Algorithmic Approaches for Biological Data, Lecture #8

Katherine St. John

City University of New York
American Museum of Natural History

17 February 2016

Outline

- More on Pattern Finding: Regular Expressions

Outline

- More on Pattern Finding: Regular Expressions
- Processing CSV files

Recap: Regular Expressions

Recap: Regular Expressions

Regular Expression	Description of Matching Strings
[ACGT]*	A DNA sequence– any string consisting only of A, C, G, and T

Recap: Regular Expressions

Regular Expression	Description of Matching Strings
[ACGT]*	A DNA sequence– any string consisting only of A, C, G, and T
*[ACGU]**	A RNA sequence– any string consisting only of A, C, G, and U

Recap: Regular Expressions

Regular Expression	Description of Matching Strings
[ACGT]*	A DNA sequence– any string consisting only of A, C, G, and T
*[ACGU]**	A RNA sequence– any string consisting only of A, C, G, and U
[AT]+	*1 or more repeats of AT: AT, ATAT, ATATAT,...*

Recap: Regular Expressions

Regular Expression	Description of Matching Strings
[ACGT]*	A DNA sequence– any string consisting only of A, C, G, and T
[ACGU]*	A RNA sequence– any string consisting only of A, C, G, and U
[AT]+	1 or more repeats of AT: AT, ATAT, ATATAT,...
ATG[ATGC]{30,1000}A{5,10}	A sequence beginning with ATG and ending with 5 to 10 A's. Overall length is 38 to 1013 bp.

Recap: Regular Expressions

Regular Expression	Description of Matching Strings
[ACGT]*	A DNA sequence– any string consisting only of A, C, G, and T
[ACGU]*	A RNA sequence– any string consisting only of A, C, G, and U
[AT]+	1 or more repeats of AT: AT, ATAT, ATATAT,...
ATG[ATGC]{30,1000}A{5,10}	A sequence beginning with ATG and ending with 5 to 10 A's. Overall length is 38 to 1013 bp.
([ACGT]{3})*	A DNA sequence that exactly breaks into codons (3-letter sequences).

Recap: Regular Expressions

Regular Expression	Description of Matching Strings
[ACGT]*	A DNA sequence– any string consisting only of A, C, G, and T
[ACGU]*	A RNA sequence– any string consisting only of A, C, G, and U
[AT]+	1 or more repeats of AT: AT, ATAT, ATATAT,...
ATG[ATGC]{30,1000}A{5,10}	A sequence beginning with ATG and ending with 5 to 10 A's. Overall length is 38 to 1013 bp.
([ACGT]{3})*	A DNA sequence that exactly breaks into codons (3-letter sequences).
ATG([ATGC]{3}){30,1000}A{5,10}	An open reading frame (ORF): a sequence that starts with the start codon ATG, followed by any number of codons, and ending with a stop codon (TAA, TAG, or TGA).

RE Searching

- Can be used as a conditional test:

RE Searching

- Can be used as a conditional test:

  ```
  if re.search(r"GC[ATGC]GC", dna):
      print "Pattern found!"
  ```

RE Searching

- Can be used as a conditional test:

  ```
  if re.search(r"GC[ATGC]GC", dna):
      print "Pattern found!"
  ```

- Search returns a match object:

RE Searching

- Can be used as a conditional test:

  ```
  if re.search(r"GC[ATGC]GC", dna):
      print "Pattern found!"
  ```

- Search returns a match object:

  ```
  dna = "ACTCGTACGAAAGCTGCTTATACGCGCG"
  m = re.search(r"GC[ATGC]GC, dna)
  print "The matching string is",
  m.group()
  print "Match starts at", m.start()
  print "Match ends at", m.end()
  ```

RE Searching

- Can be used as a conditional test:

  ```
  if re.search(r"GC[ATGC]GC", dna):
      print "Pattern found!"
  ```

- Search returns a match object:

  ```
  dna = "ACTCGTACGAAAGCTGCTTATACGCGCG"
  m = re.search(r"GC[ATGC]GC, dna)
  print "The matching string is",
  m.group()
  print "Match starts at", m.start()
  print "Match ends at", m.end()
  ```

RE `findall()`

- `findall()` returns a list of strings: `import re`

  ```
  dna = "ACTGCATTATATCGTACGAAATTATACGCGCG"
  runs = re.findall("[AT]4,100", dna)
  print runs
  ```

RE findall()

- findall() returns a list of strings: import re

  ```
  dna = "ACTGCATTATATCGTACGAAATTATACGCGCG"
  runs = re.findall("[AT]4,100", dna)
  print runs
  ```

  output:
  ['ATTATAT', 'AAATTATA']

Regular Expressions Overview

Python 2.7 Regular Expressions

Non-special chars match themselves. Exceptions are special characters:

```
\       Escape special char or start a sequence.
.       Match any char except newline, see re.DOTALL
^       Match start of the string, see re.MULTILINE
$       Match end of the string, see re.MULTILINE
[]      Enclose a set of matchable chars
R|S     Match either regex R or regex S.
()      Create capture group, & indicate precedence
```

After '[', enclose a set, the only special chars are:

```
]       End the set, if not the 1st char
-       A range, e.g. a-c matches a, b or c
^       Negate the set only if it is the 1st char
```

Quantifiers (append '?' for non-greedy):

```
{m}     Exactly m repetitions
{m,n}   From m (default 0) to n (default infinity)
*       0 or more. Same as {,}
+       1 or more. Same as {1,}
?       0 or 1. Same as {,1}
```

Special sequences:

```
\A      Start of string
\b      Match empty string at word (\w+) boundary
\B      Match empty string not at word boundary
\d      Digit
\D      Non-digit
\s      Whitespace [ \t\n\r\f\v], see LOCALE,UNICODE
\S      Non-whitespace
\w      Alphanumeric: [0-9a-zA-Z_], see LOCALE
\W      Non-alphanumeric
\Z      End of string
\g<id>  Match prev named or numbered group,
        '<' & '>' are literal, e.g. \g<0>
        or \g<name> (not \g0 or \gname)
```

Special character escapes are much like those already escaped in Python string literals. Hence regex '\n' is same as regex '\n'.

```
\a      ASCII Bell (BEL)
\f      ASCII Formfeed
\n      ASCII Linefeed
\r      ASCII Carriage return
\t      ASCII Tab
\v      ASCII Vertical tab
\\      A single backslash
\xHH    Two digit hexadecimal character goes here
\OOO    Three digit octal char (or just use an
        initial zero, e.g. \O9)
\DD     Decimal number 1 to 99, match
        previous numbered group
```

Extensions. Do not cause grouping, except 'P<name>':

```
(?iLmsux)   Match empty string, sets re.X flags
(?:...)     Non-capturing version of regular parens
(?P<name>...) Create a named capturing group.
(?P=name)   Matches whatever matched prev named group
(?#...)     A comment is ignored.
(?=...)     Lookahead assertion, match without consuming
(?!...)     Negative lookahead assertion
(?<=...)    Lookbehind assertion, match if preceded
(?<!...)    Negative lookbehind assertion
(?(id/name)yes|no) Match 'y' if group 'id' matched, else 'n'
```

Flags for re.compile(), etc. Combine with ' | ':

```
re.I == re.IGNORECASE   Ignore case
re.L == re.LOCALE       Make \w, \b, and \s locale dependent
re.M == re.MULTILINE    Multiline
re.S == re.DOTALL       Dot matches all (including newline)
re.U == re.UNICODE      Make \w, \b, \d, and \s unicode dependent
re.X == re.VERBOSE      Verbose (unescaped whitespace in pattern
                        is ignored, and '#' marks comment lines)
```

Module level functions:

```
compile(pattern[, flags]) -> RegexObject
match(pattern, string[, flags]) -> MatchObject
search(pattern, string[, flags]) -> MatchObject
findall(pattern, string[, flags]) -> list of strings
finditer(pattern, string[, flags]) -> iter of MatchObjects
split(pattern, string[, maxsplit, flags]) -> list of strings
sub(pattern, repl, string[, count, flags]) -> string
subn(pattern, repl, string[, count, flags]) -> (string, int)
escape(string) -> string
purge() # Use re cache
```

RegexObjects (returned from `compile()`):

```
.match(string[, pos, endpos]) -> MatchObject
.search(string[, pos, endpos]) -> MatchObject
.findall(string[, pos, endpos]) -> list of strings
.finditer(string[, pos, endpos]) -> iter of MatchObjects
.split(string[, maxsplit]) -> list of strings
.sub(repl, string[, count]) -> string
.subn(repl, string[, count]) -> (string, int)
.flags    # int, passed to compile()
.groups   # int, Number of capturing groups
.groupindex # {}, Maps group names to ints
.pattern  # string, Passed to compile()
```

MatchObjects (returned from `match()` and `search()`):

```
.expand(template)  # string, Backslash & group expansion
.group([group1, ...]) -> string or tuple of strings, 1 per arg
.groups([default]) -> tuple of all groups, non-matching=default
.groupdict([default]) -> {}, Named groups, non-matching=default
.start([group]) -> int, Start/end of substring match by group
.end([group]) -> int, Group defaults to 0, the whole match
.span([group]) -> tuple (match.start(group), match.end(group))
.pos       int, Passed to search() or match()
.endpos    int, "
.lastindex int, Index of last matched capturing group
.lastgroup string, Name of last matched capturing group
.re        regex, As passed to search() or match()
.string    string, "
```

Gleaned from the python 2.7 're' docs.
http://docs.python.org/library/re.html
https://github.com/tartley/python-regex-cheatsheet
Version: v0.3.3

Regular Expressions: Matches

Non-special chars match themselves. Exceptions are special characters:

```
\        Escape special char or start a sequence.
.        Match any char except newline, see re.DOTALL
^        Match start of the string, see re.MULTILINE
$        Match end of the string, see re.MULTILINE
[]       Enclose a set of matchable chars
R|S      Match either regex R or regex S.
()       Create capture group, & indicate precedence
```

After '[', enclose a set, the only special chars are:

```
]    End the set, if not the 1st char
-    A range, eg. a-c matches a, b or c
^    Negate the set only if it is the 1st char
```

Quantifiers (append '?' for non-greedy):

```
{m}     Exactly m repetitions
{m,n}   From m (default 0) to n (default infinity)
*       0 or more. Same as {,}
+       1 or more. Same as {1,}
?       0 or 1. Same as {,1}
```

Regular Expressions: Special Characters

Special character escapes are much like those already
escaped in Python string literals. Hence regex '\n' is
same as regex '\\n':

```
\a   ASCII Bell (BEL)
\f   ASCII Formfeed
\n   ASCII Linefeed
\r   ASCII Carriage return
\t   ASCII Tab
\v   ASCII Vertical tab
\\   A single backslash
\xHH   Two digit hexadecimal character goes here
\OOO   Three digit octal char (or just use an
       initial zero, e.g. \0, \09)
\DD    Decimal number 1 to 99, match
       previous numbered group
```

Regular Expressions: Special Sequences

Special sequences:

```
\A    Start of string
\b    Match empty string at word (\w+) boundary
\B    Match empty string not at word boundary
\d    Digit
\D    Non-digit
\s    Whitespace [ \t\n\r\f\v], see LOCALE,UNICODE
\S    Non-whitespace
\w    Alphanumeric: [0-9a-zA-Z_], see LOCALE
\W    Non-alphanumeric
\Z    End of string
\g<id>  Match prev named or numbered group,
        '<' & '>' are literal, e.g. \g<0>
        or \g<name> (not \g0 or \gname)
```

Challenges

email.txt

In pairs:

- Design a program that will extract all the email addresses from a file.

Challenges

email.txt

In pairs:

- Design a program that will extract all the email addresses from a file.
- First, figure out a good regular expression to match email formats.

Challenges

email.txt

In pairs:

- Design a program that will extract all the email addresses from a file.

- First, figure out a good regular expression to match email formats.

- Next, use it to find the first match.

Challenges

email.txt

In pairs:

- Design a program that will extract all the email addresses from a file.

- First, figure out a good regular expression to match email formats.

- Next, use it to find the first match.

- Last, add a loop to find all.

CSV Files

- Very structured– the columns and rows matter.

CSV Files

- Very structured– the columns and rows matter.
- To keep that format as a text file:

CSV Files

- Very structured– the columns and rows matter.
- To keep that format as a text file:
    - columns separated by commas (',') and

CSV Files

- Very structured– the columns and rows matter.
- To keep that format as a text file:
    - columns separated by commas (',') and
    - rows separated by new lines ('\n')

CSV Files

- Very structured– the columns and rows matter.
- To keep that format as a text file:
    - columns separated by commas (',') and
    - rows separated by new lines ('\n')
- Rows look like:
  "DOT 84 FLUID 11383",Ceyx lepidus collectoris,Solomon Islands,New Georgia Group,Vella Lavella
  Island,Oula River camp,,,,07 47 30 S,156 37 30 E,Paul R. Sweet,7-May-04,,PRS-2672,,,"Tissue Fluid "
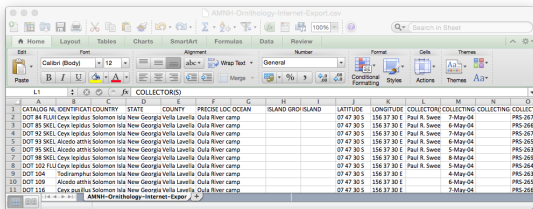
CSV Files

- Built-in package for reading CSV files. To use it:

CSV Files

- Built-in package for reading CSV files. To use it:
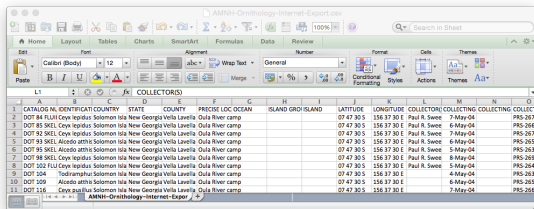 - At top of file, include: `import csv`

CSV Files

- Built-in package for reading CSV files. To use it:
    - At top of file, include: `import csv`
    - Open the file normally: `f = open("in.csv", "rU")`

CSV Files

- Built-in package for reading CSV files. To use it:
  - At top of file, include: `import csv`
  - Open the file normally: `f = open("in.csv", "rU")`
    `"rU"` avoids errors with different newlines, and accepts all variants.

CSV Files

- Built-in package for reading CSV files. To use it:
  - ▶ At top of file, include: `import csv`
  - ▶ Open the file normally: `f = open("in.csv", "rU")`
    "rU" avoids errors with different newlines, and accepts all variants.
  - ▶ Create a reader: `reader = csv.DictReader(f)`

CSV Files

- Built-in package for reading CSV files. To use it:
  - ▸ At top of file, include: `import csv`
  - ▸ Open the file normally: `f = open("in.csv", "rU")`
    "rU" avoids errors with different newlines, and accepts all variants.
  - ▸ Create a reader: `reader = csv.DictReader(f)`
    Uses column names in first line of csv file to access row data.

CSV Files

- Built-in package for reading CSV files. To use it:
  - At top of file, include: `import csv`
  - Open the file normally: `f = open("in.csv", "rU")`
    "rU" avoids errors with different newlines, and accepts all variants.
  - Create a reader: `reader = csv.DictReader(f)`
    Uses column names in first line of csv file to access row data.
  - Read in lines from reader:

CSV Files

- Built-in package for reading CSV files. To use it:
  - At top of file, include: `import csv`
  - Open the file normally: `f = open("in.csv", "rU")`
    "rU" avoids errors with different newlines, and accepts all variants.
  - Create a reader: `reader = csv.DictReader(f)`
    Uses column names in first line of csv file to access row data.
  - Read in lines from reader: `for row in reader:`

CSV Files

- Built-in package for reading CSV files. To use it:
    - At top of file, include: `import csv`
    - Open the file normally: `f = open("in.csv", "rU")`
      "rU" avoids errors with different newlines, and accepts all variants.
    - Create a reader: `reader = csv.DictReader(f)`
      Uses column names in first line of csv file to access row data.
    - Read in lines from reader: `for row in reader:`
    - To access individual entries in a row:

CSV Files

- Built-in package for reading CSV files. To use it:
  - ▶ At top of file, include: `import csv`
  - ▶ Open the file normally: `f = open("in.csv", "rU")`
    "rU" avoids errors with different newlines, and accepts all variants.
  - ▶ Create a reader: `reader = csv.DictReader(f)`
    Uses column names in first line of csv file to access row data.
  - ▶ Read in lines from reader: `for row in reader:`
  - ▶ To access individual entries in a row:
    `if "Malaysia" in row['COUNTRY']: ...`

CSV Files

In pairs,

1. Write a program that will count female specimens in the CSV file.

CSV Files

In pairs,

1. Write a program that will count female specimens in the CSV file.

2. Write a program that returns the fraction missing a collection date.

CSV Files

In pairs,

1. Write a program that will count female specimens in the CSV file.

2. Write a program that returns the fraction missing a collection date.

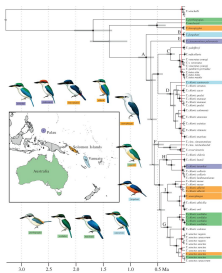3. Return a list of the species ("IDENTIFICATION" row) in this file.

Recap
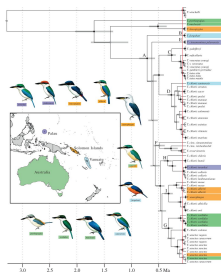
Anderson *et al.* 2014

- Install `anaconda` for lab today.
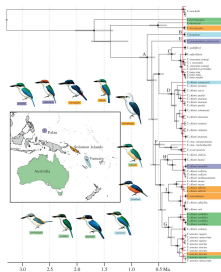
Recap

Anderson *et al.* 2014

- Install anaconda for lab today.
- Email lab reports to kstjohn@amnh.org

Recap

Anderson *et al.* 2014

- Install anaconda for lab today.
- Email lab reports to kstjohn@amnh.org
- Challenges available at rosalind.info