CIS192 Python Programming

Regular Expressions and maybe OS

Robert Rand

University of Pennsylvania

October 1, 2015

Outline

What are Regular Expressions

- Formal specification of a language (set of strings)
- A regular language is one that can be recognized by a DFA
  - If you've taken CIS 262 (Automata, Computability, and Complexity), you know regular languages
  - Otherwise: Take CIS 262 (it's really interesting stuff)!
- Deciding if a string is in a regular language is very efficient
- Used for: Find/Replace, Syntax highlighting, Lexing (Compilers)
- Python (and most PLs) support more than regular languages
- Using non-regular features can lead to exponential run-times

XKCD


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
REKT!
TAP TAP
Python

Core Features

- Specify character literals to match directly
  - `'a'` matches exactly 1 a
  - `'word'` matches exactly the string word
- `'A|B'` matches for either of two regexs A or B
  - `'foo|bar'` matches: foo, bar
- `'AB'` matches the regex A followed by regex B
  - `'(a|aa)(b|bb)'` matches: ab, aab, abb, and aabb
- `'x?'` matches 0 or 1 occurrences of x
  - `'a?'` matches: empty string, a

Character Repetitions

- `'x*'` matches 0 or more x's
  - `'a*'` matches: empty string, a, aa, aaa, ...
- `'x+'` matches 1 or more x's
  - `'a+'` matches: a, aa, aaa, ...
- `'x{m}'` matches exactly m occurrences of x
- `'x{m, n}'` matches between m and n occurrences of x
  - omitting m sets 0 as the lower bound
  - omitting n uses an infinite upper bound
- `'+'` `'*'` `'?'` and `'{m, n}'` are greedy (biggest match)
  - `'a+'` will match aaa in aaab
- `'+?'` `'*?'` `'??'` and `'{m, n}?'` are lazy (smallest match)
  - `'a+?'` will match a in aaab

Special Characters

- `'.'` is a wildcard that matches any single character
  - `'.'` will match G or 3 or a space, but not \n
- `'^'` matches the beginning of the string
  - `'^h'` will find a match in hij but not in fghi
- `'$'` matches the end of the string
- To match a literal special character uses a `'\'`
  - `'\*'` matches the literal *

Character Classes

- Specify a set of characters within `'[]'`
  - `'[aeiou]'` matches any vowel
- Specify character ranges with `'-'` inside `'[]'`
  - `'[a-z]'` matches lowercase and `'[0-9]'` matches digits
- a leading `'^'` within `'[]'` negates the set
  - `'[^aeiou]'` matches any consonant
  - Actually `r'[^aeiouAEIOU_\d\W]'` matches any consonant
- Built-in character classes
  - `'\d'` → `'[0-9]'`
  - `'\D'` → `'[^0-9]'`
  - `'\s'` → `r'[ \t\n\r\f\v]'`
  - `'\S'` → `r'[^ \t\n\r\f\v]'`
  - `'\w'` → `'[a-zA-Z0-9_]'`
  - `'\W'` → `'[^a-zA-Z0-9_]'`

Using a Regex

- `re.compile(regex)` will create a regular expression object
  - You can then use the methods of that object for matching
- Methods of the re module can use a regex string directly
- Compiling a legit regular expression can be exponential
  - Using the compiled object afterwards will be linear
- There are a bunch of flags that change the meaning of the regex
  - Either specify them in the string with `'(?aiLmsux)'`
  - Or in `re.compile(flags=)`

re Methods

- `re.search(regex, string)`
  - returns the first match for `regex` in `string`
- `re.match(regex, string)`
  - returns a match if the `regex` matches at the start of `string`
- `re.finditer(regex, string)`
  - returns an iterator of all non-overlapping matches
- `re.findall(regex, string)`
  - returns a list of all substrings that match in `string`
  - Basically `[m.group() for m in re.finditer(...)]`
- `re.sub(regex, repl, string)`
  - Extends `string.replace(old, new)` to regular expressions
- `re.split(regex, string)`
  - Extends `string.split(sep)` to regular expressions

Match Objects

- Some of the `re` methods will return `Match` object, not strings
- A `Match` object m supports:
- `m.group(id/name)`: Returns the string of the captured group
  - `m.group()`: returns the full string that matched
- `m.groupdict()`: A dictionary of named groups to strings
- `m.start(id/name)`: The index in the original string where the group (id/name) starts
- `m.end(id/name)`: the index at which the group ends
- `m.span(id/name)`: a tuple of start and end

Non-regular features 1

- `'(A)'` matches the regex A and numbers it for reference
  - `r'(\w)(\w)\1\2'` matches: abab, XyXy
- `'(?P<name>A)'` matches regex A and calls it name
  - `'(?P<delim>.).*(?P=delim)'` matches: x123x
- `'(?:A)'` matches the regex A but it can't be referenced
  - `r'(?:a\d*z)+'` matches az, a1za2z
- Any named or numbered group can be accessed after the match
- `'(?# a comment)'` is a comment and will be ignored

Non-regular features 2

- Look-aheads do not consume input but affect which strings match
- `'(?=A)'` is a positive look-ahead for the regex A
  - searching in 1
2
3
4
  - `r'<br/>\d(?=<br/>)'` matches
2,
3
  - `r'<br/>\d<br/>'` will find only
2
- `'(?!A)'` is a negative look-ahead for the regex A
  - `'turtle (?!soup)'` matches 'turtle shell' not 'turtle soup'
- `'(?<=A)'` is a positive look-behind for the regex A
- `'(?<!A)'` is a negative look-behind for the regex A
- look-behinds must match strings of fixed length
- `'(?(id/name)THEN|ELSE)'` matches
  - the regex THEN if group id/name exists
  - the regex ELSE if group id/name does not exists

Outline

os.path

- Useful operations for doing os independent filepath manipulation
- Allow cross-platform code
- `os.path.join(path1, path2, path3, ...)`
  - Uses / on POSIX and \ on Windows
- `os.path.realpath(path)` returns the absolute path of a file
- `os.path.exists(path)`
- `os.path.isfile(path)`
- `os.path.isdir(path)`

os Functions

- `os.getcwd()`: returns a string for the current working directory
- `os.walk(top)`: A generator that
  for each directory in the tree rooted at `top`
  yields the tuple `(dirpath, dirnames, filenames)`
- `s = os.stat(path)`: a stat object with useful info on the file
  - `s.st_size` file size, `s.st_ctime` creation time, ...
- `os.mkdir(path)`: create a directory named path
- `os.chdir(path)`: `cd` into path
- `os.remove(path)`: `rm` path if not a directory
- `os.rmdir(path)`: `rm -r` path if it's an empty directory
- recursive deletion, copy, and move are in the `shutil` module