String Manipulation

Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing. Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas adds to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

In [1]:
```
import pandas as pd
import numpy as np
```

Data types and conversion

In [2]:
```
s = pd.Series([1, 2, 3, None])
s
s.dtype
```

Out[2]:
```
dtype('float64')
```

In [3]:
```
s = pd.Series([1, 2, 3, None], dtype=pd.Int64Dtype())
#s = pd.Series([1, 2, 3, None], dtype="Int64")
s
print(s.isna())
s.dtype
```

```
0    False
1    False
2    False
3     True
dtype: bool
```

Out[3]:
```
Int64Dtype()
```

In [4]:
```
s = pd.Series(['one', 'two', None, 'three'], dtype=pd.StringDtype())
s
```

Out[4]:
```
0      one
1      two
2     <NA>
3    three
dtype: string
```

In [5]:
```
df = pd.DataFrame({"A": [1, 2, None, 4],
                   "B": ["one", "two", "three", None],
                   "C": [False, None, False, True]})
df
df["A"] = df["A"].astype("Int64")
df["B"] = df["B"].astype("string")
df["C"] = df["C"].astype("boolean")
df
print(df["A"].dtype)
print(df["B"].dtype)
print(df["C"].dtype)
```

```
Int64
string
boolean
```

String Object Methods

In many string munging applications, built-in string methods are sufficient.

In [6]:
```
val = "a,b,    guido"
val.split(",")
```

Out[6]:
```
['a', 'b', '    guido']
```

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

In [7]:
```
pieces = [x.strip() for x in val.split(",")]
pieces
```

Out[7]:
```
['a', 'b', 'guido']
```

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

In [8]:
```
first, second, third = pieces
first + "::" + second + "::" + third
```

Out[8]:
```
'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 [9]:
```
"::".join(pieces)
```

Out[9]:
```
'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 [10]:
```
print("guido" in val)
```

```
True
```

In [11]:
```
my_string = "Hello, world!"
index_of_comma = my_string.index(",")
#index_of_plus = my_string.index("+")
print("Index of comma:", index_of_comma)
#print(index_of_plus)
```

```
Index of comma: 5
```

In [12]:
```
new_string = "example:value"
index_of_colon = new_string.find(":")
index_of_plus = new_string.find("+")
print("Index of colon:", index_of_colon)
print(index_of_plus)
```

```
Index of colon: 7
-1
```

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

In [13]:
```
val.count(",")
```

Out[13]:
```
2
```

In [14]:
```
val.replace(",", "::")
#val.replace(",", "")
```

Out[14]:
```
'a::b::    guido'
```

Argument	Description
`count`	Return the number of non-overlapping occurrences of substring in the string.
`endswith`	Returns `True` if string ends with suffix.
`startswith`	Returns `True` if string starts with prefix.
`join`	Use string as delimiter for concatenating a sequence of other strings.
`index`	Return position of first character in substring if found in the string; raises `ValueError` if not found.
`find`	Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found.
`rfind`	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found.
`replace`	Replace occurrences of string with another string.
`strip`, `rstrip`, `lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower`	Convert alphabet characters to lowercase.
`upper`	Convert alphabet characters to uppercase.
`casefold`	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
`ljust`, `rjust`	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

Regular Expression

Regular expressions provide a flexible way to search or match (often more complex) string patterns in text.

The art of writing regular expressions could be a chapter of its own and thus is outside the this seminar's scope. There are many excellent tutorials and references available on the internet and in other books.

In [15]:
```
import re
```

In [16]:
```
text = "foo    bar\t baz  \nqux"
re.split(r"\s+", text)
#\s+ represents one or more whitespace characters.
#The \s is a shorthand character class representing any whitespace character (space, tab, newli
#The + quantifier means "one or more occurrences."
```

Out[16]:
```
['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 [17]:
```
regex = re.compile(r"\s+")

regex.split(text)
```

Out[17]:
```
['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 [18]:
```
regex = re.compile(r"\s+")
regex.findall(text)
```

Out[18]:
```
['    ', '\t ', '  \n']
```

Creating a regex object with re.compile is highly recommended if you intend to apply the same expression to many strings; doing so will save CPU cycles.

"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 [19]:
```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
Prashant prashant@gmail.com"""
pattern = r"[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
#[A-Z0-9._%+-]+: Matches one or more uppercase letters, digits, dots, underscores,
#percent signs, plus signs, or hyphens before the "@" symbol.@: Matches the "@" symbol.
#[A-Z0-9.-]+: Matches one or more uppercase letters, digits, dots, or hyphens after the "@" symb
#in the domain name.\.: Matches the dot (.) before the top-level domain (TLD).
#[A-Z]{2,4}: Matches two to four uppercase letters, representing the TLD (e.g., com, org).

# 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 [20]:
```
regex.findall(text)
```

Out[20]:
```
['dave@google.com',
 'steve@gmail.com',
 'rob@gmail.com',
 'ryan@yahoo.com',
 'prashant@gmail.com']
```

In [21]:
```
m_search = regex.search(text)
print(m_search)
m_match = regex.match(text)
print(m_match)
#regex.match returns None, as it only will match if the pattern occurs at the start of the stri
```

```
<re.Match object; span=(5, 20), match='dave@google.com'>
None
```

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

In [22]:
```
print(regex.sub("REDACTED", text))
```

```
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
Prashant 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 [23]:
```
pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [24]:
```
new_text = """prashant@example.com
example1@example.com
hello@example.com
example@com
example"""
m= regex.match(new_text)
print(m)
m.groups()
```

```
<re.Match object; span=(0, 20), match='prashant@example.com'>
```

Out[24]:
```
('prashant', 'example', 'com')
```

In [25]:
```
m= regex.findall(new_text)
m
```

Out[25]:
```
[('prashant', 'example', 'com'),
 ('example1', 'example', 'com'),
 ('hello', 'example', '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 [26]:
```
m= regex.sub(r"'Username:\1, Domain:\2, Suffix:\3", new_text)
print(m)
```

```
'Username:prashant, Domain:example, Suffix:com
'Username:example1, Domain:example, Suffix:com
'Username:hello, Domain:example, Suffix:com
example@com
example
```

There is much more to regular expressions in Python, most of which is outside the scope of this seminar

Argument	Description
`findall`	Return all non-overlapping matching patterns in a string as a list
`finditer`	Like `findall`, but returns an iterator
`match`	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None
`search`	Scan string for match to pattern; returning a match object if so; unlike `match`, the match can be anywhere in the string as opposed to only at the beginning
`split`	Break string into pieces at each occurrence of pattern
`sub, subn`	Replace all (`sub`) or first n occurrences (`subn`) of pattern in string with replacement expression; use symbols `\1, \2, ...` to refer to match group elements in the replacement string

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:

These functions are "vectorized" in the sense that they operate on entire arrays of data at once, rather than iterating over each element individually. This approach significantly improves performance, especially for large datasets, by leveraging the optimizations available in the underlying numpy library.

In [27]:
```
data = {"Dave": "dave@google.com", "Steve": "steve@gmail.com",
        "Rob": "rob@gmail.com", "Wes": np.nan}
data = pd.Series(data)
data
```

Out[27]:
```
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                  NaN
dtype: object
```

In [28]:
```
data.isna()
```

Out[28]:
```
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 [29]:
```
data.str.contains("gmail")
```

Out[29]:
```
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

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

In [30]:
```
pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"
data.str.findall(pattern, flags=re.IGNORECASE)
```

Out[30]:
```
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. Either use str.get or index into the str attribute:

In [45]:
```
matches = data.str.findall(pattern, flags=re.IGNORECASE)
matches
```

Out[45]:
```
Dave     [(dave, google, com)]
Steve    [(steve, gmail, com)]
Rob        [(rob, gmail, com)]
Wes                        NaN
dtype: object
```

In [55]:
```
result = matches.str.get(1)
print(result)
```

```
Dave    NaN
Steve   NaN
Rob     NaN
Wes     NaN
dtype: float64
```

In [56]:
```
matches.str[0]
```

Out[56]:
```
Dave     (dave, google, com)
Steve    (steve, gmail, com)
Rob        (rob, gmail, com)
Wes                      NaN
dtype: object
```

You can similarly slice strings using this syntax:

In [57]:
```
data.str[:5]
```

Out[57]:
```
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`extract`	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
`endswith`	Equivalent to `x.endswith(pattern)` for each element
`startswith`	Equivalent to `x.startswith(pattern)` for each element
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve i-th element)
`isalnum`	Equivalent to built-in `str.alnum`
`isalpha`	Equivalent to built-in `str.isalpha`
`isdecimal`	Equivalent to built-in `str.isdecimal`
`isdigit`	Equivalent to built-in `str.isdigit`
`islower`	Equivalent to built-in `str.islower`
`isnumeric`	Equivalent to built-in `str.isnumeric`
`isupper`	Equivalent to built-in `str.isupper`
`join`	Join strings in each element of the Series with passed separator
`len`	Compute length of each string
`lower, upper`	Convert cases; equivalent to `x.lower()` or `x.upper()` for each element

Method	Description
`match`	Use `re.match` with the passed regular expression on each element, returning matched groups as list
`pad`	Add whitespace to left, right, or both sides of strings
`center`	Equivalent to `pad(side='both')`
`repeat`	Duplicate values (e.g., `s.str.repeat(3)` is equivalent to `x * 3` for each string)
`replace`	Replace occurrences of pattern/regex with some other string
`slice`	Slice each string in the Series
`split`	Split strings on delimiter or regular expression
`strip`	Trim whitespace from both sides, including newlines
`rstrip`	Trim whitespace on right side
`lstrip`	Trim whitespace on left side

#END OF PART 3 of Data Wrangling
By: Prashant Bikram Shah