CHAPTER 9

Pattern Matching and Text Mining

Learning goal: You can find patterns in sequences and natural language texts.

9.1 IN THIS CHAPTER YOU WILL LEARN

- How to express a consensus sequence using a regular expression
- How to use Python regular expression tools to search substrings in strings
- How to search for the occurrence of a functional motif in a protein sequence
- How to search a given word or a set of words in a text (e.g., scientific abstracts)
- How to identify motifs in nucleotide sequences (e.g., transcription factor or miRNA binding sites)

9.2 STORY: SEARCH A PHOSPHORYLATION MOTIF IN A PROTEIN SEQUENCE

9.2.1 Problem Description

A sequence functional motif is defined as a short amino acid or nucleotide sequence containing one or more residues involved in a function. Phosphorylation sites, mannosylation sites, recognition motifs, glycosylation sites, transcription binding sites, etc. are examples of functional motifs. Sequence functional motifs can be represented using a special notation called *regular expressions*. A regular expression, sometimes called *regexp*, is a string syntax composed of characters and metacharacters that represent *sets* of strings. In other words, if you want to represent several strings with a single expression, it is necessary to introduce rules and symbols that allow "meta" meanings like wildcards, repeated characters, and logical groups. An example often used in biology is the character *N* in DNA sequences. The sequence AGNNT could be AGAAT, AGCTT, AGGGT, or one of many other alternatives. Regular expressions work in a similar way but use a more complex set of special characters.

Suppose you want to represent, with a single expression, the following peptide strings: "AFL," "GFI," "AYI," "GWI," "GFI," "AWI," "GWL," and "GYL." If you use a symbolic expression such as "[AG]" to indicate that in a position of a string you might find either "A" or "G," you can use the expression "[AG][FYW][IL]" to represent all of the peptides above.

Notice that we are using not the literal meaning of "[" and "]" but a "meta" meaning. In this case, "[" and "]" are called *metacharacters*, and the expression encoding a set of strings, through the use of characters and metacharacters, is called a *regular expression*.

Another example is represented by functional motifs, which are usually short and may contain invariable and variable positions. For instance, you can represent a Ser/Thr phophorylation motif as [ST]Q. This expression, when searched in a protein sequence, will have a hit in correspondence of two different subsequences: "SQ" and "TQ," i.e., either a serine or a threonine followed by a glutamine. The first position of the motif is variable, while the second position is conserved. There are several publicly available resources dedicated to functional motifs (e.g., ELM: http://elm.eu.org; PROSITE: http://prosite.expasy.org/; etc.). Searching occurrences of a functional motif in a protein sequence, or in a data set of protein sequences, can be used as a procedure to infer the function of proteins. This is exactly what tools such as ScanProsite (http://prosite.expasy.org/scanprosite/) do.

The following program simulates one of the functions of ScanProsite; i.e., the program searches for a phosphorylation motif in a protein sequence, and returns the first occurrence of the motif.

9.2.2 Example Python Session

```
import re
seq = 'VSVLTMFRYAGWLDRLYMLVGTQLAAIIHGVALPLMMLI'
pattern = re.compile('[ST]Q')
match = pattern.search(seq)
if match:
    print '%10s' % (seq[match.start() - 4: match.end() + 4])
    print '%6s' % match.group()
else:
    print "no match"
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

9.3 WHAT DO THE COMMANDS MEAN?

In the first line of the program, a module called `re` is imported. The `re` module provides metacharacters, rules, and functionalities to write and interpret regular expressions and match them to string variables. You can find an exhaustive tutorial on Python regular expressions by A.M. Kuchling at http://docs.activestate.com/activepython/2.5/python/regex/regex.html.

In the line

```
pattern = re.compile('[ST]Q')
```

the phosphorylation motif to be searched, in the form of the string `'[ST]Q'`, is converted into a new object by the `compile()` method of the `re` module. The conversion is mandatory, otherwise characters such as `"["` would be interpreted as a simple square bracket and not as a metacharacter with a precise meaning. The `re.compile()` function returns a *regular expression object*. The `re` module provides methods to handle regular expression objects.

9.3.1 Compiling Regular Expressions

`compile()` is the method to compile a string and convert it into a regular expression object (the `RegexpObject`).

```
>>> import re
>>> regexp = re.compile('[ST]Q')
>>> regexp
<_sre.SRE_Pattern object at 0x22de0>
```

The string can also be recorded in a variable:

```
>>> motif = '[ST]Q'
>>> regexp = re.compile(motif)
```

It is possible to pass arguments (compilation flags) to the `compile()` method, thus modifying some aspects of how regular expressions work. For instance, you can ignore uppercase and lowercase by

```
>>> regexp = re.compile(motif, re.IGNORECASE)
```

See Appendix A, Section A.2.17, subsection "Regular Expression Compilation Flags."

In the Python session in Section 9.2.2, the compiled pattern is searched in the sequence stored in the variable `seq` and printed to the screen using a number of methods: `search()`, `group()`, `start()`, and `end()`. We will discuss now what these methods do and what they return.

9.3.2 Pattern Matching

Once your regular expression is compiled, and you have a `RegexpObject`, you can search for its matches in a string using `RegexpObject` methods.

`RegexpObject` methods return `Match objects`, the content of which can be extracted using `Match object` methods. See Appendix A, Section A.2.17, subsection "`re` Module Methods." This is not conceptually different from file reading: when you want to access the content of a file, you first have to open the file, thus creating a "file object," then you have to use methods of file objects (e.g., `read()`, `readline()`, etc.) to read the content of the file.

The `search()` function scans a string, looking for a location where the regular expression matches for the first time. This means that the `search()` method will return at most one single match object per sequence. We use the `group()` method of match objects to print the first match:

```
>>> motif = 'R.[ST][^P]'
>>> regexp = re.compile(motif)
>>> print regexp
<_sre.SRE_Pattern object at 0x57b00>
>>> seq = 'RQSAMGSNKSKPKDASQRRRSLEPAENVHGAGGGAFPASQRPSKP'
>>> match = regexp.search(seq)
>>> match
<_sre.SRE_Match object at 0x706e8>
>>> match.group()
'RQSA'
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

The regular expression `'R.[ST][^P]'` will match a substring with an arginine (`R`) in the first position, any amino acid in the second position (.), either a serine or a threonine in the third position (`[ST]`), and any amino acid but a proline in the last position (`[^P]`). Notice that the `search()` method returns not the matching substring directly but a `Match object`, which encodes the matching substring and its start and end positions along the sequence. This information can be retrieved using the following methods of a match object:

- `match.group()` returns the matching substring.
- `match.span()` returns a tuple containing the (start, end) positions of the match.
- `match.start()` returns the start position of the match.
- `match.end()` returns the end position of the match.

Should you only be interested in finding the regular expression matching at the first position of a sequence, you can use the method `match()`.

Here, we propose the previous example, using `match()` instead of `search()`:

```
>>> match1 = regexp.match(seq)
>>> match1
<_sre.SRE_Match object at 0x70020>
>>> match1.group()
'RQSA'
```

Notice that the `match` and `match1` variables have identical values in this specific case.

In summary, both the `search()` and the `match()` methods return a `Match object`, which can be assigned to a variable in order to use its content through the `Match object` methods `group()`, `span()`, `start()`, and `end()`:

```
>>> match1.span()
(0, 4)
>>> match1.start()
0
>>> match1.end()
4
```

Notice that UNIX/Linux provides a command to search for regular expressions matches in files (see Box 9.2).

Q & A: WHAT IF I WANT TO FIND ALL MATCHES OF A REGULAR EXPRESSION IN A STRING AND NOT ONLY THE FIRST ONE?

The `re` module offers two methods for this purpose: `findall()`, which returns a list containing all the matching substrings, and `finditer()`, which finds all the `Match objects` corresponding to the regular expression matches and returns them in the form of an *iterator*. More generally, an iterator is a "container" of objects that can be traversed in Python using a `for` loop. In this specific case, the iterator contains a set of `Match objects`, which can be individually accessed using `Match object` methods, such as `group()`, `span()`, `start()`, and `end()`:

```
>>> all = regexp.findall(seq)
>>> all
['RQSA', 'RRSL', 'RPSK']
>>> iter = regexp.finditer(seq)
>>> iter
<callable-iterator object at 0x786d0>
>>> for s in iter:
...   print s.group()
...   print s.span()
...   print s.start()
...   print s.end()
...
RQSA
(0, 4)
0
4
RRSL
(18, 22)
18
22
RPSK
(40, 44)
40
44
```

9.3.3 Grouping

It is possible to divide a regular expression in subgroups, each matching a different component of interest. Suppose that, in the previous example, you wanted to know what amino acid type is matched by the ".". We can create a group delimiting the "." with round brackets and then get the matching amino acid type using the `group()` method, as follows:

```
import re
seq = 'QSAMGSNKSKPKDASQRRRSLEPAENVHGAGGGAFPASQRPSKP'
pattern1 = re.compile('R(.)[ST][^P]')
match1 = pattern1.search(seq)
print match1.group()
print match1.group(1)
pattern2 = re.compile('R(.{0,3})[ST][^P]')
match2 = pattern2.search(seq)
print match2.group()
print match2.group(1)
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

The output of the program is:

```
RRSL
R
RRRSL
RR
```

The `group()` method with no argument or the argument equal to 0 always returns the complete matching substring, whereas subgroups are numbered from left to right in increasing order (starting with 1).

Notice that subgroups could be nested, and to know the corresponding number, you have to count the number of open round brackets from left to right.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

You can also pass multiple arguments to the `group()` method. In this case, it will return a tuple containing the values for the corresponding groups:

```
>>> m.group(2, 1, 2)
('b', 'abc', 'b')
```

Finally, the `groups()` method returns a tuple with the substrings corresponding to all subgroups:

```
>>> m.groups()
('abc', 'b')
```

It is also possible to assign a name to each subgroup in order to selectively retrieve its content. For example, you can label a first group of a regular expression with the name `w1` and a second group with `w2` and later retrieve the identity of the match of each group using the `group()` function with the group name (`w1` or `w2`) as argument:

```
>>> pattern = 'R(?P<w1>.{0,3})[ST](?P<w2>[^P])'
>>> regexp = re.compile(pattern)
>>> m1 = regexp.search(seq)
>>> m1.group('w1')
'RR'
>>> m1.group('w2')
'L'
```

The group label must be put between < and > symbols (i.e., `<name>`) and inserted in the round brackets of the group preceded by `?P` (i.e., `(?P<name>...)`). A group is selectively accessible passing the label to the `group()` function as an argument (`group(<name>)`).

9.3.4 Modifying Strings

The `re` module also provides three methods that allow modifying strings: `split(s)`, `sub(r, s, [c])`, and `subn(r, s, [c])`.

The method `split(s)` splits the string `s` at the matches of a regular expression. In the following example, a string will be split at all "|" symbols. Notice that the character "|" is also a metacharacter in the regular expression syntax (see Box 9.1). To tell Python to interpret it as a normal character, you have to put a backslash ("\") before the metacharacter. This is a general rule to make Python distinguish metacharacters from normal characters.

```
import re
separator = re.compile('\|')
annotation = 'ATOM:CA|RES:ALA|CHAIN:B|NUMRES:166'
columns = separator.split(annotation)
print columns
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

BOX 9.1 CHARACTERS AND METACHARACTERS

Not every character in a regular expression is a metacharacter. The regular expression metacharacters are

```
[] ^ $ \ . | * + ? { } ( )
```

`[]`

Square brackets are used to indicate a class of characters. For example, if you search a match of `[abc]` in a string `s`, you will find it if `s` contains `'a'`, `'b'`, or `'c'`.

In particular, `[a-z]` matches the class of alphabet characters from `a` to `z`, whereas `[0-9]` matches the integers between 0 and 9.

`^`

`[^a]` indicates the complement of `a`, i.e., every character different from `a`. `^a` (not enclosed in square brackets) indicates that a match exists in `s` only if `a` is in the first position of `s`.

`$`

`a $` indicates that a match exists in `s` only if `a` is in the last position of `s`.

`\`

The meaning of \ depends on whether \ is followed by a metacharacter or a character. In the first case, it "protects" the metacharacter by restoring its literal meaning; in the second case, its meaning depends on the character that follows.

- `\d` corresponds to `[0-9]`;
- `\D` corresponds to `[^0-9]`;
- `\s` corresponds to `[\t\n\r\f\v]`, i.e., any whitespace character
- `\S` corresponds to `[^\t\n\r\f\v]`, i.e., any character that is not a whitespace
- `\w` corresponds to `[a-zA-Z0-9]`, i.e., any alphanumeric character
- `\W` corresponds to `[^a-zA-Z0-9]`, i.e., any character that is not alphanumeric.

`.`

This corresponds to any character except the newline character.

`|`

This is the `OR` operator. If placed between two regular expressions, matches will be searched either with the regexp on its left or with the one on its right.

`( )`

Round brackets are used to create subgroups in a regular expression.

Repetitions: `* + ? { }`

These metacharacters are used to find a match with repeated things.

`*`	The preceding character can be matched zero or more times. `a*bc` will match "`bc`," "`abc`," "`aabc`," "`aaabc`," "`aaaabc`," etc.
`+`	The preceding character can be matched one or more times. `a+bc` will match "`abc`," "`aabc`," "`aaabc`," "`aaaabc`," etc. but not "`bc`."
`?`	The preceding character can be matched zero times or once. `can-?can` will match both "`can-can`" and "`cancan`."
`{m,n}`	This qualifier means that at least *m* and at most *n* repetitions of the preceding character will be matched.

BOX 9.2 GREP: THE UNIX/LINUX COMMAND TO SEARCH WORDS IN FILES

`grep` is the UNIX/Linux command for searching text files for lines matching a regular expression.

```
grep ArticleTitle PMID.html
```

will return all the lines of the `PMID.html` text file matching the word `ArticleTitle`.

You can use the asterisk to indicate any character:

```
grep Ar*le mytext.txt
```

will return all the `mytext.txt` lines having at least a word starting with `Ar` and ending with `le`.

You can use more complicated regular expressions. For example,

```
grep ^'>' 3G5U.fasta
```

will return all the lines of the file `3GU.fasta` starting with the character `'>'`. In this case you have to use quotation marks because `'>'` is a metacharacter in UNIX/Linux (it is the redirection character).

This code will produce a list with the split elements from the `annotation` string:

```
['ATOM:CA', 'RES:ALA', 'CHAIN:B', 'NUMRES:166']
```

The `RegexpObject` method `sub(r, s, [c])` returns a new string where nonoverlapping occurrences of a given pattern in the `s` string are all replaced with the value of `r` (if the optional argument `c` is not specified). In the following example, the pattern is '\|' (encoded in the `separator RegexpObject`), and it will be replaced by `'@'` in the `s` string:

```
import re
separator = re.compile('\|')
annotation = 'ATOM:CA|RES:ALA|CHAIN:B|NUMRES:166'
new_annotation = separator.sub('@', annotation)
print new_annotation
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

This results in:

```
ATOM:CA@RES:ALA@CHAIN:B@NUMRES:166
```

If the pattern is not found in the `s` string, `s` is returned unchanged. The optional argument `c` is the maximum number of pattern occurrences to be replaced; `c` must be a nonnegative integer. For example, set `c` to `2` in the previous example:

```
new_annotation = separator.sub('@', annotation, 2)
print new_annotation
```

Only the first two separators are replaced:

```
ATOM:CA@RES:ALA@CHAIN:B|NUMRES:166
```

The method `subn(r, s, [c])` does the same but returns a tuple of two elements, where the first element is the new string and the second is the number of replacements that were performed:

```
new_annotation = separator.subn('@', annotation)
print new_annotation
```

which results in:

```
('ATOM:CA@RES:ALA@CHAIN:B@NUMRES:166', 3)
```

9.4 EXAMPLES

Example 9.1 How to Convert a PROSITE Regular Expression into a Python Regular Expression

PROSITE (http://prosite.expasy.org/) is a resource for protein domains, families, and functional sites, which are described by either signature patterns (i.e., regular expressions) or profiles (i.e., tables of position-specific amino acid weights and gap costs). The regular expression syntax used in PROSITE (see http://prosite.expasy.org/scanprosite/scanprosite-doc.html#pattern_syntax), however, is different from the one used in Python. It can be very useful to be able to automatically convert one into the other.

The following simple script performs this task:

```
pattern = '[DEQN]-x-[DEQN](2)-C-x(3,14)-C-x(3,7)\
    -C-x-[DN]-x(4)-[FY]-x-C'
pattern = pattern.replace('{', '[^')
pattern = pattern.replace('}', ']')
pattern = pattern.replace('(', '{')
pattern = pattern.replace(')', '}')
pattern = pattern.replace('-', '')
pattern = pattern.replace('x', '.')
pattern = pattern.replace('>', '$')
pattern = pattern.replace('<', '^')
print pattern
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

The PROSITE regular expression in the example corresponds to the calcium-binding EGF-like domain signature (PROSITE ID: EGF_CA; PROSITE AC: PS01187).

Example 9.2 How to Find Transcription Factor Binding Sites in a Genomic Sequence

Suppose you have a list of transcription factor binding sites (TFBSs) and want to find out if and where they occur in the genome of a given organism. This can be easily done using Python regular expression tools. You need the text file with the nucleotide sequence of the genome you want to search and the list of TFBSs in a format that can be read by a computer program. For example, the Transcription Factor Database (http://cmgm.stanford.edu/help/manual/databases/tfd.html) uses the following format:

```
UAS(G)-pMH100 CGGAGTACTGTCCTCCG ! J Mol Biol 209: 423-32 (1989)
TFIIIC-Xls-50 TGGATGGGAG ! EMBO J 6: 3057-63 (1987)
HSE_CS_inver0 CTNGAANNTTCNAG ! Cell 30: 517-28 (1982)
ZDNA_CS 0 GCGTGTGCA ! Nature 303: 674-9 (1983)
GCN4-his3-180 ATGACTCAT ! Science 234: 451-7 (1986)
```

In this example, the `'TFBS.txt'` file contains this list of TFBSs, and the `'genome.txt'` file contains the sequence in FASTA format of, e.g., a whole chromosome of a eukaryotic organism of your choice.

```
import re
genome_seq = open('genome.txt').read()
# read transcription factor binding site patterns
sites = []
for line in open('TFBS.txt'):
    fields = line.split()
    tf = fields[0]
    site = fields[1]
    sites.append((tf, site))

# match all TF's to the genome and print matches
for tf, site in sites:
    tfbs_regexp = re.compile(site)
    all_matches = tfbs_regexp.findall(genome_seq)
    matches = tfbs_regexp.finditer(genome_seq)
    if all_matches:
        print tf, ':'
        for tfbs in matches:
           print '\t', tfbs.group(), tfbs.start(), tfbs.end()
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

Example 9.3 Extract the Title and the Abstract Text from a PubMed HTML Page

If you go to a PubMed abstract web page (e.g., http://www.ncbi.nlm.nih.gov/pubmed/18235848), you can easily access the corresponding HTML source code. For example, this can be done through the "Develop" → "Show Page Source" link in the Safari menu or "Tools" → "Web Developer" → "Page Source" in the Firefox menu. Spend a few minutes exploring this page. You will see that the title of the paper is enclosed between the tags `<h1>` and `</h1>`, whereas the text of the abstract is enclosed between `<h3>Abstract</h3><div class = ""><p>` and `</p>`. These details are relevant for the selective extraction of the title and the abstract from a PubMed HTML abstract page.

The example script opens the HTML web page from a Python script and parses it in order to selectively fetch some parts of it (the title and the abstract in this case).

```
import urllib2

import re
pmid = '18235848'
url = 'http://www.ncbi.nlm.nih.gov/pubmed?term=%s' % pmid
handler = urllib2.urlopen(url)
html = handler.read()
title_regexp = re.compile('<h1>.{5,400}</h1>')
title_text = title_regexp.search(html)
abstract_regexp = re.compile('<h3>Abstract</h3><div class\
    = ""><p>.{20,3000}</p></div>')
abstract_text = abstract_regexp.search(html)
print 'TITLE:', title_text.group()
print 'ABSTRACT:', abstract_text.group()
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

The `urllib2` module (see Recipe 13) provides tools to connect to a URL and retrieve its content. `urlopen()` is the method for URL opening (i.e., the method that establishes the connection). Its argument must be a URL, and, similarly to the `open()` built-in function for file opening, it returns a file-type Python object (a `handler`), which, as such, owns a number of methods that can be used to read its content (`read()`, `readline()`, `readlines()`, `close()`). The `read()` method reads the `handler` content as a single string of text.

Once the HTML source code is stored in a variable (`html`) in the form of a single string, we can use the tools provided by the `re` module to parse it.

In this example, by having a look at the HTML at hand (you should save the HTML text to a file and manually examine it), you can identify `<h1>` and `</h1>` as unique delimiting tags of the title. Therefore, a regular expression is defined as follows:

```
<h1>.{5,400}</h1>
```

This regular expression will match any text between 5 and 400 characters delimited by `<h1>` and `</h1>`. This choice univocally identifies the title. It must be noticed that the number of characters allowed between the two tags must be adapted to the maximum number of characters that can occur in titles of scientific papers. A similar procedure is applied for the selection of the abstract.

If you want to recursively extract the title and abstract from a list of PMIDs, you have to put the code in a `for` loop:

```
pmids = ['18235848', '22607149', '22405002', '21630672']
for pmid in pmids:
    url = 'http://www.ncbi.nlm.nih.gov/pubmed?term=%s'+%pmid
     ...
```

The list of PMIDs can be read from a text file and stored in the Python list.

Example 9.4 Detect a Specific Word or a Set of Words in a Scientific Abstract

You can use what you learned in Example 9.3 to detect a word or a set of words in a scientific abstract. More generally, this example can be applied to perform very simple text mining and can be compared to the "find" tool available in Microsoft Word.

```
import urllib2
import re
# word to be searched
keyword = re.compile('schistosoma')
# list of PMIDs where we want to search the word
pmids = ['18235848', '22607149', '22405002', '21630672']
for pmid in pmids:
    url = 'http://www.ncbi.nlm.nih.gov/pubmed?term=%s' +%pmid
    handler = urllib2.urlopen(url)
    html = handler.read()
    title_regexp = re.compile('<h1>.{5,400}</h1>')
    title = title_regexp.search(html)
    title = title.group()
    abstract_regexp = re.compile('<h3>Abstract</h3><p>.\
        {20,3000}</p></div>')
    abstract = abstract_regexp.search(html)
    abstract = abstract.group()
    word = keyword.search(abstract, re.IGNORECASE)
    if word:
        # display title and where the keyword was found
        print title
        print word.group(), word.start(), word.end()
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

If you want to identify all the occurrences of a word in a text, you can use the `finditer()` method:

```
import urllib2
import re
# word to be searched
word_regexp = re.compile('schistosoma')
# list of PMIDs where we want to search the word
pmids = ['18235848', '22607149', '22405002', '21630672']
for pmid in pmids:
    url = 'http://www.ncbi.nlm.nih.gov/pubmed?term=%s' +%pmid
    handler = urllib2.urlopen(url)
    html = handler.read()
    title_regexp = re.compile('<h1>.{5,400}</h1>')
    title = title_regexp.search(html)
    title = title.group()
    abstract_regexp = re.compile('<h3>Abstract</h3><p>.\
        {20,3000}</p></div>')
    abstract = abstract_regexp.search(html)
    abstract = abstract.group()
    words = keyword.finditer(abstract)
    if words:
    # display title and where the keyword was found
    print title
    for word in words:
        print word.group(), word.start(), word.end()
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

9.5 TESTING YOURSELF

Exercise 9.1 Detecting Disulphide Bridge Patterns

Find all the Uniprot (SwissProt) sequences with pairs of cysteines separated by at most four residues.

Hint: Download Uniprot (SwissProt) sequences in FASTA format from http://www.uniprot.org/.

Hint: Search the following regular expression: `C.{1,4}C`.

Exercise 9.2 Parsing Moby Dick

Copy and paste to a text file the text from Herman Melville's *Moby Dick* (available from www.gutenberg.org). Is the word *captain* or *whale* used more frequently in the book? Write a program to search the occurrences of words in both lowercase and uppercase.

Hint: Remember that to search a character in both lowercase and uppercase, you can use the regular expression `[Aa]` or the `re.IGNORECASE` flag.

Exercise 9.3 Searching Phosphorylation Sites in Human Kinases

Search for threonine and serine phosphorylation sites in Uniprot (SwissProt) human kinases.

Hint: For the sake of simplicity, you can use the regular expression shown in this chapter (`'R.[ST][^P]'`) for phosphorylation sites.

Hint: You have to parse the file downloaded in Exercise 9.1 and filter out all records that do not have the keywords *kinase* and *Homo sapiens* in the header.

Exercise 9.4 Manually Identify Suitable HTML Tags

Print (or save to a file) a PubMed HTML page for a given publication, examine the source code of the page carefully, and try to identify unique tags delimiting the part of the HTML text containing the authors of the paper.

Hint: You could just print the content of the HTML variable of Example 9.3 or go to a PubMed abstract page and fetch the source code.

Hint: Each author is associated with a web link, so the HTML text containing authors' names will look a bit messed up.

Exercise 9.5

Write a regular expression to extract the authors from the HTML page of Exercise 9.4.

Hint: A good start for such a regular expression might be `<div class = "auths">`.