STATS 507
Data Analysis in Python

Week 8: Text Encoding, Regular Expressions, Network Programming, and HTML

Adapted from slides by Keith Levin and C. Budak

Structured data

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

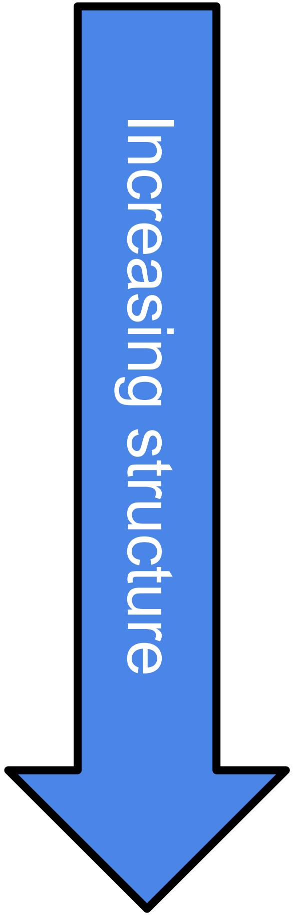
Encoding: how do bits correspond to symbols?

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)

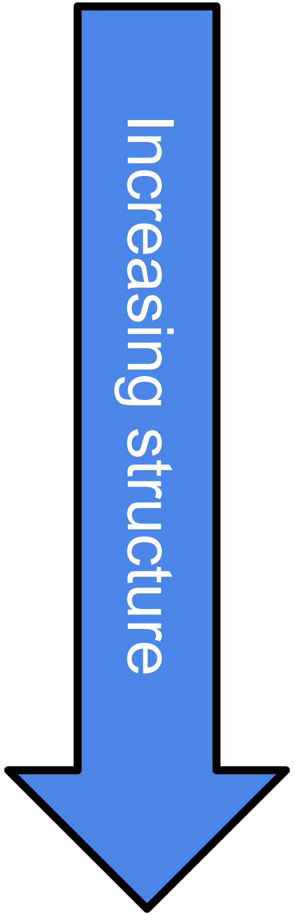
Encoding: how do bits correspond to symbols?

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

Delimited files: words grouped into sentences, documents

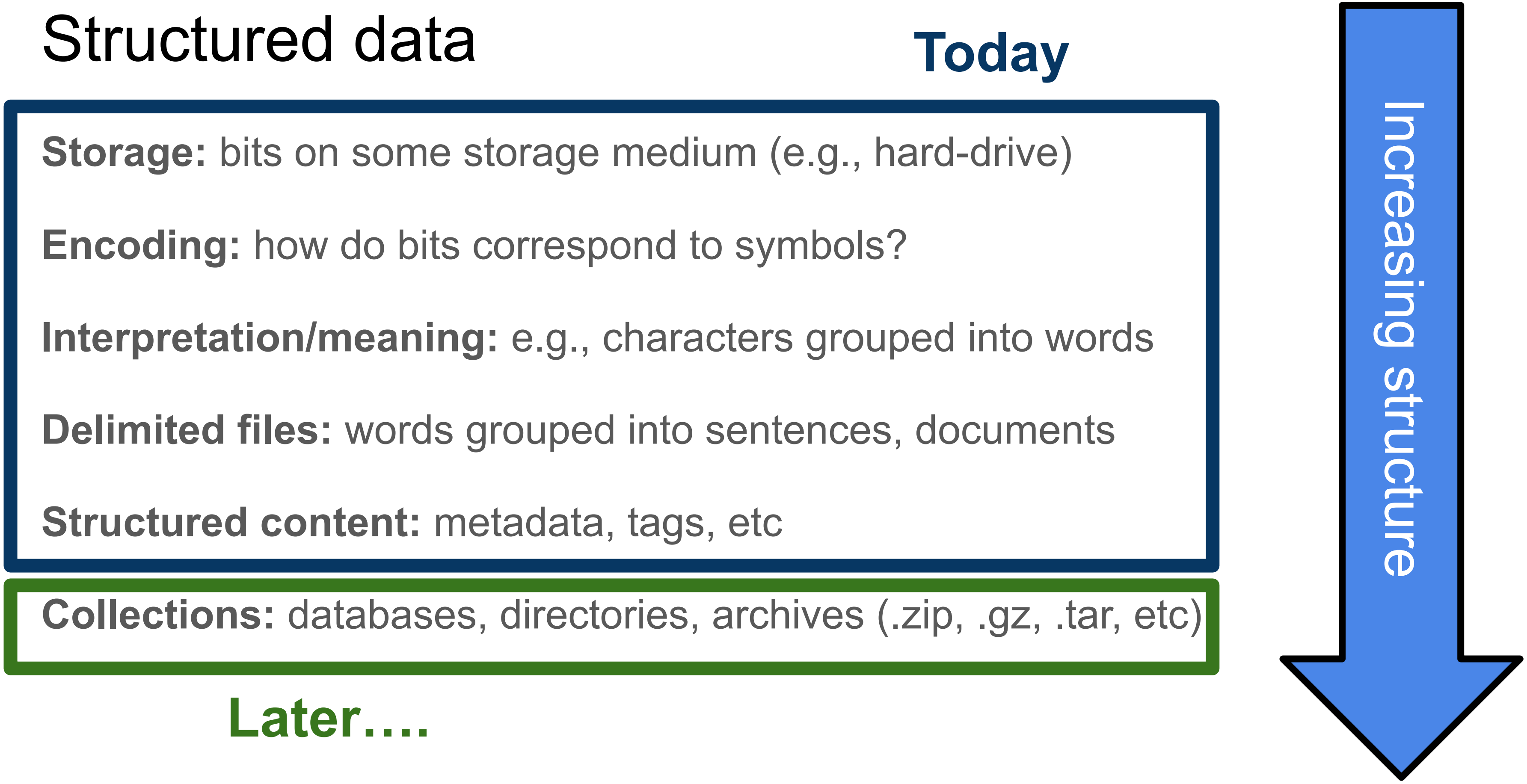
Structured content: metadata, tags, etc

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



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)

Later….

Text data is ubiquitous

Examples:

Biostatistics (DNA/RNA/protein sequences)

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

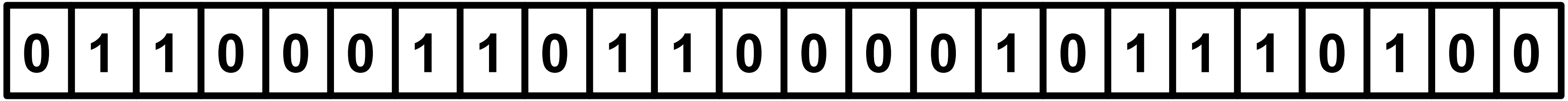
Log files (program names, IP addresses, user IDs, etc)

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

Social media (Facebook, twitter, etc)

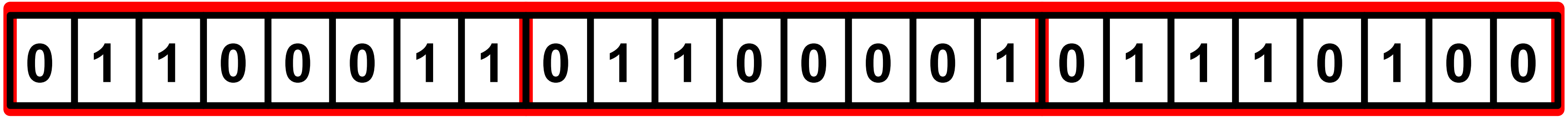
How is text data stored?

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



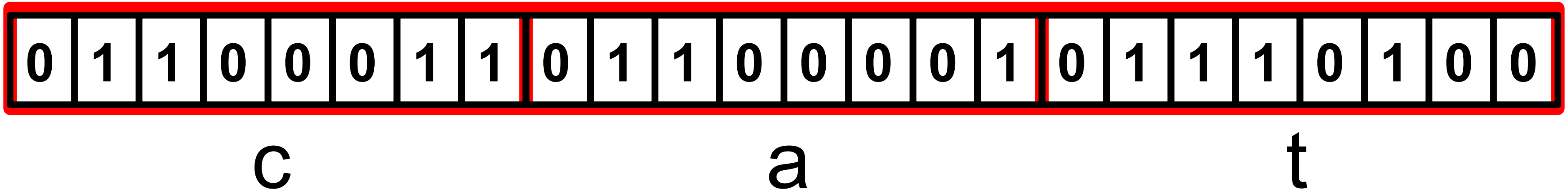
0 1 1 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0

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



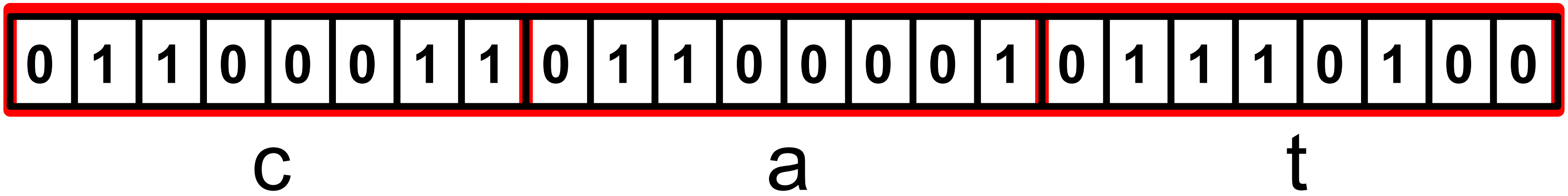
0 1 1 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0

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



0 1 1 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0

How is text data stored?



0 1 1 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0

c a t

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

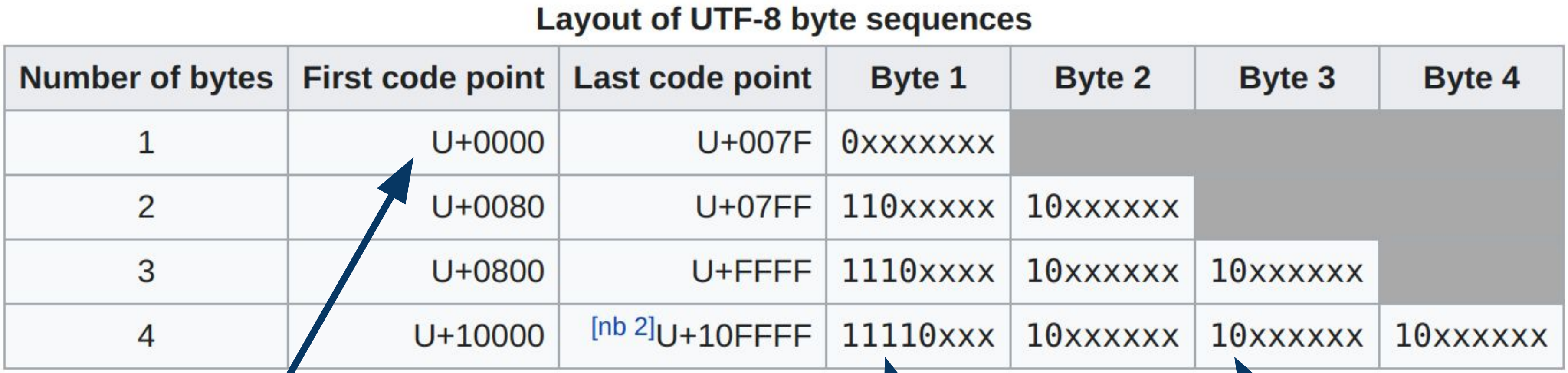
- Unique number assigned to a given character U+XXXX
- 'U+' for unicode, XXXX is the code point (in hexadecimal)
- Example: 😎=U+1F60E, ∰=U+2230; http://www.unicode.org/ for more

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 assume scripts are encoded in unicode by default

UTF-8 in depth



Layout of UTF-8 byte sequences

Number of bytes	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	U+0000	U+007F	0xxxxxxx			
2	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	U+10000	[nb 2]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Backward compatible with ASCII

The number of 1s in the prefix encodes the number of bytes

A prefix of 10 means this byte is not the start of a new character

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.

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.
PERL!
TAP TAP

Image credit: Randall Munroe, XKCD #208

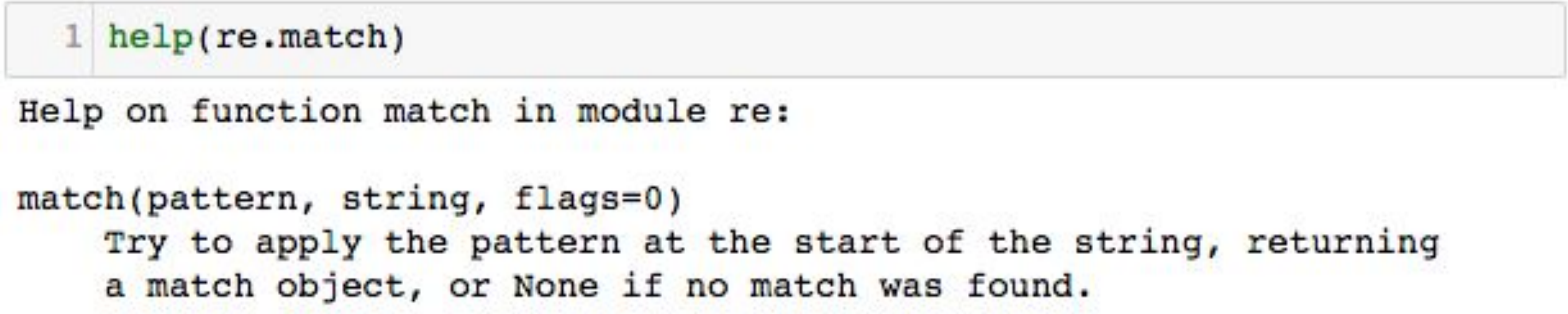
Regular expressions in Python: the `re` package

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.

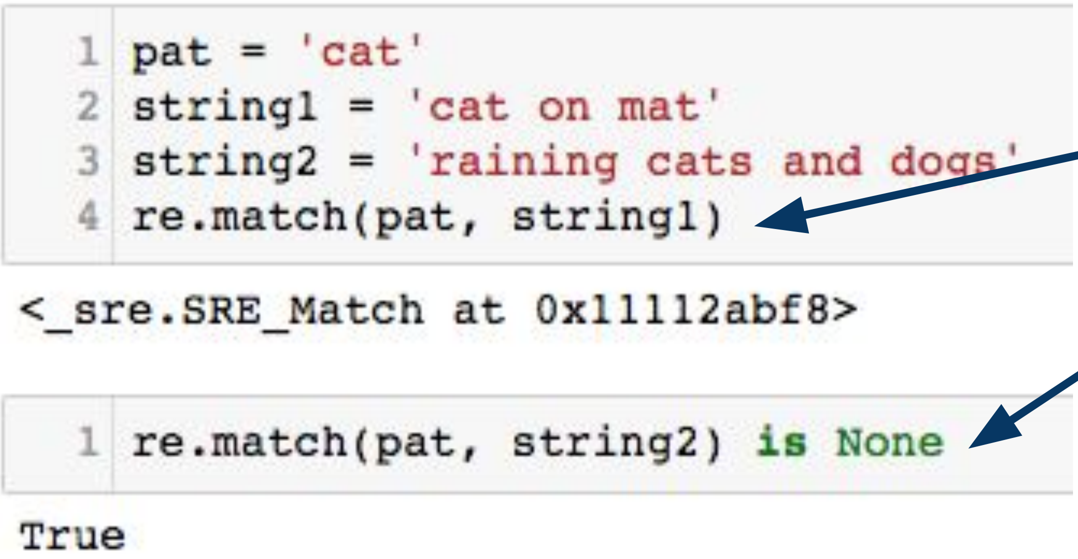
See https://docs.python.org/3/library/re.html for additional information and more functions (e.g., splitting and substitution).

Gentle introduction: https://docs.python.org/3/howto/regex.html#regex-howto



```
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.
```



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

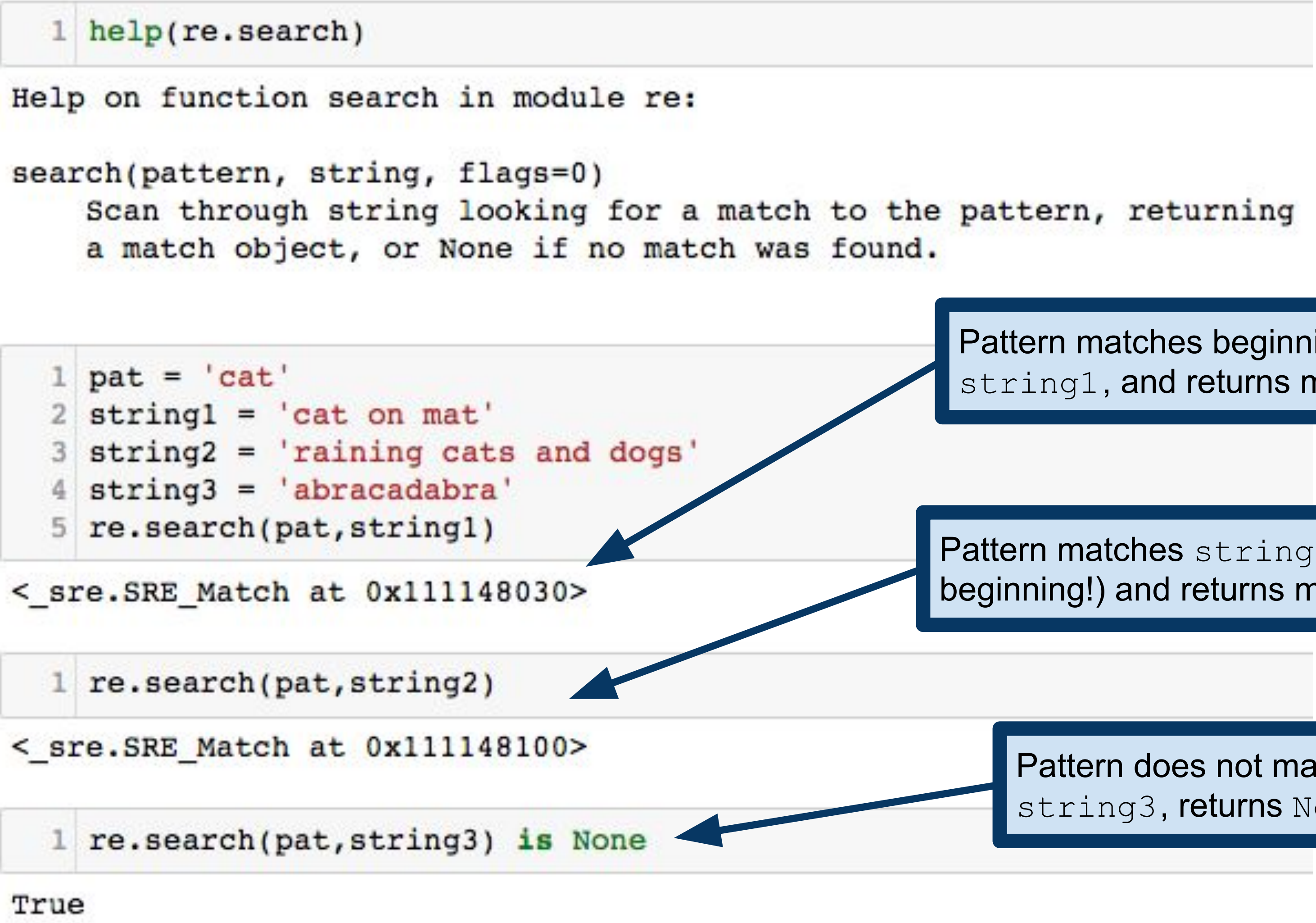
```
<_sre.SRE_Match at 0x11112abf8>
```

```
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.



```
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.
```

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

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

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

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

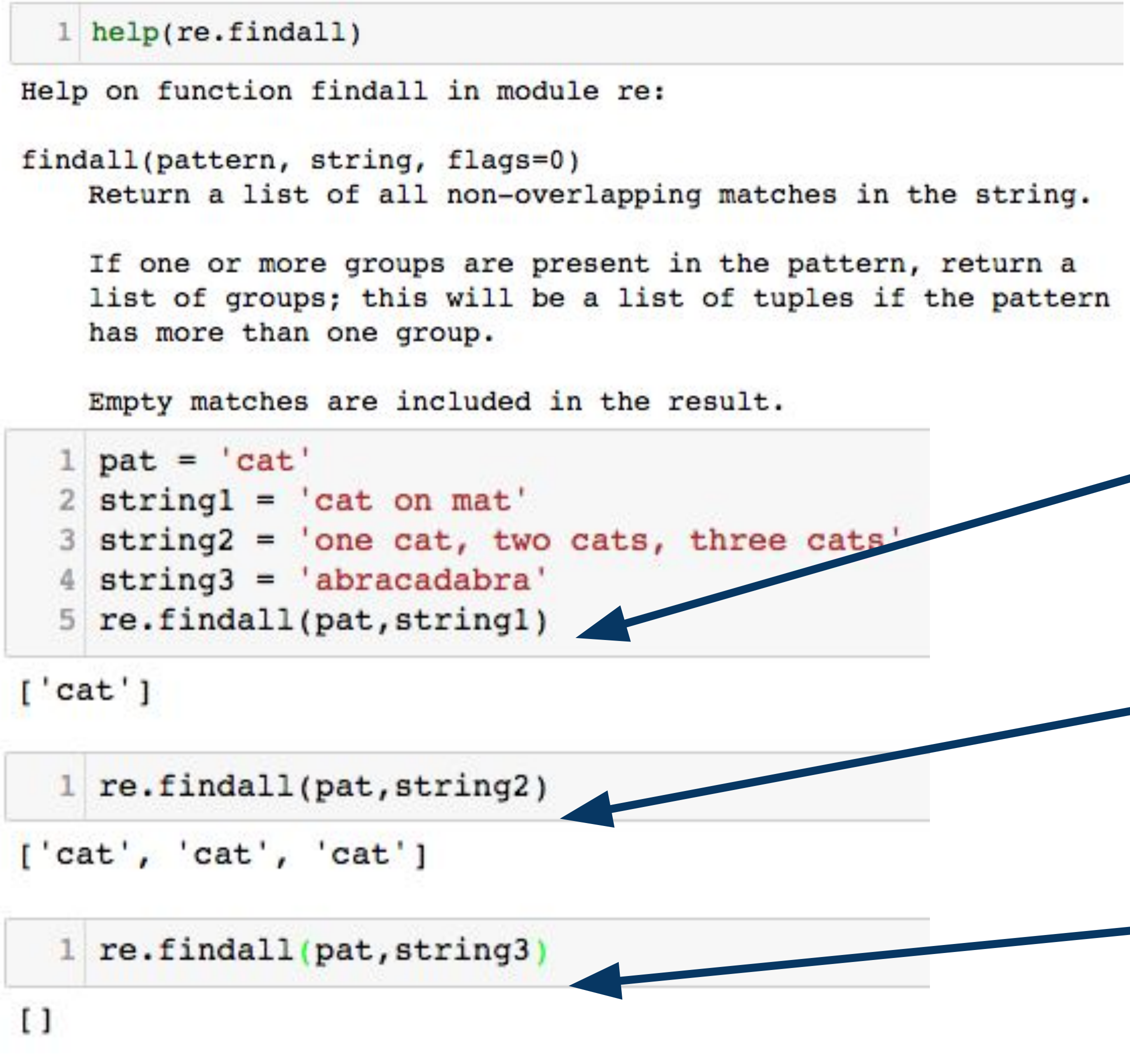
```
re.search(pat,string3) is None
```

```
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’`

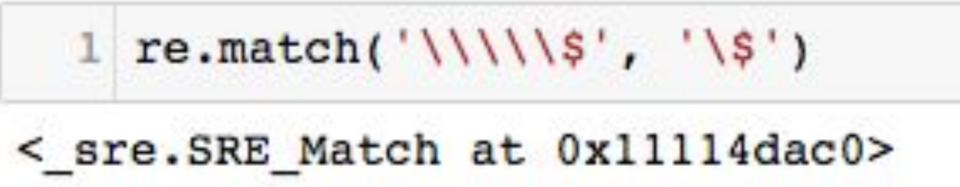
Matching classes of characters (e.g., digits, whitespace, alphanumerics)

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:



```
re.match('\\\\\\$', '\$')
```

```
<_sre.SRE_Match at 0x1114dac0>
```

Note that

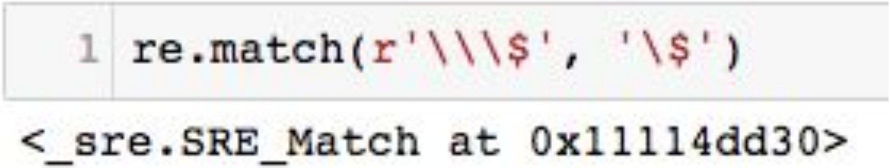
`'\$' == '\\$'`

Gosh, that was a lot of backslashes...

Regular expressions often written as `r‘text’`

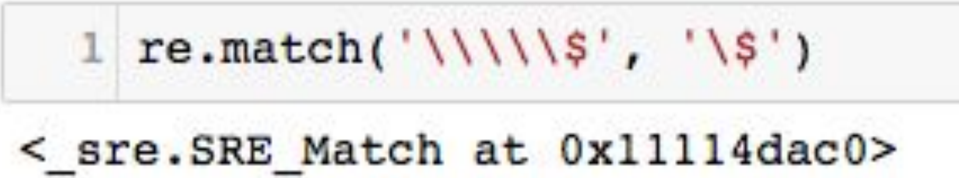
Prepending the regex with `‘r’` makes things a little more sane

- `’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’`.



```
re.match(r'\\\$', '\$')
```

```
<_sre.SRE_Match at 0x11114dd30>
```



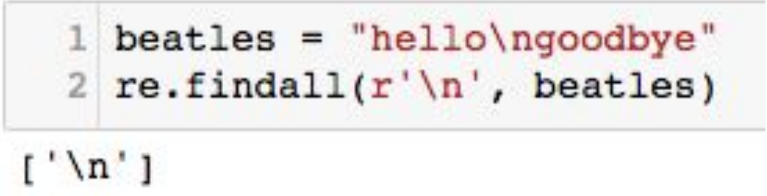
```
re.match('\\\\\$', '\$')
```

```
<_sre.SRE_Match at 0x11114dac0>
```

Note: Python also includes support for unicode regexes

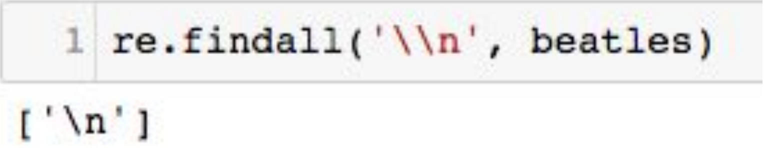
More about raw text

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



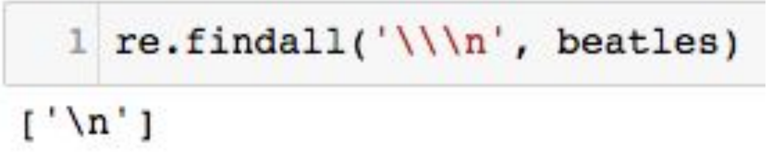
```
beatles = "hello\ngoodbye"
re.findall(r'\n', beatles)
```

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



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

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



```
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.”

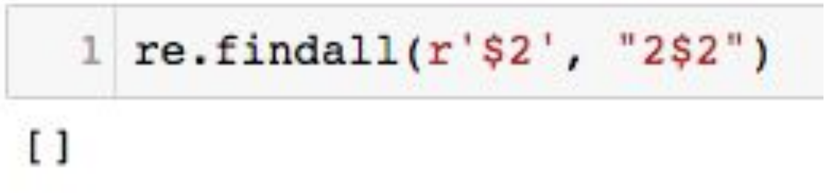
Special characters: basics

Some characters have special meaning

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

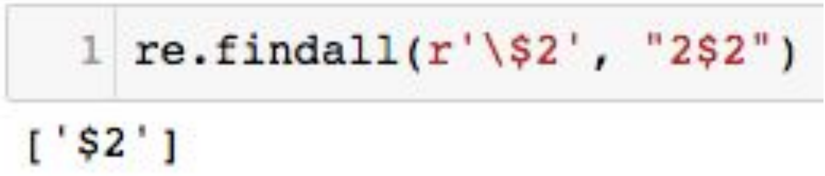
We'll talk about some of these today, for others, refer to documentation

Important: special characters must be escaped to match literally!



```
re.findall(r'$2', "2$2")
```

```
[]
```



```
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

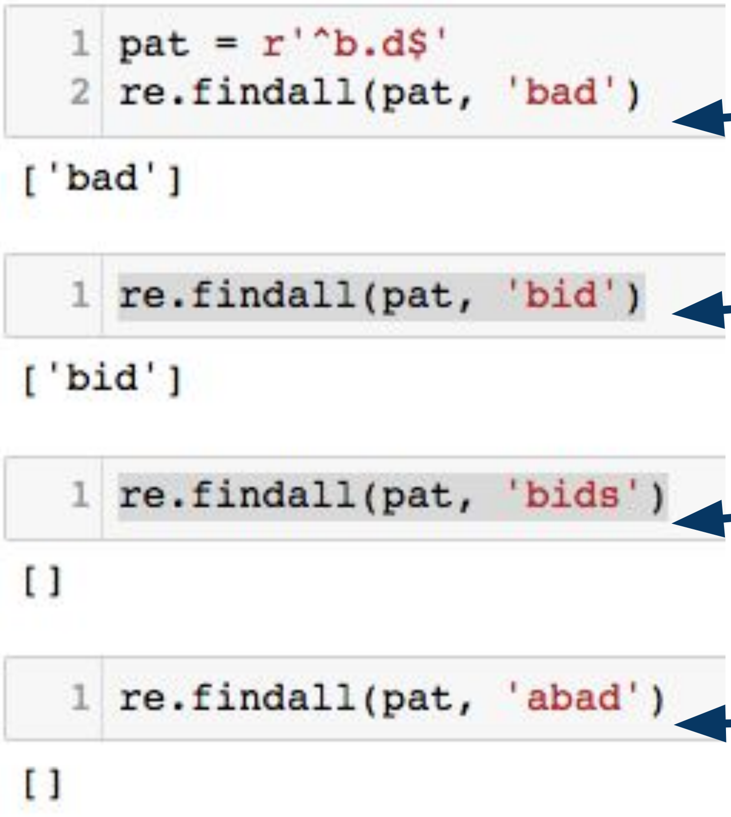
`‘\s’` : matches whitespace (spaces, tabs, newlines)

`‘\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



```
pat = r'^b.d$'
re.findall(pat, 'bad')
```

```
['bad']
```

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

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

```
['bid']
```

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

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

```
[]
```

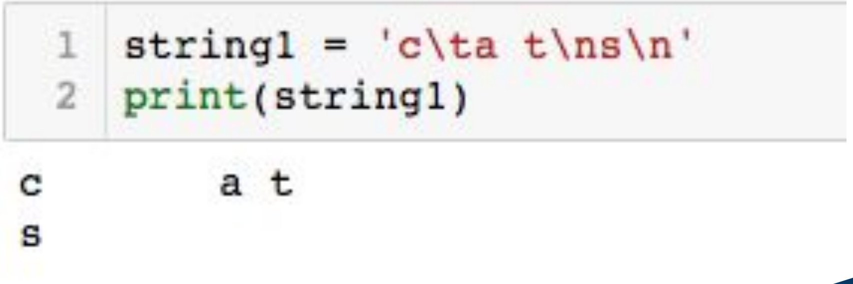
Matching fails because of 's' at end of string, which means that 'd' is not followed by end-of-line.

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

```
[]
```

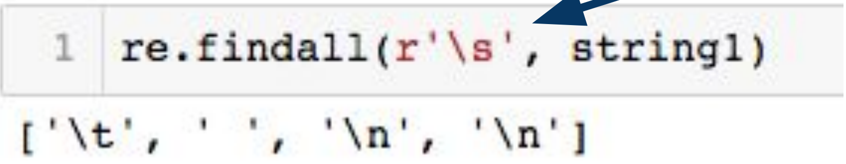
Matching fails because of 'a' at start of string, which means that 'b' is not the start of the string.

Example: whitespace and boundaries



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

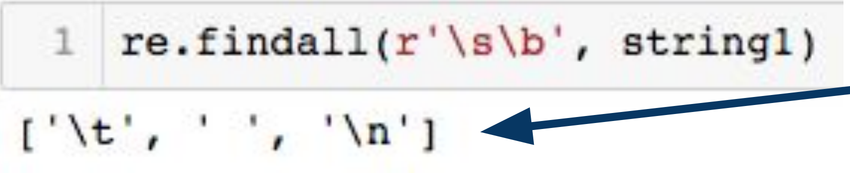
```
c       a t
s
```



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

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

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



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

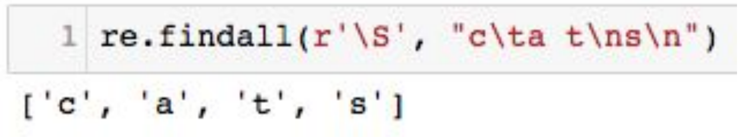
```
['\t', ' ', '\n']
```

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:

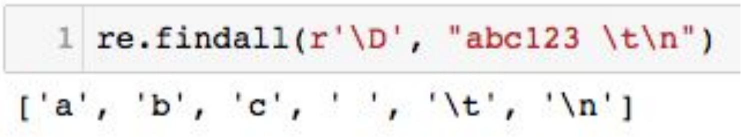
‘\S’ : matches anything that **isn’t** whitespace



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

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

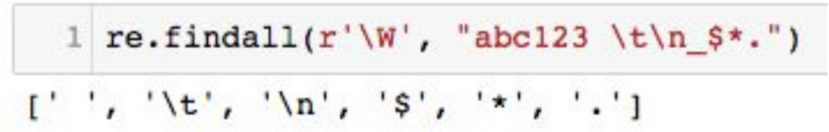
‘\D’ : matches any character that **isn’t** a digit



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

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

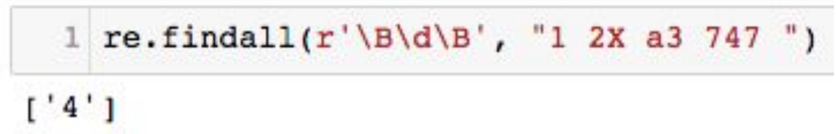
‘\W’ : matches any **non-word** character



```
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 ")
```

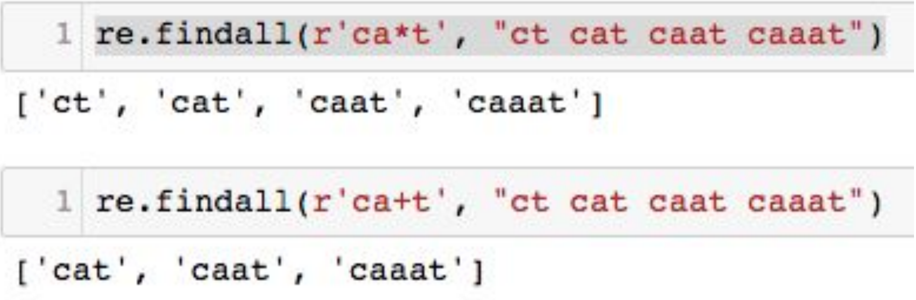
```
['4']
```

Matching and repetition

`‘*’` : zero or more of the previous item

`‘+’` : one or more of the previous item

`‘?’` : zero or one of the previous item

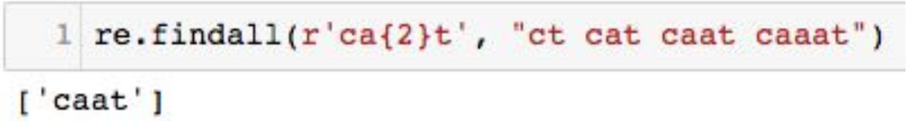


```
re.findall(r'ca*t', "ct cat caat caaat")
```

```
['ct', 'cat', 'caat', 'caaat']
```

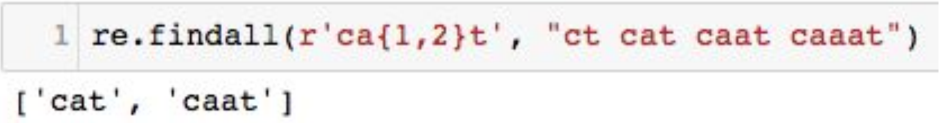
```
re.findall(r'ca+t', "ct cat caat caaat")
```

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



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

```
['caat']
```



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

```
['cat', '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\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’` ★

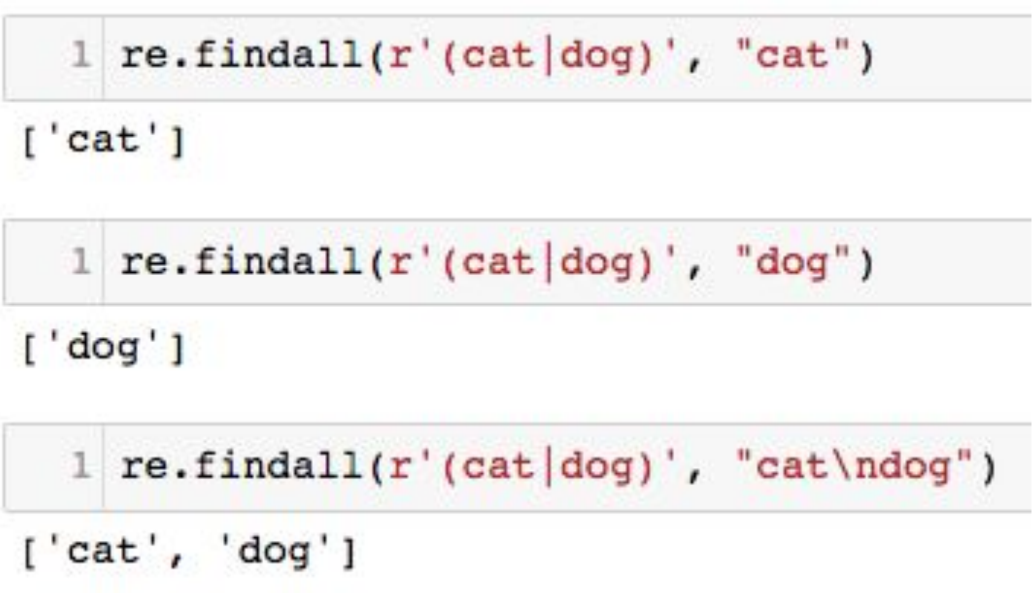
`‘Boeing 747’` ✗

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)'`



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

```
['cat']
```

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

```
['dog']
```

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

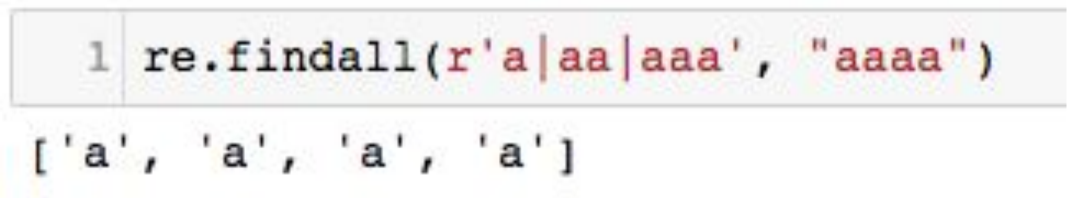
```
['cat', 'dog']
```

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']
```

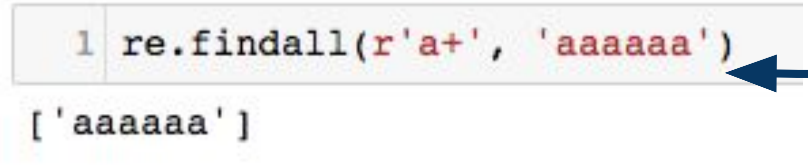
Matching with `‘|’` is ***lazy***
Tries to match each regex separated by `‘|’`, in order, left to right.
As soon as it matches something, it returns that match…

...and starts trying to make another match.

Note: this behavior can be changed using flags. Refer to documentation.

Matching and greediness

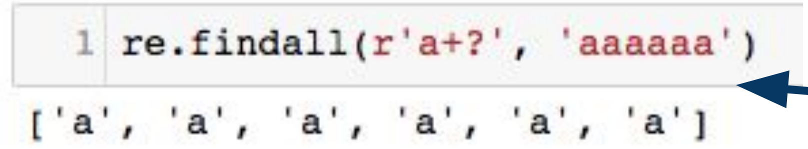
Pipe operator `'|'` is lazy. But, confusingly, python `re` module is usually **greedy:**



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

```
['aaaaaa']
```

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



```
re.findall(r'a+?', 'aaaaaa')
```

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

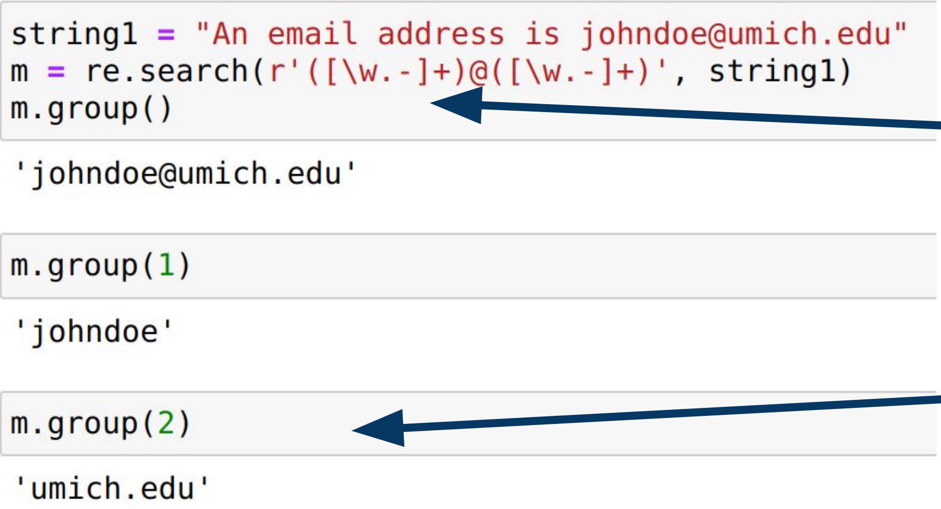
`'?'` modifies operators like `'+'` and `'*'` to **not** be greedy, and we get lazy matching, like when using `'|'`.

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



```
string1 = "An email address is johndoe@umich.edu"
m = re.search(r'([\w.-]+)@([\w.-]+)', string1)
m.group()
```

```
'johndoe@umich.edu'
```

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

```
'johndoe'
```

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

```
'umich.edu'
```

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

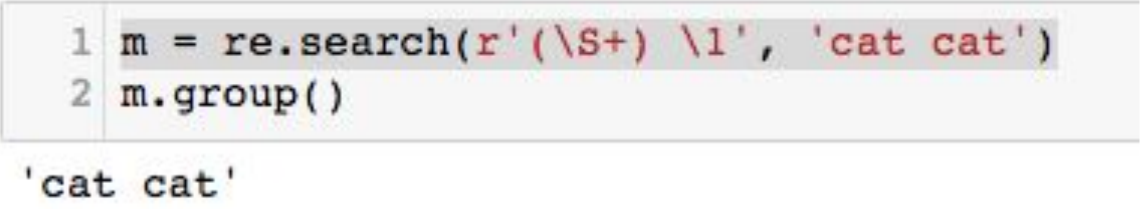
Can access groups (parts of the regex in parentheses) in numerical order. Each set of parentheses gets a group, in order from left to right.

Note: `re.findall` has similar functionality!

Backreferences

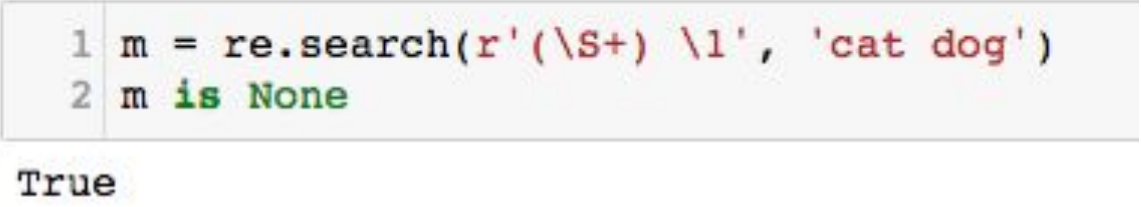
Can refer to an earlier match ***within the same regex!***
‘\N’, where N is a number, references the N-th group

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



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

```
'cat cat'
```



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

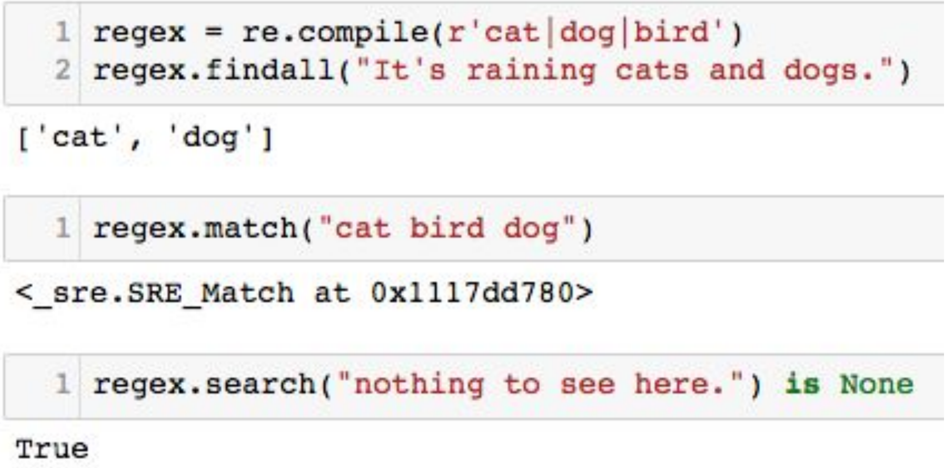
`re.DOTALL` : `‘.’` matches any character, **including** newline.

See https://docs.python.org/3/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



```
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
```

Practice with regular expressions

On Canvas, see **`Files/in-class practice/week8practice.ipynb`**

Intermission

Lots of interesting data resides on websites

HTML : **H**yper**T**ext **M**arkup **L**anguage

Specifies basically everything you see on the Internet

XML : E**X**tensible **M**arkup **L**anguage

Designed to be an easier way for storing data, similar framework to HTML

JSON : **J**ava**S**cript **O**bject **N**otation

Designed to be a saner version of XML

SQL : **S**tructured **Q**uery **L**anguage

IBM-designed language for interacting with databases

APIs : **A**pplication **P**rogramming **I**nterface

Allow interaction with website functionality (e.g., Google maps)

Three Aspects of Data on the Web

Location: URL (Uniform Resource Locator), IP address

Specifies location of a computer on a network

Protocol: HTTP, HTTPS, FTP, SMTP

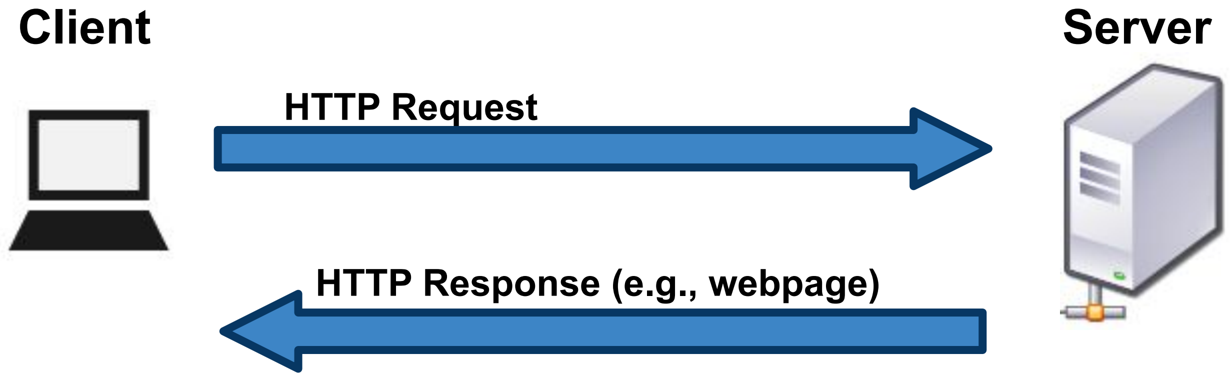
Specifies how computers on a network should communicate with one another

Content: HTML, JSON, XML (for example)

Contains actual information, e.g., tells browser what to display and how

We'll mostly be concerned with website content. Wikipedia has good entries on network protocols. The classic textbook is *Computer Networks* by A. S. Tanenbaum.

Client-server model



Client

Server

HTTP Request

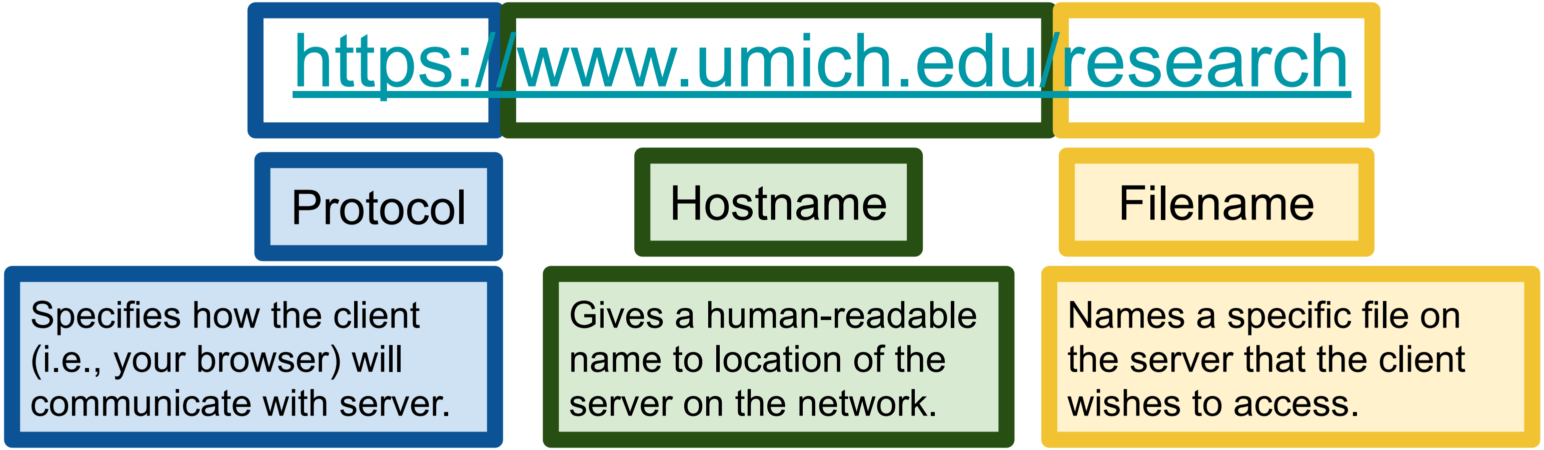
HTTP Response (e.g., webpage)

Client asks the server for information, server returns information.

HTTP is

- **Connectionless:** after a request is made, the client disconnects and waits
- **Media agnostic:** any kind of data can be sent over HTTP
- **Stateless:** server and client “forget about each other” after a request

Anatomy of a URL



https://www.umich.edu/research

Protocol

Specifies how the client (i.e., your browser) will communicate with server.

Hostname

Gives a human-readable name to location of the server on the network.

Filename

Names a specific file on the server that the client wishes to access.

Note: often the extension of the file will indicate what type it is (e.g., html, txt, pdf, etc), but not always. Often, one must determine the type of the file based on its contents. This can almost always be done automatically.

Accessing websites in Python: `urllib`

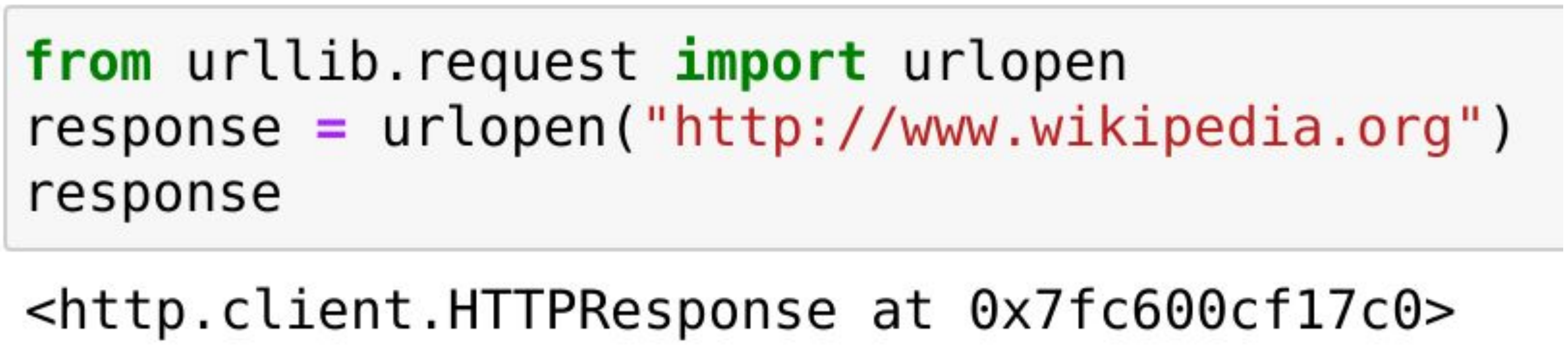
Python library for opening URLs and interacting with websites
https://docs.python.org/3/howto/urllib2.html

Software development community is moving towards **requests**
https://requests.readthedocs.io/en/master/
a bit over-powered for what we want to do, but feel free to use it in HWs

Note: Python 3 split what was previously `urllib2` in Python 2 into several related submodules of `urllib`. You should be aware of this in case you end up having to migrate code from Python 2 to Python 3 or vice-versa.

Using urllib

`urllib.request.urlopen()` : opens the given url, returns a file-like object



```
from urllib.request import urlopen
response = urlopen("http://www.wikipedia.org")
response
```

```
<http.client.HTTPResponse at 0x7fc600cf17c0>
```

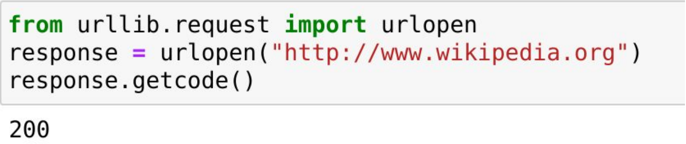
Three basic methods
`getcode()` : return the HTTP status code of the response
`geturl()` : return URL of the resource retrieved (e.g., see if redirected)
`info()` : return meta-information from the page, such as headers

getcode()

HTTP includes success/error status codes

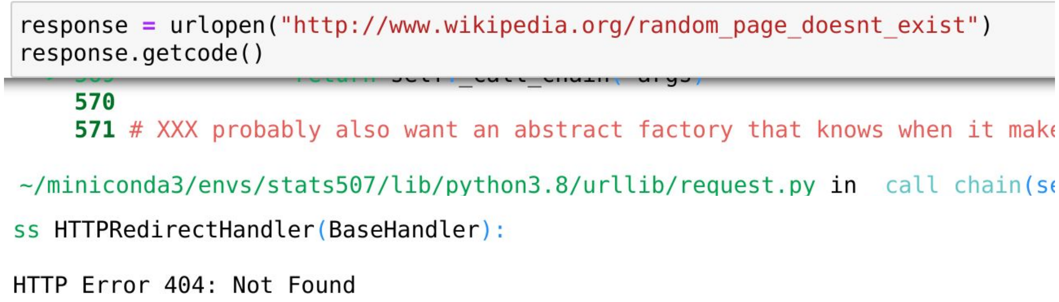
Ex: 200 OK, 301 Moved Permanently, 404 Not Found, 503 Service Unavailable

See https://en.wikipedia.org/wiki/List_of_HTTP_status_codes



```
from urllib.request import urlopen
response = urlopen("http://www.wikipedia.org")
response.getcode()
```

```
200
```

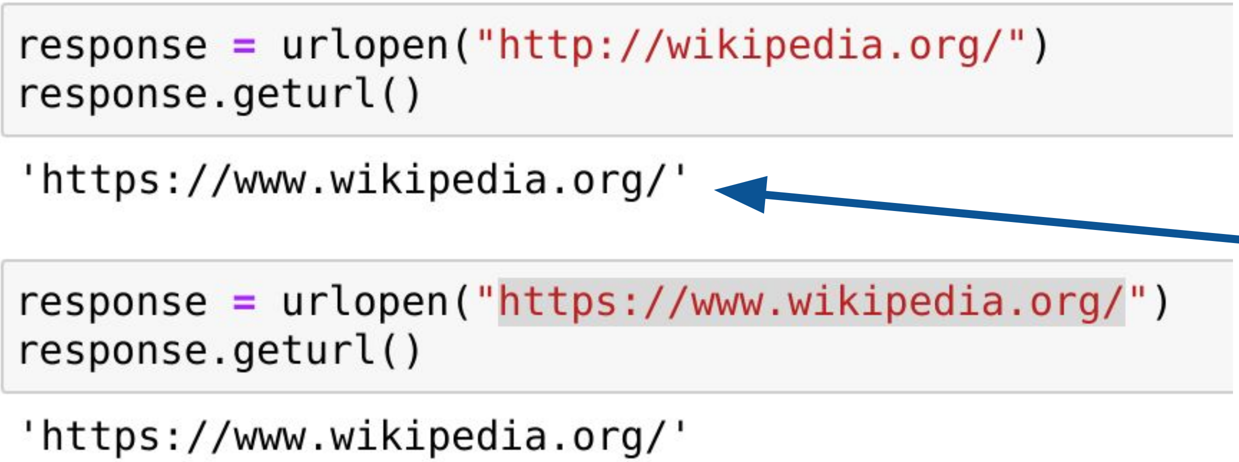


```
response = urlopen("http://www.wikipedia.org/random_page_doesnt_exist")
response.getcode()
```

```
    570
    571 # XXX probably also want an abstract factory that knows when it make

~/miniconda3/envs/stats507/lib/python3.8/urllib/request.py in  call chain(se

ss HTTPRedirectHandler(BaseHandler):

HTTP Error 404: Not Found
```

Note: I cropped a some of the output here

geturl()



```
response = urlopen("http://wikipedia.org/")
response.geturl()
```

```
'https://www.wikipedia.org/'
```

```
response = urlopen("https://www.wikipedia.org/")
response.geturl()
```

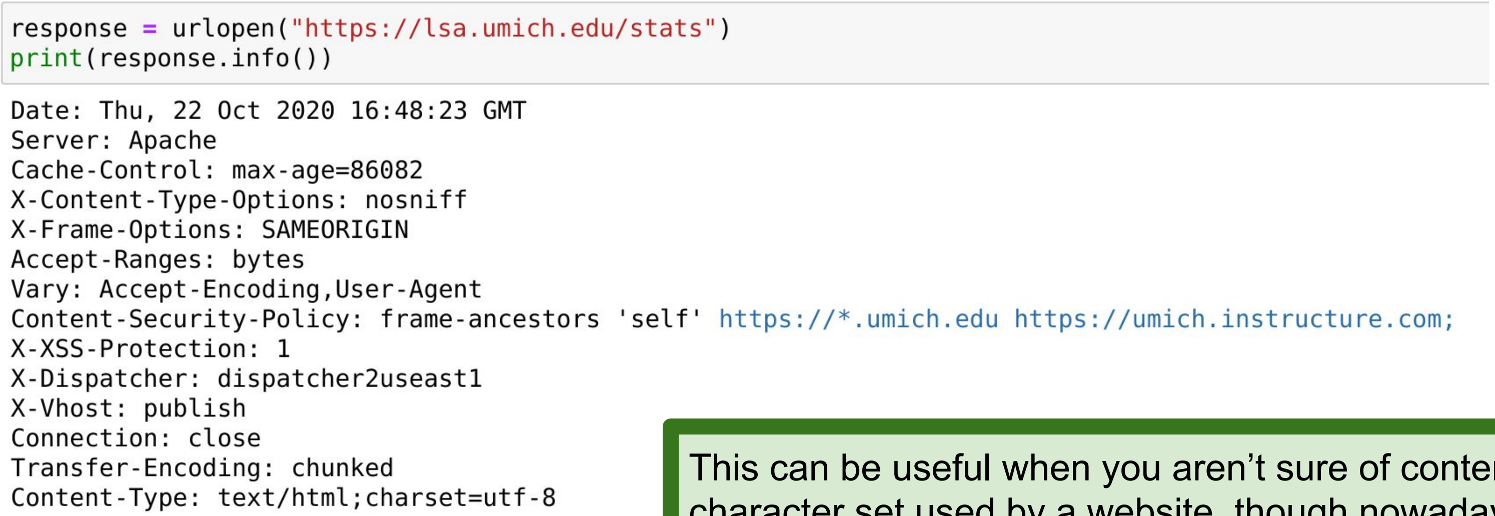
```
'https://www.wikipedia.org/'
```

Different URLs, owing to automatic redirect.

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

info()

Returns a dictionary-like object with information about the page you retrieved.



```python
response = urlopen("https://lsa.umich.edu/stats")
print(response.info())
```

```
Date: Thu, 22 Oct 2020 16:48:23 GMT
Server: Apache
Cache-Control: max-age=86082
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Accept-Ranges: bytes
Vary: Accept-Encoding,User-Agent
Content-Security-Policy: frame-ancestors 'self' https://*.umich.edu https://umich.instructure.com;
X-XSS-Protection: 1
X-Dispatcher: dispatcher2useast1
X-Vhost: publish
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html;charset=utf-8
```

This can be useful when you aren’t sure of content type or character set used by a website, though nowadays most of those things are handled automatically by parsers.

HTML Crash Course

HTML is a markup language.

```
<tag_name attr1=”value” attr2=”differentValue”>String contents</tag_name>
```

Basic unit: **tag**

(usually) a start and end tag, like `<p>contents</p>`

Contents of a tag may contain more tags:

```
<head><title>The Title</title></head>
<p>This tag links to <a href=”google.com”>Google</a></p>
```

HTML Crash Course

```
<tag_name attr1=”value” attr2=”differentValue”>String contents</tag_name>
```

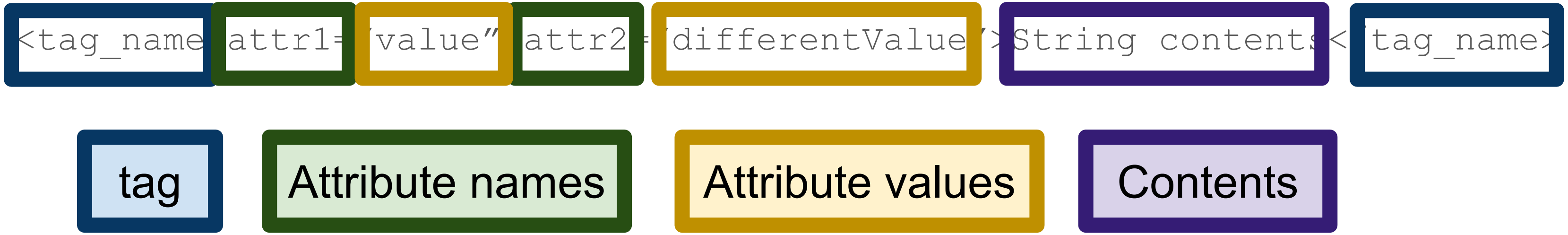
Tags have attributes, which are specified after the tag name, in (key,value) pairs of the form `key=”val”`

Example: hyperlink tags

```
    <a href=”umich.edu/~johndoe”>personal webpage</a>
```

The `href` attribute specifies where the hyperlink should point.

HTML Crash Course: Recap



```
<tag_name attr1="value" attr2="differentValue">String contents</tag_name>
```

tag

Attribute names

Attribute values

Contents

Of special interest in your homework: HTML tables

https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table

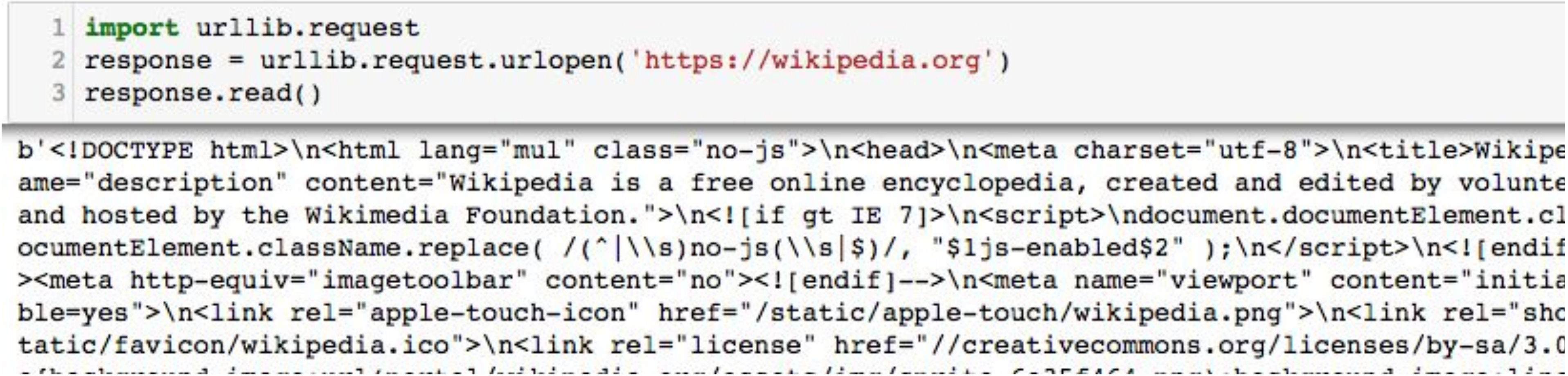
https://www.w3schools.com/html/html_tables.asp

https://www.w3.org/TR/html401/struct/tables.html

Okay, back to `urllib`

`urllib` reads a webpage (full of HTML) and returns a “response” object

The response object can be treated like a file:



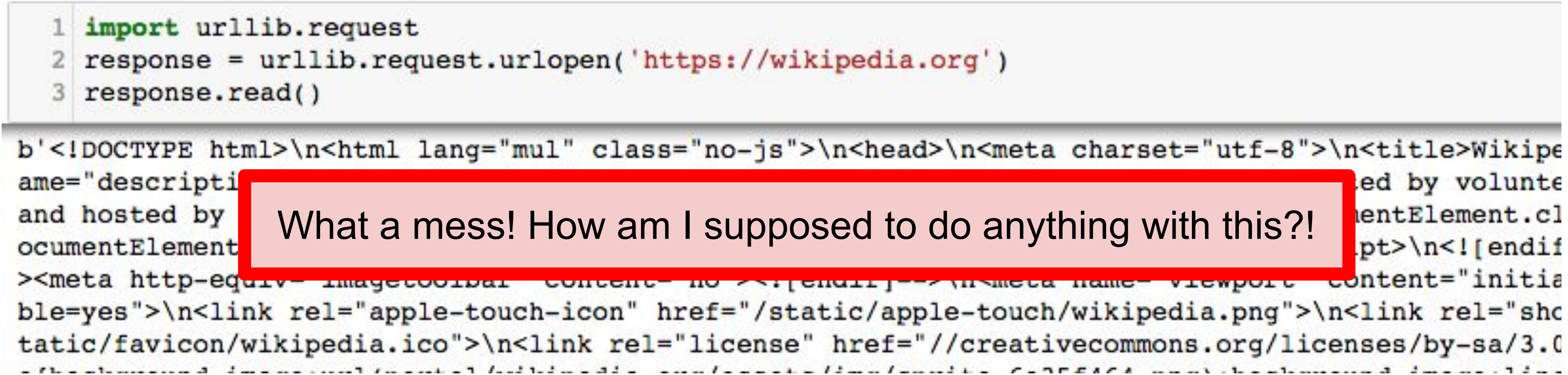
```
import urllib.request
response = urllib.request.urlopen('https://wikipedia.org')
response.read()
```

```
b'<!DOCTYPE html>\n<html lang="mul" class="no-js">\n<head>\n<meta charset="utf-8">\n<title>Wikipe
ame="description" content="Wikipedia is a free online encyclopedia, created and edited by volunte
and hosted by the Wikimedia Foundation.">\n<![if gt IE 7]>\n<script>\ndocument.documentElement.cl
ocumentElement.className.replace( /(^|\\s)no-js(\\s|$)/, "$1js-enabled$2" );\n</script>\n<![endif
><meta http-equiv="imagetoolbar" content="no"><![endif]-->\n<meta name="viewport" content="initia
ble=yes">\n<link rel="apple-touch-icon" href="/static/apple-touch/wikipedia.png">\n<link rel="sho
tatic/favicon/wikipedia.ico">\n<link rel="license" href="//creativecommons.org/licenses/by-sa/3.0
```

Okay, back to `urllib`

`urllib` reads a webpage (full of HTML) and returns a “response” object

The response object can be treated like a file:



```python
import urllib.request
response = urllib.request.urlopen('https://wikipedia.org')
response.read()
```

```
b'<!DOCTYPE html>\n<html lang="mul" class="no-js">\n<head>\n<meta charset="utf-8">\n<title>Wikipe
ame="descripti                                                                    ed by volunte
and hosted by                                                                     entElement.cl
ocumentElement                                                                    pt>\n<![endif
><meta http-equiv="imagetoolbar" content="no"><![endif]-->\n<meta name="viewport" content="initia
ble=yes">\n<link rel="apple-touch-icon" href="/static/apple-touch/wikipedia.png">\n<link rel="sho
tatic/favicon/wikipedia.ico">\n<link rel="license" href="//creativecommons.org/licenses/by-sa/3.0
```

What a mess! How am I supposed to do anything with this?!

Parsing HTML/XML in Python: beautifulsoup

Python library for working with HTML/XML data
Builds nice tree representation of markup data…
...and provides tools for working with that tree

Documentation: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Good tutorial:
http://www.pythonforbeginners.com/python-on-the-web/beautifulsoup-4-python/

Installation: `pip install beautifulsoup` or follow instructions for conda or...

Parsing HTML/XML in Python: beautifulsoup

BeautifulSoup turns HTML mess into a (sometimes complex) tree

Four basic kinds of objects:

Tag: corresponds to HTML tags

`<[name] [attr]=”xyz”>[string]</[name]>`)

Two important attributes: tag.name, tag.string

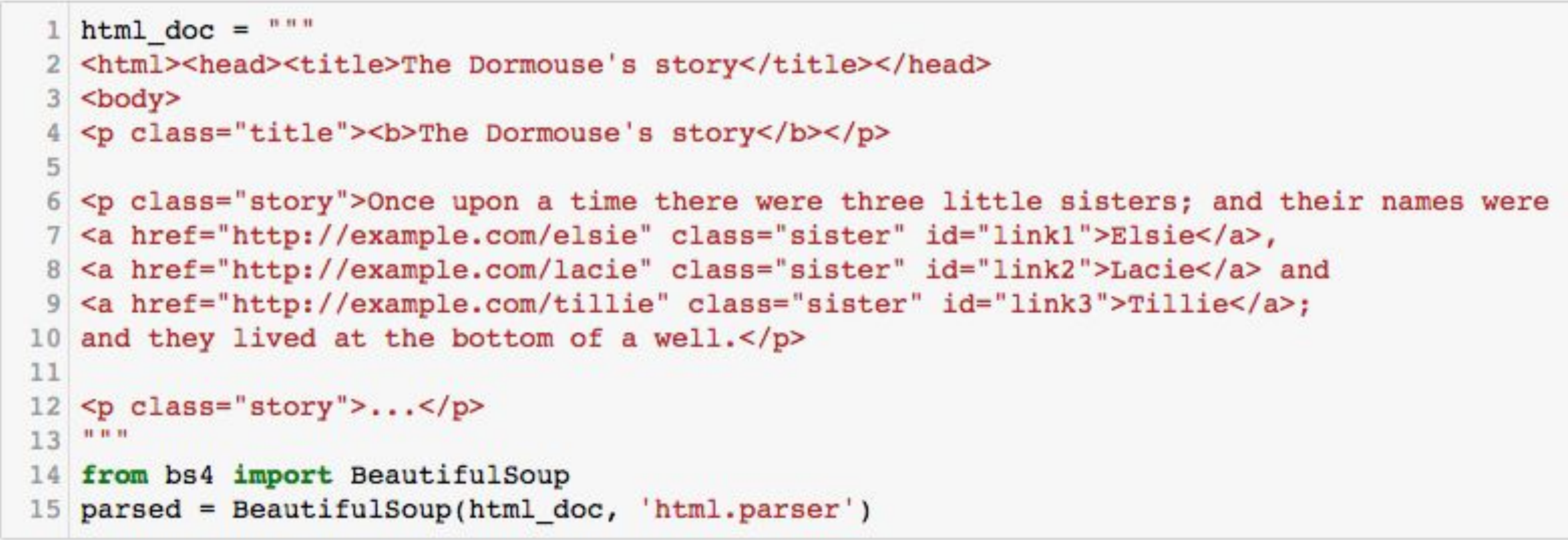
Also has dictionary-like structure for accessing attributes

NavigableString: special kind of string for use in `bs4`

BeautifulSoup: represents the HTML document itself

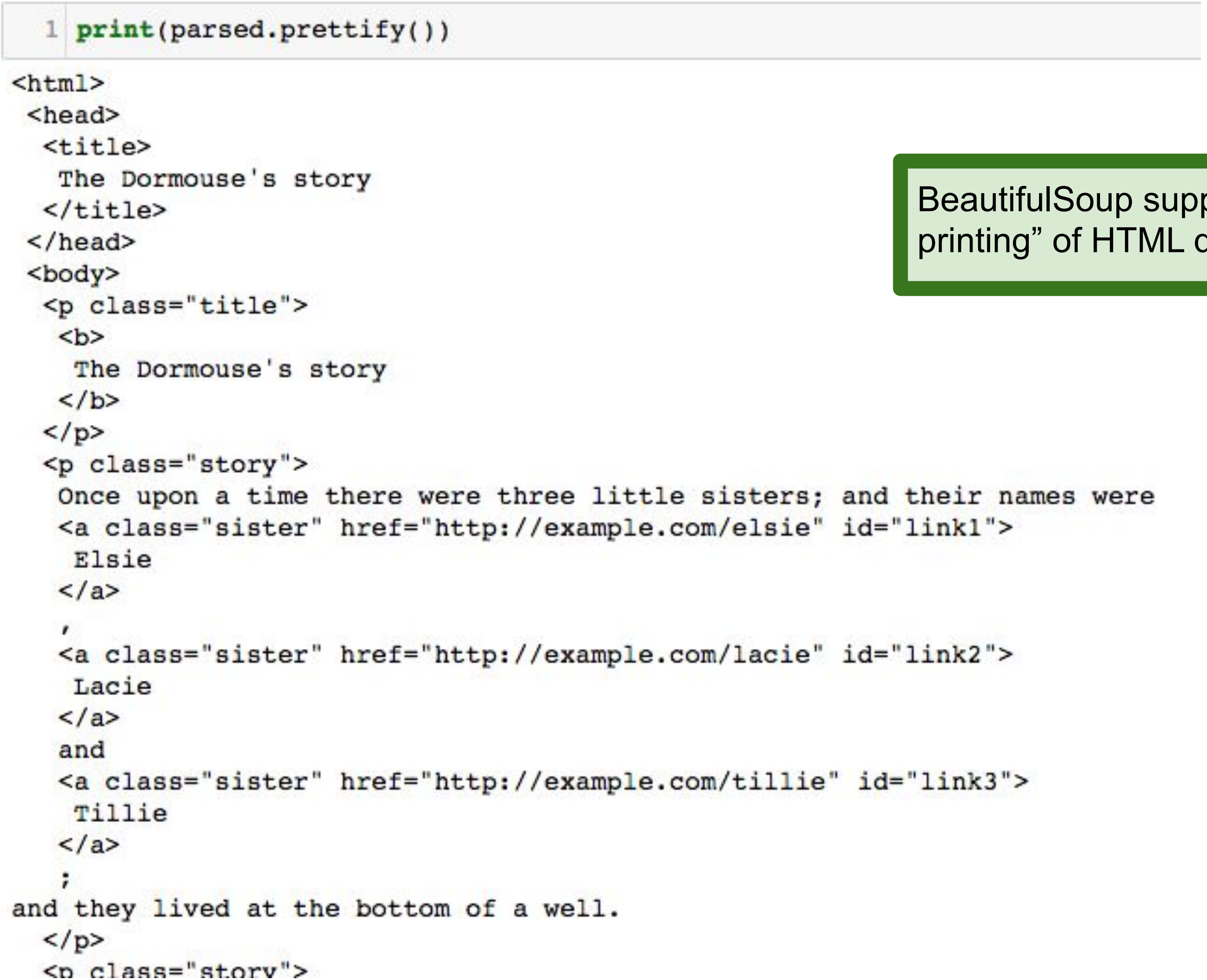
Comment: special kind of NavigableString for HTML comments

Example (from the BeautifulSoup docs)



```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""
from bs4 import BeautifulSoup
parsed = BeautifulSoup(html_doc, 'html.parser')
```

Follow along at home: https://www.crummy.com/software/BeautifulSoup/bs4/doc/#quick-start

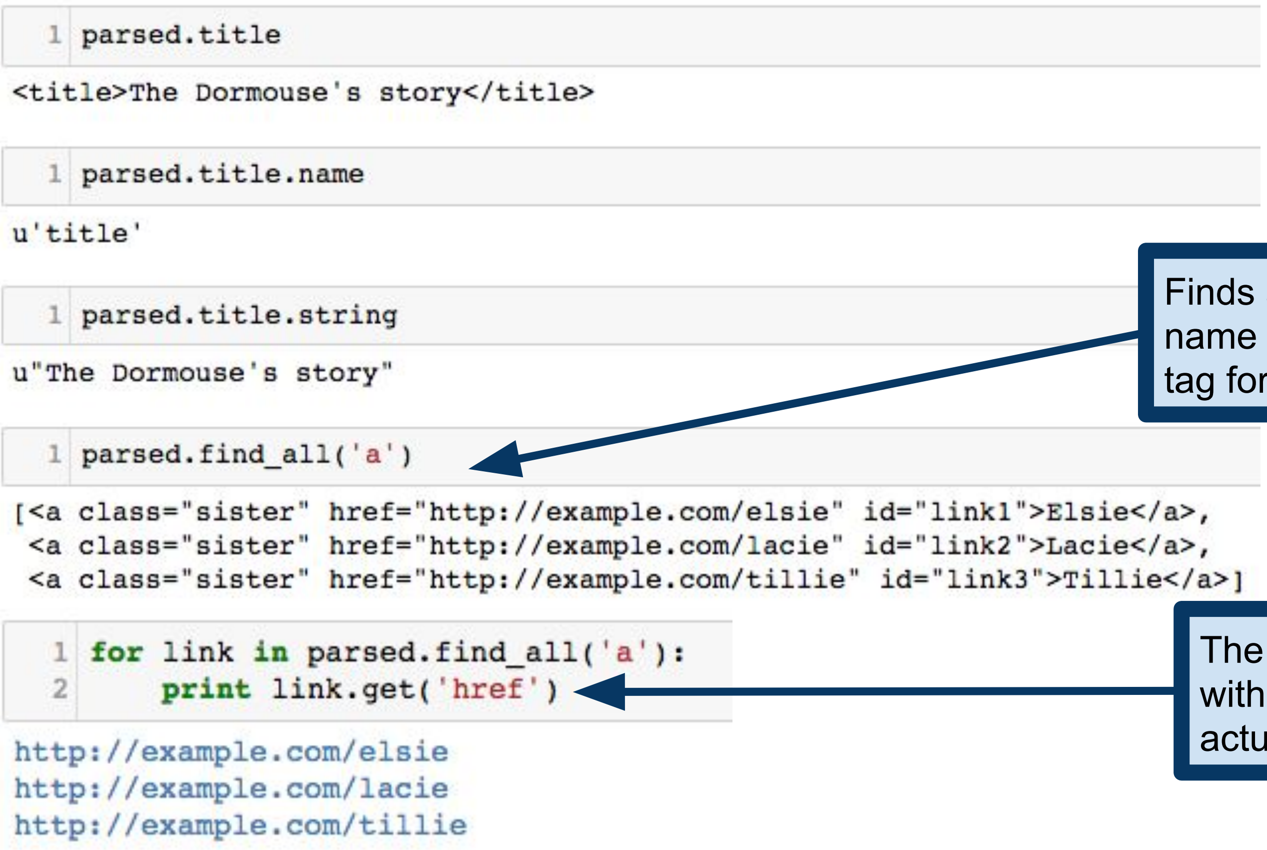


```python
print(parsed.prettify())
```

```
<html>
 <head>
  <title>
   The Dormouse's story
  </title>
 </head>
 <body>
  <p class="title">
   <b>
    The Dormouse's story
   </b>
  </p>
  <p class="story">
   Once upon a time there were three little sisters; and their names were
   <a class="sister" href="http://example.com/elsie" id="link1">
    Elsie
   </a>
   ,
   <a class="sister" href="http://example.com/lacie" id="link2">
    Lacie
   </a>
   and
   <a class="sister" href="http://example.com/tillie" id="link3">
    Tillie
   </a>
   ;
and they lived at the bottom of a well.
  </p>
  <p class="story">
```

BeautifulSoup supports “pretty printing” of HTML documents.

BeautifulSoup allows navigation of the HTML tags



```
parsed.title
```

```
<title>The Dormouse's story</title>
```

```
parsed.title.name
```

```
u'title'
```

```
parsed.title.string
```

```
u"The Dormouse's story"
```

```
parsed.find_all('a')
```

```
[<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

```
for link in parsed.find_all('a'):
    print link.get('href')
```

```
http://example.com/elsie
http://example.com/lacie
http://example.com/tillie
```

Finds all the tags that have the name ‘a’, which is the HTML tag for a link.

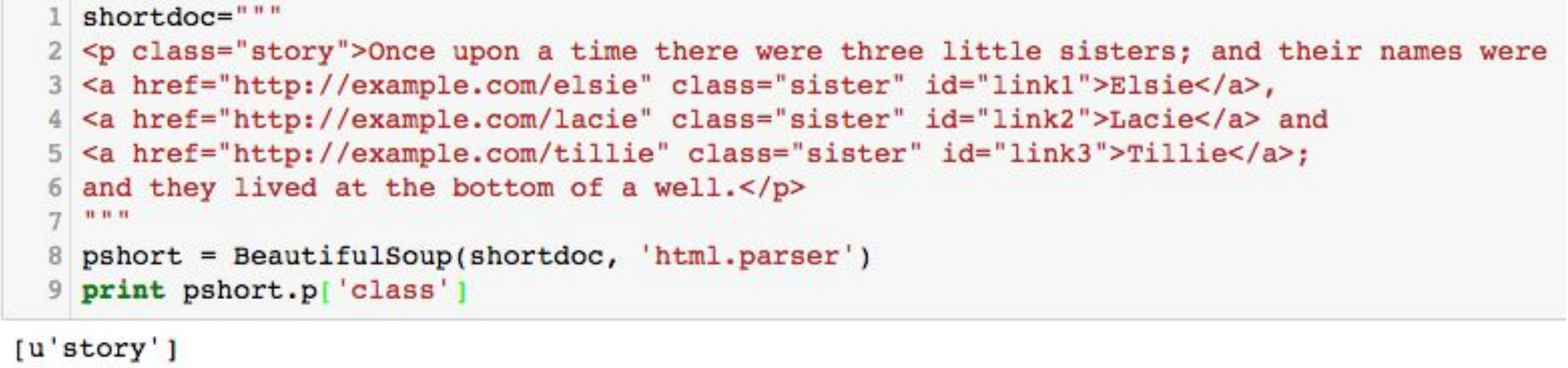
The ‘href’ attribute in a tag with name ‘a’ contains the actual url for use in the link.

A note on attributes

HTML attributes and Python attributes are **different things!**

But in `BeautifulSoup` they collide in a weird way

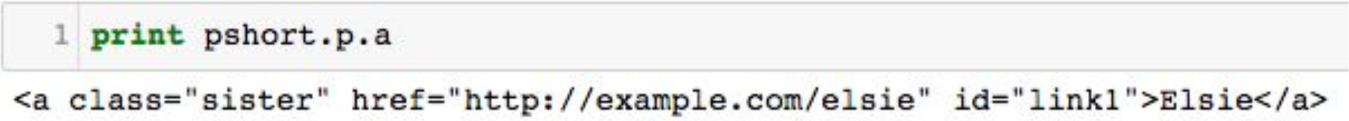
`BeautifulSoup` tags have their HTML attributes accessible like a dictionary:



```
shortdoc="""
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
"""
pshort = BeautifulSoup(shortdoc, 'html.parser')
print pshort.p['class']
```

```
[u'story']
```

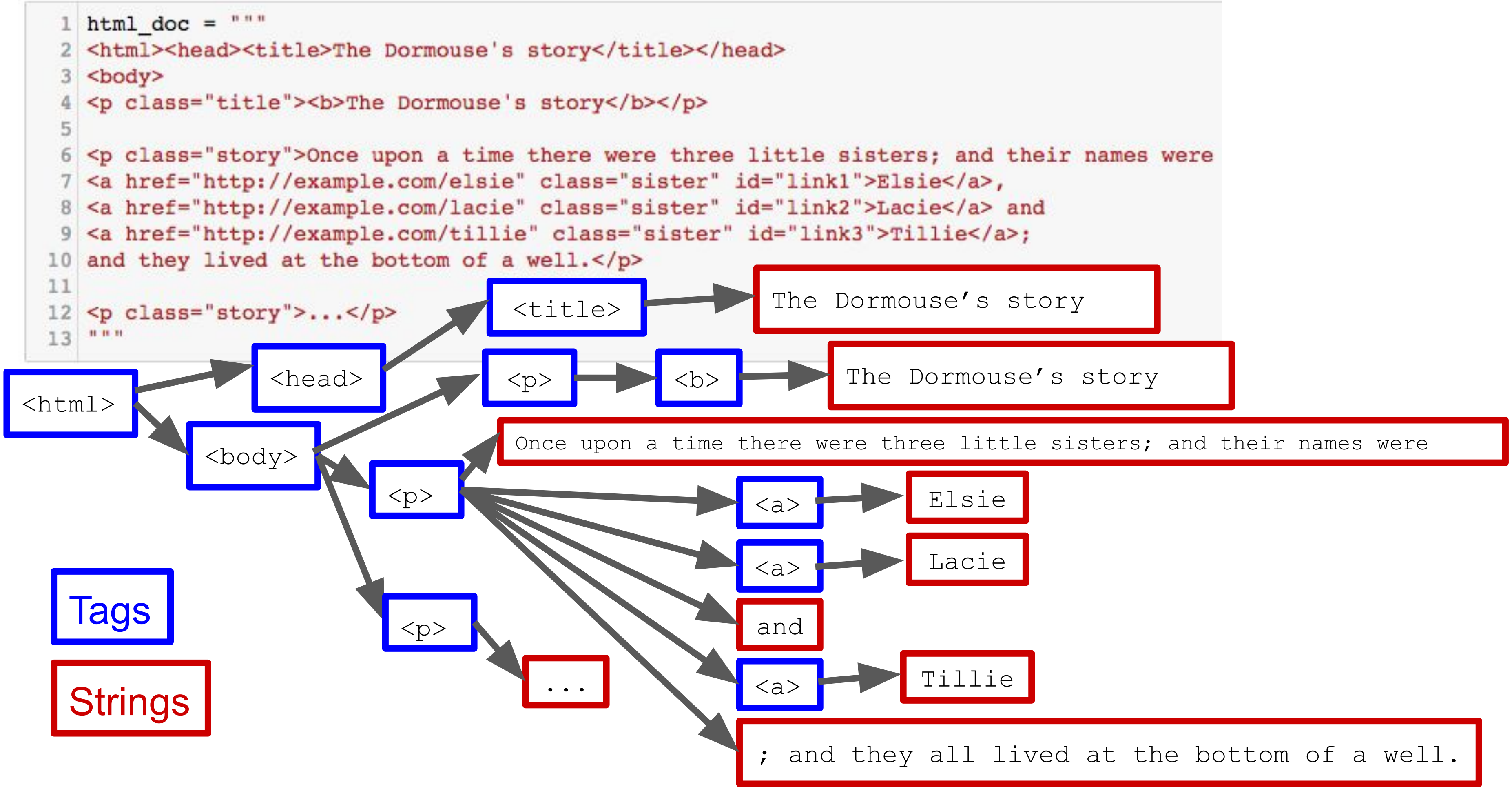
`BeautifulSoup` tags have their *children* accessible as Python attributes:



```
print pshort.p.a
```

```
<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

HTML tree structure



```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""
```

<html>

<head>

<title>

The Dormouse’s story

<body>

<p>

<b>

The Dormouse’s story

<p>

Once upon a time there were three little sisters; and their names were

<a>

Elsie

<a>

Lacie

and

<a>

Tillie

; and they all lived at the bottom of a well.

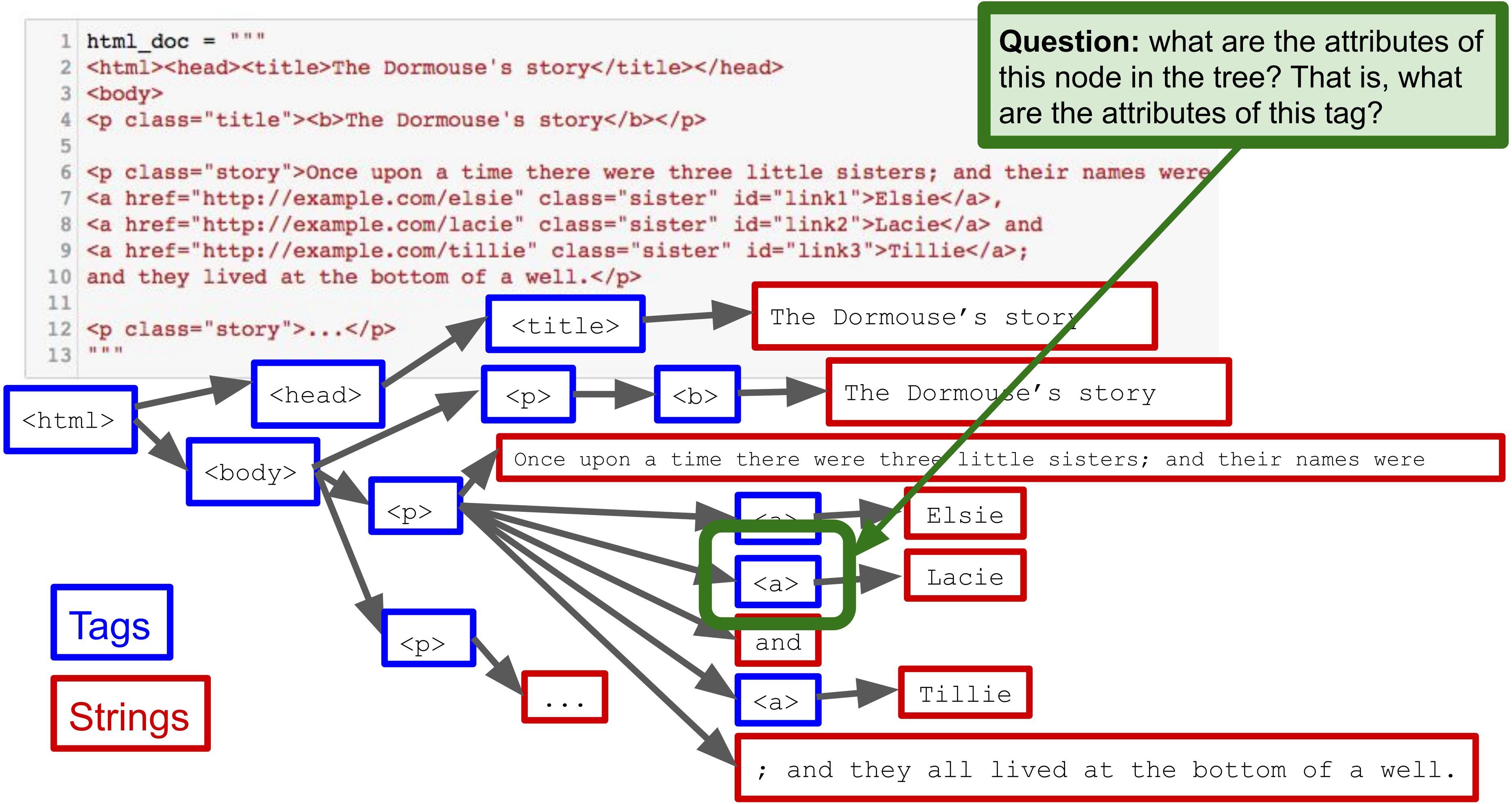
<p>

...

Tags

Strings

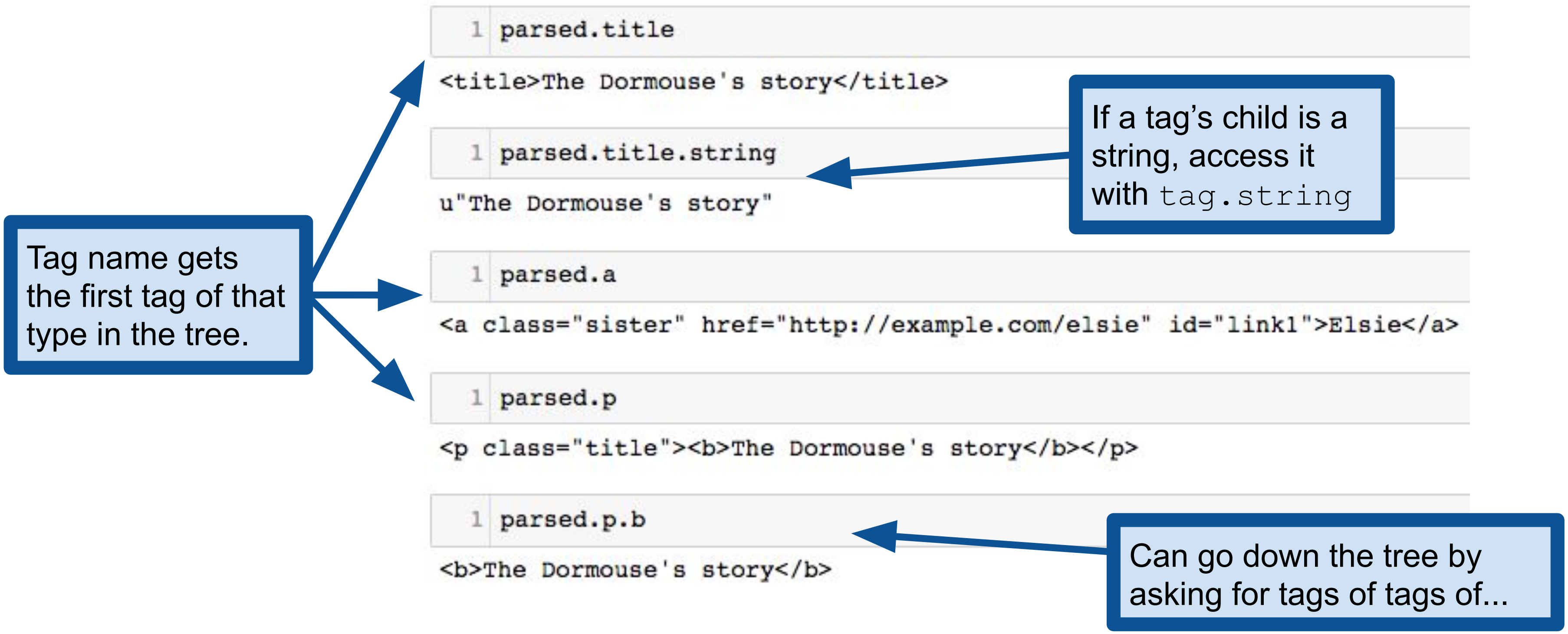
HTML tree structure



```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""
```

Question: what are the attributes of this node in the tree? That is, what are the attributes of this tag?

Navigating the HTML tree



```
parsed.title
```

```
<title>The Dormouse's story</title>
```

```
parsed.title.string
```

```
u"The Dormouse's story"
```

```
parsed.a
```

```
<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

```
parsed.p
```

```
<p class="title"><b>The Dormouse's story</b></p>
```

```
parsed.p.b
```

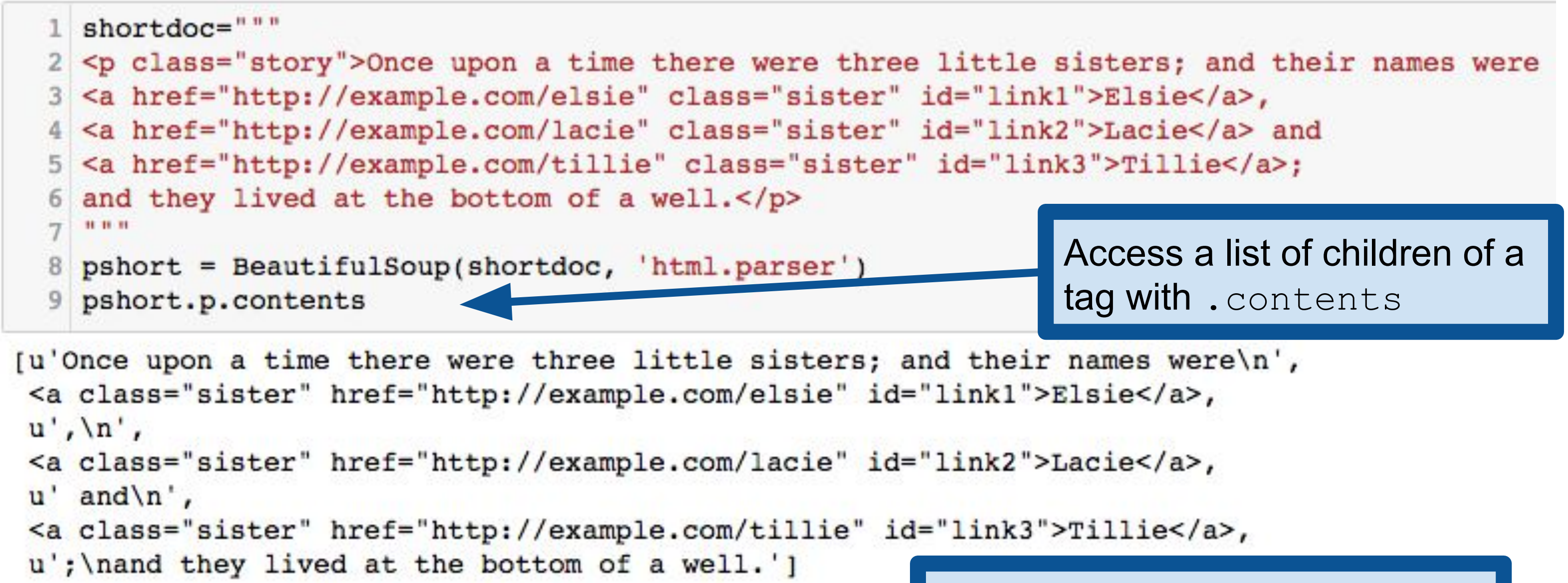
```
<b>The Dormouse's story</b>
```

Tag name gets the first tag of that type in the tree.

If a tag’s child is a string, access it with `tag.string`

Can go down the tree by asking for tags of tags of...

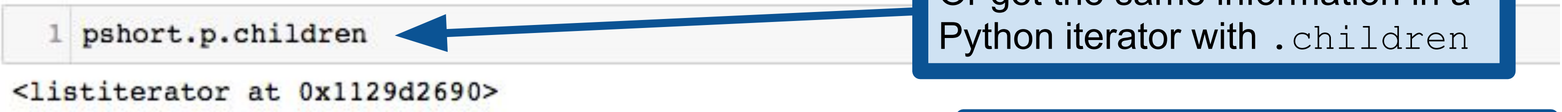
Navigating the HTML tree



```
shortdoc="""
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
"""
pshort = BeautifulSoup(shortdoc, 'html.parser')
pshort.p.contents
```

```
[u'Once upon a time there were three little sisters; and their names were\n',
 <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
 u',\n',
 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
 u' and\n',
 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>,
 u';\nand they lived at the bottom of a well.']
```

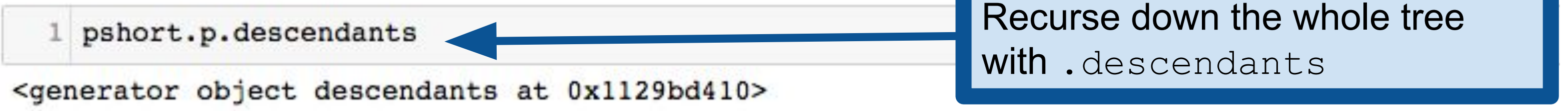
Access a list of children of a tag with `.contents`



```
pshort.p.children
```

```
<listiterator at 0x1129d2690>
```

Or get the same information in a Python iterator with `.children`



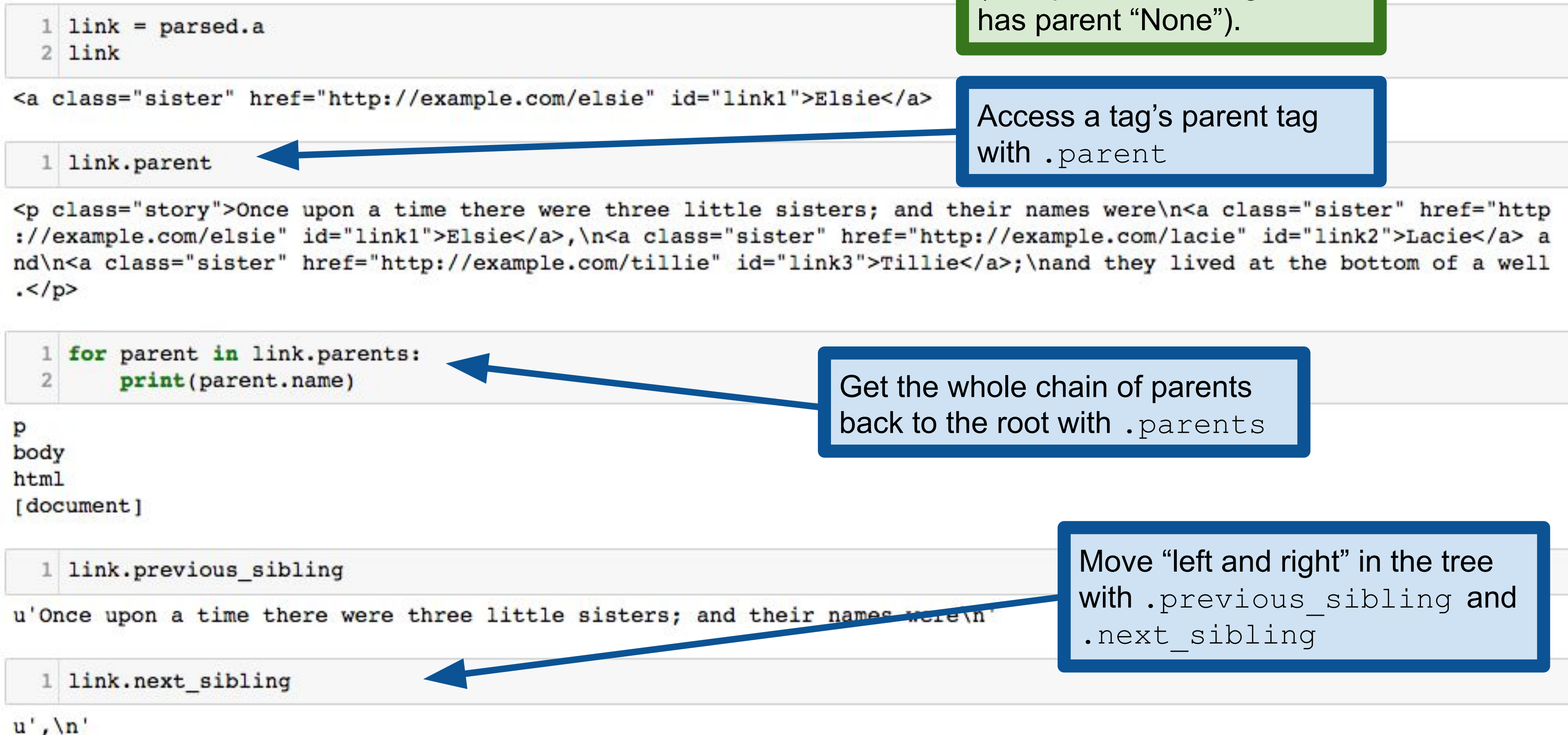
```
pshort.p.descendants
```

```
<generator object descendants at 0x1129bd410>
```

Recurse down the whole tree with `.descendants`

Navigating the HTML tree

The tree structure means that every tag has a parent (except the “root” tag, which has parent “None”).



```
link = parsed.a
link
```

```
<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

```
link.parent
```

```
<p class="story">Once upon a time there were three little sisters; and their names were\n<a class="sister" href="http
://example.com/elsie" id="link1">Elsie</a>,\n<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> a
nd\n<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;\nand they lived at the bottom of a well
.</p>
```

```
for parent in link.parents:
    print(parent.name)
```

```
p
body
html
[document]
```

```
link.previous_sibling
```

```
u'Once upon a time there were three little sisters; and their names were\n'
```

```
link.next_sibling
```

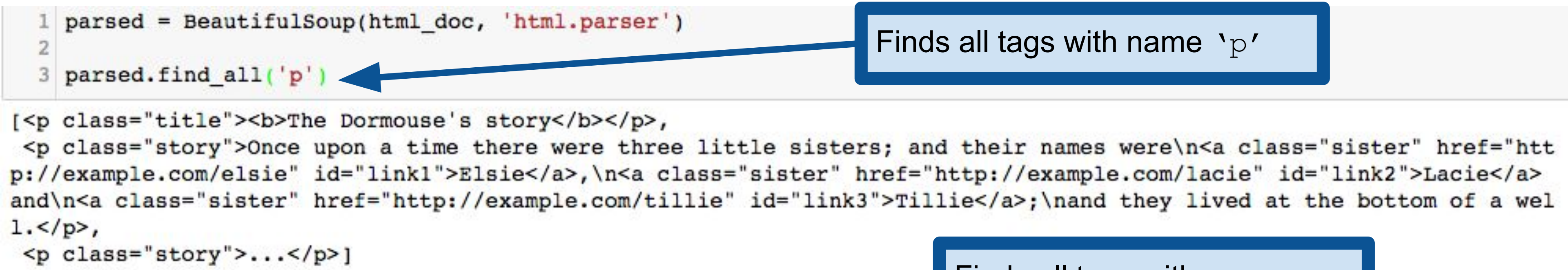
```
u',\n'
```

Access a tag’s parent tag with `.parent`

Get the whole chain of parents back to the root with `.parents`

Move “left and right” in the tree with `.previous_sibling` and `.next_sibling`

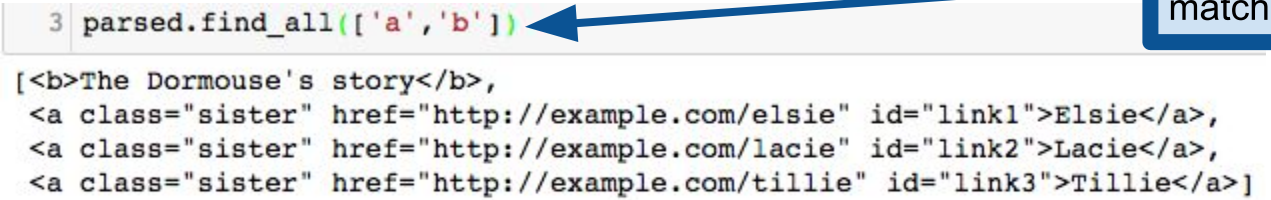
Searching the tree: `find_all` and related methods



```
parsed = BeautifulSoup(html_doc, 'html.parser')

parsed.find_all('p')
```

```
[<p class="title"><b>The Dormouse's story</b></p>,
 <p class="story">Once upon a time there were three little sisters; and their names were\n<a class="sister" href="htt
p://example.com/elsie" id="link1">Elsie</a>,\n<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
and\n<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;\nand they lived at the bottom of a wel
l.</p>,
 <p class="story">...</p>]
```

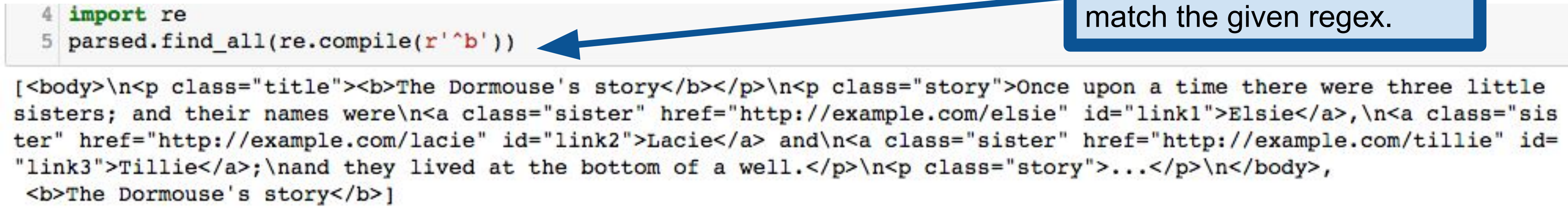
Finds all tags with name `'p'`



```
parsed.find_all(['a','b'])
```

```
[<b>The Dormouse's story</b>,
 <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

Finds all tags with names matching **either** `'a'` or `'b'`

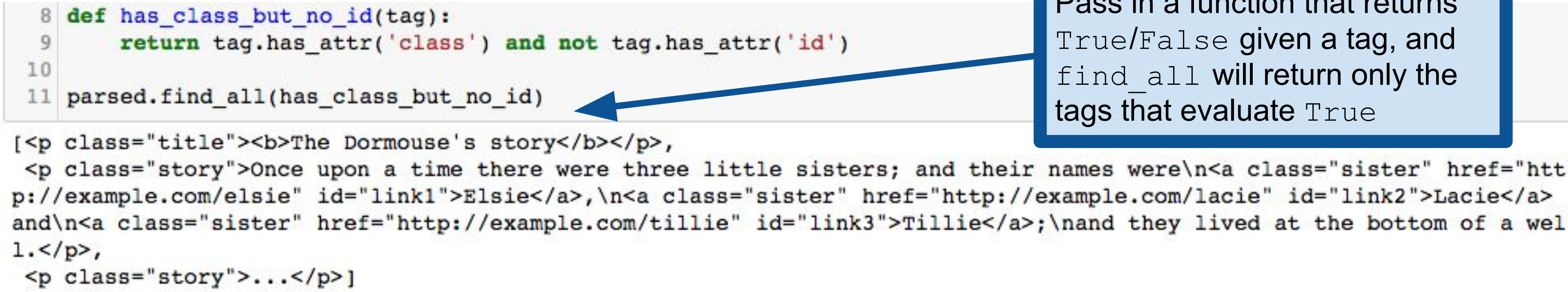


```
import re
parsed.find_all(re.compile(r'^b'))
```

```
[<body>\n<p class="title"><b>The Dormouse's story</b></p>\n<p class="story">Once upon a time there were three little
sisters; and their names were\n<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,\n<a class="sis
ter" href="http://example.com/lacie" id="link2">Lacie</a> and\n<a class="sister" href="http://example.com/tillie" id=
"link3">Tillie</a>;\nand they lived at the bottom of a well.</p>\n<p class="story">...</p>\n</body>,
 <b>The Dormouse's story</b>]
```

Finds all tags whose names match the given regex.

More about `find_all`



```python
def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')

parsed.find_all(has_class_but_no_id)
```

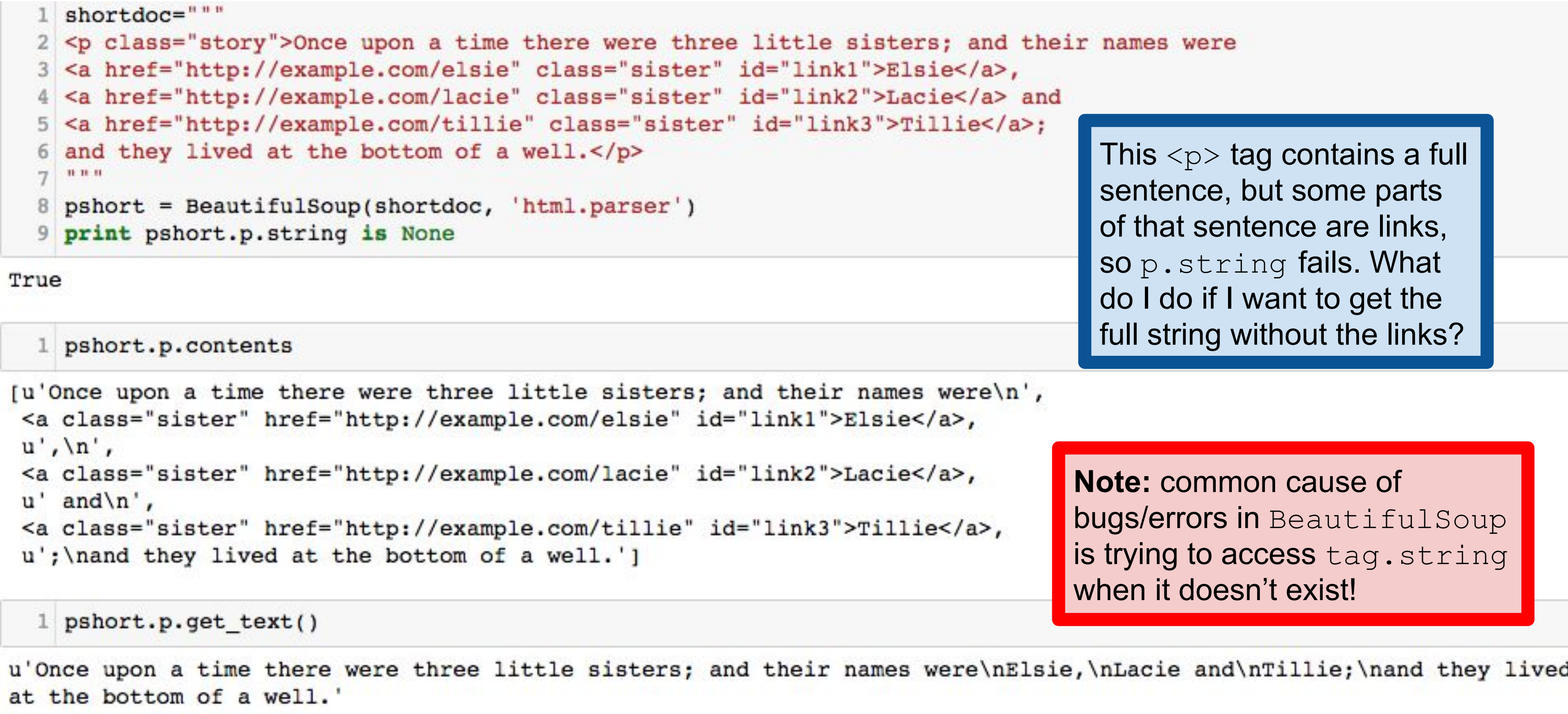
```
[<p class="title"><b>The Dormouse's story</b></p>,
 <p class="story">Once upon a time there were three little sisters; and their names were\n<a class="sister" href="htt
p://example.com/elsie" id="link1">Elsie</a>,\n<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
and\n<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;\nand they lived at the bottom of a wel
l.</p>,
 <p class="story">...</p>]
```

Pass in a function that returns `True`/`False` given a tag, and `find_all` will return only the tags that evaluate `True`

Note: by default, `find_all` recurses down the whole tree, but you can have it only search the immediate children of the tag by passing the flag `recursive=False`.

See https://www.crummy.com/software/BeautifulSoup/bs4/doc/#find-all for more.

Flattening contents: `get_text()`



```
shortdoc="""
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
"""
pshort = BeautifulSoup(shortdoc, 'html.parser')
print pshort.p.string is None
```

```
True
```

```
pshort.p.contents
```

```
[u'Once upon a time there were three little sisters; and their names were\n',
 <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
 u',\n',
 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
 u' and\n',
 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>,
 u';\nand they lived at the bottom of a well.']
```

```
pshort.p.get_text()
```

```
u'Once upon a time there were three little sisters; and their names were\nElsie,\nLacie and\nTillie;\nand they lived
at the bottom of a well.'
```

This `<p>` tag contains a full sentence, but some parts of that sentence are links, so `p.string` fails. What do I do if I want to get the full string without the links?

Note: common cause of bugs/errors in `BeautifulSoup` is trying to access `tag.string` when it doesn't exist!

XML - eXtensible Markup Language, .xml

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

Core idea: separate data from its presentation

Note that HTML *doesn't* do this-- the HTML for the webpage **is** the data

But XML is tag-based, very similar to HTML

BeautifulSoup will parse XML
https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

We won't talk much about XML, because it's falling out of favor, replaced by...

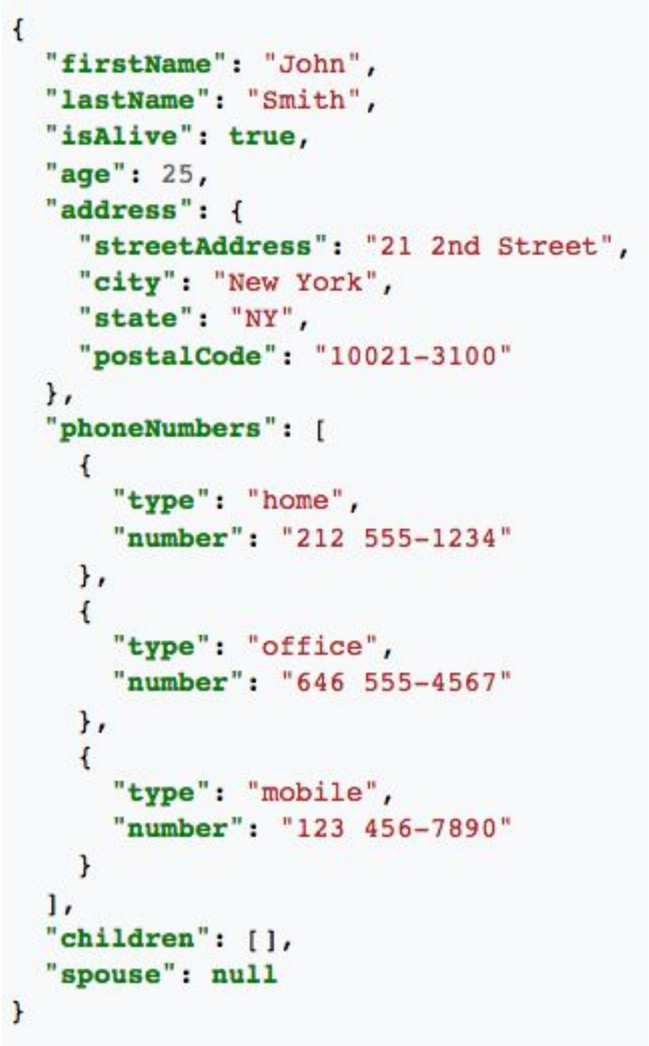
JSON - JavaScript Object Notation

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

Commonly used by website APIs

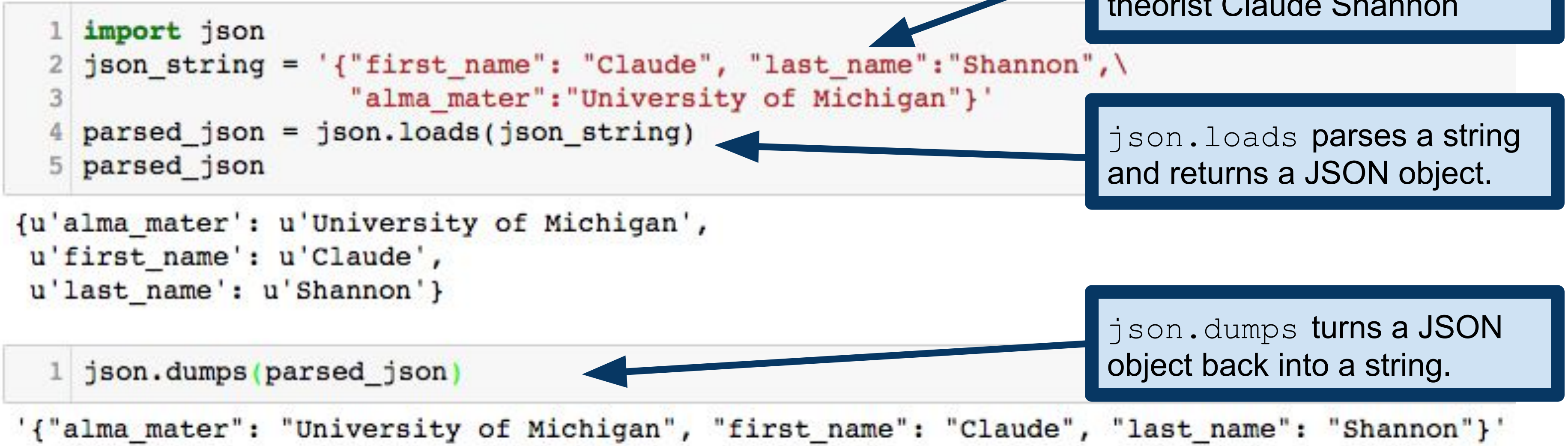
Basic building blocks:
attribute–value pairs
array data

Example (right) from wikipedia:
Possible JSON representation of a person



```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

Python `json` module



```python
import json
json_string = '{"first_name": "Claude", "last_name":"Shannon",\
                 "alma_mater":"University of Michigan"}'
parsed_json = json.loads(json_string)
parsed_json
```

```
{u'alma_mater': u'University of Michigan',
 u'first_name': u'Claude',
 u'last_name': u'Shannon'}
```

```python
json.dumps(parsed_json)
```

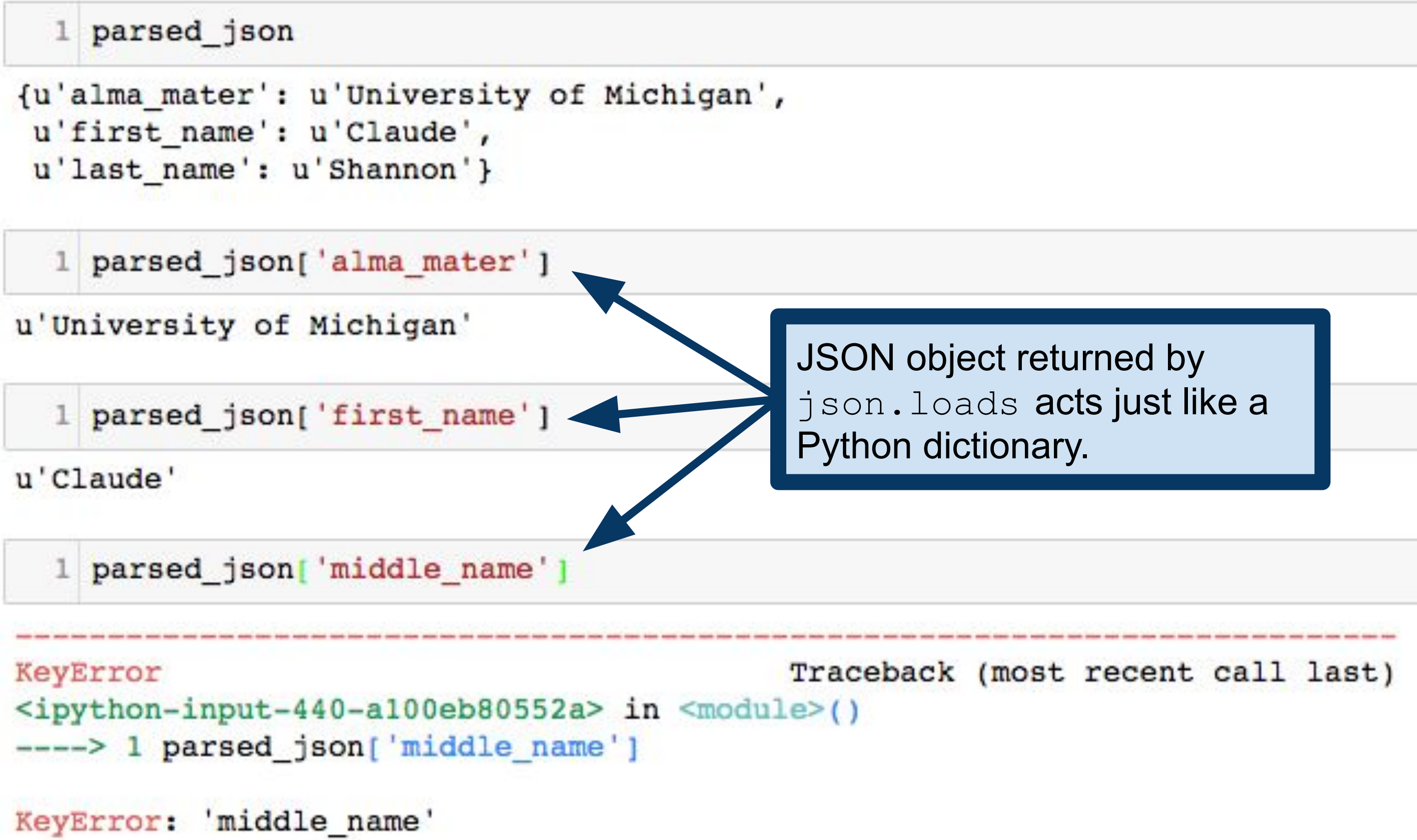
```
'{"alma_mater": "University of Michigan", "first_name": "Claude", "last_name": "Shannon"}'
```

JSON string encoding information about information theorist Claude Shannon

`json.loads` parses a string and returns a JSON object.

`json.dumps` turns a JSON object back into a string.

Python `json` module



```
parsed_json
```

```
{u'alma_mater': u'University of Michigan',
 u'first_name': u'Claude',
 u'last_name': u'Shannon'}
```

```
parsed_json['alma_mater']
```

```
u'University of Michigan'
```

```
parsed_json['first_name']
```

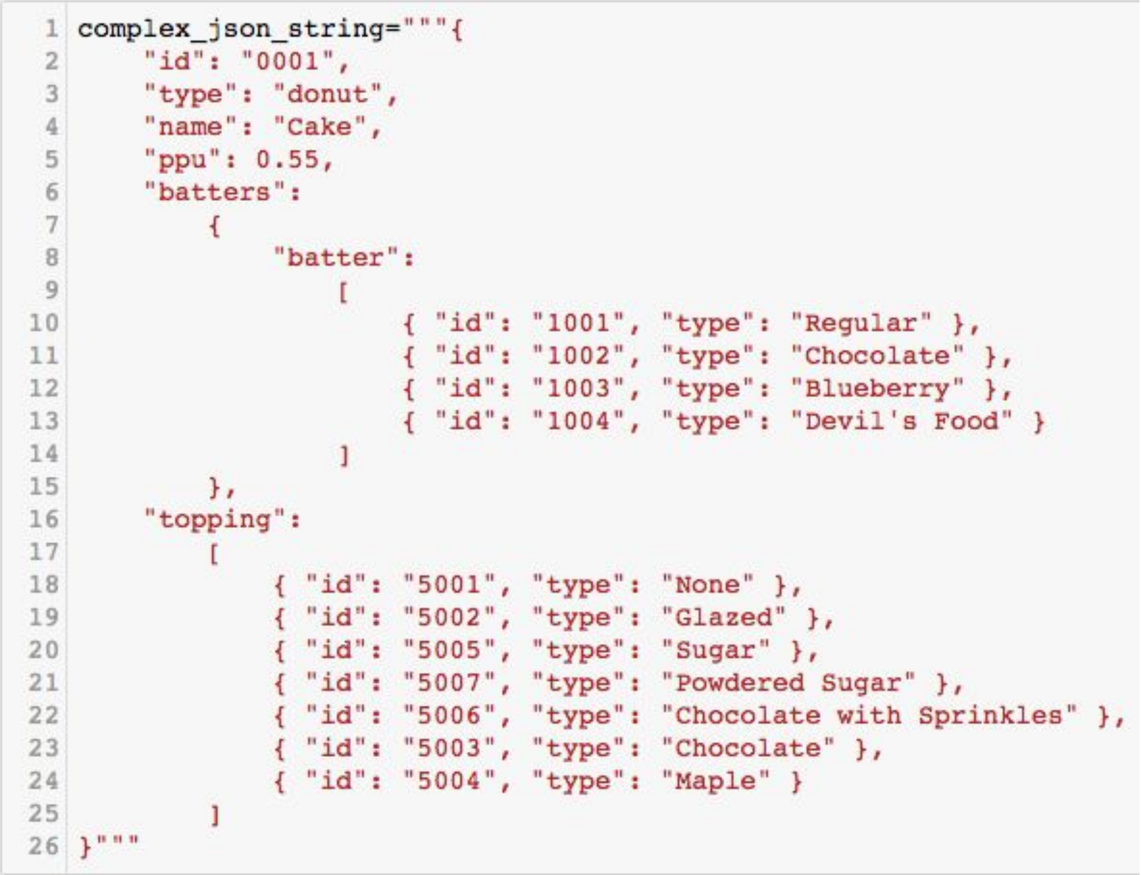
```
u'Claude'
```

```
parsed_json['middle_name']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-440-a100eb80552a> in <module>()
----> 1 parsed_json['middle_name']

KeyError: 'middle_name'
```

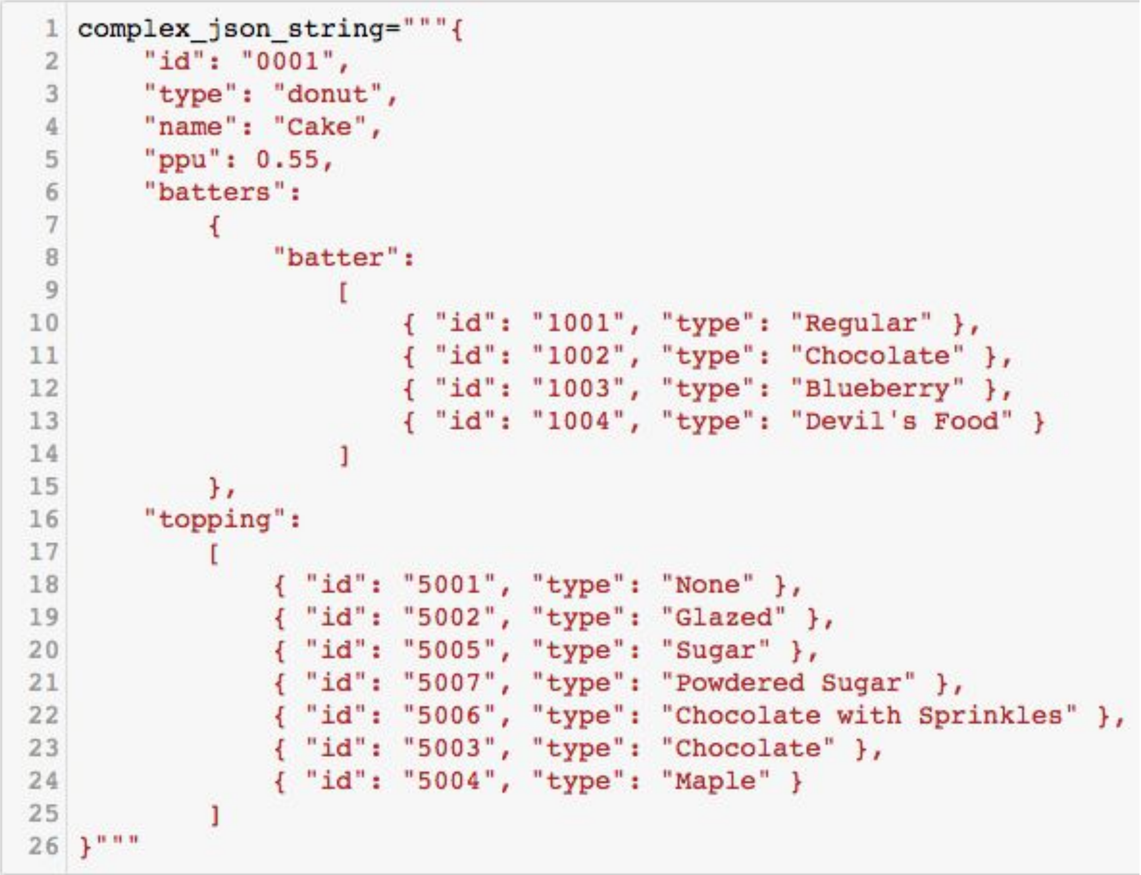
JSON object returned by `json.loads` acts just like a Python dictionary.

JSON objects can have very complicated structure



```
complex_json_string="""{
    "id": "0001",
    "type": "donut",
    "name": "Cake",
    "ppu": 0.55,
    "batters":
        {
            "batter":
                [
                    { "id": "1001", "type": "Regular" },
                    { "id": "1002", "type": "Chocolate" },
                    { "id": "1003", "type": "Blueberry" },
                    { "id": "1004", "type": "Devil's Food" }
                ]
        },
    "topping":
        [
            { "id": "5001", "type": "None" },
            { "id": "5002", "type": "Glazed" },
            { "id": "5005", "type": "Sugar" },
            { "id": "5007", "type": "Powdered Sugar" },
            { "id": "5006", "type": "Chocolate with Sprinkles" },
            { "id": "5003", "type": "Chocolate" },
            { "id": "5004", "type": "Maple" }
        ]
}"""
```

JSON objects can have very complicated structure



```
complex_json_string="""{
    "id": "0001",
    "type": "donut",
    "name": "Cake",
    "ppu": 0.55,
    "batters":
        {
            "batter":
                [
                    { "id": "1001", "type": "Regular" },
                    { "id": "1002", "type": "Chocolate" },
                    { "id": "1003", "type": "Blueberry" },
                    { "id": "1004", "type": "Devil's Food" }
                ]
        },
    "topping":
        [
            { "id": "5001", "type": "None" },
            { "id": "5002", "type": "Glazed" },
            { "id": "5005", "type": "Sugar" },
            { "id": "5007", "type": "Powdered Sugar" },
            { "id": "5006", "type": "Chocolate with Sprinkles" },
            { "id": "5003", "type": "Chocolate" },
            { "id": "5004", "type": "Maple" }
        ]
}"""
```

This can get out of hand quickly, if you’re trying to work with large collections of data. For an application like that, you are better off using a database, about which we’ll learn in our next lecture.