Data Cleaning and Preparation

Part 4

String Manipulation

String Object Methods

- In many string munging and scripting applications, built-in string methods are sufficient.
- As an example, a comma-separated string can be broken into pieces with `split`:

```
In [132]: val = 'a,b,  guido'

In [133]: val.split(',')
Out[133]: ['a', 'b', '  guido']
```

- `split` is often combined with `strip` to trim whitespace (including line breaks):

```
In [134]: pieces = [x.strip() for x in val.split(',')]

In [135]: pieces
Out[135]: ['a', 'b', 'guido']
```

- These substrings could be concatenated together with a two-colon delimiter using addition:

```
In [136]: first, second, third = pieces

In [137]: first + '::' + second + '::' + third
Out[137]: 'a::b::guido'
```

- But this isn’t a practical generic method.
- A faster and more Pythonic way is to pass a list or tuple to the `join` method on the string `'::'`:

```
In [138]: '::'.join(pieces)
Out[138]: 'a::b::guido'
```

- Other methods are concerned with locating substrings.
- Using Python's `in` keyword is the best way to detect a substring, though `index` and `find` can also be used:

```
In [139]: val
Out[139]: 'a,b,  guido'

In [140]: 'guido' in val
Out[140]: True

In [141]: val.index(',')
Out[141]: 1

In [142]: val.find(':')
Out[142]: -1
```

- Note the difference between `find` and `index` is that `index` raises an exception if the string isn't found (versus returning `-1`):

```
In [143]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-143-2c016e7367ac> in <module>
----> 1 val.index(':')

ValueError: substring not found
```

- Relatedly, `count` returns the number of occurrences of a particular substring:

```
In [144]: val.count(',')
Out[144]: 2
```

- `replace` will substitute occurrences of one pattern for another.
- It is commonly used to delete patterns, too, by passing an empty string:

```
In [145]: val.replace(',', '::')
Out[145]: 'a::b::  guido'

In [146]: val.replace(',', '')
Out[146]: 'ab  guido'
```

Regular Expressions

- *Regular expressions* provide a flexible way to search or match (often more complex) string patterns in text.
- A single expression, commonly called a `regex`, is a string formed according to the regular expression language.
- Python's built-in `re` module is responsible for applying regular expressions to strings.

- The `re` module functions fall into three categories: pattern matching, substitution, and splitting.
- Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes.
- Let's look at a simple example: suppose we wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines).
- The regex describing one or more whitespace characters is `\s+`:

```
In [147]: import re

In [148]: text = "foo    bar\t baz  \tqux"

In [149]: re.split('\s+', text)
Out[149]: ['foo', 'bar', 'baz', 'qux']
```

- When you call `re.split('\s+', text)`, the regular expression is first compiled, and then its `split` method is called on the passed text.
- You can compile the regex yourself with `re.compile`, forming a reusable regex object:

```
In [150]: regex = re.compile('\s+')

In [151]: regex.split(text)
Out[151]: ['foo', 'bar', 'baz', 'qux']
```

- If, instead, you wanted to get a list of all patterns matching the regex, you can use the `findall` method:

```
In [152]: regex.findall(text)
Out[152]: ['    ', '\t ', '  \t']
```

- `match` and `search` are closely related to `findall`.
- While `findall` returns all matches in a string, `search` returns only the first match.
- More rigidly, `match` only matches at the beginning of the string.

```
In [153]: text = """Dave dave@google.com
          Steve steve@gmail.com
          Rob rob@gmail.com
          Ryan ryan@yahoo.com
          """
          pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

          # re.IGNORECASE makes the regex case-insensitive
          regex = re.compile(pattern, flags=re.IGNORECASE)
```

- Using `findall` on the text produces a list of the email addresses:

```
In [154]: regex.findall(text)
Out[154]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

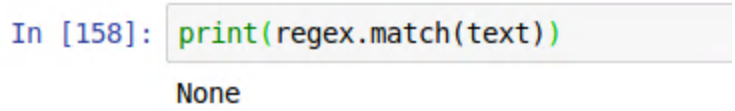
- `search` returns a special match object for the first email address in the text.
- For the preceding regex, the match object can only tell us the start and end position of the pattern in the string:

```
In [155]: m = regex.search(text)

In [156]: m
Out[156]: <re.Match object; span=(5, 20), match='dave@google.com'>

In [157]: text[m.start():m.end()]
Out[157]: 'dave@google.com'
```

- `regex.match` returns `None`, as it only will match if the pattern occurs at the start of the string:



- Relatedly, `sub` will return a new string with occurrences of the pattern replaced by the a new string:

```
In [159]: print(regex.sub('REDACTED', text))

Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

- Suppose you wanted to find email addresses and simultaneously segment each address into its three components: username, domain name, and domain suffix.
- To do this, put parentheses around the parts of the pattern to segment:

```
In [160]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [161]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

- A match object produced by this modified regex returns a tuple of the pattern components with its `groups` method:

```
In [162]: m = regex.match('wesm@bright.net')

In [163]: m.groups()
Out[163]: ('wesm', 'bright', 'net')
```

- `findall` returns a list of tuples when the pattern has groups:

```
In [164]: regex.findall(text)

Out[164]: [('dave', 'google', 'com'),
           ('steve', 'gmail', 'com'),
           ('rob', 'gmail', 'com'),
           ('ryan', 'yahoo', 'com')]
```

- `sub` also has access to groups in each match using special symbols like `\1` and `\2`.
- The symbol `\1` corresponds to the first matched group, `\2` corresponds to the second, and so forth:

```
In [165]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))

Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Vectorized String Functions in pandas

- Cleaning up a messy dataset for analysis often requires a lot of string munging and regularization.
- To complicate matters, a column containing strings will sometimes have missing data.

```
In [166]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
                  'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [167]: data = pd.Series(data)

In [168]: data

Out[168]: Dave     dave@google.com
          Steve    steve@gmail.com
          Rob        rob@gmail.com
          Wes                  NaN
          dtype: object

In [169]: data.isnull()

Out[169]: Dave     False
          Steve    False
          Rob      False
          Wes       True
          dtype: bool
```

- You can apply string and regular expression methods can be applied (passing a `lambda` or other function) to each value using `data.map`, but it will fail on the NA (null) values.
- To cope with this, Series has array-oriented methods for string operations that skip NA values.
- These are accessed through Series's `str` attribute; for example, we could check whether each email address has `'gmail'` in it with `str.contains`:

```
In [170]: data.str.contains('gmail')

Out[170]: Dave     False
          Steve     True
          Rob       True
          Wes        NaN
          dtype: object
```

- Regular expressions can be used, too, along with any `re` options like `IGNORECASE`:

```
In [171]: pattern
Out[171]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [172]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[172]: Dave     [(dave, google, com)]
          Steve    [(steve, gmail, com)]
          Rob        [(rob, gmail, com)]
          Wes                        NaN
          dtype: object
```

- There are a couple of ways to do vectorized element retrieval.

```
In [173]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [174]: matches

Out[174]: Dave     True
          Steve    True
          Rob      True
          Wes       NaN
          dtype: object
```

In [178]:
```
data.str.extract(pattern, flags=re.IGNORECASE)
```

Out[178]:

	0	1	2
Dave	dave	google	com
Steve	steve	gmail	com
Rob	rob	gmail	com
Wes	NaN	NaN	NaN