String Matching: Inheritance

You are not logged in.

Please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

1) Preparation

This lab assumes you have Python 3.9 or later installed on your machine (3.11+ recommended).

The following file contains code and other resources as a starting point for this practice exercise: ZIP FOLDER

This practice exercise is optional and ungraded but it is designed to help you prepare for this week's `Symbolic Algebra` lab.

These problems are also a good way to practice writing code with good style. Upon submitting a correct solution to a problem, you will get access to staff solutions and explanations for each problem.

While we encourage students to collaborate on the concept questions, please refrain from collaborating on the coding problems except with staff members. To allow all students the benefit of working through the problems individually, the course academic integrity policy applies to your solution and the official solution code-- meaning no sharing or posting of answers allowed, especially on public or shared internet spaces.

2) Introduction

In this problem, we will create a small library of classes for finding patterns in strings. We will use classes to represent various kinds of patterns we might be interested to match. Instances of those classes, then, will represent specific patterns we want to match.

For each class below, implement a method `match(text)`. If the pattern matches the given piece of text (starting at the beginning), this method should return a string of text associated with the match.
If there is no match at the given location, you should instead return `None`.

If there are multiple possible matches at a given point, your `match` methods should always return the *longest possible match*.

Look we encourage you to look for opportunities to practice inheritance and avoid repeated code!

3) Primitives

Implement the following primitive classes, each of which represents a type of pattern, in `practice.py`:

- `Dot()` matches any single character
- `Verbatim(string)` matches some text exactly to the `string` given at initialization time.
- `CharFrom(chars)` matches a single character contained in the given `chars` iterable.
- `Digit()` matches any single number 0-9

For example:

```
>>> print(Verbatim('tomato').match("tomatoes"))
'tomato'
>>> print(Verbatim('tomato').match("i don't like tomatoes"))
None
```

```
>>> print(Dot().match(''))
None
>>> print(Dot().match('hello'))
'h'
>>> print(Dot().match('ello'))
'e'
```

```
>>> print(CharFrom('abc').match('cat'))
'c'
>>> print(CharFrom('abc').match('at'))
'a'
>>> print(CharFrom('abc').match('t'))
None
>>> print(CharFrom('abc').match(''))
None
```

```
>>> print(Digit().match('.12'))
None
>>> print(Digit().match('12'))
'1'
>>> print(Digit().match('2'))
'2'
```

Note that Verbatim and Dot have already been implemented for you.

Paste your definition of the CharFrom and Digit classes below: The test cases are a subset of the `test.py` file so if your code is not passing the test cases online but is passing all the test cases locally, let us know!

```
1
2  class CharFrom():
3      """
4      Matches any single character from the given iterable object of charact
5      """
6      def __init__(self, chars):
7          raise NotImplementedError
8
9      def match(self, text):
10         raise NotImplementedError
11
12 class Digit():
13     """
14     Matches any single integer digit 0-9.
15     """
16     def __init__(self):
17         raise NotImplementedError
18
19     def match(self, text):
20         raise NotImplementedError
21
```

4) Combinations

In addition, implement the following combinations:

- `Sequence(patterns)` should take in a list of patterns (instances of these classes) and should match the given arguments in order, from left to right. If the first pattern produces a match, we should start trying to match the second pattern where the first ended, and so on.
- `Alternatives(patterns)` should take in a list of patterns. In order to determine a match, it should try each of the patterns in order until one matches, and it should return the first match. If none of the given patterns matches, then this pattern does not match.
- `Repeat(pattern, n_min, n_max)` should try to match the given pattern a minimum of `n_min` times but no more than `n_max` times. Note that this should match the maximum possible number of repetitions, up to the given `n_max`.
- `Star(pattern)` should match the given pattern an arbitrary number of times (the empty string should also be a valid match, which corresponds to matching the given pattern 0 times).

Examples:

```
>>> print(Sequence([CharFrom('abc'), Verbatim('dog')]).match('btadog'))
None
>>> print(Sequence([CharFrom('abc'), Verbatim('dog')]).match('adog'))
'adog'
>>> print(Sequence([CharFrom('abc'), Verbatim('dog')]).match('dog'))
None
```

```
>>> print(Alternatives([CharFrom('abc'), Verbatim('dog')]).match('catdog'))
'c'
>>> print(Alternatives([CharFrom('abc'), Verbatim('dog')]).match('atdog'))
'a'
>>> print(Alternatives([CharFrom('abc'), Verbatim('dog')]).match('tdog'))
None
>>> print(Alternatives([CharFrom('abc'), Verbatim('dog')]).match('doga'))
'dog'
```

```
>>> print(Repeat(CharFrom('abc'), 2, 5).match('abcabca'))
'abcab'
>>> print(Repeat(CharFrom('abc'), 2, 5).match('bcabca'))
'bcabc'
>>> print(Repeat(CharFrom('abc'), 2, 5).match('cabca'))
'cabca'
>>> print(Repeat(CharFrom('abc'), 2, 5).match('abca'))
'abca'
>>> print(Repeat(CharFrom('abc'), 2, 5).match('bca'))
'bca'
>>> print(Repeat(CharFrom('abc'), 2, 5).match('call'))
'ca'
>>> print(Repeat(CharFrom('abc'), 2, 5).match('all'))
None
```

```
>>> print(Star(Verbatim('hello')).match(''))
''
>>> print(Star(Verbatim('hello')).match('hello'))
'hello'
>>> print(Star(Verbatim('hello')).match('hellohello'))
'hellohello'
>>> print(Star(Verbatim('hello')).match('hellohellohello'))
'hellohellohello'
>>> print(Star(Verbatim('hello')).match('hellohellohelloheyhello'))
'hellohellohello'
```

Note that Sequence has already been implemented for you.

Ask staff for the password to release the Repeat Solution:

Paste your definition of the Alternatives, Repeat, Number, and Star classes below: The test cases are a

subset of the `test.py` file so if your code is not passing the test cases online but is passing all the test cases locally, let us know!

```
class Alternatives():
    """
    Matches if _any_ of the given patterns match, by trying them in the o
    they were given.
    """
    def __init__(self, patterns):
        raise NotImplementedError

    def match(self, text):
        raise NotImplementedError


class Repeat():
    """
    Matches if the given pattern (given as an instance of one of these cl
    exists repeated between n_min (inclusive) and n_max (inclusive) times
    This matching should be greedy (i.e., it should match as many repetit
    as possible up to `n_max` times).  It should not match if there are f
    than `n_min` repetitions.
    """
    def __init__(self, pattern, n_min, n_max):
        raise NotImplementedError

    def match(self, text):
        raise NotImplementedError


class Number():
    """
    Matches if the text matches one or more consecutive digits (no limit)
    This matching should be greedy (i.e., it should match as many consecu
    digits as possible).  It should not match if there is not a digit at
    given location.
    """
    def __init__(self):
        raise NotImplementedError

    def match(self, text):
        raise NotImplementedError


class Star():
    """
    Matches the given pattern (an instance of one of these classes) repea
    arbitrary number of times.  0 times (matching the empty string) is a
    match.
    """
    def __init__(self, pattern):
        raise NotImplementedError

```

```
52      def match(self, text):
53          raise NotImplementedError
54
```

5) Pattern Search

Finally, all of these classes should support an additional method `find_all(text)`. This method should be a generator that yields all **non-overlapping** matches of the pattern that exist within the given piece of text (as 3-tuples `(start, end, text)`), starting with the left-most match.

You should implement exactly one method called `find_all`.

For example:

```
>>> x = Repeat(Sequence([CharFrom('abc'), CharFrom('def')]), 2, 5)
>>> y = x.find_all('adadadadadadadgadgadgadgadgad')
>>> y
<generator object find_all at SOME_MEMORY_LOCATION>
>>> for i in y:
...     print(i)
...
(0, 10, 'adadadadad')
(10, 14, 'adad')
>>> y = x.find_all('adadadadadadadgadgadgadgadgadbe')
>>> for i in y:
...     print(i)
...
(0, 10, 'adadadadad')
(10, 14, 'adad')
(27, 31, 'adbe')
```

Paste all your code below: The test cases are a subset of the `test.py` file so if your code is not passing the test cases online but is passing all the test cases locally, let us know!

```
1  # paste your code below!
2
```