Analyze data with pandas

This notebook has code to load a CSV of Townsville animal complaints into a pandas dataframe for some basic analysis.

It would be a good idea to bookmark the pandas documentation site for future reference. See also these reference notebooks for various pandas operations:

- Pandas - Importing data
- Pandas - Filtering columns and rows
- Pandas - Grouping data
- Pandas - Using the apply method
- Pandas - Joining data
- Pandas - Cleaning data

In [ ]:
```
# importing pandas AS an easier-to-type alias, pd, is the convention
import pandas as pd
```

Importing data

You can import many different kinds of data into a pandas from a variety of sources. In this example, we'll load data directly from a CSV hosted on the federal Australian open data portal.

Here's the pandas documentation for the `read_csv()` method.

In [ ]:
```
df = pd.read_csv('https://data.gov.au/data/dataset/5a005841-f4f2-4c52-82db-8cce70715d72/re
```

Get to know your data a little

Once you have a dataframe loaded with data, pandas has a number of handy methods to check it out:

- `.head()` : Shows you the first 5 records in the data frame (optionally, if you want to see a different number of records, you can pass in a number)
- `.tail()` : Same as head(), but it pull records from the end of the dataframe
- `.sample(n)` will give you a sample of n rows of the data -- just pass in a number
- `.info()` will give you a count of non-null values in each column -- useful for seeing if any columns have null values
- `.describe()` will compute summary stats for numeric columns
- `.columns` will list the column names
- `.dtypes` will list the data types of each column
- `.shape` will give you a pair of numbers: (number of rows, number of columns)

In [ ]:
```
df.head()
```

In [ ]:
```
df.tail()
```

In [ ]:
```
df.sample(10)
```

In [ ]:
```
df.info()
```

In [ ]:
```
df.describe()
```

In [ ]:
```
df.columns
```

In [ ]:
```
df.dtypes
```

In [ ]:
```
df.shape
```

In [ ]:
```
# the .shape attribute is a tuple -- number of rows, number of columns --
# which means you can access these items like a list
num_rows = df.shape[0]
num_cols = df.shape[1]
```

In [ ]:
```
print(num_rows)
```

Sort the data

To sort a data frame, use the `sort_values()` method. At a minimum, you need to tell it which column to sort on.

Sorting on the `Suburb` column:

In [ ]:
```
df.sort_values('Suburb')
```

... or to sort descending:

In [ ]:
```
df.sort_values('Suburb', ascending=False)
```

Filtering data

Let's look at two kinds of filtering: Selecting one or more columns to filter vertically, and filtering rows of data based on some criteria.

Selecting columns

Right now we're working with a `DataFrame` object, which has a two-dimensional tabular layout. Selecting one column will return a `Series` object. (Selecting multiple columns of data will return another `DataFrame` object.)

You can select a column of data using dot notation `.` or bracket notation: `[]`. If you want to select a single column of data and the column name doesn't have spaces, you can use a period ("dot notation").

You could also pass the name of the column as a string inside square brackets ("bracket notation"); if your column names have spaces (avoid this if you can), you *must* use bracket notation.

Therefore the following two code blocks are equivalent:

In [ ]:
```
df.Suburb
```

In [ ]:
```
df['Suburb']
```

... if we wanted to slice out the "Complaint Type" column, however, we must use bracket notation because the column label has a space in it:

In [ ]:
```
df['Complaint Type']
```

💥 Bonus math: Using `value_counts()` to compare groups

Often, the purpose of selecting a single column of data like this is to perform an integrity check or analyze the values in one column specifically.

For instance, you can use the `value_counts()` method on a column of data to produce a quick frequency chart of values, similar to building a pivot table with counts in a spreadsheet program, or using a `COUNT(*)` aggregate in SQL.

For example, perhaps we wanted to know the most frequent type of complaint:

In [ ]:
```
df['Complaint Type'].value_counts()
```

What percentage of the total does that represent? Let's make a few moves to find out:

- Save the results of the `value_counts()` operation as a variable, but while we're at it, use the `reset_index()` method to turn the Series into a DataFrame
- Rename the columns in the new dataframe
- Create a new column by doing a little math for each type of complaint -- dividing the count for each one into the total number of records in this data, which we stored earlier with the variable `num_rows`, and multiplying the result by 100 to get the percentage of total

In [ ]:
```
df_complaint_type = df['Complaint Type'].value_counts().reset_index()
```

In [ ]:
```
# rename the column headers
df_complaint_type.columns = ['complaint_type', 'complaint_count']
```

In [ ]:
```
df_complaint_type.head()
```

In [ ]:
```
# If the operations to create a new column are this straightforward, you
# can use the method of bracket-indexing the new column name and assigning
# to that column the results of the operation on the right-hand side
df_complaint_type['pct'] = (df_complaint_type['complaint_count'] / num_rows) * 100
```

In [ ]:
```
df_complaint_type
```

If the operation needed to create a new column is more complex than this kind of arithmetic, you'll probably need to *apply* a function -- here's a notebook with more information on how to do that.

Filtering rows

Maybe I'm only interested in complaints in Alice River. How would I target those specific rows? With a filter that looks like this:

In [ ]:
```
df_alice_river = df[df['Suburb'] == 'Alice River']
```

In [ ]:
```
df_alice_river.head()
```

In [ ]:
```
# verify what we ended up with
df_alice_river.Suburb.value_counts()
```

So the syntax for filtering is: Drop in the variable name of the dataframe, then open a set of flat brackets, then inside those brackets type the name of the dataframe variable *again*, then use dot or bracket accession to extract a column of data in a series and perform some sort of conditional comparison, then close the outer flat brackets.

This is ... kind of bananas! Not much more to say other than: This is the way the authors of this library chose to write the syntax. You'll find more detailed information and examples in this notebook -- as with other things, it comes with practice.

... and much more

For more example of grouping, joining and cleaning data, see the reference notebooks.