PATTERN MATCHING WITH REGULAR EXPRESSIONS

CS 3080: Python Programming

University of Colorado
Colorado Springs

Regular expressions

- You may be familiar with searching for text by pressing ctrl-F and typing in the words you’re looking for.
- Regular expressions go one step further: They **allow you to specify a pattern of text to search for.**

“Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you’re a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through.”

- Cory Doctorow

Time to code: Finding Patterns of Text Without Regular Expressions

- Say you want to find a phone number in a string. You know the pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers.

- Here’s an example: 415-555-4242.

- Let’s write a function isPhoneNumber() to check whether a string matches this pattern, returning either True or False

Time to code: Finding Patterns of Text Without Regular Expressions

- To check if there is a phone number in a large string:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my
office.'
for i in range(len(message)):
    chunk = message[i:i+12]
    if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)

print('Done')
```

Finding Patterns of Text with Regular Expressions

- What if the number is 415.555.4242, (415) 555-4242, 415-555-4242 x99? The isPhoneNumber() function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.

Finding Patterns of Text with Regular Expressions

- What if the number is 415.555.4242, (415) 555-4242, 415-555-4242 x99? The isPhoneNumber() function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.
- **Regular expressions**, called *regexes* for short, are descriptions for a pattern of text.
- For example, a \d in a regex stands for a digit character (any single numeral 0 to 9). So, the isPhoneNumber() function as a regex could be as:
  - ***\d\d\d-\d\d\d-\d\d\d\d***
- Or even more sophisticated:
  - *\d{3}-\d{3}-\d{4}*

Finding Patterns of Text with Regular Expressions

- What if the number is 415.555.4242, (415) 555-4242, 415-555-4242 x99? The isPhoneNumber() function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.
- **Regular expressions**, called *regexes* for short, are descriptions for a pattern of text.
- For example, a \d in a regex stands for a digit character (any single numeral 0 to 9). So, the isPhoneNumber() function as a regex could be as:
  - ***\d\d\d-\d\d\d-\d\d\d\d***
- Or even more sophisticated:
  - *\d{3}-\d{3}-\d{4}*

Complexity of a regex is O(n) where n is the length of the text to search.

Creating a regex object

(1) Pass desired pattern to re.compile(), and store the resulting Regex object in phoneNumRegex

```
import re

phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d’)


# By putting an r before the first quote of the string value,
# you can mark the string as a raw string, which does not escape
# characters. Typing r'\d\d\d-\d\d\d-\d\d\d\d' is much easier
# than typing '\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d'.
```

Matching regex objects

(2) Then call search() on phoneNumRegex, and pass search() the string we want to search. The result of the search is stored in mo

```
import re

phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d’)
mo = phoneNumRegex.search('My number is 415-555-4242.’)
print('Phone number found: ' + mo.group())
# Phone number found: 415-555-4242
```

(3) Call group() on mo to return the match. mo.group() displays the whole match

Matching regex objects

```
import re

phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d’)
mo = phoneNumRegex.search('My number is 415-555-4242.’)
print('Phone number found: ' + mo.group())
# Phone number found: 415-555-4242
```

- The **mo** variable name is just a generic name to use for Match objects.
- If .search() doesn’t find any coincidence, then the mo object is None.

Review of Regular Expression Matching

- While there are several steps to using regular expressions in Python, each step is fairly simple:
  1. *Import the regex module with import re.*
  2. *Create a Regex object with the re.compile() function. (Remember to use a raw string.)*
  3. *Pass the string you want to search into the Regex object's search() method. This returns a Match object or None if nothing is found.*
  4. *Call the Match object's group() method to return a string of the actual matched text.*

Grouping with Parentheses

- Adding **parentheses** will create groups in the regex: (\d\d\d)-(\d\d\d-\d\d\d\d).

```
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)’)
mo = phoneNumRegex.search('My number is 415-555-4242.’)
mo.group(1)        # '415’
mo.group(2)        # '555-4242’
mo.group(0)        # '415-555-4242’
mo.group()         # '415-555-4242’
mo.groups()        # ('415', '555-4242’)
areaCode, mainNumber = mo.groups()
print(areaCode)    # 415
print(mainNumber) # 555-4242
```

Grouping with Parenthesis

- Parentheses have a special meaning in regular expressions, but what do you do if you need to match a parenthesis in your text?
  - *For example, area code in parenthesis*

```
phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)’)
```

- The `\(` and `\)` escape characters in the raw string passed to re.compile() will match actual parenthesis characters.

Matching Multiple Groups with the Pipe

- The | character is called a pipe. You can use it anywhere you want to match one of many expressions.
  - *For example, the regular expression `r'Batman|Tina Fey’` will match either 'Batman' or 'Tina Fey’.*
  - *When both Batman and Tina Fey occur in the searched string, **the first occurrence of matching text** will be returned as the Match object.*

```
heroRegex = re.compile (r'Batman|Tina Fey’)
mo = heroRegex.search('Batman and Tina Fey.’)
mo.group()  # 'Batman'
```

Matching Multiple Groups with the Pipe

- You can also use the pipe to match one of several patterns as part of your regex.

```
batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
mo = batRegex.search('Batmobile lost a wheel’)
mo.group()  # 'Batmobile’
mo.group(1) # 'mobile’
```

mo.group() returns full matched text 'Batmobile’, while mo.group(1) returns just part of the matched text inside the first parentheses group

Matching Multiple Groups with the Pipe

- You can also use the pipe to match one of several patterns as part of your regex.

```
batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
mo = batRegex.search('Batmobile lost a wheel’)
mo.group()  # 'Batmobile'
mo.group(1) # 'mobile'
```

- *If you need to match an actual pipe character, escape it with a backslash, like* `\|`*. The same happens with all the following characters in the next slides.*

Optional Matching with the Question Mark

- The ? character flags the group that precedes it as an optional part of the pattern.

```
batRegex = re.compile(r'Bat(wo)?man’)
mo1 = batRegex.search('The Adventures of Batman’)
mo1.group() # 'Batman’

mo2 = batRegex.search('The Adventures of Batwoman’)
mo2.group() # 'Batwoman'
```

Optional Matching with the Question Mark

- The ? character flags the group that precedes it as an optional part of the pattern.

```
phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d’)
mo1 = phoneRegex.search('My number is 415-555-4242’)
mo1.group() # '415-555-4242’

mo2 = phoneRegex.search('My number is 555-4242’)
mo2.group() # '555-4242'
```

Matching Zero or More with the Star

- The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again.

```
batRegex = re.compile(r'Bat(wo)*man’)

mo1 = batRegex.search('The Adventures of Batman’)
mo1.group()  # 'Batman’

mo3 = batRegex.search('The Adventures of Batwowowowoman’)
mo3.group()  # 'Batwowowowoman'
```

Matching One or More with the Plus

```
batRegex = re.compile(r'Bat(wo)+man’)
mo1 = batRegex.search('The Adventures of Batwoman’)
mo1.group() # 'Batwoman’

mo2 = batRegex.search('The Adventures of Batwowowowoman’)
mo2.group() # 'Batwowowowoman’

mo3 = batRegex.search('The Adventures of Batman’)
print(mo3 == None) # True
```

Matching Specific Repetitions with Curly Brackets

- If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets.
  - *For example, the regex `(Ha){3}` will match the string `'HaHaHa'`, but it will not match `'HaHa'`*

- These two regular expressions match identical patterns:
  - *`(Ha){3}`*
  - *`(Ha)(Ha)(Ha)`*

- But (Ha){3} returns only one group

```
print(mo.group(0)) # HaHaHa
print(mo.group(1)) # Ha
print(mo.group(2)) # IndexError: no such group
```

Matching Specific Repetitions with Curly Brackets

- Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets.
  - *For example, these two regular expressions match identical patterns:*
  - *`(Ha){3,5}`*
  - *`((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))`*

- You can also leave out the first or second number in the curly brackets to leave the minimum or maximum unbounded. For example,
  - *`(Ha){3,}` will match three or more instances of the `(Ha)` group,*
  - *`(Ha){,5}` will match zero to five instances*

Greedy and Nongreedy Matching

- Python’s regular expressions are **greedy** by default, which means that in ambiguous situations they will match **the longest string possible**.

```
greedyHaRegex = re.compile(r'(Ha){3,5}’)
mo1 = greedyHaRegex.search('HaHaHaHaHa’)
mo1.group() # 'HaHaHaHaHa’, even if it could also be ‘HaHaHa’ or
‘HaHaHaHa’
```

Greedy and Nongreedy Matching

- The **nongreedy** version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

```
nongreedyHaRegex = re.compile(r'(Ha){3,5}?’)
mo2 = nongreedyHaRegex.search('HaHaHaHaHa’)
mo2.group() # 'HaHaHa'
```

findall() method

- While search() will return a Match object of the first matched text in the searched string, the **findall()** method will return the strings of every match in the searched string.

```
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000’)
# ['415-555-9999', '212-555-0000’]

phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000’)
# [('415', '555', '1122'), ('212', '555', '0000’)]

phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)’)
print(phoneNumRegex.findall('No phone.’)) # []
```

Character classes

- The character class **\d** is shorthand for the regular expression **(0|1|2|3|4|5|6|7|8|9).**

- The character class **[0-5]** will match only the numbers 0 to 5; this is much shorter than typing **(0|1|2|3|4|5).**

Character classes

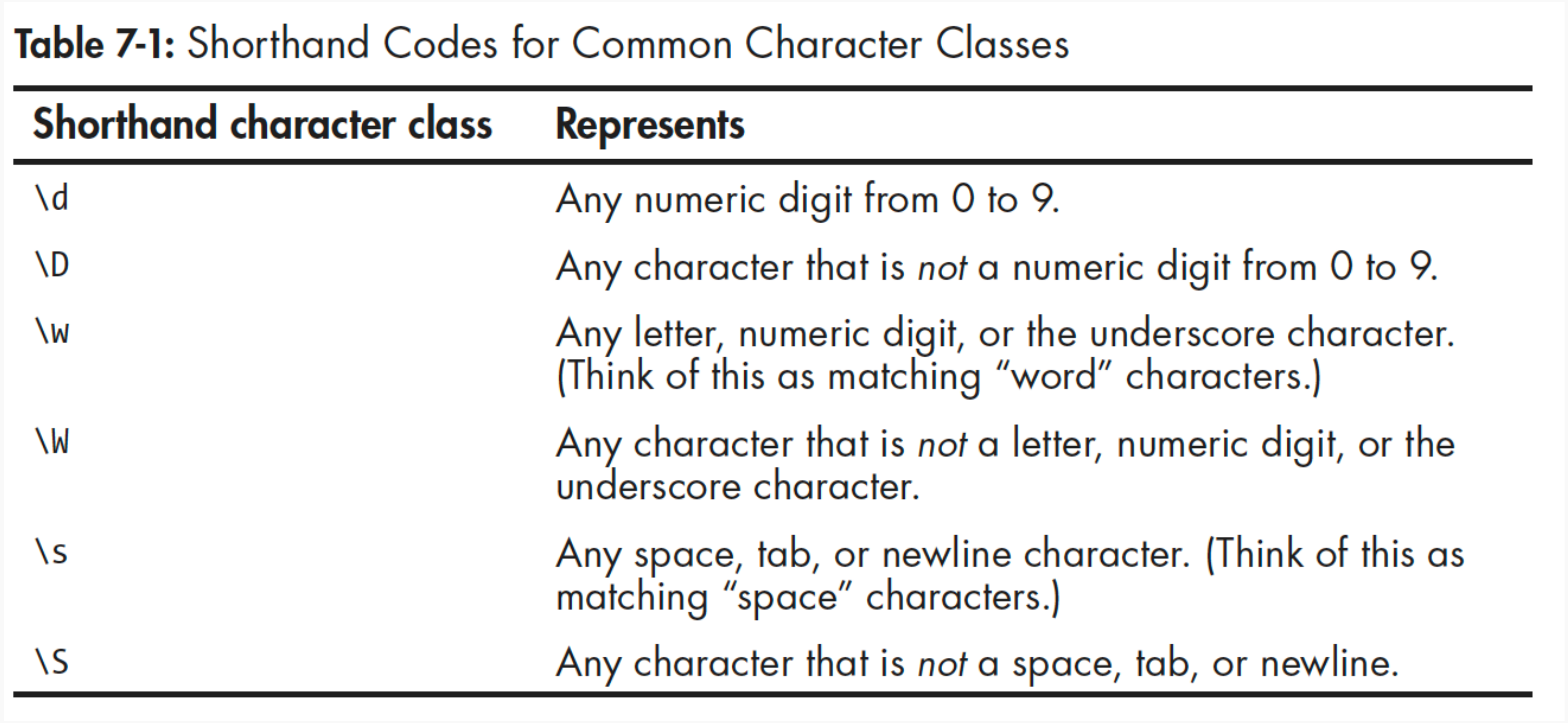

Table 7-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents
`\d`	Any numeric digit from 0 to 9.
`\D`	Any character that is *not* a numeric digit from 0 to 9.
`\w`	Any letter, numeric digit, or the underscore character. (Think of this as matching "word" characters.)
`\W`	Any character that is *not* a letter, numeric digit, or the underscore character.
`\s`	Any space, tab, or newline character. (Think of this as matching "space" characters.)
`\S`	Any character that is *not* a space, tab, or newline.

Character class example

```
xmasRegex = re.compile(r'\d+\s\w+’)
xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, ‘
                  '8 maids, 7 swans, 6 geese, 5 rings, 4 birds, ‘
                  '3 hens, 2 doves, 1 partridge’)
# ['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids',#
'7 swans', '6 geese', '5 rings', '4 birds', '3 hens', '2 doves',#
'1 partridge']
```

Making Your Own Character Classes

- There are times when you want to match a set of characters but the shorthand character classes (\d, \w, \s, and so on) are too broad. So you can create your own:
  - *[aeiouAEIOU], will match any vowel, both lowercase and uppercase*
  - *[a-zA-Z0-9], will match all lowercase letters, uppercase letters, and numbers*
  - *[a-z], will match all lowercase letters*
  - *etc*

Making Your Own Character Classes

- **Note that inside the square brackets, the normal regular expression symbols are not interpreted as such.**
- This means you **do not need to escape** the ., *, ?, or () characters with a preceding backslash. For example,
  - *[0-5.] will match digits 0 to 5 and a period.*
  - *You do not need to write it as [0-5\.]. This will match digits 0 to 5 and the period.*
  - *If you want to match the backslash too, you have to scape it: [0-5\\.]*

Making Your Own Character Classes

- By placing a **caret character (^) just after the character class's opening bracket**, you can make a **negative character class**. A negative character class will match all the characters that are not in the character class.

```
consonantRegex = re.compile(r'[^aeiouAEIOU]’)
consonantRegex.findall('RoboCop eats baby food. BABY FOOD.’)
# ['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y’, ‘ ',
'f', 'd', '.', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

The Caret and Dollar Sign Characters

- You can also use the **caret symbol (^) at the start of a regex** to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put **a dollar sign ($) at the end of the regex** to indicate the string must end with this regex pattern.
- And you can use the **^ and $ together** to indicate that the entire string must match the regex.

The Wildcard Character

- The . (or dot) character in a regular expression is called a **wildcard** and will match **any character** except for a newline.

```
atRegex = re.compile(r'.at’)
atRegex.findall('The cat in the hat sat on the flat mat. Somewhat!’)
# ['cat', 'hat', 'sat', 'lat', 'mat', 'hat’]
```

- And to include the newline character too:

```
newlineRegex = re.compile(r’.at', re.DOTALL)
```

Matching Everything with Dot-Star

```
nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)’)
mo = nameRegex.search('First Name: Yanyan Last Name: Zhuang_75+’)
mo.group(1) # ’Yanyan’
mo.group(2) # ’Zhuang_75+'
```

Matching Everything with Dot-Star

- The dot-star uses greedy mode: It will always try to match as much text as possible.
- To match any and all text in a nongreedy fashion, use the dot, star, and question mark (.*?).
- Like with curly brackets, the question mark tells Python to match in a nongreedy way.

Review of Regex Symbols 1

- The ? matches zero or one of the preceding group.
- The * matches zero or more of the preceding group.
- The + matches one or more of the preceding group.
- The {n} matches exactly n of the preceding group.
- The {n,} matches n or more of the preceding group.
- The {,m} matches 0 to m of the preceding group.
- The {n,m} matches at least n and at most m of the preceding group.
- {n,m}? or *? or +? performs a nongreedy match of the preceding group.

Review of Regex Symbols 2

- ^spam means the string must begin with spam.
- spam$ means the string must end with spam.
- The . matches any character, except newline characters.
- \d, \w, and \s match a digit, letter, or space character, respectively.
- \D, \W, and \S match anything except a digit, letter, or space character, respectively.
- [abc] matches any character between the brackets (such as a, b, or c).
- [^abc] matches any character that isn’t between the brackets.

Case-Insensitive Matching

```
robocop = re.compile(r'robocop', re.I) # Or re.IGNORECASE
robocop.search('RoboCop is part man and part machine').group()
# 'RoboCop’

robocop.search('ROBOCOP protects the innocent.').group()
# 'ROBOCOP’

robocop.search(‘It is robocop?').group()
# 'robocop'
```

Substituting Strings with the sub() Method

```
namesRegex = re.compile(r'Agent \w+’)
namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents’
                           ' to Agent Bob.')
# 'CENSORED gave the secret documents to CENSORED.'
```

Substituting Strings with the sub() Method

- In the first argument to sub() , you can type \1 , \2 , \3 , and so on, to mean “Enter the text of group 1 , 2 , 3 , and so on, in the substitution.”

```
agentNamesRegex = re.compile(r'Agent (\w)\w*’)
agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that'
                                ' Agent Eve knew Agent Bob was a ‘
                                'double agent.’)
# A**** told C**** that E**** knew B**** was a double agent.'
```

Managing Complex Regexes

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}
(\s|-|\.)\d{4}(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

Managing Complex Regexes

- Use the re.VERBOSE and triple-quote syntax

```
phoneRegex = re.compile(r'''(
(\d{3}|\(\d{3}\))?                # area code
(\s|-|\.)?                        # separator
\d{3}                             # first 3 digits
(\s|-|\.)                         # separator
\d{4}                             # last 4 digits
(\s*(ext|x|ext.)\s*\d{2,5})?      # extension
)''', re.VERBOSE)
```

Combining re.IGNORECASE, re.DOTALL , and re.VERBOSE

```
someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```