DSC 10 Reference Sheet

Below, `df` is a DataFrame, `ser` is a Series, `babypandas` has been imported as `bpd`, and `numpy` has been imported as `np`.

Building and Organizing DataFrames

Each function/method below creates a new dataframe.

`bpd.DataFrame()`
: Creates empty DataFrame.

`bpd.read_csv(path_to_file)`
: Creates a DataFrame by reading from a CSV file.

`df.assign(Name_of_Column=column_data)`
: Adds/replaces a column.

`df.drop(columns=column_name)`
: Drops a single column.

`df.drop(columns=[col_1_name, ..., col_k_name])`
: Drops every column in a list of column names.

`df.set_index(column_name)`
: Move the column to the index.

`df.reset_index()`
: Move the index to a column.

`df.sort_values(by=column_name)`
: Sort the entire DataFrame in ascending order by the values in a column.

`df.sort_values(by=column_name, ascending=False)`
: Sort the entire DataFrame in descending order.

`left.merge(right, left_on=left_column, right_on=right_column)`
: Perform a join between the tables `left` and `right`.

`left.merge(right, left_index=True, right_on=right_column)`
: Perform a join using `left`'s index instead of a column. Can also be done with `right_index=True`.

Series Methods

Series have the following methods; each returns a single number:

`.count()`, `.max()`, `.min()`, `.sum()`, `.mean()`, `.median()`

Applying

`df.get(column_name).apply(function_name)` applies a function to every entry in the column; returns a Series of the same size containing the results.

Plotting

`df.plot(kind=kind, x=col_x, y=col_y)`
: Draw a plot. `kind` may be `'scatter'`, `'line'`, `'bar'`, or `'barh'` (for a horizontal bar chart. If `x` is omitted, the index is used.

`df.get(column_name).plot(kind='hist', bins=n_bins)`
: Plot a histogram of the data in the given column. `n_bins` can be a number of bins, or a sequence specifying bin locations and widths.

Retrieving Information

`df.shape[0]` and `df.shape[1]`
: The number of rows and the number of columns, respectively.

`df.get(column_name)`
: Retrieve column. Returns a Series.

`df.get([col_1_name, ..., col_k_name])`
: Retrieve several columns. Returns a DataFrame.

`ser.loc[label]`
: Retrieve an element by the row label.

`ser.iloc[position]`
: Retrieve an element by its integer position.

`df.index[position]`
: Retrieve the element in the index by its integer position.

`df.take([position_1, ..., position_k])`
: Select several rows using by integer position.

`df[bool_arr]`
: Select rows using a Boolean array. Returns a DataFrame. See: Boolean Indexing.

Boolean Indexing

Select a subset of a DataFrame's rows by constructing a Boolean array `condition` with a likewise number of rows. The expression `df[condition]` results in a DataFrame containing only those rows whose corresponding element in `condition` is `True`. Boolean arrays are easily constructed by comparing an array/index/Series to a value using the comparison operators: `>`, `<`, `==`, `<=`, `>=`, `!=`.

`bool_arr_1 & bool_arr_2`
: Combine two Boolean arrays into one by "and"-ing them.

`df[df.get(column_name) > 42]`
: Retrieve all rows for which the given column is bigger than 42.

`df[(df.get(col_1) > 42) & (df.get(col_2) < 100]`
: Retrieve all rows for which the given column is between 42 and 100. Parenthesis are important!

`df[df.get(column_name).str.contains(pattern)]`
: Retrieve all rows for which the given column contains the string pattern.

`df[df.index > 2]`
: Retrieve all rows for which the index is greater than 2.

`df[df.index.str.contains(pattern)]`
: Retrieve all rows for which the index contains the string pattern.

Grouping

Use `df.groupby(column_name)`, followed by one of these aggregation functions:

`.mean()`, `.median()`, `.count()`, `.max()`, `.min()`, `.sum()`

The result is a new table whose index contains the group names. Only those columns whose data type permits the selected aggregation method are kept – for instance, '.sum()' will drop columns containing strings.

`df.groupby([col_1_name, ..., col_k_name])` creates subgroups, first grouping by `col_1_name`, then, within each group, grouping by `col_2_name`, and so on. It is recommended that you follow a subgrouping `.groupby()` with `.reset_index()`.

NumPy

`arr[index]`
: Get the element at position `index` in the array `arr`. The first element is `arr[0]`.

`np.append(arr, value)`
: A copy of `arr` with `value` appended to the end.

`np.count_nonzero(arr)`
: Returns the number of non-zero entries in an array. `True` counts as one, `False` counts as zero.

`np.random.choice(arr, n, replace=True)`
: Draw `n` things from `arr` with replacement. If `replace=False` is used, the sampling is done without replacement. If `n` is omitted, one thing is drawn.

`np.arange(start, stop, step)`
: An array of numbers starting with `start`, increasing/decreasing in increments of `step`, and stopping before (excluding) `stop`. If `start` or `step` are omitted, the default values are 0 and 1, respectively.

`for`-loops

```
for <loop variable> in <sequence>:
  <loop body>
```

Performs the loop body for every element of the sequence. For example, to print the squares of the first 10 numbers:

```
for i in np.arange(10):
  print(i**2)
```

`if`-statements and Booleans

Conditionally execute code. The `elif` and `else` blocks are optional.

```
if <condition>:
  <if body>
elif <second_condition>:
  <elif body>
elif <third_condition>:
  <elif body>
...
else:
  <else body>
```

A Boolean variable is either `True` or `False`. Booleans can be combined with `and` and `or`.

Comparisons result in Boolean variables. Comparisons can be performed with the operators: `==` (equality), `!=` (inequality), `<`, `>`, `<=`, `>=`.

Functions

```
def function_name(argument_1, ..., argument_k):
  <function body>
```

Give a name to a piece of code. For example, to square a number:

```
def square_a_number(number):
  return number**2
```