More about regular expressions

You were previously introduced to regular expressions and a couple of symbols that you can use to construct regular expression patterns. In this reading, you'll explore additional regular expression symbols that can be used in a cybersecurity context. You'll also learn more about the `re` module and its `re.findall()` function.

Basics of regular expressions

A **regular expression (regex)** is a sequence of characters that forms a pattern. You can use these in Python to search for a variety of patterns. This could include IP addresses, emails, or device IDs.

To access regular expressions and related functions in Python, you need to import the `re` module first. You should use the following line of code to import the `re` module:

```
import re
```

Regular expressions are stored in Python as strings. Then, these strings are used in `re` module functions to search through other strings. There are many functions in the `re` module, but you will explore how regular expressions work through `re.findall()`. The `re.findall()` function returns a list of matches to a regular expression. It requires two parameters. The first is the string containing the regular expression pattern, and the second is the string you want to search through.

The patterns that comprise a regular expression consist of alphanumeric characters and special symbols. If a regular expression pattern consists only of alphanumeric characters, Python will review the specified string for matches to this pattern and return them. In the following example, the first parameter is a regular expression pattern consisting only of the alphanumeric characters `"ts"`. The second parameter, `"tsnow, tshah, bmoreno"`, is the string it will search through. You can run the following code to explore what it returns:

```
import re
re.findall("ts", "tsnow, tshah, bmoreno")
```

Reset

```
['ts', 'ts']
```

The output is a list of only two elements, the two matches to `"ts"`: `['ts', 'ts']`.

If you want to do more than search for specific strings, you must incorporate special symbols into your regular expressions.

Regular expression symbols

Symbols for character types

You can use a variety of symbols to form a pattern for your regular expression. Some of these symbols identify a particular type of character. For example, **`\w`** matches with any alphanumeric character.

Note: The **`\w`** symbol also matches with the underscore (**_**).

You can run this code to explore what **`re.findall()`** returns when applying the regular expression of **`"\w"`** to the device ID of **`"h32rb17"`**.

```
import re
re.findall("\w", "h32rb17")
```

Reset

```
['h', '3', '2', 'r', 'b', '1', '7']
```

Because every character within this device ID is an alphanumeric character, Python returns a list with seven elements. Each element represents one of the characters in the device ID.

You can use these additional symbols to match to specific kinds of characters:

- **`.`** matches to all characters, including symbols
- **`\d`** matches to all single digits [0-9]
- **`\s`** matches to all single spaces
- **`\.`** matches to the period character

The following code searches through the same device ID as the previous example but changes the regular expression pattern to **`"\d"`**. When you run it, it will return a different list:

```
import re
re.findall("\d", "h32rb17")
```

Reset

```
['3', '2', '1', '7']
```

This time, the list contains only four elements. Each element is one of the numeric digits in the string.

Symbols to quantify occurrences

Other symbols quantify the number of occurrences of a specific character in the pattern. In a regular expression pattern, you can add them after a character or a symbol identifying a character type to specify the number of repetitions that match to the pattern.

For example, the **+** symbol represents one or more occurrences of a specific character. In the following example, the pattern places it after the **`"\d"`** symbol to find matches to one or more occurrences of a single digit:

```python
import re
re.findall("\d+", "h32rb17")
```

Reset

```
['32', '17']
```

With the regular expression **`"\d+"`**, the list contains the two matches of **`"32"`** and **`"17"`**.

Another symbol used to quantify the number of occurrences is the ***** symbol. The ***** symbol represents zero, one, or more occurrences of a specific character. The following code substitutes the ***** symbol for the **+** used in the previous example. You can run it to examine the difference:

```python
import re
re.findall("\d*", "h32rb17")
```

Reset

```
['', '32', '', '', '17', '']
```

Because it also matches to zero occurrences, the list now contains empty strings for the characters that were not single digits.

If you want to indicate a specific number of repetitions to allow, you can place this number in curly brackets (**`{ }`**) after the character or symbol. In the following example, the regular expression pattern **`"\d{2}"`** instructs Python to return all matches of exactly two single digits in a row from a string of multiple device IDs:

```python
import re
re.findall("\d{2}", "h32rb17 k825t0m c2994eh")
```

Reset

```
['32', '17', '82', '29', '94']
```

Because it is matching to two repetitions, when Python encounters a single digit, it checks whether there is another one following it. If there is, Python adds the two digits to the list and goes on to the next digit. If there isn't, it proceeds to the next digit without adding the first digit to the list.

Note: Python scans strings left-to-right when matching against a regular expression. When Python finds a part of the string that matches the first expected character defined in the regular expression, it continues to compare the subsequent characters to the expected pattern. When the pattern is complete, it starts this process again. So in cases in which three digits appear in a row, it handles the third digit as a new starting digit.

You can also specify a range within the curly brackets by separating two numbers with a comma. The first number is the minimum number of repetitions and the second number is the maximum number of repetitions. The following example returns all matches that have between one and three repetitions of a single digit:

```
import re
re.findall("\d{1,3}", "h32rb17 k825t0m c2994eh")
```

Reset

```
['32', '17', '825', '0', '299', '4']
```

The returned list contains elements of one digit like `"0"`, two digits like `"32"` and three digits like `"825"`.

Constructing a pattern

Constructing a regular expression requires you to break down the pattern you're searching for into smaller chunks and represent those chunks using the symbols you've learned. Consider an example of a string that contains multiple pieces of information about employees at an organization. For each employee, the following string contains their employee ID, their username followed by a colon (`:`), their attempted logins for the day, and their department:

```
employee_logins_string = "1001 bmoreno: 12 Marketing 1002 tshah: 7 Human Resources 1003 sgilmore: 5 Finance"
```

Your task is to extract the username and the login attempts, without the employee's ID number or department.

To complete this task with regular expressions, you need to break down what you're searching for into smaller components. In this case, those components are the varying number of characters in a username, a colon, a space, and a varying number of single digits. The corresponding regular expression symbols are `\w+`, `:`, `\s`, and `\d+` respectively. Using these symbols as your regular expression, you can run the following code to extract the strings:

```
import re
pattern = "\w+:\s\d+"
employee_logins_string = "1001 bmoreno: 12 Marketing 1002 tshah: 7 Human Resources 1003 sgilmore: 5 Finance"
```

```
print(re.findall(pattern, employee_logins_string))
```

Reset

```
['bmoreno: 12', 'tshah: 7', 'sgilmore: 5']
```

Note: Working with regular expressions can carry the risk of returning unneeded information or excluding strings that you want to return. Therefore, it's useful to test your regular expressions.

Key takeaways

Regular expressions allow you to search through strings to find matches to specific patterns. You can use regular expressions by importing the **`re`** module. This module contains multiple functions, including **`re.findall()`**, which returns all matches to a pattern in the form of a list. To form a pattern, you use characters and symbols. Symbols allow you to specify types of characters and to quantify how many repetitions of a character or type of character can occur in the pattern.