# *Regular Expressions in Python*

## *What Is a Regular Expression?*

- A pattern that matches all, or part of, some desired text string
- Pattern is compared to a given "target" text string
- *Succeeds* or *Fails* depending on whether the target string contains the desired text string
- Python syntax:

```
re.search( 'pattern', 'target-string' )
```

## *Two aspects of using regular expressions*

- Identifying a desired *pattern* to search for
  - "a 10-digit phone number"
  - "a first name, middle initials, last name"
  - "a color, followed by a noun"
- Using regular expression *syntax* to create the desired pattern
  - the main concern of these slides

## *first RE exercise – see the power*

- download and open file "[montcs.bloomu.edu/Datasets/Logfiles/error.log.1](montcs.bloomu.edu/Datasets/Logfiles/error.log.1)"
- *search for lines containing "Mar 11"*
  - **import re**
  - mo = re.search(**r**"Mar 11", line)
  - print( mo.group() )
  - print( mo.groups() ) # "groups(), not group()
- *search for lines containing any date in March*
  - mo = re.search(**r**"Mar \d\d", line)
  - print( mo.group() , '\n', mo.groups() ) # "group() versus groups()
- *search for lines containing "Mar \d\d", keep the date parts*
  - mo = re.search(**r**"Mar **(**\d\d**)**", line)
  - print( mo.group() , '\n', mo.groups() ) # "group() versus groups()
- *search for lines containing "Mar \d\d", keep the dates and times*
  - mo = re.search(**r**"Mar **(**\d\d**)** **(**\d\d**:**\d\d**:**\d\d**)**", line)
  - print( mo.group() , '\n', mo.groups() ) # "group() versus groups()

## *A remark…*

- Regular expressions are implemented in many languages, in not-quite-identical ways.
  - Simple Regular Expressions
  - POSIX Basic Regular Expressions (BREs)
  - POSIX Extended Regular Expressions (EREs)
  - Perl Compatible Regular Expressions (PCREs)
- Python uses a version of PCRE
  - We will look at regular expressions ("regexes") in Python. Most of the material will hold for other versions as well.
- A summary is available in montcs.bloomu.edu/Information/Regex/cheatsheet.pdf

## *Simple Examples*

- import re – use regular expressions
- mo = re.search( 'dab', 'abracadabra' )
  - is successful, mo.group() contains 'dab'
  - similar to 'abracadabra'.find('dab' )
    - doesn't show regexes' power
- mo = re.search('E', 'ABEadeABCDE')
  - successful, mo.group() contains the *first* 'E'
- lst = re.findall('E', 'ABEadeABCDE')
  - successful, lst contains both 'E' occurrences

## More examples

- mo = re.search( 'dab', 'hocus-pocus' )
  - fails: mo equals None
- re.search('dog', 'digsANDdogs')
  - successful
  - matches the *conjunction* of 'd', 'o', and 'g'
- mo = re.search( 'Cat', 'catch' )
  - fails *unless* case-sensitivity is turned off

## Search Details

- match_object = re.**search**(
  regular_expression , target_string ,
  startpos , endpos )
  - match_object contains information about the match
    - what exactly matched, where, etc.
    - (None if no match)
  - startpos, endpos are *optional* – specify where to start, stop searching within target_string
- match_object = re.**match**(
  regular_expression , target_string )
  - like re.search(), but regular expression must match *entire* target string

## *More remarks*

- Regular expressions are commonly called *regexes*, or *R.E.s*
- *Interesting* regexes form variable patterns, *i.e.* can match more than one distinct string
- Useful regexes are formed to match a desired category of strings
  - example: a phone number –
    a string of 3 digits, a separating character, 3 more digits, another separating character, then 4 digits

## *Building a regex, formally*

- Each distinct, matchable character in the pattern is an *atom*
  - a single atom is a valid, minimal regex
- Two adjacent atoms are a *conjunction*
  - logical "AND"
  - a conjunction is also a valid regex
- A *disjunction* (logical "OR") of two regexes forms a regex
  - symbolized with "**|**" (a "pipe" or "vertical bar")
- A regex in parentheses is an "atomic" regex
  - Can be used in a conjunction or disjunction

## *Disjunction examples*

- mo = re.search('a|b|c|d', 'ABCDbcdefg')
  - successful, mo.group() contains the 'b'
- mo = re.search('dog|cat', 'catsANDdogs')
  - successful, mo.group() contains 'cat'
- mo = re.search('do(g|c)', 'documentdogs')
  - successful, mo.group() contains 'doc'
- mo = re.search('|', 'abcd|efgh')
  - successful (?)
  - mo.group() probably isn't what you expect

## *Atoms*

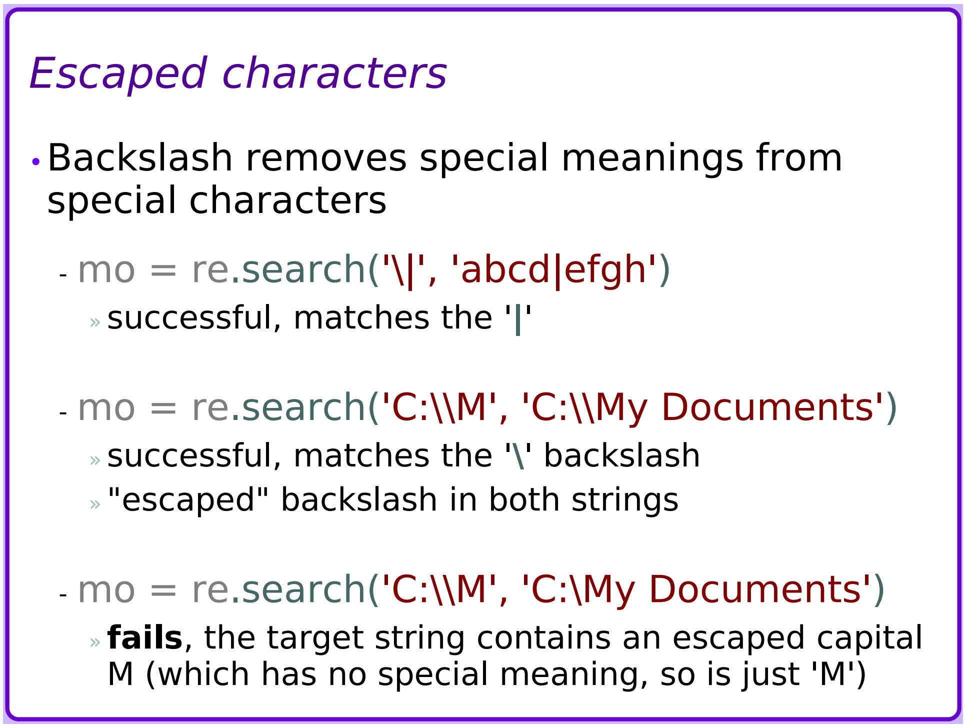
- A normal character matches itself
  - called a *literal*
  - Previous examples mostly consisted of literals
- *Escape sequences* represent some literals
  - '**\n**', '**\t**'
- Some characters have special meanings in regexes
  - period **.** , caret **^** , dollar sign **$**
  - vertical bar **|** , question mark **?** , asterisk ***** , plus **+**
  - Parentheses **()** , square brackets **[]** , curly braces **{}**
  - backslash **\**

## *Special Characters*

- Periods are very special atoms
  - Match *any* single character (with a few exceptions)
- Caret, dollar sign
  - *Positional* items, match at a location instead of a character
  - caret **^** – matches the beginning of a string
  - dollar sign **$** – matches the end of a string
- Example:
  - mo = re.search('^dog|cat$', 'catsANDdogs')
    - fails – requires "dog" at beginning of line or "cat" at end of line

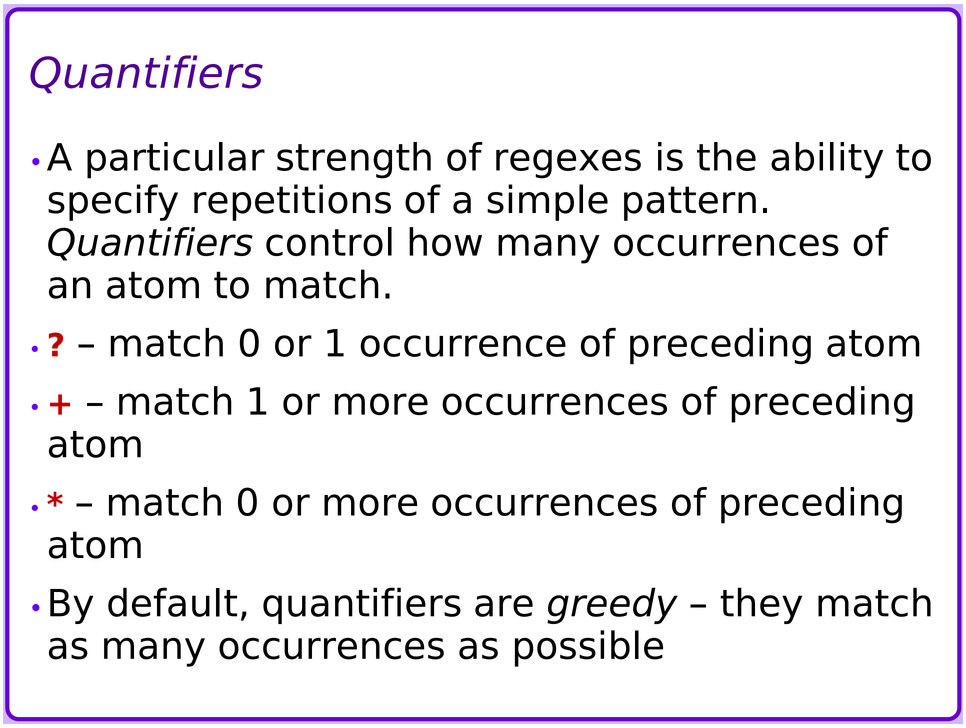
## *Special character examples*

| pattern | matches |
|---|---|
| d**.**g | dog, dig, dDg, d.g … |
| d\.g | d.g *only* |
| dog***** | do, dog, dogg, dogggggggggg … |
| dog**+** | dog, dogg, dogggggggggg … |
| dog**?** | do, dog |
| **^**dog | dog *at beginning of string only* |
| dog**$** | dog *at end of string only* |
| **[**dog**]** | d, o, *or* g *only* |
| **[**aeiouAEIOU**]** | *any uppercase or lowercase vowel* |
| **(**dog**)** | dog *as a group* |



## *Escaped characters*

- Backslash removes special meanings from special characters
  - `mo = re.search('\|', 'abcd|efgh')`
    - successful, matches the '|'
  - `mo = re.search('C:\\M', 'C:\\My Documents')`
    - successful, matches the '\' backslash
    - "escaped" backslash in both strings
  - `mo = re.search('C:\\M', 'C:\My Documents')`
    - **fails**, the target string contains an escaped capital M (which has no special meaning, so is just 'M')



## *Quantifiers*

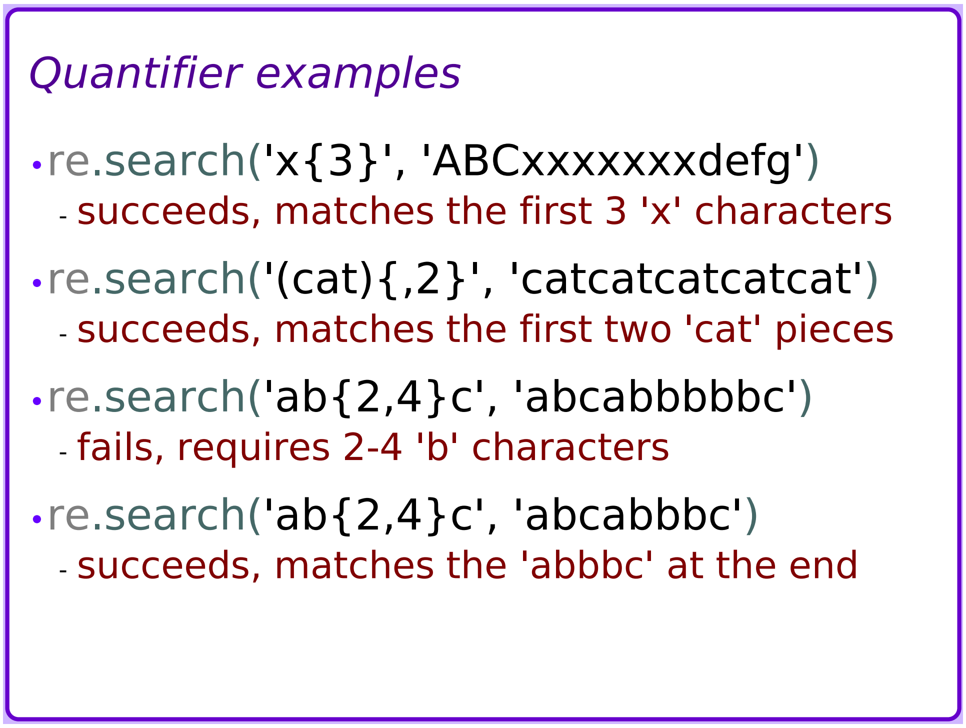
- A particular strength of regexes is the ability to specify repetitions of a simple pattern. *Quantifiers* control how many occurrences of an atom to match.
- **?** – match 0 or 1 occurrence of preceding atom
- **+** – match 1 or more occurrences of preceding atom
- ***** – match 0 or more occurrences of preceding atom
- By default, quantifiers are *greedy* – they match as many occurrences as possible

## *Simple quantifier examples*

- **abcd?efg**
  - Matches abcefg
  - Matches abcdefg
  - Doesn't match abcddefg
- **abcd+efg**
  - Matches abcdefg
  - Matches abcddefg
  - Matches abcddddddefg
  - …
- **abcd*efg**
  - Matches abcefg
  - Matches abcdefg
  - Matches abcddddddddefg
- **abc\d*efg**
  - Matches abcefg
  - Matches abc7efg
  - Matches abc9876543210efg

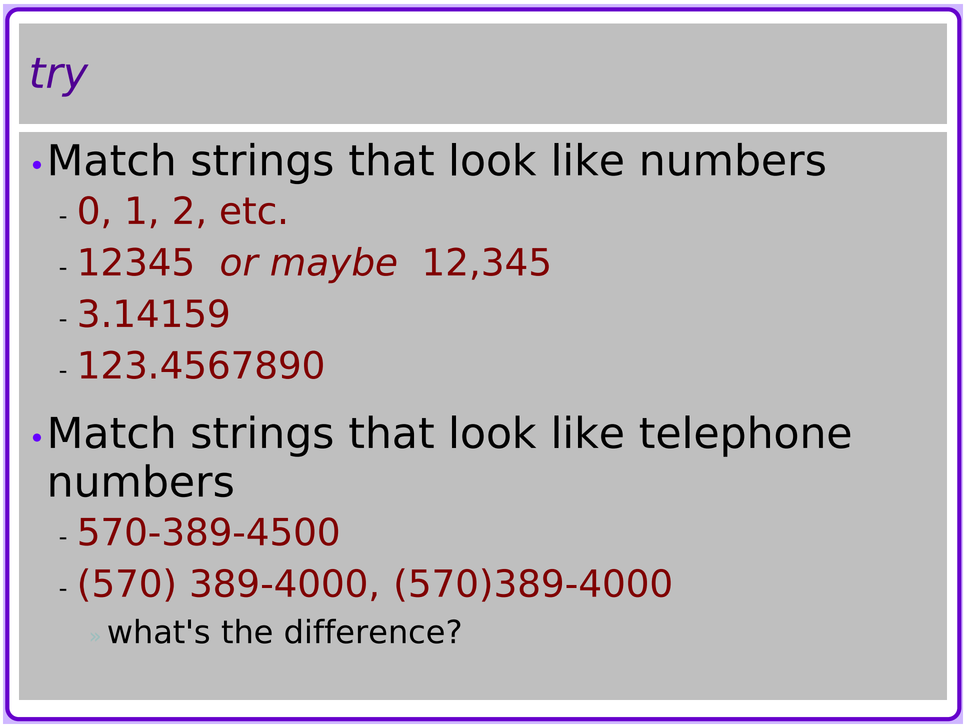
## *Constrained Quantifiers*

- Curly braces define a *range* of matches:
  - **`{n}`** – match exactly ***n*** instances of the preceding atom
  - **`{n,m}`** – match between ***n*** and ***m*** instances of the preceding atom
  - **`{n,}`** – match *at least* ***n*** instances of the preceding atom
  - **`{,m}`** – match *at most* ***m*** instances of the preceding atom



## *Quantifier examples*

- re.search('x{3}', 'ABCxxxxxxdefg')
  - succeeds, matches the first 3 'x' characters
- re.search('(cat){,2}', 'catcatcatcatcat')
  - succeeds, matches the first two 'cat' pieces
- re.search('ab{2,4}c', 'abcabbbbbc')
  - fails, requires 2-4 'b' characters
- re.search('ab{2,4}c', 'abcabbbc')
  - succeeds, matches the 'abbbc' at the end



## *try*

- Match strings that look like numbers
  - 0, 1, 2, etc.
  - 12345 *or maybe* 12,345
  - 3.14159
  - 123.4567890
- Match strings that look like telephone numbers
  - 570-389-4500
  - (570) 389-4000, (570)389-4000
    - what's the difference?

## *Special Characters - Summary*

| name | symbol(s) | meaning |
|---|---|---|
| period | **.** | match any single character |
| caret | **^** | match at beginning of string |
| dollar sign | **$** | match at end of string |
| asterisk | ***** | match arbitrary number (0 or more) of preceding regex |
| plus sign | **+** | match 1 or more of preceding regex |
| question mark | **?** | match 0 or 1 of preceding regex |
| square brackets | **[ ]** | match any 1 of the characters within the brackets |
| parentheses | **( )** | collect ("group") a regex into an atom; can be used with *** + ?** |
| curly braces | **{*n*} {*n*, *m*}**<br>**{*n*, } { , *m*}** | requires exactly *n* occurrences, or at least *n* and no more than *m*, of the preceding regex |
| backslash | **\** | escapes (cancels) the following character's special meaning |

## *Character classes*

- A class matches any one of a set of characters:
- Predefined classes represented by escaped characters
  - **\d** – matches any single numeric digit 0 .. 9
  - **.** – "universal class", matches any character
- User-defined classes created with **[ ]**
  - **[aeiouAEIOU]** – matches any single vowel

## exercise

- download and open file "montcs.bloomu.edu/Readings/Alice-in-Wonderland.txt"
- search for, count, and save lines containing any occurrence of the word "the" - but not words containing "the", such as "other"
  - print count of matching lines
- search for, count, and save lines containing words that include "the", such as "other" - but not "the" by itself
  - print count of matching lines
  - count number of occurrences of each word?
  - combine capitalization variants, such as "father" and "Father"?

## *Predefined character classes*

| escaped character | Class of characters |
|---|---|
| **\d** | any digit<br>*(same as* [0123456789] *or* [0-9] *)* |
| **\D** | any non-digit |
| **\w** | any "wordlike" character<br>*(any alphanumeric)*<br>*(same as* [a-zA-Z0-9_] *)* |
| **\W** | any non-wordlike character<br>*(punctuation, whitespace, etc.)* |
| **\s** | any "whitespace" character<br>*(same as* [ \n\t\r\v\f] *)* |
| **\S** | any non-whitespace character |
| **\b** | any word *boundary* |
| **\B** | any non-word boundary |

## *Using predefined classes in patterns*

  - '\' is an escape character in normal text strings, as well as in regexes

- Text string as regex: the escape character itself must be escaped
  - **'\\bbook\\b'** – matches " book " but not "textbook" or "bookie"
- Alternative:
  use *raw* strings, indicated with an 'r'
  - r'**\bbook\b**' – also matches " book " but not "textbook" or "bookie"

## *…general guideline*

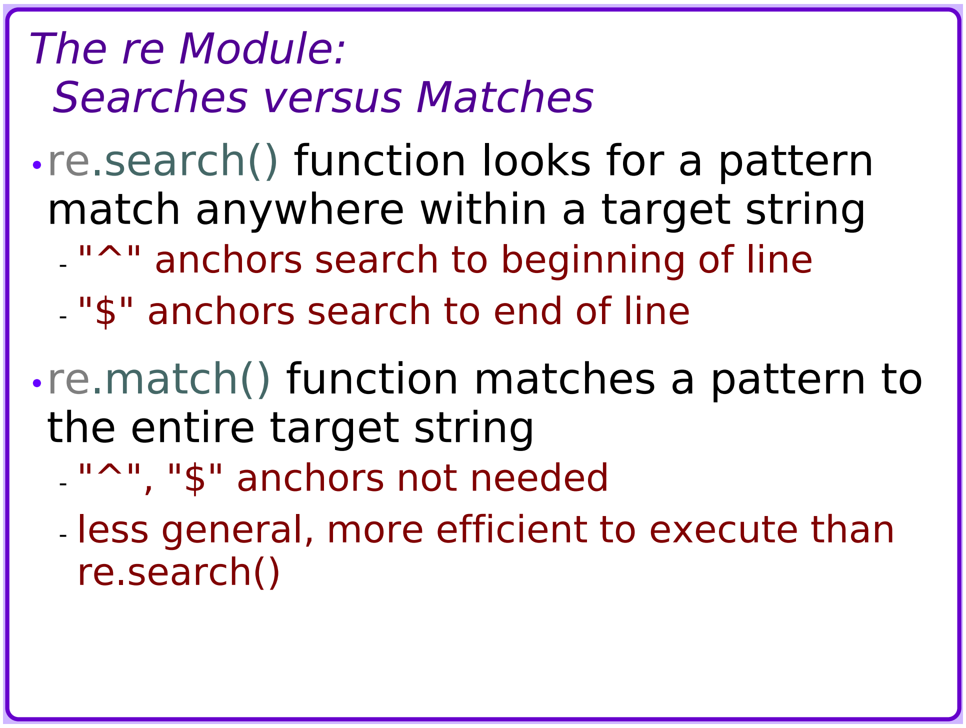
- Use raw strings to define regexes
- Always works, avoids some ambiguities
- For example:
  - re.search( r'ate\n', 'skate\n' ) – matches,
    is the same as
    re.search( 'ate\n', 'skate\n' ) – matches
  - re.search( r'ate\\n', 'skate\\n' ) – matches,
    is *not* the same as
    re.search( 'ate\\n', 'skate\\n' ) – NO match
    - matches against 'skate\n' instead

## *User-defined character classes*

- Square brackets `[]` create classes
  - Any character within brackets is matched
- Examples:
  - `[02468]` – matches any even digit (or 0)
  - ``[:;.,?-_=+%&!~^&*$@#()`'"]`` – matches any of a bunch of punctuation symbols
- Contiguous ranges of characters allowed
  - `[-.,0-9]` – matches minus sign, period, comma, or any numeric character
    - dash at beginning is just itself

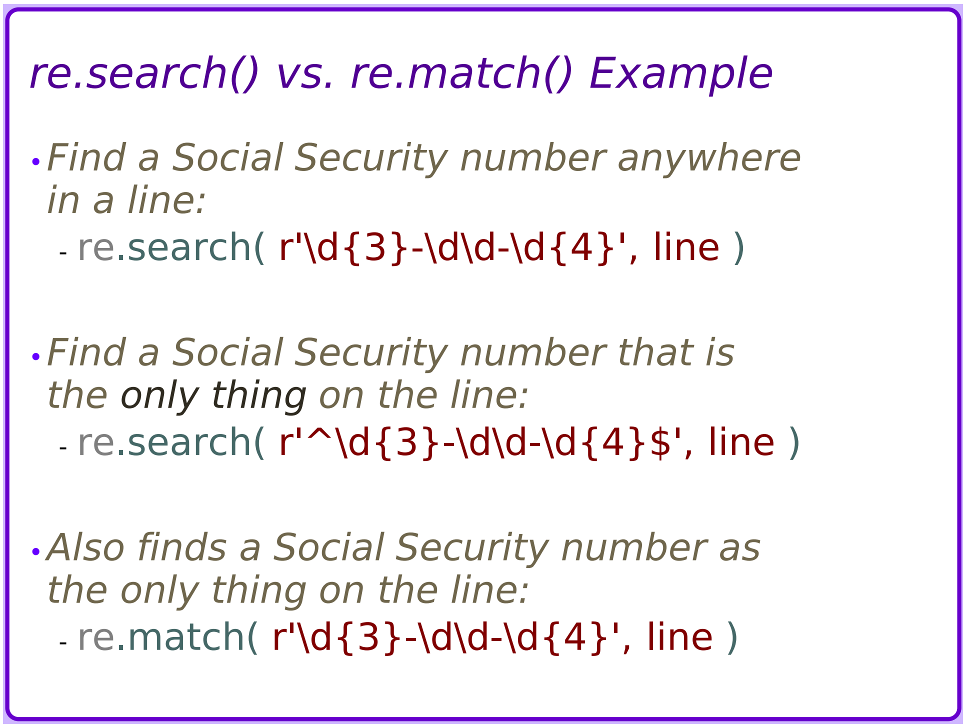
## *Negated classes and characters*

- Caret at the *beginning* of a class negates the category
  - `[^aeiou]` – matches any character *except* a lowercase vowel
  - `[aeiou^]` – matches any lowercase vowel, or a caret
- Class can contain single character
  - Provides negation of single characters
  - `[^X]` – matches any character *except* X
  - `[^X]+` – matches one or more non-X chars



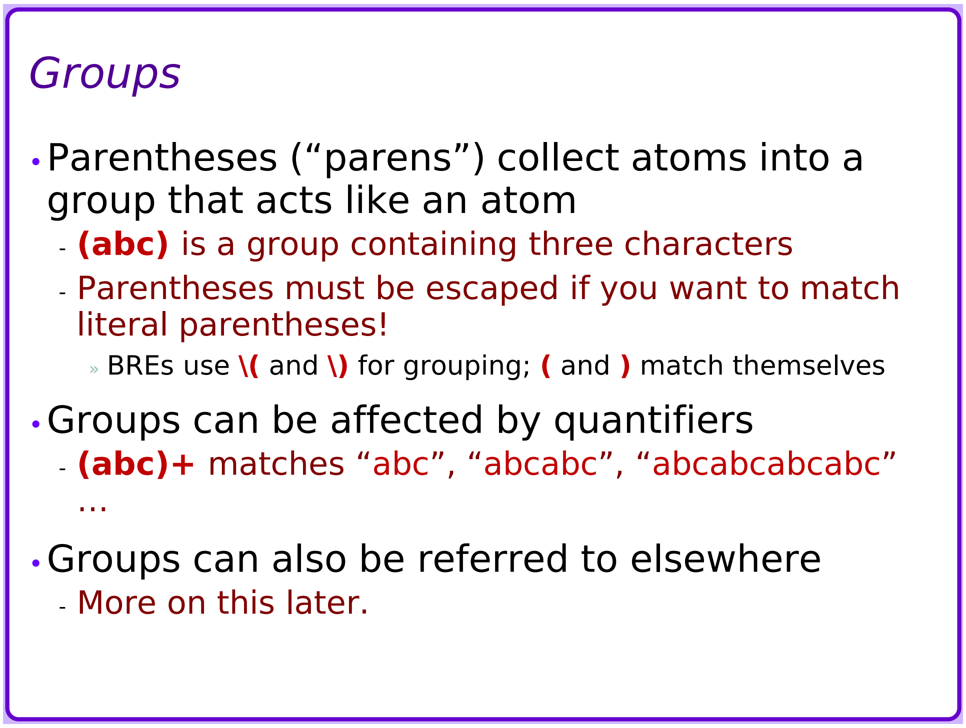
## *The re Module: Searches versus Matches*

- re.search() function looks for a pattern match anywhere within a target string
  - "^" anchors search to beginning of line
  - "$" anchors search to end of line
- re.match() function matches a pattern to the entire target string
  - "^", "$" anchors not needed
  - less general, more efficient to execute than re.search()



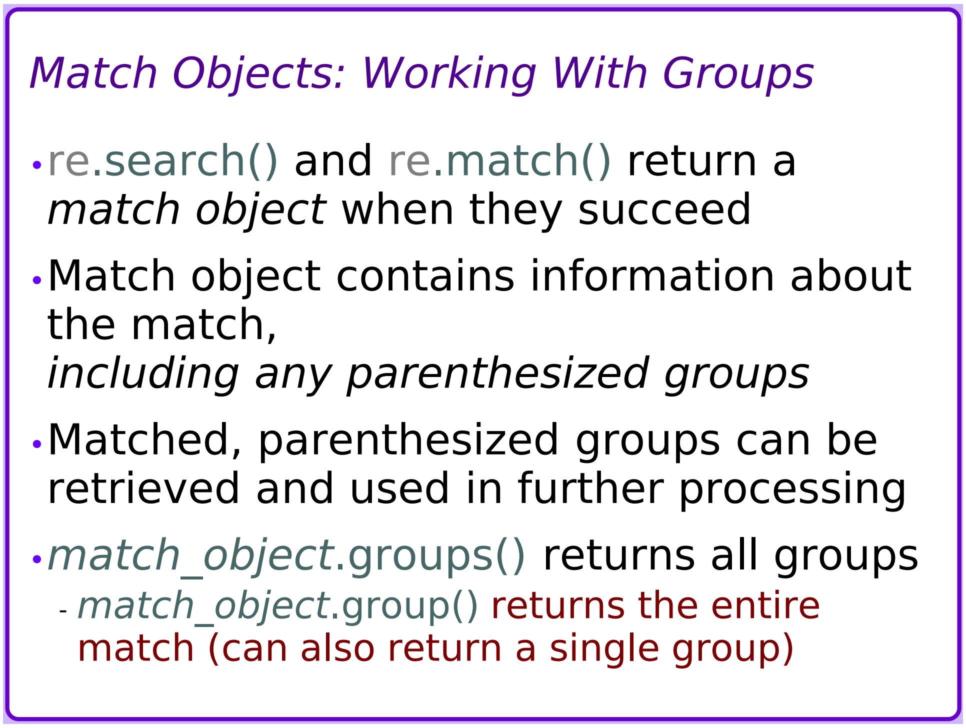
## *re.search() vs. re.match() Example*

- *Find a Social Security number anywhere in a line:*
  - re.search( r'\d{3}-\d\d-\d{4}', line )
- *Find a Social Security number that is the only thing on the line:*
  - re.search( r'^\d{3}-\d\d-\d{4}$', line )
- *Also finds a Social Security number as the only thing on the line:*
  - re.match( r'\d{3}-\d\d-\d{4}', line )



## *Groups*

- Parentheses (“parens”) collect atoms into a group that acts like an atom
  - **(abc)** is a group containing three characters
  - Parentheses must be escaped if you want to match literal parentheses!
    - BREs use **\(** and **\)** for grouping; **(** and **)** match themselves
- Groups can be affected by quantifiers
  - **(abc)+** matches “abc”, “abcabc”, “abcabcabcabc” …
- Groups can also be referred to elsewhere
  - More on this later.



## *Match Objects: Working With Groups*

- re.search() and re.match() return a *match object* when they succeed
- Match object contains information about the match,
  *including any parenthesized groups*
- Matched, parenthesized groups can be retrieved and used in further processing
- *match_object*.groups() returns all groups
  - *match_object*.group() returns the entire match (can also return a single group)

## Groups Example

a) What will this pattern match with?

`\b(\w+)\b.*\b(\w+)\b`

b) How do you enter this pattern in Python?

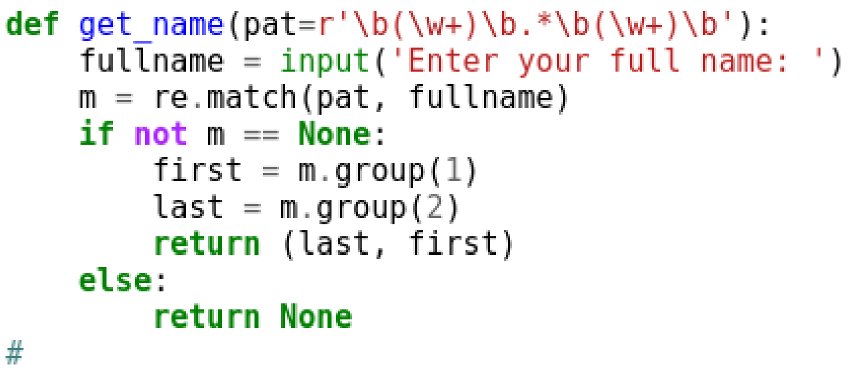
→ `"\\b(\\w+)\\b.*\\b(\\w+)\\b"`

c) or with a raw string,

`r"\b(\w+)\b.*\b(\w+)\b"`

## Groups Example 2

- Enter this function, then run it and print the result:



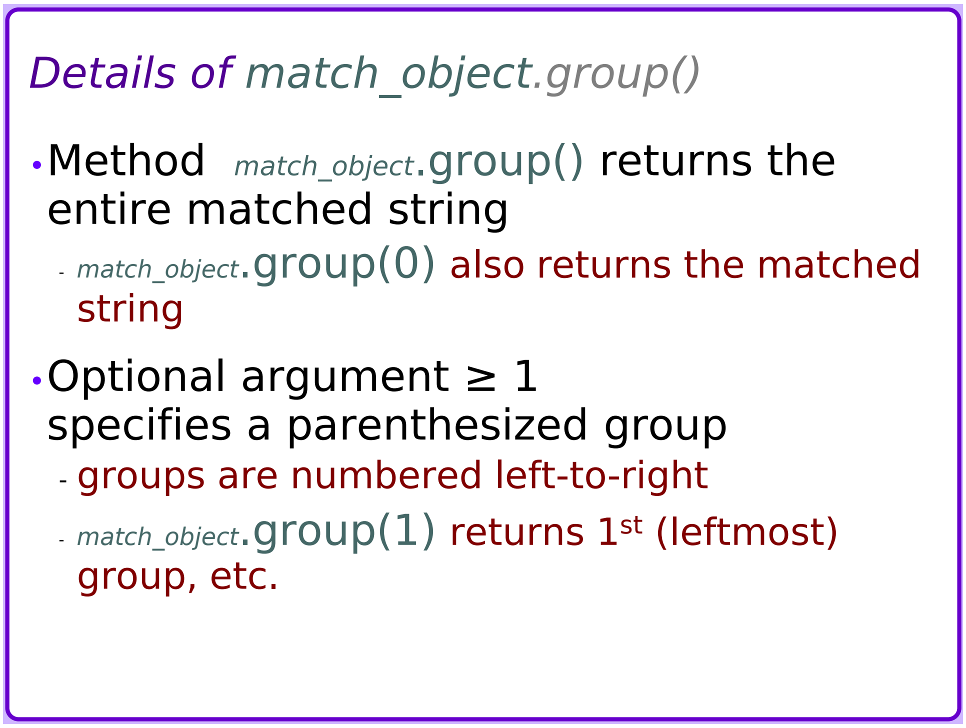
```
def get_name(pat=r'\b(\w+)\b.*\b(\w+)\b'):
    fullname = input('Enter your full name: ')
    m = re.match(pat, fullname)
    if not m == None:
        first = m.group(1)
        last = m.group(2)
        return (last, first)
    else:
        return None
#
```

## *exercise*

- Open file "montcs.bloomu.edu/Datasets/Logfiles/error.log.1"
- Search for lines containing an IP address
  - form is 4 groups of 1-3 digits, separated by a period
    - *e.g.* 123.145.167.189 , or 172.16.0.201, or 8.8.8.8 …
- Collect the matched IP addresses into a *list*, and also into a *set*
- When done, print the length of the list and the length of the set
  - Should be 1034 addresses in the list, and 230 in the set

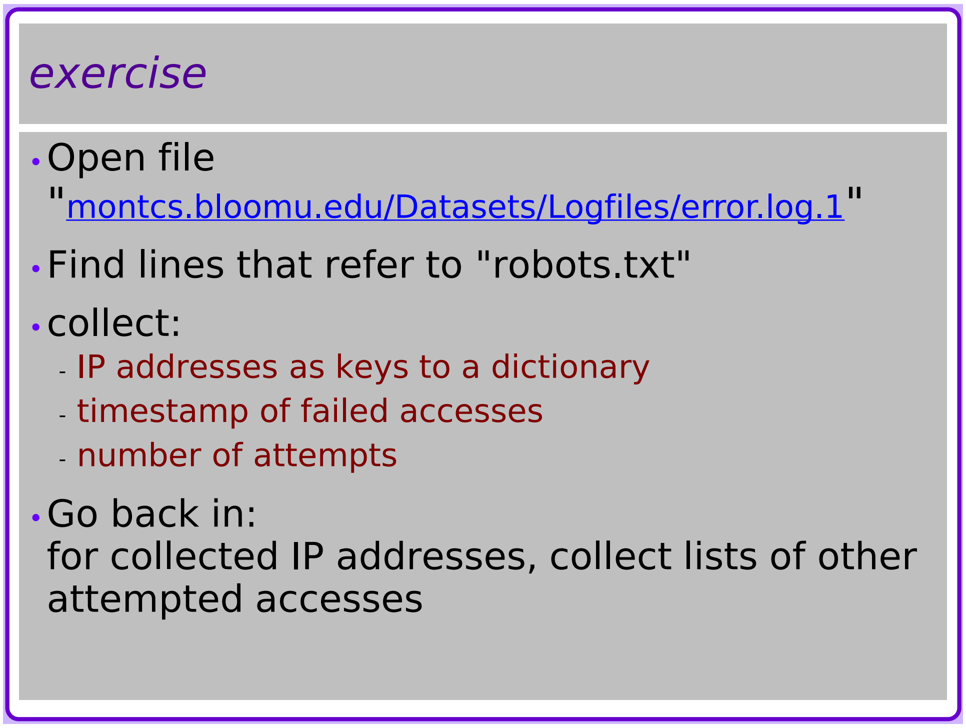
## *exercise part 2*

- Collect the matched IP addresses into a *dictionary*, whose values are the accessed files
- Should be 230 dictionary keys
- What IP tried to access to most files?



## *Details of match_object.group()*

- Method *match_object*.group() returns the entire matched string
  - *match_object*.group(0) also returns the matched string
- Optional argument ≥ 1 specifies a parenthesized group
  - groups are numbered left-to-right
  - *match_object*.group(1) returns 1st (leftmost) group, etc.

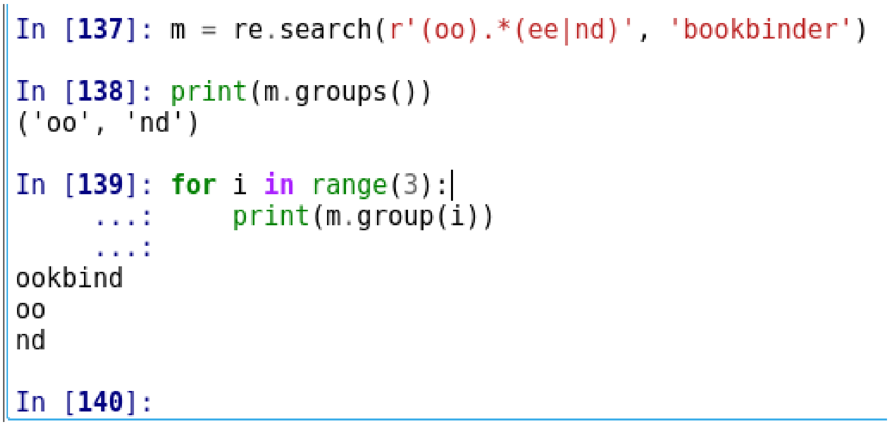


## *exercise*

- Open file "montcs.bloomu.edu/Datasets/Logfiles/error.log.1"
- Find lines that refer to "robots.txt"
- collect:
  - IP addresses as keys to a dictionary
  - timestamp of failed accesses
  - number of attempts
- Go back in:
  for collected IP addresses, collect lists of other attempted accesses

## *match_object.**groups()***

- Similar method *match_object*.groups() returns a tuple of all parenthesized groups
- example:



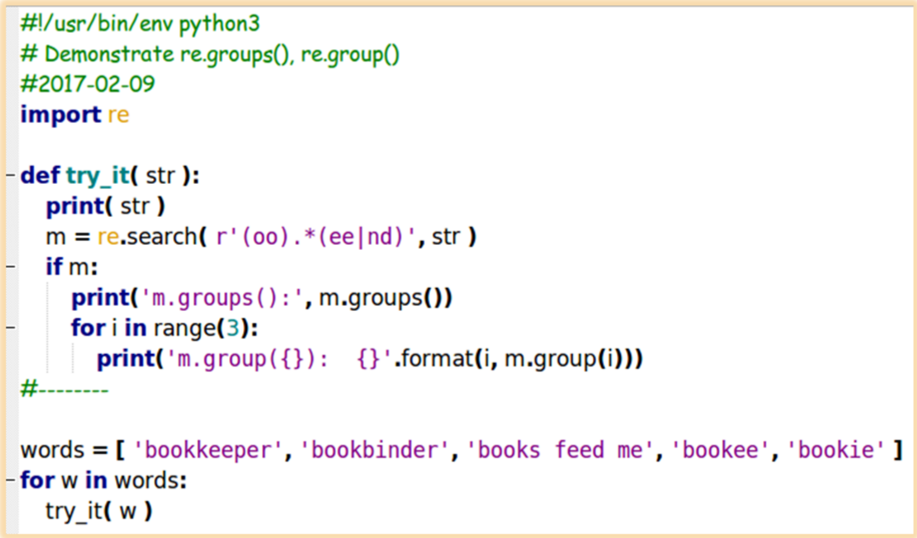
```
In [137]: m = re.search(r'(oo).*(ee|nd)', 'bookbinder')

In [138]: print(m.groups())
('oo', 'nd')

In [139]: for i in range(3):
     ...:     print(m.group(i))
     ...:
ookbind
oo
nd

In [140]:
```

## *.groups() Example, Elaborated*

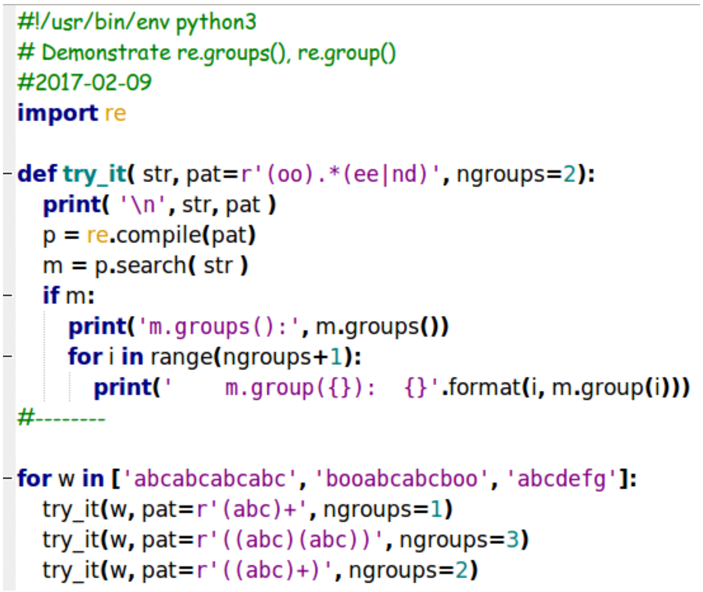


```
#!/usr/bin/env python3
# Demonstrate re.groups(), re.group()
#2017-02-09
import re

def try_it( str ):
    print( str )
    m = re.search( r'(oo).*(ee|nd)', str )
    if m:
        print('m.groups():', m.groups())
        for i in range(3):
            print('m.group({}):  {}'.format(i, m.group(i)))
#--------

words = [ 'bookkeeper', 'bookbinder', 'books feed me', 'bookee', 'bookie' ]
for w in words:
    try_it( w )
```

## What About Nested Parentheses?



```
#!/usr/bin/env python3
# Demonstrate re.groups(), re.group()
#2017-02-09
import re

def try_it( str, pat=r'(oo).*(ee|nd)', ngroups=2):
    print( '\n', str, pat )
    p = re.compile(pat)
    m = p.search( str )
    if m:
        print('m.groups():', m.groups())
        for i in range(ngroups+1):
            print('    m.group({}):  {}'.format(i, m.group(i)))
#--------

for w in ['abcabcabcabc', 'booabcabcboo', 'abcdefg']:
    try_it(w, pat=r'(abc)+', ngroups=1)
    try_it(w, pat=r'((abc)(abc))', ngroups=3)
    try_it(w, pat=r'((abc)+)', ngroups=2)
```

## Some More Details of Match Objects

| match object's member: | description |
|---|---|
| mo.start() | position (index) of beginning of match |
| mo.end() | position of end of match |
| m.span() | start & end positions of each group's match |
| | |
| mo.re | the regular expression that was used to make the match |
| mo.string | the original target string |
| mo.pos,<br>mo.endpos | starting & ending positions of search within the target string |

## *exercise*

- Open file "montcs.bloomu.edu/Datasets/Logfiles/error.log.1"
- Search for lines containing an IP address
  - form is 4 groups of 1-3 digits, separated by a period
    - *e.g.* 123.145.167.189 , or 172.16.0.201, or 8.8.8.8 …
- Collect the matched IP addresses into a *list*, and also into a *set*
- When done, print the length of the list and the length of the set
  - Should be 1034 addresses in the list, and 230 in the set

## *More Regex Functions*

- re.findall(), re.finditer() functions return all matches of a pattern within a string, as a list of match objects
- re.sub(), re.subn() functions substitute a replacement substring for the matched pattern in a target string
- re.split() splits a target string into substrings separated by the pattern
- re.compile() : precompile a regex for faster performance of repeated searches

## *re.findall()*

- The re.findall() function searches all non-overlapping occurrences of the provided pattern
  - Returns a list of all matches
- The re.finditer() function acts like re.findall(), but returns an *iterator* instead of a list of matches
  - An *iterator* is an object that provides each discovered occurrence of the pattern, one at a time – useful in "for" statements, etc.
  - Iterators provide more efficiency

## *re.sub()*

- The re.sub() function replaces pattern-matches in a target string with a replacment string
  - Returns a modified string
  - Replacement can be a string or a function
- The re.subn() function acts like re.sub(), returns the number of substitutions made as well as the modified string

## *re.compile()*

- The re.compile() function compiles a text string that represents a regular expression into a regex object
- Compiled regex objects include methods .search(), .match(), .findall()/.finditer(), .sub()/.subn()
- Regular expressions that are used repeatedly are more efficient if compiled once beforehand

## *exercise – anonymizing IP addresses*

- Open file "montcs.bloomu.edu/Datasets/Logfiles/error.log.1"
- Search for lines containing an IP address
  - form is 4 groups of 1-3 digits, separated by a period
    - *e.g.* 123.145.167.189 , or 172.16.0.201, or 8.8.8.8 ...
- Replace every IP address with "xxx.xxx.xxx.xxx"
- Write all lines to a new text file named "error.log.2"
  - Should be identical to the original file, except that all IP addresses have been anonymized