Extracting Data From Texts

Module 2 | Chapter 1 | Notebook 4

As a data scientist, you can't always choose how the data is presented. An important part of your work is to extract, merge and structure data from different sources. Texts in particular are often poorly structured, making it difficult to extract information from them. We'll look at this task now. You will learn how to:

- Read PDF files
- Extract data from text using *regular expressions*

Reading PDF documents

In the last lessons we downloaded the HTML code from web pages. We used an HTML parser to help us extract the information we needed from this code and then put it all together in a table. However, some cells contain a lot of text, and we don't need all of this. It can be very laborious to fish out the data you need from the texts using the string methods you've learned previously. This is where *regular expressions*, *regex* or *regexes* for short, come in handy. These are patterns that we are looking for in the text. For example, imagine that you want to extract an email address from a text. You don't know exactly what it is, how long it is and where it is in the text. But you know it's made up of three components: A section of text before the @ symbol, a section after the symbol, and a country abbreviation. You can express these requirements for the text you're looking for as a *regex*. This makes it possible to define general text searches. This increases the reusability of your code and makes it much shorter.

Before we move on to the investor's data in the next lesson, let's look at a sample exercise to learn more about regular expressions. This is an invoice, which we have in PDF format. Working with PDFs is not much fun, but unfortunately we can't always choose how our data is formatted. Fortunately, there's a Python module for almost every file format, which allows us to read the files. In this case it's `PyPDF2`. Import the module without an alias.

Important: `PyPDF2` is **not** part of the Anaconda distribution. Normally you'd have to install `PyPDF2` first before you can import it. But we've already installed `PyPDF2` in the Data Lab for you, so you can import it straight away.

In [1]:
```
import PyPDF2
```

Now take a look at the file. It's called *invoice.pdf* and is located in the home directory. You can find and open the file with Data Lab's *file browser*. As you can see, this is an invoice issued by *BestBooks*. Imagine that *BestBooks* has a lot of these invoices in PDF format. They would like to

automatically read the most important information, such as customer number, invoice number, invoice date, etc. and transfer it to a table. This data is similar on every invoice. For example, the date always has the format DD.MM.YYYY. However, the details are different each time. HTML texts make it easier for us to search through them by using the tags. However, this doesn't work for all text formats. That's why there are *regular expressions* that we can apply to all kinds of text.

But first we have to read the file with `PyPDF2`. Unlike `pandas`, `PyPDF2` doesn't just require the file path but also the opened document. We'll do this with the Python function `open()`. You have to specify the file path as well as whether to use read or write mode. Here are the most important modes:

Argument	Meaning	Use
'r' (default)	read	When you want to read the file without changing it
'w'	write	When you want to rewrite/overwrite the file
'a'	add	When you want to add new characters at the end of the file
'rb', 'wb', 'ab'	binary format	If the file doesn't use simple text encoding

Since the PDF format doesn't use typical character encoding (such as UTF-8), we need to use the `'rb'` mode, i.e. reading without interpreting the encoding. Open the file *invoice.pdf* and store the result in the variable `file_reader`.

In [2]:
```
file_reader = open('invoice.pdf', 'rb')
```

Now you can pass `file_reader` to `PdfFileReader` from `PyPDF2`. This will return an object that has some functions to help us work with the PDF file. Store this object under the name `pdf_reader`.

In [5]:
```
pdf_reader = PyPDF2.PdfFileReader(file_reader)
```

`PyPDF2` can only read the pages of the PDF individually. You can output the number of pages with the `my_pdf_reader.getNumPages()` method. Try it out.

In [6]:
```
pdf_reader.getNumPages()
```

Out[6]:
```
1
```

As you already saw when you opened it manually, there's only one page. You can get this page by using the command `my_pdf_reader.getPage(0)`. The `0` stands for the page's index. The first page is located at index 0. Select the first page and store the resulting *page* object with the variable name `pdf_page`.

In [7]:
```
pdf_page = pdf_reader.getPage(0)
```

Now we can finally extract the text. The *page* object has the method `my_pdf_page.extractText()` to help us with this. The text is read from the document and

returned as *string*. We can then use this *string* to work with *regexes*. Store the text as `pdf_str` and print the first 100 characters.

In [8]:
```
pdf_str = pdf_page.extractText()
pdf_str[:100]
```

Out[8]:
```
' \n \n \nBestBooks \n\n \nBücherstraße 22 \n\n \n12345 Berlin\n \nWissbegierig GmbH
\n \nLange\n-\nAllee 77\n \n80331 '
```

As you can see, we now have a *string*. Unfortunately it now contains a lot of line breaks. We'll take care of that in a second. First we should close the file again. Use the `my_file_reader.close()` method, otherwise the file is still considered to be open. This can lead to various problems, such as not being able to delete the file or changes not being carried over. So it's good form to close a file again after you've finished using it. Use the following code cell to do this.

In [9]:
```
file_reader.close()
```

Congratulations: You've read a PDF document with Python for the first time! However, the whole page is now in a single `str`. If we were to use the string methods you've learned previously, we'd very quickly reach their limitations. So now let's take a look at what we can do with *regular expressions*.

Regular expressions

Now we'll focus on the text. Since the line breaks don't follow a clearly recognizable pattern, it's best just to remove them. Otherwise, they'll mess up our search patterns. So replace all the line breaks with an empty `str`. Remember to reassign the resulting `str` back to `pdf_str`. Then print `pdf_str` to check if the line breaks are gone.

In [10]:
```
pdf_str = pdf_str.replace('\n','')
pdf_str
```

Out[10]:
```
'   BestBooks  Bücherstraße 22  12345 Berlin Wissbegierig GmbH Lange-Allee 77 80331 M
ünchen  BestBook Bücherstraße 22 12345 Berlin  Tel: 030 1234567 Email: info@bestbook
s.de  Invoice Invoice No. 2017-03-1103  Customer-ID.: 1003  Date: 12.03.2017 Please i
nclude the above in the payment reference!  QUANTITY DESCRIPTION ITEM PRICE ZEILENSUM
ME   1 Data Science for beginners 19,99 EUR 19,99 EUR 2 Joy in Learning 5,95 EUR 5,95
EUR 3 Data Analysis for Pros 17,99 EUR 17,99 EUR 4 Data! 8,99 EUR 8,99 EUR  Subtotal
52,92 EUR VAT 3,70 EUR Total 56,62 EUR          BestBook Inh: B. Wurm Bücherstraße 22
12345 Berlin Volksbank Berlin IBAN: DE12 1234 5678 0000 5646 10 Steuer-Nr.: 3322342 F
inanzamt Berlin  '
```

Now `pdf_str` looks a bit better. But the text has no real structure in this form. So reading certain details, such as the invoice number, isn't so easy. It would be very helpful to be able to search through the `str` for a pattern rather than a fixed string of characters. This is exactly what the `re` module allows us to do. You can read the documentation here.

The `re.findall()` function is particularly useful. You pass a *regular expression* (i.e. a search pattern) to the function, as well as the text you want to search through. The result is a list of

strings that match the search pattern. The simplest *regular expressions* consist of fixed character strings. Run the following cell to import `re` and look for the word `'Customer-ID'`.

In [12]:
```
import re

expression = 'Customer-ID'  # define regular expression
re.findall(expression, pdf_str)  # look for regular expression in pdf_str
```

Out[12]:
```
['Customer-ID']
```

Now we just see the word `'Customer-ID'` once. This means it only appears once in `pdf_str`.

But the strength of *regular expressions* is that they can generalize characters to actually search for common patterns. Run the following code cell to get an idea of this.

In [21]:
```
expression = r'Customer-ID\.:\s\d+'  # define regular expression
re.findall(expression ,pdf_str)  # look for regular expression in pdf_str
```

Out[21]:
```
['Customer-ID.: 1003']
```

What happened? We also got the number that comes after the customer number - and just by adding a few characters. This might seem a bit cryptic at first. Let's go through this step by step. We used the following character string for the *regex*: `r'Customer-ID\.:\s\d+'`.

You might be wondering what the `r` right in front of the `str` is there for. This tells Python that the following *string* is a *raw string*. This means that Python should use the characters as they are without interpreting them itself. For example, Python interprets the characters `'\n'` as a line break. If we want the characters just as they are, we need to add one more backslash `'\'`, so `'\\n'`, or we use a *raw string* `r'\n'` to avoid Python interpreting them. `re` also interprets characters from a *regular expression*, but follows different conventions. We use *raw strings* to avoid clashes in how things are interpreted. These clashes are rare, but that makes it more annoying to search for them.

The `r` for *raw string* is followed by `'Customer-ID'`. We already had this string of characters. It isn't interpreted by `re` and is searched for exactly as it is. Next up is `'\.'`. Here we tell `re` that it should be followed by a period. We need the backslash `'\'`, because otherwise `re` would interpret `'.'` as any character. In *regular expressions*, `'\'` is the *escape* character: it prevents the following character from being interpreted (just like how `'\\n'` in Python strings prevents characters from being interpreted as line breaks).

This is followed by a colon `':'`. Just like the letters and the hyphen `'-'`, this is matched just as it is. Then things get interesting again with `'\s'`: the backslash `'\'` here indicates that the next character should be interpreted differently (just like `'\n'` in Python strings). `'\s'` then no longer stands for the letter `'s'`, but for a whitespace character. Whitespaces are spaces, line breaks or tabs. Our result contains a space here. Then the search pattern continues with '\d'. The d here stands for *digit*, so you are indicating that there should be any digit in this position.

So far we have have specified all the characters precisely, one after the other and then we generalized them a bit with `'\s'` and `'\d'`. But *regular expressions* can also contain repetitions. That's what the `'+'` in our search pattern is for. It specifies that the preceding character (a digit `'\d'`) should occur at least once. In this case it means that every digit that follows becomes part of the result until a different character appears.

Now use a *regular expression* and `re.findall()` to find the invoice date. Remember that it follows the format DD.MM.YYYY. So there are 8 digits separated by dots in certain positions. Store the result as `invoice_date` and print it.

In [44]:
```
re.findall(r'\d\d\.\d\d\.\d\d\d\d',pdf_str)
invoice_date = re.findall(r'\d+\.\d+\.\d+',pdf_str)
```

`pdf_str` only contains the date `'12.03.2017'`. There are several *regular expressions* that we could use for this. In very few cases there is only one correct solution. In this case, `r'\d\d\.\d\d.\d\d\d\d'` would be the variant with the fewest different characters. But you could also use `r'\d\d\.\d\d.\d+` as an expression, for example.

The interpretation of *regular expressions* is not specific to Python. You can imagine it as a language in its own right. The following tables contain the most important codes:

Generalizations

Expression	Meaning
.	matches any character (except line break)
\d	digit
\w	letter, digit or underscore
\s	whitespace: space, tab, line break
\D, \W, \S	negation of \d, \w und \s

Repetitions

Expression	Meaning
{min, max}	Repetition of the preceding expression at least min-times and no more than max-times
{3}	Repeat the preceding expression exactly 3 times
+	Repeat the preceding expression at least once
*	Repeat the preceding expression at least 0 times

Combinations

Expression	Meaning
...|...	"or" operator: Matches the expression to the left or right of |

Expression	Meaning
[...]	"or" operator: Matches an expression within the brackets
[a-z]	"from-to" operator: Matches a lower case letter between a and z
[A-Z0-9]	from-to-or operator: A capital letter between A and Z or a digit between 0 and 9 is searched for
[^...]	except: Only matches expressions that are not listed in the brackets
(...)	*capture group*: All expressions within the parentheses are searched for together and form a separate result
?	Optional: The preceding expression may apply exactly zero or one times
\	*Escape*: The following character is searched for as it is (e.g. \. searches for a period)

Boundaries

Expression	Meaning
^	Matches the beginning of the string
\$	Matches the end of the string
\b	Matches the beginning or the end of a word

That's quite a lot of characters. You don't have to remember them all right now, as long as you know where to look if you need to. *Regular expressions* can be tricky, so let's practice a little more before we return to the investor's data in the next lesson.

Now find the invoice number and save it as `invoice_no`. The whole string of characters is `'Invoice No. 2017-03-1103'`.

In [42]:
```
expression = r'Invoice No\.\s\d+\-\d+\-\d+'  # define regular expression
invoice_no = re.findall(expression, pdf_str)
```

What's the total invoice amount? Save it as a floating point number named `invoice_price`. The relevant part of `pdf_str` is `'Total 56,62 EUR'`. Note that BestBooks is based in Germany, and although the invoice is mostly in English, it still uses the continental European convention of using a comma as the decimal separator. You'll need to replace this with a full stop instead.

In [33]:
```
invoice_price = re.findall(r'Total .* EUR', pdf_str)[0]
invoice_price = invoice_price.split(' ')[1].replace(',', '.')  # split the price and r
invoice_price = float(invoice_price)
print(invoice_price)
```

```
56.62
```

Does the invoice contain an email address? If so, then store it as `invoice_mail`.

In [34]:
```
expression = r'\w+@\w+\.\w{2}'
invoice_mail = re.findall(expression, pdf_str)[0]
invoice_mail
```

Out[34]:
```
'info@bestbooks.de'
```

We only get the *BestBooks* email address: info@bestbooks.de.

Now things will get a little trickier. Search for all the combinations of postcode and city and print them.

Tip: German postcodes are always made up of 5 digits and come directly **before** the city name. You can use curly brackets to indicate this: `'{5}'` means that the preceding character should occur exactly 5 times. Make sure you don't accidentally just take the last 5 digits of a longer number. You can make sure of that with `'\b'`. This is always used to mark the boundary between letters or numbers and *whitespaces* or other characters that end a word. You might have noticed that the Customer address in this invoice uses the German city name München (Munich), which contains the special character ü. It's possible that the city names here could also contain other special characters, such as Köln (Cologne). To look for letters also including the possibility of these special characters, we recommend indicating which characters **not** to look for, rather than trying to specify a list of all the potential characters that could occur. You can exclude certain characters with `'[^...]'`. Replace `'...'` with characters that should not occur. For example, with `'[^\d\s]'` you can search for characters that are neither numbers nor *whitespace*.

In [35]:
```
expression = r'\b\d{5}\s[^\d\s]+'
re.findall(expression, pdf_str)
```

Out[35]:
```
['12345 Berlin', '80331 München', '12345 Berlin', '12345 Berlin']
```

We've found the combinations `['12345 Berlin', '80331 München', '12345 Berlin', '12345 Berlin']`.

The postcode and the city are always together in one list element. But we can also separate them straight away. You can use what we call *capture groups* or *capturing groups* to do this. You do this by placing round brackets around your search patterns. The following cell uses one *capture group* for the postcode and another one for the city. They are separated by a *whitespace* character. The entire expression is therefore searched for together, but the result is split. Run the following code cell to try it out:

In [36]:
```
expression = r'(\b\d{5})\s([^\d\s]+)'
re.findall(expression, pdf_str)
```

Out[36]:
```
[('12345', 'Berlin'),
 ('80331', 'München'),
 ('12345', 'Berlin'),
 ('12345', 'Berlin')]
```

This allows you to quickly store the two pieces of information separately. The Berlin address belongs to *BestBooks*. Save the customer's postcode as `invoice_post_code` and the city as `invoice_city`.

In [39]:
```
expression = r'(\b\d{5})\s([^\d\s]+)'  # define regex with capture groups
post_and_city = re.findall(expression, pdf_str)[1]  # select the element containing Mü
invoice_post_code = post_and_city[0]
invoice_city = post_and_city[1]
invoice_city
```

Out[39]:
```
'München'
```

Now we've extracted the most interesting data from the invoice, we can summarize it all in a *dictionary*. Run the following code cell to do this.

In [45]:
```
invoice_info = {'number': invoice_no, 'date': invoice_date , 'price': invoice_price ,'
invoice_info
```

Out[45]:
```
{'number': ['Invoice No. 2017-03-1103'],
 'date': ['12.03.2017'],
 'price': 56.62,
 'city': 'München',
 'post_code': '80331'}
```

We could then turn this into a `DataFrame`, giving the data from a PDF document a structured format.

Congratulations: You've used two new modules to extract information from text. *Regular expressions* are a powerful tool which allow you to sift out data from unstructured texts. This even works in DataFrames, as you'll see in the next lesson.

Remember:

- Get text from a PDF in the following steps:
  1. Open the document: `my_file_reader = open('my_file_path', 'my_mode')`
  2. Open it with `PyPDF2`: `my_pdf_reader = PyPDF2.PdfFileReader(my_file_reader)`
  3. Select page: `my_pdf_page = my_pdf_reader.getPage(my_page_number)`
  4. Extract the text: `my_pdf_page.extractText()`
- *Regular expressions* are search patterns for texts
- Extract search patterns with `re.findall(my_expression, my_str)`

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
