Regular Expressions

Pattern and Match objects

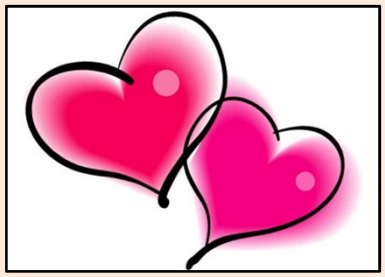
Genome 559: Introduction to Statistical and Computational Genomics

Elhanan Borenstein

A quick review

- Strings: ‘abc’ vs. “abc” vs. ‘’’ abc’’’ vs. r’abc’
- String manipulation is doable but tedious

- **Regular expressions (RE):**
  - A tiny language dedicated to string manipulation
  - It’s all about finding a good match
  - `re.findall(<regexe>, <string>)`



- **RE Basics:**
  - letters and numbers match themselves
  - Use predefined sets (e.g., \d, \W) or define youself ([a-c])
  - ^ $ \b \B allows you to match string/word boundaries
  - * + {n,m} allows you to define the number of repetitions
- Matching is greedy (trying to find the longest match)

RE Quick Reference

MATCHING CHARACTER SETS

- Most letters and numbers match themselves
- `[abc]` means either ”a” , ”b” , or “c”
- `[a-d]` means ”a”, ”b”, ”c”, or ”d”
- `[^a-d]` means anything but a, b, c or d
- `\d` matches any decimal digit (equivalent to `[0-9]`).
- `\D` matches any non-digit character (equivalent to `[^0-9]`).
- `\s` matches any whitespace character (equivalent to `[ \t\n\r\f\v]`).
- `\S` matches any non-whitespace character (equivalent to `[^ \t\n\r\f\v]`).
- `\w` matches any alphanumeric character (equivalent to `[a-zA-Z0-9_]`).
- `\W` matches any non-alphanumeric character (equivalent to the class `[^a-zA-Z0-9_]`.
- `.` matches **any** character (except newline)

MATCHING BOUNDARIES

- `^` matches the beginning of the string
- `$` matches the end of the string
- `\b` matches a word boundary
- `\B` matches position that is not a word boundary

REPETITION

- ***** : The previous character can repeat 0 or more times
- **+** : The previous character can repeat 1 or more times
- `A{1,3}` means at least one and no more than three A’s

SEMANTICS

- RS matches the concatenation of strings matched by R, S individually
- R|S matches the union (either R or S)

RE FUNCTIONS/PATTERN OBJECT METHODS

- **`re.findall(pat,str)`**
  Finds all (non-overlapping) matches
- **`re.match(pat,str)`**
  Matches only at the beginning of str
- **`re.search(pat,str)`**
  Matches anywhere in str
- **`re.split(pat,str)`**
  Splits str anywhere matches are found
- **`re.sub(pat,new_str,str)`**
  Substitutes matched patterns in str with new_str
- **`re.compile(pat)`**
  Compile a Pattern object

MATCH OBJECT METHODS

- **`group()`**:
  Returns the string that was matched
- **`group(i)`**:
  Returns the *i* sub-pattern that was matched
- **`groups()`**:
  Returns all sub-patterns that were matched as a list
- **`start()`**:
  Returns starting position of the match
- **`end()`** :
  Returns ending position of the match
- **`span()`** :
  Returns (start,end) as a tuple

What (else) can we do with RE

- `re.findall(pat,str)`
  - **finds all (nonoverlapping) matches**
- `re.match(pat,str)`
  - **matches only at the beginning of the string**
- `re.search(pat,str)`
  - **matches anywhere in the string**
- More soon to come (split, substitute,...)

What do these functions return

- `re.findall(pat,str)`
  - **finds all (nonoverlapping) matches**

If nothing was found:
returns an empty list

Otherwise:
returns a list of strings

- `re.match(pat, str)`
  - **matches only at front of string**
- `re.search(pat,str)`
  - **matches anywhere in string**

If nothing was found:
returns None

Otherwise:
returns a
"match" object

- More soon to come (split, substitute,...)

“Match” objects

- Objects designed specifically for the **re** module
- Retain information about exactly where the pattern matched, and how.
- Methods offered by a Match object:
  - `group()` : returns the string that matched
  - `start()` : returns the starting position of the match
  - `end()` : returns the ending position of the match
  - `span()` : returns (start,end) as a tuple

“Match” objects

```
>>> import re
>>> pat = r'\w+@\w+\.(com|org|net|edu)'
>>>
>>> my_match = re.search(pat, “this is not an email")
>>> print my_match
None
>>>
>>> my_match = re.search(pat, "my email is elbo@uw.edu")
>>> print my_match
<_sre.SRE_Match object at 0x895a0>
>>>
>>> my_match.group()
elbo@uw.edu
>>> my_match.start()
12
>>> my_match.end()
23
>>> my_match.span()
(12,23)
```

What got matched?

- We might want to extract information about what matched specific parts in the pattern (e.g., email name and domain)
- Extremely useful for extracting data fields from a formatted file

- We can parenthesize parts of the pattern and get information about what substring matched this part within the context of the overall match.

```
>>> pat = r`(\w+)@(\w+)\.+(com|org|net|edu)'
```

part 1 part 2 part 3

What got matched? Examples

```
>>> import re
>>> pat = r`(\w+)@(\w+)\.(com|org|net|edu)'
>>> my_match = re.search(pat, "my email is elbo@uw.edu")
>>>
>>> my_match.group()
elbo@uw.edu
>>> my_match.group(1)
elbo
>>> my_match.group(2)
uw
>>> my_match.group(3)
edu
>>> my_match.groups()
(`elbo','uw','edu')
```

```
>>> import re
>>> str = 'My birthday is 9/12/1988'
>>> pat = r'[bB]irth.* (\d{1,2})/(\d{1,2})/(\d{2,4})'
>>> match = re.search(pat,str)
>>> print match.groups()
(`9','12','1988')
```

Think how annoying and cumbersome it would be to code these yourself

What got matched? Labels

- You can even label the groups for convenience

```
>>> import re
>>> pat=r`(?P<name>\w+)@(?P<host>\w+)\.(?P<ext>com|org|net|edu)'
>>> my_match = re.search(pat, "my email is elbo@uw.edu")
>>>
>>> my_match.group(`name’)
elbo
>>> my_match.group(`host’)
uw
>>> my_match.group(`ext’)
edu
```

More **re** functions

- `re.split(pat,str)`
  - **Similar to the simple string split method, but can use patterns rather than single characters**

```
>>> import re
>>> re.split(r'chapter \d ', "chapter 1 This is … chapter 2 It was …")
['This is …', 'It was …']
```

```
>>> pat2 = r'(TAG|TAA|TGA)'
>>> re.split(pat2, my_DNA)
???
```

- `re.sub(pat,new_str,str)`
  - **Substitutes the matches pattern with a string**

```
>>> import re
>>> pat_clr = r'(blue|white|red)'
>>> re.sub(pat_clr, 'black', 'wear blue suit and a red tie')
'wear black suit and a black tie'
```

Cool substitution feature

- A very handy RE feature is the ability to use the sub-patterns you found as substitution strings.

```
>>> import re
>>> str = 'My birthday is 9/12/1988'
>>> pat = r'(\d{1,2})/(\d{1,2})/(\d{2,4})'
>>> match = re.search(pat,str)
>>> print match.groups()
('9','12','1988')
>>>
>>> rev_str = re.sub(pat,r'\2-\1-\3',str)
>>> print rev_str
'My birthday is 12-9-1988'
```

References to the sub-patterns found

Pattern objects and “compile”

- If you plan to use a pattern repeatedly, compile it to a **“Pattern” object**
- Working with a compiled Pattern object will speed up matching
- All the re functions will now work as **methods**.

```
>>> import re
>>> pat = r'\w+@\w+\.edu'
>>> pat_obj = re.compile(pat)
>>> pat_obj.findall("elbo@uw.edu and jht@uw.edu")
['elbo@uw.edu','jht@uw.edu']
>>>
>>> match_obj = pat_obj.search("my email is elbo@uw.edu")
```

Note: no need for a pattern as an argument

- Optional flags can further modify defaults, e.g., case-sensitive matching etc.

Sample problem #1

- Parse an enzymatic database file.
  - Download enzyme.txt from the course website.
  - In this file, some lines have the following format:
    Entry_code<some spaces>EC_number<some spaces>Category
    - Entry_code is always the string “ENTRY”
    - EC_number is a label that starts with “EC”, followed by a single space, followed by four 1-3 digit numbers separated by dots.
    - Category is a text descriptor (assume it can include several words).

    For example:

```
ENTRY      EC 2.4.1.130      Enzyme
ENTRY      EC 1.14.21.2      Obselete Enzyme
```

  - Read each line in the file and check whether it has this format. If so print it.

Solution #1

```
import re
import sys

file_name = sys.argv[1]
file = open(file_name,'r')

pat = r'ENTRY +EC \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3} +\b.*'
for line in file:
        line = line.strip()
        match_obj = re.match(pat,line)
        if match_obj != None:
                print line
```

```
ENTRY       EC 1.1.1.1                    Enzyme
ENTRY       EC 1.1.1.2                    Enzyme
ENTRY       EC 1.1.1.3                    Enzyme
ENTRY       EC 1.1.1.4                    Enzyme
ENTRY       EC 1.1.1.5        Obsolete    Enzyme
ENTRY       EC 1.1.1.6                    Enzyme
ENTRY       EC 1.1.1.7                    Enzyme
ENTRY       EC 1.1.1.8                    Enzyme
ENTRY       EC 1.1.1.9                    Enzyme
…
```

Sample problem #2

1. Using the same parsing process as in problem #1, now print only the EC_numbers you found.
   - Note: Print only EC_numbers that are part of lines that have the format described in problem #1. EC numbers appear in many other lines as well but those instances should not be printed.
   - Try using a single RE pattern

2. Now, print these EC numbers but include only the 1st and the 4th number elements
   (i.e., instead of EC 2.34.21.132, print EC 2.132)

Solution #2.1

```
import re
import sys

file_name = sys.argv[1]
file = open(file_name,'r')

pat = r'ENTRY +(EC \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}) +\b.*'
for line in file:
        line = line.strip()
        match_obj = re.match(pat,line)
        if match_obj != None:
                print match_obj.group(1)
```

```
EC 1.1.1.1
EC 1.1.1.2
EC 1.1.1.3
EC 1.1.1.4
EC 1.1.1.5
EC 1.1.1.6
EC 1.1.1.7
EC 1.1.1.8
EC 1.1.1.9
…
```

Solution #2.2

```
import re
import sys

file_name = sys.argv[1]
file = open(file_name,'r')

pat = r'ENTRY +EC (\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}) +\b.*'
for line in file:
    line = line.strip()
    match_obj = re.match(pat,line)
    if match_obj != None:
    print "EC " + match_obj.group(1) + "." + match_obj.group(4)
```

```
EC 1.1
EC 1.2
EC 1.3
EC 1.4
EC 1.5
EC 1.6
…
```

Problem #3

- "Translate" the first 100 lines of War and Peace to Pig Latin.
- The rules of translations are as follows:
  - If a word starts with a consonant: move it to the end and append "ay"
  - Else, for words that starts with a vowel, keep as is, but add "zay" at the end
  - Examples: beast → eastbay; dough → oughday; another→ anotherzay; if→ ifzay
- Hint: Remember the cool substitution trick we learned.