BCH2203 Python - 5. RegEx and DataFrames

Ramses van Zon

9 February 2022

Adding to our data analysis toolkit... Pattern Matching

Pattern Matching

- In applications, we may want to matched up DNA sequences, codons, proteins code, etc.
- We could use `somestring.find` etc., but those look for exact matches, which is rarely what occurs in nature.
- For instance, genes often contain more than the codons that get translated to proteins. The initial product of DNA transcription is subject to a process called splicing which removes part of the RNA called the introns, leaving the extrons to form the mRNA.

  So if we want to search for a gene that codes for a specific protein, we should really allow for the codon sequence to be interrupted by sets of non-coding triplets.
- We could write intricate loops to allow for this, but loops, particularly in Python, are very slow.

Pattern Matching Techniques

We'll look at two alternatives:

- Regular Expressions

  Also called regex, is a language to search, extract and manipulate string patterns in a larger text (here: sequence).

- Alignment

  A technique that finds imperfect matches and orders them according to a score, so more likely true matches have a higher score.

  (E.g. BLAST etc.)

We need a way to encode patterns, and that's what Regular Expressions do.

With a RegEx we can express things like:

- Match characters from a specific set.
- A character is optional.
- Specific numbers of repetitions of characters from a set.
- Disallow certain characters
- etc.

We can use them to find patterns in a string and also to transform those matches.

In Python, you can use the package `re` (or the `regex` package).

Characters

To match a specific character, use that character.

```
>>> import re
>>> sentence="The quick brown fox jumps over the 1azy dog."
>>> re.search(r"u", sentence)
<re.Match object; span=(5, 6), match='u'>
>>> for match in re.finditer(r"T", sentence): print(match)
<re.Match object; span=(5, 6), match='u'>
<re.Match object; span=(21, 22), match='u'>
```

A slash and a character form special patterns, e.g

Character	Meaning
.	Any character
\d	One digit
\w	One word character
\s	One whitespace character
\s	One whitespace character
\b	A word boundary
\n	A new line

```
>>> re.search(r"\d\w\w\w", sentence)
<re.Match object; span=(35, 39), match='1azy'>
```

About Raw Strings

- The slash syntax inside strings is already used by Python to indicate special characters.
- E.g. `"\t"` is a TAB character, `"\\"` a single slash.
- To avoid having to put double slashes all over the place, we use Python's **raw strings**
- Raw strings do not get slashed characters replaced by special characters.
- The `r` in front of the quotes of a string makes it raw.

More RegEx Syntax

Quantifiers

Metacharacter	Meaning
+	One or more
*	Zero or more
?	Zero or one
{2}	Exactly two times
{2,4}	Two to four times
*? +? {2,4}?	Non-greedy versions

Put these after a character or group.

```
>>> re.search(r"\b\w{4}\b", sentence)
<re.Match object; span=(26, 30), match='over'>
>>> re.search(r"\b\w{4}\b", sentence).group()
over
```

Non-metacharacters

To get a character without meaning, you have to escape it with a slash. E.g. `r"\."` matches a period.

Select one of a set

Use a set of characters between square brackets `[]`

Use `[^...]` to match anything not in the set.

```
>>> re.search(r"[cg]", sentence)
<re.Match object; span=(7, 8), match='c'>
>>> re.search(r"[xof]{3}", sentence)
<re.Match object; span=(16, 19), match='fox'>
```

Case insensitive searches?

Sure, prepend the pattern with `(?i)`:

```
>>> re.search(r"(?i)t", sentence)
<re.Match object; span=(0, 1), match='T'>
```

Groups

Put a pattern between `()` to create a group.

```
>>> re.search(r"(\w{5}\s)+", sentence)
<re.Match object; span=(4, 16), match='quick brown '>
```

Ways to Use Regexes

`re.search`

`re.search` returns the first match anywhere in the string.

(in fact, it may stop at a newline).

It returns a `re.Match` object which contains the span (where it was found) and the matching string.

`re.match`

This is exactly like `re.search` except the match must occur from the start of the string.

It also returns a `re.Match` object.

`re.findall`

Returns a list of all matches in the string.

Does not return spans.

`re.finditer`

Returns an iterator to go over all matches in the string. Each match returns a re.Match object and thus contains the span.

`re.replace`

Replace the match with somthing else. Returns a new string.

Adding to our data analysis toolkit... Dataframes

Key-val1-val2-... stores

- In Python, we can label values using `dicts` as key-value stores.
- What if several values are to be associated with one key?
- With the different values of different types.
- E.g.: A set of buildings of which we want to store the color, when they were built, their location, etc.

Key	Color	Built	Where
House	White	Dec 1995	...
Office	Grey	Jan 2001	...
Shed	Green	May 2008	...

- Essentially, this forms a table.
- Such tables are the bread-and-butter of relational databases.
- No built-in type for these in Python.

Key-val1-val2-... stores → Pandas

How to deal with this type of data?

- Different types of values: cannot use numpy arrays.
- Storing key-val1-val2-val3 as separate dictionaries ([key,val1], [key,val2], ...) is wasteful and inefficient.
- Could store each column as a numpy array, but awkward.
- We'd like to be able to label columns as well as rows.
- All of this is possible using DataFrame data type from the pandas package.

Key	Color	Built	Where
House	White	Dec 1995	...
Office	Grey	Jan 2001	...
Shed	Green	May 2008	...

Pandas

DataFrames

- The Python package `pandas` provides DataFrames, which are somewhat like Excel spreadsheets, but more versatile.
- Has excellent support for many input and output formats.
- Fairly easy-to-use API
- Thoroughly tested.
- Aimed at high performance (uses numpy)
- http://pandas.pydata.org

Working with pandas

```
>>> import pandas as pd
>>> name = ['Anna','William','Emma','John','James','Mary']
>>> gender = ['F', 'M', 'F', 'M', 'M', 'F']
>>> number = [2604, 9532, 2003, 9655, 5927, 7065]
>>> data = list(zip(name, gender, number))
>>> data
[('Anna', 'F', 2604),
 ('William', 'M', 9532),
 ('Emma', 'F', 2003),
 ('John', 'M', 9655),
 ('James', 'M', 5927),
 ('Mary', 'F', 7065)]
```

The 'zip' function returns a tuple iterator for constructing the table.

These are the top 3 American girl and boy names in the year 1880.

Working with pandas, continued

```
>>> df=pd.DataFrame(data,columns=['Name','Gender','Number'])
>>> df
```

	Name	Gender	Number
0	Anna	F	2604
1	William	M	9532
2	Emma	F	2003
3	John	M	9655
4	James	M	5927
5	Mary	F	7065

- The data have been cast into a 'DataFrame' type.
- Note how there are now index numbers and column headings.

```
>>> df.to_csv('births1880.csv', index=False, header=True)
```

Pandas and Microsoft Excel Sheets

Writing

```
>>> print(df)
      Name Gender  Number
0     Anna      F    2604
1  William      M    9532
2     Emma      F    2003
3     John      M    9655
4    James      M    5927
5     Mary      F    7065
>>> df.to_excel('births1880.xlsx', index=False, header=True)
```

Reading

```
>>> xl = pd.ExcelFile('births1880.xlsx')
>>> xl.sheet_names
['Sheet1']
>>> newdf = xl.parse('Sheet1')
```

About Tabular file formats

Despite their versatility and open-source alternatives, using a proprietary format like an Excel file ties you to a non-free vendor-specific solution.

1. Comma separated values:

   `pandas.read_csv`, `pandas.DataFrame.to_csv`

2. JSON:

   `pandas.read_json`
   `pandas.DataFrame.to_json`

3. HTML tables:

   `pandas.read_html`
   `pandas.DataFrame.to_html`

Note:

- For (mostly) numeric data, you should use **binary formats**.
- To go in the other direction and store your tables in a relational database, such as `sqlite`, is possible too, but out of scope here.

Pandas: DataFrame functions

Some DataFrame functions

```
>>> df['Name'].describe()
count   6
unique  6
top     Anna
freq    1
Name:Name,dtype:object

>>> df.Number.describe()
count       6.000000
mean     6131.000000
std      3297.861792
min      2003.000000
25%      3434.750000
50%      6496.000000
75%      8915.250000
max      9655.000000
Name:Number,dtype:float64
```

```
>>> s = df.sort_values('Number')
>>> type(s)
pandas.core.frame.DataFrame
>>> print(s.head(2))
    Name  Gender  Number
2   Emma       F    2003
0   Anna       F    2604
>>> df['Name'].count()
6
>>> df.shape
(6, 3)
>>> df.Number.sum()
36786
>>> print(df.dtypes)
Name      object
Gender    object
Number    int64
dtype:    object
```

Use “help(df)” to look at all the available functions.

Adding columns

```
>>> d = [0, 1, np.nan, 2]
>>> df = pd.DataFrame(d)
>>> print(df)
     0
0   0.0
1   1.0
2   NaN
3   2.0
>>> df.columns = ['Rev']
>>> print(df)
    Rev
0   0.0
1   1.0
2   NaN
3   2.0
>>> df[1:2]
    Rev
1   1.0
```

```
>>> df['NewCol'] = 5
>>> print(df)
     Rev  NewCol
  0   0.0    5
  1   1.0    5
  2   NaN    5
  3   2.0    5
>>> df['test'] = df['Rev']+1
>>> print(df)
     Rev  NewCol  test
  0   0.0      5   1.0
  1   1.0      5   2.0
  2   NaN      5   NaN
  3   2.0      5   3.0
>>> print(df['test'])
  0   1.0
  1   2.0
  2   NaN
  3   3.0
Name: test, dtype: float64
```

Adding rows

```
>>> df.xs(3)
Rev      2.0
NewCol   5.0
test     3.0
Name: 3, dtype: float64

>>> df2=df.append(df.xs(3), ignore_index=True)
>>> print(df2)
   Rev  NewCol   test
0  0.0       5    1.0
1  1.0       5    2.0
2  NaN       5    NaN
3  2.0       5    3.0
4  2.0       5    3.0
```

You may only append structures which are of type DataFrame or Series.

```
>>> df2.append([(-1, -2, -3)])
     0   1   2 NewCol Rev test
0  NaN NaN NaN      5   0    1
1  NaN NaN NaN      5   1    2
2  NaN NaN NaN      5 NaN  NaN
3  NaN NaN NaN      5   2    3
4  NaN NaN NaN      5   2    3
0   -1  -2  -3    NaN NaN  NaN

>>> df2.append(pd.DataFrame(
...             [(-1, -2, -3)],
...             columns=['Rev','NewCol','test']))
    Rev  NewCol  test
0     0       5     1
1     1       5     2
2   NaN       5   NaN
3     2       5     3
4     2       5     3
0    -1      -2    -3
```

Real data example

Let's play with real data!

It's more fun to play with real data.

- The file 311-service-requests.csv.zip is available on the course website.
- Uncompress the file and put it somewhere easy to access.
- This is service request data from New York City (NYC Open Data)

  (Btw, Toronto now has open data too).

Let's play with real data!

```
>>> filename = "/path/to/my/311-service-requests.csv"
>>> data = pd.read_csv(filename,low_memory=False)
>>>
>>> data.shape
(111069, 52)
>>>
>>> data.columns
Index(['Unique Key', 'Created Date', 'Closed Date', 'Agency','Agency Name', 'Complaint Type', 'Descriptor', 'Location
Type', 'Incident Zip', 'Incident Address', 'Street Name','Cross Street 1', 'Cross Street 2', 'Intersection Street 1',
'Intersection Street 2', 'Address Type', 'City', 'Landmark','Facility Type', 'Status', 'Due Date', 'Resolution Action
Updated Date', 'Community Board', 'Borough', 'X Coordinate(State Plane)', 'Y Coordinate (State Plane)', 'Park Facility
Name', 'Park Borough', 'School Name', 'School Number','School Region', 'School Code', 'School Phone Number',
'School Address', 'School City', 'School State', 'School Zip', 'School Not Found', 'School or Citywide Complaint',
'Vehicle Type', 'Taxi Company Borough', 'Taxi Pick Up Location', 'Bridge Highway Name', 'Bridge Highway Direction',
'Road Ramp', 'Bridge Highway Segment', 'Garage Lot Name', 'Ferry Direction', 'Ferry Terminal Name', 'Latitude',
'Longitude', 'Location'], dtype='object')
```

Real data, continued

```
>>> data.values[0]
array([26589651, '10/31/2013 02:08:41 AM', nan, 'NYPD', 'New York City Police Department', 'Noise - Street/Sidewalk',
  'Loud Talking', 'Street/Sidewalk', 11432.0, '90-03 169 STREET', '169 STREET', '90 AVENUE', '91 AVENUE', nan, nan,
  'ADDRESS', 'JAMAICA', nan, 'Precinct', 'Assigned', '10/31/2013 10:08:41 AM', '10/31/2013 02:35:17 AM', '12 QUEENS',
  'QUEENS', 1042027.0, 197389.0, 'Unspecified', 'QUEENS', 'Unspecified', 'Unspecified', 'Unspecified', 'Unspecified',
  'Unspecified', 'Unspecified', 'Unspecified', 'Unspecified', 'Unspecified', 'N', nan, nan, nan, nan, nan, nan, nan,
  nan, nan, nan, nan, 40.70827532593202, -73.79160395779721, '(40.70827532593202, -73.79160395779721)'], dtype=object)
```

Specifying the index gives us all the values in that row.

Real data, continued more

```
>>> data[0:3]
   Unique Key          Created Date            Closed Date Agency
0    26589651 10/31/2013 02:08:41 AM                    NaN   NYPD
1    26593698 10/31/2013 02:01:04 AM                    NaN   NYPD
2    26594139 10/31/2013 02:00:24 AM 10/31/2013 02:40:32 AM   NYPD

                         Agency Name           Complaint Type
0   New York City Police Department  Noise - Street/Sidewalk
1   New York City Police Department          Illegal Parking
2   New York City Police Department       Noise - Commercial


>>> np.unique(data["Complaint Type"].values)
array(['APPLIANCE', 'Adopt-A-Basket', 'Agency Issues', 'Air Quality', 'Animal Abuse', 'Animal Facility - No Permit',
       'Animal in a Park', 'Asbestos', 'BEST/Site Safety', 'Beach/Pool/Sauna Complaint', 'Benefit Card Replacement',
       'Bike Rack Condition', 'Bike/Roller/Skate Chronic', 'Blocked Driveway', 'Boilers','Bridge Condition', 'Broken
       Muni Meter', 'Building/Use', 'Bus Stop Shelter Placement', 'CONSTRUCTION', 'City Vehicle Placard Complaint',
       ...], dtype=object)
```

In Jupyter Notebook, `data[0:3]` instead of `print(data[0:3])` gives a scrollable table.

Real data, continued some more

Suppose we only want some of the columns?

```
>>> data.loc[12:18, ["Created Date","Complaint Type","Longtitude"]]
              Created Date            Complaint Type   Longitude
12   10/31/2013 01:20:57 AM           Illegal Parking  -73.952259
13   10/31/2013 01:20:13 AM           Noise - Vehicle  -73.836457
14   10/31/2013 01:19:54 AM                    Rodent  -73.999218
15   10/31/2013 01:14:02 AM  Noise - House of Worship  -73.970370
16   10/31/2013 12:54:03 AM   Noise - Street/Sidewalk  -74.116150
17   10/31/2013 12:52:46 AM           Illegal Parking  -73.888173
18   10/31/2013 12:51:00 AM    Street Light Condition         NaN
```

Noise-complaint data

Suppose we're only interested in the noise complaint data.

```
>>> noise = data[data["Complaint Type"]=="Noise - Street/Sidewalk"]
>>> noise[0:3]
  Unique Key          Created Date            Closed Date Agency
0   26589651 10/31/2013 02:08:41 AM                     NaN   NYPD
16  26594086 10/31/2013 12:54:03 AM 10/31/2013 02:16:39 AM   NYPD
25  26591573 10/31/2013 12:35:18 AM 10/31/2013 02:41:35 AM   NYPD

                         Agency Name            Complaint Type
0   New York City Police Department   Noise - Street/Sidewalk    ...
16  New York City Police Department   Noise - Street/Sidewalk
25  New York City Police Department   Noise - Street/Sidewalk
>>> noise.shape
(1928, 52)
```

How did that work?

The last command picked out the correct entries. How did that work?

```
>>> data["Complaint Type"] == "Noise - Street/Sidewalk"
  0          True
  1          False
  2          False
  3          False
  4          False
  ...
  111064     False
  111065     False
  111066     True
  111067     False
  111068     False
Name: Complaint Type, Length: 111069, dtype: bool
```

When we index our data with this array of booleans, we pick out the entries we're interested in.

More pruning

Combine more than one condition to restrict the search further.

```
>>> is_noise = data["Complaint Type"] == "Noise - Street/Sidewalk"

>>> in_brooklyn = data["Borough"] == "BROOKLYN"

>>> b_noise = data[is_noise & in_brooklyn]

>>> b_noise[["Complaint Type", "Borough", "Descriptor"]].head(3)

        Complaint Type            Borough     Descriptor
  31    Noise - Street/Sidewalk   BROOKLYN   Loud Music/Party
  49    Noise - Street/Sidewalk   BROOKLYN   Loud Talking
  109   Noise - Street/Sidewalk   BROOKLYN   Loud Music/Party
```

Who complains the most?

So which borough is responsible for the most complaints?

```
>>> noise = data[is_noise]

>>> noise["Borough"].value_counts()
MANHATTAN          917
BROOKLYN           456
BRONX              292
QUEENS             226
STATEN ISLAND       36
Unspecified          1
dtype: int64
```

Manhattan!