Diving In

Getting a small bit of text out of a large block of text is a challenge. In Python, strings have methods for searching and replacing: `index()`, `find()`, `split()`, `count()`, `replace()`, &c. But these methods are limited to the simplest of cases. For example, the `index()` method looks for a single, hard-coded substring, and the search is always case-sensitive. To do case-insensitive searches of a string `s`, you must call `s.lower()` or `s.upper()` and make sure your search strings are the appropriate case to match. The `replace()` and `split()` methods have the same limitations.

If your goal can be accomplished with string methods, you should use them. They're fast and simple and easy to read, and there's a lot to be said for fast, simple, readable code. But if you find yourself using a lot of different string functions with `if` statements to handle special cases, or if you're chaining calls to `split()` and `join()` to slice-and-dice your strings, you may need to move up to regular expressions.

Regular expressions are a powerful and (mostly) standardized way of searching, replacing, and parsing text with complex patterns of characters. Although the regular expression syntax is tight and unlike normal code, the result can end up being *more* readable than a hand-rolled solution that uses a long chain of string functions. There are even ways of embedding comments within regular expressions, so you can include fine-grained documentation within them.

> *If you've used regular expressions in other languages (like Perl, JavaScript, or PHP), Python's syntax will be very familiar. Read the summary of the re module to get an overview of the available functions and their arguments.*