STATS 701
Data Analysis using Python

Lecture 15: Text Encoding and Regular Expressions

Some slides adapted from C. Budak

Structured data

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

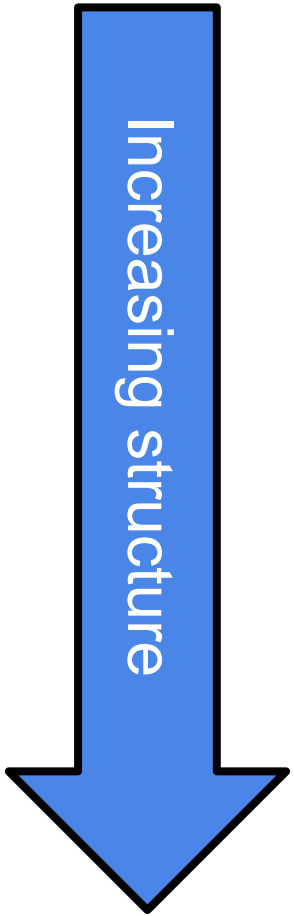
Encoding: how do bits correspond to symbols?

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

Today

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

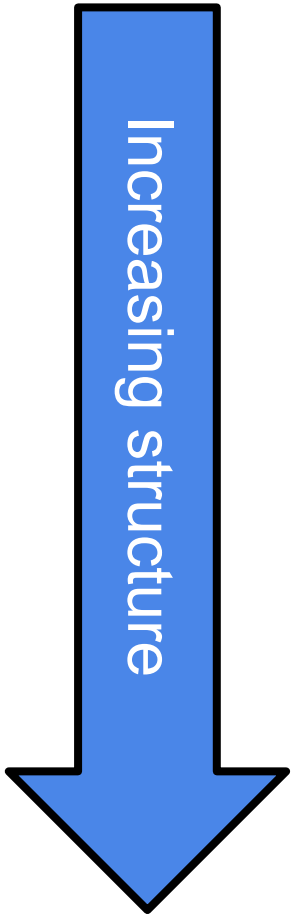
Encoding: how do bits correspond to symbols?

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)



Structured data

Today

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

Encoding: how do bits correspond to symbols?

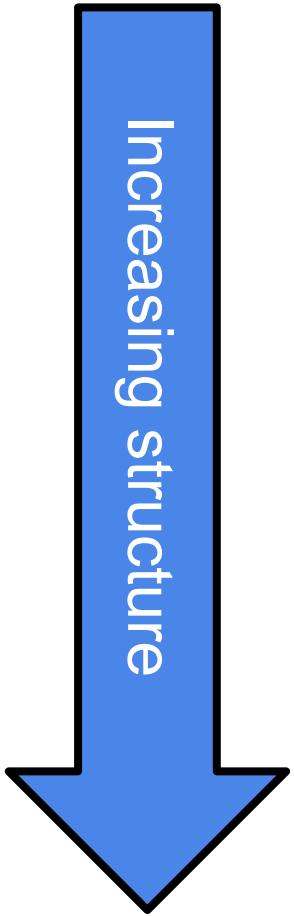
Interpretation/meaning: e.g., characters grouped into words

Delimited files: words grouped into sentences, documents

Structured content: metadata, tags, etc

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

Lectures 16 and 17



Text data is ubiquitous

Examples:

Biostatistics (DNA/RNA/protein sequences)

Databases (e.g., census data, product inventory)

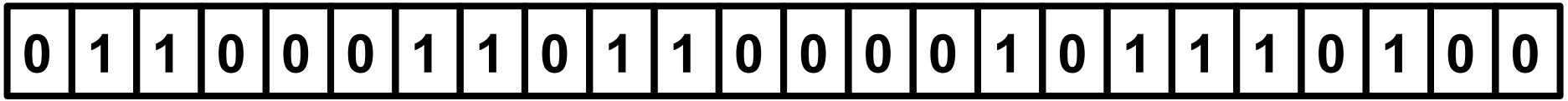
Log files (program names, IP addresses, user IDs, etc)

Medical records (case histories, doctors' notes, medication lists)

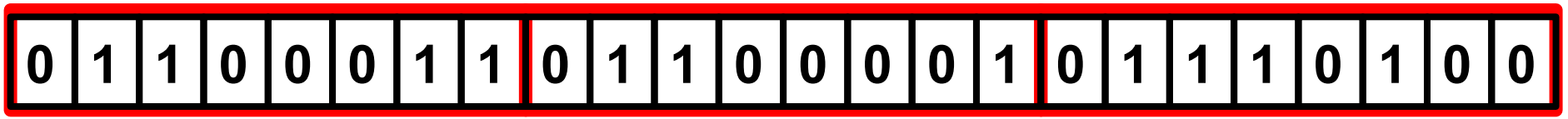
Social media (Facebook, twitter, etc)

How is text data stored?

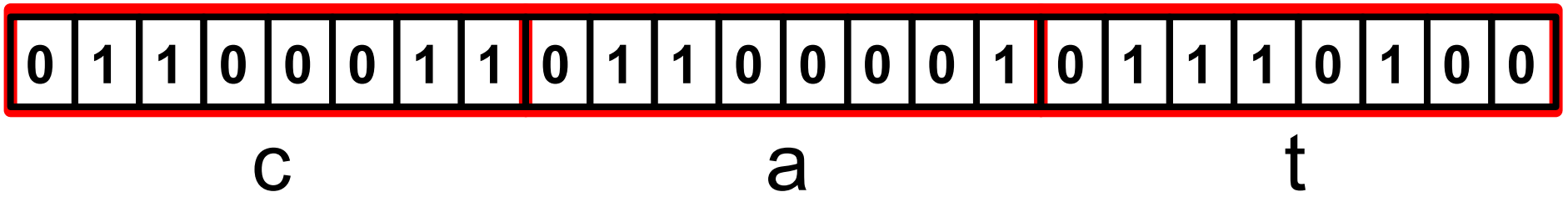
Underlyingly, every file on your computer is just a string of bits…



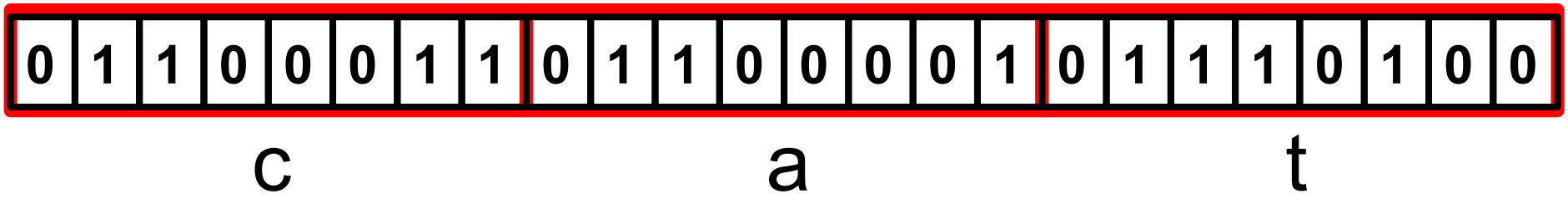
...which are broken up into (for example) bytes…



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



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 (`' '`, 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

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

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/ 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 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.

Image credit: Randall Munroe, XKCD #208

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/2/howto/regex.html#regex-howto

```
1 help(re.match)
```

```
Help on function match in module re:

match(pattern, string, flags=0)
    Try to apply the pattern at the start of the string, returning
    a match object, or None if no match was found.
```

```
1 pat = 'cat'
2 string1 = 'cat on mat'
3 string2 = 'raining cats and dogs'
4 re.match(pat, string1)
```

```
<_sre.SRE_Match at 0x11112abf8>
```

```
1 re.match(pat, string2) is None
```

```
True
```

Pattern matches beginning of `string1`, and returns match object.

Pattern matches `string2`, but **not** at the beginning, so match fails and returns None.

```
1 help(re.search)
```

```
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
```

```
1 pat = 'cat'
2 string1 = 'cat on mat'
3 string2 = 'raining cats and dogs'
4 string3 = 'abracadabra'
5 re.search(pat,string1)
```

```
<_sre.SRE_Match at 0x111148030>
```

```
1 re.search(pat,string2)
```

```
<_sre.SRE_Match at 0x111148100>
```

```
1 re.search(pat,string3) is None
```

```
True
```

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`.

```
help(re.findall)
```

```
Help on function findall in module re:

findall(pattern, string, flags=0)
    Return a list of all non-overlapping matches in the string.

    If one or more groups are present in the pattern, return a
    list of groups; this will be a list of tuples if the pattern
    has more than one group.

    Empty matches are included in the result.
```

```
pat = 'cat'
string1 = 'cat on mat'
string2 = 'one cat, two cats, three cats'
string3 = 'abracadabra'
re.findall(pat,string1)
```

```
['cat']
```

```
re.findall(pat,string2)
```

```
['cat', 'cat', 'cat']
```

```
re.findall(pat,string3)
```

```
[]
```

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 0x11114dac0>
```

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 `r'\n'` is a two-character string, equivalent to `'\\n'`.

But…

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

```
['\n']
```

```
1 re.findall('\\n', beatles)
```

```
['\n']
```

```
1 re.findall('\\\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']
```

`'.'` matches `'a'`, and start- and end-lines match correctly.

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

```
['bid']
```

`'.'` matches `'i'`, and start- and end-lines match correctly.

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

```
[]
```

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

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

```
[]
```

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 re.findall(r'\s', string1)
```

```
['\t', ' ', '\n', '\n']
```

'\s' matches any whitespace. That includes spaces, tabs and newlines.

```
1 re.findall(r'\s\b', string1)
```

```
['\t', ' ', '\n']
```

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

```
1 re.findall(r'\S', "c\ta t\ns\n")
```

```
['c', 'a', 't', 's']
```

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

```
1 re.findall(r'\D', "abc123 \t\n")
```

```
['a', 'b', 'c', ' ', '\t', '\n']
```

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

```
1 re.findall(r'\W', "abc123 \t\n_$*.")
```

```
[' ', '\t', '\n', '$', '*', '.']
```

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

```
1 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

```
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']
```

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

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

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

```
1 re.findall(r'ca{2}t', "ct cat caat caaat")
```

```
['caat']
```

```
1 re.findall(r'ca{1,2}t', "ct cat caat caaat")
```

```
['cat', 'caat']
```

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 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 in more complicated expressions they often are, so it’s a good habit to just use them always.

```
1 re.findall(r'(cat|dog)', "cat")
```

```
['cat']
```

```
1 re.findall(r'(cat|dog)', "dog")
```

```
['dog']
```

```
1 re.findall(r'(cat|dog)', "cat\ndog")
```

```
['cat', 'dog']
```

Or clauses: | is lazy!

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

What's going on here?!

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

```
['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']
```

`'?'` 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

```
1 string1 = "My Michigan email is klevin@umich.edu"
2 m = re.search(r'([\w.-]+)@([\w.-]+)', string1)
3 m.group()
```

```
'klevin@umich.edu'
```

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

```
1 m.group(1)
```

```
'klevin'
```

```
1 m.group(2)
```

```
'umich.edu'
```

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'`?

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?
Answer: Strictly speaking, no. https://en.wikipedia.org/wiki/Regular_language
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 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

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

```
['cat', 'dog']
```

```
1 regex.match("cat bird dog")
```

```
<_sre.SRE_Match at 0x1117dd780>
```

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

```
True
```

Readings

Required:

Severance Chapter 11: Regular expressions

Recommended:

Python regex documentation: https://docs.python.org/2/howto/regex.html