AMS 595/691: Fundamentals of Computing: Part II

Lecture 4: Text File I/O and Regular Expressions

Xiangmin Jiao

Stony Brook University

Outline

1. Text File I/O

2. Regular Expressions

Print Method and Formatting

- `print()` method adds space between arguments and newline at end
- Easy adjustment using `str.ljust(`*n*`)` or `str.rjust(`*n*`)`
- Fancier output with `str.format()`
  - Use {}, {index}, or {keyword} to denote argument
  - Optional ':' and format specifier can follow index or keyword
- To prompt user to input data in command line, use `input('msg')`
- See Jupyter notebook on I/O

Read and Write Text Files

- Open file with `open(filename, mode)`, where mode can be 'r' for read, 'w' for write, and 'a' for append
- `file.write(str)` writes a string into file (no newline added)
- `str = file.read()` reads the whole file into string
- `file.readline()` reads single line
- Use `for` statement to loop through lines (instead of `file.readline()`)
- Must use `file.close()` when done reading/writing
- Alternatively, use `with` statement to close file automatically
- See Jupyter notebook on I/O

Outline

1. Text File I/O

2. Regular Expressions

Python Regular Expressions

- Regular expressions are a powerful language for matching text patterns
- Python "re" module provides regular expression support
- Use pattern to search and replace
  - `match = re.search(pat, str)` finds one match
  - `matches = re.findall(pat, str)` find all matches
  - `re.sub(pat, replacement, str)` substitute all matches with replacement string
- See Jupyter notebook on regular expressions

Basic Patterns

- ordinary characters match themselves, except for meta-characters . ^ $ * + ? { [ ] \ | ( )
- . (a period): matches any single character except newline '\n'
- \w: matches a letter or digit or underbar [a-zA-Z0-9_]; \W matches any non-word character
- \b: boundary between word and non-word
- \s: a single whitespace character ([\n\r\t\f]); \S matches any non-whitespace character
- \t, \n, \r: tab, newline, return
- \d: decimal digit [0-9]
- \: escape character
- ^, $: match the start or end of the string, respectively

Repetition

- $+$: 1 or more occurrences of the pattern to its left, e.g. 'i+' $=$ one or more i's
- *: 0 or more occurrences of the pattern to its left
- ?: match 0 or 1 occurrences of the pattern to its left
- $\{m\}$: m repetitions of the pattern to its left
- $\{m,n\}$: m to n repetitions of the pattern to its left

Square Brackets

- Square brackets can be used to indicate a set of chars, so [abc] matches 'a' or 'b' or 'c'.
- \w, \s etc. work inside square brackets too, except that dot (.) just means a literal dot
- Use a dash to indicate a range, so [a-z] matches all lowercase letters (unless dash is last)
- An up-hat (^) at the start of a square-bracket set inverts it, so [^ab] means any char except 'a' or 'b'.

Rules of Regular Expression Search

- The search proceeds through the string from start to end, stopping at the first match found
- All of the pattern must be matched, but not all of the string
- If `match = re.search(pat, str)` is successful, match is not None and in particular match.group() is the matching text
- Search tries to use up as much of the string as possible -- i.e. + and * are greedy

Group Extraction

- The “group” feature allows you to pick out parts of matching text
  - (. . .): captures group
  - (. . . (. . .)) captures sub-group
  - (abc|def) captures matches abc or def
- To define a group, add add parenthesis ( ) around a sub-pattern
  - e.g., r'([\w.-]+)@([\w.-]+)' defines two groups
- On successful search, match.group(1) is matched text corresponding to 1st group, and match.group(2) is text corresponding to 2nd group, etc. (i.e., indices of groups are 1-based instead of 0-based)
- Plain match.group() is still the whole matched text
- Matched groups can be referenced by \1, \2 etc. in replacement string