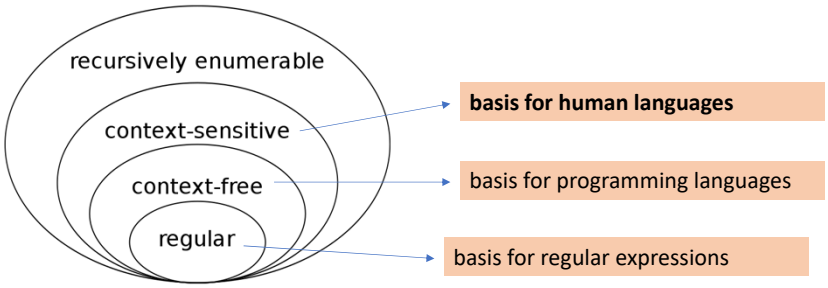
Regular Expressions (Regexes)

Overview

- What are regular expressions (regexes)
- Regex Grammar
  - character classes [..]
  - metacharacters \ $ ^ . ?
  - quantifiers: * + ?
- Python and Regexes
  - re.search
  - re.findall
  - re.sub
- Regex Fun

Regular Expressions (Regexes)

- A technology for parsing text and finding complex patterns using a simple 'regular' language
- In the 1950's, the linguist Noam Chomsky came up with a a categorization for languages of increasing complexity based on their grammar rules



Regexes go way beyond simple string matching

The `find()` method returns the index of first occurrence of the substring (if found). If not found, it returns **-1**.

Example

```
message = 'Python is a fun programming language'

# check the index of 'fun'
print(message.find('fun'))

# Output: 12
```

Regexes can do MORE!

Regex

- A *regex* defines a search pattern for strings.
- The search pattern can be anything from a simple character, a fixed string or a complex expression containing special characters describing the pattern.
- The pattern defined by the regex may match one or several times or not at all for a given string.
- Regular expressions can be used to search, edit and manipulate text.

Character Classes

- Defined by characters inside square brackets

Character Class Examples

```
[Pp]ython       Matches "Python" or "python"
rub[ye]         Matches "ruby" or "rube"
x[aeiou]z       Matches xaz xez xiz xoz xuz
[0-9]           Match any digit; same as [0123456789]
[a-z]           Match any lowercase ASCII letter
[a-zA-Z0-9]     Match any upper or lowercase ASCII letter or digit
```

http://www.regexpal.com

Characters and the character class

Regex Basics	
Regex	Usage
a	Matches the character 'a'
abc	Matches 'abc'
[abc]	Matches 'a', 'b', or 'c'
[^abc]	Matches any character except 'a', 'b', and 'c'
abc|def	Matches 'abc' or 'def'

Metacharacters

Metacharacters – characters with special meaning

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"world$"
*	Zero or more occurrences	"aix*"
+	One or more occurrences	"aix+"
{}	Exactly the specified number of occurrences	"al{2}"
|	Either or	"falls|stays"
()	Capture and group	

Special Sequences with \

\d	Returns a match where the string contains digits (numbers from 0-9)	
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	
\s	Returns a match where the string contains a white space character	
\D	Returns a match where the string DOES NOT contain digits	
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"

more examples..

Regex Character Classes	
Regex	Usage
\d	Matches any digit
\D	Matches any non-digit
\w	Matches any alphanumeric character (incl. the underscore '_' character)
\W	Matches any non-alphanumeric character
\s	Matches any whitespace character
\S	Matches any non-whitespace character
.	Matches any character
[a-z]	Matches any lowercase character from 'a' to 'z'
[A-Z]	Matches any uppercase character from 'A' to 'Z'
[0-9]	Matches any digit from 0 to 9, equivalent with \d

Quantifiers

Regex Quantifiers	
Regex	Usage
*	Matches 0 or more times
+	Matches 1 or more times
?	Matches 0 or 1 time
{3}	Matches exactly 3 times
{3,6}	Matches 3 to 6 times
{3,}	Matches 3 or more times
{,6}	Matches up to 6 times

Python and Regexes

Python Regex

```
import re   # import the regex library

txt = "Rollo scored 99 on his midterm
exam"
regex = r"\d+"
match = re.search(regex, txt)
if match:
      print ("Found a match")
```

Python uses "\" to escape characters in a Python strings (e.g. \n \t)

Use the 'r' prefix to specify a raw string (not a Python string) so that '\' does not get interpreted as a Python escape character.

see: w3 schools tutorial;
https://www.w3schools.com/python/

Regex Functions

- re.search

```
# explore re.search
mystr = "take CS7320 if you are interested in AI."
regex = '[A-Z]+\d+'
match = re.search(regex, mystr)
print (match)
print (match.group())
```

```
<re.Match object; span=(5, 11), match='CS7320'>
CS7320
```

- re.findall

```
# re.findall
mystr = "the pie cost $10.29 and the coffee cost $9.99"
# find all money references
regex = '\$\d+\.\d{2}'
mylist = re.findall(regex, mystr)
mylist

```

```
['$10.29', '$9.99']
```

- re.sub

```
#re.sub
mystr = "the pie cost $10.29 and the coffee cost $9.99"
regex = '\$\d+\.\d{2}'
new_str = re.sub(regex, '****', mystr)
new_str
```

```
'the pie cost **** and the coffee cost ****'
```

Grouping

- We often want to extract pieces of a larger phrase

```
txt = "News reports said that Rollo sold watermelons to Ralph"
```

- We want to find out who sold what to whom.

```
regex = r"\w+ sold \w+ to \w+"
```

- Put parens (..) around parts of the regex we want

```
regex = r"(\w+) sold (\w+) to (\w+)"
match = re.search(txt, regex)
print (match.group(1), match.group(2),match.group(3) )
```

Grouping and Back References (\1 \2 \3 ...)

- Often when you identify a group, you may want to see if that group value is repeated in the text
- We can use \1 to match the first group, \2 to match second group ..
- Example: we want to if the **same person** bought and then sold the **same stock**.

```
s = "The news said Rollo bought NVDA and then Rollo sold NVDA the next day."
regex = r"(\w+) bought (\w+).* \1 sold \2"
match = re.search(regex, s)
print (match.group(1), match.group(2) )
>> Rollo NVDA
```

Greedy vs. Lazy Matching

Greedy vs Lazy

When you text matching you will run into the needed for greedy/lazy evaluation. The regex default is greedy.

Greediness: Greedy quantifiers first tries to repeat the token as many times as possible, and gradually gives up matches as the engine backtracks to find an overall match.

Laziness: Lazy quantifier first repeats the token as few times as required, and gradually expands the match as the engine backtracks through the regex to find an overall match.

You mainly need to think about greedy/lazy when using the dot '.' to match **anything**. Usually it is the lazy version you want. Add **?** as in: : **.*?**

Example:

Test string: stackoverflow

Greedy regex: s.*o matches **stackoverflo**

Lazy regex: s.*?o matches: **stacko**

Summary

- Regexes are based on Regular Languages (from Chomsky)
- Used to identify complex strings in text
- They are FAST!
- Character classes, Metacharacters, Quantifiers
- Greedy vs Lazy Matching
- Python functions
  - re.search
  - re.findall
  - re.sub