ECE 364
Software Engineering Tools Laboratory

Lecture 6

Python: Regular Expressions

Lecture Summary

- Regular Expressions in Python

Python Regular Expressions

- Python provides very sophisticated regular expression functionality
    - Searching and matching
        - Ability to specify named “groups” to get values of matched strings
    - Substitution/Find-Replace
    - Pattern Splitting

Regular Expressions (2)

- The most basic regex is just the characters of a string we want to match

- A very simple regex: `blue`
  - Will match the string “blue”

- Another very simple regex: `1a2b3c`
  - Will match the string “1a2b3c”

Regular Expressions (3)

- Boolean “or”: Choose between 2 or more alternatives
  - A vertical bar (pipe) “|” indicates an “or”
- Example: `red|green|blue`
  - Will match “red” or “green” or “blue”

Regular Expressions (4)

- Grouping: Assign precedence and scope to regular expressions
  - Use parenthesis “`(<regex>)`” to form a group

- Example `can|r|t`
  - Matches string “can” or “r” or “t”

- Example: `ca(n|r|t)`
  - Matches strings “can” or “car” or “cat”

Regular Expressions (5)

- Quantifiers: Express the number of repetitions of a preceding element
  - Zero or one repetitions: `?`
  - Zero or more repetitions: `*`
  - One or more repetitions: `+`

- Example: `card?`
  - Matches “car” or “card”
- Example: `ab*c`
  - Matches “ac” or “abc” or “abbc” or “abbbc” or …
- Example: `be+f`
  - Matches “bef” or “beef” or “beeef” or “beeeef” or…

Regular Expressions (6)

- Range Quantifier: Express the number of repetitions of a preceding element
  - `{m,n}` – between m and n repetitions
  - `{m,}` – at least m repetitions
  - `{n}` – exactly n repetitions

- Example: `fo{1,5}`
  - Will match “fo” or “foo” or “fooo” or “foooo” or “fooooo”

- Example: `60{3,}`
  - Will match “6000” or “60000” or “600000” or …

- Example: `60{2}`
  - Will match “600”

Regular Expressions (7)

- Quantifiers and groups can be used together

- Example: `(ca(t|r))+`
  - Matches “cat” or “car” or “catcat” or “catcar” or “carcar” or “catcarcat” or …

- Example: `EC?E( (1|2|3)+)*`
  - Matches “EE” or “ECE” or “EE 111” or “ECE 2” or “EE 13 2” or “ECE 123 32 1” or …

Regular Expressions (8)

- Character Set: A compact representation to represent a set of characters to match
  - Use square brackets to denote a set: [<characters>]

- How could we match all integers?
  - (0|1|2|3|4|5|6|7|8|9)+

- Another way is to match characters in a set
  - [0123456789]+

- An even shorter way is to take advantage of the regular ordering of the digits
  - [0-9]+

Regular Expressions (9)

- If a set of characters can form a natural range (e.g. 0 – 9, a-z) then you can use `<start><dash><end>` notation
  - Multiple ranges can be placed inside the square brackets
  - Ranges can be used in conjunction with single characters
  - Can invert the set using a ^ character at the beginning

- Example: `[a-z0-9]+`
  - Will match any alphanumeric string

- Example: `[aeiou0-9]*`
  - Will match any string containing only vowels and numbers

- Example `[^a-z0-9]+`
  - Will match 1 or more characters that are **NOT** a-z or 0-9.

Regular Expressions (10)

- To match any single character use a period “.”
  - Will match all characters including whitespace

- Example: ...
  - Will match “abc” or “a 1” or “ 3” or …

- Example: .*
  - Will match: “any string of any length”

Regular Expressions (11)

- Escaping: Characters that have special meaning must be escaped when used in a regex
  - Prepend a backslash character (`\`)

- Example: `[0-9]*\.[0-9]+`
  - Will match ".345" or "12.4" or "9.11111" or …

- Example: `[a-f0-9]+(\+[a-f0-9]+)?`
  - Will match: "a4f" or "beef+a6b" or …

Regular Expressions (12)

- Search vs. Match: An implementation detail of the regex functionality provided
  - Search refers to finding a match somewhere in a string (substring)
  - Match refers to checking if the entire string matches
- Some regex engines will preform search by default
  - Check documentation to find out
  - Try out some basic examples

Regular Expressions (13)

- Start of String: the position right before the first character
  - Can be matched using the ‘`^`’ character
- End of String: the position right after the last character
  - Can be matched using the ‘`$`’ character

- Example: `[0-9]+`
  - A search would match: “abc 012 def” or “999 bbb” or …
  - A match would match: “123” or “999” or …

- Example: `^[0-9]+$`
  - Will match only strings that contain digits 0 to 9 and nothing else
  - A match would match: “123” or “999” or …
  - A search would match: “123” or “999” or …

Symbol Summary

Or:	|
Grouping:	( )
Quantifiers:	? * +
Range Quantifiers:	{ , }
Character Set:	[ ]
A Single Character:	.
Start & End:	^ $

Special Sequences in Python

- Commonly used expressions have shorthand sequences

`\d`	Equivalent to `[0-9]`
`\D`	Equivalent to `[^0-9]`
`\w`	Equivalent to `[a-zA-Z0-9_]`
`\W`	Equivalent to `[^a-zA-Z0-9_]`
`\s`	Equivalent to any whitespace character
`\S`	Equivalent to any non whitespace character
`\\`	Matches a literal backslash
`\b`	Match the empty string the forms the boundary of a word

Special Sequences (2)

- Special sequences in Python regex conflict with escaped characters
  - `\b` Backspace in a string literal
  - `\b` Word boundary in a regex

- To avoid this conflict, regular expressions are written as “raw” strings
  - Typical String: `"this is a string"`
  - Raw String: `r"this is a raw string"`

Groups

- A group is formed by parenthesizing (part of) the regular expression
  - `([a-z]+|[0-9]+)`
  - `hello (world|ee364)`

- In Python, a group also specifies the text you want to extract from a matched string
  - Python reserves the `0`th group as the entire string that matches the regular expression

Groups (2)

- Example: Search for an email address and get the username and domain

Regex: `([\w.-]+)@([\w.-]+)`

Input: `"foo mgoldfar@purdue.edu baz"`

Group 0:	mgoldfar@purdue.edu
Group 1:	mgoldfar
Group 2:	purdue.edu

Groups (3)

- Groups can be given names

```
(?P<GroupName> … )
```

Regex: `(?P<user>[\w.-]+)@(?P<domain>[\w.-]+)`

Input: `"foo mgoldfar@purdue.edu baz"`

Group 0:	mgoldfar@purdue.edu
Group “user”:	mgoldfar
Group “domain”:	purdue.edu

Greedy and Non-greedy

- Quantifiers in a regex match as much as possible
- `?` is appended to the quantifier to indicate non-greedy behaviour
  - A non-greedy match will match as little as possible

Greedy	Non-Greedy
`(pattern)+`	`(pattern)+?`
`(pattern)*`	`(pattern)*?`
`(pattern)?`	`(pattern)??`
`(pattern){n}`	`(pattern){n}?`
`(pattern){n,m}`	`(pattern){n,m}?`

Greedy vs. Non-greedy (2)

- Example: Match an HTML tag name:

Regex: `<(.*)>`

Input: `"<h1>ECE 364</h1>"`

Group 0: `"<h1>ECE 364</h1>"`

Group 1: `"h1>ECE 364</h1"`

- None of the groups contain what we want because the `*` is greedy

Greedy vs. Non-greedy (3)

- Example: Match an HTML tag name:

Regex: `<(.*?)>`

Input: `"<h1>ECE 364</h1>"`

Group 0: `"h1"`

Group 1: `"h1"`

- Non-greedy `*?` operator results in the correct behavior

`re` Module

- The `re` module provides access to regular expression functionality
- Always remember to `import re` before using any regular expressions

Match and Search

- `re.match(<pattern>, <string>)`
  - If zero or more characters at the beginning of *`<string>`* match the regex *`<pattern>`*, return a **MatchObject**
  - Return **None** if the string does not match the pattern

- `re.search(<pattern>, <string>)`
  - Scan through *`<string>`* looking for a location where the regex *`<pattern>`* produces a match, and return a corresponding **MatchObject**
  - Return **None** if no position in the string matches the pattern

Match and Search (2)

```
input = "foo bar@baz.bin 923"
expr = r"([\w.-]+)@([\w.-]+)"

if re.match(expr, input):
     print("The input starts with an email!")
elif re.search(expr, input):
     print("The input contains an email.")
else:
     print("No email found.")
```

The `MatchObject`

- When a regular expression finds a match, a `MatchObject` is returned
  - `None` is returned if there is no match

- This object contains information about the matched string

The `MatchObject` (2)

- `m.group(g)`
  - Returns the string contained in the `g`th group

- `m.group(g1, g2, g3, …)`
  - Returns a tuple containing the `g1`th, `g2`nd, `g3`rd, … groups

- `m.groupdict()`
  - Returns the a dictionary of all named groups keyed by group name

- Arguments to `m.group()` can be an index or a string representing the group name

The `MatchObject` (3)

```
m = re.match(r"(?P<int>\d+)\.(\d*)", "3.14")

m.group(0) returns "3.14"
m.group(1) returns "3"
m.group(2) returns "14"

m.group("int") returns "3"

m.group(0, "int", 2) returns ("3.14", "3", "14")

m.groups() returns the tuple ("3", "14")

m.groupdict() returns {"int" : "3"}
```

The MatchObject (4)

```
pattern = r"[0-9]+(?P<foo>\.[0-9]+)"
m = re.search(pattern, "Hello 56.43 World")

if m:
  gp = m.groupdict()
  print(gp["foo"])
else:
  print("Not found")
```

Split

- re.split(*pattern*, *string*, *maxsplit=0*)
  - Split *string* by the occurrences of *pattern*
  - If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list

```
>>> re.split(r"\W+", "foo, bar, baz.")
['foo', 'bar', 'baz', '']
```

Find

- re.findall(*pattern*, *string*)
  - Return all non-overlapping matches of *pattern* in *string*, as a list of strings
  - The *string* is scanned left-to-right, and matches are returned in the order found.

```
>>> re.findall(r"[0-9]+", "hello 56.78 world 25")
['56', '78', '25']
```

```
>>> re.findall(r"[\w.]+@[\w.]+", "26 bar@biz.com baz@foo 99")
['bar@biz.com', 'baz@foo']
```

Substitution

- `re.sub(`*`pattern, repl, string, count=0`*`)`
  - Return the string obtained by replacing the leftmost non-overlapping occurrences of *`pattern`* in *`string`* by the replacement *`repl`*
  - If the pattern isn’t found, *`string`* is returned unchanged

```
>>> re.sub(r"[0-9]+", "NUM", "Hello 267 World 8")
'Hello NUM World NUM'

>>> re.sub(r"[0-9]+", "NUM", "Hello 267 World 8", 1)
'Hello NUM World 8'
```

Substitution (2)

- re.subn(*pattern*, *repl*, *string*, *count=0*)
  - Performs the substitution in the same way re.sub() does
  - Returns a tuple containing the new string and the number of occurrences of `pattern` replaced

```
>>> re.subn(r"[0-9]+", "NUM", "Hello 267 World 8”)
('Hello NUM World NUM', 2)

>>> re.subn(r"[0-9]+", ”NUM", "Hello 267 World 8", 1)
('Hello NUM World 8', 1)
```

Flags

- Flags can be used to modify how the regular expression engine behaves.
  - Passed to the functions covered in the previous slides
  - Combine flags with a bit-wise or operator

`re.I` or `re.IGNORECASE`	Perform non case-sensitive matching
`re.M` or `re.MULTILINE`	Make `^` and `$` apply to each line (not the entire string)
`re.S` or `re.DOTALL`	Make `.` match all characters, even newline
`re.X` or `re.VERBOSE`	Ignore un-escaped whitespace and comments.

Flags (2)

```
input = "foo bar@BAZ.com 923"
expr = r"([\w.-]+)@([\w.-]+)"

if re.match(expr, input, re.I):
     print("The input starts with an email!")
elif re.search(expr, input, re.I):
     print("The input contains an email.")
else:
     print("No email found.")
```

Compiled Regular Expressions

- A regular expression can be *compiled* into a special object.
    - Improves the performance when performing lots of repeated matches or searches
    - You should compile your regular expression if it is going to be used multiple times (i.e. in a loop)

```
Reg_Exp = re.compile(expression[, flags])
```

- The regular expression can then be passed around like any other Python value.

Compiled RegEx (2)

- Functions of a compiled regex are identical to the module level functions

```
Reg_Exp = re.compile(expression[, flags])

Reg_Exp.search(string)

Reg_Exp.match(string)

Reg_Exp.findall(string)

Reg_Exp.split(string[, maxsplit])

Reg_Exp.sub(replacement, string[, count])

Reg_Exp.subn(replacement, string[, count])
```