Regular expressions

21 October 2024

Regular expressions

Regular expressions (regex) are patterns which match parts of text. Optionally, they can also replace. They are powerful ways of finding and changing strings.

The most important point is that regular expressions are not specific to Python. They're ubiquitous in programming and usually available in any piece of software that works with sequences of characters.

Data preparation is a large part of most DH projects. Regex is a key tool, whatever your software or programming language. Learn regex once: use everywhere!

Regex has limitations. We'll come back to the at the end, but regex only works with strings. In regex, the number *5* is a string, not an integer. That means that you can find sequences of numbers but with regex alone you can't increment those numbers or do other mathematical operations on them.

Fortunately a programming language can do that. So if you combine regex with something like Python you can have the best of both.

Let's read in *Persuasion* again, as we did last week. Make sure it's in the same folder as this notebook, or add a path that points to it:

[ ]:
```
with open('persuasion.txt', 'r', encoding="utf-8") as f:
    persuasion = f.read()
```

Last week we had a really awkward way of looking for the context in which 'Anne' occurs:

[ ]:
```
persuasion.find("Anne")
```

[ ]:
```
persuasion[2113:2153]
```

To get the next occurrence we'd have to look in the slice after 2133 (note that parts of a slice can be omitted).

[ ]:
```
persuasion[2153:].find("Anne")
```

But...

[ ]:
```
persuasion[4300:4340]
```

We could programmatically loop through the text and look for the string we want in each fragment. Here's an approach to splitting the text in Python, using a `while` loop. But this is getting complicated immediately.

[ ]:
```
offset = 0
while offset < 1000:
    persuasion_chunk = persuasion[offset:offset + 100]
    print(f"{offset}:{offset + 100}") # show the chunk range
    print(persuasion_chunk)
    offset += 100
    #print("finished")
```

We're only looking for literal strings so we can't ask for the context around a string like *Anne* because we don't know in advance what that context is. This is where *regular expressions*, also called *regex* can help.

First we need to import Python's `re` module. It's part of the standard library so it will be installed in any normal installation of Python.

[ ]:
```
import re
```

Now we can use regex, in which some characters are *literal* and some are *special*. Nearly every regex has a combination of both.

But let's start just by running up a regex that only looks for the literal string *Anne*. Note that, because we imported the whole `re` library, we have to refer to `re.findall`, not just `findall`.

We'll read the results into a variable called `anne_context`.

[ ]:
```
anne_context = re.findall(r"Anne", persuasion)
```

The results are now held in `anne_context`.

[ ]:
```
anne_context
```

A key special character in regex is `.` and it means *any character* so we can add this either side of our literal *Anne* string to get the context, eg 10 characters either side.

[ ]:
```
anne_context = re.findall(r".........Anne.........", persuasion)
anne_context
```

What are we getting back from Python here? Is it a string? How can we check?

Because this turns out to be a list, we can use slicing again. For example to get the last mentions of Anne in *Persuasion* we can do this:

[ ]:
```
anne_context[-5:]
```

If you're ever unsure about the syntax for lists, create a small list of your own to check your intuition.

[ ]:
```
mylist = [1, 2, 3, 4, 5, 6]
mylist[-5:]
```

With the regex, we can always add or subtract more full points to get more or less context.

But there is a problem with the results of the regex. Since this is a list, we can get its length:

[ ]:
```
len(anne_context)
```

[ ]:
```
persuasion.count("Anne")
```

These kinds of sense checks are good to build in to your thinking, and your code, as much as possible.

The next special character we'll use is `?`, meaning *one or none* of the preceding characters.

If we're not sure of the spelling of *Anne* we can now allow for *Ann* as well

[ ]:
```
anne_context = re.findall(r".........Anne?.........", persuasion)
```

We can also use `[^]` to ask for any characters other than the ones after the `^` symbol.

[ ]:
```
no_anne_context = re.findall(r".......Ann[^e]+......", persuasion)
```

[ ]:
```
no_anne_context
```

But we can also use this to make the characters around our string option. This is pretty crude but let's do it anyway:

[ ]:
```
anne_context = re.findall(r".?.?.?.?.?.?.?.?Anne.?.?.?.?.?.?.?.?.?",␣
↪persuasion)
```

[ ]:
```
len(anne_context)
```

A much better way is to give a range of how many characters we want to match, using `{` and `}`.

[ ]:
```
anne_context = re.findall(r".{0,20}Anne.{0,20}", persuasion)
```

[ ]:
```
len(anne_context)
```

We're still missing three…

Last week we weren't sure if there were characters called *Annette* or places called *Annecy* in the text. With regex we can check that. Let's look for *A* followed by any number of lower-case letters.

Square brackets represent a *character class*, meaning *any one of these in any order*. `[a-z]` is a convenience to save you from typing `[abcdefghijklmnopqrstuvwxyz]` every time.

`+` is like the `?` we saw above, but it means *one or more*.

```
[ ]: capital_a = re.findall(r"A[a-z]+", persuasion)
     capital_a
```

```
[ ]: capital_a.sort()
     capital_a
```

```
[ ]: set(capital_a)
```

So there are, apparently, characters called *Alicia*, *Archibald* and *Atkinson*.

```
[ ]: len(set(capital_a))
```

Can we use regex to look at all the verbs associated with Anne in *Persuasion*? Here's a first attempt:

```
[ ]: annes_verbs = re.findall(r"Anne [^ ]+ed\W", persuasion)
```

```
[ ]: annes_verbs
```

These aren't, of course, all of Anne's verbs. Regex only operates on sequences of characters.

We've now seen quite a lot of the regex syntax you'll ever need to find things with. To sum up:

`.` any character

`+` one or more of the preceding (by default, matches as much as possible: 'greedy')

`?` one or none of the preceding

`*` one or none of the preceding (by default, matches as much as possible: 'greedy')

`[]` a character class, 'find any of these, in any order'

`[^]` a negated character class 'find anything that is not one of these'

What about if you want to find literal versions of the above, like a literal full stop?

Put a `\` in front of it to *escape* it: make it not special. For example `\?` matches a literal question mark.

some shortcuts `\w` any non-whitespace character

`\W` any whitespace character, including punctuation

`[0-9]` any number

`[a-z]` any lowercase letter

`[A-Z]` any uppercase letter

But if you're new to regex this will still be a lot to take in. Practice is the only way to learning regex, so don't worry. The key thing is to remember that there are many situations where regex will make your life easier and you can look up the syntax any time you need to.

Last week, splitting on whitespace was too crude for us to get all the words from *Persuasion*. Regex allows us to fix that.

```
[ ]: persuasion_words = re.findall(r'\w+', persuasion)
```

```
[ ]: biggest_words = sorted(persuasion_words, key=len, reverse=True)
```

```
[ ]: biggest_words[:10]
```

```
[ ]: from collections import Counter
```

```
[ ]: mycounts = Counter(persuasion_words)
     mycounts.most_common(10) # or whatever number required
```

What about replacing? For that, in Python, we use `re.sub`. It works the same way as `findall` but we need an extra argument to the function: the thing we want to put in place of what we found. As always, the simplest possible example is a good place to start.

```
[ ]: sample = "Anne Elliot"
     print(sample)
     sample = re.sub(r"Anne? El+iot+", "the principal character", sample)
     print(sample)
```

The most powerful part of replacement is re-using parts of the find, for example to add to them or move them around.

To do this, put round brackets around a part of the regex you want to recall in the replacement, this is known as a *capture group*.

In the replacement text the contents of the first set of brackets are referred to with `\\1`, the second set as `\\2` and so on. In regex this is known as a *back reference*.

```
[ ]: sample = "Anne Elliot"
     print(sample)
     print("But let's swap the names around:")
     sample = re.sub(r"(Anne?) (El+iot+)", "\\2, \\1", sample)
     print(sample)
```

When not to use regular expressions.

Because regex work on strings, they cannot reliably *parse* data, that is work with its structure.

Once you get good at regex, you might be tempted to use it to parse structured data. Here's a simple example of data in *CSV* (*comma-separated values*) format:

```
character,novel,occurence_count
Anne Elliot,Persuasion,486
Emma Woodhouse,Emma,397
Elizabeth Bennett,Pride and Prejudice,292
Fanny Price,Mansfield Park,331
```

If you try to extract the middle column you will be trying to parse the data with regex. This is highly unreliable and inadvisable.

Group work

finding

1. Find the context around another main character in *Persuasion*, Captain Wentworth.
2. Does Captain ever get abbreviated to *Capt.*?
3. Find the word following *Anne*. Can you make a unique list of these? Can you take account of punctuation between *Anne* and the following word?
4. Can you create an alphabetised list of all 9-letter words in *Persuasion*?
5. By default in Python, a `.` won't run past a `\n` character. Can you modify one of the above searches to include characters from the next line? You might need to look at the `re` (https://docs.python.org/3/library/re.html)[documentation] for the answer to this.

replacing Use `re.sub` to replace some text in `persuasion`. If you work on the whole novel, Python will have no trouble with this, but it might be hard to see the results. You might prefer to create a slice of Persuasion of a few hundred characters, so you can see the output of your replacement more easily.

This will overwrite the text of `persuasion`, so if you prefer you can create a string with a different variable name, eg:

```
modified_persuasion = re.sub(r"search string", "replacement", persuasion)
```

finally Can you explain why, above, we got slightly more results for `persuasion.count("Anne")`, when compared to `re.findall(r"{0,20}Anne{0,20}, persuasion)`? This is a bit tricky! Maybe create a small text of your own to test the way these two behave.