Jython

Introduction to Jython programming

Agenda

- Module 1 - Introduction to Jython
- Module 2 - Jython language and semantics
- Module 3 - Data types
- Module 4 - Regular expressions
- Module 5 - Functions, debugging, modules, and packages
- Module 6 - Objects, classes and exceptions
- Module 7 - Java integration
- Module 8 - Testing
- Module 9 - System programming
- Module 10 - Conclusion

Topics

- Introduction to regular expressions
- Metacharacters
- Basic searching
- Compile, matching, groups
- Modifying strings
- Quiz
- Q & A

Introduction to regular expressions

Background

- Regular expressions
  - Tiny, highly specialized language
    - Common to many programming languages
  - Rules for set of strings to match
    - Not all possible string processing task can be accomplished
- Compiled into bytecodes and run by a matching engine

Objects

- RegexObject - compiled regular expression
  - AKA PatternObject
- MatchObject - matched pattern

RegexObject

Overview

- Compiling creates a reusable pattern object

```
pattern = re.compile(r'<HTML>')
pattern.match("<HTML>")
```

- Alternate, one-time match

```
re.match(r'<HTML>', "<HTML>")
```

Basic searching

Raw strings

- Raw strings are important
  - Eliminate the need to multiple escape characters

Characters	Stage
\section	Test string to be matched
\\section	Escaped "\" character for re.compile
\\\\section	Escaped "\" character for string literal

- Syntax

```
re.compile(r"\section")
```

Searching1

- *match(string [,pos [,end]])*
  - Returns a MatchObject if there is a match

```
jython2.5.3 — java — 107×39
cerro-colorado:jython2.5.3 rereidy$ java -jar jython.jar
impoJython 2.5.3 (2.5:c56500f08d34+, Aug 13 2012, 14:48:36)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java1.8.0_45
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> patt = re.compile(r'<HTML>')
>>> patt.match("<HTML><HEAD>")
<org.python.modules.sre.MatchObject object at 0x2>
>>>

>>> m = patt.match("xx")
>>> print type(m)
<type 'NoneType'>
>>>
```

Searching2

- match attempts to match at the beginning of the string

```
jython2.5.3 — java — 107×39
cerro-colorado:jython2.5.3 rereidy$ java -jar jython.jar
Jython 2.5.3 (2.5:c56500f08d34+, Aug 13 2012, 14:48:36)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java1.8.0_45
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> patt = re.compile(r'<HTML>')
>>> m = patt.match("    <HTML>")
>>> type(m)
<type 'NoneType'>
>>>
```

- Use search()

```
>>> m = patt.search("    <HTML>")
>>> type(m)
<type 'org.python.modules.sre.MatchObject'>
>>>
```

- Alternate form of match()

```
>>> patt = re.compile(r'<HTML>')
>>> m = patt.match("        <HTML>", 2)
>>> type(m)
<type 'org.python.modules.sre.MatchObject'>
>>>
```

Splitting and substitution

split()

- Syntax - *`split(string, maxsplit=0)`*

substitution

- Syntax - *`sub(repl, string, count=0)`*
  - Returns a string
- Alternate
  - Syntax - *`subn(repl, string, count=0)`*
    - Returns a tuple - string and count of substitutions

MatchObject

Overview

- Represents a matched pattern
- Set of operations to work with captured groups
  - match
  - search
  - finditer

Grouping

- Syntax - `group([group1, …])`
  - Subgroups of the match
- Groups can be named
- Alternate:
  - `groups()`
    - Tuple of the matched patterns
  - `groupdict()`
    - Use with named groups - dictionary with name as key
- Indexing
  - `start([group])` - index where pattern match starts
  - `end([group])` - index of where pattern ends
  - `span()` - tuple of start and end position

More grouping

- Non capturing groups
- Atomic grouping
  - Not supported by the re module
  - Install the regex module
    - Cannot install into Jython

Flags

- Modify some aspects of regex functionality

Flag	Meaning
`DOTALL`, `S`	Make `.` match any character, including newlines
`IGNORECASE`, `I`	Do case-insensitive matches
`LOCALE`, `L`	Do a locale-aware match
`MULTILINE`, `M`	Multi-line matching, affecting `^` and `$`
`VERBOSE`, `X`	Enable verbose REs, which can be organized more cleanly and understandably.
`UNICODE`, `U`	Makes several escapes like `\w`, `\b`, `\s` and `\d` dependent on the Unicode character database.

- Multiple flags can be combined using the bitwise or operator ("|")

Back references

Basics

- Match text captured into a group
- Match duplicated words in a string

```
import re

patt = re.compile(r'(\w+) \1')
m = patt.findall(r'hello hello world world xy xy')
for line in m:
    print "duplicate word: ", line
```

Group

Backreference

```
duplicate word:  hello
duplicate word:  world
duplicate word:  xy
```

Group numbering

- Each set of capturing parenthesis from left to right are assigned a number

Group 1 Group 2 Group N

```
patt = re.compile(r'(\w+)\s*(\d{1,5})...(\W+)')
```

- Best practice: Name all groups and reference them by the name in `sub()` and `subn()` functions

Metacharacters

List of metacharacters

Metacharacter	Description
.	Any character except newline
^	Beginning of line; complementing in character class
$	End of line
*	Zero or more match
+	One or more match
?	Zero or one match
{ }	Repetition {m} exactly m characters; {m,n} match from m to n characters; {m,n}? m to n characters as few as possible
[ ]	Specify character class
\	Escape metacharacters; special character sequences (next slide)
|	Alternation (or operator)
( )	Grouping

Character classes

- Indicates characters in a set
  - Characters can be listed individually
  - Ranges of characters can be indicated by giving two characters and separating them by a '-'
  - Special characters lose their special meaning inside sets
  - Character classes such as \w or \S (defined below) are also accepted inside a set, although the characters they match depends on whether **LOCALE** or **UNICODE** mode is in force.
  - Characters that are not within a range can be matched by *complementing* the set
  - Match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both [()[\]{}] and []()[{}] will both match a parenthesis.

Metacharacters

- Not active inside classes
  [golf$] - will match the characters ‘g’, ‘o’, ‘l’, ‘f’, and ‘$’

```
>>> import re
>>> re.findall(r'[golf$]', "golf$")
['g', 'o', 'l', 'f', '$']
>>> re.findall(r'[golf]$', "golf")
['f']
>>>
```

Sequences

Sequence	Description
\d	This matches any decimal digit; this is equivalent to the class [0-9]
\D	This matches any non-digit character; this is equivalent to the class [^ 0-9]
\s	This matches any whitespace character; this is equivalent to the class [\ t\ n\ r\ f\ v]
\S	This matches any non-whitespace character; this is equivalent to the class [^ \t\ n\ r\ f\ v]
\w	This matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_]
\W	This matches any non-alphanumeric character; this is equivalent to the class [^ a-zA-Z0-9_]
\b	Matches empty string at beginning of word
\B	Matches empty string (not at beginning of word)
\A	Matches at the start of the string
\Z	Matches at end of string

Quantifiers

Basics

- Quantifiers stick to character, toke, or subexpression directly to the left
  - In A+ the quantifier "+" applies to the character A
  - In \w* the quantifier "*" applies to the token \w
  - In carrots? the quantifier "?" applies to the character s—not to carrots
  - In `(?:apple,|carrot,)+` the quantifier "+" applies to the subexpression `(?:apple,|carrot,)`

Quantifier	Description
+	once or more
A+	One or more As, as many as possible (greedy), giving up characters if the engine needs to backtrack (docile)
A+?	One or more As, as few as needed to allow the overall pattern to match (lazy)
A++	One or more As, as many as possible (greedy), not giving up characters if the engine tries to backtrack (possessive)
*****	zero times or more
A*	Zero or more As, as many as possible (greedy), giving up characters if the engine needs to backtrack (docile)
A*?	Zero or more As, as few as needed to allow the overall pattern to match (lazy)
A*+	Zero or more As, as many as possible (greedy), not giving up characters if the engine tries to backtrack (possessive)
?	zero times or once
A?	Zero or one A, one if possible (greedy), giving up the character if the engine needs to backtrack (docile)
A??	Zero or one A, zero if that still allows the overall pattern to match (lazy)
A?+	Zero or one A, one if possible (greedy), not giving the character if the engine tries to backtrack (possessive)
{x,y}	x times at least, y times at most
A{2,9}	Two to nine As, as many as possible (greedy), giving up characters if the engine needs to backtrack (docile)
A{2,9}?	Two to nine As, as few as needed to allow the overall pattern to match (lazy)
A{2,9}+	Two to nine As, as many as possible (greedy), not giving up characters if the engine tries to backtrack (possessive)
A{2,}	Two or more As, greedy and docile as above.
A{2,}?	Two or more As, lazy as above.
A{2.}+	Two or more As, possessive as above.
A{5}	Exactly five As. Fixed repetition: neither greedy nor lazy.

Greediness

As many as possible

- Default behavior of quantifier is to match as many as possible
  - \d+ == "one or more" digits

```
re.search(r"\d+", "123")
```

Greedyness

- The "*", "+", and "?" qualifier are greedy
  - Will match as much of the text as possible

```
re.findall(r"<.*>", "<H1>title</H1>")
```

- Will match the entire string

```
>>> re.findall(r"<.*>", "<H1>title</H1>")
['<H1>title</H1>']
>>>
```

- Add a "?" after "*"

```
>>> re.findall(r"<.*?>", "<H1>title</H1>")
['<H1>', '</H1>']
>>>
```

Looking around

Basics

- Match previous (look behind) or ulterior (look ahead) value relative to the current position
  - Assertion without consuming
  - Return positive or negative result
- Look ahead
  - Positive - expression preceded by "?="

    ```
    (?=regex)
    ```

  - Negative - expression preceded by "?!"

    ```
    (?!regex)
    ```

- Look behind
  - Positive - expression preceded by "?<="

    ```
    (?<=regex)
    ```

  - Negative - expression preceded by "?<!"

    ```
    (?<!regex)
    ```

File filtering

Common tasks

- Filtering log files for interesting information
- Filtering data files in ETL jobs

Quiz

1. What is another name for the RegexObject?

A. PatternObject

2. What type of object does the groups() method return?

A. Tuple

3. How many groups in the following regex?

```
re.compile(r'((\d{1,3})\.(\d{1,3})\.(\d{1,3|)\.
(\d{1,3})')
```

A. 5

Q & A

Exercises

1. Write a string search program to find all matches of "fox" in the following sentence:

   The quick, foxy brown fox-like animal.

2. Modify the program to only match the word "foxy" (note: the word should match exactly).

3. Write a program to return all digits in a phone number in the format of "999-999-9999" (do not return the "-" characters). Provide 2 solutions for this problem.

4. Write a regex to display the names of all included files in stdio.h (#include <file>).

   NOTE: If you do not have a stdio.h file on your system, use the file located here: https://www.gnu.org/software/m68hc11/examples/stdio_8h-source.html

5. Modify filt1.py to accept the word length form the command line.

   Note: You will need to modify the file path in the open() call for your system.

   Make sure:

   1. The argument is an integer (use a regex)
      a. The argument is reasonable (i.e. positive, etc.)
   2. Print a usage message if no command line argument