Chapter No. 3
"Grouping"
In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.3 "Grouping"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

Félix López started his career in web development before moving to software in the currency exchange market, where there were a lot of new security challenges. Later, he spent four years creating an IDE to develop games for hundreds of different mobile device OS variations, in addition to creating more than 50 games. Before joining ShuttleCloud, he spent two years working on applications with sensor networks, Arduino, ZigBee, and custom hardware. One example is an application that detects the need for streetlight utilities in major cities based on existing atmospheric brightness. His first experience with Python was seven years ago, He used it for small scripts, web scrapping, and so on. Since then, he has used Python for almost all his projects: websites, standalone applications, and so on. Nowadays, he uses Python along with RabbitMQ in order to integrate services.

He's currently working for ShuttleCloud, an U.S.-based startup, whose technology is used by institutions such as Stanford and Harvard, and companies such as Google.

I would like to thank @panchoHorrillo for helping me with some parts of the book and especially my family for supporting me, despite the fact that I spend most of my time with my work ;)

For More Information:
**Víctor Romero** currently works as a solutions architect at MuleSoft, Inc. He started his career in the dotcom era and has been a regular contributor to open source software ever since. Originally from the sunny city of Malaga, Spain, his international achievements include integrating the applications present in the cloud storage of a skyscraper in New York City, and creating networks for the Italian government in Rome.

I would like to thank my mom for instilling the love of knowledge in me, my grandmother for teaching me the value of hard work, and the rest of my family for being such an inspiration. I would also like to thank my friends and colleagues for their unconditional support during the creation of this book.
Mastering Python Regular Expressions

Text processing has been one of the most relevant topics since computer science took its very first baby steps. After a few decades of investigation, we now have one of the most versatile and pervasive tools that exist: regular expressions. Validation, search, extraction, and replacement of text are operations that have been simplified thanks to Regular Expressions.

This book will initially cover regular expressions from a bird's-eye view, proceeding step-by-step to more advanced topics such as regular expression specifics on Python or grouping, workaround, and performance. All the topics will be covered with Python-specific examples that can be used straightaway in the Python console.

What This Book Covers

Chapter 1, Introducing Regular Expressions, will introduce the basics of the regular expression syntax from a non-Python-specific point of view.

Chapter 2, Regular Expressions with Python, will cover the Python's API for regular expressions and its quirks from a Python-specific point of view.

Chapter 3, Grouping, covers the regular expression functionality to extract portions of information, apply quantifiers to specific parts, and perform correct alternation.

Chapter 4, Look Around, explains the concept of zero-width assertions and the different types of look-around mechanisms.

Chapter 5, Performance of Regular Expressions, will cover different tools to measure the speed of a regular expression, the details of the regular expression module of Python, and different recommendations to improve the performance of regular expressions.

For More Information:
Grouping is a powerful tool that allows you to perform operations such as:

- Creating subexpressions to apply quantifiers. For instance, repeating a subexpression rather than a single character.
- Limiting the scope of the alternation. Instead of alternating the whole expression, we can define exactly what has to be alternated.
- Extracting information from the matched pattern. For example, extracting a date from lists of orders.
- Using the extracted information again in the regex, which is probably the most useful property. One example would be to detect repeated words.

Throughout this chapter, we will explore groups, from the simplest to the most complex ones. We'll review some of the previous examples in order to bring clarity to how these operations work.

**Introduction**

We've already used groups in several examples throughout *Chapter 2, Regular Expressions with Python*. Grouping is accomplished through two metacharacters, the parentheses `( )`. The simplest example of the use of parentheses would be building a subexpression. For example, imagine you have a list of products, the ID for each product being made up of two or three sequences of one digit followed by a dash and followed by one alphanumeric character, 1-a2-b:

```python
>>> re.match(r"(\d-\w){2,3}" , ur"1-a2-b")
<_sre.SRE_Match at 0x10f690738>
```

As you can see in the preceding example, the parentheses indicate to the regex engine that the pattern inside them has to be treated like a unit.
Let's see another example; in this case, we need to match whenever there is one or more ab followed by c:

```
>>> re.search(r"(ab)+c", ur"ababc")
<_sre.SRE_Match at 0x10f690a08>
>>> re.search(r"(ab)+c", ur"abbc")
None
```

So, you could use parentheses whenever you want to group meaningful subpatterns inside the main pattern.

Another simple example of their use is limiting the scope of alternation. For example, let's say we would like to write an expression to match if someone is from Spain. In Spanish, the country is spelled España and Spaniard is spelled Español. So, we want to match España and Español. The Spanish letter ñ can be confusing for non-Spanish speakers, so in order to avoid confusion we'll use Espana and Espanol instead of España and Español.

We can achieve it with the following alternation:

```
>>> re.search("Espana|ol", "Espanol")
<_sre.SRE_Match at 0x1043cfe68>
>>> re.search("Espana|ol", "Espana")
<_sre.SRE_Match at 0x1043cfed0>
```

The problem is that this also matches ol:

```
>>> re.search("Espana|ol", "ol")
<_sre.SRE_Match at 0x1043cfb28>
```

So, let's try character classes as in the following code:

```
>>> re.search("Espan[aol]", "Espanol")
<_sre.SRE_Match at 0x1043cfid0>

>>> re.search("Espan[aol]", "Espana")
<_sre.SRE_Match at 0x1043cf850>
```

It works, but here we have another problem: It also matches "Espano" and "Espan1" that doesn't mean anything in Spanish:

```
>>> re.search("Espan[a|ol]", "Espano")
<_sre.SRE_Match at 0x1043cfb28>
```
The solution here is to use parentheses:

```python
>>> re.search("Espan(a|ol)", "Espana")
<_sre.SRE_Match at 0x10439b648>

>>> re.search("Espan(a|ol)", "Espanol")
<_sre.SRE_Match at 0x10439b918>

>>> re.search("Espan(a|ol)", "Espan")
None

>>> re.search("Espan(a|ol)", "Espano")
None

>>> re.search("Espan(a|ol)", "ol")
None
```

Let's see another key feature of grouping, **capturing**. Groups also capture the matched pattern, so you can use them later in several operations, such as `sub` or in the regex itself.

For example, imagine you have a list of products, the IDs of which are made up of digits representing the country of the product, a dash as a separator, and one or more alphanumeric characters as the ID in the DB. You're requested to extract the country codes:

```python
>>> pattern = re.compile(r"(\d+)-\w+")
>>> it = pattern.finditer(r"1-a\n20-baer\n34-afcr")
>>> match = it.next()
>>> match.group(1)
'1'

>>> match = it.next()
>>> match.group(1)
'20'

>>> match = it.next()
>>> match.group(1)
'34'
```

In the preceding example, we've created a pattern to match the IDs, but we're only capturing a group made up of the country digits. Remember that when working with the `group` method, the index 0 returns the whole match, and the groups start at index 1.

Capturing groups give a huge range of possibilities due to which they can also be used with several operations, which we would discuss in the upcoming sections.
Backreferences

As we've mentioned previously, one of the most powerful functionalities that grouping gives us is the possibility of using the captured group inside the regex or other operations. That's exactly what backreferences provide. Probably the best known example to bring some clarity is the regex to find duplicated words, as shown in the following code:

```python
>>> pattern = re.compile(r"(\w+) \1")
>>> match = pattern.search("hello hello world")
>>> match.groups()
('hello',)
```

Here, we're capturing a group made up of one or more alphanumeric characters, after which the pattern tries to match a whitespace, and finally we have the \1 backreference. You can see it highlighted in the code, meaning that it must exactly match the same thing it matched as the first group.

Backreferences can be used with the first 99 groups. Obviously, with an increase in the number of groups, you will find the task of reading and maintaining the regex more complex. This is something that can be reduced with named groups; we'll see them in the following section. But before that, we still have a lot of things to learn with backreferences. So, let's continue with another operation in which backreferences really come in handy. Recall the previous example, in which we had a list of products. Now, let's try to change the order of the ID, so we have the ID in the DB, a dash, and the country code:

```python
>>> pattern = re.compile(r"(\d+)-(\w+)"")
>>> pattern.sub(r"\2-\1", "1-a\n20-baer\n34-afcr")
'a-1\nbayer-20\nafcr-34'
```

That's it. Easy, isn't it? Note that we're also capturing the ID in the DB, so we can use it later. With the highlighted code, we're saying, "Replace what you've matched with the second group, a dash, and the first group".

As with the previous example, using numbers can be difficult to follow and to maintain. So, let's see what Python, through the re module, offers to help with this.
Named groups

Remember from the previous chapter when we got a group through an index?

```python
>>> pattern = re.compile(r"(\w+) (\w+)"
>>> match = pattern.search("Hello→world")
>>> match.group(1)
'Hello'
>>> match.group(2)
'world'
```

We just learnt how to access the groups using indexes to extract information and to use it as backreferences. Using numbers to refer to groups can be tedious and confusing, and the worst thing is that it doesn't allow you to give meaning or context to the group. That's why we have named groups.

Imagine a regex in which you have several backreferences, let's say 10, and you find out that the third one is invalid, so you remove it from the regex. That means you have to change the index for every backreference starting from that one onwards. In order to solve this problem, in 1997, Guido Van Rossum designed named groups for Python 1.5. This feature was offered to Perl for cross-pollination.

Nowadays, it can be found in almost any flavor. Basically it allows us to give names to the groups, so we can refer to them by their names in any operation where groups are involved.

In order to use it, we have to use the syntax, `(?P<name>pattern)`, where the `P` comes from Python-specific extensions (as you can read in the e-mail Guido sent to Perl developers at [http://markmail.org/message/oyezhwvefrotacc3](http://markmail.org/message/oyezhwvefrotacc3)).

Let's see how it works with the previous example in the following code snippet:

```python
>>> pattern = re.compile(r"(?P<first>\w+) (?P<second>\w+)"
>>> match = re.search("Hello world")
>>> match.group("first")
'Hello'
>>> match.group("second")
'world'
```

So, backreferences are now much simpler to use and maintain as is evident in the following example:

```python
>>> pattern = re.compile(r"(?P<country>\d+)-(?P<id>\w+)"
>>> pattern.sub(r"\g<id>-\g<country>", "1-a\n20-baer\n34-afcr")
'a-1\nbaer-20\nafcr-34'
```

For More Information:

As we see in the previous example, in order to reference a group by the name in the sub operation, we have to use \g<name>.

We can also use named groups inside the pattern itself, as seen in the following example:

```python
>>> pattern = re.compile(r"(?P<word>\w+) \1")
>>> match = pattern.search(r"hello hello world")
>>> match.groups()
('hello',)
```

This is simpler and more readable than using numbers.

Through these examples, we've used the following three different ways to refer to named groups:

<table>
<thead>
<tr>
<th>Use</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inside a pattern</td>
<td>(?P=name)</td>
</tr>
<tr>
<td>In the repl string of the sub operation</td>
<td>\g&lt;name&gt;</td>
</tr>
<tr>
<td>In any of the operations of the MatchObject</td>
<td>match.group('name')</td>
</tr>
</tbody>
</table>

### Non-capturing groups

As we've mentioned before, capturing content is not the only use of groups. There are cases when we want to use groups, but we're not interested in extracting the information; alternation would be a good example. That's why we have a way to create groups without capturing. Throughout this book, we've been using groups to create subexpressions, as can be seen in the following example:

```python
>>> re.search("Españ(a|ol)", "Español")
<_sre.SRE_Match at 0x10e90b828>
>>> re.search("Españ(a|ol)", "Español").groups()
('ol',)
```
You can see that we've captured a group even though we're not interested in the content of the group. So, let's try it without capturing, but first we have to know the syntax, which is almost the same as in normal groups, (?:pattern). As you can see, we've only added ?:. Let's see the following example:

```python
>>> re.search("Españ(?:a|ol)", "Español")
<_sre.SRE_Match at 0x10e912648>
>>> re.search("Españ(?:a|ol)", "Español").groups()
()
```

After using the new syntax, we have the same functionality as before, but now we're saving resources and the regex is easier to maintain. Note that the group cannot be referenced.

**Atomic groups**

They're a special case of non-capturing groups; they're usually used to improve performance. It disables backtracking, so with them you can avoid cases where trying every possibility or path in the pattern doesn't make sense. This concept is difficult to understand, so stay with me up to the end of the section.

The `re` module doesn't support atomic groups. So, in order to see an example, we're going to use the `regex` module: [https://pypi.python.org/pypi/regex](https://pypi.python.org/pypi/regex).

Imagine we have to look for an ID made up of one or more alphanumeric characters followed by a dash and by a digit:

```python
>> data = "aaaaabbbbbbcccccdddadaaa"
>>> regex.match("(\w+)-\d", data)
```

Let's see step by step what's happening here:

1. The regex engine matches the first a.
2. It then matches every character up to the end of the string.
3. It fails because it doesn't find the dash.
4. So, the engine does backtracking and tries the same with the following a.
5. Start the same process again.
Grouping

It tries this with every character. If you think about what we're doing, it doesn't make any sense to keep trying once you have failed the first time. And that's exactly what an atomic group is useful for. For example:

```python
>>> regex.match(r"(?>(\w+)-\d", data)
```

Here we've added `?`, which indicates an atomic group, so once the regex engine fails to match, it doesn't keep trying with every character in the data.

### Special cases with groups

Python provides us with some forms of groups that can help us to modify the regular expressions or even to match a pattern only when a previous group exists in the match, such as an if statement.

### Flags per group

There is a way to apply the flags we've seen in Chapter 2, Regular Expressions with Python, using a special form of grouping: `(?iLm`sux)`.  

<table>
<thead>
<tr>
<th>Letter</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>re.IGNORECASE</td>
</tr>
<tr>
<td>L</td>
<td>re.LOCALE</td>
</tr>
<tr>
<td>m</td>
<td>re.MULTILINE</td>
</tr>
<tr>
<td>s</td>
<td>re.DOTALL</td>
</tr>
<tr>
<td>u</td>
<td>re.UNICODE</td>
</tr>
<tr>
<td>x</td>
<td>re.VERBOSE</td>
</tr>
</tbody>
</table>

For example:

```python
>>> re.findall(r"(?u)\w+" ,ur"ñ")
[u'ñ']
```

The above example is the same as:

```python
>>> re.findall(r"\w+" ,ur"ñ", re.U)
[u'ñ']
```

We've seen what these examples do several times in the previous chapter. Remember that a flag is applied to the whole expression.
yes-pattern|no-pattern

This is a very useful case of groups. It tries to match a pattern in case a previous one was found. On the other hand, it doesn't try to match a pattern in case a previous group was not found. In short, it's like an if-else statement. The syntax for this operation is as follows:

```
(?(id/name)yes-pattern|no-pattern)
```

This expression means: if the group with this ID has already been matched, then at this point of the string, the yes-pattern pattern has to match. If the group hasn't been matched, then the no-pattern pattern has to match.

Let's see how it works continuing with our trite example. We have a list of products, but in this case the ID can be made in two different ways:

- The country code (two digits), a dash, three or four alphanumeric characters, a dash, and the area code (2 digits). For example: 34-adrl-01.
- Three or four alphanumeric characters. For example: adrl.

So, when there is a country code, we need to match the country area:

```
>>>pattern = re.compile(r"((\d\d-)?\w{3,4})(\1{1}(-\d\d))")
```

As you can see in the previous example, there is a match when we have a country code and area code. Note that when there is a country code but no area code, there is no match:

```
>>>pattern.match("34-erte")
None
```

And what's no-pattern for? Let's add another constraint to the previous example: if there is no country code there has to be a name at the end of the string:

- The country code (2 digits), a dash, three or four alphanumeric characters, a dash, and the area code (2 digits). For example: 34-adrl-01
- Three or four alphanumeric characters, followed by three or four characters. For example: adrl-sala.
Grouping

Let's see it in action:

```python
>>> pattern = re.compile(r"(\d{3,4})-(?1)(\d\d)|[a-z]{3,4}"$

>>> pattern.match("34-erte-22")
<_sre.SRE_Match at 0x10f6ee750>
```

As expected, if there is a country code and an area code, there is a match.

```python
>>> pattern.match("34-erte")
None
```

In the preceding example, we do have a country area, but there is no area code, so there is no match.

```python
>>> pattern.match("erte-abcd")
<_sre.SRE_Match at 0x10f6ee880>
```

And finally, when there is no country area, there must be a name, so we have a match.

Note that no-pattern is optional, so in the first example, we've omitted it.

Overlapping groups

Throughout Chapter 2, Regular Expressions with Python, we've seen several operations where there was a warning about overlapping groups: for example, the `findall` operation. This is something that seems to confuse a lot of people. So, let's try to bring some clarity with a simple example:

```python
>>> re.findall(r"(a|b)+", "abaca")
['a', 'a']
```

What's happening here? Why does the following expression give us 'a' and 'a' instead of 'aba' and 'a'?  

---

For More Information:  
Let's look at it step by step to understand the solution:

As we can see in the preceding figure, the characters \texttt{aba} are matched, but the captured group is only formed by \texttt{a}. This is because even though our regex is grouping every character, it stays with the last \texttt{a}. Keep this in mind because it's the key to understanding how it works. Stop for a moment and think about it, we're requesting the regex engine to capture all the groups made up of \texttt{a} or \texttt{b}, but just for one of the characters and that's the key. So, how can you capture the groups made of several 'a' or 'b' in any order? The following expression does the trick:

```python
>>> re.findall(r'((?:a|b)+)', 'abbaca')
['abba', 'a']
```

We're asking the regex engine to capture every group made up of the subexpression \texttt{(a|b)} and not to group just one character.

One last thing on this — if we would want to obtain every group made of \texttt{a} or \texttt{b} with \texttt{findall}, we could write this simple expression:

```python
>>> re.findall(r'(a|b)', 'abaca')
['a', 'b', 'a', 'a']
```

In this case, we're asking the regex engine to capture a group made of \texttt{a} or \texttt{b}. As we're using \texttt{findall}, we get every pattern matched, so we get four groups.
Rule of Thumb

It's better to keep regular expressions as simple as you can. So, you should begin with the simplest expression and then build more complex expressions step by step and not the other way around.

Summary

Don't allow the simplicity of the chapter to fool you, what we have learned throughout this chapter will be very useful in your day-to-day work with regex, and it'll give you a lot of leverage.

Let's summarize what we have learned so far. First, we have seen how a group can help us when we need to apply quantifiers to only some part of the expression.

We have also learned how to use the captured groups in the pattern again or even in the replacement string in the \texttt{sub} operation, thanks to backreferences.

In this chapter, we have also viewed named groups, a tool for improving the readability and future maintenance of the regex.

Later on, we have learned to match a subexpression just in case a previous group exists or on the other hand, to match it when a previous group doesn't exist.

Now that we know how to use groups, it's time to learn a more complex subject very close to groups; look around!

For More Information:

\url{www.packtpub.com/mastering-python-regular-expressions/book}
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.