Lecture 12: 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)

Increasing structure
Structured data

Today

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 13 and 14
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...

...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.

We’ll concentrate (today, at least) 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 (\x20, 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>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>[DEVICE CONTROL 2]</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>[DEVICE CONTROL 3]</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>[DEVICE CONTROL 4]</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>[NEGATIVE ACKNOWLEDGE]</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>[SYNCHRONOUS IDLE]</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>[ENG OF TRANS. BLOCK]</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</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></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>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>[UNIT SEPARATOR]</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 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 UNIX/Linux 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/](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

Note: the sunglasses emoji doesn’t render in PDF conversion, so it is an image, here.
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 for matching patterns in text

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.
Regular expressions in Python: the re package

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/3/library/re.html for additional information and more functions (e.g., splitting and substitution).

Gentle introduction: https://docs.python.org/3/howto/regex.html#regex-howto
Pattern matches beginning of \texttt{string1}, and returns match object.

Pattern matches \texttt{string2}, but \textbf{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`. 

True
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:

```
1 re.match('\\\\\\\\$', '\$')
<_sre.SRE_Match at 0x11114d4c0>
```
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`.

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

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

But...

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!

```python
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']

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

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

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

- `'.'` matches `'a'`, and start- and end-lines match correctly.
- `'.'` matches `'i'`, and start- and end-lines match correctly.
- Matching fails because of `'s'` at end of string, which means that `'d'` is not followed by end-of-line.
- Matching fails because of `'a'` at start of string, which means that `'b'` is not the start of the string.
Example: whitespace and boundaries

```
1  string1 = 'c\ta t\ns\n'
2  print(string1)
```

```
re.findall(r'\s', string1)
['\t', ' ', '\n', '\n']
```

```
re.findall(r'\s\b', string1)
['\t', ' ', '\n']
```

'\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

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

\W' : matches any non-word character

\B' : matches NOT at a word boundary
Matching and repetition

`*` : zero or more of the previous item
`+` : one or more of the previous item
`?` : zero or one of the previous item

```
1 re.findall(r'ca*t', "ct cat caat caaat")
[['ct', 'cat', 'caat', 'caaat']]
```

```
1 re.findall(r'ca+t', "ct cat caat caaat")
[['cat', 'caat', 'caaat']]
```

```
1 re.findall(r'ca{4}', "ct cat caat caaat")
[['caat']]
```

```
1 re.findall(r'ca{3,}', "ct cat caat caaat")
[['cat', 'caat']]
```

```
1 re.findall(r'ca{2,5}', "ct cat caat caaat")
[['caat']]
```

`{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\tq qq'

'Boeing 747'
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'
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 parentheses tend to make for easier reading and avoid possible ambiguity. 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?!

```
re.findall(r'a|aa|aaa', "aaaa")
```

[ 'a', 'a', 'a', 'a', 'a' ]

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:

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

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

```
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

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

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

m.group()  # 'klevin@umich.edu'

m.group(1)  # 'klevin'

m.group(2)  # 'umich.edu'
```

- `'re.search' returns a match object. The group attribute is the whole string that was matched.
- 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.

**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.

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

'cat cat'
```

```
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’?
- What about ‘([a-zA-Z]+) .*\1’?
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 `re` module

Optional flag modifies behavior of `re.findall`, `re.search`, etc.

**Ex:** `re.search(r‘dog’, ‘DOG’, re.IGNORECASE)` matches.

`re.IGNORECASE`: ignore case when forming a match.

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

`re.DOTALL`: `.` matches any character, **including** newline.

See [https://docs.python.org/2/library/re.html#contents-of-module-re](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
```