String Pattern Matching

It is easy to check if a string `p` is a substring of another string `t`. We just write:

```
p in t
```

This will evaluate to `True` if yes and `False` if no. The string `p` will be called the **pattern string** and the string `t` is the **text string**. We want to find the pattern `p` in the text `t`. In this assignment, you will implement your own version of this pattern matching that has some extra features.

Objectives

1. Learn to use Python modules.
2. Learn to implement objects.
3. See example of wrap.
4. Define object method with optional arguments.
5. See the difference in performance between different algorithms.

Assignment

Make a class for patterns called `Pattern` and save it in a file called **pattern.py**. (Remember that exact names and capitalization matter). The pattern object should store the pattern string. You will create an instance of the `Pattern` class by specifying the pattern string as in the following.

```
pat1 = Pattern('hello, world!')
pat2 = Pattern('this is a pattern string!!')
```

Part 1

The `Pattern` class should implement a method called `_findMatch(self, text)` that checks if the pattern that was specified when you created the object matches the text and returns the index of the first occurrence of this pattern. If the pattern doesn't appear in `text`, then `_findMatch()` should return -1.

Consider the following example:

```
p = Pattern("abc")
print(p._findMatch("ababcabc"))
```

The first occurence of `abc` in `ababcabc` is at index 2. So the output of the above code is `2`.

```
print(p._findMatches("ababgabg"))
```

The pattern `abc` doesn't appear in the text, `ababgabg`, so the output of the above code is `-1`.

Implement this by using the very simple `find()` method that is available for strings.

Hint: `_findMatch` method should consist of a single line that calls `find`. This method is an example of a wrapper, meaning that it does nothing more than calling another method.

Please refer the link for more information on how to use `find`:
https://www.programiz.com/python-programming/methods/string/find

Note: `find` method returns -1 on failure.

Part 2

The `Pattern` class should implement an advanced version of `_findMatch` that checks if the pattern matches the text and returns a **list** of indices of **all** the occurences of the pattern. Call this method `_findMatches(self, text)`. If the pattern doesn't appear in `text`, then `_findMatches()` should return an empty list `[]`.

Consider the following example:

```
p = Pattern("abc")
print(p._findMatches("abcababcababcabc"))
```

The pattern occurs at indices 0, 5 and 12. The output of the above code is `[0, 5, 12]`

```
print(p._findMatches("ababgabg"))
```

The pattern `abc` doesn't appear in the text, `ababgabg`, so the output of the above code is -1.

Consider the following example that shows the difference between `_findMatch` and `_findMatches`:

```
p = Pattern("abc")
print("The output of _findMatch is", p._findMatch("abcdeabc"))
print("The output of _findMatches is", p._findMatches("abcdeabc"))
```

Output

```
The output of _findMatch is 0
The output of _findMatches is [0, 5]
```

In the first case, `_findMatch` returns `0` as that is the index of the first occurrence of the pattern. In the second case, the pattern occurs at indices `0` and `5`. Therefore, the output is `[0, 5]`.

Hint: The method `_findMatch` in part 1 searches `text` from the start index of `0` and gives you the index of the first occurrence of the pattern. Add an optional argument `start` to `_findMatch` that lets you specify where to start (default value is 0). Call `_findMatch` within `_findMatches`. To find all occurrences, you just have to change the start index within `_findMatches` each time you make a call to `_findMatch`.

First, use a for loop that traverses through `text` and increments the start index by 1. Append the index returned by `_findMatch` to a list **if the index isn't already in the list**. This gives a slow implementation. Now make a fast implementation that sets the start index to match `index+1` (index here is the return value of `_findMatch`); note that there is no longer a need to check if the index is new, because it always will be.

With a big enough text string that has very few matches of a given pattern, there should be a noticeable difference in performance between the two implementations.

Note: There is a test case included that tests for really long patterns. You might want to disable this test while testing locally until you implement the fast version because it can run for a really long time.

Part 3

Your `Pattern` class should also support wildcard matches. The pattern object will store a single character (a character is a string of length 1) called the wildcard character. The wildcard character is **optional** - the pattern can either contain a wildcard character or not. When comparing characters in the text and the pattern, a wildcard character in the pattern will always be a match regardless of the character it is being compared to.

Note: You have to modify `_findMatch` to support this functionality. If you have implemented `_findMatches` using the hints provided in part 2, then you shouldn't have to make any changes to `_findMatches`.

Consider the following example:

```
p = Pattern("ab+", "+")
print(p._findMatch("abcabdefg"))
```

The output of this code will be `0`. This is because the `+` in the pattern is the wildcard and it can match with any character in the text. In this example, `+` matches with `c`.

```
print(p._findMatches("abcabdefg"))
```

The output of this code will be `[0, 3]`. The `+` matches with `c` in the first occurrence and `d` in the second occurrence.

Note: The wildcard can be any character. It is not just special characters.

You should set the wildcard when initializing a new Pattern object. Having no wildcard character should be indicated by setting it to `None`. So, the following code should run without error.

```
p_nowildcard = Pattern('abc')
assert(p_nowildcard._findMatch('abc') == 0)

p_wildcard = Pattern('a*c', '*')
assert(p_wildcard._findMatch('abc') == 0)
```

In the first case, the wildcard has been set to `None`, and `_findMatch` will call `find` as before. In second case, the wildcard has been set to `*` and `_findMatch` will look for the pattern based on the new code that you write.

Part 4

Finally, your class should support case sensitivity.

Create an attribute in the class called `case_sensitive` and set its value to `False` initially (the default should be not case sensitive). Implement a method called `_set_case_sensitive(self, case)` that takes a boolean and allows you to change the attribute, `case_sensitive`. This lets us know whether the pattern matching should ignore the case of the characters.

Perform checks (check the attribute `case_sensitive` in `_findMatch` and `_findMatches` to see if the pattern matching is case sensitive or not. The `String` class has methods `upper()` and `lower()` that are useful here.

Consider the following example:

```
p = Pattern("abc")
print(p._findMatches("ABCababcabababc"))
```

The output of this code is `[0, 5, 12]` even though the case isn't the same. This is because the default is that the pattern matching is not case sensitive. Now, consider the following:

```
p = Pattern("abc")
p._set_case_sensitive(True)
print(p._findMatches("ABCababcabababc"))
```

The output of this code is `[5, 12]` because the pattern matching is now case-sensitive. The upper case `ABC` will not be considered a match.