Regular Expressions

Optimization, Python, and Git

Optimizing Regular Expressions

- Regular expressions are extremely powerful, but can be quite time consuming
- A Google search for "optimizing regular expressions" returns dozens of articles and blogs about the subject
- My basic rules of thumb:
  - Get it working first
  - Don't be afraid to look for another solution

Why Regular Expressions can be Slow

- Some of this is implementation dependent, but regular expressions work by going through a string one character at a time, looking for matches
  - If there are a lot of comparisons to be made at each character in the string, this will slow the regex down
- Some regexes require backtracking to determine if there is a match or not
  - The more backtracking, the longer the regex will take to execute
  - A good resource on this is Catastrophic Backtracking
    - Although written as an ad for their product, it does have a lot of helpful information

Optimization Tip 1

- Don't use regular expressions if you don't have to
  - This is especially true if the pattern you are searching for is all literals

In [ ]:
```
%%bash
time perl src/perl/slow.pl
```

In [ ]:
```
%load src/perl/slow.pl
```

In [ ]:
```
%%bash
time perl src/perl/fast.pl
```

In [ ]:
```
%load src/perl/slow.pl;
```

Optimization Tip 2

- If you can, use ^ and $ anchors
  - Limiting where a match can occur can make a regex fail faster

In [ ]:
```
%%bash
time perl src/perl/anchored.pl
```

In [ ]:
```
%%bash
time perl src/perl/unanchored.pl
```

Optimization Tip 3

- Avoid quantifiers if you don't need to use them
- If you need to use them, see if you can use the non-greedy version

In [ ]:
```
%%bash
time perl src/perl/greedy.pl
```

In [ ]:
```
%%bash
time perl src/perl/nongreedy.pl
```

Optimization Tip 4

- Structure your alternations efficiently
  - Alternations are searched left-to-righ, so put the (suspected) most common first

In [ ]:
```
%%bash
time perl src/perl/good_alt.pl
```

In [ ]:
```
%%bash
time perl src/perl/bad_alt.pl
```

Optimization Tip 5

- Use non-capturing groups
  - If you are just using grouping to apply a quantifier or something else over a part of a pattern, consider a non-capturing group `(?:pattern)`

In [ ]:
```
%%bash
time perl src/perl/capture.pl
```

In [ ]:
```
%%bash
time perl src/perl/noncapture.pl
```

Regex in Python

Intro to re Module

- Regular expressions are not built into the core python language
  - Available by importing the standard `re` module
- Matching and substitution are done using methods
- To avoid having to escape the \ character in Python so the regex can process use a raw string
  - `r'This is a raw python string\n'`

In [ ]:
```
import re
```

Simple Matching

- The `re` module has 4 methods to performing matching
  - re.match
  - re.search
  - re.findall
  - re.finditer
- All methods take the arguments (pattern, string, optional_flags)

In [ ]:
```
if re.match(r'needle',r'Is there a needle in this haystack?'):
    print "match"
```

In [ ]:
```
if re.search(r'needle',r'Is there a needle in this haystack?'):
    print "match"
```

The Match Object

- Regular expressions don't evaluate to `true` or `false` in Python
  - If a match is found, a `MatchObject` is returned
  - If no match is found, `None` is returned
- The `MatchObject` can be used to access groups found in the match, as well as information such as position

In [ ]:
```
re.search(r'needle',r'Is there a needle in this haystack?')
```

In [ ]:
```
match = re.search(r'(\w+)\sneedle',r'Is there a needle in this haystack?')
print match.group(0)
print match.group(1)
```

re.findall and re.finditer

- Rather than using a g modifier, Python has two specialized functions
- re.findall returns the groups themselves
- re.finditer returns an iterator over `MatchObjects`

In [ ]:
```
re.findall(r'\b\w*a\w*\b', r'Is there a needle in this haystack?')
```

In [ ]:
```
re.findall(r'\b(\w*)a(\w*)\b', r'Is there a needle in this haystack?')
```

Backreferencing

- Backreferencing works exactly the same in Python
- Python also allows named groups, but personally I find it messy

In [ ]:
```
re.findall(r'(\w)\1', r'Is there a needle in this haystack?')
```

In [ ]:
```
re.findall(r'(?P<a_letter>\w)(?P=a_letter)', r'Is there a needle in this haystac
k?')
```

Substitution

- Substitution is done using the `re.sub` method

```
re.sub(pattern,replacement,string,count=0,flags=0)
```

- `re.sub` is global by default. To do only one substitution set the `count` parameter to 1
- `replacement` can be either a string or a function that takes a `MatchObject` as it's argument
- Back references are done using `\1` instead of `$1`

Substitution Examples

In [ ]:
```
re.sub(r'(\w)\1','oo',r'Is there a needle in this haystack?')
```

In [ ]:
```
re.sub(r'(\w)\1',r'\1',r'Is there a needle in this haystack?')
```

Splitting Strings

- The `re` module can split strings using the `split` method

```
re.split(regex,string,limit,flags)
```

In [ ]:
```
re.split(r'[aeiou]+',r'Is there a needle in this haystack?')
```

Using Flags

- Flags in Python are constans of the `re` module
  - re.I and re.IGNORECASE are equivalent to the i modifier in Perl
  - re.M and re.MULTILINE are equivalent to the m modifier in Perl
- To use multiple flags, you must "or" them together

Flag Examples

In [ ]:
```
re.search(r'n(\w)\1dle',r'Is there a\n NOODLE in this haystack')
```

In [ ]:
```
match = re.search(r'n(\w)\1dle',r'Is there a\n NOODLE in this haystack',flags =
re.I | re.M)
print match.start(), match.end(), match.pos, match.group(0), match.group(1)
```

Compliling Regular Expressions

- If a regular expression is going to be used over and over again, you should compile it to the languages internal representation
  - Most languages have a concept of compilation
  - In Python, calling `re.compile(pattern,flags)` will return a `RegexObject`
- The methods of a `RegexObject` are mostly the same as the `re` module, but the pattern is no longer passsed as an argument

Compiling Regular Expressions Examples

In [ ]:

```
regex = re.compile(r'n(\w)\1dle',flags = re.I | re.M)
if regex.search('iS ThERe a \nNOoDLE iN This HaYStaCK'):
    print "Match"

print regex.sub(r"z\1\1",'Is there a noodle in Baltimore?')

for match in regex.finditer('You shouldn\'t sew your clothes with a noodle, no m
atter how many needles you have'):
    print match.group(0)
```