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

## Writing Functions

A custom functions is written as follows:

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

For example, this function squares a number:

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

## 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()`, `.std()`

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.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.

`np.percentile(array, p)`
: Compute the $p$th percentile of the numbers in `array`. Example: `np.percentile(array, 95)` computes the 95th percentile.

`np.append(array, x)`
: Return a new array which has x appended to the end of `array`.

## `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), `<`, `>`, `<=`, `>=`.

## Statistics and Hypothesis Testing

A **sample** is a subset of a **population**. A **statistic** is a number computed using the sample. The field of statistics is about using a sample to say something about the population.

A **experiment** is a process whose outcome is random; for example, flipping a 100 coins. An **observed statistic** is a statistic computed from the outcome of an experiment; for example, the number of Heads observed. A **model** is a set of assumptions about how the data was generated. For example: the result of a coin flip is equally-likely to be Heads or Tails. **Hypothesis testing** is the process of testing a model for validity.

To perform a **hypothesis test**, we first establish a **null hypothesis**: this is a precise assumption about how the data was generated. For instance: the coin is fair. The **alternative hypothesis** is the opposite of the null hypothesis. In our example, the alternative hypothesis is that the coin is not fair.

To **test** the hypothesis, we compute the probability of seeing an outcome at least as extreme as the observed statistic under the assumptions of the null hypothesis; this is called the **p-value**. In practice, we do this by simulating a bunch of outcomes using the null hypothesis and counting how many times the outcome is more extreme than what was originally observed.

## `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)
```

## Random Sampling

`np.random.choice(array)`
: Return an element from the array at random.

`np.random.choice(array, n)`
: Return $n$ elements from the array at random, with replacement.

`np.random.choice(array, n, replace=False)`
: Return $n$ elements from the array at random, without replacement.

`np.random.multinomial(n, [p_1, ..., p_k])`
: Return an array of length $n$ in which each element contains the number of occurrences of an event, where the probability of the $i$th event is `p_i`. For instance, if an M&M is red with probability 0.2, green with probability 0.5, and brown with probability 0.3, then a random selection of 100 M&Ms is given by:

  `np.random.multinomial(100, [0.2, 0.5, 0.3])`.

  The result is a random count of each color. For instance, it might be `[22, 45, 33]`.

`np.random.permutation(array_or_series)`
: Return a new array by randomly shuffling the input.

`table.sample(n)`
: Return a new table by randomly sampling $n$ rows from `table`, without replacement.

`table.sample(n, replace=True)`
: Return a new table by randomly sampling $n$ rows from `table`, with replacement.

## The Bootstrap

When we compute a statistic from a sample, such as the median salary of San Diego city employees, we should remember that the sample is random. Therefore, the result could have been different. The Bootstrap allows us to answer: "how different could it have been?" by giving us an approximation of the distribution of the sample statistic.
Suppose `salaries` contains a column called "Salary" containing the salary of each employee in a sample. The observed median salary is:

```
observed = salaries.get('Salary').median()
```

To make a 95% confidence interval for the median salary, we run the bootstrap by re-sampling our data with replacement, over and over again:

```
boot_medians = np.array([])
for i in np.arange(5000):
  # 1. re-sample the data
  boot_salaries = salaries.sample(
    salaries.shape[0], replace=True
  )

  # 2. compute the statistic on the bootstrap sample
  boot_median = boot_salaries.get('Salary').median()

  # 3. save the result
  boot_medians = np.append(boot_medians, boot_median)
```

The endpoints of the 95% confidence interval are:

```
left = np.percentile(boot_medians, 2.5)
right = np.percentile(boot_medians, 97.5)
```

## Permutation Testing

We use a permutation test to determine if two groups were drawn from the same population. For instance, suppose we give two versions of the exam, A and B. We gather the results in a table `scores` with two columns: "Version" and "Score", containing the version and score for each exam taken. We notice that the score on version A was typically higher. We can test whether A was indeed significantly harder than B by using a permutation test.

First, we decide on a test statistic, such as difference between the mean score in each group. If we care only about whether the exams were different, we use an **unsigned** statistic, such as the absolute difference. If we care whether A was harder, we use a **signed** statistic, such as the signed difference:

```
def difference_in_means(scores):
  means = scores.groupby('Version').mean()
  return (
    means.get('Score').loc['A']
    -
    means.get('Score').loc['B']
  )
```

Then, to run a permutation test with 5000 iterations:

```
statistics = np.array([])
for i in np.arange(5000):
  # 1. shuffle the versions
  shuffled = scores.assign(
    Version=np.random.permutation(
      scores.get("Version")
    )
  )

  # 2. compute the test statistic
  statistic = difference_in_means(shuffled)

  # 3. save the result
  statistics = np.append(statistics, statistic)
```

We then compute the observed test statistic, which in this case is the difference in mean between the two versions:

```
observed = difference_in_means(scores)
```

To compute the $p$-value, we use `np.count_nonzero`:

```
p_value = np.count_nonzero(statistics > actual) / 5000
```

## Standard Units, Correlation, Regression

Given a collection of numbers $x_1, \ldots, x_n$, the representation of $x_i$ in **standard units** is

$$z_i = \frac{x_i - \text{mean}}{\text{STD}}$$

If `x` is an array of numbers, we can convert each number to standard units with the code: `x_su = (x - np.mean(x))/np.std(x)`

If `x` and `y` are two arrays of the same size, and `x_su` and `y_su` are the arrays after converting to standard units, then the **correlation coefficient** is `r = (x_su * y_su).mean()`

In standard units, the slope $m$ of the regression line is $r$ (the correlation coefficient) and the intercept is 0.

In original units, the slope $m$ and intercept $b$ of the regression line are:

$$m = r \cdot \frac{\text{SD of } y}{\text{SD of } x}$$

$$b = (\text{mean of } y) - m \cdot (\text{mean of } x)$$

## Spread of a Distribution

The **standard deviation** of a collection of numbers $x_1, \ldots, x_n$ is $\sqrt{\frac{1}{n} \cdot (x_1 + x_2 + \ldots + x_n)}$.

Chebyshev's Theorem: at most $1 - 1/z^2$ of the data is within $z$ STDs of the mean.

| Range | All Distributions | Normal Distribution |
|---|---|---|
| mean $\pm 1$ STD | at least 0% | about 68% |
| mean $\pm 2$ STDs | at least 75% | about 95% |
| mean $\pm 3$ STDs | at least 88.8% | about 99.73% |

## The Standard Normal Curve

The standard normal curve has mean 0, standard deviation 1.

To compute the area under the standard normal curve from $-\infty$ to $z$, use

```
scipy.stats.norm.cdf(z)
```

$z$ must be in standard units.

To compute the area under the standard normal curve between $z_2$ and $z_1$, where $z_1 < z_2$:

```
scipy.stats.norm.cdf(z_2) - scipy.stats.norm.cdf(z_1)
```

## The Central Limit Theorem

The sample is random. So the sample mean is too.

Since it is random, the sample mean has a distribution.

We could get this distribution using the bootstrap, but there is a better way:

The CLT: "the distribution of the sample mean is approximately normal, no matter the distribution of the population."

The mean of the distribution is the population mean.

The STD of the distribution is

$$\frac{\text{population STD}}{\sqrt{\text{sample size}}}$$

The population mean and the population STD can be replaced with the sample mean and sample STD without much loss in accuracy.

To make a 95% confidence interval for the sample mean:

$$\left[\text{sample mean} - 2 \times \frac{\text{sample std}}{\sqrt{\text{sample size}}}, \quad \text{sample mean} + 2 \times \frac{\text{sample std}}{\sqrt{\text{sample size}}}\right]$$

For all of this to work, the sample has to be randomly drawn with replacement.