# CHAPTER 7
# Pattern Matching

This chapter treats an enormously important topic and the Python module that supports it. We'll use its features frequently in examples later in the book. As you gain familiarity with them, you'll be using them increasingly more often. Some patience is required, though: the topic is quite technical, and you can't absorb it all at once. You'll need to come back to this periodically and explore it further. Be ambitious in experimenting with it. This is the only chapter in the book that's like this.

The mysterious topic is string pattern matching using *regular expressions*. The term "pattern matching" should be enough of a hint for you to realize how important this topic is for working with bioinformatics data. It's also important for processing web pages and for many other kinds of work a bioinformatics programmer needs to do. Among other things, restriction enzyme binding sites can be expressed as regular expressions. A carefully constructed regular expression can take the place of many lines of string manipulations, loops, and iterations. Once you've learned to use regular expressions, you'll have a very powerful tool in your hand.

If you have never encountered regular expressions before—or even if you've used only their most basic features—you should be aware that the topic is rather large, and learning it is not easy. More than most aspects of programming, learning to use regular expressions requires experimentation. Start by using the basics, and whenever you can't do what you want with what you already know, go back to this chapter (or some other documentation) and learn a little more.

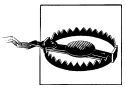
Despite how much this chapter gives you to absorb, there are many advanced aspects of regular expressions that aren't covered here at all. One could write an entire book about regular expressions, and in fact several people have.* Regular expression facilities are found in the libraries of many languages, as well as in editors, IDEs, and other tools.

* See, in particular, Jeffrey Friedl's *Mastering Regular Expressions*, Third Edition, and Jan Goyvaerts and Steven Levithan's *Regular Expressions Cookbook*, both published by O'Reilly.

Unfortunately, the details of regular expressions differ somewhat among different programming languages and editors, and not all resources cover Python's version exactly.

Working out the details of a regular expression you need for a particular task can be frustrating. You should design and test regular expressions outside of Python before putting them into a program. Some Python IDEs incorporate a regular expression editor and tester, but IDLE does not. An excellent interactive web page for testing regular expressions can be found at *http://re-try.appspot.com* (with a hyphen between "re" and "try"); you should use it often.

Regular expressions are strings used as patterns to perform matching and searching operations on other strings, which we'll call the "targets." The result of a successful operation is a *match object*, which has quite a few fields and methods providing details about the match, such as the position within the target where the pattern was found. If the match or search fails, the result is `None`, rather than a match object.



Python makes a distinction between *matching* and *searching*, which other languages do not. Matching looks only at the *start* of the target string, whereas searching looks for the pattern *anywhere* in the target. In discussing regular expressions it is very difficult to limit the use of the term "match" to Python's matching operations, and we will use it for both cases.

## Fundamental Syntax

Within a regular expression string, certain characters have special significance. The most frequently used are similar to the wildcard characters of command-line file utilities such as `ls` or `dir`. The wildcard characters don't have quite the same meaning in regular expressions as they do on the command line, but the idea is similar.

Backslashes have special significance within regular expressions. Backslashes also have special significance within Python strings in general. Unfortunately, what a backslash means to Python isn't always the same as what it means in a regular expression, and to include a backslash inside a regular Python string requires doubling it as \\ (just as strings can include quotes by preceding them with backslashes).

To help reduce confusion and simplify the use of backslashes in regular expression strings, Python provides a *raw string* syntax: preceding a string with the character `r` or `R` tells Python to ignore the usual special interpretation of backslashes within the string. Whatever backslashes appear in a raw string are just backslashes. *Always use raw strings for regular expressions*. Here is a demonstration of the difference:

```
>>> '\n'
'\n'
>>> len('\n')
```

```
1
>>> r'\n'
'\\n'                          # a slash followed by a newline character
>>> len(r'\n')
2
```

## Fixed-Length Matching

Except for characters that have special interpretations, each character in a regular expression matches only that character in the target. We'll start by looking at how restriction enzyme recognition sites can be represented as simple regular expressions.

### Literal matches

The recognition site of the restriction enzyme EcoRI is `GAATTC`. The regular expression `'GAATTC'` will match exactly those characters. Since `'GAATTC'` has no special regular expression characters in it, searching for it isn't really any different from using `target.find(`*`pattern`*`)`. Table 7-1 gives a few examples.

*Table 7-1. Literal regular expression matches*

| Pattern | Target | Match position | Search position |
|---|---|---|---|
| GAATTC | GAATTC | 0 | 0 |
| | TTGAATTC | None | 2 |
| | AATGTGAATTCT | None | 5 |

Many restriction enzymes recognize sites with more flexibility than a literal match. For instance, EcoRII recognizes either `CCAGG` or `CCTGG`. We don't really need regular expressions to look for any of several substrings: we could easily write functions for matching or searching that do this, as in Example 7-1.

*Example 7-1. Functions for multiple match and search*

```
def multi_match(target, patterns):
    """Return True if target begins with any of the strings in patterns"""
    for pattern in patterns:
        if target.startswith(pattern):
            return True

def multi_search(target, patterns):
    """Return the first position in target where any of the strings in patterns is found"""
    return min([target .find(pattern)
                for pattern in patterns
                if target.find(pattern) >= 0])
```

Then we could write this:

```
multi_match(sequence, ('CCAGG', 'CCTGG'))
```

to see if `sequence` begins with either of EcoRII's recognition sites, and this:

```
multi_search(sequence, ('CCAGG', 'CCTGG'))
```

to find the first position matched by either site.

The enzyme DsaI recognizes any of `CCGCGG`, `CCGTGG`, `CCACGG`, or `CCATGG`. To use these functions for this enzyme's recognition sites we would have to provide a list of four patterns. SecI recognizes any site of the form `CCNNGG`; that means we'd have to include 16 sequences in the list of patterns for those sites. This would get out of hand pretty quickly. For instance, CjuI recognizes `CA`, followed by `C` or `T`, followed by any five bases, followed by a `G` or an `A`, followed by `TG`. This is 2*4*4*4*4*4*2 = 4,096 possibilities!

### Character sets

The simplest tool provided by regular expressions to manage this "combinatorial explosion" of possibilities is the ability to specify a *character set* rather than a single character. A character set consists of one or more characters enclosed in square brackets. Each character set in the pattern corresponds to exactly one character in the target: if none of the characters in the set matches that character in the target, the pattern is not a match. A regular expression for DsaI's site would be:

```
'CC[GA][TC]GG'
```

SecI's sites can be represented as:

```
'CC[TCAG][TCAG]GG'
```

And CjuI's can be represented as:

```
'CA[CT][TCAG][TCAG][TCAG][TCAG][TCAG][GA]GG'
```

Character sets can contain character *ranges* in addition to individual characters. A range consists of two characters separated by a minus sign (`-`). This is equivalent to including the first character, the last character, and every character in between in the square brackets. For example, `'[A-Za-z_]'` would match an underscore or any uppercase or lowercase ASCII letter.

Characters that have special meanings in other regular expression contexts *do not* have special meanings within square brackets. The only character with a special meaning inside square brackets is a `^`, and then only if it is the first character after the left (opening) bracket. In that case it negates the character set, meaning the set matches any character *except* the ones specified inside the square brackets.

If you want to include a right bracket in a character set, you have to make it the first character; otherwise, it will be interpreted as the end of the character set. Similarly, if you want to include a minus sign in the character set you have to make it the first character, or it will be interpreted as showing a range.

You can also preface a `^`, `]`, or `-` with a backslash. While backslashes are used for a lot of things in regular expressions, they are always used with at least one other character, and technically are not one of the characters with "special meanings." To include a backslash in a regular expression—in a character set or by itself—you must use two

backslashes. The usual backslashed characters, such as `\t` for tab, can also be included in character sets.

Within a regular expression, a period matches *any* character. In effect, it represents a universal character set. Table 7-2 gives some examples.

*Table 7-2. Character sets in regular expressions*

| Pattern | Matches |
|---|---|
| `[ACTG]` | One DNA base character |
| `[A-Za-z_]` | One underscore or letter |
| `[^0-9]` | Any character *except* a digit |
| `[-+/*^]` | Any of +, -, /, *, ^; ^ does not negate the others because it is not the first character in the set |
| `[0-9\t]` | A tab or a digit |
| `.` | Any character |

### Character classes

For convenience, the regular expression syntax offers some notation for *character classes*. These are similar to character sets in that they represent any of a certain group of characters. In fact, a character class can even be included in a character set, the way a range can. Character classes are simply convenient abbreviations for commonly used character sets and ranges that might be included as part of character sets. Table 7-3 lists Python's regular expression character classes.

*Table 7-3. Character classes in regular expressions*

| Character | Matches |
|---|---|
| `\d` | Any digit |
| `\D` | Any nondigit |
| `\s` | Any whitespace character |
| `\S` | Any nonwhitespace character |
| `\w` | Any character considered part of a word |
| `\W` | Any character not considered part of a word |

### Boundaries

We've already seen in previous chapters that Python makes it very easy to treat a text file as a collection of lines. Several of the chapters that follow this one include examples of regular expressions used in processing plain text. These will include sequence data, protein data, web pages, and text in structured formats.

There is regular expression notation to indicate the beginning or end of a target, the beginning or end of a line within a target, and word boundaries. These *boundary*

indicators are a bit unusual in that they don't actually match any characters: they just indicate positions within the target. They're listed in Table 7-4.

*Table 7-4. Boundaries in regular expressions*

| Character | Matches |
|---|---|
| `^` | The start of a line or the beginning of the pattern |
| `$` | The end of a line or the end of the pattern |
| `\A` | The start of the pattern only |
| `\Z` | The end of the pattern only |
| `\b` | The boundary between a word and nonword character or vice versa |
| `\B` | Anywhere except the boundary between a word and nonword character or vice versa |

Table 7-5 gives a few examples of how boundary indicators are used in regular expressions.

*Table 7-5. Line beginnings and endings in regular expressions*

| Pattern | Matches |
|---|---|
| `^CG` | A target line or string starting with `CG` |
| `TATATA$` | A target line or string ending with `TATATA` |

## Variable-Length Matching

The real power of regular expressions comes from the ability to specify that certain parts of a pattern can be repeated to match the target.

### Repetition

Repetition is specified using the notation shown in Table 7-6. Each of these repetition notations refers to the regular expression that directly precedes it. A single letter by itself is a regular expression, as are a character set and a period.

*Table 7-6. Repetition characters in regular expressions*

| Character | Matches |
|---|---|
| `?` | Zero or one repetitions of the preceding regular expression |
| `*` | Zero or more repetitions of the preceding regular expression |
| `+` | One or more repetitions of the preceding regular expression |
| `{`*n*`}` | Exactly *n* repetitions of the preceding regular expression |
| `{`*m*`,`*n*`}` | Between *m* and *n* (inclusive) repetitions of the preceding regular expression |

Using the brace notation, we can collapse the repeated elements of recognition sites into very brief expressions. SecI's sites can be represented as `'CC[TCAG]{2}GG'`, CjuI's 4,096 recognition sequences can be represented as `'CA[CT][TCAG]{5}[GA]GG'`, and the sequences XcmI recognizes can be expressed as `'CCA[TCAG]{9}TGG'`—that's 262,144 possibilities represented as a 15-character regular expression!

Restriction enzyme recognition sites are fixed-length sequences. Variation in what sequences a particular enzyme recognizes arises from positions that can match two, three, or four bases instead of just one. While the number of possible combinations within a fixed-length sequence might be enormous, the number of bases to be matched is fixed for any restriction enzyme.

However, regular expressions are far more general than that. Except for {*m*}, the repetition characters produce regular expressions that can match targets of different lengths. We'll need more general kinds of targets to match against, so for now we'll use arbitrary base and amino acid sequences. Table 7-7 gives some examples.

*Table 7-7. Repetition characters in regular expressions*

| Pattern | Matches |
|---|---|
| `CC[TCAG]{2}GG` | CC, followed by any two DNA bases, followed by GG |
| `(TA){3,8}` | Between three and eight repetitions of TA, inclusive |
| `[GC]*` | Zero or more Gs and Cs (in any combination) |
| `A+` | One or more As |
| `AT?AA` | AAA or ATAA only |

## Greedy Versus Nongreedy Matching

The repetition-matching characters listed in Table 7-7 match the maximum possible portion of the target, provided the rest of the regular expression matches the rest of the target. This is often referred to as *greedy matching*. But much of the time—perhaps most of the time in bioinformatics—greedy matches absorb a far larger portion of the target than was intended.

Suppose you want to search an RNA string for the first occurrence of the polyadenylation signal `AAUAAA`. The regular expression `r'[UCAG]+AAUAAA[UCAG]+'` will match from the beginning of the string to the *last* occurrence of `AAUAAA`. That is because the initial `[UCAG]+` grabs as much of the string as it can, which, since it includes all possible bases, is nearly the entire sequence. All that remains following that part of the match is the last occurrence of `AAUAAA` and one or more bases after it.

Adding a question mark after one of `*`, `+`, or `?` changes the behavior to minimal possible matching, or *nongreedy matching*. The nongreedy matching characters are listed in Table 7-8. To return to the example of trying to find the first `AAUAAA` in an RNA sequence, adding a question mark after each of the plus signs in the pattern will change it so that

it matches the *first* occurrence instead of the last. The pattern would be `r'[UCAG]+?AAUAAA[UCAG]+?'`.

*Table 7-8. Nongreedy regular expression repetition characters*

| Characters | Matches |
|---|---|
| `*?` | Zero or more repetitions of the preceding regular expression, nongreedily |
| `+?` | One or more repetitions of the preceding regular expression, nongreedily |
| `??` | Zero or one repetitions of the preceding regular expression, nongreedily (rare) |

The concept of nongreedy matching is a confusing one. When a pattern that contains repetition characters is matched against a target string, there are often many different ways the pattern can match. For example, the pattern `r'[TA]+[UTAG]*` could match the target `'TATATATA'` by matching `[TA]+` against all four `TA`s and `[UTAG]*` against no characters, `[TA]*` against three `TA`s and `[UTAG]*` against the final `TA`, and so on.

The definition of greedy matching is that each variable portion of the pattern will match against as *much* of the target as it can, as long as the rest of the pattern can match the rest of the string. In nongreedy matching, each variable portion of the pattern will match as *little* of the target as possible as long as the rest of the pattern can match the rest of the string.

## Grouping and Disjunction

A portion of a regular expression can be enclosed inside parentheses. (To use `(` or `)` as a literal, either precede it with a backslash or put it inside square brackets.) This has much the same effect as using parentheses in algebraic expressions. A repetition character that follows a parenthesized regular expression indicates the repetition of the expression inside the parentheses.

Parentheses do more than just group parts of a regular expression, though. The parts of the target that match groups in a regular expression are assigned numbers in sequence (starting with 1). One part of a regular expression can refer to an earlier part using the syntax `\i`, where *`i`* is the group number. Suppose you wanted to match a date in the format YYYY-MM-DD, but allow any of several characters to be used in place of the hyphen as long as the same character is used in both places. Here's one way to work this out:

```
'\d\d\d\d-\d\d-\d\d'                    # brute-force match using hyphen
'\d{4}-\d{2}-\d{2}'                     # clearer
1 '\d{4}[-.,:']\d{2}[-.,:']\d{2}'       # various punctuation, can be different
2 '\d{4}([-.,:'])\d{2}\1\d{2}'          # punctuation must be the same
```

Remember that when including a hyphen in a character set, it must be the first character; otherwise, it is interpreted as indicating a range.

The last step in the preceding sequence enclosed the first character set in parentheses. This allowed a later "back reference" to whatever the set matched in the target, designated by `\1`.

Groups have a much more important use than back references, though: after a match has succeeded, the characters of the target that correspond to each group can be obtained from the match object returned by functions and methods that perform matches. This is a very powerful capability, which we'll explore as we go along—not just in the rest of this chapter, but as we use regular expressions in subsequent chapters as well.

The character | between two regular expressions is similar to `or` in Boolean expressions. It says to try a match with the first expression and then, if it fails, try the second. Any number of regular expressions can be separated by vertical bars. Each one is tried in turn until one matches or there are none left to try. If you enclose the whole sequence inside parentheses, later parts of the same regular expression and the resulting match object can tell what part of the target matched that group without even knowing which of the alternatives was matched.

# The Actions of the re Module

The `re` module provides functions and methods that use regular expressions for a variety of actions. Matching and searching functions and methods return *match objects* when successful; otherwise, they return `None`. We'll discuss match objects a bit later. For now, we can just treat them as true values.

## Functions

First, we'll look at some of the `re` module's functions (the meaning of the *`flags`* arguments to these functions is described in the next section):

`re.match(`*`pattern`*`,` *`target`*`[,` *`flags`*`])`
: Returns a match object if the regular expression *`pattern`* matches zero or more characters starting at the beginning of the *`target`* string, under the control of the *`flags`* value

`re.search(`*`pattern`*`,` *`target`*`[,` *`flags`*`])`
: Returns a match object for the first place in the *`target`* string that *`pattern`* matches

`re.findall(`*`pattern`*`,` *`target`*`[,` *`flags`*`])`
: Returns a list of all nonoverlapping matches in *`target`* as a list of strings or, if the pattern included groups, a list of lists of strings

`re.finditer(`*`pattern`*`, `*`target`*`[, `*`flags`*`=0])`
: Returns an iterator that produces a match object for each place in *`target`* that *`pattern`* matches

`re.split(`*`pattern`*`, `*`target`*`[, `*`maxsplit`*`=0[, `*`flags`*`=0]])`
: Returns a list of strings obtained by splitting the target at each place where *`pattern`* matches, up to *`maxsplit`* splits (all if `0` or omitted)

`re.sub(`*`pattern`*`, `*`replacement`*`, `*`target`*`[, `*`count`*`=0[, `*`flags`*`=0]])`
: Returns the string obtained by replacing each match of *`pattern`* to *`target`* with *`replacement`*, up to *`count`* matches (all if `0` or omitted)

`re.subn(`*`pattern`*`, `*`replacement`*`, `*`target`*`[, `*`count`*`=0[, `*`flags`*`=0]])`
: Performs the same actions as `re.sub`, but returns a tuple (*`newstring`*`, `*`number`*), where *`newstring`* is the string with the replacements made and *`number`* is the number of substitutions made

The module also provides two important functions that don't do matches or searches:

`re.escape(`*`string`*`)`
: Returns a string with all characters other than letters and digits preceded by slashes, for when you want to use a string as a literal regular expression pattern and don't want any characters taking on special meanings

`re.compile(`*`pattern`*`[, `*`flags`*`])`
: Returns a *regular expression object* constructed from *`pattern`* and *`flags`*; the significance of this function will be discussed shortly

## Flags

Some details of matching and searching are controlled by flags. Table 7-9 describes the `re` module's flag values. You can refer to a flag by either its short or its long form.

*Table 7-9. Regular expression match and search flags*

| Long form | Short form | Effect |
|---|---|---|
| `re.ASCII` | `re.A` | Restrict `\w`, `\W`, `\b`, `\B`, `\s`, and `\S` to matching based on ASCII characters only, not the full Unicode character set |
| `re.IGNORECASE` | `re.I` | Ignore case |
| `re.LOCALE` | `re.L` | Make `\w`, `\W`, `\b`, `\B`, `\s`, and `\S` dependent on the local language |
| `re.DOTALL` | `re.S` | Allow `.` (dot) to match a newline |
| `re.MULTILINE` | `re.M` | Allow `^` to match at the beginning of each line in addition to the beginning of the string and `$` to match at the end of each line (after the newline) in addition to the end of the string |
| `re.VERBOSE` | `re.X` | Permit writing of formatted regular expressions that can even include comments |

Functions in the module that take flag arguments expect a single value, not an arbitrary number of flags or a sequence of flags. When you need to specify multiple flags, you must use the | operator to combine them. The | operator—in an algebraic context, not within a regular expression—performs a *bitwise logical or*. Instead of treating the operands as two values that are each either true or false, bitwise logical operations treat each *bit* of each operand as a separate true or false value. For example, the bitwise or of 12 and 6 is 14:

```
   12    1100
|   6    0110
=  14    1110
```

It is a common computing technique to define a set of flags that are each integers with just a single bit. That allows the bitwise or operator to act as a set union operator for the flags, so that any combination of flags can be passed as a single value. For example, `re.I` is `2` (binary `00010`), `re.M` is `8` (binary `01000`), and `RE.S` is `16` (binary `10000`), so if you wanted to pass the combination of the three flags as an argument you would pass the expression `re.I | re.M | re.S`. (The order doesn't matter.) The binary result of that expression is `11010`, but using the names for the flags and the | operator allows you to ignore all that.

You probably won't use the `LOCALE` flag much. The `ASCII` flag introduces a limitation that is usually not necessary, except perhaps to avoid confusion in your mind about which Unicode characters are "space" or "word" characters. (This is an intricate topic that you'll probably want to stay away from if you can.) The `IGNORECASE` flag, as it suggests, ignores the case of letters in the target when performing the match.

You will use `DOTALL` and `MULTILINE` quite frequently, especially when using regular expressions to process sequence data. Without `DOTALL`, periods won't match newlines. Without `MULTILINE`, `^` and `$` will match only at the beginning and end of the target, respectively. When matching against a target string with internal newline characters, such as a sequence read directly from a FASTA file, you need these flags (specified as `re.DOTALL | re.MULTILINE` or `re.S | re.M`). Matching text containing URLs raises the same problem—you can write a regular expression to match hyperlinks (and we will), but there's no requirement in HTML that they appear entirely on one line.

The `VERBOSE` flag is quite different from the others. It doesn't control specific details of the match, but rather lets you write regular expressions in a more readable format, much like regular code. Unless preceded by a backslash, whitespace and comment characters are ignored.

Example 7-2 shows a regular expression that could be used to match decimal numbers. Note that the space after the `d` on the first line is ignored: *all* whitespace is ignored unless preceded by a backslash. Without the `VERBOSE` flag set, and assuming therefore that the comments and newlines are removed, the beginning of the pattern would read "one digit followed by one or more spaces" instead of "one or more digits."

*Example 7-2. A VERBOSE regular expression for decimal numbers*

```
r"""\d +  # match one or more digits (remember to use raw strings!)
    \.    # match a decimal point (not any character, because backslashed)
    \d *  # match 0 or more digits"""
```

## Methods

The `re.compile` function returns a regular expression object that encapsulates a data structure built from the pattern and controlled by the flags provided as arguments. Regular expression objects have methods corresponding to the module's `match` and `search` functions. Some of them have *`startpos`* and *`endpos`* arguments that provide more control over matches and searches than the corresponding module functions. Using the compiled version is more efficient when the regular expression is used more than once, since otherwise the regular expression will have to be compiled each time it is used.

The parameters of match object methods are different from those of the corresponding module functions. Since the pattern and flags are incorporated into the regular expression object, they aren't included in the parameters. Compiled regular expression objects have the following methods:

`match(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns a match object if the regular expression matches zero or more characters starting at the beginning of *`target`*

`search(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns a match object for the first match in the *`target`* string

`findall(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns a list of all nonoverlapping matches in *`target`* as a list of strings or, if the pattern included groups, a list of lists of strings

`finditer(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns an iterator that produces a match object for each match

`split(`*`target`*`[, `*`maxsplit`*`=0])`
: Returns a list of strings obtained by splitting the target at each match, up to *`maxsplit`* splits (all if `0` or omitted)

`sub(`*`replacement`*`, `*`target`*`[, `*`count`*`=0])`
: Returns the string obtained by replacing each match with *`replacement`*, up to *`count`* matches (all if `0` or omitted); *`replacement`* can be a function, in which case it is called with the result of each match and its return value is used to replace the corresponding match in *`target`*

`subn(`*`replacement`*`, `*`target`*`[, `*`count`*`=0])`
: Same as `re.sub`, but returns not just the string, but a tuple (*`string`*`, `*`number`*) where *`number`* is the number of matches; *`replacement`* can be a function, in which case it

is called with the result of each match and its return value is used to replace the corresponding match in *target*

# Results of re Functions and Methods

Many of the `re` module's functions and regular expression object methods return match objects. These provide a wealth of information about the successful matches, some accessible as fields and some by calling methods.

The most powerful feature of match objects is that they capture the parts of each match in a target string that corresponds to one of the pattern's parenthesized groups. As a result, regular expressions and the functions and methods that use them can give much more than a yes or no answer to matches and searches. The groups captured in the match objects are an analysis of the structure of the matched string. We'll see how this works in examples later in this chapter.

## Match Object Fields

Match object fields are accessed directly using dot notation. That is, if *`mobj`* is a match object, the `startpos` field will be accessed as *`mobj`*`.startpos`. The fields include:

`startpos`
: The value of `startpos` passed to the function or method that returned the match object

`endpos`
: The value of `endpos` passed to the function or method that returned the match object

`re`
: The regular expression passed to the function or method that returned the match object

`string`
: The target passed to the function or method that returned the match object

`lastindex`
: The index (integer) of the last group in the match, or `None` if there were no groups

## Match Object Methods

Match object methods include the following. In any match object method with group numbers as parameters, `0` refers to the entire match, `1` to the first parenthesized group in the match, `2` to the second, and so on:

`group([`*`groupnumber1`* `[,` *`groupnumber2`* `[, ....]]])`
: With no arguments, returns the entire match; with one argument returns the string that matched the corresponding group number, and with multiple arguments returns a tuple with the strings corresponding to the specified groups

`groups([`*`default`*`])`
: Returns a tuple containing all the groups of the match; for any group in the pattern that was not used in matching the target, the value in the tuple will be *`default`* (or `None` if not specified)

`start([`*`groupnumber`*`])`
: Returns the starting position of the target string that was matched by the group with index *`groupnumber`*; with no arguments, returns the whole part of the target that was matched

`end([`*`groupnumber`*`])`
: Returns the ending position of the target string that was matched by the group with index *`groupnumber`*; with no arguments, returns the whole part of the target that was matched

`span([`*`groupnumber`*`])`
: Returns the pair (*`startpos`*`,` *`endpos`*) corresponding to the range of the target string that was matched by the group with index *`groupnumber`*; with no arguments, returns the first and last position of the whole part of the target that was matched

`expand(`*`string`*`)`
: Performs "backslash substitution" like the regular expression object `sub` method, but also replaces each occurrence of `\1`, `\2`, etc., in *`string`* with the value of the corresponding matched group

# Putting It All Together: Examples

Now that we've looked at the features of the `re` module in some detail, it's time to put them together and see how they work in some useful examples.

## Some Quick Examples

We'll start with a few simple examples to give you a more concrete idea of how to use the `re` module's facilities; then we'll move on to more substantial examples.

### Using regular expressions to ignore case

The `find` and `index` methods of `str` do not provide any way to search while ignoring letter case. In some of our examples, we got around that problem by converting the target string to all uppercase or all lowercase. This is fine for simple cases, but it's awkward for situations in which you want to obtain information from the target string in its original case: you'd have to first search a single-case copy for the positions where

what you are looking for are found, then use those positions to extract the corresponding substrings from the original.

Regular expression searching provides a simple solution: just use the `re.IGNORE` flag. Despite their general power, there is no reason to reserve regular expressions for complicated situations. If you want to search `target` for `string`, ignoring case, this expression is all you need:

```
re.search(string, target, re.IGNORE)
```

### Listing files in a directory, excluding some

Chapter 6 showed a function for listing the contents of a directory while excluding files with certain extensions. Example 7-3 shows a more flexible version of this function based on a list of file patterns instead of just file extensions. The function joins regular expressions in the list into a single pattern with the disjunction character | separating them.

*Example 7-3. Directory listing filtered by regular expressions*

```
def ls(path = ".", ignorepats = (r'.+\.pyc$', r'.+~$', r'^\#')):
    if ignorepats:
        pat = re.compile('|'.join(ignorepats))
        # construct an RE disjunction containing each item of
        # ignorepats to create an RE that matches any of the items
    for filnam in os.listdir(path):
        if not ignorepats or not pat.search(filnam):
            print(filnam)
```

### Finding open reading frames

Regular expressions are ideal for all sorts of sequence feature detection. For example, they dramatically simplify finding open reading frames when compared to the kind of loops we saw in the examples of Chapter 4. We don't even have to worry about an extra base or two left over at the end of the sequence, the way we did in the loop-based code. Here's all we need:

```
openpat = re.compile('''
    ([TCAG]{3})*?           # 0 or more codons
    (ATG                    # start codon; begin match group
     ([TCAG]{3})*?          # 0 or more codons
     )                      # end match group
    (TAA|TGA|TAG)           # a stop codon
''', re.I | re.X )
```

The two occurrences of `([TCAG]{3})*?` force the match to consider three bases at a time. The pattern `[TCAG]{3}` is equivalent to `[TCAG][TCAG][TCAG]`. The first occurrence consumes codons that precede the `ATG` that begins the open reading frame, three at a time. The second finds the codons within the open reading frame. The parentheses around the first one are there not because we care about the result, but simply to make the

`*?` apply to the codon characters, since regular expressions don't allow one repetition indicator to directly follow another.

It would be nice if we could just use `findall` to find all the matches, but that would find the `ATG`s in *any* reading frame—for the same sequence, it might find one in reading frame 1 and another in reading frame 3, which makes no sense. We need to specify the reading frame we want the regular expression to use, so instead of `findall` we'll use `match`, which will force the regular expression to always start its match at the beginning of the string. We still need a loop to start each `match` at the end of the previous one and to collect the results, though. Example 7-4 shows a function that uses the compiled regular expression to find all the open reading frames starting in a given frame.

*Example 7-4. Finding open reading frames with a regular expression*

```
def open_reading_frames(seq, frame=1):
    lst = []
    matchobj = openpat.match(seq, frame-1)
    while m:
        lst.append(m.group(2))
        matchobj = openpat.match(seq, matchobj.end())
    return lst
```

An even more compact version can be written as a generator, as shown in Example 7-5.

*Example 7-5. Regular-expression-based open reading frame generator*

```
def open_reading_frames_generator(seq, frame=1):
    matchobj = openpat.match(seq, frame-1)
    while matchobj:
        yield matchobj.group(2)
        m = openpat.match(seq, matchobj.end())
```

The expression `open_reading_frames_generator(seq, frame)` will return a generator, and every time `next` is called on that generator the next open reading frame will be returned.

## Extracting Descriptions from Sequence Files

The web page at *http://rebase.neb.com/rebase/rebase.seqs.html* has links for viewing and downloading DNA and protein sequences for restriction enzymes. The sequences are grouped into files according to enzyme category, but the format of the files is consistent. Assume we've downloaded one of the DNA files to the file *rebaseseqs.txt*, and we want to extract the description data for each of its sequences.

The description data in these files has a rigid format:

```
>EnzymeName   RecognitionSequence  SequenceLength Type
```

The number of spaces separating the fields is fixed: 3, 2, and 1. Not all sequences in these files designate a `RecognitionSequence` field. Examples in Chapters 3 and 4 showed a variety of ways of approaching a problem like this using comprehensions, loops, and

iterations. We're going to see now how much simpler a solution using regular expressions can be. If we can define an appropriate regular expression, we can use `re.findall` to find all the descriptions, doing the repetitions for us that we would otherwise have to program with a comprehension, loop, or iteration. Not only that, but a regular expression with grouping will even parse the results into their fields.

The first step is to create a regular expression that matches the whole description line. To simplify things, we'll assume that the contents of the datafile have been read and named `data`:

```
re.findall(r'^>.*$', data, re.MULTILINE)
```

This regular expression matches target substrings characterized by:

- `>` at the beginning of a line
- Followed by any characters
- Until the end of the line

We use `re.MULTILINE` so `^` and `$` can match lines within `data`; otherwise, they would match only at the beginning and end of the whole string. Using this pattern with `re.search` returns a list of strings such as:

```
'>M.AacDam   GATC  855 nt'
```

The next step is to match the fields of the data individually.

```
re.findall(r'^>[^ ]+   [\w]*  \d+ .+$', data, re.M)
```

When you are first learning about them, even a regular expression as basic as this one looks quite cryptic. When you encounter one, try writing down what you think it means. Just go from left to right, as you would with an algebraic expression. This one is read as;

- A `>` at the beginning of a line
- One or more nonspace characters, captured as a group
- Exactly three spaces
- Zero or more word characters, captured as a group
- Exactly two spaces
- One or more digits, captured as a group
- Exactly one space
- One or more characters at the end of the line, in a group
- The end of the line

You can also use the `re.VERBOSE` (`re.X`) flag and comment the expression itself. Just be sure to put backslashes in front of any spaces inside the regular expression, because with that flag *all* whitespace—space, tabs, newlines—is ignored, apart from whitespace

indicated by escaped characters (\ \n\t). Here's how we could combine our regular expression with the explanation that follows it:

```
    r'''
^>          # a > at the beginning of the line
([^\ ]+)    # one or more nonspace characters, captured as a group
\ \ \       # exactly three spaces
([\w]*)     # zero or more word characters, captured as a group
\ \         # exactly two spaces
(\d+)       # one or more digits, captured as a group
\           # exactly one space
(.+?)$      # one or more characters at the end of the line, in a group
'''
```

The result returned by `findall` is the same as for the earlier regular expression. We take this step in developing the regular expression to make sure that we have the pieces described accurately. For instance, if there were only two spaces after the first `+`, there wouldn't be any matches. In the next step we simply put parentheses around the parts we are interested in so that we can extract them from the match objects as groups:

```
re.findall(r'^>([^ ]+)   ([\w]*)  (\d+) (.+$)', data, re.M)
```

The result we get now is different, and more useful: instead of a list of strings, we get back a list of tuples of strings. Each tuple contains the portion of the matched string corresponding to one of the parenthesized portions of the regular expression. For the line:

```
'>M.AacDam   GATC  855 nt'
```

the tuple in the list of results would be:

```
('M.AacDam', 'GATC', '855', 'nt')
```

Earlier in the chapter, it was observed that regular expressions used with bioinformatics data usually contain nongreedy repetition characters. The reason we didn't need them here is that the pattern *anchored* the match to the beginning and end of a single line. Since the goal was to extract single lines and the information they contained, using greedy repetition characters did not create any problems in this case.

## Extracting Entries From Sequence Files

Also in Chapters 3 and 4 were a number of examples that involved reading descriptions and/or sequences from FASTA files. They required function definitions to use string methods to search and split either individual lines or the entire file contents. As a result they were somewhat complex.

### Using regular expressions to extract entries

Regular expressions make getting the next sequence from the file a much simpler task. To extract a description and all the sequence characters, the following suffices:

```
re.search(r'^>[^>]*', src).group()
```

This reads as "match the next > that starts a line through all characters that aren't a >." This takes care of the problem of needing to read the description line to find the end of the previous sequence. We'd like to do better than that, though. First we'll separate the description from the sequence, capturing two groups:

```
re.search(r'^>(.*)$([^>]+)', src, re.M).groups()
```

The plus at the end prevents the pattern from matching a (possibly partial) description line not followed by any sequence characters, which could happen at the end of the `src` string. The `$` serves to split the match into everything on the description line and everything else.

Then, all we have to do is use ordinary string functions to clean up each of the two groups returned. Example 7-6 shows a complete definition of `next_item` that doesn't use subfunctions to read the fields from the file. (The cleanup code, however, *is* in a separate function, so those details are out of the way of the basic matching loop.) In this example, `next_item` searches for the next part of the text that matches the compiled regular expression, then calls `extract_from_match` to process the result. If there is no match, `matchobj` will be `None` and `extract_from_match` will return `None`. Otherwise, it will return a tuple of the form `(`*`description`*`,`*`sequence`*`)`. The description, found in group 1 of the match object, is broken up into a tuple by splitting at the occurrences of | and stripping the spaces from the results. The sequence, found in group 2, has its newline characters removed.

*Example 7-6. Defining next_item for FASTA with a regular expression*

```
pat = re.compile(r'^>(.*)$([^>]+)', re.MULTILINE)

def next_item(src):
    return extract_from_match(pat.search(src))

def extract_from_match(matchobj):
    return (matchobj and                              # return None when match fails
            (([field.strip() for field in matchobj.group(1).split('|')],
              matchobj.group(2).replace('\n', '')),
              matchobj.end()))
```

A really powerful feature of this solution is that we can accommodate variations in the file format with only slight changes to either the pattern or the cleanup in `match_item`.

### Keeping track of the position between calls

Unfortunately, this definition of `next_item` has a major problem. Regular expression operations expect their arguments to be strings (or things like strings, such as bytes and bytearrays); they do not work with open files. The `src` parameter of `next_item` must therefore be a string representing the entire file contents. However, the preceding definition will return the same item each time it is called.

We can fix this problem by keeping track of where the last match ended and starting from there the next time `next_item` is called. This is called *state maintenance*. One way to keep track of a value in between function calls is:

1. Add a parameter for the previous value to the function.
2. Return the value along with the other value(s) the function returns.
3. Have each function supply the previous value as one of its arguments.
4. Assign a name to the value part of the result of the function.

This is probably easier to illustrate with an example than with a written description. Let's consider how we could implement a `findall` function for plain strings using `str.find`. Example 7-7 shows a straightforward iterative definition like many we've already seen. Each time the target substring is found, the function adds its position to a list it is accumulating and then searches again, starting at the next character after the substring.

*Example 7-7. A findall function for ordinary strings*

```
def findall(string, substring):
    """Return a list of all nonoverlapping positions in string where substring appears"""
    positions = []
    pos = string.find(substring)
    while pos >= 0:
        positions.append(pos)
        pos = string.find(substring, pos + len(substring))
    return positions
```

Example 7-8 shows an alternative approach with two functions: one just finds the next occurrence of the target, and the other calls it and collects the results. There's no particular advantage to doing things this way for this simple example; it simply illustrates the technique so we can use it for something more substantial next.

*Example 7-8. Keeping track of a value between calls*

```
def findnext(string, substring, startpos=0):
    """Return the position of the next nonoverlapping occurrence of
    substring in string, beginning at startpos where substring appears"""
    pos = string.find(substring, startpos)
    if pos < 0:
        return -1, -1
    else:
        return pos, pos + len(substring)

def findeach(string, substring):
    positions = []
    pos, startpos = findnext(string, substring)
    while pos >= 0:
        positions.append(pos)
        pos, startpos = findnext(string, substring, startpos)
    return positions
```

The important point to note about these two functions is that `findnext` needs to know where to start looking for the substring each time it is called. It's up to `findeach` to keep track of that next starting position between calls. We've defined `findnext` so that it returns two positions: the position where the substring starts and the position where search should recommence. In `findeach`, the two values returned from `findnext` are assigned, and the second is fed back into `findnext` when it is called. This example is so simple that `findeach` could compute where to start the next search as easily as `findnext` could, but if `findnext` does anything more interesting that might well not be the case.

Example 7-9 shows a definition of `next_item` that applies this technique to keep track of the search position between calls. It also shows an example of the function's use.

*Example 7-9. Defining next_item to return and receive a position*

```
def next_item(src, pos):
    return extract_from_match(pat.search(src, pos))

def extract_from_match(matchobj):
    return ((None, -1)                                  # pos doesn't matter here
            if not matchobj
            else
            (([field.strip() for field in matchobj.group(1).split('|')],
              matchobj.group(2).replace('\n', '')),
             matchobj.end()))                           # new value of pos to be remembered

def find_item(src, testfn):
    item, pos = next_item(src, 0)
    while (item):
        if testfn(item):
            return item
        item, pos = next_item(src, pos)
```

Another thing we can do is wrap `next_item` in a generator. Keeping track of the values of parameters and other names assigned in the function is one of the primary responsibilities of the generator mechanism. Another is keeping track of where the `yield` occurred in the definition (there may be several) so that execution can resume right after that statement when `next` is called on the generator object. Example 7-10 shows what that would look like.

*Example 7-10. Defining next_item as a generator*

```
def item_generator(src):
    pos = 0
    item, pos = next_item(src, pos)
    while item:
        yield item
        item, pos = next_item(src, pos)

def find_item(src, testfn):
    itemgen = item_generator(src)
    item = next(itemgen)
```

```
    while (item):
        if testfn(item):
            return item
        item = next(itemgen)
```

We've chosen to leave the definition of `next_item` unchanged from its definition in Example 7-9. Instead of adding complexity to `next_item` and `extract_from_match`, we've defined a second function to produce the generator object that keeps track of the position. Each time `next` is called on the generator object returned by `item_generator`, execution resumes in `item_generator` and continues until the `yield`. All this generator is doing is holding onto the value of `pos` between calls to `next_item`.

Notice also the changes in `find_item`. It no longer does anything to keep track of the position and provide it as an argument to `next_item`. In fact, it no longer calls `next_item` at all. Instead, it calls `item_generator` once at the beginning of the function, then uses `next(`*`itemgen`*`)` each time it wants another item.

Example 5-4 showed a class whose methods share state using instance fields. The problem here is not quite the same as the one solved by the class example. There, the emphasis was on avoiding a lot of useless argument-passing for values that were used unchanged by many methods. Here, the challenge is to *maintain* changing state between one function call and the next, not just *share* it. In the technique demonstrated in the preceding example, one function returns values to its caller that it will need again later, and the caller passes them back as arguments the next time it calls the function. State maintenance is even more of a reason to use a class than state sharing, and usually you would use a class rather than the tricks of the previous example. We'll see examples of a two such classes in Chapter 8 (in Examples 8-17 and 8-18).

### Buffering input

Another problem with these definitions is that code using them must read the entire contents of a file to pass `find_item`. If the file is extremely large this could take an inconvenient amount of time and space, especially if the sequence is located early in the data. And even if the intent is to do something with every sequence in a file, you still may not want to read the entire contents all at once.

The solution to this sort of problem is to interpose a *buffer* between the file and `next_item`. A buffer is a standard programming mechanism that turns processing a source or destination—reading from a file, writing to a file, searching a string, etc.—into a two-step process. A file buffer contains a moderate portion of the file's bytes or characters. Reading is done from the buffer, rather than from the file itself. When the end of the buffer is reached, its contents (or most of them, anyway) are discarded and another piece of the file is read into the buffer. For writing, the process is reversed: writing is directed to the buffer, and when it's full its contents are written out to the file and the buffer is emptied. (This is called *flushing* the buffer.)

The main reason for buffering is that input and output operations that involve hardware storage devices are many orders of magnitude slower than reading from and writing into the computer's memory. Buffering replaces many small external interactions with many small internal interactions and only the occasional bulk transfer from or to external storage. As noted in Chapter 6 (in "The Python runtime environment: sys" on page 213), `sys.stdout` is a buffered file stream for this reason.

Python's implementation of input and output streams includes buffering capabilities, but programs can also implement their own, higher-level buffering mechanisms. We can add a buffer to Example 7-10's definition of `item_gen`, enabling it to pass strings to `re` functions and methods without first having to read the entire contents of a huge file into memory. This will improve efficiency in the event that a call to `find_item` locates a targeted item after examining only a small portion of the file.

The strategy will be to read a number of characters from the file and read sequences from the string returned; then, when all the sequences of the string have been read, another chunk of characters will be read. What's tricky about this is that the string beginning with the last `>` is almost definitely incomplete; the rest of the sequence will be in the next chunk.

The process is illustrated in Figures 7-1, 7-2, and 7-3. It's not easy to develop code like this, but it shouldn't be too hard to understand.

Example 7-11 shows a `get_item` function added to manage the buffering.

*Example 7-11. Buffered regular expression search of a file*

```
def get_item(fil, buffer, pos, chunksize):
    """Return a possibly incomplete item along with the new value
    of buffer and the end position of the successful match"""
    item, endpos = next_item(buffer, pos)                # initialize loop
    while not item:                                      # look for next item
        chunk = fil.read(chunksize)                      # read next chunk
        if not chunk:
            return None                                  # end of file
        buffer += chunk                                  # add chunk to buffer
        item, endpos = next_item(buffer, pos)            # try again
    return item, buffer, endpos                          # beginning of an item
```

The purpose of `get_item` is to call `next_item` until it returns an item. Each time `next_item` fails to find an item, `get_item` reads `chunksize` characters from the file and appends them to the buffer. When enough of the file has been read into the buffer for `next_item` to find an item, `get_item` returns the item, the (possibly extended) buffer, and the ending position of the match. This follows the approach discussed earlier, whereby values are maintained between calls by returning them to the caller and having the caller pass them back in. In this case, two values are tracked: the buffer itself and the position within the buffer where the search should resume.

The end of the buffer might contain a FASTA description and the beginning of its sequence, but not the entire sequence. How can we know whether `get_item` has

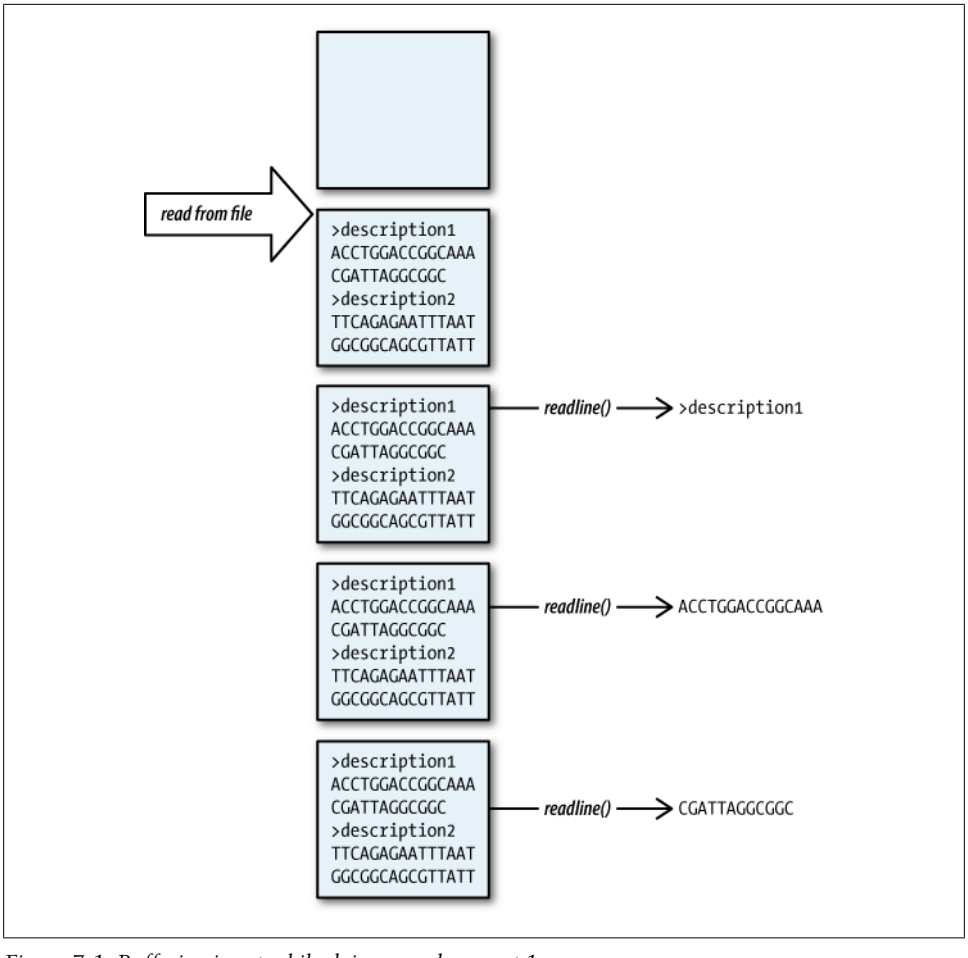



*Figure 7-1. Buffering input while doing searches, part 1*

returned a complete sequence? The solution is to always read a second sequence ahead of the one to be returned. As long as the beginning of another sequence is found, we know the current one is complete. When the current sequence is the last in the buffer, looking for the next one will cause the buffer to be extended until a next sequence is found. At that point, the current sequence must be reread to get its full contents.

The outline of the code is similar to a lot of loop-based functions: it reads an item, then loops until there are no more items. In this case, however, the code starts by reading *two* items and loops until there is no more *second* item. The technical term for reading one item beyond the targeted one is *lookahead*, for obvious reasons. Lookahead is required whenever some information from the next item is needed to process the current item (including, at a minimum, whether there *is* a next item). In the normal case, each time around the loop, the name of the first item is rebound to the second item and a new second item is read. This code also has to deal with the abnormal case when no

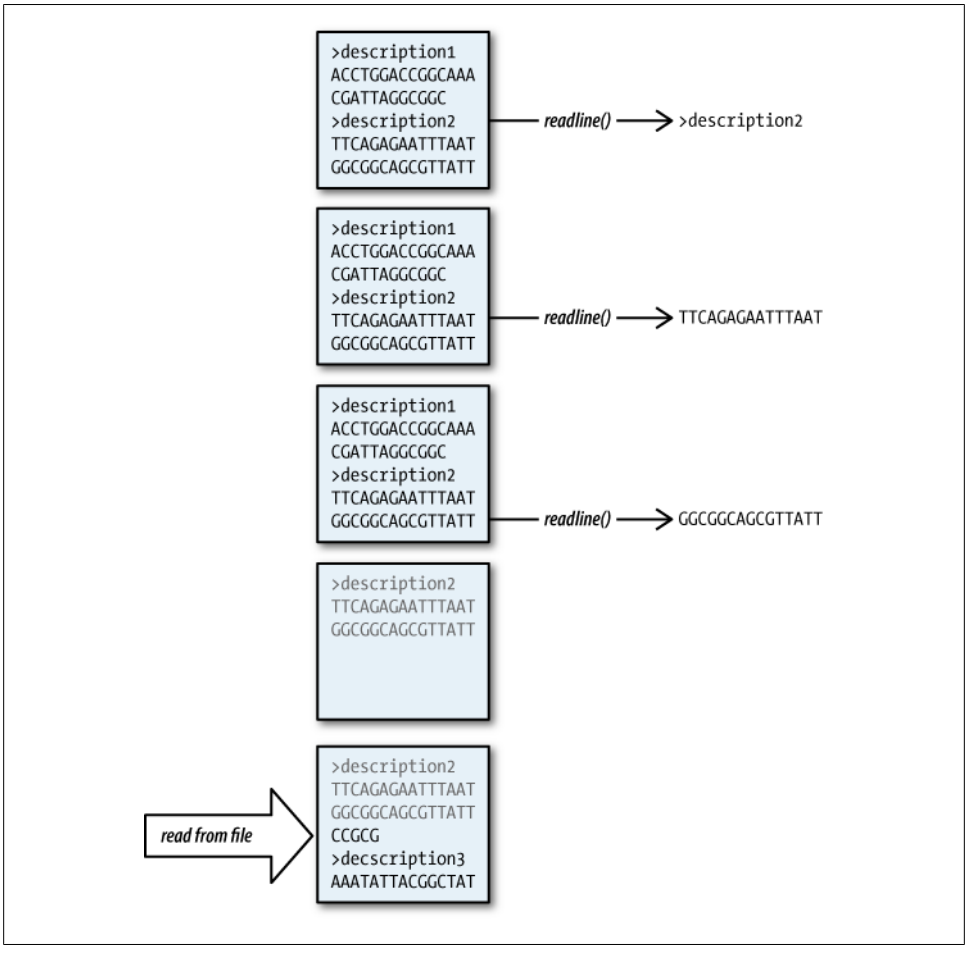



*Figure 7-2. Buffering input while doing searches, part 2*

second item is found, because that is when the buffer will be extended (except at the end of the file). The outline looks like this:

```
1 initialize
2 first item = read item
3 nextitem = read item
4 while nextitem:
5     if finding the next item caused the buffer to be enlarged
6         reread item from the buffer because it was incomplete
7         shorten buffer to exclude previously read items
8     yield item
9     item = nextitem
10    nextitem = read item
11 yield item       # at end of file
```

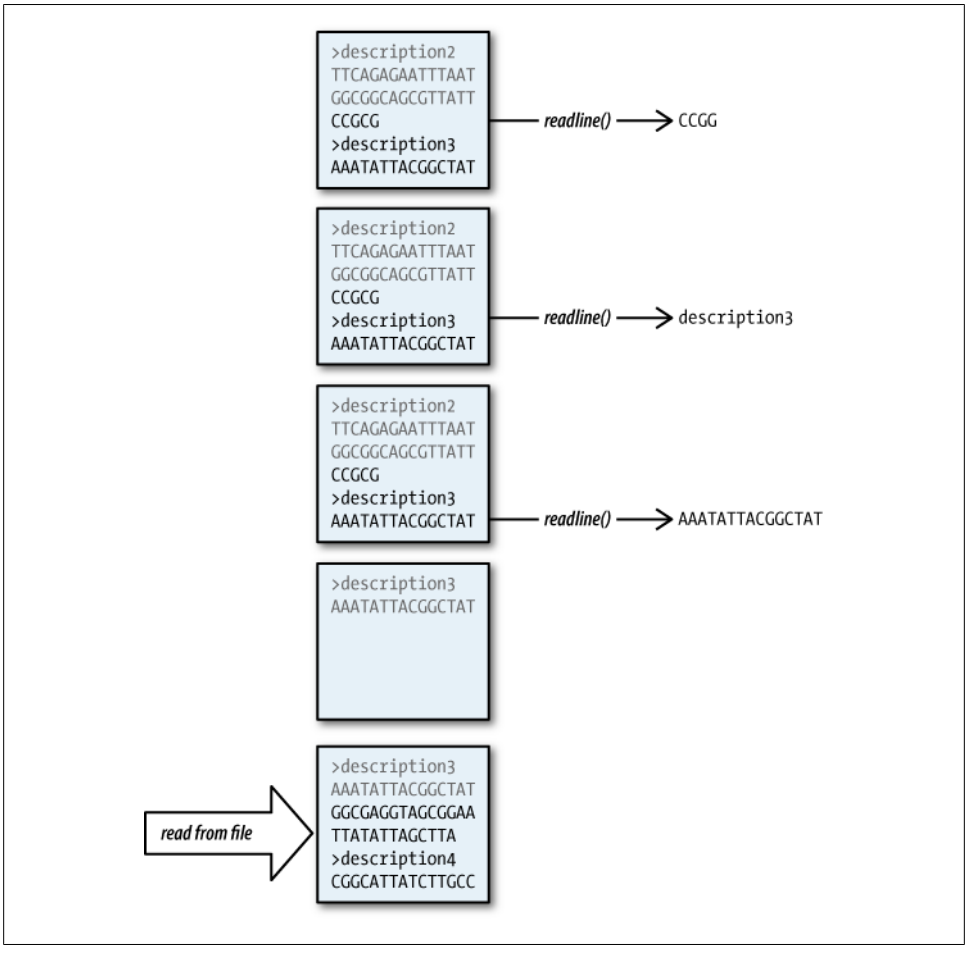



*Figure 7-3. Buffering input while doing searches, part 3*

Example 7-12 shows code that implements this outline. The solution is complicated by having to maintain three positions: the beginning of the current item (`curpos`), the beginning of the next item (`nxtpos`), and the end of the next item (`nxtend`). (At the end of the buffer, `nxtend` will equal the length of the buffer). A numbered line in the code, together with any following unnumbered lines, implements the correspondingly numbered part of the outline.

*Example 7-12. Buffered regular expression generator for a file*

```
def item_generator(file, chunksize=4000):
1     curpos = 0
2     itm, buffer, nxtpos = get_item(file, '', curpos, chunksize)
3     nxtitm, buffer, nxtend = get_item(file, buffer, nxtpos,
```

```
4     while nxtitm:
5         if len(buffer) > nxtend:
6             itm, nxtpos = next_item(buffer, curpos)

7             nxtend -= nxtpos

8         yield itm
9         itm, curpos, nxtpos = nxtitm, nxtpos, nxtend
10         nxtitm, buffer, nxtend = get_item(file, buffer, nxtpos,

11     yield next_item(buffer, pos)[0]
```

# Tips, Traps, and Tracebacks

## Tips

The importance of constructing regular expressions piece by piece cannot be emphasized enough. Make sure each pattern is working as expected before making it more complex. Does this sound familiar? It should: this book has repeatedly stressed this approach for defining functions. Unfortunately, regular expressions don't have the ability to call other regular expressions, so you can't really build them out of truly separate pieces. You can, however, still proceed in a step-by-step manner, increasing the complexity as you go.

Debugging regular expressions is far more difficult than debugging programs. You can't work your way step by step through the match the way you can with a Python debugger. There are tools for testing and debugging regular expressions, but even without one you can often accomplish a lot by simply printing the results of a `findall`, one to a line. Another trick that's useful when a regular expression isn't doing what you expect is to remove parts of it gradually and repeat the search until you locate the problem.

As you start building more complex regular expressions, you will experience more than a little frustration. Here are a few pointers:

- Use a regular expression testing tool such as re-try. You don't need the entire source file for testing your regular expression—just grab a section that will produce a match or two once your regular expression is working.
- Remember to use raw strings for regular expressions.
- In general, it's probably not worth trying to tune a regular expression so that a group such as FASTA sequence characters matches multiple lines but excludes the newlines. Just use `replace('\n', '')` on the result.
- Separating fields by a delimiter character such as a vertical bar on a FASTA description line can be done within the regular expression, but if you are having trouble accomplishing that, just have the expression match the whole line and use the following to get a list of the fields:

```
[field.strip() for field in line.split('|')]
```

- To search any of a list of patterns, search for:

```
'|'.join(list_of_patterns)
```

## Traps

- Remember to use the | operator to combine flags; `re` functions that take a flag argument, including `re.compile`, only accept one such argument. If you forget and use commas instead you will get an error message like the following:

```
TypeError: fn() takes at most 2 positional arguments (3 given)
```

  The exact numbers will depend on which function you called and how many arguments you supplied.
- Beware of vertical bars in regular expressions! They represent disjunction. If you are trying to match a vertical bar, as in FASTA headers, you must use a backslash (`\|`).
- When attempting to capture a group, make sure the repetition characters are inside the group unless you really do mean to repeat the group. That is, if you want a sequence of digits, write `(\d+)` and not `(\d)+`. The latter will return the last digit only, a rather mysterious result. This is an easy mistake to make, especially if you casually put parentheses around a small part of a partially completed regular expression.
- When matching the contents between "delimiter" characters—quotes, parentheses, brackets, braces, angle brackets, etc.—a * or +, and even a nongreedy `*?` or `+?`, will often match too much of the target string. In these situations you should match a repetition of "anything but the closing delimiter." For example, to match something enclosed in quotes, use `"[^"]+"` instead of `".+?"`.
- Make sure to use the `re.DOTALL` (`re.S`) flag if a newline can occur within text you want matched by `.*`, `.+`, `.*?`, or `.+?`, and not to use it if you don't want newlines matched.
- Make sure to use `re.MULTILINE` (`re.M`) if you are using anchors—`^` for the beginning of a line and `$` for the end. Otherwise, `^` will match only the beginning of the entire string, and `$` will match only the end.
- If you use `re.VERBOSE` (`re.X`), make sure to put a backslash in front of any space that you mean to be part of the pattern; otherwise, it will be ignored just like the whitespace you use for making the string readable.
- A negated character set such as `[^>]*` will match line feeds. This pattern:

```
r'^>[^>]*'
```

  will match an entire FASTA entry—the description and all of the sequence characters, with newlines embedded, regardless of whether `re.DOTALL` is specified.

If `re.MULTILINE` is specified, a dollar sign added to the end of that expression will have no effect because the character set will still match newlines. To prevent negated character sets from matching newlines you must include the newlines explicitly in the character set, either as `\n` or `\s` (which would exclude all whitespace).

## Tracebacks

- The same kinds of errors that occur when parentheses aren't balanced in code, especially in algebraic expressions, can easily happen with group parentheses in regular expressions, producing the error:

  ```
  sre_constants.error: unbalanced parenthesis
  ```

- The error:

  ```
  sre_constants.error: nothing to repeat
  ```

  can be particularly irksome. If you are developing a pattern incrementally, the obvious place to look for the source of this problem is where you last added a repetition character, or perhaps a pair of parentheses.

- The error:

  ```
  sre_constants.error: multiple repeat
  ```

  means that somewhere in the regular expression there are adjacent repetition characters. Perhaps the most likely culprit is putting the question mark before the plus sign or asterisk in a nongreedy repetition, instead of after. This problem can also arise when, while developing a regular expression, you accidentally remove the character(s) that separated two repetition characters.

- Unfortunately, the error won't point you to where in the expression the problem occurs; in fact, if you are using `re.VERBOSE` (`re.X`) it won't even show you which line to look at—the error always points to the end of the function call in which the expression appears. Note also that the line number of the offending function call appears early in the traceback, followed by many calls to `compile`, `parse`, and related subfunctions. This contrasts with most tracebacks, where it is the last few lines that show you where in your code the error occurred.