Case study: Parsing Phone Numbers

So far you've concentrated on matching whole patterns. Either the pattern matches, or it doesn't. But regular expressions are much more powerful than that. When a regular expression *does* match, you can pick out specific pieces of it. You can find out what matched where.

This example came from another real-world problem I encountered, again from a previous day job. The problem: parsing an American phone number. The client wanted to be able to enter the number free-form (in a single field), but then wanted to store the area code, trunk, number, and optionally an extension separately in the company's database. I scoured the Web and found many examples of regular expressions that purported to do this, but none of them were permissive enough.

\d matches any numeric digit *(0–9)*. *\D* matches anything but digits.

Here are the phone numbers I needed to be able to accept:

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212
- 1-800-555-1212
- 800-555-1212-1234
- 800-555-1212x1234
- 800-555-1212 ext. 1234
- work 1-(800) 555.1212 #1234

Quite a variety! In each of these cases, I need to know that the area code was `800`, the trunk was `555`, and the rest of the phone number was `1212`. For

those with an extension, I need to know that the extension was `1234`.

Let's work through developing a solution for phone number parsing. This example shows the first step.

```
import re
phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$')          #①
print (phonePattern.search('800-555-1212').groups())             #②
#('800', '555', '1212')

print (phonePattern.search('800-555-1212-1234'))                 #③
#None

print (phonePattern.search('800-555-1212-1234').groups())        #④
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 8, in <module>
# print (phonePattern.search('800-555-1212-1234').groups()) #\u2463
#AttributeError: 'NoneType' object has no attribute 'groups'
```

① Always read regular expressions from left to right. This one matches the beginning of the string, and then (`\d{3}`). What's `\d{3}?` Well, `\d` means "any numeric digit" (0 through 9). The `{3}` means "match exactly three numeric digits"; it's a variation on the `{n,m}` syntax you saw earlier. Putting it all in parentheses means "match exactly three numeric digits, *and then remember them as a group that I can ask for later*". Then match a literal hyphen. Then match another group of exactly three digits. Then another literal hyphen. Then another group of exactly four digits. Then match the end of the string.

② To get access to the groups that the regular expression parser remembered along the way, use the `groups()` method on the object that the `search()` method returns. It will return a tuple of however many groups were defined in the regular expression. In this case, you defined three groups, one with three digits, one with three digits, and one with four digits.

③ This regular expression is not the final answer, because it doesn't handle a phone number with an extension on the end. For that, you'll need to expand the regular expression.

④ And this is why you should never "chain" the `search()` and `groups()` methods in production code. If the `search()` method returns no matches, it returns `None`, not a regular expression match object. Calling `None.groups()`

raises a perfectly obvious exception: None doesn't have a `groups()` method.

(Of course, it's slightly less obvious when you get this exception from deep within your code. Yes, I speak from experience here.)

```
import re
phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$')  #①
print (phonePattern.search('800-555-1212-1234').groups())      #②
#('800', '555', '1212', '1234')

print (phonePattern.search('800 555 1212 1234') )              #③
#None

print (phonePattern.search('800-555-1212') )                   #④
#None
```

① This regular expression is almost identical to the previous one. Just as before, you match the beginning of the string, then a remembered group of three digits, then a hyphen, then a remembered group of three digits, then a hyphen, then a remembered group of four digits. What's new is that you then match another hyphen, and a remembered group of one or more digits, then the end of the string.

② The `groups()` method now returns a tuple of four elements, since the regular expression now defines four groups to remember.

③ Unfortunately, this regular expression is not the final answer either, because it assumes that the different parts of the phone number are separated by hyphens. What if they're separated by spaces, or commas, or dots? You need a more general solution to match several different types of separators.

④ Oops! Not only does this regular expression not do everything you want, it's actually a step backwards, because now you can't parse phone numbers *without* an extension. That's not what you wanted at all; if the extension is there, you want to know what it is, but if it's not there, you still want to know what the different parts of the main number are.

The next example shows the regular expression to handle separators between the different parts of the phone number.

```
import re
phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')  #①
print (phonePattern.search('800 555 1212 1234').groups() )           #②
#('800', '555', '1212', '1234')

print (phonePattern.search('800-555-1212-1234').groups())            #③
#('800', '555', '1212', '1234')

print (phonePattern.search('80055512121234'))                        #④
#None
print (phonePattern.search('800-555-1212') )                         #⑤
#None
```

① Hang on to your hat. You're matching the beginning of the string, then a group of three digits, then `\D+`. What the heck is that? Well, `\D` matches any character except a numeric digit, and `+` means "1 or more". So `\D+` matches one or more characters that are not digits. This is what you're using instead of a literal hyphen, to try to match different separators.

② Using `\D+` instead of `-` means you can now match phone numbers where the parts are separated by spaces instead of hyphens.

③ Of course, phone numbers separated by hyphens still work too.

④ Unfortunately, this is still not the final answer, because it assumes that there is a separator at all. What if the phone number is entered without any spaces or hyphens at all?

⑤ Oops! This still hasn't fixed the problem of requiring extensions. Now you have two problems, but you can solve both of them with the same technique.

The next example shows the regular expression for handling phone numbers without separators.

```
import re
phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')   #①
print (phonePattern.search('80055512121234').groups())                #②
#('800', '555', '1212', '1234')

print (phonePattern.search('800.555.1212 x1234').groups())            #③
#('800', '555', '1212', '1234')

print (phonePattern.search('800-555-1212').groups())                  #④
#('800', '555', '1212', '')

print (phonePattern.search('(800)5551212 x1234'))                     #⑤
```

```
#None
```

① The only change you've made since that last step is changing all the `+` to `*`. Instead of `\D+` between the parts of the phone number, you now match on `\D*`. Remember that `+` means "1 or more"? Well, `*` means "zero or more". So now you should be able to parse phone numbers even when there is no separator character at all.

② Lo and behold, it actually works. Why? You matched the beginning of the string, then a remembered group of three digits (`800`), then zero non-numeric characters, then a remembered group of three digits (`555`), then zero non-numeric characters, then a remembered group of four digits (`1212`), then zero non-numeric characters, then a remembered group of an arbitrary number of digits (`1234`), then the end of the string.

③ Other variations work now too: dots instead of hyphens, and both a space and an `x` before the extension.

④ Finally, you've solved the other long-standing problem: extensions are optional again. If no extension is found, the `groups()` method still returns a tuple of four elements, but the fourth element is just an empty string.

⑤ I hate to be the bearer of bad news, but you're not finished yet. What's the problem here? There's an extra character before the area code, but the regular expression assumes that the area code is the first thing at the beginning of the string. No problem, you can use the same technique of "zero or more non-numeric characters" to skip over the leading characters before the area code.

The next example shows how to handle leading characters in phone numbers.

```
import re
phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')          #①
print (phonePattern.search('(800)5551212 ext. 1234').groups())                  #②
#('800', '555', '1212', '1234')

print (phonePattern.search('800-555-1212').groups())                            #③
#('800', '555', '1212', '')

print (phonePattern.search('work 1-(800) 555.1212 #1234'))                      #④
#None
```

① This is the same as in the previous example, except now you're matching `\D*`, zero or more non-numeric characters, before the first remembered group (the area code). Notice that you're not remembering these non-numeric characters (they're not in parentheses). If you find them, you'll just skip over them and then start remembering the area code whenever you get to it.

② You can successfully parse the phone number, even with the leading left parenthesis before the area code. (The right parenthesis after the area code is already handled; it's treated as a non-numeric separator and matched by the `\D*` after the first remembered group.)

③ Just a sanity check to make sure you haven't broken anything that used to work. Since the leading characters are entirely optional, this matches the beginning of the string, then zero non-numeric characters, then a remembered group of three digits (`800`), then one non-numeric character (the hyphen), then a remembered group of three digits (`555`), then one non-numeric character (the hyphen), then a remembered group of four digits (`1212`), then zero non-numeric characters, then a remembered group of zero digits, then the end of the string.

④ This is where regular expressions make me want to gouge my eyes out with a blunt object. Why doesn't this phone number match? Because there's a `1` before the area code, but you assumed that all the leading characters before the area code were non-numeric characters (`\D*`). Aargh.

Let's back up for a second. So far the regular expressions have all matched from the beginning of the string. But now you see that there may be an indeterminate amount of stuff at the beginning of the string that you want to ignore. Rather than trying to match it all just so you can skip over it, let's take a different approach: don't explicitly match the beginning of the string at all. This approach is shown in the next example.

```
import re
phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')         #①
print (phonePattern.search('work 1-(800) 555.1212 #1234').groups())       #②
#('800', '555', '1212', '1234')
```

```
print (phonePattern.search('800-555-1212').groups())                    #③
#('800', '555', '1212', '')


print (phonePattern.search('80055512121234').groups())                  #④
#('800', '555', '1212', '1234')
```

① Note the lack of `^` in this regular expression. You are not matching the beginning of the string anymore. There's nothing that says you need to match the entire input with your regular expression. The regular expression engine will do the hard work of figuring out where the input string starts to match, and go from there.

② Now you can successfully parse a phone number that includes leading characters and a leading digit, plus any number of any kind of separators around each part of the phone number.

③ Sanity check. This still works.

④ That still works too.

See how quickly a regular expression can get out of control? Take a quick glance at any of the previous iterations. Can you tell the difference between one and the next?

While you still understand the final answer (and it is the final answer; if you've discovered a case it doesn't handle, I don't want to know about it), let's write it out as a verbose regular expression, before you forget why you made the choices you made.

```
import re
phonePattern = re.compile(r'''
                # don't match beginning of string, number can start anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
    \D*         # optional separator is any number of non-digits
    (\d{3})     # trunk is 3 digits (e.g. '555')
    \D*         # optional separator
    (\d{4})     # rest of number is 4 digits (e.g. '1212')
    \D*         # optional separator
    (\d*)       # extension is optional and can be any number of digits
    $           # end of string
    ''', re.VERBOSE)

print (phonePattern.search('work 1-(800) 555.1212 #1234').groups())  #①
#('800', '555', '1212', '1234')
```

```
print (phonePattern.search('800-555-1212'))                    #②
#('800', '555', '1212', '')
```

① Other than being spread out over multiple lines, this is exactly the same regular expression as the last step, so it's no surprise that it parses the same inputs.

② Final sanity check. Yes, this still works. You're done.