Regular Expressions

Curtis Huttenhower (chuttenh@hsph.harvard.edu)

Eric Franzosa (franzosa@hsph.harvard.edu)

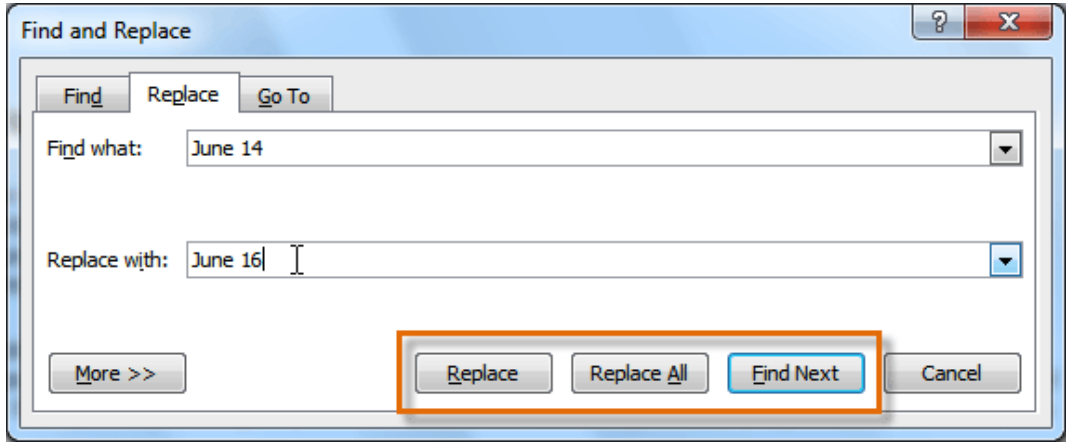
http://huttenhower.sph.harvard.edu/bst281

Summary

- Syntax for regular expressions (REs)
- REs in Python
- REs on the command line
- Pattern sensitivity and precision
- RegexOne activity

Regular expressions

- Also called “regexps” or “regexes” or “REs” as shorthand
- A language for describing patterns in strings
- Useful for
  - Asking if a **pattern** occurs in a **text**
  - “Capturing” all or part of the pattern for manipulation
  - Replacing all or part of the pattern
- A much more powerful version of find-and-replace, implemented in many text editors and programming languages (including Python)



https://xkcd.com/208/

Introduction to slide style

- Description of RE pattern concept / use cases

Pattern using the concept

Text with a match to the pattern

Text without a match (possibly a *near-miss*)

Trivial regular expressions: exact matching

- A RE pattern is coded as a string

```
Hello         # pattern
```

```
Hello, world # text
```

```
hello, world # matching is case-sensitive
```

The wildcard character

- A dot (`.`) represents a wildcard (matches any character)
- Use `\.` to specifically match a dot (similar for other special characters)

```
.ello
```

```
Hello, world
```

```
hello, world
```

```
Mellow yellow
```

```
..e..
```

```
Jennifer and Marty went to the store
```

Pre-defined character classes

- \d matches any digit (0-9)
- \D matches any non-digit character (letters, punctuation, etc.)
- \s matches any whitespace character (space, tab, newline)

```
\d\d\d\d
```

```
Yesterday, December 7, 1941--a date which will live in infamy
```

```
\d\D
```

```
In 2265, starship crew members enjoy playing 3D chess
```

```
\s\s
```

```
Some people add two spaces after a sentence.__Others add one.
```

Pre-defined character classes

- \w matches any “word” character (A-Z, a-z, 0-9, and “_”)
- \W matches any non-word character (punctuation and space)

```
\W\w\w\w\w\W
```

```
It’s my party and I’ll cry if I_want_to
```

```
In_2265, crew member enjoy playing 3D chess
```

Pre-defined character classes

- `\t` (tab) `\n` (newline) and `\\` (backslash) match as in Python

```
\t\d\d\d\d\n
```

```
1981    1982    1983    1984    1985
1986    1987    1988    1989    1990
```

Custom character classes (square brackets, [ ])

- [AB] matches “A” and “B”
- [A-E] matches any character “A” through “E”
- [A-Za-z0-9_] matches any word character (equivalent to \w)

```
[ACGT]
```

```
A DNA sequence is represented by the letters A, C, G, and T
```

```
[D-M]r
```

```
Dr. Grant has a Ph.D., while Mr. Hammond does not
```

Custom character classes

- Negate a character class with an initial “^”
- [^ABC] matches any character EXCEPT “A” or “B” or “C”
- [^A-E] matches any character EXCEPT “A” through “E”

```
[^ACGT]
```

```
A DNA sequence is represented by the letters A, C, G, and T
```

```
[^D]r
```

```
Dr. Grant has a Ph.D., while Mr. Hammond does not
```

Boundaries

- ^ matches the start of a string (*outside of a character class*)
- $ matches the end of a string
- \b matches a “word boundary” (beginning/end of a line, whitespace, or a non-word character)match anything looking like a word

```
^John
```

```
John Smith, have you met my friend John Doe?
```

```
deer$
```

```
Doe, a deer, a female deer
```

```
deer\b   don’t match deers, deery    \bdeer\b:isolate deer as a single word
```

```
Doe, a deer, a female deer
```

Repetition

- ? Indicates a single, optional match

```
AB?C   B is optional
```

AC

ABC

ABBC

Repetition

- ? Indicates a single, optional match
- * matches 0 or more occurrences

AB*C

AC

ABC

ABBC

Repetition

- ? Indicates a single, optional match
- * matches 0 or more occurrences
- + matches 1 or more occurrences

```
AB+C    any number of B
```

AC

ABC

ABBC

What's another way of representing "AB+C"?

```
ABB*C
```

Repetition with subpatterns

- To describe repetition of more than one character, enclose the “sub-pattern” in parentheses
- (*Parentheses have a 2nd function in REs that we’ll see shortly*)

```
(CAT)+
```

```
ACTAGTGATCGATGCTGTAGTGCTAGCTGATCTGTAGCATGCTGACTAGCTGGTCTGCATCATTAGCTAGCTAGTACTAGTGTGCGT
```

```
(a[a-z])+
```

```
Who doesn’t like Bananarama?
```

Bounded repetition

- `{n}` matches exactly *n* repetitions
- `{n,m}` matches *n* to *m* repetitions, inclusive
- `{n,}` matches at least *n* repetitions
- `{,m}` matches at most *m* repetitions

```
[A-Z][a-z]{2,12}
```

```
“Rutherfordium” is the longest chemical element name, while “Tin” is the shortest
```

```
(CAT){3,}
```

```
GATACGACATCATCATAGCATAG contains a DNA repeat region
```

Choice

- | (pipe) behaves as a logical OR
- Often combined with parentheses to indicate a choice of sub-patterns

```
\w+\.(txt|py)\b
```

```
my_script.py  my_input.txt  README.md  my_output.txt
```

```
(a[a-z])+
```

```
Who doesn't like Bananarama?
```

Summary

- Syntax for regular expressions (REs)
- **REs in Python**
- REs on the command line
- Pattern sensitivity and precision
- RegexOne activity

REs in Python

- Bundled in the `re` module
  - `import re`
- Three (most-commonly used) functions in this module
  - `re.search( )`
  - `re.finditer( )`
  - `re.sub( )`

re.search

- `re.search( pattern, text )`
- "Pattern" and "text" are string data (possibly stored in variables)

```
re.search( r"C.T", "CAT" )
# is equivalent to
strPattern = "C.T"
strText = "CAT"
re.search( strPattern, strText )
```

- By convention, we use raw strings to define a pattern
  - `strPattern = r"C\tT"`
  - This means "see '\t' as '\' followed by 't' and not the special 'tab' character"
  - Forces the RE engine to process "\t" as "tab" and not Python

re.search

- re.search finds the first valid hit to a pattern
- Consider re.search(r“A+”, “ABAAAAAAAAAAAAAAAAAA”)
  - The match will be the single A before B and not the long string of As to follow
- re.search returns a special “Match” data object if a hit is found
  - Otherwise it returns None

```
print( re.search( r“C.T”, “CAT” ) )
```

```
<_sre.SRE_Match object at 0x2b7c6b71c3f0>
```

```
print( re.search( r“C.T”, “BAT” ) )
```

```
None
```

```
Match = re.search( strPattern, strText )
if Match:
    # do something with Match
```

The Match object

- The Match object stores data about the Match
- Match.start( ) and Match.end( ) return the (Python-style) coordinates of the match in the text

```
Match = re.search( r“.at”, “the cat with the hat sat on the mat” )
#..........................0123456789
print( Match.start( ), Match( end ) )
```

```
4 7
```

```
strText = “the cat with the hat sat on the mat”
Match = re.search( r“.at”, strText )
if Match:
    iStart, iEnd = Match.start( ), Match.end( )
    print( strText[iStart:iEnd] )
```

```
‘cat’
```

Capture groups

- An important use for parentheses in REs is capturing parts of the matched pattern (and avoiding the start-end business of the previous slide)

```
strText = “the cat with the hat sat on the mat”
Match = re.search( r“(.at)”, strText )
if Match:
    print( Match.group( 1 ) )
```

```
‘cat’
```

```
Match = re.search( r“(.at) (.at)”, strText )
if Match:
    print( Match.groups( ) )
```

```
(‘hat’, ‘sat’)
```

Capture groups

- Parentheses used to define sub-patterns are also captured
- If you don't want this behavior, use (?:*subpattern*)
- Especially important when nesting

```
strText = "GGGCATCATCATGGG"
Match = re.search( r"((?:CAT){3})", strText )
if Match:
    print( Match.group( 1 ) )
```

```
'CATCATCAT'
```

re.finditer

- A Pythonic method for finding multiple matches to a pattern in a text
- “iter” refers to “iterable” – the quality that allows data to be looped over in a Python for loop

```
strText = “the cat with the hat sat on the mat”
for Match in re.finditer( r“(.at)”, strText )
    print( Match.group( 1 ) )
```

```
‘cat’
‘hat’
‘sat’
‘mat’
```

re.finditer

- A Pythonic method for finding multiple matches to a pattern in a text
- "iter" refers to "iterable" – the quality that allows data to be looped over in a Python for loop

```
strText = "AAAAAAAAAAAAA---AAAAAAAAAAAAA"
for Match in re.finditer( r"(A+)", strText )
    print( Match.group( 1 ) )
```

```
'AAAAAAAAAAAAA'
'AAAAAAAAAAAAA'
```

- Why not "A" or "AA" or "AAA"?
  - REs are "greedy"
  - Starting from the left side of the string, find the longest match ending at position *p*, then start looking again at position *p*+1

Repetition extra

- `.*` matches as much as possible (often as “filler” in a pattern)
  - Illustrates RE’s “greedy” pattern matching
- `.*?` matches as little as possible

```
www\..*\.com
```

```
www.facebook.com and www.reddit.com are two popular websites
```

```
www\..*?\.com
```

```
www.facebook.com and www.reddit.com are two popular websites
```

re.sub

- The RE equivalent of find-and-replace
- Usage: `re.sub( strFind, strReplace, strText )`
- Returns a new string (not a `Match` object)

```
re.sub( r“T”, “U”, “ATGAGTTAA” )
```

```
‘AUGAGUUAA’
```

```
re.sub( r“[AT]+”, “”, “AAAAGGCTTAGCATAGGCCAATA” )
```

```
‘GGCGCGGCC’ # note that ALL valid matches are replaced, similar to re.finditer
```

Substitution with capture groups

- Usage: `re.sub( strFind, strReplace, strText )`
- Groups captured from `strFind` can be used in `strReplace`
- First group represented by `\\1`, second by `\\2`, and so forth

```
re.sub( r“Do you like (\w+)?”, “Yes, I like \\1!”, “Do you like Pigs?” )
```

```
‘Yes, I like Pigs!’
```

```
re.sub( r“([A-Z][a-z]+) ([A-Z][a-z]+)”, “\\2, \\1”, “John Doe” )
```

```
Doe, John
```

Summary

- Syntax for regular expressions (REs)
- REs in Python
- **REs on the command line**
- Pattern sensitivity and precision
- RegexOne activity

*nix utilities: grep

What does it do?
Isolate lines of a data stream (file or STDIN) that match a pattern

Usage
`$ grep pattern my_file`
`$ cat *.txt | grep pattern`

Options	What does it do?
-P	Richer pattern options (regular expressions); more on these in a later lecture

Usage
`$ grep -P 'regexp' my_file`

*nix utilities: sed

What does it do?
Edit a data stream, most often used for find/replace operations

Usage
`$ sed “s/find/replace/g” my_file`
`$ sed “s/apple/banana/g” my_file # replace all instances of “apple” with “banana”`

*nix utilities: `perl -pe`

What does it do?
Edit a data stream, most often used for find/replace operations

Usage
`$ perl -pe "s/`*find*`/`*replace*`/g" my_file`
`$ perl -pe "s/apple/banana/g" my_file`

- Perl is a scripting language, like Python, heavily geared toward text processing
  - Hence, REs are a core part of the language
- Perl is famous for it's one-liners, which can be executed on the command line
- RE concepts that don't work in `sed` will work with `perl -pe`

Summary

- Syntax for regular expressions (REs)
- REs in Python
- REs on the command line
- **Pattern sensitivity and precision**
- RegexOne activity

Pattern sensitivity and precision

- Think carefully about your REs and test them
- Incorrect assumptions will lead to missed hits (loss of sensitivity)
- Incorrect assumptions will lead to nonsense hits (loss of precision)

```
[A-Z][a-z]+ [A-Z][a-z]+ # matches a first name + last name
```

```
Jan Levinson
```

```
Jan Levinson-Gould
```

```
Jan R. Levinson
```

```
Puerto Rico
```

```
Potassium Chloride
```

Pattern sensitivity and precision

- Think carefully about your REs and test them
- Incorrect assumptions will lead to missed hits (loss of sensitivity)
- Incorrect assumptions will lead to nonsense hits (loss of precision)

```
GENE#1234
```

```
GENE#1234
```

```
GENE#12345
```

https://xkcd.com/208/

F. Murray Abraham

123 East 21st Street

Suite 1B

Wilkes-Barre, PA 18701

USA

Wilkes-Barre, PA 18701

```
\b[A-Z]{2} \d{5}\b
```

Data that are well-defined tend to be easier to find with REs!

Summary

- Syntax for regular expressions (REs)
- REs in Python
- REs on the command line
- Pattern sensitivity and precision
- **RegexOne activity**