Bioinformatics Programming Class 6

Goal
Today you will learn the 'fine art' of creating aesthetically beautiful (and practical) regular expressions for text pattern matching.

Introduction
Pattern matching is a basic technique, essential in many fields, especially in bioinformatics. Soon you will be asked to parse text files (e.g. pdb, fasta), find amino acid patterns over one or more sequences, filtering large amount of data, etc. Parsing files is boring, but a necessary evil. Learn how to efficiently perform pattern matching will save you hours of work that you could invest in solving real scientific problems. Before we start talking about regular expressions let's travel in time, back in 1950, before Stephen Cole Kleene defined the regular sets and with them laid the groundwork for implementing the regular expressions.

Q1: Without using the power of regular expressions, your task is to write a short python script to print all the lines in the file 'alignment.txt' that contain the letter 'A' followed by one or more '-'. Which lines does your script print?

Hint: you can look up for a fixed pattern in Python using the function find.

```python
>>> 'last'.find('st')
2
>>> 'last'.find('abc')
-1
```

Python is powerful and handy language. In fact you could have accomplished your task in just 3 lines. Now imagine that an evil TA would ask you to rewrite your script to print the lines that contain an 'A' followed by any number of '-' , followed by exactly 1 'B'.

How would you proceed? It is not so trivial anymore. I don't know how many lines it would take to write such script, but my guess is that you don't want to know it either. Yes, it gets complex and boring very soon.

Luckily it's 2011 and regular expressions have been implemented in Python.
Regular Expression

A regular expression (or RE) specifies a set of strings that matches it; to start using regular expressions you need to import the appropriate module.

```python
>>> import re
```

The functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

As far as we are concerned in this course, a regular expression is a string that can contain both special and ordinary characters. Most ordinary characters, like 'A', 'a', or '0', are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string 'last'.

```python
>>> if re.search('last', 'last but not least'): print 'found!

found
```

Special characters have (surprise, surprise!) special meanings and they are: . (the dot), ^, $, (,), [, ], {, }, +, ?, *, \, \. If you want to use one of this special characters as ordinary character, (e.g. to match the word last?) you need to escape the special character prepending '\'.

```python
>>> if re.search('last\?', 'Are you the last?'): print 'found!

found!
```

Here follows a brief and (very) incomplete introduction to the special characters and their meanings. A more gentle and comprehensive introduction is available at http://docs.python.org/library/re.html.

- `'.'` matches any character except a newline.
  ```python
  >>> if re.search('.l.st', 'last but not least'): print 'found!
  
  found!
  ```
- `'*` causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. `ab*` will match 'a', 'ab', or 'a' followed by any number of 'b's.
- `'+' causes the resulting RE to match 1 or more repetitions of the preceding RE. `ab+` will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.
- `'?` causes the resulting RE to match 0 or 1 repetitions of the preceding RE. `ab?` will match either 'a' or 'ab'.
- `{m}` specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, `a{6}` will match exactly six 'a' characters, but not five.
- `{m,n}` causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, `a{3,5}` will match from 3 to 5 'a' characters. Omitting m specifies a lower bound of zero, and omitting n specifies an
infinite upper bound. As an example, $a\{4,\}b$ will match `aaaab` or a thousand 'a' characters followed by a `b`, but not `aaab`.

- `[]` are used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a `-`. Special characters are not active inside sets. For example, `[akm$]` will match any of the characters 'a', 'k', 'm', or '$'; `[a-z]` will match any lowercase letter, and `[a-zA-Z0-9]` matches any letter or digit.

**Q2:** Rewrite the script of Q1, this time using RE.

**Q3:** Write a regular expression that matches the following description:

- 1 or more 'A' followed by
- exactly 0 or 1 'T' followed by
- exactly 1 question mark '?'

The resulting RE should be able to match 'WHAAAAAAAAT?' or 'WHAAAAA?', but not 'WHAAATT?'

**Q4:** Write a regular expression that matches the following description:

- A or G followed by
- a single A, C, G, or T followed by
- 0 or more '-' followed by
- any combination of 5 nucleotides, but T

The resulting RE should be able to match 'AAAAAAA' or 'GA------GACAA', etc...

As you might have noticed, multiple RE can match the same strings. It is up to your aesthetic sense to build the most beautiful (i.e. general and synthetic) ones.

Another useful function is `split`, which splits the string by the occurrences of pattern.

```python
>>> re.split('A+B+', '1AB2AAAAAAAAB3ABBBBBB4')
[ '1', '2', '3', '4' ]
```

The '*', '+', and '?' qualifiers are all greedy; they match as much text as possible! Sometimes this behaviour isn’t desired. Adding '?' after the qualifier makes it perform the match in non-greedy or minimal fashion.

**Q5:** If you split the string '1AB2AAAAAAAAB3ABBBBBB4' using the regular expression 'A.*B', which list do you expect as result? Why?

**Q6:** Write a regular expression that split the string '1ASD2ASCIIASSSSSSSD3D4' in the list ['1', '2', '3', '4'].

**Q7:** Write a regular expression that split the string '1AABS2ACCCCSSSSS3AS4' in the list ['1', '2', '3', '4'].

```python
>>> re.split('A+B+', '1AB2AAAAAAAAB3ABBBBBB4')
[ '1', '2', '3', '4' ]
```
**Q8:** Write a regular expression that split the string '15Z2Z.53-4' in the list ['1', '2', '3', '4'].

One special operator we haven’t discussed so far is (). () matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed. If a group matches multiple times, only the last match is accessible!

```python
>>> match = re.search('A([0-9]+)A','A123A')
>>> match.group(0)
'A123A'

Group 0 always return the full regular expression matched.

```python
>>> match.group(1)
'123'

Group n returns the substring that matches the n-th group

```python
>>> re.search('A([0-9]+)','A123A456A').group(1)
'123'

The search function matches only the first occurrence of the pattern. The findall function instead returns all non-overlapping matches (or groups if any) of pattern in string, as a list of strings.

```python
>>> re.findall('A([0-9]+)','A123A456A')
['123', '456']
```

**Q9:** What is the difference between the two RE ‘abab?c’ and ‘ab(ab)?c’? Do they match the same strings? If not, give a counter example (i.e. a string which matches just one of the two).

**Q10:** Modify the script of Q2 in order to print the first character after the gaps following an A. For example, for the string ‘XXXA--------BYYY’ your script should print B. (It can be 4 lines long!)