STATS 700-002
Data analysis using Python

Lecture 0: Introduction and Administrivia
“Data science” has completely changed our world
Course goals

- Survey popular tools in academia/industry for data analysis and exploration
- Establish a broad background in programming
- Learn how to read documentation and quickly get new tools up and running
- Learn basic distributed computing frameworks

These tools will be obsolete some day...

...but not your ability to learn new frameworks and solve problems!
Course structure

Unit 1: Dealing with structured data
   regular expressions, retrieving web data, SQL

Unit 2: Data visualization and Introduction to the Command Line
   Creating plots with matplotlib, basics of the UNIX/Linux command line

Unit 3: Big data and parallel programming
   Process large data sets with Hadoop/Spark

Unit 4: Data analysis
   Modeling and analysis, TensorFlow

Schedule (tentative) and other information available on course webpage:
   umich.edu/~klevin/teaching/Fall2017/STATS700-002
Prerequisites

I assume that you have some background in programming and statistics

Come speak to me if:
- you have never used Python before
- you have never taken a probability or statistics course

This course is probably not for you if:
- you have no programming background

Good Python crash courses:
Course information

Instructor: Keith Levin
- Email: klevin@umich.edu
- Office: 272 WH
- OH: TuTh 2:30-4:00 in 313 WH or by appointment

GSI: Roger Fan
- Email: rogerfan@umich.edu
- OH: Tu 10:30-11:30 in SLC or by appointment

Textbook: None
- Readings posted to canvas and to the website

Grading: Four HWs
- ~25% per HW
- No midterm, no final
- No class project, owing to time constraints
- Late days (see syllabus)

See syllabus on Canvas or at umich.edu/~klevin/teaching/Fall2017/STATS700-002/syllabus.pdf
Policies

Don’t plagiarize!
● You may discuss homeworks with your fellow students...
● ...but you must submit your own work
● Disclose in your homework whom (if anyone) you worked with

Late homeworks are not allowed!
● Instead, we have “late days”, of which get 7, to spend as you please
● Note that the due date of the last homework is fixed to be the day of the final, so you may not use late days for that.

Refer to the syllabus for details.
Survey time!

1. Raise your hand if you were in 607A, either this semester (taught by Ambuj Tewari) or a previous semester.

2. Raise your hand if you have used Python before.

3. Raise your hand if you have used jupyter/iPython in the past.

4. Raise your hand if you have used the UNIX/Linux command line.

5. Raise your hand if you have used the Python `matplotlib` package.

6. Raise your hand if you prefer Canvas over a course webpage.
Things to do very soon:

Pick an editor/IDE for python
   or just use a text editor, or just write directly in jupyter

Familiarize yourself with jupyter:
   https://jupyter.readthedocs.io/en/latest/content-quickstart.html

Pick a version of Python to use
   You may use either 2.7 or 3.3+, though I will use 2.7 for all examples

Get a flux/fladoop username
   Fill out form here: http://arc-ts.umich.edu/hpcform/
   List me (Keith Levin, klevin@umich.edu) as your “advisor”
   Include a note that you are in STATS700-002 and you need Fladoop
Other things

HW1 is posted to canvas and the website. **Get started now!**

If you run into trouble, come to office hours for help

- But also please post to the discussion board on Canvas
- If you’re having trouble, at least one of your classmates is, too
- You’ll learn more by explaining things to each other than by reading stackexchange posts!
Lecture 1: Text encoding and regular expressions
Some slides adapted from C. Budak
Structured data

**Storage:** bits on some storage medium (e.g., hard-drive)

**Encoding:** how do bits correspond to symbols?

**Interpretation/meaning:** e.g., characters grouped into words

**Delimited files:** words grouped into sentences, documents

**Structured content:** metadata, tags, etc

**Collections:** databases, directories, archives (.zip, .gz, .tar, etc)
Structured data

**Storage**: bits on some storage medium (e.g., hard-drive)

**Encoding**: how do bits correspond to symbols?

**Interpretation/meaning**: e.g., characters grouped into words

**Delimited files**: words grouped into sentences, documents

**Structured content**: metadata, tags, etc

**Collections**: databases, directories, archives (.zip, .gz, .tar, etc)
Structured data

- **Storage**: bits on some storage medium (e.g., hard-drive)
- **Encoding**: how do bits correspond to symbols?
- **Interpretation/meaning**: e.g., characters grouped into words
- **Delimited files**: words grouped into sentences, documents
- **Structured content**: metadata, tags, etc
- **Collections**: databases, directories, archives (.zip, .gz, .tar, etc)

*Lectures 2 and 3*
Text data is ubiquitous

Examples:

- Biostatistics (DNA/RNA/protein sequences)
- Databases (e.g., census data, product inventory)
- Log files (program names, IP addresses, user IDs, etc)
- Medical records (case histories, doctors’ notes, medication lists)
- Social media (Facebook, twitter, etc)
How is text data stored?

Underlyingly, every file on your computer is just a string of bits...

...which are broken up into (for example) bytes...

...groups of which correspond to (in the case of text) characters.

```
cat
```
How is text data stored?

Some encodings (e.g., UTF-8 and UTF-16) use “variable-length” encoding, in which different characters may use different numbers of bytes.

- Not a concern in this course, but you should be aware of it!

We’ll concentrate on ASCII, which uses fixed-length encodings.
ASCII (American Standard Code for Information Interchange)

8-bit* fixed-length encoding, file stored as stream of bytes

Each byte encodes a character

   Letter, number, symbol or “special” characters (e.g., tabs, newlines, NULL)

**Delimiter**: one or more characters used to specify boundaries

   **Ex:** space (‘ ’, ASCII 32), tab (‘\t’, ASCII 9), newline (‘\n’, ASCII 10)

https://en.wikipedia.org/wiki/ASCII

*technically, each ASCII character is 7 bits, with the 8th bit reserved for error checking
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[NULL]</td>
<td>32</td>
<td>20</td>
<td>[SPACE]</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>[START OF HEADING]</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>[START OF TEXT]</td>
<td>34</td>
<td>22</td>
<td>*</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>[END OF TEXT]</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>[END OF TRANSMISSION]</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>[ENQUIRY]</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>[ACKNOWLEDGE]</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>[BELL]</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>[BACKSPACE]</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>[HORIZONTAL TAB]</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>[VERTICAL TAB]</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>[FORM FEED]</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>[CARRIAGE RETURN]</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>[SHIFT OUT]</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>[SHIFT IN]</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>[DATA LINK ESCAPE]</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>[DEVICE CONTROL 1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>[DEVICE CONTROL 2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>[DEVICE CONTROL 3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>[DEVICE CONTROL 4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>[NEGATIVE ACKNOWLEDGE]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>[SYNCHRONOUS IDLE]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>[ENG OF TRANS. BLOCK]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>[CANCEL]</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>[END OF MEDIUM]</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>[SUBSTITUTE]</td>
<td>58</td>
<td>3A</td>
<td>;</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>[ESCAPE]</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>[FILE SEPARATOR]</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td>l</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>[GROUP SEPARATOR]</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>}</td>
<td>125</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>[RECORD SEPARATOR]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>[UNIT SEPARATOR]</td>
<td>62</td>
<td>3F</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>\</td>
<td>127</td>
<td>7F</td>
<td>[DEL]</td>
</tr>
</tbody>
</table>
Caution!

Different OSs may follow slightly different conventions when saving text files!

Most common issue:
- UNIX/Linux/MacOS: newlines stored as ‘\n’
- DOS/Windows: stored as ‘\r\n’ (carriage return, then newline)

When in doubt, use a tool like *nix xxd (hexdump) to inspect raw bytes
xxd is also in MacOS; available in cygwin on Windows
Unicode

Universal encoding of (almost) all of the world’s writing systems

Each symbol is assigned a unique code point, a four-hexadecimal digit number

- Unique number assigned to a given character U+XXXX
- ‘U+’ for unicode, XXXX is the code point (in hexadecimal)
- Example: □=U+1F60E, ₤=U+2230; http://www.unicode.org/ for more

Variable-length encoding

- UTF-8: 1 byte for first 128 code points, 2+ bytes for higher code points
- Result: ASCII is a subset of UTF-8

Newer versions (i.e., 3+) of python encode scripts in unicode by default
Matching text: regular expressions ("regexes")

Suppose I want to find all addresses in a big text document. How to do this?

Regexes allow concise specification of characters, strings, patterns to match

Specifics vary from one program to another (perl, grep, vim, emacs), but the basics that you learn in this course will generalize with minimal changes.

Image credit: Randall Munroe, XKCD #208
Regular expressions in Python

Three basic functions:

- `re.match()`: tries to apply regex at start of string.
- `re.search()`: tries to match regex to any part of string.
- `re.findall()`: finds all matches of pattern in the string.

See [https://docs.python.org/2/library/re.html](https://docs.python.org/2/library/re.html) or [https://docs.python.org/3/library/re.html](https://docs.python.org/3/library/re.html) for additional information and more functions (e.g., splitting and substitution).

Gentle introduction: [https://docs.python.org/2/howto/regex.html#regex-howto](https://docs.python.org/2/howto/regex.html#regex-howto)
Pattern matches beginning of string1, and returns match object.

Pattern matches string2, but not at the beginning, so match fails and returns None.
Pattern matches beginning of string1, and returns match object.

Pattern matches string2 (not at the beginning!) and returns match object.

Pattern does not match anything in string3, returns None.
Pattern matches string1 once, returns that match.

Pattern matches string2 in three places; returns list of three instances of cat.

Pattern does not match anything in string3, returns empty list.
What about more complicated matches?

Regexes would not be very useful if all we could do is search for strings like ‘cat’

Power of regexes lies in specifying complicated patterns. Examples:

- Whitespace characters: ‘\t’, ‘\n’, ‘\r’
- Matching classes of characters (e.g., digits, whitespace, alphanumerics)
- Special characters: .  ^  $  *  +  ?  {  }  [  ]  \  |  (  )
  ○ We’ll discuss meaning of special characters shortly

Special characters must be escaped with backslash ‘\’

**Ex:** match a string containing a backslash followed by dollar sign:

```python
1 re.match('\\\\\\\\$\', '\$')
<_sre.SRE_Match at 0x111114dac0>
```
Gosh, that was a lot of backslashes...

Regular expressions often written as `r'text'`

Prepending the regex with `'r'` makes things a little more sane

- `'r'` for **raw text**
- Prevents python from parsing the string
- Avoids escaping every backslash
- **Ex:** `'\n'` is a single-character string, a new line, while `r'\n'` is a two-character string, equivalent to `\\n`.

```
1 re.match(r'\\\\$', '\$')
<_sre.SRE_Match at 0x11114dd30>
```

```
1 re.match('\\\\\\\\\\\\\\$', '\$')
<_sre.SRE_Match at 0x11114dac0>
```

**Note:** Python also includes support for unicode regexes
More about raw text

Recall ‘\n’ is a single-character string, a new line, while \n\n is a two-character string, equivalent to ‘\n’.

But...

```python
beatles = "hello\ngoodbye"
1 re.findall(r'\n', beatles)
['\n']
```

Has to do with Python string parsing.

From the documentation (emphasis mine): “This is complicated and hard to understand, so it’s highly recommended that you use raw strings for all but the simplest expressions.”
Special characters: basics

Some characters have special meaning

These are: . ^ $ * + ? { } [ ] \ | ( )

We’ll talk about some of these today, for others, refer to documentation

**Important:** special characters must be escaped to match literally!

```
1 re.findall(r'$2', '2$2')
[]
```

```
1 re.findall(r'\$2', '2$2')
['$2']
```
Special characters: sets and ranges

Can match “sets” of characters using square brackets:

- ‘[aeiou]’ matches any one of the characters ‘a’, ‘e’, ‘i’, ‘o’, ‘u’
- ‘[^aeiou]’ matches any one character NOT in the set.

Can also match “ranges”:

- Ex: ‘[a-z]’ matches lower case letters
  - Ranges calculated according to ASCII numbering
- Ex: ‘[0-9A-Fa-f]’ will match any hexadecimal digit
- Escaped ‘-’ (e.g. ‘[a\-z]’) will match literal ‘-’
  - Alternative: ‘-’ first or last in set to match literal

Special characters lose special meaning inside square brackets:

- Ex: ‘[ (+*)]’ will match any of ‘(’, ‘+’, ‘*’, or ‘)’
- To match ‘^’ literal, make sure it isn’t first: ‘[ (+*) ^]’
Special characters: single character matches

\^: matches beginning of a line

\$: matches end of a line (i.e., matches “empty character” before a newline)

\.: matches any character other than a newline

\s: matches whitespace (spaces, tabs, newlines)

\d: matches a digit (0,1,2,3,4,5,6,7,8,9), equivalent to r\[0-9]\`

\w: matches a “word” character (number, letter or underscore ‘_’)

\b: matches boundary between word (\w) and non-word (\W) characters
Example: beginning and end of lines, wildcards

```
1 pat = r'\^b.d$'
2 re.findall(pat, 'bad')
['bad']
```

Matching fails because of 's' at end of string, which means that 'd' is not followed by end-of-line.

```
1 re.findall(pat, 'bid')
['bid']
```

```
1 re.findall(pat, 'bids')
[]
```

Matching fails because of 'a' at start of string, which means that 'b' is not the start of the string.

```
1 re.findall(pat, 'abad')
[]
```
Example: whitespace and boundaries

```python
1 string1 = "c\ta t\ns\n"
2 re.findall(r'\s', string1)
[', , , , ]
1 re.findall(r'\s\b', string1)
[', , , ]
```

'\s' matches any whitespace. That includes spaces, tabs and newlines.
The trailing newline in string1 isn’t matched, because it isn’t followed by a whitespace-word boundary.
Character classes: complements

`\s`, `\d`, `\w`, `\b` can all be complemented by capitalizing:

`\S` : matches anything that **isn’t** whitespace

```
re.findall(r'\S', "c\ta t\ns\n")
['c', 'a', 't', 's']
```

`\D` : matches any character that **isn’t** a digit

```
re.findall(r'\D', "abc123 \t\n")
['a', 'b', 'c', ' ', '\t', '\n']
```

`\W` : matches any **non-word** character

```
re.findall(r'\W', "abc123 \t\n$_*".)
[ ', '\t', '\n', '$', '*', '.']
```

`\B` : matches **NOT** at a word boundary

```
re.findall(r'\B\d\B', "1 2X a3 747 ")
['4']
```
Matching and repetition

'*' : zero or more of the previous item

'+' : one or more of the previous item

'?': zero or one of the previous item

'{4}': exactly four of the previous item

'{3,}': three or more of previous item

'{2,5}': between two and five (inclusive) of previous item
Test your understanding

Which of the following will match `r'^\d{2,4}\s'$`?

'7 a1'

'747 Boeing'

'C7777 C7778'

'12345 '

'1234\tqq'

'Boeing 747'
Test your understanding

Which of the following will match `r’ ^\d{2,4} \s’`?

- `'7 al'` ✗
- `'747 Boeing'` ✨
- `'C7777 C7778'` ✗
- `'12345 '` ✗
- `'1234\tqq'` ✨
- `'Boeing 747'` ✗
Or clauses: |

'|' ("pipe") is a special character that allows one to specify "or" clauses

Example: I want to match the word "cat" or the word "dog"

Solution: `'(cat|dog)'`

Note: parentheses are not strictly necessary here, but in more complicated expressions they often are, so it’s a good habit to just use them always!
Or clauses: | is lazy!

What happens when an expression using pipe can match many different ways?

What’s going on here?!

Matching with ‘|’ is lazy
- Tries to match each regex separated by ‘|’, in order, left to right.
- As soon as it matches something, it returns that match...
- ...and starts trying to make another match.
- **Note:** this behavior can be changed using flags. Refer to documentation.
Matching and greediness

Pipe operator ‘|’ is lazy. But, confusingly, python `re` module is usually greedy:

```python
1 re.findall(r'a+', 'aaaaaa')
['aaaaaa']
```

‘a+’ gobbles up the whole string, because Python regexes are greedy.

```python
1 re.findall(r'a+?', 'aaaaaa')
['a', 'a', 'a', 'a', 'a', 'a', 'a']
```

‘?’ modifies operators like ‘+’ and ‘*’ to not be greedy, and we get lazy matching, like when using ‘|’.

From the documentation: Repetition qualifiers (*, +, ?, {m,n}, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix ?, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression (?:a{6})* matches any multiple of six 'a' characters.
Extracting groups

Python `re` lets us extract things we matched and use them later

**Example:** matching the user and domain in an email address

```python
string1 = "My Michigan email is klevin@umich.edu"

m = re.search(r'([\w.-]+)@([\w.-]+)', string1)

m.group()  # 're.search' returns a match object. The group attribute is the whole string that was matched.

m.group(1)  # Can access groups (parts of the regex in parentheses) in numerical order. Each set of parentheses gets a group, in order from left to right.

m.group(2)
```

**Note:** `re.findall` has similar functionality!
Backreferences

Can refer to an earlier match *within the same regex!* \N, where N is a number, references the N-th group

Example: find strings of the form ‘\x \x’, where X is any non-whitespace string.

```python
1 m = re.search(r'(\S+) \1', 'cat cat')
2 m.group()
'cat cat'
```

```python
1 m = re.search(r'(\S+) \1', 'cat dog')
2 m is None
True
```
Backreferences

Backrefs allows very complicated pattern matching!

Test your understanding:
Describe what strings `(\d+) ([A-Z]+): \1+\2` matches?
What about `([a-zA-Z]+).*\1`?

Tougher question:
Is it possible to write a regular expression that matches palindromes?
Better answer: ...but if your matcher provides enough bells and whistles...
Options provided by Python \texttt{re} module

Optional flag modifies behavior of \texttt{re.findall}, \texttt{re.search}, etc.

\textbf{Ex:} \texttt{re.search(r'dog', 'DOG', re.IGNORECASE)} matches.

\texttt{re.IGNORECASE} : ignore case when forming a match.

\texttt{re.MULTILINE} : `^`, `$` match start/end of any line, not just start/end of string

\texttt{re.DOTALL} : `.` matches any character, \textbf{including} newline.

See \url{https://docs.python.org/2/library/re.html#contents-of-module-re} for more.
Debugging

When in doubt, test your regexes!

- A bit of googling will find you lots of tools for doing this
- Compiling and then using the re.DEBUG flag can also be helpful
- compiling also good for using a regex repeatedly, like in your homework

```python
regex = re.compile(r'cat|dog|bird')
regex.findall("It's raining cats and dogs.")

[ 'cat', 'dog' ]

regex.match("cat bird dog")
<_sre.SRE_Match at 0x1117dd780>

regex.search("nothing to see here.") is None

True
```
Readings (this lecture)

Required:
  Severance Chapter 11: Regular expressions

Recommended:
  Python regex documentation: https://docs.python.org/2/howto/regex.html
  Jupiter documentation: https://jupyter.readthedocs.io/en/latest/content-quickstart.html
Readings (next lecture)

Required:
    Severance Chapter 12 (HTTP, HTML), Chapter 13 (XML, JSON)
    BeautifulSoup documentation (just Quick Start)
        https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Recommended:
    BeautifulSoup documentation (everything up to sections about CSS)
        https://www.crummy.com/software/BeautifulSoup/bs4/doc/