PYTHON Regular Expressions

Rev 4/25/2023

Data Links

All of the slides, demonstration programs and lab data can be found here in a zip file:

rebrand.ly/handouts

The results of each lab will be made available here:

rebrand.ly/lastlab

Instructor – Bill Eastridge

A lot of the information we will cover is available at the following web site:

billsexcellentprogramming.site

Bill's email address is:

bill@billsexcellentprogramming.site

Why Use Regular Expressions?

- Regular expressions (regexes or REs) are used to examine text data.
- A lot of text data we process is not in a uniform format. Some examples are log files, textual reports and web-site html.
- In the above cases, we need a way to peruse this textual data and determine if it contains items of interest.
- Regular Expressions implementations allow us to perform these types of operations.
- Most modern languages support some form of regular expressions.
- We will discuss a subset of Python's regex capability
- Just to make sure everyone is on the same page, what are the various categories of textual data?

Regular Expressions Topics

- Structure of the functions we will study
  - We will use **findall**, **finditer**, **search** and **match** to implement REs
  - We will not cover the **sub**, **split** and **compile** in any detail
- Simple RE patterns without metacharacters
- Patterns using metacharacters
  - Start with the simpler metacharacters and the basics of comparisons
  - How to control comparisons with flags
  - Move on to enclosures and the abbreviations (special sequences)
  - Start looking into groups and how they work
  - Specify the appropriate context of a potential match
  - Discuss the impact of greedy operators and lazy matches
  - Do several labs
- Conclude with a discussion of some of the items we have skipped.

Python RE Functions

- Functions from the re module we will cover
  - re.findall(pattern, string, flags=0) → returns a list
  - re.finditer(pattern, string, flags=0)
  - re.search(pattern, string, flags=0)
  - re.match(pattern, string, flags=0)

  } returns match objects or None
- What are match objects?
  - A match object contains:
    - the characters that were captured
    - the starting and ending positions of the characters
    - the methods **group**, **start**, **end** and **span**.
  - The **match** and **search** functions return at most one match object
  - The **finditer** function can return multiple match objects
- Note: The **match** function matches only at the beginning of the string.

Pattern Example

r'(?<!\d)\d{6}(?!\d)'

r'([$£¥])?(\d{1,3},\d{3}|\d{1,6})\.?(\d{1,2})?'

We should understand how to construct these patterns by the end of the session.

Regular Expressions (RE)

- The **re** module must be imported to use RE functions
- The pattern contains:
  - Ordinary characters such as letters, numbers, punctuation and white space for literal matches
  - Metacharacters which control how matching is done.
  - Simple examples without metacharacters:

    re.findall('ab', 'abracadabra') -> ['ab', 'ab']

    re.search('dab', 'abracadabra') → re.Match object; span=(6, 9), match='dab'

    re.finditer('ab', 'abracadabra') → <re.Match object; span=(0, 2), match='ab'>

    <re.Match object; span=(7, 9), match='ab'>

    re.findall('z', 'abracadabra') -> []

Regular Expressions (RE)

- The pattern also contains metacharacters which control how matching is done. → . * + ? ^ $ { } [ ] \ | ( )
- Let's start with discussing the first few → **. * + ? |**
- Basic rule - There are no overlapping matches.
- Simple examples with metacharacters

  re.findall('a.', 'abracadabra') -> ['ab', 'ac', 'ad', 'ab']
  re.findall('b.c', 'ab3cabcabdc8') -> ['b3c', 'bdc']
  re.findall('ab*', 'ab3cabbca3bdc8') -> ['ab', 'abb', 'a']
  re.findall('ab+', 'ab3cabbca3bdc8') -> ['ab', 'abb']
  re.findall('ab?', 'ab3cabbca3bdc8') -> ['ab', 'ab', 'a']
  re.findall('a|b', 'abracadabra') -> ['a', 'b', 'a', 'a', 'a', 'b', 'a']
  re.findall('b.a|ab*', 'abracadabra') -> ['ab', 'a', 'a', 'ab', 'a']

- You can have multiple pipe (|) metacharacters in a pattern.

The metacharacters *, + and ? are also called quantifiers.

Metacharacters and Their Meaning

\ Drop (or escape) the special meaning of character following it

^ Matches the beginning of the string

$ Matches the end of the string or just before the newline at the end of the string

. Matches any character except newline

? Matches zero or one occurrence of the previous RE.

* Any number of occurrences (including 0 occurrences)

+ One or more occurrences of the previous RE.

| Means OR (Matches with any of the characters separated by it.)

[] Represent a character class.

{} Indicate number of occurrences of a preceding RE to match.

() Enclose a group of REs or a context-testing extension.

Regular Expressions (RE)

- As a rule, you should use raw strings for regex patterns.
- [] - brackets - indicate a class of characters:

  Inside brackets, the logic is "or"
  Characters can be listed individually, e.g. [amk]
  Ranges of characters can be specified, e.g. [a-z] or [a-zA-Z]
  The ^ symbol means "not" in brackets, e.g. [^a-z] or [^a-zA-Z]

- Examples:

  re.findall(r'[cm8]', 'ab3cabbca3bdc8') -> ['c', 'c', 'c', '8']

  re.findall(r'[0-9]', 'ab3cabbca3bdc8') -> ['3', '3', '8']

  re.findall(r'[^a-z]', 'ab3caBBca3bdc8') -> ['3', 'B', 'B', '3', '8']

  re.findall(r'[^a-zA-Z]', 'ab3caBBca3bdc8') -> ['3', '3', '8']

  re.findall(r'[5-9][0-6]', '234567890123456') -> ['56', '90', '56']

  re.findall(r'b[^0-9]', 'ab3cabbca3bdc8') -> ?
  re.findall(r'[a-z]+', 'ab3cabbca3bdc8') -> ?

  re.findall(r'[a-zA-Z]+', 'I leave home early') -> ['I', 'leave', 'home', 'early']

  re.findall(r'[a-zA-Z]*', 'I leave home early') -> ?

Lab 01

Write a regex that will find all words in the following text that are contractions but not words that have apostrophes indicating missing leading or trailing letters. (See the lab01_starter.py in your data)

I've been thinkin' that we couldn't have done more in our effort to beat 'em.

Flags

Abbreviation	Full Name	Definition
re.I	re.IGNORECASE	Ignore case
re.M	re.MULTILINE	make begin/end {^, $} consider each newline. (Does not apply to match)
re.S	re.DOTALL	make dot match newline too.
re.X	re.VERBOSE	allow comment in regex.
re.A	re.ASCII	perform ASCII-only

Flag Options for the Previous Lab.

```
txt = ("I've been thinkin' that we couldn't have "
          "done more in our effort to beat 'em.")
lst = re.findall(r"[a-zA-Z]+'[a-zA-Z]+", txt)
print(lst)

#  An introduction to flags
lst = re.findall(r"[a-z]+'[a-z]+",
                  txt, re.IGNORECASE)  # or re.I
print(lst)

lst = re.findall(r"""
                 [a-z]+  # Find one or more ASCII letters
                 '       # Followed by an apostrophe
                 [a-z]+  # Followed by one or more ASCII letters
                 """, txt, re.IGNORECASE | re.VERBOSE)
             # or """, txt, flags=re.IGNORECASE | re.VERBOSE)
print(lst)
```

Regular Expressions (RE)

- {} - braces - indicate the number of characters expected:

  Inside braces, there can be one number or two

  One number requires that number of characters.

  e.g. [0-9]{5} - looks for 5 consecutive numbers

  Two numbers requires that range of characters.

  e.g. [a-z]{3,6} - looks for 3 to 6 consecutive lower-case letters

- Examples:

  re.findall(r'[a-z]{2}', 'a3bcaBBca3bdc8') -> ['bc', 'ca', 'bd']

  re.findall(r'[a-z]{2,4}', 'ab3caBBca3bdc8') -> ['ab', 'ca', 'ca', 'bdc']

  re.findall(r'[a-zA-Z]{2,4}', 'ab3caBBca3bdc8') -> ['ab', 'caBB', 'ca', 'bdc']

  re.findall(r'[0-9]{5}', '90210,78283-1337') -> ['90210', '78283']

  re.findall(r'[0-9]{5}-[0-9]{4}', '90210,78283-1337') -> ['78283-1337']

Regular Expressions (RE)

- Combined operations:

  re.findall(r'[0-9]{5}', '90210,78283-1337') -> ['90210', '78283']

  re.findall(r'[0-9]{5}-[0-9]{4}', '90210 78283-1337') -> ['78283-1337']

  re.findall(r'[0-9]{5}-[0-9]{4}|[0-9]{5}', '90210 78283-1337') ->
  ['90210', '78283-1337']

  re.findall(r'[0-9]{5}|[0-9]{5}-[0-9]{4}', '90210 78283-1337') ->
  ['90210', '78283'] **Note**

  x = "What!? Who told you that? I can't believe it!"

  re.findall(r'[a-zA-Z]+', x) ->
  ['What', 'Who', 'told', 'you', 'that', 'I', 'can', 't', 'believe', 'it']

  re.findall(r"[a-zA-Z']+", x) ->
  ['What', 'Who', 'told', 'you', 'that', 'I', "can't", 'believe', 'it']

- Why is the last example considered sloppy?

Regular Expressions (RE)

- Combined operations:

  x = "Serial Nums 27-473, 210-34 are recalled. 1-144, 222-5 are OK"

  - Extract the serial numbers from the above.
    The serial numbers are 1-3 digits both before and after a dash.

    ```
    re.findall(r'[0-9]{1,3}-[0-9]{1,3}', x)
    ```

  x = "phone 210-555-1234, cell 202.555.4321, fax 212 555 2314"

  - Extract the phone numbers with different separators.

    ```
    re.findall(r"[0-9]{3}.[0-9]{3}.[0-9]{4}", x)
    re.findall(r"[0-9]{3}\.[0-9]{3}\.[0-9]{4}", x)  # Note: the period is escaped
    re.findall(r'[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}', x)
    ```

  - Note - a dash inside brackets must be first or last in the list i.e., [.\-] or must be escaped. i.e., [.\-] or[. -] or as above

- What if the phone number is preceded by a 1 and a dash?
  - e.g., 1-800-555-5555

Lab 02 – IP Addresses

You downloaded a file called ipaddress.txt. Read this entire file into memory and then detect and collect all entries in the file that fit the format of an IPv4 address. A valid IPv4 address must be in the form of xxx.xxx.xxx.xxx, where xxx is a 1-to-3-digit number from 0-255.

Just check to make sure there are four segments of one to three digits where each segment is separated by a period. Do not try to verify with your regex whether the range of the number exceeds 255. While it's possible to create a regex that will do this, it is very complicated. Print your results. One of the addresses will be incorrect.

Regular Expressions – Special Sequences

- \d - Matches any decimal digit. The same as [0-9] (in ASCII)
  \D - Matches any non-digit. The same as [^0-9] (in ASCII)
- \w - Matches any word character. The same as [a-zA-Z0-9_] (in ASCII)
  \W - The same as [^a-zA-Z0-9_]. (in ASCII)
- \s - Matches whitespace characters (includes [\t\n\r\f\v]) (in ASCII)
  \S - The same as [^ \t\n\r\f\v] (in ASCII)
- Note: \w are also known as word characters while \W are non-word characters
- Examples:

```
re.findall(r'\d{5}', '90210,78283-1337') -> ['90210', '78283']

re.findall(r'\d{5}-\d{4}', '90210, 90210-1234') -> ['90210-1234']


x = "What!?  Who told you that? I can't believe it!"
re.findall(r'\w+', x) ->
                    ['What', 'Who', 'told', 'you', 'that', 'I', 'can', 't', 'believe', 'it']
re.findall(r"\w+'\w+|\w+", x) ->
                     ['What', 'Who', 'told', 'you', 'that', 'I', "can't", 'believe', 'it']
```

Regular Expressions - Special Sequences

- \w - Matches any word character. The same as [a-zA-Z0-9_] in ASCII.
  \W - The same as [^a-zA-Z0-9_] in ASCII.
  - What if I wanted to exclude the underscore from \w? → [^\W_]
    The same as [a-zA-Z0-9] in ASCII.
  - What if I wanted only alphabetic characters? → [^\W\d_]
    The same as [a-zA-Z] in ASCII.
  - What if I want just punctuation? There is no perfect answer.
    - [^\w\s] → includes all punctuation except the underscore
    - [^a-zA-Z\d\s] → includes all punctuation and nothing else but only in the ASCII world
  - It is important to understand the use of the ^ symbol in brackets

Lab 03

Change the results of the previous two labs to use the shorthand notation discussed on the previous slide.

Two Other Metacharacters

- We have not covered the ^ and $ metachracters.
- An example of their use is shown in the demo program beginend.py
- The ^ character forces a match to occur at the beginning of a string. The $ character forces a match at the end.
- The MULTILINE flag causes these metacharacters to treat a newline character as the end of one string and the start of another.

```
srvrs = ['OKCUC120\n', '37SQFSRU6\n', '22LONOLRZ\n',
         '345SQR-OC2\n', '1115BZQBO\n', 'ZC49OLR2\n']

for rec in srvrs:
    fnd = re.findall(r'^\d{2}\w+[\dO]$', rec)
    if fnd:
        print(fnd)
```

```
['37SQFSRU6']
['1115BZQBO']
```

Groups

- The last enclosure we will study is () which, among other things, creates groups with numbered indexes.
- In order to do a match, some of the requirements can be in groups and some can be outside of groups.
  - Both of these determine whether there is a match.
- Using **findall** with groups provides only the matches found with the groups. Matches outside of groups are not captured.
- The numbering of indexes applies only to **finditer**, **search** and **match.**
  - Only the **group()** or **group(0)** method gives you the matches outside of groups.

Groups

- The **findall** function with grouping still presents results in a list.
- Only those items discovered through patterns in parentheses are included in the result. Pattern results outside of parentheses are excluded.
- The results of two or more patterns in parentheses are presented in a tuple.

```
re.findall('(ab)', 'abracadabra')
['ab', 'ab'] #  Note: no difference
re.findall('(ab)(ra)', 'abracadabra')
[('ab', 'ra'), ('ab', 'ra')] #  Results presented in a tuple
re.findall('(ab)(ra).(ad)', 'abracadabra')
[('ab', 'ra', 'ad')] # One occurrence is found without the "c"
re.findall('(ab)(ra).(db)?', 'abracadabra')
[('ab', 'ra', '')]  # The optional item is still accounted for.
```

Groups

- The **search**, **match** and **finditer** functions still produce match objects.
- Each match object can have multiple group entries.
  - The **groups** method places all group results in a tuple just like **findall**
  - The first entry (**group(0)** or just **group()**) contains the entire matched string with all results either inside or outside of parentheses
  - The remaining entries (**group(1)** thru **group(N)**) contain the results from each set of parentheses.
  - Each match object has an attribute **lastindex** which contains the number of the last group entry.
  - Optional groups not found result in a **None** value.

```
s = re.search('(ab)(rc)?..(ca)', 'abracadabra')
print(s.lastindex)
3
print(s.groups())
('ab', None, 'ca')
print(s.group(0), s.group(1), s.group(2), s.group(3))
abraca ab None ca
```

Note: The groups method

0 1 2 3

Sample Problem

- What if I have a bunch of phone numbers in varying formats.
- I want to isolate the actual numbers in a consistent way.
- If something is missing, I want to make note of it.
- As you can see below, **findall** works, but doesn't give you access to the items outside of the groups.

```
ph = '+1-800-555-5555, 800.222.2222, 1800 333 3333, -800-444-4444'
```

```
print(re.findall('1?[-. ]?\d{3}[-. ]\d{3}[-. ]\d{4}', ph))
['1-800-555-5555', ' 800-222-2222', '1800-333-3333', '-800-444-4444']
pprint(re.findall('(1)?[-. ]?(\d{3})[-. ](\d{3})[-. ](\d{4})', ph))
[('1', '800', '555', '5555'),
 ('', '800', '222', '2222'),
 ('1', '800', '333', '3333'),
 ('', '800', '444', '4444')]
```

Groups with Numbered Indexes

- Grouping with **finditer**, **search** and **match** also provides an opportunity to get consistent results with added benefits.
- The example below uses the same phone numbers and RE pattern:

```
ph = '+1-800-555-5555, 800.222.2222, 1800 333 3333, -800-444-4444'
fnd = re.finditer('(1)?[-. ]?(\d{3})[-. ](\d{3})[-. ](\d{4})', ph)
for mat in fnd:
    print(mat.group(), 'Last Index -', mat.lastindex)
    print(mat.group(1), mat.group(2), mat.group(3), mat.group(4))
```

Note:
If the ? is placed inside the parens as in (1?), you get an empty string instead of None when the item is not found)

```
1-800-555-5555 Last Index - 4
1 800 555 5555
 800.222.2222 Last Index - 4
None 800 222 2222
1800 333 3333 Last Index - 4
1 800 333 3333
-800-444-4444 Last Index - 4
None 800 444 4444
```

Understanding Our Initial Example

r'([$£¥])?(\d{1,3},\d{3}|\d{1,6})\.?(\d{1,2})?'

```
cur = '$274.37  85.01  £1,225.88  ¥34567'
```

```
re.findall(r"""              # Use groups to capture desired data
           ([$£¥])?          # Find an optional currency symbol
           (\d{1,3},\d{3}    # Find 1 to 3 digits, a comma and 3 more digits
           |                 # or
           \d{1,6})          # find 1 to 6 digits with no comma
           \.?               # find an optional period
           (\d{1,2})?        # Find optional fractional currency
           """, cur, re.VERBOSE)
```

[('$', '274', '37'), ('', '85', '01'), ('£', '1,225', '88'), ('¥', '34567', '')]

- Is there an easier way to accommodate a comma in the number?
  r'([$£¥])?([\d,]{1,7})(\.\d{1,2})?' # a more disciplined way is better
- How much discipline do you want/need?

Lab 04

Read the ipaddress.txt file. Using both the **findall** and **finditer** functions, place each segment of each IP address found in a separate group. Do not include the separators in these groups, and do not exclude the erroneous address. Print the results as shown below. Neatness is not necessary.

FINDALL

```
[('192', '168', '1', '167'),
 ('172', '30', '128', '27'),
 ('192', '168', '1', '54'),
 ('192', '168', '1', '54'),
 ('172', '30', '128', '27'),
 ('192', '168', '1', '54'),
 ('192', '168', '1', '121'),
 ('90', '0', '0', '338'),
 ('172', '30', '128', '27')]
```

FINDITER

```
192 168   1 167, 192.168.1.167
172  30 128  27, 172.30.128.27
192 168   1  54, 192.168.1.54
192 168   1  54, 192.168.1.54
172  30 128  27, 172.30.128.27
192 168   1  54, 192.168.1.54
192 168   1 121, 192.168.1.121
 90   0   0 338, 90.0.0.338
172  30 128  27, 172.30.128.27
```

The **groups** method is not used on the **finditer** results here. Use groups on that result to verify the output is the same as with **findall**.

Extension Notation

- A '(' followed immediately by a '?' is considered extension notation. We will look at three of them.
- One form of extension notation applies names to group indexes. It is the only form of extension notations that creates a group.
- Another form we will study is the testing of context.
- The final form is called a non-capturing group.

Groups with Named Indexes

- Creating match objects gives the added benefit of being able to name each group element using extension notation (?P<name>...).

```
ph = '+1-800-555-5555, 800.222.2222, 1800 333 3333, -800-444-4444'
fnd = re.finditer("""
                  (?P<cntry>\d)?[-. ]?   # Extract country code
                  (?P<area>\d{3})[-. ]   # Extract area code
                  (?P<prefix>\d{3})[-. ] # Extract prefix
                  (?P<line>\d{4})        # Extract line number
                  """,ph, re.VERBOSE)
for mat in fnd:
    print(mat.group())  # or print(mat.group(0))
    print(mat.group('cntry'), mat.group('area'),
          mat.group('prefix'), mat.group('line'))
```

```
1-800-555-5555
1 800 555 5555
 800.222.2222
None 800 222 2222
1800 333 3333
1 800 333 3333
-800-444-4444
None 800 444 4444
```

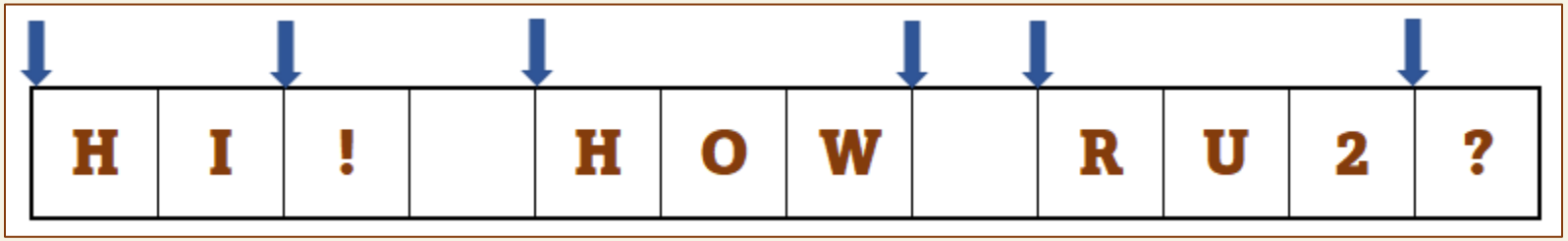
See the demo program grouping_demo.py

Context

- You have the ability to determine the appropriateness of the leading and trailing content of any potential match.
- These are called assertions. The simplest assertion is \b. It forces the match to fall on a word boundary (\w vs \W).
- The remaining four assertions are group extensions. They insist on a certain context, but they consume nothing. All extensions begin with (? as discussed earlier.
- These are referred to as lookahead and lookbehind assertions.
- Each one can be negative or positive
  - (?-- lookbehind pattern)Consuming pattern(?-- lookahead pattern) where the dashes are the characters that determine the assertion type.

Lookahead/Lookbehind Using /b

- \b - Word boundary. Boundary between a \w character and a \W character.
  1. Before the first character in the string if the first character is a word character (\w).
  2. Between two characters in the string if one character is a word character (\w) and the other is not (\W).
  3. After the last character in a string if the last character is a word character (\w)



Examples of \b

Simple Example

```
x = 'HI! HOW RU2?'
print(re.findall(r'\b[A-Z]+\b', x))
['HI', 'HOW']
print(re.findall(r'[A-Z]+', x))
['HI', 'HOW', 'RU']
```

More Complex Example with \b

```
y = 'User,(IP)-192.168.1.121,,GROUP,0:0:0:0:0:0,90.0.0.338,,'
pattern = r"""\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
               (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
               (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
               (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b"""
print(re.findall(pattern, y, re.VERBOSE))
[('192', '168', '1', '121')]
```

without \b

```
print(re.findall(pattern, y, re.VERBOSE))
[('192', '168', '1', '121'), ('90', '0', '0', '33')]
```

More Flexible Assertions

- There are four assertions corresponding to negative/positive lookahead/lookbehind. These use group extension notation.
- In the following definition, the … represents a pattern.
  - **(?=…)** – matches if … follows the potential match
  - **(?<=…)** – matches if … precedes the potential match
  - **(?!…)** – matches if … doesn't follow the potential match
  - **(?<!…)** – matches if … doesn't precede the potential match
- Lookahead assertions go the right of the capturing pattern.
- Lookbehind assertions go the left of the capturing pattern.
- (?-- lookbehind pattern)Consuming pattern(?-- lookahead pattern)
- As shown, both assertions are placed in parentheses.

More Flexible Assertions

- A lookahead assertion goes to the right of the capture pattern.
- A lookbehind assertion goes to the left of the capture pattern.
- Positive assertions:
  - **(?=...)** – matches if … follows the potential match
    re.findall('\w+(?=st)', 'cost run against') → ['co', 'again']
  - **(?<=...)** – matches if … precedes the potential match
    re.findall('(?<=\s)\w+', 'cost run against') → ['run', 'against']
- Negative assertions:
  - **(?!...)** – matches if … doesn't follow the potential match
    re.findall('[a-zA-Z]{2,5}(?![\d!])', 'Hi! How RU2?') → ['How']
  - **(?<!...)** – matches if … doesn't precede the potential match
    re.findall('(?<!\s)[a-zA-Z]{2,5}', 'Hi! How RU2?') → ['Hi', 'ow']

Elementary Examples

x = '1159486012539756033136578961237439760'

Find all 6's followed by a zero.
print(re.findall(r'6(?=0)', x)) → ['6', '6', '6']

Find all 6's not followed by a zero.
print(re.findall(r'6(?!0)', x)) → ['6', '6']

Find all 6's preceded by a 7 or 8.
print(re.findall(r'(?<=[78])6', x)) → ['6', '6']

Find all 6's preceded by a 1,2,3 or 4 and followed by a 5.
print(re.findall(r'(?<=[1-4])6(?=5)', x)) → ['6']

- The full program (context1.py) is in your downloads

More Complex Examples

xstr = "1023422 102376 123 434355 123456789 12bb23"

Find all non-overlapping, six-digit numbers
print(re.findall(r'\d{6}', xstr)) → ['102342', '102376', '434355', '123456']

Find six digits surrounded by whitespace
print(re.findall(r'(?<=\s)\d{6}(?=\s)', xstr)) → ['102376', '434355']

Find six digits with no following or leading digits
print(re.findall(r'(?<!\d)\d{6}(?!\d)', xstr)) → ['102376', '434355']

- The last pattern shown above is the same one we reviewed at the beginning of the session.
- The full program (context2.py) is in your downloads

Greed - Borrowed from Python Tutorial

The pattern is a[bcd]*b. This matches the letter 'a', zero or more letters from the class [bcd], and finally ends with a 'b'. Now imagine matching this RE against the string 'abcbd'.

Step	Matched	Explanation
1	a	The a in the RE matches.
2	abcbd	The engine matches [bcd]*, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match b, but the current position is at the end of the string, so it fails.
4	abcb	Back up, so that [bcd]* matches one less character.
5	*Failure*	Try b again, but the current position is at the last character, which is a 'd'.
6	abc	Back up again, so that [bcd]* is only matching bc.
6	abcb	Try b again. This time the character at the current position is 'b', so it succeeds.

Greedy Operators

- Formally, the greedy metacharacters are as follows: *, +, ?, {}
- These metacharacters/operators consume as much as possible.
- Sometimes, they cause you to get invalid results.
- Using the ? metacharacter (in a different way) can solve the problem.
- Examples from the demo program greedydemo.py

```
html = "<ul><li>The basics</li><li>Intercepting errors</li></ul>"
print(re.findall(r'<.*>', html))  # Greedy matching
['<ul><li>The basics</li><li>Intercepting errors</li></ul>']
print(re.findall(r'<.*?>', html)) # Lazy matching
['<ul>', '<li>', '</li>', '<li>', '</li>', '</ul>']

txt = 'The bananas are 25 cents and the apples are 46 cents'
print(re.findall('(bananas)[\w\s]+([0-9]{2,3})', txt))  # Greedy
[('bananas', '46')]
print(re.findall('(bananas)[\w\s]+?([0-9]{2,3})', txt)) # Lazy
[('bananas', '25')]
```

* In Python 3.11, there is a way to prevent backtracking using +

Other Regular Expressions Features

- re.split (in place of the string method of the same name)
  x = 'able-bodied--people-must---pitch-in'
  re.split(r'-+', x) -> ['able', 'bodied', 'people', 'must', 'pitch', 'in']
- re.sub (in place of the string method **replace**)
  x = 'able-bodied--people-must---pitch-in'
  re.sub(r'-+', ';' x) -> 'able;bodied;people;must;pitch;in'
- The compile function creates a regex object from a pattern and any optional flags. The object has access to methods that are the same name as the functions we have been using.

Other Regular Expressions Features

- Non-capturing groups – actually capture data, just not as a group. A (?: starts the group. A pattern follows with a) at the end.
- Non-capturing groups used in conjunction with regular groups will show only through group(0)
- Some examples: (from demo program non_cap_demo.py)

```
re.findall(r'\b\w+(?:st|in)\b',
                    'cost akin more run against')
['cost', 'akin', 'against']

re.findall(r'\d{5}(?:-\d{4})?',
                    '90210 90210-1234')
['90210', '90210-1234']

re.findall('\d{1,3}(?:\.\d{1,3}){3}',
             '192.168.0.1  192.168.1.12')
['192.168.0.1', '192.168.1.12']
```

Other Regular Expressions Features

- To avoid confusion, use raw strings when using regular expressions. Often, there isn't a need. It's just good to play it safe.
- A useful tutorial for all of Python's regular expressions can be found at https://docs.python.org/3/howto/regex.html.
  This tutorial covers a great deal more than our short course presented here.

The End