Regular Expressions

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu
You should have researched one of these topics on the LGA:

- Reference Counting
- Smart Pointers
- Valgrind

Explain to your group!
**Regular expression** languages describe a search pattern on a string.

- They are called *regular*, since they implement a **regular language**: a language which can be described using a finite state machine.
- Typically used for determining if a string matches a pattern, replacing a pattern in a string, or extracting information from a string.
- Regular expression languages are a *family of languages*, rather than just a single language. Many modern regular expression languages were inspired by Perl’s regular expression syntax.
Python’s regular expression language can be accessed using the `re` module:

```python
>>> import re
```

Regular expressions can be compiled using `re.compile`. This returns a regular expression object:

```python
>>> p = re.compile(r'ab[cd]')
```

There’s a number of things we might want to do with `p` here:

- `p.match`: Match the beginning of a string
- `p.fullmatch`: Match the whole string, without allowing characters at the end
- `p.search`: Match anywhere in the string
- `p.finditer`: Iterate over all of the matches in the string
[abcd] is a character set. It matches a single a, b, c, or d, only once.

Character sets also support a shorthand for ranges of characters, for example:

- [0-9] matches a single digit
- [a-z] matches a lowercase letter
- [A-Z] matches an uppercase letter

These can even be combined:

- [a-zA-Z2] will match a single lowercase letter, uppercase letter, or the digit 2.

A ^ (caret) at the beginning of a character set negates the set:

- [^0-9] will match a single character that is not a digit.
As a convenience, Python gives us access to a few nice character sets:

- \s matches any whitespace character
- \S matches any non-whitespace character
- \d matches any digit
- \D matches any non-digit
- \w matches any "word" character (capital letters, lowercase letters, digits, and underscores)
- \W matches any non-word character
Any character

The . matches any character, exactly once.

- t.ck will match tick, tock, and tuck, but not truck.

To match a literal period, write "\.".
Match Objects

When we call match, fullmatch, or search, we get back a **match object**, or None if it did not match. When we iterate over finditer, we iterate on all of the match objects found.

```python
>>> p = re.compile(r'[cd][ao][tg]')
>>> for word in 'cat', 'dog', 'cog', 'dat', 'datt':
...     print(bool(p.match(word)))
True
True
True
True
True
```

```python
>>> for word in 'orange', 'apple', 'datum':
...     print(bool(p.match(word)))
False
False
True
```
Often times, we want to match the previous group a certain number of times:

- `?` will match 0 or 1 times
- `+` will match 1 or more times
- `*` will match 0 or more times
- `{n}` will match exactly `n` times
- `{m,n}` will match between `m` and `n` times

For example:

- `a?b` matches `ab` as well as `b`
- `[A-Z]*` matches any amount of capital letters, including none at all
- `[0-9]+` matches one or more digits
- `.*` matches any character, zero or more times
Grouping

Grouping allows us to:

- Specify groups of characters to repeat
- Alternate on different sets of characters
- Capture the matched group and retrieve it in our match object

Groups are written in parentheses, and alternation is specified using a vertical bar (|):

- Thanks?( you)? matches:
  - Thanks
  - Thank
  - Thank you
  - Thanks you

- Thank(s| you) matches:
  - Thanks
  - Thank you
On our match objects, we can obtain the result of a capture by calling .group:

```python
>>> p = re.compile(r'My name is (\w+) and I like (\w+)')
>>> m = p.match('My name is Jack and I like computers')
>>> m.group(1)
'Jack'
>>> m.group(2)
'computers'
>>> m.group(0)  # the whole match
'My name is Jack and I like computers'
```
Non-capturing Groups

Groups which begin with `?:` are **non-capturing groups**. This means that they will not provide any visible group in the match object:

```python
>>> p = re.compile(r'\w+')(?:, | and) I like (\w+)
>>> m = p.match('My name is Jack and I like computers')
>>> m.group(1)
'Jack'
>>> m.group(2)
'computers'
>>> m = p.match('My name is Jack, I like computers')
>>> m.group(1)
'Jack'
>>> m.group(2)
'computers'
```
Greedy operators

+ *, and ? are called greedy operators since they will try and match as many characters as possible, this may lead to undesired results:

```python
>>> p = re.compile(r'^(.*)$')
>>> for m in p.finditer('#hello a b c #world# '):
...     print(m.group(1))
hello a b c #world
```

If we wanted to match as little as possible, we can use the non-greedy version of the operator, which would be +?, *?, or ??.

```python
>>> p = re.compile(r'^(.*)$')
>>> for m in p.finditer('#hello a b c #world# '):
...     print(m.group(1))
hello world
```
Anchors match a certain kind of occurrence in a string, but not necessarily any characters.

- `^` anchors to the beginning of a string, or to the beginning of a line when `re.MULTILINE` is passed to `re.compile`
- `$` anchors to the end of a string, or to the end of a line when `re.MULTILINE` is passed to `re.compile`
- `\b` anchors to the boundary of a word: the transition from a `\w` to a `\W`, or visa versa. Also anchors to the beginning or end of a string.

Examples:

- `foo\b.*` matches `foo` and `foo-dle`, but not `foodle`
- `^$` matches the empty string
- `//--.*(``\n$`|`$`) matches `//-- hello` and `//-- hello\n`, but not `//-- hello\n\n`
Tip: Making Long REs Readable

Sometimes, when regular expressions get long, you need a way to comment them and break up sections to let other programmers (or yourself) know what’s going on.

When you pass `re.VERBOSE` to `re.compile`, whitespaces are ignored, and `#` starts a comment until the end of line:

```python
p = re.compile(r'''
    (\w+)  # first name
    \s+
    (\w+)  # last name
    \s+
    ([2-9]d{2}-[2-9]d{2}-d{4})  # phone number
''', re.VERBOSE)
```
RE Examples, and any Questions?

- Matching a decimal number:
  \[0-9]+\.?[0-9]^{*}

- Matching a C/C++ identifier:
  \[A-Za-z_][A-Za-z0-9_]{*}

- Matching a Mines Email address:
  (\[A-Za-z0-9.-]+)@\(mymail\.)?\(mines\).edu
Finite State Machines

A **finite state machine** is any machine which has a finite number of states, and can only be in one state at a time. The machine has *transitions* that move it from one state to another.

![State Diagram for Home Phone](image)

**Figure**: A state diagram for your home phone
Regular Expressions as Finite State Machines

Regular expressions can be represented as finite state machines as well. Consider the following regular expression:

\(^{\textbf{\textdollar}}fr?ee\$\)

This matches both free and fee, we can write this in a state diagram like this:

![Diagram of a finite state machine]

Required Formalisms

- Any state which \textit{could} be a terminating state should be placed in \textbf{double circles}.
- The transitions have the letters on them. The states do not.
- Transitions correspond to only a single character, so repetition and groups must be encoded using the FSA.
Recall the regular expression for C and C++ identifiers:

```
[A-Za-z_][A-Za-z0-9_]*
```

![Diagram of a finite automaton with states S0 and S1, transitions labeled with A-Za-z_.](image)
This is an open source tool developed by Sam Sartor (took CSCI-400 last semester) to help you visualize regular expressions using finite state graphs:

http://gh.samsartor.com/regess/
With your learning group, translate each of these REs to a state diagram:

- \([A-Z]^+\)
- \([A-Z]?x\) (try using \(\epsilon\) for the "no character" transition)
- \(([A-Z][1-5])^+\) (hint: draw a transition going backwards)

Write your names on your paper and turn in for \textbf{bonus} learning group participation points.