Various utilities help to process a file in fine detail of a character, line, word, column, row, and so on, allowing us to manipulate a text file in many ways. Regular expressions are the core of pattern-matching techniques, and most of the text-processing utilities come with support for it. By using suitable regular expression strings, we can produce the desired output, such as filtering, stripping, replacing, and searching.

This chapter includes a collection of recipes, which walk through many contexts of problems based on text processing that will be helpful in writing real scripts.

Using regular expressions

Regular expressions are the heart of text-processing techniques based on pattern matching. For fluency in writing text-processing tools, one must have a basic understanding of regular expressions. Using wild card techniques, the scope of matching text with patterns is very limited. Regular expressions are a form of tiny, highly-specialized programming language used to match text. A typical regular expression for matching an e-mail address might look like `[a-z0-9_]+@[a-z0-9]+\.[a-z]+`.

If this looks weird, don't worry, it is really simple once you understand the concepts through this recipe.

How to do it...

Regular expressions are composed of text fragments and symbols, which have special meanings. Using these, we can construct any suitable regular expression string to match any text according to the context. As **regex** is a generic language to match texts, we are not introducing any tools in this recipe. However, it follows in the other recipes in this chapter.

Let's see a few examples of text matching:

- To match all words in a given text, we can write the regex as follows:

  ```
  ( ?[a-zA-Z]+ ?)
  ```

  `?` is the notation for zero or one occurrence of the previous expression, which in this case is the space character. The `[a-zA-Z]+` notation represents one or more alphabet characters (a-z and A-Z).

- To match an IP address, we can write the regex as follows:

  ```
  [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
  ```

  Or:

  ```
  [[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]
  {1,3}
  ```

We know that an IP address is in the form `192.168.0.2`. It is in the form of four integers (each from 0 to 255), separated by dots (for example, `192.168.0.2`).

`[0-9]` or `[:digit:]` represents a match for digits from 0 to 9. `{1,3}` matches one to three digits and `\.` matches the dot character (`.`).

> This regex will match an IP address in the text being processed. However, it doesn't check for the validity of the address. For example, an IP address of the form `123.300.1.1` will be matched by the regex despite being an invalid IP. This is because when parsing text streams, usually the aim is to only detect IPs.

How it works...

Let's first go through the basic components of regular expressions (regex):

regex	Description	Example
`^`	This specifies the start of the line marker.	`^tux` matches a line that starts with `tux`.
`$`	This specifies the end of the line marker.	`tux$` matches a line that ends with `tux`.
`.`	This matches any one character.	`Hack.` matches `Hack1`, `Hacki`, but not `Hack12` or `Hackil`; only one additional character matches.
`[]`	This matches any one of the characters enclosed in `[chars]`.	`coo[kl]` matches `cook` or `cool`.
`[^]`	This matches any one of the characters except those that are enclosed in `[^chars]`.	`9[^01]` matches `92` and `93`, but not `91` and `90`.
`[-]`	This matches any character within the range specified in `[]`.	`[1-5]` matches any digits from 1 to 5.
`?`	This means that the preceding item must match one or zero times.	`colou?r` matches `color` or `colour`, but not `colouur`.
`+`	This means that the preceding item must match one or more times.	`Rollno-9+` matches `Rollno-99` and `Rollno-9`, but not `Rollno-`.
`*`	This means that the preceding item must match zero or more times.	`co*l` matches `cl`, `col`, and `coool`.