LING 388: Computers and Language

Lecture 13

Administrivia

- I'll be away next week (and the following week)
- Week after next is Spring Break anyway
- Lecture 14 will be pre-recorded and posted on the course website
  - there will be some regex exercises
  - optional: for extra credit only

Today's Topics

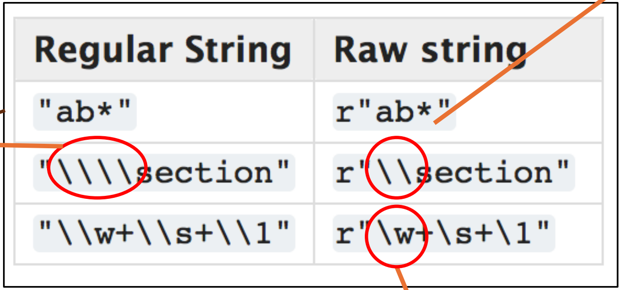
- Recall those string methods (*from last time*)?
  - `word.startswith(string)` *prefix*
  - `word.endswith(string)` *suffix*
  - `word.istitle()` `titlecase`
- More complex patterns:
  - regular expressions (*regex*)
  - A class exercise
- Homework 6

Python's re module

- Regular Expressions in Python:
  - a pattern matching language (*used by many other programming languages*)
  - the re module is written in C (*an efficient programming language*)
  - we write the pattern using a string, a raw string.
- https://docs.python.org/3/howto/regex.html

\ is a meta-character in re
\\ means a backslash character

to match a literal backslash, one has to write '\\\\' as the RE string, because the regular expression must be \\, and each backslash must be expressed as \\ inside a regular Python string literal.



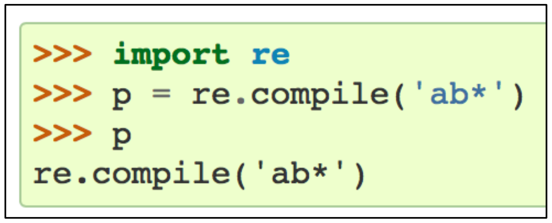
Regular String	Raw string
`"ab*"`	`r"ab*"`
`"\\\\section"`	`r"\\section"`
`"\\w+\\s+\\1"`	`r"\w+\s+\1"`

But, there's a big problem, called the *Backslash Plague*

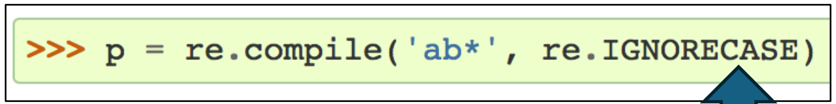
\w means an alphanumeric character in re
\s means a whitespace character in re
\1 means a reference to group 1

Python's re module

- One way (*not the easiest way*) is to compile a regex
- Then use the compiled pattern *p* using *p*.match()

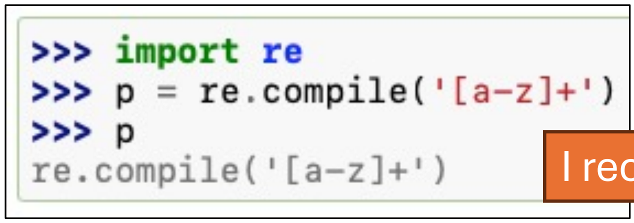


```
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

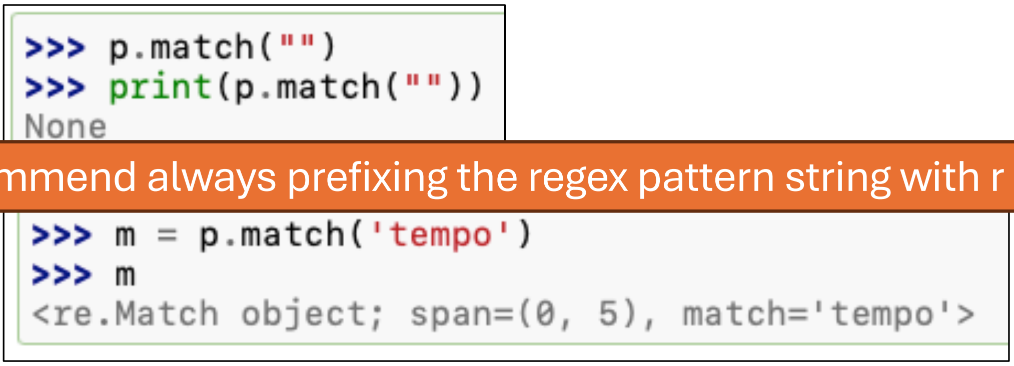


```
>>> p = re.compile('ab*', re.IGNORECASE)
```

option: *ignore upper vs. lower case*



```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```



```
>>> p.match("")
>>> print(p.match(""))
None
```

I recommend always prefixing the regex pattern string with r

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Please Read!

- More examples from https://docs.python.org/3/howto/regex.html

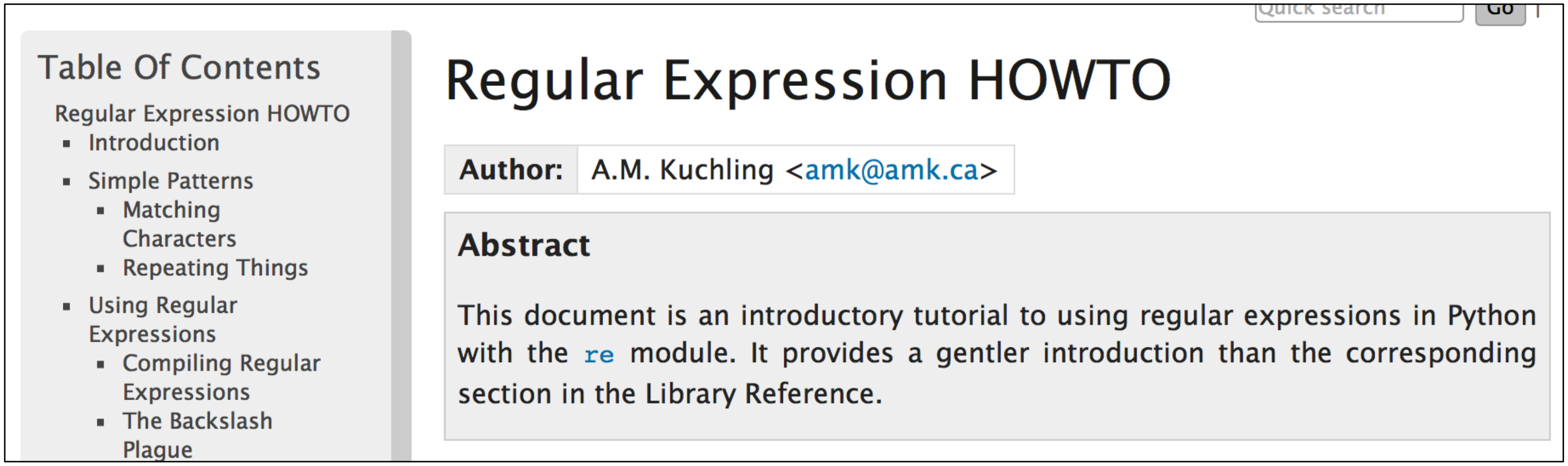


Table Of Contents

Regular Expression HOWTO
- Introduction
- Simple Patterns
  - Matching Characters
  - Repeating Things
- Using Regular Expressions
  - Compiling Regular Expressions
  - The Backslash Plague

Regular Expression HOWTO

Author: A.M. Kuchling <amk@amk.ca>

Abstract

This document is an introductory tutorial to using regular expressions in Python with the `re` module. It provides a gentler introduction than the corresponding section in the Library Reference.

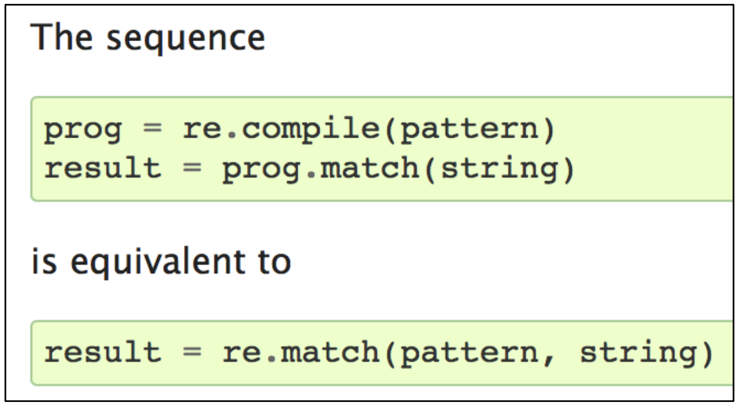
Python regex

Unicode characters ok in Python 3.x

- **Summary**:
  - \w a character [A-Za-z0-9_]
  - \d [0-9]
  - \b word boundary
  - \s space character [\t\n\r\f\v]
- **Operators**:
  - * zero or more repeats
  - + one or more repeats
  - ( ) grouping
- Raw string prefix (avoid needing to escape backslash \):
  - r"\w+"

- **Negation**: (uses capitalized version of the lowercase meta-characters)
  - \W anything not in \w
  - \D anything not in \d
- Full Documentation:
  https://docs.python.org/3/library/re.html

Python's re module



The sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```
result = re.match(pattern, string)
```

Python has two ways to use regexs:

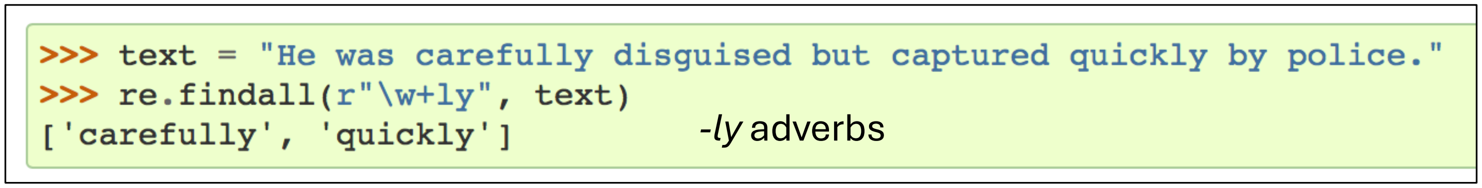
- *this can be very confusing for a beginner*

Method 1:

- explicitly compile a regex,
- then call *regex*`.match(`*string*`)`
- disadvantage: two statements needed

Method 2 (simpler):

- `re.match(`*regex*`, `*string*`)`



```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

-ly adverbs

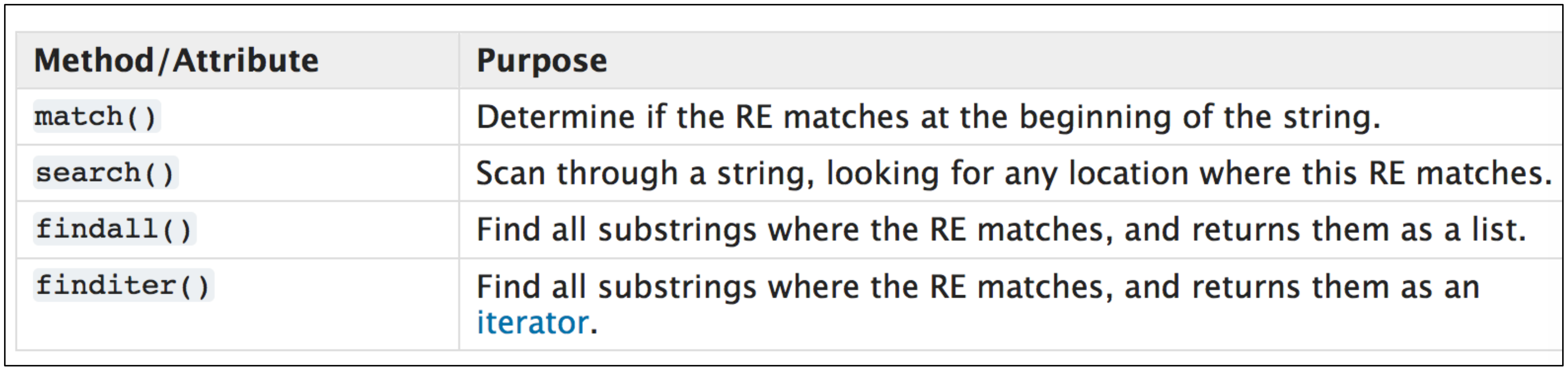
Python's re module

- `re.findall(regex, string)`
- Example:
  - `import re`
  - `text = "Quickly or slowly we go to Alyssa's house"`
  - `re.findall(r"\w+ly", text)`
- Do you see what's strange about the result?

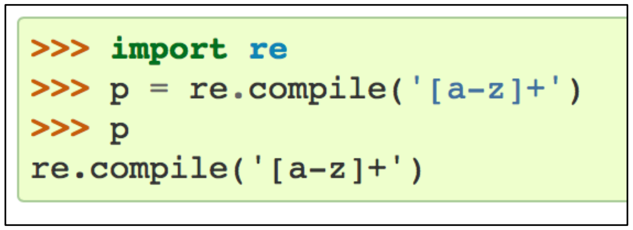
Python's re module

- **`re.findall(regex, string)`**
- Example:
  - `import re`
  - `text = "Quickly or slowly we go to Alyssa's house"`
  - `re.findall(r"\w+ly\b",text)`
- \b means word boundary.
- Can you see the difference in the result?

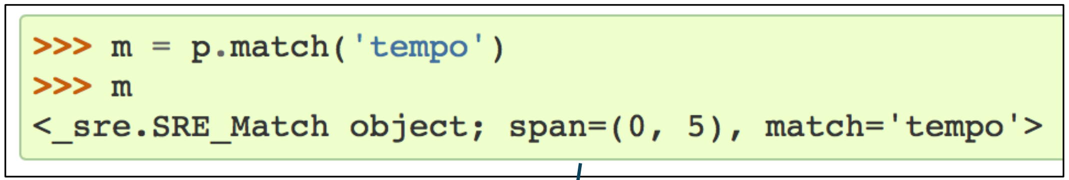
Python's re module



Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator.



```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```



```
>>> m = p.match('tempo')
>>> m
<_sre.SRE_Match object; span=(0, 5), match='tempo'>
```

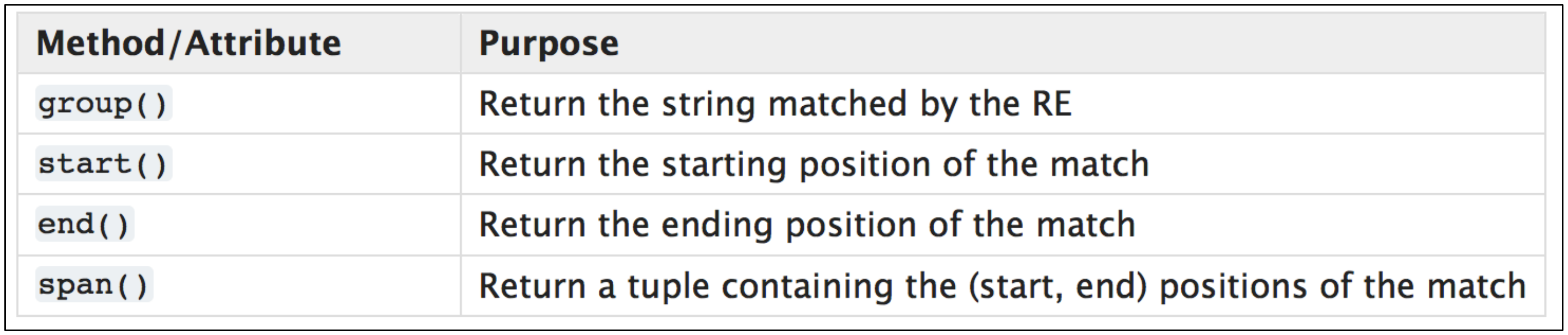
can access this components using methods on the match object

Python's re module

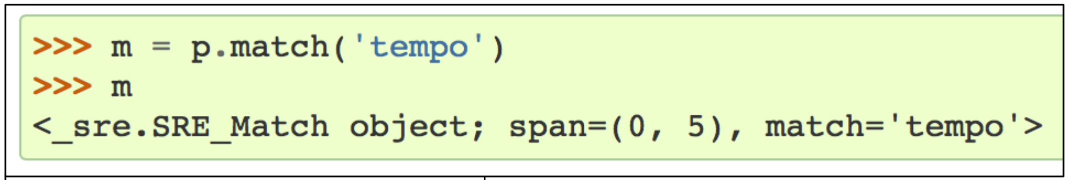
- Another potentially confusing part of the regex implementation is the result of `re.match(regex, string)`
  - If there is no match, None is returned.
  - If there is a match, a **match object** is returned.
- Can use this in a condition, e.g.:

```
if re.match(regex, string):
   print("Matched!")
else:
   print("Sorry, no match!")
```

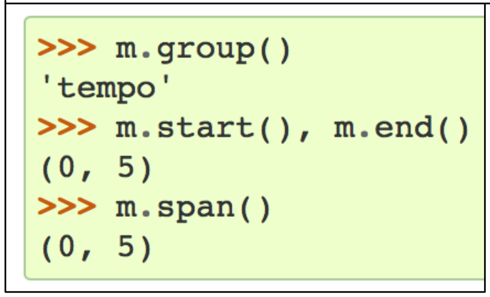
Python's re module



Method/Attribute	Purpose
`group()`	Return the string matched by the RE
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple containing the (start, end) positions of the match



```
>>> m = p.match('tempo')
>>> m
<_sre.SRE_Match object; span=(0, 5), match='tempo'>
```



```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

Python's re module

- Grouping (using round brackets) can be very useful
- Example (from a few slides back):

```
 import re
text = "Quickly or slowly we go to Alyssa's house"
re.findall(r"\w+ly\b",text)
['Quickly', 'slowly']
```

- But what if we didn't want the 'ly' suffix?
- Solution:

```
re.findall(r"(\w+)ly\b",text)
['Quick', 'slow']
```

Python's re module

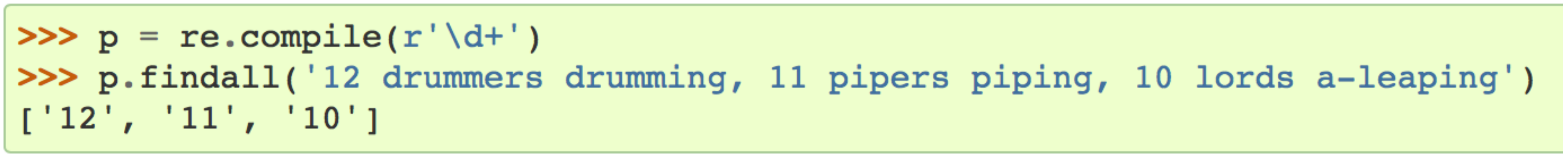
- Grouping (using round brackets) can be very useful
- What if there is more than one group in the regex?
- Example:
    - `text = "Quickly or slowly we go to Alyssa's house"`
    - `re.findall(r"(\w+)(ly)\b",text)`
    - `[('Quick', 'ly'), ('slow', 'ly')]`
- Answer:
    - it returns them as tuples

Class Exercise

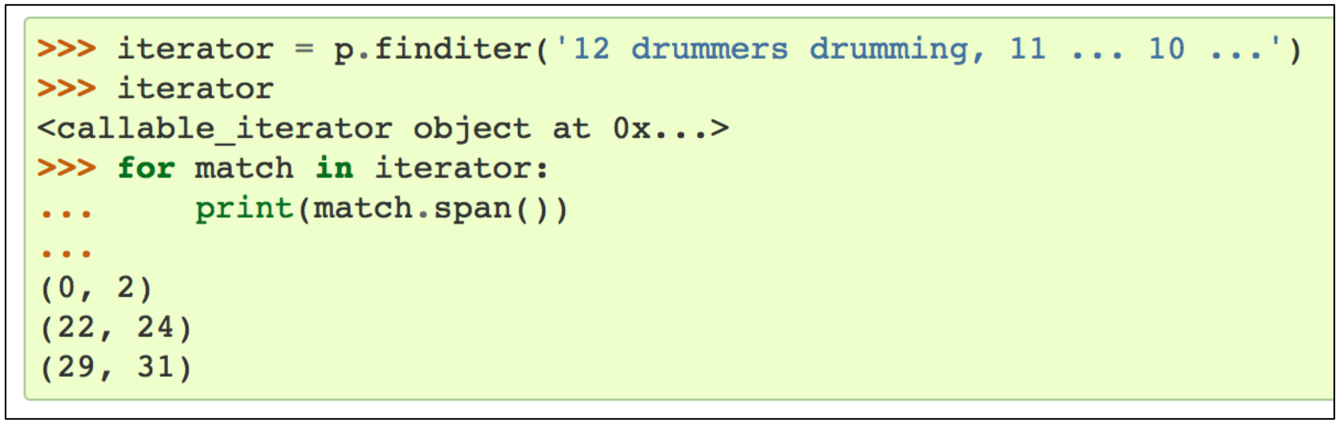
- *Looking Glass* again:
  - `raw = open('looking-glass.txt', encoding='utf-8', errors='ignore').read()`
- *-ly* search:
  - how many words end in *–ly* and what is the frequency distribution?

```
1. len(re.findall(r"\b\w+ly\b", raw))
2. len(set(re.findall(r"\b\w+ly\b", raw)))
3. nltk.FreqDist(re.findall(r"\b\w+ly\b", raw))
```

Python's re module

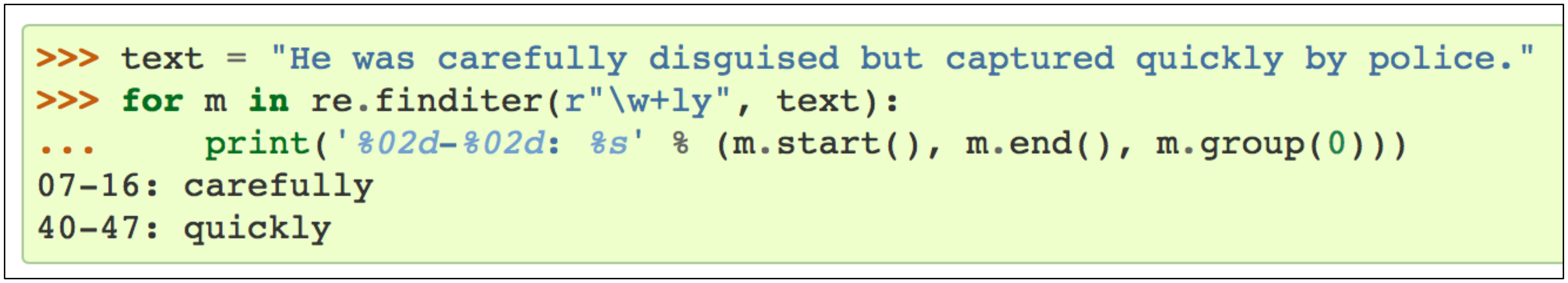


```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```



```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

Python's re module



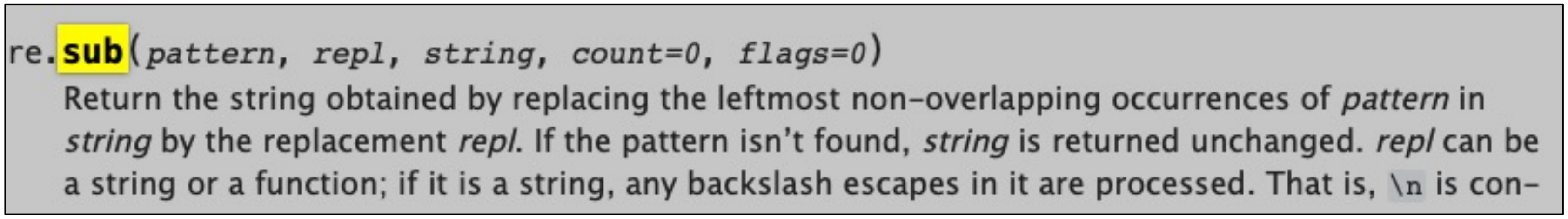
```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

re.sub()

Recall `.split()` (*by space*) vs. `nltk.word_tokenize()`?

- String:

```
p1 = 'Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do. Once or twice she had peeped
into the book her sister was reading, but it had no pictures or
conversations in it, "and what is the use of a book," thought Alice,
"without pictures or conversations?"'
```



re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is con-

- Code:

```
p2 = p1.split()
import re
for word in p2:
      print(re.sub(r"[?\"',.]", "", word))
```

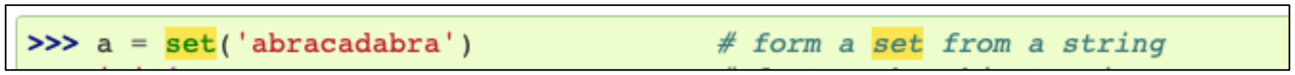
replace any punctuation symbol here by "" (empty string), i.e. delete it

Homework 6

- Question 1:
  - create a new corpus by lowercasing all the words in the *Jane Austen* novel `austen-emma.txt` in *nltk.corpus.gutenberg*
    - *see previous lecture for how to access the words*
  - *recall a corpus is just a list of words*
    - *you can use a list comprehension and .lower() to create a new list of words and assign the result to a variable.*
  - What is the vocabulary size before and after lowercasing?

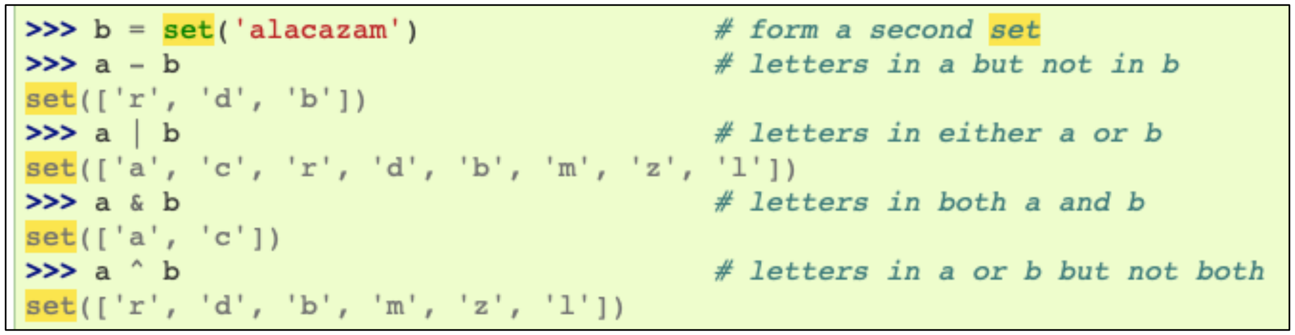
Homework 6

- Question 2: consider sets



```
>>> a = set('abracadabra')                    # form a set from a string
```

- Python has some set operations, e.g. – (*difference*), | (*union*), & (*intersection*) and ^ (*union but excluding the ones in common*).



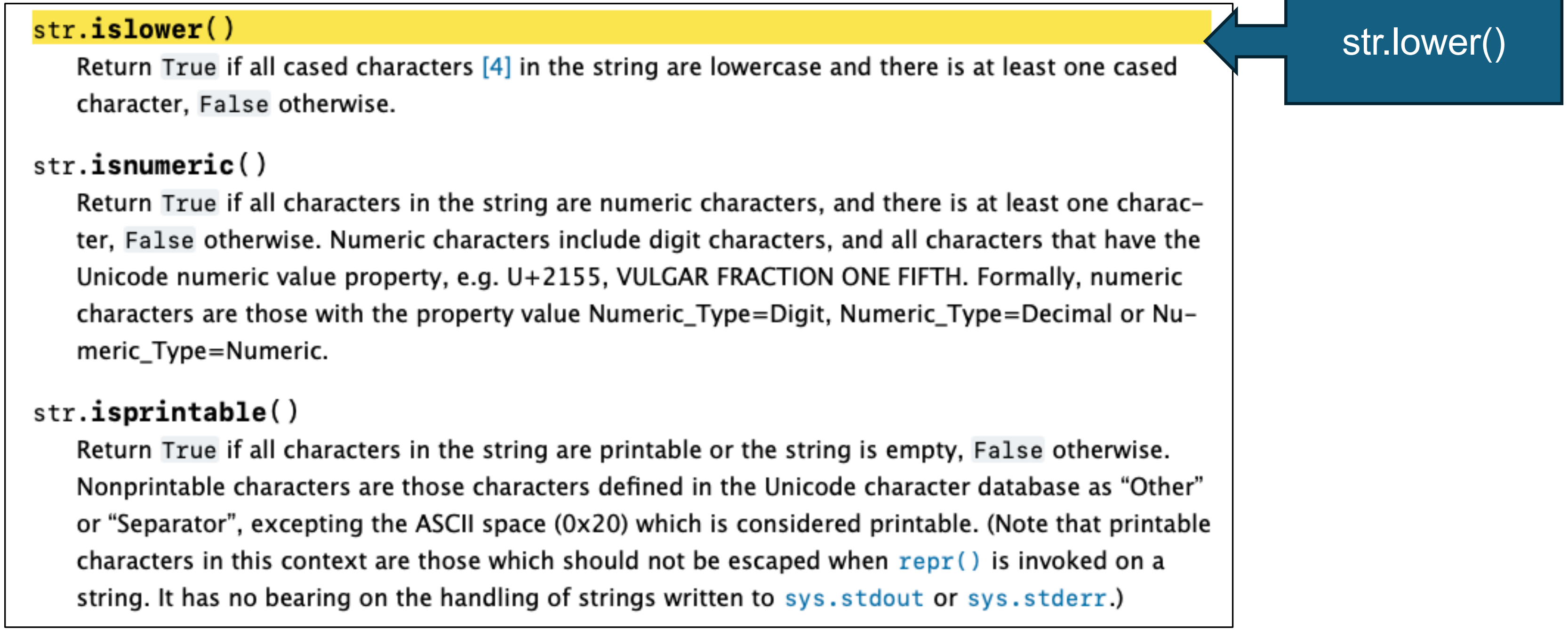
```
>>> b = set('alacazam')                       # form a second set
>>> a - b                                     # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                                     # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                                     # letters in both a and b
set(['a', 'c'])
>>> a ^ b                                     # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Homework 6

- Question 2:
  - Python set operation: – (difference)
  - Suppose `vocab` (*original*) and `lcvocab` (*lc* = lowercase) are our sets.
  - Compute the size of
    - `vocab – lcvocab` and
    - `lcvocab – vocab`.
  - Explain the two different set difference operations, i.e. *what they compute for vocabulary*, why they do different things and give different results.
  - Can you give example(s) to illustrate your answer?

Homework 6

https://docs.python.org/3/library/stdtypes.html#str.islower



str.**islower**()

Return `True` if all cased characters [4] in the string are lowercase and there is at least one cased character, `False` otherwise.

str.**isnumeric**()

Return `True` if all characters in the string are numeric characters, and there is at least one character, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

str.**isprintable**()

Return `True` if all characters in the string are printable or the string is empty, `False` otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

Homework 6

- Submit to sandiway@arizona.edu
- SUBJECT: 388 Homework 6 *YOUR NAME*
- One PDF file only
  - include Python terminal and any screenshots in your answer
- Deadline:
  - midnight Monday