Lecture 3: Advanced data wrangling with Pandas

Contents

DSCI 511
Python Programming for Data Science

Lecture learning objectives

- Manipulate strings in Python with methods like `find`, `replace` and `join`.
- Access string methods in Pandas via `Series.str`.
- Understand how to use regular expressions in Pandas for wrangling strings.
- Differentiate between datetime object in Pandas such as `Timestamp`, `Timedelta`, `Period`, `DateOffset`.

- Create these datetime objects with functions like `pd.Timestamp()`, `pd.Period()`, `pd.date_range()`, `pd.period_range()`.
- Index a datetime index with partial string indexing.
- Perform basic datetime operations like splitting a datetime into constituent parts (e.g., `year`, `weekday`, `second`, etc), apply offsets, change timezones, and resample with `.resample()`.

```
import pandas as pd
import numpy as np
import altair as alt
pd.set_option('display.max_rows', 10)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 import pandas as pd
      2 import numpy as np
----> 3 import altair as alt
      4 pd.set_option('display.max_rows', 10)

ModuleNotFoundError: No module named 'altair'
```

Strings

Strings are a core data type in Python used for text-based data. They are enclosed in single (`'`) or double (`"`) quotes.

```
single = 'I love Python!'
double = "I love Data Science!"
single
```

```
'I love Python!'
```

Sometimes our strings may themselves contain quotation marks or apostrophes.

```
single = 'That's weird'
# raises error
```

```
  Cell In[3], line 1
    single = 'That's weird'
                          ^
SyntaxError: unterminated string literal (detected at line 1)
```

We can use a backlash `\` (called an escape character) to prevent Python from interpreting `'` as a string delimiter

```
single = 'That\'s weird'
single
```

```
"That's weird"
```

You may also have strings that contain backslashes, so you need to tell Python not to treat them as escape characters.

```
path1 = 'Documents\\Lecture1'
path2 = r'Documents\Lecture1\solutions.ipynb'
```

Manipulating Strings

- Strings can be concatenated using the `+` operator.
- The `len()` function returns the length of a string, i.e. the number of characters it contains.
- Use the `lower()`, `upper()` and `capitalize()` methods for case conversion.
- You can remove leading and trailing whitespace from strings using `strip()`, `lstrip()`, and `rstrip()`.

```
first = 'florence'
last = 'nightingale'
first + last
```

```
'florencenightingale'
```

```
len(first)
```

```
8
```

```
first.upper()
```

```
'FLORENCE'
```

```
fullname = '   Florence Nightingale       '
fullname.strip()
```

```
'Florence Nightingale'
```

Note: A whitespace character is any character that is displayed as “blank space” when text is rendered for display. Usual spaces and tabs (`'t'`) are both whitespace characters. You might also frequently encounter new line characters (`'\n'`), which force line breaks. Less commonly used whitespace characters are carriage returns, vertical tabs, and form feeds.

Substrings, Splitting and Joining

- You can slice a string using integer indexing.
- The `find()` and `index()` methods help you locate substrings.
- The `replace()` method allows you to replace occurences of one substring with another.
- The `split()` method splits one string into a list of substrings based on a delimiter
- The `join()` method joins a list of strings into one, using a specified delimiter

```
text = 'The-University-of-British-Columbia'
text.find('v')
```

```
7
```

```
text[1:].find('v')
```

```
6
```

```
text.find('z')
```

```
-1
```

```
text.replace('-',' ') # Note this returns a new string, and does not modify the original in place.
```

```
'The University of British Columbia'
```

```
words = text.split('-')
words
```

```
['The', 'University', 'of', 'British', 'Columbia']
```

```
' '.join(words)
```

```
'The University of British Columbia'
```

Formatted Strings

Python can “fill in the blanks” using specified input to create strings. One way of doing this is using f-strings. These are useful when you want to print statements including variables.

```python
num = 12
print(f"There are {num} months in a year")
```

```
There are 12 months in a year
```

The `print()` function above has some useful options that you can find in the Python documentation.

Using String Methods in Pandas

In previous lectures, we saw that Pandas can apply arithmetic operations on an entire Series object "all at once". We can do the same for text data using `Series.str` which gives us access to the string methods we have seen.

```python
imdb = pd.read_csv('data/imdb.csv')
imdb['Genre']
```

```
0                        Drama
1                 Crime, Drama
2         Action, Crime, Drama
3                 Crime, Drama
4                 Crime, Drama
                 ...
995      Comedy, Drama, Romance
996             Drama, Western
997        Drama, Romance, War
998                 Drama, War
999    Crime, Mystery, Thriller
Name: Genre, Length: 1000, dtype: object
```

```
genres = imdb['Genre'].str.split(',', expand = True)
genres
```

	0	1	2
0	Drama	None	None
1	Crime	Drama	None
2	Action	Crime	Drama
3	Crime	Drama	None
4	Crime	Drama	None
...	...	...	...
995	Comedy	Drama	Romance
996	Drama	Western	None
997	Drama	Romance	War
998	Drama	War	None
999	Crime	Mystery	Thriller

1000 rows × 3 columns

Notice there's some whitespace in the resulting columns!

```
genres.loc[1,1]
```

```
' Drama'
```

We know how to fix this

```
genres[1] = genres[1].str.strip()
genres.loc[1,1]
```

```
'Drama'
```

There are many string methods available to use in pandas. A full list is available in the [documentation](#).

Regular Expressions or REGEX

- A regular expression (regex) is a sequence of characters that defines a search pattern.
- Regex can do some truly magical things, so keep it in mind for complicated text wrangling!
- Regex can also be difficult to get right, but there are many online tools (e.g. [RegExr](#)) that can help you find the correct patterns.

You will learn more about regex in DSCI 521, so you don't need to learn it now. Here, just as an example, we use regex can find all movie titles in our IMDB dataset that start with a vowel and end with a consonant.

```
findpattern = imdb['Series_Title'].str.findall(r'^[AEIOU].*[^aeiou]$')
findpattern.loc[:10]
```

```
0                []
1                []
2                []
3                []
4                []
5                []
6                []
7                []
8       [Inception]
9                []
10               []
Name: Series_Title, dtype: object
```

Let's break down that regex:

- The leading `^` specifies the start of a string.
- Square brackets match a single character, so `[AEIOU]` is saying we want the first character to be a vowel.
- `.` matches any character and `*` means '0 or more times', so `.*` will match any number of any characters in the middle of our string
- Having a `^` inside and at the start of square brackets indicates a "not" and `$` matches the end of a string. So `[^aeiou]$` means we don't want the *last* character to be a vowel.

We could have used regex to split the genres as well, which would allow us to specify a greater number of possible delimiters.

```
imdb['Genre'].str.split('[,:.!]', expand=True)
```

	0	1	2
0	Drama	None	None
1	Crime	Drama	None
2	Action	Crime	Drama
3	Crime	Drama	None
4	Crime	Drama	None
...	...	...	...
995	Comedy	Drama	Romance
996	Drama	Western	None
997	Drama	Romance	War
998	Drama	War	None
999	Crime	Mystery	Thriller

1000 rows × 3 columns

Many `DataFrame.str()` methods accept regular expression as default, but you don't have to use regex if you don't need it.

```
imdb['Genre'].str.contains('ri', regex=False) # This tests for exact matches
```

```
0      False
1       True
2       True
3       True
4       True
       ...
995    False
996    False
997    False
998    False
999     True
Name: Genre, Length: 1000, dtype: bool
```

Time Series Data

Data Scientists often work with “time series” data, i.e. data that is indexed by time periods. We will breifly introduce time series in pandas here; you will encounter them much more in DSCI 574.

Pandas provides several datetime objects, including:

- Timestamp
- Timedelta
- Period
- DateOffset

We explore a few of them below.

Creating Datetimes

Most commonly you will want to

- Create a single point in time with `pd.Timestamp()`.
- Create a span of time with `pd.Period()`.
- Create an array of times with `pd.date_range()` or `pd.period_range()`.

```
pd.Timestamp('July 29, 2005') # parsed from string
```

```
Timestamp('2005-07-29 00:00:00')
```

```
pd.Timestamp(year = 2005, month = 7, day = 9) # pass data directly
```

```
Timestamp('2005-07-09 00:00:00')
```

Above we have a specific point in time. We can use `pd.Period()` to specify a span of time (like a day)

```
span = pd.Period('2005-07-09')
print(span)
print(span.start_time)
print(span.end_time)
```

```
2005-07-09
2005-07-09 00:00:00
2005-07-09 23:59:59.999999999
```

```
span + pd.Timedelta('1 day')
```

```
Period('2005-07-10', 'D')
```

Pandas understands when a `Timestamp` lies within a `Period`.

```
point = pd.Timestamp('2005-07-09 12:00')
span = pd.Period('2005-07-09')
print(f'Point: {point}')
print(f' Span: {span}')
print(f'Point in span? {span.start_time < point < span.end_time}')
```

```
Point: 2005-07-09 12:00:00
 Span: 2005-07-09
Point in span? True
```

Often, you will want to create **arrays** of datetimes, not just single values. In pandas, these come in the form of a `DatetimeIndex`, `PeriodIndex` or `TimedeltaIndex`.

```
pd.date_range('2020-09-01 12:00',
              '2020-09-11 12:00',
              freq='2d')
```

```
DatetimeIndex(['2020-09-01 12:00:00', '2020-09-03 12:00:00',
               '2020-09-05 12:00:00', '2020-09-07 12:00:00',
               '2020-09-09 12:00:00', '2020-09-11 12:00:00'],
              dtype='datetime64[ns]', freq='2D')
```

```
pd.period_range('2020-09-01',
                '2020-09-11',
                freq='d')
```

```
PeriodIndex(['2020-09-01', '2020-09-02', '2020-09-03', '2020-09-04',
             '2020-09-05', '2020-09-06', '2020-09-07', '2020-09-08',
             '2020-09-09', '2020-09-10', '2020-09-11'],
            dtype='period[D]')
```

We can use `Timedelta` to add or subtract time from a `DatetimeIndex` or `PeriodIndex`.

```
pd.date_range('2020-09-01 12:00', '2020-09-11 12:00', freq='D') + pd.Timedelta('1.5 hours')
```

```
DatetimeIndex(['2020-09-01 13:30:00', '2020-09-02 13:30:00',
               '2020-09-03 13:30:00', '2020-09-04 13:30:00',
               '2020-09-05 13:30:00', '2020-09-06 13:30:00',
               '2020-09-07 13:30:00', '2020-09-08 13:30:00',
               '2020-09-09 13:30:00', '2020-09-10 13:30:00',
               '2020-09-11 13:30:00'],
              dtype='datetime64[ns]', freq='D')
```

Finally, pandas represents missing datetimes with `NaT`.

Converting Existing Data

It's fairly common to have an array of dates as strings. We can use `pd.to_datetime()` to convert these to a datetime format.

```
cycling = pd.read_csv('data/cycling_data.csv')
cycling.head() # Date column will be read in as strings
```

	Date	Name	Type	Time	Distance	Comments
0	10 Sep 2019, 00:13:04	Afternoon Ride	Ride	2084	12.62	Rain
1	10 Sep 2019, 13:52:18	Morning Ride	Ride	2531	13.03	rain
2	11 Sep 2019, 00:23:50	Afternoon Ride	Ride	1863	12.52	Wet road but nice weather
3	11 Sep 2019, 14:06:19	Morning Ride	Ride	2192	12.84	Stopped for photo of sunrise
4	12 Sep 2019, 00:28:05	Afternoon Ride	Ride	1891	12.48	Tired by the end of the week

```python
cycling['Date'] = pd.to_datetime(cycling['Date'])
cycling['Date']
```

```
0    2019-09-10 00:13:04
1    2019-09-10 13:52:18
2    2019-09-11 00:23:50
3    2019-09-11 14:06:19
4    2019-09-12 00:28:05
5    2019-09-16 13:57:48
6    2019-09-17 00:15:47
7    2019-09-17 13:43:34
8    2019-09-18 13:49:53
9    2019-09-18 00:15:52
10   2019-09-19 00:30:01
11   2019-09-19 13:52:09
12   2019-09-20 01:02:05
13   2019-09-23 13:50:41
14   2019-09-24 00:35:42
15   2019-09-24 13:41:24
16   2019-09-25 00:07:21
17   2019-09-25 13:35:41
18   2019-09-26 00:13:33
19   2019-09-26 13:42:43
20   2019-09-27 01:00:18
21   2019-09-30 13:53:52
22   2019-10-01 00:15:07
23   2019-10-01 13:45:55
24   2019-10-02 00:13:09
25   2019-10-02 13:46:06
26   2019-10-03 00:45:22
27   2019-10-03 13:47:36
28   2019-10-04 01:08:08
29   2019-10-09 13:55:40
30   2019-10-10 00:10:31
31   2019-10-10 13:47:14
32   2019-10-11 00:16:57
Name: Date, dtype: datetime64[ns]
```

We can actually tell pandas to do this within the `read_csv` statement. Let's also set the datetime as the Index.

```
cycling = pd.read_csv('data/cycling_data.csv',
                      parse_dates=True,
                      index_col = 0)
cycling
```

	Name	Type	Time	Distance	Comments
Date					
2019-09-10 00:13:04	Afternoon Ride	Ride	2084	12.62	Rain
2019-09-10 13:52:18	Morning Ride	Ride	2531	13.03	rain
2019-09-11 00:23:50	Afternoon Ride	Ride	1863	12.52	Wet road but nice weather
2019-09-11 14:06:19	Morning Ride	Ride	2192	12.84	Stopped for photo of sunrise
2019-09-12 00:28:05	Afternoon Ride	Ride	1891	12.48	Tired by the end of the week
2019-09-16 13:57:48	Morning Ride	Ride	2272	12.45	Rested after the weekend!
2019-09-17 00:15:47	Afternoon Ride	Ride	1973	12.45	Legs feeling strong!
2019-09-17 13:43:34	Morning Ride	Ride	2285	12.60	Raining
2019-09-18 13:49:53	Morning Ride	Ride	2903	14.57	Raining today
2019-09-18 00:15:52	Afternoon Ride	Ride	2101	12.48	Pumped up tires
2019-09-19 00:30:01	Afternoon Ride	Ride	48062	12.48	Feeling good
2019-09-19 13:52:09	Morning Ride	Ride	2090	12.59	Getting colder which is nice
2019-09-20 01:02:05	Afternoon Ride	Ride	2961	12.81	Feeling good
2019-09-23 13:50:41	Morning Ride	Ride	2462	12.68	Rested after the weekend!
2019-09-24 00:35:42	Afternoon Ride	Ride	2076	12.47	Oiled chain, bike feels smooth
2019-09-24 13:41:24	Morning Ride	Ride	2321	12.68	Bike feeling much smoother
2019-09-25 00:07:21	Afternoon Ride	Ride	1775	12.10	Feeling really tired
2019-09-25 13:35:41	Morning Ride	Ride	2124	12.65	Stopped for photo of sunrise
2019-09-26 00:13:33	Afternoon Ride	Ride	1860	12.52	raining

	Name	Type	Time	Distance	Comments
Date					
2019-09-26 13:42:43	Morning Ride	Ride	2350	12.91	Detour around trucks at Jericho
2019-09-27 01:00:18	Afternoon Ride	Ride	1712	12.47	Tired by the end of the week
2019-09-30 13:53:52	Morning Ride	Ride	2118	12.71	Rested after the weekend!
2019-10-01 00:15:07	Afternoon Ride	Ride	1732	NaN	Legs feeling strong!
2019-10-01 13:45:55	Morning Ride	Ride	2222	12.82	Beautiful morning! Feeling fit
2019-10-02 00:13:09	Afternoon Ride	Ride	1756	NaN	A little tired today but good weather
2019-10-02 13:46:06	Morning Ride	Ride	2134	13.06	Bit tired today but good weather
2019-10-03 00:45:22	Afternoon Ride	Ride	1724	12.52	Feeling good
2019-10-03 13:47:36	Morning Ride	Ride	2182	12.68	Wet road
2019-10-04 01:08:08	Afternoon Ride	Ride	1870	12.63	Very tired, riding into the wind
2019-10-09 13:55:40	Morning Ride	Ride	2149	12.70	Really cold! But feeling good
2019-10-10 00:10:31	Afternoon Ride	Ride	1841	12.59	Feeling good after a holiday break!
2019-10-10 13:47:14	Morning Ride	Ride	2463	12.79	Stopped for photo of sunrise
2019-10-11 00:16:57	Afternoon Ride	Ride	1843	11.79	Bike feeling tight, needs an oil and pump

Indexing Datetimes

Datetime index objects are just like regular index objects and can be selected, sliced, filtered etc. We can do **partial string indexing** to select datetimes within a certain range.

```
cycling.loc['2019-10'] # All logs for Oct 2019
```

	Name	Type	Time	Distance	Comments
Date					
2019-10-01 00:15:07	Afternoon Ride	Ride	1732	NaN	Legs feeling strong!
2019-10-01 13:45:55	Morning Ride	Ride	2222	12.82	Beautiful morning! Feeling fit
2019-10-02 00:13:09	Afternoon Ride	Ride	1756	NaN	A little tired today but good weather
2019-10-02 13:46:06	Morning Ride	Ride	2134	13.06	Bit tired today but good weather
2019-10-03 00:45:22	Afternoon Ride	Ride	1724	12.52	Feeling good
2019-10-03 13:47:36	Morning Ride	Ride	2182	12.68	Wet road
2019-10-04 01:08:08	Afternoon Ride	Ride	1870	12.63	Very tired, riding into the wind
2019-10-09 13:55:40	Morning Ride	Ride	2149	12.70	Really cold! But feeling good
2019-10-10 00:10:31	Afternoon Ride	Ride	1841	12.59	Feeling good after a holiday break!
2019-10-10 13:47:14	Morning Ride	Ride	2463	12.79	Stopped for photo of sunrise
2019-10-11 00:16:57	Afternoon Ride	Ride	1843	11.79	Bike feeling tight, needs an oil and pump

Exact matching:

```
cycling.loc['2019-10-10 00:10:31']
```

```
Name                              Afternoon Ride
Type                                        Ride
Time                                        1841
Distance                                   12.59
Comments    Feeling good after a holiday break!
Name: 2019-10-10 00:10:31, dtype: object
```

And **slicing**:

```
cycling.loc['2019-10-01':'2019-10-13'] # raises error
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[37], line 1
----> 1 cycling.loc['2019-10-01':'2019-10-13'] # raises error

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1191, in _LocationIndexer.__geti
   1189 maybe_callable = com.apply_if_callable(key, self.obj)
   1190 maybe_callable = self._check_deprecated_callable_usage(key, maybe_callable)
-> 1191 return self._getitem_axis(maybe_callable, axis=axis)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1411, in _LocIndexer._getitem_ax
   1409 if isinstance(key, slice):
   1410     self._validate_key(key, axis)
-> 1411     return self._get_slice_axis(key, axis=axis)
   1412 elif com.is_bool_indexer(key):
   1413     return self._getbool_axis(key, axis=axis)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1443, in _LocIndexer._get_slice_
   1440     return obj.copy(deep=False)
   1442 labels = obj._get_axis(axis)
-> 1443 indexer = labels.slice_indexer(slice_obj.start, slice_obj.stop, slice_obj.step)
   1445 if isinstance(indexer, slice):
   1446     return self.obj._slice(indexer, axis=axis)

File ~/miniconda3/lib/python3.11/site-packages/pandas/core/indexes/datetimes.py:697, in DatetimeIndex.s
    694     in_index &= (end_casted == self).any()
    696 if not in_index:
--> 697     raise KeyError(
    698         "Value based partial slicing on non-monotonic DatetimeIndexes "
    699         "with non-existing keys is not allowed.",
    700     )
    701 indexer = mask.nonzero()[0][::step]
    702 if len(indexer) == len(self):

KeyError: 'Value based partial slicing on non-monotonic DatetimeIndexes with non-existing keys is not a
```

oops, that failed. The error suggests the index might not be sorted (i.e. it may not be monotonic). Let's fix that and try again.

```
cycling.sort_index().loc['2019-10-01':'2019-10-13']
```

	Name	Type	Time	Distance	Comments
Date					
2019-10-01 00:15:07	Afternoon Ride	Ride	1732	NaN	Legs feeling strong!
2019-10-01 13:45:55	Morning Ride	Ride	2222	12.82	Beautiful morning! Feeling fit
2019-10-02 00:13:09	Afternoon Ride	Ride	1756	NaN	A little tired today but good weather
2019-10-02 13:46:06	Morning Ride	Ride	2134	13.06	Bit tired today but good weather
2019-10-03 00:45:22	Afternoon Ride	Ride	1724	12.52	Feeling good
2019-10-03 13:47:36	Morning Ride	Ride	2182	12.68	Wet road
2019-10-04 01:08:08	Afternoon Ride	Ride	1870	12.63	Very tired, riding into the wind
2019-10-09 13:55:40	Morning Ride	Ride	2149	12.70	Really cold! But feeling good
2019-10-10 00:10:31	Afternoon Ride	Ride	1841	12.59	Feeling good after a holiday break!
2019-10-10 13:47:14	Morning Ride	Ride	2463	12.79	Stopped for photo of sunrise
2019-10-11 00:16:57	Afternoon Ride	Ride	1843	11.79	Bike feeling tight, needs an oil and pump

For getting all results between a certain time of day (potentially on different days) we can use `df.between_time()`.

```
cycling.between_time('00:00', '01:00')
```

	Name	Type	Time	Distance	Comments
Date					
2019-09-10 00:13:04	Afternoon Ride	Ride	2084	12.62	Rain
2019-09-11 00:23:50	Afternoon Ride	Ride	1863	12.52	Wet road but nice weather
2019-09-12 00:28:05	Afternoon Ride	Ride	1891	12.48	Tired by the end of the week
2019-09-17 00:15:47	Afternoon Ride	Ride	1973	12.45	Legs feeling strong!
2019-09-18 00:15:52	Afternoon Ride	Ride	2101	12.48	Pumped up tires
2019-09-19 00:30:01	Afternoon Ride	Ride	48062	12.48	Feeling good
2019-09-24 00:35:42	Afternoon Ride	Ride	2076	12.47	Oiled chain, bike feels smooth
2019-09-25 00:07:21	Afternoon Ride	Ride	1775	12.10	Feeling really tired
2019-09-26 00:13:33	Afternoon Ride	Ride	1860	12.52	raining
2019-10-01 00:15:07	Afternoon Ride	Ride	1732	NaN	Legs feeling strong!
2019-10-02 00:13:09	Afternoon Ride	Ride	1756	NaN	A little tired today but good weather
2019-10-03 00:45:22	Afternoon Ride	Ride	1724	12.52	Feeling good
2019-10-10 00:10:31	Afternoon Ride	Ride	1841	12.59	Feeling good after a holiday break!
2019-10-11 00:16:57	Afternoon Ride	Ride	1843	11.79	Bike feeling tight, needs an oil and pump

Manipulating Datetimes

For more complex filtering, we may have to **decompose** our timeseries. There are many methods and attributes that we can access.

```
cycling.index.weekday
```

```
Index([1, 1, 2, 2, 3, 0, 1, 1, 2, 2, 3, 3, 4, 0, 1, 1, 2, 2, 3, 3, 4, 0, 1, 1,
       2, 2, 3, 3, 4, 2, 3, 3, 4],
      dtype='int32', name='Date')
```

```
cycling.index.second
```

```
Index([ 4, 18, 50, 19,  5, 48, 47, 34, 53, 52,  1,  9,  5, 41, 42, 24, 21, 41,
       33, 43, 18, 52,  7, 55,  9,  6, 22, 36,  8, 40, 31, 14, 57],
      dtype='int32', name='Date')
```

```
cycling.index.day_name()
```

```
Index(['Tuesday', 'Tuesday', 'Wednesday', 'Wednesday', 'Thursday', 'Monday',
       'Tuesday', 'Tuesday', 'Wednesday', 'Wednesday', 'Thursday', 'Thursday',
       'Friday', 'Monday', 'Tuesday', 'Tuesday', 'Wednesday', 'Wednesday',
       'Thursday', 'Thursday', 'Friday', 'Monday', 'Tuesday', 'Tuesday',
       'Wednesday', 'Wednesday', 'Thursday', 'Thursday', 'Friday', 'Wednesday',
       'Thursday', 'Thursday', 'Friday'],
      dtype='object', name='Date')
```

```
cycling.index.month_name()
```

```
Index(['September', 'September', 'September', 'September', 'September',
       'September', 'September', 'September', 'September', 'September',
       'September', 'September', 'September', 'September', 'September',
       'September', 'September', 'September', 'September', 'September',
       'September', 'September', 'October', 'October', 'October', 'October',
       'October', 'October', 'October', 'October', 'October', 'October',
       'October'],
      dtype='object', name='Date')
```

If you're acting on a Series instead of a DatetimeIndex, you can access these methods using `.dt`.

```python
s = pd.Series(pd.date_range('2011-12-29', '2011-12-31'))
s.year  # raises error
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/var/folders/2w/xsy5y_ms0t74ssbfqt_m0tr80000gp/T/ipykernel_20747/3954837213.py in ?()
      1 s = pd.Series(pd.date_range('2011-12-29', '2011-12-31'))
----> 2 s.year  # raises error

~/miniconda3/lib/python3.11/site-packages/pandas/core/generic.py in ?(self, name)
   6295             and name not in self._accessors
   6296             and self._info_axis._can_hold_identifiers_and_holds_name(name)
   6297         ):
   6298             return self[name]
-> 6299         return object.__getattribute__(self, name)

AttributeError: 'Series' object has no attribute 'year'
```

```python
s.dt.year
```

```
0    2011
1    2011
2    2011
dtype: int32
```

Resampling and Aggregating

One of the most common operations you will want do when working with time series is resampling to a coarser/finer/regular resolution. For example, you may want to resample daily data to weekly data.

We can do that with the `.resample()` method. For example, let's resample the irregular cycling timeseries to a regular 12-hourly series

```
cycling.head()
```

	Name	Type	Time	Distance	Comments
Date					
2019-09-10 00:13:04	Afternoon Ride	Ride	2084	12.62	Rain
2019-09-10 13:52:18	Morning Ride	Ride	2531	13.03	rain
2019-09-11 00:23:50	Afternoon Ride	Ride	1863	12.52	Wet road but nice weather
2019-09-11 14:06:19	Morning Ride	Ride	2192	12.84	Stopped for photo of sunrise
2019-09-12 00:28:05	Afternoon Ride	Ride	1891	12.48	Tired by the end of the week

```
cycling.resample('1D')
```

```
<pandas.core.resample.DatetimeIndexResampler object at 0x108407750>
```

The parameter `'1D'` is telling pandas to sample once daily. The letter `D` is the alias for the Daily offset, some other commonly used ones include

- `'h'`, `'min'` and `'s'` for hours, minutes, and seconds (resp.)
- `'B'` for business day
- `'MS'` and `'ME'` for month start and month end (resp.)

A more complete list is available [here](#).

Let's examine this `Resampler` object a little more.

```python
cycling.resample('2D').groups
```

```
{Timestamp('2019-09-10 00:00:00'): np.int64(4),
 Timestamp('2019-09-12 00:00:00'): np.int64(5),
 Timestamp('2019-09-14 00:00:00'): np.int64(5),
 Timestamp('2019-09-16 00:00:00'): np.int64(8),
 Timestamp('2019-09-18 00:00:00'): np.int64(12),
 Timestamp('2019-09-20 00:00:00'): np.int64(13),
 Timestamp('2019-09-22 00:00:00'): np.int64(14),
 Timestamp('2019-09-24 00:00:00'): np.int64(18),
 Timestamp('2019-09-26 00:00:00'): np.int64(21),
 Timestamp('2019-09-28 00:00:00'): np.int64(21),
 Timestamp('2019-09-30 00:00:00'): np.int64(24),
 Timestamp('2019-10-02 00:00:00'): np.int64(28),
 Timestamp('2019-10-04 00:00:00'): np.int64(29),
 Timestamp('2019-10-06 00:00:00'): np.int64(29),
 Timestamp('2019-10-08 00:00:00'): np.int64(30),
 Timestamp('2019-10-10 00:00:00'): np.int64(33)}
```

The output isn't so decipherable, but our data has been aggregated into *groups*. Since some groups have more than one entry, we need to tell pandas what to do with these. One option is to apply `mean()` to get an aggregated summary for each group.

```
dfr = cycling[["Time","Distance"]].resample('1D').mean()
dfr
```

	Time	Distance
Date		
2019-09-10	2307.5	12.825
2019-09-11	2027.5	12.680
2019-09-12	1891.0	12.480
2019-09-13	NaN	NaN
2019-09-14	NaN	NaN
2019-09-15	NaN	NaN
2019-09-16	2272.0	12.450
2019-09-17	2129.0	12.525
2019-09-18	2502.0	13.525
2019-09-19	25076.0	12.535
2019-09-20	2961.0	12.810
2019-09-21	NaN	NaN
2019-09-22	NaN	NaN
2019-09-23	2462.0	12.680
2019-09-24	2198.5	12.575
2019-09-25	1949.5	12.375
2019-09-26	2105.0	12.715
2019-09-27	1712.0	12.470
2019-09-28	NaN	NaN

	Time	Distance
Date		
2019-09-29	NaN	NaN
2019-09-30	2118.0	12.710
2019-10-01	1977.0	12.820
2019-10-02	1945.0	13.060
2019-10-03	1953.0	12.600
2019-10-04	1870.0	12.630
2019-10-05	NaN	NaN
2019-10-06	NaN	NaN
2019-10-07	NaN	NaN
2019-10-08	NaN	NaN
2019-10-09	2149.0	12.700
2019-10-10	2152.0	12.690
2019-10-11	1843.0	11.790

There's quite a few `NaN`s in there. Some days do not have any data points– we can choose how to handle these using `fillna` (for example, we could fill 0 distance and 0 time).

The resampled index still has much of the same functionality we have seen before:

```
dfr['Weekday'] = dfr.index.day_name()
dfr.head(10)
```

	Time	Distance	Weekday
Date			
2019-09-10	2307.5	12.825	Tuesday
2019-09-11	2027.5	12.680	Wednesday
2019-09-12	1891.0	12.480	Thursday
2019-09-13	NaN	NaN	Friday
2019-09-14	NaN	NaN	Saturday
2019-09-15	NaN	NaN	Sunday
2019-09-16	2272.0	12.450	Monday
2019-09-17	2129.0	12.525	Tuesday
2019-09-18	2502.0	13.525	Wednesday
2019-09-19	25076.0	12.535	Thursday

We will study aggregate objects in pandas more thoroughly in Lecture 4, when we study the `groupby()` method.

Categorical Data

What is categorical data? A categorical variable takes on a limited, and usually fixed, number of possible values. Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

What is a Pandas categorical data type? Internally, the data structure consists of a `categories` array and an integer array of `codes` which point to the real value in the `categories` array.

Why use Pandas categoricals?

- They allow categories to be ordered, which is useful for mapping attributes like color or size to category levels in plots.
- They reduce memory usage and improve performance by storing repeated values more efficiently.
- They improve the performance of certain operations, such as groupby and sorting, by leveraging the categorical nature of the data.

Source: https://pandas.pydata.org/docs/user_guide/categorical.html

Examples of categorical data in Pandas

Let's load in `bean.csv` from the `data` directory. This file contains a small sample of a dataset from the [UC Irvine Machine Learning Repository](#).

```
bean = pd.read_csv('data/bean.csv')
bean.head()
```

	Unnamed: 0.1	Unnamed: 0	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentrici
0	111	1561	43144	765.795	258.033633	213.190144	1.210345	0.5633
1	394	2156	56857	952.885	328.132376	221.256783	1.483039	0.7384
2	478	372	35589	687.923	235.063906	192.994447	1.217983	0.5708
3	560	8551	43694	794.323	305.549901	182.507247	1.674180	0.8020
4	106	5409	91641	1147.618	448.764563	261.330642	1.717229	0.8129

There 18 columns in the dataset. Most are numerical, but the last column contains text data

```
bean.columns
```

```
Index(['Unnamed: 0.1', 'Unnamed: 0', 'Area', 'Perimeter', 'MajorAxisLength',
       'MinorAxisLength', 'AspectRation', 'Eccentricity', 'ConvexArea',
       'EquivDiameter', 'Extent', 'Solidity', 'roundness', 'Compactness',
       'ShapeFactor1', 'ShapeFactor2', 'ShapeFactor3', 'ShapeFactor4',
       'Class'],
      dtype='object')
```

```
bean['Class']
```

```
0         SEKER
1      BARBUNYA
2         SEKER
3          SIRA
4          CALI
         ...
267       SEKER
268      BOMBAY
269       SEKER
270       SEKER
271    BARBUNYA
Name: Class, Length: 272, dtype: object
```

A column of text data could have any number of distinct entries– potentially each one could be different. Let's check how many unique entries this column contains.

```
bean['Class'].nunique()
```

```
7
```

There are only 7 possibilities! As it turns out, they correspond to different classes of beans.

Pandas has a `category` data type for columns where the variable takes one of a small (usually fixed) number of categories. We can convert existing columns from `object` to `category` dtype as follows:

```
bean['Class'] = bean['Class'].astype('category')
bean['Class']
```

```
0         SEKER
1      BARBUNYA
2         SEKER
3          SIRA
4          CALI
         ...
267       SEKER
268      BOMBAY
269       SEKER
270       SEKER
271    BARBUNYA
Name: Class, Length: 272, dtype: category
Categories (7, object): ['BARBUNYA', 'BOMBAY', 'CALI', 'DERMASON', 'HOROZ', 'SEKER', 'SIRA']
```

```
bean['Class'].dtype
```

```
CategoricalDtype(categories=['BARBUNYA', 'BOMBAY', 'CALI', 'DERMASON', 'HOROZ', 'SEKER',
                  'SIRA'],
, ordered=False, categories_dtype=object)
```

We can look up the categories of a column by accessing the `Series.cat.categories` attribute. So to see the categories of the new ‘Class’ column, write:

```
bean['Class'].cat.categories
```

```
Index(['BARBUNYA', 'BOMBAY', 'CALI', 'DERMASON', 'HOROZ', 'SEKER', 'SIRA'], dtype='object')
```

Having categorical data is really useful for data vizualization. For example, we can plot bean length and width on the x and y axes (resp.) of a plot, and colour each data point based on the class it belongs to.

```
alt.Chart(bean).mark_point().encode(
    x='MajorAxisLength',
    y='MinorAxisLength',
    color=alt.Color('Class', scale=alt.Scale(scheme='category10')),
).interactive()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[58], line 1
----> 1 alt.Chart(bean).mark_point().encode(
      2     x='MajorAxisLength',
      3     y='MinorAxisLength',
      4     color=alt.Color('Class', scale=alt.Scale(scheme='category10')),
      5 ).interactive()

NameError: name 'alt' is not defined
```

We can clearly see that the beans form clusters based on length and width!

What if we wanted to change the order the colours were assigned to our categories? The default is alphabetical. We could do this to change to reverse alphabetical:

```
bean['Class'].unique()
```

```
['SEKER', 'BARBUNYA', 'SIRA', 'CALI', 'DERMASON', 'HOROZ', 'BOMBAY']
Categories (7, object): ['BARBUNYA', 'BOMBAY', 'CALI', 'DERMASON', 'HOROZ', 'SEKER', 'SIRA']
```

```
category_order = sorted(bean['Class'].unique(), reverse=True)
category_order
# Convert 'species' column to a categorical type with the specified order
bean['Class'] = pd.Categorical(bean['Class'], categories=category_order, ordered=True)
bean['Class'].cat.categories
```

```
Index(['SIRA', 'SEKER', 'HOROZ', 'DERMASON', 'CALI', 'BOMBAY', 'BARBUNYA'], dtype='object')
```

What will our plot look like now?

```
alt.Chart(bean).mark_point().encode(
    x='MajorAxisLength',
    y='MinorAxisLength',
    color=alt.Color('Class', scale=alt.Scale(scheme='category10'))
)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[61], line 1
----> 1 alt.Chart(bean).mark_point().encode(
      2     x='MajorAxisLength',
      3     y='MinorAxisLength',
      4     color=alt.Color('Class', scale=alt.Scale(scheme='category10'))
      5 )

NameError: name 'alt' is not defined
```

Let's load another dataset that contains data about book sales on Amazon by reading in `data/publishers.csv`. This data was sourced from CORGIS.

```
pub = pd.read_csv('data/publishers.csv', index_col = 0)
```

Each row contains information on a particular book (although for some reason the dataset omits the book's title and author!) Let's see how many unique entries are contained in the `'publisher.type'` column.

```
pub['publisher.type'].value_counts()
```

```
publisher.type
small/medium     226
big five         168
indie            124
single author     95
Name: count, dtype: int64
```

There are only four categories, and they have a natural ordering among them. We may want to sort our dataset based on the 'size' of the publisher– but our column is an `object` dtype and contains strings. Strings are sorted alphabetically!

```
pub = pub.sort_values(by = 'publisher.type', ascending = False)
print(pub.iloc[0]['publisher.type']) # Check the first row
print(pub.iloc[240]['publisher.type']) # Somewhere in the middle
print(pub.iloc[440]['publisher.type']) # Somewhere else in the middle
print(pub.iloc[-1]['publisher.type']) # Check the last row
```

```
small/medium
single author
indie
big five
```

That sort wasn't useful here. Pandas categories, in contrast to strings, can be given any order we choose.

```
pub['publisher.type'] = pub['publisher.type'].astype('category')
pub['publisher.type'] = (pub['publisher.type'].cat.reorder_categories(
                                ['single author', 'indie', 'small/medium', 'big five'], ordered=True)
                        )
```

```
pub = pub.sort_values('publisher.type', ascending = False)
```

```
print(pub.iloc[0]['publisher.type']) # Check the first row
print(pub.iloc[240]['publisher.type']) # Somewhere in the middle
print(pub.iloc[440]['publisher.type']) # Somewhere in the middle
print(pub.iloc[-1]['publisher.type']) # Check the last row
```

```
big five
small/medium
indie
single author
```

The Data Frame also contains a column called `'genre'`. That's another candidate for categorical data, so let's see what unique values it contains.

```
pub['genre'].value_counts()
```

```
genre
nonfiction          321
genre fiction       201
children             55
fiction              19
comics               14
foreign language      2
Comics                1
Name: count, dtype: int64
```

Oops, looks like two spellings (`'comics'` and `'Comics'`) are listed for the same genre! This is a common occurrence. In general, it is best to do your data wrangling first and only convert to the `category` dtype once your data is cleaned up.