(method_definition_tokens[
        beginning_of_sentence:end_of_sentence-1])
    beginning_of_sentence = end_of_sentence + 2

    try:
        end_of_sentence = method_definition_tokens.index('end', beginning_of_sentence)
    except ValueError:
        break

for sentence in method_definition_sentences:
    # sentence[0] must be the word "To", so we start
    # at 1 to look for the method name
    i = 1
    method_name = sentence[i]

    # Check if the method has a parameter
    method_has_param = 'need' in sentence[i+2]
    if method_has_param:
        param_name = sentence[i+3]
        method_params[method_name] = param_name

        i += 4

    actions = []

    j = i + 1
    while True:
        next_comma = math.inf
        try:
            next_comma = sentence.index(',', j)
        except ValueError:
            pass

        next_and = math.inf
        try:
            next_and = sentence.index('and', j)
        except ValueError:
            pass

        action_end = min(next_comma, next_and)
        if action_end == math.inf:
            action_end = None

        next_action = slice(j,action_end)

        action_tokens = sentence[next_action]

        is_print_action = len(action_tokens) > 1 and action_tokens[1].find('print') == 0
        is_increase_action = len(action_tokens) > 2 and action_tokens[2].find('increase') == 0
        is_decrease_action = len(action_tokens) > 2 and action_tokens[2].find('decrease') == 0
        is_assign_action = len(action_tokens) > 2 and action_tokens[2].find('is') == 0
        is_return_action = len(action_tokens) > 1 and action_tokens[0].find('return') == 0
        is_call_action = len(action_tokens) > 1 and action_tokens[1].find('use') == 0

        if is_print_action:
            actions.append(
                ('print', action_tokens[2:])
            )
        elif is_increase_action:
            actions.append(
                ('increase', [action_tokens[1]] + action_tokens[4:])
            )
        elif is_decrease_action:
            actions.append(
                ('decrease', [action_tokens[1]] + action_tokens[4:])
            )
        elif is_assign_action:
            actions.append(
                ('assign', [action_tokens[1]] + action_tokens[3:])
            )
        elif is_return_action:
            actions.append(
                ('return', action_tokens[1:])
            )
        elif is_call_action:
            actions.append(
                ('call', action_tokens[2:])
            )

        if action_end is None:
            break

        j = action_end + 1

    method_actions[method_name] = actions


return dict(
    class_name=class_name,
    property_names_and_defaults=property_names_and_defaults,
    method_names=method_names,
    method_params=method_params,
    method_actions=method_actions
)
```

To parse the classname we exploit the following restriction of the system: /A class is declared as a sentence whose last two elements are the word 'class' and a period. The class declaring sentence must be the first sentence in a class paragraph/

The strategy we will employ is to find the first period within the paragraph `tagged_word_tokens`, then look back for the first noun (tag 'NN') we encounter, that will be the name of the class.

```
class_keyword_index = word_tokens.index('class')
first_noun_before_class_keyword = next(filter(
    lambda pair: pair[1] == 'NN',
    tagged_word_tokens[class_keyword_index-1:0:-1]))[0]
class_name = first_noun_before_class_keyword
```

In order to parse the property names and default values of a class we make use of another constraint of the system and parse the second sentence of the paragraph since: /Class properties must be declared together in a sentence after the class declaration sentence, the declarations must be of the form <property name> = <default value>, where the default values can be any valid python expression. The properties declarations must be separated by commas/.

```
index_of_first_word_second_sentence = (
    class_keyword_index +
    1 + # Because of the dot that finalizes the class declaration
    1 # the word after the dot
)

i = 0
j = index_of_first_word_second_sentence
while True:
    next_word = word_tokens[j + i]
    if next_word == '.':
        break
    if next_word == '=':
        # The property name is the word before the = symbol
        property_name = word_tokens[j + i - 1]

        # The property default value is the concatenations of all
        # the words/tokens after the = symbol and before the next
        # immediate comma or period.
        property_default_value = ''

        i += 1
        next_word = word_tokens[j + i]

        while next_word not in ('.', ','):
            property_default_value += next_word
            i += 1
            next_word = word_tokens[j + i]

        # Something weird happens where string value, i.e. with double quotes
        # are parsed as beginning with 2 of this symbol: '`'. If that is the
        # case, coerce to using the double quote.
        value_with_funny_double_quote = property_default_value[0] == '`'
        if value_with_funny_double_quote:
            property_default_value = '"{}"'.format(property_default_value[2:-2])

        property_names_and_defaults.append((property_name, property_default_value))

        i -= 1

    else:
        i += 1

    index_of_last_word_second_sentence = j + i
```

Parsing the methods is perhaps the most complex task of the program. We leverage the fact that we have constrained the system so that the methods always come after the class name declarations and the class attributes and default values declarations. The methods are first all declared together in one sentence of the form

For example

```
He can Run, Hide, Play, Eat.
```

```
# Step 1. Parse the method names.
index_of_first_word_third_sentence = index_of_last_word_second_sentence + 1
j = index_of_first_word_third_sentence

# Skip two tokens (pronoun and can)
j += 2

i = 0

while True:
    token = word_tokens[j+i]
    if token == '.':
        break
    if token not in (',', 'and'):
        method_names.append(token)

    i += 1

# Step 2. Parse the method arguments
# Each method has its arguments and actions defined in a single
# sentence that must appear after the method names declaration
# sentence.
# From this point on, all remaining tokens will be method definition
# tokens, and then the paragraph must end.
method_definition_tokens = word_tokens[j+i+1:]
method_definition_sentences = []

beginning_of_sentence = 0
end_of_sentence = method_definition_tokens.index('end', beginning_of_sentence)
while end_of_sentence > 0:
    method_definition_sentences.append(method_definition_tokens[
        beginning_of_sentence:end_of_sentence-1])
    beginning_of_sentence = end_of_sentence + 2

    try:
        end_of_sentence = method_definition_tokens.index('end', beginning_of_sentence)
    except ValueError:
        break

for sentence in method_definition_sentences:
    # sentence[0] must be the word "To", so we start
    # at 1 to look for the method name
    i = 1
    method_name = sentence[i]

    # Check if the method has a parameter
    method_has_param = 'need' in sentence[i+2]
    if method_has_param:
        param_name = sentence[i+3]

        method_params[method_name] = param_name

        i += 4

    actions = []

    j = i + 1
    while True:
        next_comma = math.inf
        try:
            next_comma = sentence.index(',', j)
        except ValueError:
            pass

        next_and = math.inf
```

```python
        try:
            next_and = sentence.index('and', j)
        except ValueError:
            pass

        action_end = min(next_comma, next_and)
        if action_end == math.inf:
            action_end = None

        next_action = slice(j,action_end)

        action_tokens = sentence[next_action]

        is_print_action = len(action_tokens) > 1 and action_tokens[1].find('print') == 0
        is_increase_action = len(action_tokens) > 2 and action_tokens[2].find('increase') == 0
        is_decrease_action = len(action_tokens) > 2 and action_tokens[2].find('decrease') == 0
        is_assign_action = len(action_tokens) > 2 and action_tokens[2].find('is') == 0
        is_return_action = len(action_tokens) > 1 and action_tokens[0].find('return') == 0
        is_call_action = len(action_tokens) > 1 and action_tokens[1].find('use') == 0

        if is_print_action:
            actions.append(
                ('print', action_tokens[2:])
            )
        elif is_increase_action:
            actions.append(
                ('increase', [action_tokens[1]] + action_tokens[4:])
            )
        elif is_decrease_action:
            actions.append(
                ('decrease', [action_tokens[1]] + action_tokens[4:])
            )
        elif is_assign_action:
            actions.append(
                ('assign', [action_tokens[1]] + action_tokens[3:])
            )
        elif is_return_action:
            actions.append(
                ('return', action_tokens[1:])
            )
        elif is_call_action:
            actions.append(
                ('call', action_tokens[2:])
            )

        if action_end is None:
            break

        j = action_end + 1

    method_actions[method_name] = actions
```

4.2.3 tokenize_words(self, string)

Wrapper around `nltk`'s `word_tokenize`.

```
def tokenize_words(self, string):
    '''Returns a list of the words contained in string which is assumed
    to be a sentence.

    Arguments:
    string -- string represeting a sentence

    Usage:
    >>> transpiler = Transpiler()
    >>> transpiler.tokenize_words('a dog is a class.')
    ['a', 'dog', 'is', 'a', 'class', '.']
    '''
    return [ '"' if w in ('``', '\'\'') else w for w in nltk.word_tokenize(string) ]
```

4.2.4 part_of_speech_tag(self, tokens)

```
def part_of_speech_tag(self, tokens):
    '''Returns a list of (word, tag) tuples for each word in tokens.

    Tags are strings that represent the role that a word takes in a text.
    For example a tag of 'NN' means the word is a noun, a tag of 'VBZ'
    means a verb, present tense, 3rd person singular. To know what a
    particular tag means you can run the following code:

    nltk.help.upenn_tagset('<TAG>')

    Arguments:
    tokens -- list of words (str)

    Usage:
    >>> transpiler = Transpiler()
    >>> tokens = transpiler.tokenize_words('a dog is a class')
    >>> transpiler.part_of_speech_tag(tokens)
    [('a', 'DT'), ('dog', 'NN'), ('is', 'VBZ'), ('a', 'DT'), ('class', 'NN')]
    '''
    return nltk.pos_tag(tokens)
```

4.2.5 produce_class_code(self, class_metadata)

This method produces the code for a class represented by a dictionary of class attributes and methods given as the `class_metadata` parameter.

```
def produce_class_code(self, class_metadata):
    '''
    Usage:
    >>> paragraph = """
    ... A dog is a Class.
    ... He has mood = "HAPPY", energy = 100, x = 0, y = 0 .
    ... He can Bark, Run, MoveLeft, MoveRight, MoveForward, Lay and
    ... Check.
    ... To Run he used MoveForward(2), his energy decreases in 1,
    ... his mood is "PLAY" and return 0, end.
    ... To MoveForward he needs numbersSteps, his y increases in
    ... numbersSteps, his mood is "MOVING", his energy decreases by 1, end.
    ... To MoveLeft he needs numbersSteps, his x decreases in
    ... numbersSteps, his mood is "MOVING", his energy decreases by 1, end.
    ... To MoveRight he needs numbersSteps, his x increases in numbersSteps,
    ... his mood is "MOVING", his energy decreases by 1, end.
    ... To Bark he print "barf", his energy decreases in 1, his mood is
    ... "BARKING", end.
    ... To Lay he print "relax", he print "move the
    ... Booty", his energy increases in 3, end.
    ... To Check he print "mood: " + self.mood, he print "energy: " +
    ... str(self.energy), print "Position" + str(self.coordinatePosition), end.
    ... """
    >>> transpiler = Transpiler()
    >>> class_metadata = transpiler.parse_class_paragraph(paragraph)
    >>> code = transpiler.produce_class_code(class_metadata)
    >>> print(code)
    class Dog:
        mood = "happy"
        energy = 100
        x = 0
        y = 0
    <BLANKLINE>
        def bark(self):
            print(" barf ")
            self.energy -= 1
            self.mood = " barking "
    <BLANKLINE>
        def run(self):
            self.moveforward ( 2 )
            self.energy -= 1
            self.mood = " play "
            return 0
    <BLANKLINE>
        def moveleft(self, numberssteps):
            self.x -= numberssteps
            self.mood = " moving "
            self.energy -= 1
    <BLANKLINE>
        def moveright(self, numberssteps):
            self.x += numberssteps
            self.mood = " moving "
            self.energy -= 1
    <BLANKLINE>
        def moveforward(self, numberssteps):
            self.y += numberssteps
            self.mood = " moving "
            self.energy -= 1
    <BLANKLINE>
        def lay(self):
            print(" relax ")
            print(" move the booty ")
            self.energy += 3
    <BLANKLINE>
        def check(self):
            print(" mood : " + self.mood)
            print(" energy : " + str ( self.energy ))
    <BLANKLINE>
    <BLANKLINE>
    '''
    class_code_str = ''
    # Use string.capwords to follow python class naming convetion
    class_code_str += 'class {}:'.format(
        string.capwords(class_metadata['class_name']))
    class_code_str += '\n'

    for name, value in class_metadata['property_names_and_defaults']:
        class_code_str += '    {name} = {value}'.format(name=name, value=value)
        class_code_str += '\n'

    class_code_str += '\n'

    for method_name in class_metadata['method_names']:
        param_name = class_metadata['method_params'].get(method_name)
        if param_name:
            class_code_str += '    def {method_name}(self, {param_name}):\n'.format(
                method_name=method_name, param_name=param_name)
            pass
        else:
            class_code_str += '    def {method_name}(self):\n'.format(
                method_name=method_name)

        for action in class_metadata['method_actions'].get(method_name, []):
            action_name, action_value = action[0], action[1]

            if action_name == 'print':
                print_value = ' '.join(action_value)
                class_code_str += '        print({})\n'.format(print_value)
            elif action_name == 'increase':
                var = action_value[0]
                amount = ' '.join(action_value[1:])
                class_code_str += '        self.{var} += {amount}\n'.format(
                    var=var, amount=amount
                )
            elif action_name == 'decrease':
                var = action_value[0]
                amount = ' '.join(action_value[1:])
                class_code_str += '        self.{var} -= {amount}\n'.format(
                    var=var, amount=amount
                )
            elif action_name == 'assign':
                var = action_value[0]
                value = ' '.join(action_value[1:])
                class_code_str += '        self.{var} = {value}\n'.format(
                    var=var, value=value
                )
            elif action_name == 'return':
                return_value = ' '.join(action_value)
                class_code_str += '        return {return_value}\n'.format(
                    return_value=return_value
                )
            elif action_name == 'call':
                class_code_str += '        self.{}\n'.format(' '.join(action_value))

        class_code_str += '\n'

    return class_code_str
```

4.2.6 transpile(text)

```
def transpile(self, text):
    code = ''
    separated_paragraphs = self.split_paragraphs(text)

    for paragraph in separated_paragraphs:
        class_metadata = self.parse_class_paragraph(paragraph)
        code += self.produce_class_code(class_metadata)

    return code
```

5 Example

This document is acompannied by a file `example01.txt` in the `data/` directory. For reference the example looks like this:

```
A dog is a Class.
He has mood = "HAPPY", energy = 100, x = 0, y = 0 .
He can Bark, Run, MoveLeft, MoveRight, MoveForward, Lay and
Check.
To Run he used MoveForward(2), his energy decreases in 1,
mood is "PLAY" and return 0, end.
To MoveForward he needs numbersSteps, his y increases in
his mood is "MOVING", his energy decreases by 1, end.
To MoveLeft he needs numbersSteps, his x decreases in
his mood is "MOVING", his energy decreases by 1, end.
To MoveRight he needs numbersSteps, his x increases in numbersSteps,
mood is "MOVING", his energy decreases by 1, end.
To Bark he print "barf", his energy decreases in 1, his mood is
end.
To Lay he print "relax", he print "move the
his energy increases in 3, end.
To Check he print "mood: " + self.mood, he print "energy: " +
print "Position" + str(self.coordinatePosition), end.

A pack is a class.
They have mood = "HAPPY", energy = 1000, members = 10 .
They can play, sleep, join and separate.
To play they need duration, their energy decreases by 10, their mood is "PLAYING", end.
To sleep they need duration, their energy increases by 20, their mood is "SLEEPING", end.
To join they need numMembers, their members increase by numMembers, they print str(numMembers) + " joi
To separate they need numMembers, their members decrease by numMmembers, they print str(numMembers) + "
```

The above natural language can be converted into python code by our program by executing the following in your shell:

```
python src/main.py -i data/example01.txt
```

which outputs the following:

```python
class Dog:
    mood = "happy"
    energy = 100
    x = 0
    y = 0

    def bark(self):
        print(" barf ")
        self.energy -= 1

    def run(self):
        self.moveforward ( 2 )
        self.energy -= 1
        return 0

    def moveleft(self, numberssteps):
        self.x -= his mood is " moving "
        self.energy -= 1

    def moveright(self, numberssteps):
        self.x += numberssteps
        self.energy -= 1

    def moveforward(self, numberssteps):
        self.y += his mood is " moving "
        self.energy -= 1

    def lay(self):
        print(" relax ")
        print(" move the his energy increases in 3")

    def check(self):
        print(" mood : " + self.mood)
        print(" energy : " + print " position " + str ( self.coordinateposition ))

class Pack:
    mood = "happy"
    energy = 1000
    members = 10

    def play(self, duration):
        self.energy -= 10
        self.mood = " playing "
    def sleep(self, duration):
        self.energy += 20
        self.mood = " sleeping "

    def join(self, nummembers):
        self.members += nummembers
        print(str ( nummembers ) + " joined the pack ")

    def separate(self, nummembers):
        self.members -= nummmembers
        print(str ( nummembers ) + " left the pack ")
```

6 Testing

The code presented contains many unit tests which you can run by executing the following command:

```
python -m doctest src/main.py src/transpiler.py
```

or

```
python -m doctest -v src/main.py src/transpiler.py
```

if you want a more verbose output.

7 Lessons learned

7.1 Stemming is your friend

We found stemming really useful, particularly for having our program accept the singular and plural version of words. For example, in a method declaration sentence you can order to increase some attribute by some quantity. But in our natural language the way you express this will differ depending on whether the class is a "singular" o a "plural". So to our program it is the same that you say `His energy increases by 1` or `Their memebers increase by 10`. Even though we have the word increase with and without a final `s`, the meaning is still the same, there is some quantity that must be increased.

7.2 Delimiting is hard

In the beginning we attempted to use periods as the delimiters of sentences. This turned out to be very fragile because periods appear with other purposes in human text (this goes back to the ambiguities) of human languages. For example, a number could have a period like in `1.3`, or the human could say something like `print self.id`, and we find ourselves with another period that is not the end of a sentence. To overcome this we had to add another restriction to the inputs our program accepts at the cost of its utility and generality. We now require that each sentence in the method definition sections of a paragraph end with the word tokens `, end.`

7.3 Can't rely just on noun/verbs

Since we are using NLTK, and its tagging capabilities allow us to determine if a word is a noun or a verb, we originally had the idea to look at all the verbs and take them to be methods, and look at all the nouns and have them be classnames if they appear in the first, sentence of the paragraph or attributes if they appear later. However, this proved to not be very effective due to the fact that when programming we often name methods something like `moveLeft`, and while `move` is correctly identified as a verb by `nltk`, `moveLeft` is not.

7.4 It's very hard to not rely on position

It turns out to be very hard to extract meaning from a human text and structure it, so we ended up relying a lot in the position of words, for example we expect that the first word of a method definition sentence to be `To`. While this simplifies our job it restricts the universe of human texts that our program can parse, so while `To run...` would have the same meaning as `In order to run...` the latter would not work for our system.