Chapter 3

EXPLORING THE FLEXIBILITY OF REGULAR EXPRESSIONS

Now that you understand some of the different categories of tools and approaches to regular expressions, you will explore the flexibility of the language in greater detail. We will show you how to tailor queries to accomplish specialized tasks and address many text manipulation challenges you are likely to face.

Character sets: Making your own wildcards

Defining custom character sets with `[ ]`

Although the wildcards you now know are sufficient for many common searches, it is often necessary to create your own special wildcards. For example, there is no built-in regular expressions wildcard which represents only the letters of the alphabet. You can create wildcards by designating a set of characters inside square brackets `[ ]`. The characters within the brackets are treated as a single character in the search. For example `[AGCT]` matches just one `A` or `G` or `C` or `T`. Just as with `\w`, if you want to match one or more characters in a row, you would put a plus sign after the closing bracket:

```
[AGCT]+
```

You can also indicate a range of numbers or letters so you don't have to specify each one, or you can even mix individual characters with ranges. Any uppercase letter is matched by `[A-Z]`, while `[0-9\.]` matches any digit or a decimal point. The expression in the brackets is case-sensitive, so to match any letter (lowercase or uppercase) specify `[A-Za-z]`. Note that there is no separator between the two ranges; each dash is defining a range between the two characters adjacent to it. You can of course include a dash in the character set by escaping it, so `[A-Z\-]` matches any uppercase letter or a dash.

CHARACTER RANGES These are based on the order in the ASCII character set (see Chapter 1 and Appendix 6). This makes it possible to specify a single range that spans both upper and lowercase characters, in the form `[A-z]`. However, this range will also include the six punctuation marks `[\]^_`` which fall between `Z` and `a` in the ASCII character set. Consult the table in Appendix 6 to see what ranges are both permissible and useful.

Applying custom character sets

It is often possible to perform what would at first seem to be a complex reformatting task by breaking it into a sequence of a few simple search and replace operations, one after the other. In the next example, you'll convert latitude and longitude values indicated by N, E, S, W into positive and negative values. You'll also join two related lines of data into one line.

Our earlier latitude and longitude example was a bit unusual in that the starting data had both +/- at the beginning and N, E, S, or W at the end. Usually you would only have one or the other. The following data set (available as `LatLon.txt` in the `examples` folder) has five locations, with latitudes and longitudes on consecutive lines:

```
 21 17'24.68"N
157 51'41.50"W
 38 30'36.62"N
 28 17'16.87"W
  8 59'53.30"S
157 58'13.70"W
 10 24'47.84"N
 51 21'54.61"E
 22 52'41.65"S
  48 9'46.62"E
```

Start writing an expression to combine each pair of latitude and longitude lines into a single tab-delimited line using positive and negative values. For example, the record specified by the first two lines will become:

```
21 17'24.68"    -157 51'41.50"
```

Notice that you want to remove the invisible end-of-line character `\r` from lines that end with `"N` or `"S`, but not from lines that end with `"E` or `"W`. This can be accomplished by searching for:

```
(\"[NS])\r
```

and replacing it with:

```
\1\t
```

`[NS]` means "either `N` or `S`," and because `\r` is outside the `()`, it is not part of the replacement `\1`, so the line breaks will be replaced by tabs.

Try `\n` for `\r`.

In this example, you could search without the preceding `\"`, and you could also write a query for values that end with single quotes. By using the quote marks instead of the letters alone, it makes our query more robust if the data file includes other names, but less flexible in situations where the seconds (`"`) might not be specified.

After this initial search and replace, the data appear as follows, with the value pairs joined to form single lines:

```
21 17'24.68"N     157 51'41.50"W
38 30'36.62"N      28 17'16.87"W
 8 59'53.30"S     157 58'13.70"W
10 24'47.84"N      51 21'54.61"E
22 52'41.65"S       48 9'46.62"E
```

Next, use the compass points (characters N, S, E, or W) to decide whether to add a minus sign (-) before removing the compass points from the file. Latitudes to the south of the equator and longitudes west of the prime meridian are by convention negative, so you can find coordinates followed by W or S and replace them with the same coordinates preceded by a minus sign.

The first part of the query is any combination of the digits and symbols used to indicate latitude or longitude values, but it matches only if they are followed by a `W` or `S`:

```
([0-9]+ [0-9 \'\"\.]+)[WS]
```

← This matches either W or S, not both together

Replace with `-\1`:

```
 21 17'24.68"N   -157 51'41.50"
 38 30'36.62"N    -28 17'16.87"
 -8 59'53.30"    -157 58'13.70"
 10 24'47.84"N     51 21'54.61"E
-22 52'41.65"       48 9'46.62"E
```

To remove north and east, search for `[NE]` and replace with nothing. You do not need to capture any text.

The time it takes to complete these three commands in succession on a single file with many entries will be much faster than doing it by hand. Of course, if you have to do such conversions on many different files, or repeatedly over a long time, you will want to write a program or script that automates the full sequence of modifications. In later chapters, we will show you how to do exactly that.

Negations: Defining custom character sets with `[^]`

You can place as many characters as you like within `[]`, but this can quickly get out of hand. If the character set is large you may forget something. Thus it is sometimes easier to specify the characters you *don't* want to match. This can be accomplished using the caret (`^`, typed by shift 6 on U.S. keyboards) as the first character inside the square brackets. In these cases, the bracketed term matches any character (letters, numbers, punctuation, and white space including the end-of-line \r) *except* those that follow the caret. For example, searching for [^A-Z\r] and replacing it with x will x-out anything in the document except capital letters and the end of the line.

This may seem like a relatively esoteric trick, but it has at least one very important use: it is a good shortcut for recognizing the columns of a character-delimited file. The delimiter is just a character that is reserved to separate the values of each column, most commonly a tab or comma. For instance, because \t specifies a single tab character, each row in a three column tab-delimited file has this format, where X, Y, and Z are the data values:

```
X\tY\tZ\r
```

The act of pulling individual pieces of data from a line like this is called **parsing**, and this is a critical skill for analyzing data. To separate tab-delimited data quickly, you can use the negation character to build a wildcard [^\t]+ that matches "anything except tab."

Using this approach, a general search term for capturing a column of tab-delimited data would be:

```
([^\t]+)\t
```

In other words, "anything except a tab, followed by a tab." There are tabs between each field, not after each field. In a line with three columns of data, for instance, there are only two tabs since no tab follows the final column.

To pull values out of tab-delimited files, copy and paste this once for each column of data in your data set, which will load your data into \1, \2, \3, etc. This provides a very simple way to change the order of columns in a text file, or to modify certain columns. If one program outputs data in a tab-delimited text file in the order X\tY\tZ and another program expects these data in the order Z\tX\tY, you can search for ([^\t]+)\t([^\t]+)\t([^\t]+) to capture the values and then write them out in the new sequence with \3\t\1\t\2.

Note that since you are using uncaptured tabs in the search term to demarcate the text you want to capture, it is necessary to include tabs in the replacement term or they will be removed. Of course, you can use another delimiter in the replacement term if you wish. For instance, to switch the column order and change the delimiter to a comma in one step, you would use a replacement term of \3,\1,\2. You would also want to use \r either at the end or within the square brackets, because the end-of-line character is also included in [^\t].

As written, ([^\t]+)\t([^\t]+)\t([^\t]+) will match only the first, second, and third columns of each line. This is because regular expressions match the first instance of the specified pattern that they encounter, and usually each non-overlapping instance of the pattern after that (i.e., the last part of one match can't be the first part of the next match). If there are only three columns, this regular expression will match all three. If there are more than three columns but less than six, it will match the first three and leave the others untouched. If there are more than six columns, it will match two or more times, depending on the actual number of columns. If there are less than three columns, it won't match at all since no line would contain the full search term. Sometimes you may want to explicitly match only lines that contain exactly three columns, match only the first three columns no matter how many columns there are after that, or even anchor the search to the end of the line to modify only the last columns. The next section will give you the tools required for all these tasks.

Boundaries: ^beginnings and endings$

Some regular expressions symbols don't correspond to characters, but instead match the boundaries next to characters at certain places in a word or line. The most useful of these are shown in Table 3.1.

The ^ can be a bit confusing since it has different meanings in different contexts. We've just told you that it is used to negate character sets, but that was within square brackets. Outside of brackets it has this different meaning of matching the beginning of a line, right before the first character. The $ matches the end of the line, just before the special line-ending character. If you "replace" the beginning or ending anchor (you aren't really replacing it, because the beginning of the line is still there), it has the effect of inserting a character before or after the rest of the line. In contrast, if you replace the line-ending \r you'll end up combining multiple lines into one.

An example of using ^ would be to add a leading column that says Sample to each line of a tab-delimited text file. To do this, search for ^ and replace it with Sample\t. You can also be more specific than this. For instance, you could append a > to the beginning of lines that start with a digit but not other lines by searching for ^(\d) and replacing it with >\1.

Boundary matching has another important function beyond tacking text onto the beginning or ending of a line: it is a convenient way to anchor searches. For example, your tab-delimited file may identify illegal or missing values using the placeholder -1 in the first column, but you want to read it with another program that expects NaN ("Not a Number," a standard symbol for illegal values, or sometimes inapplicable or missing data, used by many programs). In other columns, a -1 could be part of a legitimate value that you don't want to replace with NaN. Here you would use the anchored search term ^-1\t and replace with NaN\t. This will only find that combination when it occurs at the beginning of a line. You don't need to include ^ in the replace term since the process doesn't eliminate the boundary.

In the earlier example of replacing genus names with the first initial and a period, you had to capture the species name because you didn't want it replaced with a period. Now, with anchoring, it is easy to write a robust query which will only replace the first word

TABLE 3.1 Boundary terms for regular expression queries

Term	Matches
^	The boundary at the beginning of a line
$	The boundary at the end of a line

with its initial, even if other words or fields follow after. Here is the example text we worked with before:

```
Agalma elegans
Frillagalma vitiazi
Cordagalma tottoni
Shortia galacifolia
Mus musculus
```

This time, use this search term:

```
^(\w)\w+
```

and this replacement term: `\1\.` ← Note the escaped dot which replaces the characters after the first

to obtain this result:

```
A. elegans
F. vitiazi
C. tottoni
S. galacifolia
M. musculus
```

Adding more precision to quantifiers

*Another quantifier: * for zero or more*

Whereas the + quantifier indicates that the preceding element can match one or more times, the * quantifier indicates that it can match zero or more times. This is useful, indeed critical, if you are unsure whether the element will be present in all cases.

At times, you may want to grab part of a line and not spend time parsing the rest. Here, the all-inclusive wildcard `.*` can be very useful. If you put this dot-asterisk after a more specific search expression, it will match everything left over by the query, up to the end of the line. (Remember that `.` matches everything except `\r` or `\n`.) You can either capture this information and re-use it in the replacement query, or delete it if the data are unwanted.

Modifying greediness with ?

One important aspect of the + and * quantifiers is that they are "greedy," meaning they match the maximum number of characters that they can. This can sometimes cause surprising and difficult to understand results. For example, you might want to search through a sequence and find all the characters up until a series of repeated As at the end, and then delete that tail of As:

```
CCAAGAGGACAACAAGACATTTAACAAATCACATCTTTGTATTTTTGGTTAGAGTTGAAAAAAA
```

Your first inclination might be to search for `(\w+)A*`, or `([ACGT]+)A*` and replace with `\1`. If you try this, though, you'll notice that your `A*` term (zero or more As) doesn't find anything! Even if you use `A+` (which runs the risk of failing if the sequence doesn't end in `A` at all) you will only trim off the last `A` of the sequence. Remember, regular expressions are eager to please, and they try to match as much of a line as possible. Quantifiers likewise also try to match, left to right, as many characters as they possibly can. In this case, `\w+` is able to match the final As, while still remaining consistent with a match by `A*` (zero or more). To get around this, you could specify that there must be a "non-A" character before the final `A*`, which leaves `A*` with something to match:

```
(\w+[CGT])A*
```

← This works as long as there are no gaps or other unanticipated characters

Another way to do it, though, is to force the first quantifier to produce a *minimum* match by adding a `?` after the + sign:

```
(\w+?)A*
```

← This won't work quite yet

This search term represents the shortest series of letters which match a word followed by zero or more As. It still won't work because a single `C` by itself would match that term minimally. To make it extend to the end of the sequence, you have to anchor the final string of As using the `$`:

```
(\w+?)A*$
```

How you solve these problems often comes down to a matter of personal preference, but you should be aware of potentially unanticipated consequences of both greedy and non-greedy quantifiers.

Controlling the number of matches with {}

We've shown you how to match an entity once (with just a single character, character set, or wildcard), one or more times (by adding + after an entity), or zero or more times (by adding * after an entity). It is often necessary, though, to have finer-scale control than this, such as defining the exact number of times an entity should match in a sequence, or the minimum and maximum number of times an entity can match.

You can do this with curly brackets `{}`. The curly bracket term is placed after a character or character set. For example, `{3}` matches the preceding entity exactly 3 times, while `{2,5}` matches the preceding entity at least 2 times but no more than 5 times. You can also leave out the maximum value: `{3,}` matches at least 3 times, but more if it can. You can see additional examples in Table 3.2.

TABLE 3.2 Some examples of using curly brackets to control the number of matches

Start	Search	Replace with	Result
CTAAAAGCATAAAAAAAAAAAA	A{8,}		CTAAAAGCAT
34.2348753443	(\d+\.)(\d{3})\d+	\1\2	34.234

Putting it all together

Imagine that your colleague has sent you a data file (available as examples /Ch3observations.txt) with dates and variables in the following format:

```
13 January, 1752 at 13:53△ -1.414△ 5.781△ Found in tide pools
17 March, 1961 at 03:46△ 14△ 3.6△ Thirty specimens observed
1 Oct., 2002 at 18:22△ 36.51△ -3.4221△ Genome sequenced to confirm
     ...800 more lines like that...
20 July, 1863 at 12:02△ 1.74△ 133△ Article in Harper's
```

Data are separated by a combination of tabs (represented by the little grey triangles) and spaces. The month names may or may not be abbreviated, and the X and Y values can be positive or negative. The last column contains comments that can be discarded. Assume that our goal is to rearrange the data into columns containing the following items, separated by tabs. As an added challenge, you will switch the position of the year and day values from how they are presented in the file. Our desired output fields will look like this:

Year	Mon.	Day	Hour	Minute	X data	Y data

This operation requires separating values from each other based on several different properties. The first step is to look at the lines of data and think what you can use to identify and separate the values. At this point, don't even think about how you will phrase the actual search term, just describe the characteristics in words.

If you separate the fields just by spaces and tabs, you will end up with the following elements:

```
13   January,   1752   at   13:53   -1.414   5.781   Found in tidepool
```

This is a promising start. Now put parentheses around the parts you want to keep, and then put numbers to go with the parentheses. This is still being done on the big-picture scale of just thinking about what you have relative to what you want:

(13)	(Jan)uary,	(1752)	at	(13):(53)	(-1.414)	(5.781)	etc.
1	2	3		4 5	6	7	

Some things to notice: In group 2, you only want to save the first three characters, which constitute the month abbreviation. There are no group numbers for the "at" or the comments at the end of the line because you don't want to save that in the final file and therefore don't capture them. In the time field, you want to save hours and minutes separately. Finally, you want to save the X and Y data fields.

This is a lot to think about at once, but if you go through piece by piece like this, it becomes a more tractable problem.

Now that you have a clear view of the "keepers" from the original file, you need to figure out how to rephrase them as a query. Take the specific characters from the example line and replace them with wildcards as needed. First take care of the characters inside the parentheses and then move on to everything between:

(13)	(Jan)uary,	(1752)	at	(13):	(53)	(-1.414)	(5.781)	etc.
(\d+)	(\w\w\w)[\w\,\.]*	(\d+)	at	(\d+):	(\d+)	([-\d\.]+)	([-\d\.]+)	.*
1	2	3		4	5	6	7	

Taken together, the second line of this table—our emerging query string—looks quite complicated. But each element on its own is not that bad. Group 1 is pretty straightforward: the day should always be one or more digits.

Group 2 will be three or more letters, with an optional period or comma (or both) after it. You only want to capture the first three characters. Because the period is treated as a wildcard by default, you need to escape it with a preceding backslash, and for good measure you also escape the comma.

Skip the at and move on to the time in groups 4 and 5, which should always be some digits separated by a colon. The data values that follow can be positive or negative, so our set in the brackets includes digits, a dash, and a period (since the number may have a decimal point).

Before forming the replace string, take another pass through the query and deal with the spaces and intervening characters which you don't want to keep:

```
(13)  (Jan)uary,  (1752)   at     (13):(53)         (-1.414)          (5.781)
(\d+)\s+(\w{3})[\w\,\.]*\s+(\d+)\sat\s(\d+):(\d+)\s+([-\d\.]+)\s+([-\d\.]+).*
 1      2                  3          4     5         6                 7
```

The spaces require inserting \s+ (meaning one or more white space characters) between the other elements. You could also use actual spaces.

At this point in your query building, or even before, it is good to run a test to make sure you haven't missed something. You can open your data file in TextWrangler and type the query into the Find field (check again that Use Grep is turned on). Then click the find button and make sure that the whole line gets highlighted. This means that the text has been found. From there you can hit ⌘G to repeat the last find, and each line of the data file should highlight in succession.

Take a minute to look back at the search query, and appreciate how utterly inscrutable this pattern appears. Building it up part-by-part made it easier for you to keep track of things as you went. This divide-and-conquer approach is one of the most important work practices you can take away from this book.

Generating the replacement query

Despite all this work, you aren't yet done. You still need to carry out the replacement, so that you can generate the desired output. Fortunately, generating a replacement query is usually easier than making the search query.

Recall that the parts of your query between parentheses get saved in memory for use in generating the replacement text. You can access these elements in your replacement text using a backslash followed by the number of the group.

To just replace the original text with the fields in the order they were found, but separated by tabs, the following would do:

```
\1\t\2\t\3\t\4\t\5\t\6\t\7
```

The required replacement is a bit more complicated, because you need to rearrange some of the variables in the output.

Here are the search groups in the order that they appear in the original dataset:

Field:	Day	Mon	Year	Hour	Min	X	Y
Group Number:	1	2	3	4	5	6	7

The next step toward generating the replacement string is to rearrange the fields into the right order. After that, insert any desired punctuation (in this case a period after field 2), and finally insert the tabs which will let the fields be imported into separate columns:

Year	Mon.	Day	Hour	Min	X data	Y data
\3	\2\.	\1	\4	\5	\6	\7
\3\t\2\.\t\1\t\4\t\5\t\6\t\7						

This replacement string will let you use a single search and replace command (albeit complex) to reformat, edit, and reorganize your data files. Here we see the result:

```
1752△ Jan.△ 13△ 13△ 53△ -1.414△ 5.781
1961△ Mar.△ 17△ 03△ 46△ 14△ 3.6
2002△ Oct.△ 1△ 18△ 22△ 36.51△ -3.4221
```

When working with data, you don't have to accomplish all the substitutions in one step. You can often do something with multiple steps—although if it is a series of operations you do repeatedly, you might want to consider revisiting the process after reading the Python programming chapters. You can also sometimes use a handy trick of creating an intermediate replacement with a unique placeholder (for example, ZZZ or ### to represent a line break), which you then replace in a later search operation.

Constructing robust searches

If you have been following along and typing the examples, or trying some of these tools out on your own data, at least a few regular expressions will probably have failed due to typos or other problems. Failure can result from mistakes in designing the query, errors in the data file, or not anticipating the full range of potential variation within the data file. Since at least some regular expressions you try will fail some of the time, it is important to think not only about how to make your queries work, but how to make them fail. This isn't to say you don't want to design them to work, but that you want to design them to fail in particular ways when they don't work. It is preferable for a search to fail entirely than for it to seem to work, but fail in subtle and imperceptible ways.

When a query fails, there are two possible results: First, it can choke entirely, failing to provide any output. Second, it can make unanticipated and incorrect replacements. In the latter case, you may not even notice, and this will jeopardize your analyses. Because this can be disastrous, it is often best to overspecify your searches beyond what is required to accomplish a task. Overspecification involves using as many elements in your search as there are in the search line, and making them as specific as possible. For example, instead of capturing `(.+)` at the end of the line, spell out the fields that you expect to see there. This is most important for complex queries that are difficult to understand at one glance, as well as for queries that will be applied to large datasets, meaning that only a small fraction of the results can be verified by eye; it is also important in cases where the consistency of the input file is dubious, such as when the file format is unclear and you are limited to making a best guess, or when the data were entered by hand and there is a good chance of typos.

To make your assumptions explicit, add in some redundancies. For instance, if you think that headers are designated with the character > at the start of a line, look for ^> instead of just > or just the start of the line (^). This is because if you only use >, you could be misled if > also occurs within the header or in some other part of the file.

Another easy way to make your query more robust is to force the search to match the entire line. You can do this by including terms to match even the parts of the text you aren't processing, or by anchoring queries with ^ and $ at each end. This is a good way to increase the chances your queries are behaving as you expect.

You should look for other ways to add redundancy once your basic query is working. For example, most programs will report back the number of times a substitution has been made, so be sure to check this against the anticipated number of changes (or the number of lines in the file) to see whether there are some improperly formatted lines buried within the file that the query did not match.

The importance of thinking about the way your custom-built computational solutions will fail, and engineering them to fail in a defined way, extends well beyond regular expressions. Most commercial software goes through extensive testing before it is released, yet every major package still has bugs. Chances are that you will be using the computational tools you make for important analyses after far less testing. In addition, since many types of biological data don't have explicitly defined data formats, you are likely to encounter bad data files on a regular basis. You need to build in safety nets for yourself—putting data in and getting an answer out is not enough to verify that you are on the right track.

SUMMARY

You have learned how to:

- Define your own character sets with `[ ]`
- Isolate fields of tab-delimited text with `([^\t]+)\t`
- Match the boundaries at the beginning (`^`) and endings (`$`) of lines
- Define a wider array of quantifiers with `*` and `{}`
- Control the "greediness" of your quantifiers with `?` to produce a minimum match
- Build up complex queries step by step

Moving forward

Regular expressions, applied in the context of a text editor, can be one of your most versatile and accessible tools (Table 3.3). We use them nearly every day in our own work. You can use regexp to insert, delete, and simplify text as shown in this chapter, and you can also perform other general manipulations. The table found in Appendix 2 can serve as a quick reference guide for your query constructions.

Try to subdivide your task into combinations of these operations. Using them together with some of the other features of TextWrangler, such as Sort and Process Lines Containing... can support advanced file manipulations without writing a script. You can also use the editor's ability to search across multiple files (Search▸MultiFile Search...) as a means of extending your capabilities.

TABLE 3.3 Some common operations that can be performed with regular expression

Operation	Find	Replace
Split elements like `nano_128.dat`	`(\w+)_(\d+)\.(\w+)`	`\1\t\2\t\3`
Merge or rearrange columns of values	`(\w+)\t(\w+)`	`\2 \1`
Join multiple lines into one (`\n` or `\r`, depending on the program)	`\r` `\n`	`,`
Split a single line into multiple lines	`,`	`\r`
Transform a list of unique elements into longer items (e.g., URL, table, line of script)	See examples in Chapter 6	
Convert a list of names to move the first name and initials to the end, separated by a comma (won't work with "Jr.," etc.)	`(.*) (\w+)$`	`\2, \1`
Delete relative to the occurrence of `x`		
Beginning to last occurrence	`^.*X(.*)`	`\1`
Beginning to first occurrence	`^[^X\r]*(X)`	
Same as above	`^.*(X)`	
After the first occurrence to the end	`(X).*?$`	
Up to the last occurrence	`^.*(X)`	
Caution: Finding no match for `x` will result in deleting the whole line		