Chapter 2

REGULAR EXPRESSIONS: POWERFUL SEARCH AND REPLACE

Many of the computing tasks in biology amount to a series of simple text manipulations. There is a wide variety of programs that facilitate these text modifications for specific types of data, but there is also a single set of general tools that can be used in many different contexts: regular expressions. These are flexible means of searching and replacing text. In this chapter you will learn the components of regular expressions, while later chapters will add further flexibility and control to these tools.

A widespread language for search and replace

One of the most common computer problems faced by a scientist is to reorganize a text file generated by one program so that its content can be understood by another program. In some cases only a few manual changes may be needed. At other times simple search and replace is sufficient, such as when a file contains data separated by commas and must be modified to suit a program that expects data separated by tabs. Often, though, the required manipulations can be too complicated for these approaches, for example when the order of elements in each line needs to be rearranged. All of us have at some point made these complex changes by hand, but this gets tedious if many files need to be fixed or the files are long. There is also the danger that you may be in for more than you anticipated: most of us have spent several hours manually reformatting a file, only to realize that we have to do it all over again because something about the original file wasn't quite right.

A surprising number of these seemingly complex problems can be addressed using a powerful language for search and replace known as **regular expressions**.[1]

---

[1]Regular expressions are also sometimes referred to as regexp, regex, or even **grep**. The latter is an acronym for "**g**lobal **r**egular **e**xpression **p**rint," a common tool for using regular expressions that is so frequently used that it is often used as a synonym of the language as well.

Regular expressions are widely used and are built into many environments, including text editors, programming languages, some Internet search engines, and many applications. Because regular expressions are so powerful and such a portable tool, they are the first skill you will learn here. As you proceed, they will allow you to perform complex text manipulations across a wide range of environments using one skill set.

Regular expressions can do anything that can be done by simpler search and replace tools, such as those found in a word processing program. This includes replacing one chunk of exact text (such as "jellyfish") with another ("scyphozoan"). You can also leave the replacement term empty, which leads to the deletion of the matched text from the file.

Like many other search and replace tools, regular expressions can employ **wildcards**. These are special characters in the query that can match more than one particular character in the text being searched. Wildcards greatly extend the utility of a search and replace tool when the text you want to match is variable. This would be the case, for instance, if you wanted to find all digits (i.e., number characters), but you don't know what the digits will actually be. Compared to most other search and replace tools, regular expressions allow more flexibility and precision in the way that wildcards are designed and used. There are many ready-to-use wildcards, and you can define your own wildcards that match any sets of characters you like.

The functionality of regular expressions extends well beyond customizable wildcards. Regular expressions also make it possible to capture all or part of the search term and use it in the replacement term. It is this capability that makes regular expressions so versatile for extracting data from complex text files and redisplaying them in a different sequence. You could, for example, find any sequence of digits followed by "cm" (not just a particular number), and then insert those digits (with or without the "cm") elsewhere in the text.

To start getting familiar with regular expressions, you will first use them through the dialog box of a text-editing program. In Appendix 2, you will find tables summarizing the permitted syntax. Once you get comfortable with the language, you will find yourself doing a lot of quick "one-off" file processing using a few well-conceived searches in a text editor. Regular expressions are so powerful that many of the programs you will write later will do little more than open a file, run a series of replacement operations on the contents (perhaps applying different transformations to different portions of the file), and save the results.

Understanding the components of this new toolbox

Setting up the text editor

Open TextWrangler (the text editor introduced in Chapter 1) and create a new document. Windows and Linux users should consult Appendix 1 for comparable editors that support regular expressions.

Start just by doing a normal, old-fashioned replacement operation. In your document, type the text:

```
Agalma elegans
```

Now select Find from the Search menu (or hit ⌘F).[2] In the dialog box, check the Grep box (Figure 2.1), and make sure that Case Sensitive is also checked. Unless you uncheck them, these options will remain selected each time you open the program.
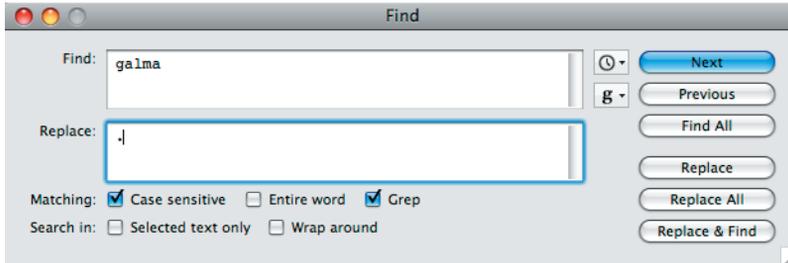
FIGURE 2.1 The Find and Replace dialog box in TextWrangler

In the Find box type:

```
galma
```

and in the Replace box type a period by itself:

```
.
```

Then click the Replace All button. As expected, where it originally said:

```
Agalma elegans
```

it now says:

```
A. elegans
```

With the document window in the foreground, undo this last operation (⌘Z) so that the text again says Agalma elegans. A literal search term works fine if you are dealing with only one species, but if your data file includes other species this one-size-fits-all approach breaks down:

Try ctrl Z.

```
Agalma elegans
Frillagalma vityazi
Cordagalma tottoni
```

---

[2] The symbol refers to the Command key on Apple keyboards, just to the left of the space button. It is not the ctrl key (Ctrl), which will be used later in terminal operations. In Windows programs, the find command will likely be triggered by ctrl F.
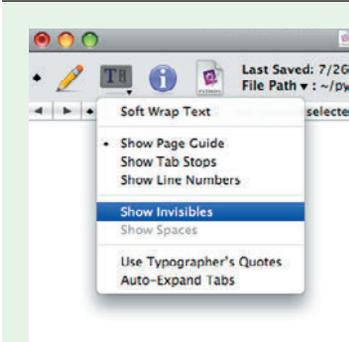
INVISIBLE CHARACTERS
By default, placeholder symbols for spaces are not shown in most editing programs, but it is sometimes helpful to see these and other invisible characters, such as tabs. A drop-down menu in **TextWrangler** allows you to turn on this option by selecting **Show Invisibles** and then **Show Spaces**. In older versions of the program, the menu is under a different small icon, and in other programs you might have to search the help to find the equivalent command. For the moment, turn on both invisibles and spaces, and you should see little place-holders in your text file.

Here, the same replacement will give you:

```
A. elegans
Frilla. vityazi
Corda. tottoni
```

Equally awkward will be the situation where the search term occurs outside the instance you want to replace:

```
Mus musculus
```

Replacing this text (using `Replace All` and the query `us`) will cause more problems than it solves:

```
M. m.cul.
```

Within the next few pages, you will be able to build regular expressions to handle all these cases and many more.

Your first wildcard: \w *for letters and digits*

Some programs do not have \w. See Appendix 1 or use jEdit.

To add more flexibility to your searches, you can use a **wildcard**. A wildcard is a special character that represents a specific variety of characters, such as any numeric digit. There are different wildcards that match different ranges or sets of characters, many of which are listed and explained in Appendix 2.

There are several different notations for using regular expression wildcards. Most commonly, wildcards are represented by a letter preceded by a backslash. The first wildcard introduced here is \w. It matches any letter (`A–z`) or digit (`0–9`), as well as the underscore character (`_`).

To test this wildcard, type the following latitude and longitude into a new blank document:

```
+40 46'N +014 15'E
+21 17'N -157 52'W
```

In this case, imagine that you want to get rid of all the trailing direction indicators, for example north (`N`) and east (`E`). You could run four different searches to get rid of `N`, `E`, `S`, and `W` in turn. Instead, to create one search term that covers all these cases, use the \w wildcard. Remember that \w does not match just alphabet characters, but also digits. If you search and delete \w, you will end up with:

```
+ ' + '
+ ' - '
```

(Try it, then undo to get the original text back.) To get around this problem, take advantage of the observation that the letters you want in this case always come after a single tick mark. So you can use this search query:

```
'\w
```

> 🔔 **SPECIAL PUNCTUATION** The data tick marks (`'`) in latitudes and longitudes, or in references to chemical structures (`5'-AGCT-3'`), are different from the "curly" quote mark (') used in contractions such as `won't` as well as in quoting text. Most text editors create tick marks when you use the double- and single-quote keys on the keyboard, but word processors often employ a "smart quotes" feature which automatically inserts curly quotes instead. On Mac OS X, you can insert curly quotes using the `option` and `shift` keys in combination with open and close brackets: `option` `]` makes opening single quotes and `option` `[` makes opening double quotes, while adding the `shift` key in either case makes the appropriate closing quotes.
>   The degree (°) symbol, which you may encounter in latitudes, longitudes, angles, and temperatures, can be entered on Mac OS X using the key combination `option` `shift` 8. (Remember this as an alternative to `*`.)
>   In general, if there is a special punctuation character in the text you are working with and you would like to include it in your search term, it is usually easiest to simply copy the character from the document and paste it into the search box. This will ensure that you are using the right character—far from a trivial concern, since many punctuation marks look quite similar. It also avoids the need to hunt down the particular keystrokes required to create obscure characters.

and replace with just a tick mark by itself, leaving:

```
+40 46' +014 15'
+21 17' -157 52'
```

As you can see, this searches for any letter, number, or underscore that comes immediately after a tick mark, and replaces it with the tick mark itself.

Note that the letters used to form wildcards are **case sensitive**, which is to say that `\W` does not mean the same as `\w` (in fact, it means the opposite!). On the other hand, letters in a search term which are *not* wildcards will only match the same letter of the same case, so searching for `agalma` will usually not find `Agalma`.

Capturing text with `()`

Perhaps the most important characteristic of regular expressions searches is the ability to use parentheses to capture portions of the original text and use them in

> **TROUBLESHOOTING** If you can't get your search and replace commands to work, there are a number of things to check. Your text editor must support grep searches, and you should make sure you have **Use Grep** selected. Capitalization and spaces matter, so check for unnoticed spaces at the beginning or end of your search term. (You can highlight all the text in the search term to reveal spaces.) While creating search terms, remember that just because you can't see spaces doesn't mean they can be ignored. They are treated just like any other character. **TextWrangler** will help clarify the structure of queries in the search dialog box by highlighting special characters in red and wildcards in blue, leaving normal characters in black. Other editors may have slightly different wildcards, as described in Appendix 1.

creating the replacement text. Datasets often have extraneous characters which make it difficult to import the files as-is into a graphing program or spreadsheet.

As a simple example, imagine you have a list of ordinal numbers:

```
5th
3rd
2nd
4th
```

You want to delete the letters after the numbers and have just the numbers remain in a column. It is often helpful to think in general terms about what you are hoping to achieve with a search and replace, before trying to translate that into a regular expressions term. Drawing from what you have learned so far, you could write a search term like:

```
\w\w\w
```

which would match the three characters in sequence. You want to keep the first character (that is, just the number) and remove the last two. If you try to just search for `\w\w`, it will match `5t`, then `3r`, etc. Regular expressions are non-overlapping. For instance, `\w\w` wouldn't match `5t` and then `th` in the first line since the `t` would be in both matches.

The solution is to capture certain parts of the text that the search term finds and put these into the replacement term. To do this, place parentheses around the parts of the search term you want to save. In the example above, you can capture the first character (the number) of the three as follows:

```
(\w)\w\w
```

To place that captured portion in the replacement term, use the backslash character, followed by a number indicating which set of parentheses you extracted the text from. In this case, you just have one set, so the replacement term would be:

> 💡 Other text editors and some pro-gramming languages use $1, $2, etc., instead of \1, \2. Confirm the behavior of the editor or language you are using with a simple test before spending time trying to figure out why things are not working.

```
\1
```

Searching with `(\w)\w\w` and replacing with `\1` on the list of ordinal numbers above gives:

```
5
3
2
4
```

Try it out in TextWrangler and move the parentheses to capture different parts of the three characters.

To capture two parts of the text, you could use a search query like `(\w)(\w\w)`. In this case, the number will be inserted where you place `\1` in your replacement

term and the two letters that follow will be inserted where you place \2 in the replacement term.

Remember that both search terms and replacement terms can contain a combination of normal text and regular expressions. For example, the replacement term `Position: \1` converts the list to:

```
Position: 5
Position: 3
Position: 2
Position: 4
```

This example query will not work properly for numbers of two or more digits, but later you will see how to accommodate many more general cases.

Quantifiers: Matching one or more entities using +

Wildcards such as \w provide the flexibility of changing different types of characters at once. To be really useful, however, a search term should also be able to handle different counts of characters as well, and then be able to modify the replacement text depending on what it finds. By default, each wildcard matches a single character. Methods of adjusting the number of times an element such as a wildcard matches are called **quantifiers**. Plus (+) immediately after a character indicates that the term should match one or more times in succession. For example, the term \w+ could represent a single character, such as the letter a, or many characters, such as the number 123.

To illustrate further, let's return to the first example of replacing genus and species names with their abbreviated forms. Here is our original list:

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

> 💡 Instead of retyping text snippets from scratch each time you modify them, simply bring the text document to the foreground and choose **Undo** (⌘Z) to return them to their original state. You can also test search terms with **Find** rather than with **Replace**. Type ⌘G, the shortcut for **Find Next**, to find each instance if there are multiple matches.

You can use \w+ by itself to generate a search term that will match an entire word anywhere in this list of names. Searching for one or more word characters will match with any sequence of such characters found together up until the next non-word character (such as a space, punctuation, or the end of the line). To preserve the first letter of each word, add another non-repeated \w before it, and capture that first letter with ().

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

To generate the replacement term, use the captured letter (represented by \1) followed by a period:

```
\1.
```

You could test this search and replace against the example list now, but you can probably imagine why it won't work correctly yet.

There are several ways to fix the search term so it *does* work; which way is best will depend on the rest of the data file you are modifying. In the present case, the search term will be modified to capture the second word independently—that is, just the species name—so that you can add it back into the replacement text.

Try doing `Replace All` using the following search and replace pair:

Find	Replace
`(\w)\w+ (\w+)`	`\1. \2`

This combination will generate an abbreviated genus and species list as below:

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

Notice that the second captured term is `\w+`, with the quantifier placed inside the parentheses, so all of the characters up to the next non-word character (in this case, an end-of-line character) are preserved and can be recovered with `\2`. Any spaces occurring in the text have been typed directly into the search query.

Analysis programs often require a shortened version of the taxon name separated only by an underscore. It should be clear how to modify the replacement term above to create a name in the format `A_elegans`. You can even reuse the captured text, so as to preserve the original genus and species pair while at the same time generating a shortened version:

W L

For jEdit use $2.

Find	Replace
`(\w)(\w+) (\w+)`	`\1\2 \3 \1_\3`

```
Agalma elegans A_elegans
Frillagalma vitiazi F_vitiazi
Cordagalma tottoni C_tottoni
Shortia galacifolia S_galacifolia
Mus musculus M_musculus
```

With the three building blocks of regular expressions you have learned—wildcard, quantifier, and capture terms—you can already do some very powerful manipulations.

*Escaping punctuation characters with *

With all these special uses for `+` and `()`, you might begin to wonder how you would search for these characters themselves in your text. Again, the backslash is used to modify how a character is interpreted. To remove the special meaning of punctuation in a search term, put `\` before the character. This is a general trick you will see in other contexts, and is referred to as **escaping** the character. This technique even applies to the `\` character itself: to search for `\`, use `\\`. In fact, so many punctuation marks have special meanings that it is often a good idea to use a preceding `\` whenever you want such a mark to be taken literally. When escaping, the `\` has the opposite effect that it has with letters, where it *gives* them special meaning, as with `\w`.

For example, to obtain the final element in `Physalia physalis (Linnaeus)`, start by generating a search to match each part of the text:

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

In this formulation, none of the text will be captured to `\1  \2`, etc., because the parentheses are escaped by the preceding backslashes. Now just place parentheses around the word characters in the final element:

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

The variable `\1` now contains the text `Linnaeus`.

As you will have noticed, the appearance of regular expressions can get very confusing very quickly. It is almost always easiest to copy an actual example of the text you will be searching, and paste two copies of it into your text editor window. Then progressively edit one of the copies into the search term. Once you have a good search term typed into the text window, copy it and paste it into the search box of the `Find` dialog. Provided that `Use Grep` is checked *before* you paste, the search should be interpreted correctly. If it is not checked, on the other hand, the program may escape out all your punctuation for you, so that your wildcards and quantifiers are interpreted as normal characters instead.

Here is a similar step-by-step approach to generating a complex search term, using the same example text. Spaces have been inserted in the first steps to make the elements more distinct:

Original text	`Physalia physalis (Linnaeus)`
Search with matches	`\w+ \w+ \( \w+ \)`
Text with captures	`(\w+) (\w+) \( (\w+) \)`
No extra space	`(\w+) (\w+) \((\w+)\)`
Replacement text	`\1_\2_\3`

Applying the search leaves the text without parentheses:

```
Physalia_physalis_Linnaeus
```

TABLE 2.1 Some of the most common search wildcards	
Search term	**Meaning**
\w	A **word** character, including letters, numbers, and the underscore
\t	A **tab** character (can also be used in replacements)
\s	A **white space** character, including spaces, tabs, and the end-of-line
\r \n	**End-of-line** markers (can also be used in replacements) TextWrangler uses \r, but jEdit and Python programs will use \n
\d	A **digit**, from 0 to 9
.	**Any** letter, number, or symbol, except end-of-line characters

More special search terms: \s \t \r . \d

In addition to \w, there are many other wildcards and special characters. A few of the most general of these are listed in Table 2.1, and a more complete list is provided in Appendix 2.

The tab character (\t) is widely used in regular expressions searches because tabs often separate columns of data from each other. By inserting tabs between captured text in the replacement term (for example \1\t\2\t\3 in the genus-species-author search), you can rapidly reformat a plain text file into spreadsheet-suitable format. This works with data copied from PDF files or Web documents—anywhere that information occurs in a predictable manner.

As an example, consider latitude and longitude data that might be generated by a GPS (note that there is a single space between the degrees and minutes):

```
-9 59.8'S -157 58.2'W
+21 17.4'N +157 51.6'W
+38 30.5'N +28 17.2'W
+40 46.1'N +14 15.8'E
+10 24.8'N +51 21.9'E
```

To get these into a spreadsheet with different fields in different columns, the goal is to capture each degree–minute pair and separate them using the replacement \1\t\2\t\3\t\4. For the moment, it is okay to preserve the positive and negative symbols at the beginning of each value.

The latitude and longitude formats are the same, so one general search term can be duplicated to handle both. The job of this search term is to capture all the values before the space, then the values between the space and the tick mark (').

First look at some of the challenges to generating a universal query for these data:

- The symbol right before the number can be either a plus or a minus (+ or –).

- The number of digits used to specify degrees can have one, two, or three digits, so this calls for a flexible quantifier (+).

- There is a decimal place in the middle of the minutes value, which must be accounted for (and escaped so it is not read as a wildcard).

- The character at the end of the field can be either N, E, S, or W.

This is a lot to think about, but well within the abilities you already have. To begin, copy one of the lines into a new document so you can begin substituting regular expressions text for the original text. Then put parentheses around the items to capture:

Original text	+38 30.5'N
Mark captures	(+38) (30.5)'N
Regular expression with wildcards	(.\d+) (\d+\.\d)\'\w
Replacement text	\1\t\2\t

In order to read regular expressions, look at the backslash and the character that follows it as a single entity. Otherwise, it becomes too confusing, especially with parentheses and periods. Reading the search expression in the third line from left to right would translate to something like:

"Any character followed by one or more digits, to be saved as \1. Next, a space symbol. Then save as \2 one or more digits followed by a decimal point, followed by a single digit. Finally, a tick mark and a single word character; these go uncaptured and are therefore deleted."

To search for the longitude values that follow, duplicate this general term, separating the fields with one or more white space characters:

```
(.\d+) (\d+\.\d)\'\w\s+(.\d+) (\d+\.\d)\'\w
   ↳first values        spaces↵      ↳second values
```

Account for all the captured fields with an expanded replacement term:

```
\1\t\2\t\3\t\4
```

When generating the search term, remember that a literal period character needs to be preceded by a backslash, and that + applies to the character immediately before it, either by itself or in a sequence of like characters.

By default, TextWrangler and other editors apply the search term line-by-line to a file. If you want to search across lines, you can end the search term with the line ending character (\r, or sometimes \n). This is a way of joining lines, and if you want to preserve line endings, you will have to add that \r to the end of the replacement term.

Example: Reformatting molecular data files

Now that you understand the key elements of regular expressions, you are ready for a larger reformatting job. Regular expressions are very useful when data are provided in one format but are needed in another format. The following protein sequences (available in the examples folder as FPexamples.fta) have headers with the accession number, a description of the protein, and the genus and species of origin in brackets:

```
>CAA58790.1= GFP [Aequorea victoria]
MSKGEELFTGVVPILVELDGDVNGQKFSVRGEGEGDATYGKLTLKFICTTGKLPVPWPTL...
>AAZ67342.1= GFP-like red fluorescent protein [Corynactis californica]
MSLSKQVLPRDVKMRYHMDGCVNGHQFIIEGEGTGKPYEGKKILELRVTKGGPLPFAFDI...
>ACX47247.1= green fluorescent protein [Haeckelia beehleri]
MEFEPEFFNKPVPLEMTLRGCVNGKEFMIFGKGEGDASKGNIKGKWILSHSEDGKCPMSW...
>ABC68474.1= red fluorescent protein [Discosoma sp. RC-2004]
MRSSKNVIKEFMRFKVRMEGTVNGHEFEIEGEGEGRPYEGHNTVKLKVTKGGPLPFAWDI...
```

Instead of this format, it might be preferable to work with these sequences if the names were shortened to preserve just the initial identifier and the genus name, while removing spaces and leaving the sequence information untouched:

```
>CAA58790_Aequorea
MSKGEELFTGVVPILVELDGDVNGQKFSVRGEGEGDATYGKLTLKFICTTGKLPVPWPTLV...
>AAZ67342_Corynactis
MSLSKQVLPRDVKMRYHMDGCVNGHQFIIEGEGTGKPYEGKKILELRVTKGGPLPFAFDIL...
```

You can use the fact that each name begins with > and the species names are in [ ] to construct a query that only modifies the headers and not the sequences.

Start by copying a line of data into a new file and identifying the portions that you would like to capture:

Original text	>CAA58790.1= GFP [Aequorea victoria]
Mark the captures with ()	(>CAA58790).1= GFP [(Aequorea) victoria]
Add wildcards (extra space for clarity)	(>\w+) .+ \[(\w+) .+
Final query (no spaces)	(>\w+).+\[(\w+).+
Replacement	\1_\2
Result	>CAA58790_Aequorea

This query finds and saves the first word after the > symbol, up to the period, which is not in the \w character set. The period is used as a wildcard to match everything up to the next square bracket, indicated by \[. The first word in the

brackets is captured as \2 and the rest of the line after the space is discarded. It is a little confusing that the period is acting as a wildcard, yet in this situation matches an actual period. Also, in the search query, the > does not need to be escaped with \, but it wouldn't hurt to do so.

For a replacement string, you can join the captured identifier (which includes the > since it is inside the parentheses) and the genus using an underscore.

Comments about generating regular expressions

Regular expressions searches are picky. Although you can write an expression to capture almost any type of text, if you specify something and it is not there, the search will fail. Getting one part of the query wrong often leads to the entire query failing to match. It is therefore important to anticipate the full range of variability in your data.

Another problem can occur if hidden characters find their way into a regular expressions Find box; typically this happens when pasting text from the document into the box. Check for spaces and extra return characters both in your document and in the query itself—they won't be visible in the Find box, even if Show Invisibles is checked in the document options.

A table of regular expression terms and their usage can be found in Appendix 2.

> A general approach to debugging your expression is to cut out various portions of the search, and test different subsets of the search to see where it fails when you add something back in. It can be informative to do **Find** without **Replace** to highlight the matching text. Proofreading a manuscript requires close attention, but proofing regular expressions often requires even more careful scrutiny of every character.

SUMMARY

You have learned how to:

- Create search and replace queries and apply them in a text editor
- Use the following wildcards and special characters:

    \w for letters, numbers and the underscore

    . for any character except line breaks

    \d for numbers

    \r (or \n) for line breaks

    \s for spaces, tabs, and end-of-line characters

    \t for tabs

- Capture portions of the search with ()
- Reuse captured text with \1
- Escape punctuation and special characters with \
- Use the plus + quantifier to repeat a character or wildcard