14-Strings-and-Regular-Expressions

February 11, 2017

1 String Manipulation and Regular Expressions

One place where the Python language really shines is in the manipulation of strings. This section will cover some of Python's built-in string methods and formatting operations, before moving on to a quick guide to the extremely useful subject of *regular expressions*. Such string manipulation pattens come up often in the context of data science work, and is one big perk of Python in this context.

Strings in Python can be defined using either single or double quotations (they are functionally equivalent):

```
In [1]: x = 'a string'
        y = "a string"
        x == y

Out[1]: True
```

In addition, it is possible to define multi-line strings using a triple-quote syntax:

```
In [2]: multiline = """
        one
        two
        three
        """
```

With this, let's take a quick tour of some of Python's string manipulation tools.

1.1 Simple String Manipulation in Python

For basic manipulation of strings, Python's built-in string methods can be extremely convenient. If you have a background working in C or another low-level language, you will likely find the simplicity of Python's methods extremely refreshing. We introduced Python's string type and a few of these methods earlier; here we'll dive a bit deeper

1.1.1 Formatting strings: Adjusting case

Python makes it quite easy to adjust the case of a string. Here we'll look at the `upper()`, `lower()`, `capitalize()`, `title()`, and `swapcase()` methods, using the following messy string as an example:

```
In [3]: fox = "tHe qUICk bROWn fOx."
```

To convert the entire string into upper-case or lower-case, you can use the `upper()` or `lower()` methods respectively:

```
In [4]: fox.upper()

Out[4]: 'THE QUICK BROWN FOX.'

In [5]: fox.lower()

Out[5]: 'the quick brown fox.'
```

A common formatting need is to capitalize just the first letter of each word, or perhaps the first letter of each sentence. This can be done with the `title()` and `capitalize()` methods:

```
In [6]: fox.title()

Out[6]: 'The Quick Brown Fox.'

In [7]: fox.capitalize()

Out[7]: 'The quick brown fox.'
```

The cases can be swapped using the `swapcase()` method:

```
In [8]: fox.swapcase()

Out[8]: 'ThE QuicK BrowN FoX.'
```

1.1.2 Formatting strings: Adding and removing spaces

Another common need is to remove spaces (or other characters) from the beginning or end of the string. The basic method of removing characters is the `strip()` method, which strips whitespace from the beginning and end of the line:

```
In [9]: line = '         this is the content         '
        line.strip()

Out[9]: 'this is the content'
```

To remove just space to the right or left, use `rstrip()` or `lstrip()` respectively:

```
In [10]: line.rstrip()

Out[10]: '         this is the content'

In [11]: line.lstrip()

Out[11]: 'this is the content         '
```

To remove characters other than spaces, you can pass the desired character to the `strip()` method:

```
In [12]: num = "000000000000435"
         num.strip('0')

Out[12]: '435'
```

The opposite of this operation, adding spaces or other characters, can be accomplished using the `center()`, `ljust()`, and `rjust()` methods.

For example, we can use the `center()` method to center a given string within a given number of spaces:

```
In [13]: line = "this is the content"
         line.center(30)

Out[13]: '     this is the content      '
```

Similarly, `ljust()` and `rjust()` will left-justify or right-justify the string within spaces of a given length:

```
In [14]: line.ljust(30)

Out[14]: 'this is the content           '

In [15]: line.rjust(30)

Out[15]: '           this is the content'
```

All these methods additionally accept any character which will be used to fill the space. For example:

```
In [16]: '435'.rjust(10, '0')

Out[16]: '0000000435'
```

Because zero-filling is such a common need, Python also provides `zfill()`, which is a special method to right-pad a string with zeros:

```
In [17]: '435'.zfill(10)

Out[17]: '0000000435'
```

1.1.3 Finding and replacing substrings

If you want to find occurrences of a certain character in a string, the `find()`/`rfind()`, `index()`/`rindex()`, and `replace()` methods are the best built-in methods.

`find()` and `index()` are very similar, in that they search for the first occurrence of a character or substring within a string, and return the index of the substring:

```
In [18]: line = 'the quick brown fox jumped over a lazy dog'
         line.find('fox')
```

```
Out[18]: 16

In [19]: line.index('fox')

Out[19]: 16
```

The only difference between `find()` and `index()` is their behavior when the search string is not found; `find()` returns `-1`, while `index()` raises a `ValueError`:

```
In [20]: line.find('bear')

Out[20]: -1

In [21]: line.index('bear')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-4cbe6ee9b0eb> in <module>()
----> 1 line.index('bear')

ValueError: substring not found
```

The related `rfind()` and `rindex()` work similarly, except they search for the first occurrence from the end rather than the beginning of the string:

```
In [22]: line.rfind('a')

Out[22]: 35
```

For the special case of checking for a substring at the beginning or end of a string, Python provides the `startswith()` and `endswith()` methods:

```
In [23]: line.endswith('dog')

Out[23]: True

In [24]: line.startswith('fox')

Out[24]: False
```

To go one step further and replace a given substring with a new string, you can use the `replace()` method. Here, let's replace `'brown'` with `'red'`:

```
In [25]: line.replace('brown', 'red')

Out[25]: 'the quick red fox jumped over a lazy dog'
```

The `replace()` function returns a new string, and will replace all occurrences of the input:

```
In [26]: line.replace('o', '--')

Out[26]: 'the quick br--wn f--x jumped --ver a lazy d--g'
```

For a more flexible approach to this `replace()` functionality, see the discussion of regular expressions in Flexible Pattern Matching with Regular Expressions.

1.1.4 Splitting and partitioning strings

If you would like to find a substring *and then* split the string based on its location, the `partition()` and/or `split()` methods are what you're looking for. Both will return a sequence of substrings.

The `partition()` method returns a tuple with three elements: the substring before the first instance of the split-point, the split-point itself, and the substring after:

```
In [27]: line.partition('fox')

Out[27]: ('the quick brown ', 'fox', ' jumped over a lazy dog')
```

The `rpartition()` method is similar, but searches from the right of the string.

The `split()` method is perhaps more useful; it finds *all* instances of the split-point and returns the substrings in between. The default is to split on any whitespace, returning a list of the individual words in a string:

```
In [28]: line.split()

Out[28]: ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']
```

A related method is `splitlines()`, which splits on newline characters. Let's do this with a Haiku, popularly attributed to the 17th-century poet Matsuo Bashō:

```
In [29]: haiku = """matsushima-ya
         aah matsushima-ya
         matsushima-ya"""

         haiku.splitlines()

Out[29]: ['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']
```

Note that if you would like to undo a `split()`, you can use the `join()` method, which returns a string built from a splitpoint and an iterable:

```
In [30]: '--'.join(['1', '2', '3'])

Out[30]: '1--2--3'
```

A common pattern is to use the special character `"\n"` (newline) to join together lines that have been previously split, and recover the input:

```
In [31]: print("\n".join(['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']))

matsushima-ya
aah matsushima-ya
matsushima-ya
```

1.2 Format Strings

In the preceding methods, we have learned how to extract values from strings, and to manipulate strings themselves into desired formats. Another use of string methods is to manipulate string *representations* of values of other types. Of course, string representations can always be found using the `str()` function; for example:

```
In [32]: pi = 3.14159
         str(pi)

Out[32]: '3.14159'
```

For more complicated formats, you might be tempted to use string arithmetic as outlined in Basic Python Semantics: Operators:

```
In [33]: "The value of pi is " + str(pi)

Out[33]: 'The value of pi is 3.14159'
```

A more flexible way to do this is to use *format strings*, which are strings with special markers (noted by curly braces) into which string-formatted values will be inserted. Here is a basic example:

```
In [34]: "The value of pi is {}".format(pi)

Out[34]: 'The value of pi is 3.14159'
```

Inside the `{}` marker you can also include information on exactly *what* you would like to appear there. If you include a number, it will refer to the index of the argument to insert:

```
In [35]: """First letter: {0}. Last letter: {1}.""".format('A', 'Z')

Out[35]: 'First letter: A. Last letter: Z.'
```

If you include a string, it will refer to the key of any keyword argument:

```
In [36]: """First letter: {first}. Last letter: {last}.""".format(last='Z', first='
Out[36]: 'First letter: A. Last letter: Z.'
```

Finally, for numerical inputs, you can include format codes which control how the value is converted to a string. For example, to print a number as a floating point with three digits after the decimal point, you can use the following:

```
In [37]: "pi = {0:.3f}".format(pi)

Out[37]: 'pi = 3.142'
```

As before, here the "`0`" refers to the index of the value to be inserted. The "`:`" marks that format codes will follow. The "`.3f`" encodes the desired precision: three digits beyond the decimal point, floating-point format.

This style of format specification is very flexible, and the examples here barely scratch the surface of the formatting options available. For more information on the syntax of these format strings, see the Format Specification section of Python's online documentation.

1.3 Flexible Pattern Matching with Regular Expressions

The methods of Python's `str` type give you a powerful set of tools for formatting, splitting, and manipulating string data. But even more powerful tools are available in Python's built-in *regular expression* module. Regular expressions are a huge topic; there are there are entire books written on the topic (including Jeffrey E.F. Friedl's *Mastering Regular Expressions, 3rd Edition*), so it will be hard to do justice within just a single subsection.

My goal here is to give you an idea of the types of problems that might be addressed using regular expressions, as well as a basic idea of how to use them in Python. I'll suggest some references for learning more in Further Resources on Regular Expressions.

Fundamentally, regular expressions are a means of *flexible pattern matching* in strings. If you frequently use the command-line, you are probably familiar with this type of flexible matching with the "`*`" character, which acts as a wildcard. For example, we can list all the IPython notebooks (i.e., files with extension *.ipynb*) with "Python" in their filename by using the "`*`" wildcard to match any characters in between:

```
In [38]: !ls *Python*.ipynb
```

```
01-How-to-Run-Python-Code.ipynb 02-Basic-Python-Syntax.ipynb
```

Regular expressions generalize this "wildcard" idea to a wide range of flexible string-matching sytaxes. The Python interface to regular expressions is contained in the built-in `re` module; as a simple example, let's use it to duplicate the functionality of the string `split()` method:

```
In [39]: import re
         regex = re.compile('\s+')
         regex.split(line)
```

```
Out[39]: ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']
```

Here we've first *compiled* a regular expression, then used it to *split* a string. Just as Python's `split()` method returns a list of all substrings between whitespace, the regular expression `split()` method returns a list of all substrings between matches to the input pattern.

In this case, the input is `"\s+"`: "`\s`" is a special character that matches any whitespace (space, tab, newline, etc.), and the "`+`" is a character that indicates *one or more* of the entity preceding it. Thus, the regular expression matches any substring consisting of one or more spaces.

The `split()` method here is basically a convenience routine built upon this *pattern matching* behavior; more fundamental is the `match()` method, which will tell you whether the beginning of a string matches the pattern:

```
In [40]: for s in ["     ", "abc  ", "  abc"]:
             if regex.match(s):
                 print(repr(s), "matches")
             else:
                 print(repr(s), "does not match")
```

```
'     ' matches
'abc  ' does not match
'  abc' matches
```

Like `split()`, there are similar convenience routines to find the first match (like `str.index()` or `str.find()`) or to find and replace (like `str.replace()`). We'll again use the line from before:

```
In [41]: line = 'the quick brown fox jumped over a lazy dog'
```

With this, we can see that the `regex.search()` method operates a lot like `str.index()` or `str.find()`:

```
In [42]: line.index('fox')

Out[42]: 16

In [43]: regex = re.compile('fox')
         match = regex.search(line)
         match.start()

Out[43]: 16
```

Similarly, the `regex.sub()` method operates much like `str.replace()`:

```
In [44]: line.replace('fox', 'BEAR')

Out[44]: 'the quick brown BEAR jumped over a lazy dog'

In [45]: regex.sub('BEAR', line)

Out[45]: 'the quick brown BEAR jumped over a lazy dog'
```

With a bit of thought, other native string operations can also be cast as regular expressions.

1.3.1 A more sophisticated example

But, you might ask, why would you want to use the more complicated and verbose syntax of regular expressions rather than the more intuitive and simple string methods? The advantage is that regular expressions offer *far* more flexibility.

Here we'll consider a more complicated example: the common task of matching email addresses. I'll start by simply writing a (somewhat indecipherable) regular expression, and then walk through what is going on. Here it goes:

```
In [46]: email = re.compile('\w+@\w+\.[a-z]{3}')
```

Using this, if we're given a line from a document, we can quickly extract things that look like email addresses

```
In [47]: text = "To email Guido, try guido@python.org or the older address guido@go
         email.findall(text)

Out[47]: ['guido@python.org', 'guido@google.com']
```

(Note that these addresses are entirely made up; there are probably better ways to get in touch with Guido).

We can do further operations, like replacing these email addresses with another string, perhaps to hide addresses in the output:

```
In [48]: email.sub('--@--.--', text)

Out[48]: 'To email Guido, try --@--.-- or the older address --@--.--.'
```

Finally, note that if you really want to match *any* email address, the preceding regular expression is far too simple. For example, it only allows addresses made of alphanumeric characters that end in one of several common domain suffixes. So, for example, the period used here means that we only find part of the address:

```
In [49]: email.findall('barack.obama@whitehouse.gov')

Out[49]: ['obama@whitehouse.gov']
```

This goes to show how unforgiving regular expressions can be if you're not careful! If you search around online, you can find some suggestions for regular expressions that will match *all* valid emails, but beware: they are much more involved than the simple expression used here!

1.3.2 Basics of regular expression syntax

The syntax of regular expressions is much too large a topic for this short section. Still, a bit of familiarity can go a long way: I will walk through some of the basic constructs here, and then list some more complete resources from which you can learn more. My hope is that the following quick primer will enable you to use these resources effectively.

Simple strings are matched directly If you build a regular expression on a simple string of characters or digits, it will match that exact string:

```
In [50]: regex = re.compile('ion')
         regex.findall('Great Expectations')

Out[50]: ['ion']
```

Some characters have special meanings While simple letters or numbers are direct matches, there are a handful of characters that have special meanings within regular expressions. They are:

```
. ^ $ * + ? { } [ ] \ | ( )
```

We will discuss the meaning of some of these momentarily. In the meantime, you should know that if you'd like to match any of these characters directly, you can *escape* them with a back-slash:

```
In [51]: regex = re.compile(r'\$')
         regex.findall("the cost is $20")

Out[51]: ['$']
```

The `r` preface in `r'\$'` indicates a *raw string*; in standard Python strings, the backslash is used to indicate special characters. For example, a tab is indicated by `"\t"`:

```
In [52]: print('a\tb\tc')

a	b	c
```

Such substitutions are not made in a raw string:

```
In [53]: print(r'a\tb\tc')

a\tb\tc
```

For this reason, whenever you use backslashes in a regular expression, it is good practice to use a raw string.

Special characters can match character groups Just as the `"\"` character within regular expressions can escape special characters, turning them into normal characters, it can also be used to give normal characters special meaning. These special characters match specified groups of characters, and we've seen them before. In the email address regexp from before, we used the character `"\w"`, which is a special marker matching *any alphanumeric character*. Similarly, in the simple `split()` example, we also saw `"\s"`, a special marker indicating *any whitespace character*.

Putting these together, we can create a regular expression that will match *any two letters/digits with whitespace between them*:

```
In [54]: regex = re.compile(r'\w\s\w')
         regex.findall('the fox is 9 years old')

Out[54]: ['e f', 'x i', 's 9', 's o']
```

This example begins to hint at the power and flexibility of regular expressions.

The following table lists a few of these characters that are commonly useful:

Character	Description || Character	Description
`"\d"`	Match any digit	`"\D"`	Match any non-digit
`"\s"`	Match any whitespace	`"\S"`	Match any non-whitespace
`"\w"`	Match any alphanumeric char	`"\W"`	Match any non-alphanumeric char

This is *not* a comprehensive list or description; for more details, see Python's regular expression syntax documentation.

Square brackets match custom character groups If the built-in character groups aren't specific enough for you, you can use square brackets to specify any set of characters you're interested in. For example, the following will match any lower-case vowel:

```
In [55]: regex = re.compile('[aeiou]')
         regex.split('consequential')

Out[55]: ['c', 'ns', 'q', '', 'nt', '', 'l']
```

Similarly, you can use a dash to specify a range: for example, `"[a-z]"` will match any lowercase letter, and `"[1-3]"` will match any of `"1"`, `"2"`, or `"3"`. For instance, you may need to extract from a document specific numerical codes that consist of a capital letter followed by a digit. You could do this as follows:

```
In [56]: regex = re.compile('[A-Z][0-9]')
         regex.findall('1043879, G2, H6')

Out[56]: ['G2', 'H6']
```

Wildcards match repeated characters If you would like to match a string with, say, three alphanumeric characters in a row, it is possible to write, for example, `"\w\w\w"`. Because this is such a common need, there is a specific syntax to match repetitions – curly braces with a number:

```
In [57]: regex = re.compile(r'\w{3}')
         regex.findall('The quick brown fox')

Out[57]: ['The', 'qui', 'bro', 'fox']
```

There are also markers available to match any number of repetitions – for example, the `"+"` character will match *one or more* repetitions of what precedes it:

```
In [58]: regex = re.compile(r'\w+')
         regex.findall('The quick brown fox')

Out[58]: ['The', 'quick', 'brown', 'fox']
```

The following is a table of the repetition markers available for use in regular expressions:

Character	Description	Example
`?`	Match zero or one repetitions of preceding	`"ab?"` matches `"a"` or `"ab"`
`*`	Match zero or more repetitions of preceding	`"ab*"` matches `"a"`, `"ab"`, `"abb"`, `"abbb"`...
`+`	Match one or more repetitions of preceding	`"ab+"` matches `"ab"`, `"abb"`, `"abbb"`... but not `"a"`
`{n}`	Match `n` repetitions of preeeding	`"ab{2}"` matches `"abb"`

Character	Description	Example
`{m,n}`	Match between `m` and `n` repetitions of preceding	`"ab{2,3}"` matches `"abb"` or `"abbb"`

With these basics in mind, let's return to our email address matcher:

```
In [59]: email = re.compile(r'\w+@\w+\.[a-z]{3}')
```

We can now understand what this means: we want one or more alphanumeric character (`"\w+"`) followed by the *at sign* (`"@"`), followed by one or more alphanumeric character (`"\w+"`), followed by a period (`"\."` – note the need for a backslash escape), followed by exactly three lower-case letters.

If we want to now modify this so that the Obama email address matches, we can do so using the square-bracket notation:

```
In [60]: email2 = re.compile(r'[\w.]+@\w+\.[a-z]{3}')
         email2.findall('barack.obama@whitehouse.gov')

Out[60]: ['barack.obama@whitehouse.gov']
```

We have changed `"\w+"` to `"[\w.]+"`, so we will match any alphanumeric character *or* a period. With this more flexible expression, we can match a wider range of email addresses (though still not all – can you identify other shortcomings of this expression?).

Parentheses indicate *groups* to extract For compound regular expressions like our email matcher, we often want to extract their components rather than the full match. This can be done using parentheses to *group* the results:

```
In [61]: email3 = re.compile(r'([\w.]+)@(\w+)\.([a-z]{3})')

In [62]: text = "To email Guido, try guido@python.org or the older address guido@go
         email3.findall(text)

Out[62]: [('guido', 'python', 'org'), ('guido', 'google', 'com')]
```

As we see, this grouping actually extracts a list of the sub-components of the email address.

We can go a bit further and *name* the extracted components using the `"(?P<name> )"` syntax, in which case the groups can be extracted as a Python dictionary:

```
In [63]: email4 = re.compile(r'(?P<user>[\w.]+)@(?P<domain>\w+)\.(?P<suffix>[a-z]{3
         match = email4.match('guido@python.org')
         match.groupdict()

Out[63]: {'domain': 'python', 'suffix': 'org', 'user': 'guido'}
```

Combining these ideas (as well as some of the powerful regexp syntax that we have not covered here) allows you to flexibly and quickly extract information from strings in Python.

1.3.3 Further Resources on Regular Expressions

The above discussion is just a quick (and far from complete) treatment of this large topic. If you'd like to learn more, I recommend the following resources:

- Python's `re` package Documentation: I find that I promptly forget how to use regular expressions just about every time I use them. Now that I have the basics down, I have found this page to be an incredibly valuable resource to recall what each specific character or sequence means within a regular expression.
- Python's official regular expression HOWTO: a more narrative approach to regular expressions in Python.
- Mastering Regular Expressions (OReilly, 2006) is a 500+ page book on the subject. If you want a really complete treatment of this topic, this is the resource for you.

For some examples of string manipulation and regular expressions in action at a larger scale, see Pandas: Labeled Column-oriented Data, where we look at applying these sorts of expressions across *tables* of string data within the Pandas package.