Regular Expressions

Genome 559: Introduction to Statistical and Computational Genomics

Elhanan Borenstein
A quick review: The super $Date$ class

class Date:
    def __init__(self, day, month):
        self.day = day
        self.month = month
    def __str__(self):
        day_str = '%s' % self.day
        mon_str = self.month
        return mon_str + "-" + day_str

birthday = Date(3, "Sep")
print "It's ", birthday, ". Happy Birthday!"

It's Sep-3. Happy Birthday!
Strings

- ‘abc’
- “abc”
- ”" abc"
- r’abc’
Newlines are a bit more complicated

- ‘abc\n’
- "abc\n"
- """abc"
- r'abc\n'
Why so many?

- ‘ vs “ lets you put the other kind inside a string. Very Useful.
- ” lets you run across multiple lines.
- All 3 let you include and show *invisible* characters (using \n, \t, etc.)
- r’...’ (*raw strings*) do not support invisible character, but avoid problems with backslash. Will become useful very soon.

```python
open('C:\new\text.dat') vs.
open('C:\\new\\text.dat') vs.
open(r'C:\new\text.dat')
```
String operations

- As you recall, the string data type supports a variety of operations:

```python
>>> my_str = 'tea for too'
>>> print my_str.replace('too','two')
'tea for two'

>>> print my_str.upper()
TEA FOR TOO

>>> my_str.split(' ')  
['tea', 'for', 'too']

>>> print my_str.find("o")
5
>>> print my_str.count("o")
3
```
But ...

- What if we want to do more complex things?
  - Get rid of all punctuation marks
  - Find all dates in a long text and convert them to a specific format
  - Delete duplicated words
  - Find all email addresses in a long text
  - Find everything that “looks” like a gene name in some output file
  - Split a string whenever a certain word (rather than a certain character) occurs
  - Find DNA motifs in a Fasta file
Well ...

- We can always write a program that does that ...

```python
# assume we have a genome sequence in string variable myDNA
for index in range(0,len(myDNA)-20) :
    if (myDNA[index] == "A" or myDNA[index] == "G") and
        (myDNA[index+1] == "A" or myDNA[index+1] == "G") and
        (myDNA[index+2] == "A" or myDNA[index+2] == "G") and
        (myDNA[index+3] == "C") and
        (myDNA[index+4] == "A") and
    # and on and on and on!
    ...
        (myDNA[index+19] == "C" or myDNA[index+19] == "T") :
            print "Match found at ",index
            break
```
Regular expressions

- Regular expressions (a.k.a. RE, regexp, regexes, regex) are a highly specialized **text-matching tool**.

- They are extremely useful in searching and modifying (long) string

- Regex can be viewed as a tiny programming language embedded in Python and made available through the `re` module.

- [http://docs.python.org/library/re.html](http://docs.python.org/library/re.html)
Not only in Python

- REs are very widespread:
  - Unix utility “grep”
  - Perl
  - TextWrangler
  - TextPad
  - Python

- So, ... learning the “RE language” would serve you in many different environments as well.
Do you absolutely need regexes?

- No, everything they do, you could do yourself!

- BUT ... pattern-matching is:
  - Widely used (especially in bioinf applications)!
  - Tedious to program!
  - Error-prone!

- RE give you a flexible, systematic, compact, and automatic way to do it.
  (In truth, it’s still somewhat error-prone, but in a different way).
RE is It’s all about finding a great match

- Using this RE tiny language, you can specify patterns that you want to match

- You can then ask match questions such as:
  - “Does this string match this pattern?”
  - “Is there a match to this pattern anywhere in this string?”
  - “What are all the matches to this pattern in this string?”

- You can also use REs to modify a string
  - Replace parts of a string (sub) that match the pattern with something else
  - Break strings into smaller pieces (split) wherever this pattern is matched
A simple example

- Consider the following example:

```python
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
```

This RE means: A word that starts with ‘f’ followed by any number of alphabetical characters

- Note the `re.` prefix – `findall` is a function in the `re` module

- `findall`:
  - Format: `findall(<regexe>, <string>)`
  - Returns a list of all non-overlapping substrings that matches the `regexe`.

- REs are provided as strings.
Remember:
It’s all about matching

Regular expressions are patterns;
they “match” sequences of characters
Basic RE matching

- Most letters and numbers match themselves
  - For example, the regular expression `test` will match the string `test` exactly
  - Normally case sensitive

```python
>>> re.findall(r'test', "Tests are testers' best testimonials")
[‘test’, ‘test’]
```

- Most punctuation marks have special meanings!
  - Metacharacters: . ^ $ * + ? { [ ] \ | ( )
  - needs to be escaped by backslash (e.g., “\.” instead of “.”) to get non-special behavior
  - Therefore, “raw” string literals (r’C:\new.txt’) are generally recommended for regexes (unless you double your backslashes judiciously)
Sets

- **Square brackets** mean that any of the listed characters will do (matching one of several alternatives)
  - \([a\ b\ c]\) means either "a", "b", or "c"

- You can also give a range:
  - \([a-d]\) means "a", "b", "c", or "d"

- **Negation**: caret means **not**
  - \[^a-d]\) means anything but a, b, c or d
  - \[^5]\) means anything but 5

- **Metacharacters** are not active inside sets.
  - \([a\ k\ $]\) will match “a”, “k”, or “$”. Normally, “$” is a metacharacter. Inside a set it’s stripped of its special nature.
Predefined sets

- \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 \([^a–zA–Z0–9_]\)).

Note the pairs. Easy to remember!
Matching boundaries

- \(^\) matches the beginning of the string
- \(\$\) matches the end of the string

- \(\backslash b\) matches a word boundary
- \(\backslash B\) matches position that is not a word boundary

(A word boundary is a position that changes from a word character to a non-word character, or vice versa).

For example, \(\backslash b\text{cat}\) will match \textit{catalyst} but not \textit{location}
Wildcards

- . matches **any** character (except newline)
- If you really mean “.” you must use a backslash

**WARNING:**
- backslash is special in Python strings
- It’s special again in RE
- This means you need too many backslashes
- Use ”raw strings” to make things simpler

What does this RE means: `r'\d\d'`?
Repetitions

- Allows you to specify that a portion of the RE must/can be repeated a certain number of times.

- * : The previous character can repeat 0 or more times
  - ca*t matches "ct", "cat", "caat", "caaat" etc.

- + : The previous character can repeat 1 or more times
  - ca+t matches "cat", "caat" etc. but not "ct"

- Braces provide a more detailed way to indicate repeats
  - A{1, 3} means at least one and no more than three A’s
  - A{4, 4} means exactly four A’s
A quick example

- Remember this PSSM:

```python
re.findall(r'[^AG]{3,3}CATG[^TC]{4,4}[AG]{2,2}C[^AT]TG[^CT][CG][TC]', myDNA)
```
More examples

```python
>>> re.sub('d', 'x', 'a_b - 12')
'a_b - xx'
>>> re.sub('D', 'x', 'a_b - 12')
'xxxxxxx12'
>>> re.sub('s', 'x', 'a_b - 12')
'a_bx-x12'
>>> re.sub('S', 'x', 'a_b - 12')
'xxx x xx'
>>> re.sub('w', 'x', 'a_b - 12')
'xxx - xx'
>>> re.sub('W', 'x', 'a_b - 12')
'a_bxxx12'
>>> re.sub('^', 'x', 'a_b - 12')
'xa_b - 12'
>>> re.sub('$', 'x', 'a_b - 12')
'a_b - 12x'
>>> re.sub('b', 'x', 'a_b - 12')
'a_b - 12'
>>> re.sub('B', 'x', 'a_b - 12')
'xa_bx - x12x'
>>> re.sub(r'\b', 'x', 'a_b - 12')
'xa_bx - x12x'
>>> re.sub('B', 'x', 'a_b - 12')
'ax_xb x-x 1x2'
```
RE Semantics

- If $R, S$ are regexes:
  - $RS$ matches the concatenation of strings matched by $R, S$ individually
  - $R | S$ matches the union (either $R$ or $S$)
  - $this | that$ matches ‘this’ and ‘that’, but not ‘thisthat’.

- Parentheses can be used for grouping
  - $(abc) +$ matches ‘abc’, ‘abcabc’, ‘abcabcabc’, etc.
Conflicts?

- Check this example:

```python
>>> import re
>>> mystring = "This contains 2 files, hw3.py and uppercase.py."
>>> all_matches = re.findall(r'.+\..+\..+', mystring)
>>> print(all_matches)
['This contains 2 files, hw3.py and uppercase.py']
```

- What do you think all_matches contains?

What happened?
Matching is greedy

- Our RE matches “hw3.py”
- Unfortunately ...
  - It also matches: “This contains 2 files, hw3.py”
  - And it even matches: “This contains 2 files, hw3.py and uppercase.py”
- Python will choose the longest match!
- Solution:
  - Break my text first into words (not an ideal solution)
  - I could specify that no spaces are allowed in my match

```python
>>> import re
>>> mystring = "This contains 2 files, hw3.py and uppercase.py."
>>> all_matches = re.findall(r'.+\..py', mystring)
>>> print(all_matches)
[’This contains 2 files, hw3.py and uppercase.py’]```
A better version

- This will work:

```python
>>> import re
>>> mystring = "This contains 2 files, hw3.py and uppercase.py."
>>> all_matches = re.findall(r'[^ ]+\.py', mystring)
>>> print all_matches
['hw3.py','uppercase.py']
```

```
r".+\.py"  "Two files: hw3.py and uppercase.py."

r"\w+\.py"  "Two files: hw3.py and UPPER.py."
```
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

- And also split and substitute...
What do these functions return

- `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

And also split and substitute...
For more on RegExp
(and to learn how to use RegExp to edit strings)
see next presentation !!
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
The regular expression language is relatively small and restricted

- Not all possible string processing tasks can be done using regular expressions.
- Some tasks can be done with RE, but the expressions turn out to be extremely complicated.

In these cases, you may be better off writing a Python code to do the processing:

- Python code may take longer to write
- It will be slower than an elaborate regular expression
- But ... it will also probably be more understandable.
Suppose you are not sure:

- whether the format you are using for a certain command is the correct one
- or whether `range(4)` returns 0 to 4 or 0 to 3
- or whether string has a method “reverse”
- or whether you are allowed to break inside a nested loop

What should you do?
Code like a pro ...  

- **JUST RUN IT!!!**
- Don’t be afraid:
  - Running a bugged code will not harm your computer!
  - (it also should not hurt your self-esteem)
  - It doesn’t cost anything
  - It will be faster (and more accurate) than you trying to “think it through”
  - In many cases, the error message or output will be extremely informative

“The freedom to run experiments is the most precious luxury of computational biologists”  

Nanahle Nietsnerob
Sample problem #1

- Download the course webpage (e.g., use the “save as” option). Write a program that reads this webpage text and scan for all the email addresses in it.

- An email address usually follows these guidelines:
  - Upper or lower case letters or digits
  - Starting with a letter
  - Followed by a the “@” symbol
  - Followed by a string of alphanumeric characters. No spaces are allowed
  - Followed by a the dot “.” symbol
  - Followed by a domain extension. Assume domain extensions are always 3 alphanumeric characters long (e.g., “com”, “edu”, “net”).
import sys
import re

file_name = sys.argv[1]
file = open(file_name, "r")
text = file.read()

addresses = re.findall(r'[^A-Za-z0-9]*\w*\w+\w{3,3}', text)
print(addresses)

['jht@uw.edu', 'elbo@uw.edu']
Sample problem #2

1. Download and save warandpeace.txt. Write a program to read it line-by-line. Use re.findall to check whether the current line contains one or more “proper” names ending in “...ski”. If so, print these names:

```
['Bolkonski'],
['Bolkonski'],
['Bolkonski'],
['Bolkonski'],
['Volkonski'],
['Volkonski'],
['Volkonski'],
['Volkonski'],
```

2. Now, instead of printing these names for each line, insert them into a dictionary and just print all the “...ski” names that appear in the text at the end of your program (preferably sorted):

```
Aski
Bitski
Bolkonski
Borovitski
Bronnitski
Czartoryski
Golukhovski
Gruzinski
```
import sys
import re

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

names_dict = {}  # A dictionary for storing all names
for line in file:
    names = re.findall(r'\w+ski', line)
    if len(names) > 0:
        print names

file.close()
Solution #2.2

```python
import sys
import re

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

names_dict = {} # A dictionary for storing all names
for line in file:
    names = re.findall(r'\w+ski', line)
    for name in names:
        names_dict[name] = 1

file.close()

name_list = names_dict.keys()
name_list.sort()

for name in name_list:
    print name
```
Challenge problem

- “Translate” 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
  - happy → appyhay
  - another → anotherzay
  - if → ifzay