Natural language text to python code transpiler

Pablo Munoz 0181

November 24, 2018

Contents

1 Requirements

1. Every paragraph will be a new class. And if in the source file contains the same paragraph there would be two identical classes, this would ont be an error.
2.
3. Class properties must be declared together in a sentence after the class declaration sentence, separated by commas
4. Method definition sentences must end with the tokens: `, end.`
5. We only support methods with 0 or 1 parameters.

2 Code

2.1 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.

```
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')

    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()
    transpiler.transpile(text)
```

2.2 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**.

2.2.1 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)))
```

2.2.2 `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 TODO steps:

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

```
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,
    ... coordenatePosition = (0,0). 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 coordinatePosition[0] increases in
    ... numbersSteps, his mood is "MOVING", decreases energy by 1, end. To MoveLeft
    ... he needs numbersSteps, his coordinatePosition[1] decreases in
    ... numbersSteps, his mood is "MOVING", decreases energy by 1, end. To
    ... MoveRight he needs numbersSteps, his coordinatePosition[1] increases
    ... in numbersSteps, his mood is "MOVING", decreases energy by 1, end. To Bark
    ... he print "barf, barf", his energy decreases in 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), 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'), ('coordenateposition', '(0,0)')]
    >>> 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

    #print(method_definition_sentences)

    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

    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

```
<pronoun> can <method-names>*
```

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

#print(method_definition_sentences)

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
```

2.2.3 `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 nltk.word_tokenize(string)
```

2.2.4 `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)
```

2.2.5 `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,
    ... coordenatePosition = (0,0). 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 coordinatePosition[0] increases in
    ... numbersSteps, his mood is "MOVING", decreases energy by 1, end. To MoveLeft
    ... he needs numbersSteps, his coordinatePosition[1] decreases in
    ... numbersSteps, his mood is "MOVING", decreases energy by 1, end. To
    ... MoveRight he needs numbersSteps, his coordinatePosition[1] increases
    ... in numbersSteps, his mood is "MOVING", decreases energy by 1, end. To Bark
    ... he print "barf, barf", his energy decreases in 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), 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
        coordenateposition = (0,0)
    <BLANKLINE>
        def bark(self):
            pass
    <BLANKLINE>
        def run(self):
            pass
    <BLANKLINE>
        def moveleft(self, numberssteps):
            pass
    <BLANKLINE>
        def moveright(self, numberssteps):
            pass
    <BLANKLINE>
        def moveforward(self, numberssteps):
            pass
    <BLANKLINE>
        def lay(self):
            pass
    <BLANKLINE>
        def check(self):
            pass
    <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        pass\n\n'.format(
                method_name=method_name, param_name=param_name)
            pass
        else:
            class_code_str += '    def {method_name}(self):\n        pass\n\n'.format(
                method_name=method_name)

    return class_code_str
```

3 Lessons learned

3.1 Delimiting is hard

3.2 Can't rely just on noun/verbs

Things like moveLeft are not identified as a verb