CMSC330: Regular Expressions

Chris Kauffman

Last Updated:
Mon Sep 11 03:14:41 PM EDT 2023

Logistics

Assignments

- Lecture Quiz 2 up later, due Tue
- Project 1 "Intro Python" Due Sun 10-Sep

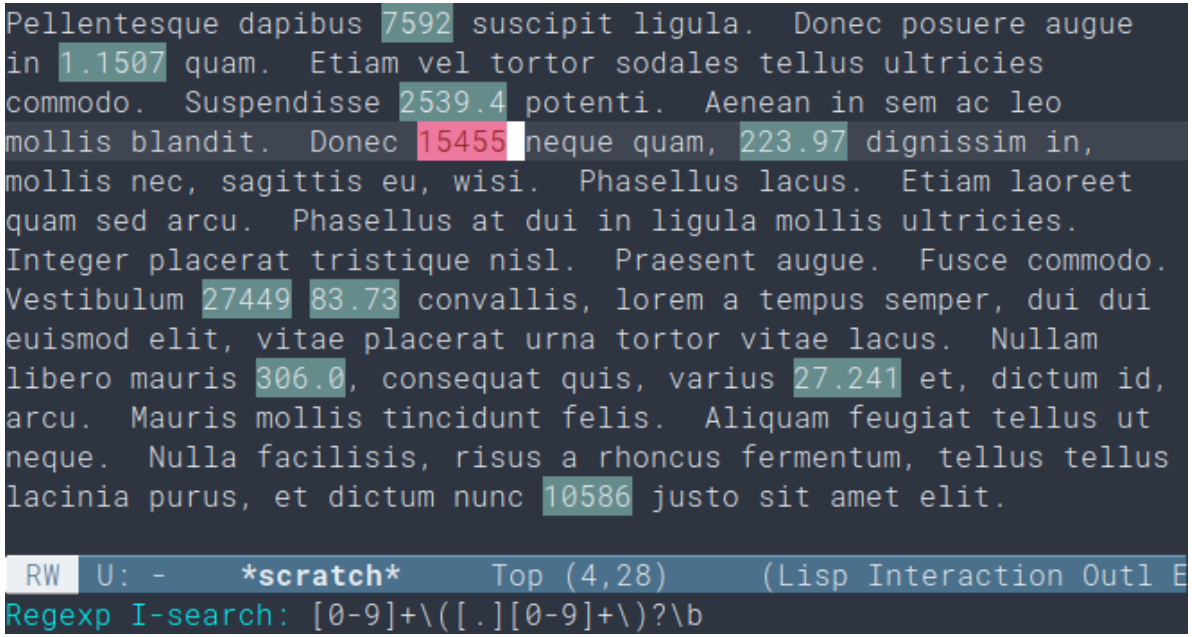
Reading

- Python re Module: Docs on the Regular Expressions Python provides
- Related Match Object

Goals

- Wrap up higher-order function
- Regular Expressions and their Uses
- Python's `re` Module for Regular Expressions

Regular Expressions Overview



- Recognizing and selecting patterns often comes up in computing situations
- A fundamental part of building Programming Language interpreters and compilers is recognizing the raw text as either correct or not:
- **Regular Expressions** address these problems; they are a
  - Mini-language for describing text patterns
  - Set of tools to detect the patterns described
  - An underlying theory of what can be recognized efficiently

A Motivating Example

From "The Five Essential Phone-Screen Questions" by Steve Yegge

Let's say you're on my team, and we have to identify the pages having probable U.S. phone numbers in them. To simplify the problem slightly, assume we have 50,000 HTML files in a Unix directory tree, under a directory called "/website". We have 2 days to get a list of file paths to the editorial staff. You need to give me a list of the .html files in this directory tree that appear to contain phone numbers in the following two formats:

`(xxx)-xxx-xxxx AND xxx-xxx-xxxx.`

How would you solve this problem? Keep in mind our team is on a short (2-day) timeline.
– Steve Yegge

Solutions

Here are some facts for you to ponder:

- Our Contact Reduction team really did have exactly this problem in 2003. This isn't a made-up example.
- Someone on our team produced the list within an hour, and the list supported more than just the 2 formats above.
- About 25% to 35% of all software development engineer candidates, independent of experience level, cannot solve this problem, even given the entire interview hour and lots of hints.
- Here's one of many possible solutions to the problem:

```
grep -l -R --perl-regexp \
     "\b(\(\d{3}\)\s*|\d{3}-)\d{3}-\d{4}\b"
```

- If they say, after hearing the question,

  "Um... grep?"

  then they're probably OK... Heck, if they can tell me where they'd look to find the syntax [for the regular expression], I'm fine with it.

– Steve Yegge

Exercise: The Regex Guessing Game

- `re_guessing_game.py` has a series of "rounds" that demonstrate various aspects of regular expression syntax
- Regexs are applied to some text using Python's `re.findall(regexp,text)` function to (duh) find all matching text
- Will run the code and examine the matches
- Brave neighbors will describe the meaning of symbols

Answers: The Regex Guessing Game

See `re_guessing_game.py` and `re_guessing_game.txt` in codepack for the code, output, and notes taken during class.

Summary of Symbology

Syntax	Matches...
`ab`	The fixed string `ab`
`a+`	One or more of `a`, as many as possible
`a*`	Zero or more of `a`, as many as possible
`a | b`	Match `a` or `b`
`a{2,5}`	Match 2 to 5 `a` as in `aa`, `aaa`, `aaaa`, `aaaaa`
`a{2,}`	Match 2 or more `a`
`a{,5}`	Match 0 to 5 `a`
`a?`	Match 0 or 1 `a`
`[0-9]`	Char range `0` to `9`
`\d`	Any digit character `0-9`
`[a-z]`	Any lower-case character
`[^a-z]`	Any character EXCEPT lower-case letters (Not `a-z`)
`\w`	Any word character (letter, digit, underscore `_`)
`.`	Any single character, any type
`\b`	A boundary (but don't include it in the match)
`\s`	Whitespace (spaces, tabs, newlines)

Some Warnings

> *Some people, when confronted with a problem, think I know, I'll use regular expressions. Now they have two problems.*
> *– Jamie Zawinski, Netscape Engineer and DNA Lounge proprietor (source discussion)*

- Regular Expressions are a **mini-language** or **Domain Specific Language (DSL)** often embedded in tools or full PLs
- Regex's have their own syntax and semantics which vary between Python, Java, OCaml, command line
- Regexs' text must be **compiled to a lower form** to be useful, usually compiled to a **finite state machine**
- Regex's are NOT a full PL (e.g. can't compute Fibonacci with them), will study their limits in power

Python String Forms

- Python has several string means to write strings (text data) and a couple of them are relevant to Regexs
- Since Regexs will use Escape Sequences like `\w`, folks use **raw strings** with syntax `r"the string"` which suspends normal string interp of backslashes

```
1  # pystrings.py:
2  def show_strings():
3    dq_string = "This \"is\"\na string"
4    sq_string = 'This "is"\na string'
5
6    d3_string = """This \"is\"
7  a string"""
8    s3_string = '''This "is"
9  a string'''
10
11   # raw strings suspend "special"
12   # sequences like \n
13   ra_string = r'This "is"\na string'
14
15   for k,v in locals().items():
16     print(f"{k}:\n{v}\n")
17
18 show_strings()
```

```
1  shell>> python pystrings.py
2  dq_string:
3  This "is"
4  a string
5
6  sq_string:
7  This "is"
8  a string
9
10 d3_string:
11 This "is"
12 a string
13
14 s3_string:
15 This "is"
16 a string
17
18 ra_string:
19 This "is"\na string
```

Why Raw Strings

Some backslash sequences mean one thing in ASCII/Character contexts and another in Regexs

Seq	ASCII	Regex
\b	Backspace (ASCII code 8)	Boundary of words

Raw strings suspend interpretation of backslash sequences

```
>>> print("Hello \bworld")
Helloworld
>>> print(r"Hello \bworld")
Hello \bworld
```

Essential Python Regex Functions

Function / Syntax	Purpose
`import re`	Use the regex module
CREATE STRING LISTS	
`re.findall(regex,text)`	Produce a list of all matching substrings
`re.split(regex,text)`	Produce a list of strings BETWEEN matches
CREATE `Match` OBJECTS	
`re.search(regex,text)`	Produce a `Match` object or `None`
`re.finditer(regex,text)`	Produce an iterable object for of `Matches`
ACCESS `Match` OBJECTS	
`m.group()`	Produce the whole string that matched the regex
`m[0]`	Produce the whole string that matched the regex
`m.span()`	Produce pair of `(beg,end)` index of match in string

Tour `re_essentials.py` for examples

Checking if Matches are Found

When one wants only to detect if a regex match is present, `re.search()` is often used to produce a `Match` object or `None`

```
# re_search.py:
text="""Pellentesque dapibus 7592 suscipit ligula.  Donec
25.6 posuere augue in quam 1.1507?"""

m = re.search("\d+",text)        # search for digits
if m != None:                    # if not None ...
  print(f"Found match: {m[0]}")
else:
  print("No match present")
# Found match: 7592

m = re.search("\w+pi\w+",text)  # word with 'pi' in the middle
if m:                           # Match objects are truthy
  print(f"Found match: {m[0]}") # while None is falsey
else:
  print("No match present")
# Found match: dapibus

m = re.search("\w+ily",text)    # word ends in 'ily' which
if m:                           # is not present so None
  print(f"Found match: {m[0]}") # triggers the alternative
else:                           # else to execute
  print("No match present")
# No match present
```

Modifying Strings with Regexs

- A common use of Regexs is to locate matching strings and modify them systematically
- EXAMPLE: In the following text replace all Patterns of the form `Chapter X Section Y` with `Chapter X.Y`

```
In Chapter 3 Section 5 we will discuss the merits of dynamically
typed languages. That Section should be studied as later
in Chapter 4 Section 1 we will cover static type systems with the
following Chapter 4 Section 2 providing a summary of the trade-offs
between static and dynamic. Chapter 5 Section 12 begins discusion of
logic programming...
```

- Note that replacing `Section` with `.` will modify text NOT matching the whole pattern AND the substitution contains parts of the match
- One can write "custom code" to do this but those with regex experience will easily handle these task
- Motivates the notion of **groups in regexs**

Regex Groups

- Created with the `(pat)` syntax and numbered left to right
- Pertinent Example:

```
                        Group 0
              ||||||||||||||||||||||||||
Regex: r"Chapter (\d+) Section (\d+)"
                        |||            |||
                      Group1         Group2
```

- Group 0 is always the whole match
- Group 1 is the leftmost (to its matching)
- Group N follows suit

Demonstration of Groups

```
1  # re_substitution.py:
2  import re
3
4  text="""In Chapter 3 Section 5 we will discuss the merits of dynamically typed
5  languages. That Section should be studied as later in Chapter 4
6  Section 1 we will cover static type systems with Chapter 4 Section 2
7  providing a summary of the trade-offs between static and
8  dynamic. Chapter 5 Section 12 begins discusion of logic programming...
9  """
10
11 print("\nre.findall() groups")
12 hits_w_groups = re.findall(r"Chapter (\d+) Section (\d+)", text)
13 print(f"hits_w_groups: {hits_w_groups}")
14 # hits_w_groups: [('3', '5'), ('4', '2'), ('5', '12')]
15
16 print("\nre.finditer()")
17 hits_iter = re.finditer(r"Chapter (\d+) Section (\d+)", text)
18 for m in hits_iter:
19   print(f"m[0]: {m[0]} \t m[1]: {m[1]} \t m[2]: {m[2]}")
20 # m[0]: Chapter 3 Section 5      m[1]: 3         m[2]: 5
21 # m[0]: Chapter 4 Section 2      m[1]: 4         m[2]: 2
22 # m[0]: Chapter 5 Section 12     m[1]: 5         m[2]: 12
```

Substitutions

Function	Effect
`re.sub(regex, subst, text)`	Substitute all occurrences of `regex` with `subst` in `text`
`re.sub(regex, subst, text, 3)`	Limit subs to first 3 occurrences of `regex`

Within `subst` the syntax `\1` refers to Group 1, `\1` refers to Group 2, etc.

Substitution Examples

```
# re_substitution.py:

print(f"text:\n{text}")
sub_text = re.sub(r"Chapter (\d+) Section (\d+)",
                  r"Chapter \1.\2", text)
print(f"sub_text:\n{sub_text}")
# sub_text:
# In Chapter 3.5 we will discuss the merits of dynamically typed
# languages. That Section should be studied as later in Chapter 4
# Section 1 we will cover static type systems with Chapter 4.2
# providing a summary of the trade-offs between static and
# dynamic. Chapter 5.12 begins discusion of logic programming...


print("\nre.sub() limit 3")
sub_text3 = re.sub(r"Chapter (\d+) Section (\d+)",
                   r"Chapter \1.\2", text, 2)
print(f"sub_text3:\n{sub_text3}")
# sub_text3:
# In Chapter 3.5 we will discuss the merits of dynamically
# typed languages. That Section should be studied as later
# in Chapter 4.1 we will cover static type systems with the
# following Chapter 4 Section 2 providing a summary of the trade-offs
# between static and dynamic. Chapter 5 Section 12 begins discusion of
# logic programming...
```

Python is Full of Goodies

Named Groups

Python regex groups can be named rather than numerically referenced

```
>>> details = '2018-10-25,car,2346'
>>> re.search(r'(?P<date>[^,]+),(?P<product>[^,]+)', details).groupdict()
{'date': '2018-10-25', 'product': 'car'}
```

Compiling Regexs

- Regexs must be compiled down to a finite state machine
- Mostly this is done automatically and cached for repeated use
- Sometimes its useful to do so manually as it *may* improve efficiency via

```
pattern = re.compile(r"^[a2-9tjqk]{5}$")
strs = re.findall(pattern,text)
```