Pandas Cheat Sheet

by Justin1209 (Justin1209) via cheatography.com/101982/cs/21202/

Import the Pandas Module

```
import pandas as pd
```

Create a DataFrame

```
# Method 1
df1 = pd.DataFrame({
    'name': ['John Smith', 'Jane Doe'],
    'address': ['13 Main St.', '46 Maple Ave.'],
    'age': [34, 28]
})
# Method 2
df2 = pd.DataFrame([
    ['John Smith', '123 Main St.', 34],
    ['Jane Doe', '456 Maple Ave.', 28],
    ['Joe Schmo', '9 Broadway', 51]
    ],
    columns=['name', 'address', 'age'])
```

Loading and Saving CSVs

```
# Load a CSV File in to a DataFrame
df = pd.read_csv('my-csv-file.csv')
# Saving DataFrame to a CSV File
df.to_csv('new-csv-file.csv')
# Load DataFrame in Chunks (For large Datasets)
# Initialize reader object: urb_pop_reader
urb_pop_reader = pd.read_csv('ind_pop_data.csv', chunksize=1000)
```

Loading and Saving CSVs (cont)

```
# Get the first DataFrame chunk: df_urb_pop
df_urb_pop = next(urb_pop_reader)
```

Inspect a DataFrame

df.head(5)	First 5 rows
df.info()	Statistics of columns *(row count, null values, datatype)*

Reshape (for Scikit)

```
nums = np.array(range(1, 11))
-> [ 1 2 3 4 5 6 7 8 9 10]
nums = nums.reshape(-1, 1)
-> [ [1],
[2],
[3],
[4],
[5],
[6],
[7],
[8],
[9],
[10]]
```

You can think of **reshape()** as rotating this array. Rather than one big row of numbers, nums is now a big column of numbers - there's one number in each row.

Converting Datatypes

 # Convert argument to numeric type

```
pandas.to_numeric(arg, errors="raise")
```

errors:

"raise" -> raise an exception

"coerce" -> invalid parsing will be set as NaN

DataFrame for Select Columns / Rows

```
df = pd.DataFrame([
  ['January', 100, 100, 23, 100],
  ['February', 51, 45, 145, 45],
  ['March', 81, 96, 65, 96],
  ['April', 80, 80, 54, 180],
  ['May', 51, 54, 54, 154],
  ['June', 112, 109, 79, 129]],
  columns=['month', 'east', 'north', 'south', 'west']
)
```

Select Columns

```
# Select one Column
clinic_north = df.north
--> Reshape values for Scikit learn: clinic_north.values.reshape(-1, 1)
# Select multiple Columns
clinic_north_south = df[['north', 'south']]
```

Make sure that you have a *double set of brackets* **[[ ]]**, or this command won't work!

Select Rows

```
# Select one Row
march = df.iloc[2]
# Select multiple Rows
jan_feb_march = df.iloc[:3]
feb_march_april = df.iloc[1:4]
may_june = df.iloc[-2:]
# Select Rows with Logic
january = df[df.month ==
'January']
-> <, >, <=, >=, !=, ==
march_april = df[(df.month ==
'March') | (df.month ==
'April')]
-> &, |
january_february_march =
df[df.month.isin(['January',
'February', 'March'])]
-> column_name.isin([" ", " "])
```

Selecting a Subset of a Dataframe often results in **non-consecutive indices.**

Using **.reset_index()** will create a *new DataFrame* move the old indices into a new colum called *index*.

Use **.reset_index(drop=True)** if you dont need the *index* column.
Use **.reset_index(inplace=True)** to prevent a *new DataFrame* from brein created.

Adding a Column

```
df = pd.DataFrame([
  [1, '3 inch screw', 0.5,
0.75],
  [2, '2 inch nail', 0.10,
0.25],
  [3, 'hammer', 3.00, 5.50],
  [4, 'screwdriver', 2.50, 3.00]
],
  columns=['Product ID', 'Descr-
iption', 'Cost to Manufacture',
'Price']
)
# Add a Column with specified
row-values
df['Sold in Bulk?'] = ['Yes',
'Yes', 'No', 'No']
# Add a Column with same value
in every row
df['Is taxed?'] = 'Yes'
# Add a Column with calculation
df['Revenue'] = df['Price'] -
df['Cost to Manufacture']
```

Performing Column Operation

```
df = pd.DataFrame([
  ['JOHN SMITH', 'john.smith@-
gmail.com'],
  ['Jane Doe', 'jdoe@yahoo.c-
om'],
  ['joe schmo', 'joeschmo@hotma-
il.com']
],
columns=['Name', 'Email'])
# Changing a column with an
Operation
df['Name'] = df.Name.apply(lo-
wer)
```

Performing Column Operation (cont)

```
-> lower, upper
# Perform a lambda Operation on
a Column
get_last_name = lambda x:
x.split(" ")[-1]
df['last_name'] = df.Name.apply-
(get_last_name)
```

Performing a Operation on Multiple Columns

```
df = pd.DataFrame([
  ["Apple", 1.00, "No"],
  ["Milk", 4.20, "No"],
  ["Paper Towels", 5.00, "-
Yes"],
  ["Light Bulbs", 3.75, "Yes"],
],
  columns=["Item", "Price", "Is
taxed?"])
# Lambda Function
df['Price with Tax'] = df.app-
ly(lambda row:
     row['Price'] * 1.075
     if row['Is taxed?'] ==
'Yes'
     else row['Price'],
     axis=1
)
```

We apply a **lambda to rows**, as opposed to columns, when we want to perform functi-onality that needs to access more than one column at a time.

Rename Columns

```
# Method 1
df.columns = ['NewName_1', 'NewName_2', 'NewName_3', '...']
# Method 2
df.rename(columns={
    'OldName_1': 'NewName_1',
    'OldName_2': 'NewName_2'
}, inplace=True)
```

Using **inplace=True** lets us edit the original DataFrame.

Series vs. Dataframes

 # Dataframe and Series
print(type(clinic_north)):
 # <class 'pandas.core.series.Series'>
print(type(df)):
 # <class 'pandas.core.frame.DataFrame'>
print(type(clinic_north_south))
 # <class 'pandas.core.frame.DataFrame'>

In Pandas
- a **series is a one-dimensional object** that contains any type of data.

- a **dataframe is a two-dimensional object** that can hold multiple columns of different types of data.

A *single column of a dataframe is a series*, and a **dataframe is a container of two or more series objects**.

Column Statistics

Mean = Average	df.*column*.**mean()**
Median	df.*column*.**median()**
Minimal Value	df.*column*.**min()**
Maximum Value	df.*column*.**max()**
Number of Values	df.*column*.**count()**
Unique Values	df.*column*.**nunique()**
Standard Deviation	df.*column*.**std()**
List of Unique Values	df.*column*.**unique()**

Dont't forget reset_index() at the end of a **groupby** operation

Calculating Aggregate Functions

```
# Group By
grouped = df.groupby(['col1', 'col2']).col3
.measurement().reset_index()
# -> group by column1 and column2, calculate values of column3
# Percentile
high_earners = df.groupby('category').wage
    .apply(lambda x: np.percentile(x, 75))
    .reset_index()
# np.percentile can calculate any percentile over an array of values
```

Don't forget reset.index()

Pivot Tables

```
orders =
pd.read_csv('orders.csv')
shoe_counts = orders.
groupby(['shoe_type', 'shoe_color']).
id.count().reset_index()
shoe_counts_pivot = shoe_counts.pivot(
index = 'shoe_type',
columns = 'shoe_color',
values = 'id').reset_index()
```

We have to build a temporary table where we group by the columns we want to include in the pivot table

Merge (Same Column Name)

```
sales = pd.read_csv('sales.csv')
targets = pd.read_csv('targets.csv')
men_women = pd.read_csv('men_women_sales.csv')
# Method 1
sales_targets = pd.merge(sales, targets, how=" ")
# how: "inner"(default), "outer", "left", "right"
#Method 2 (Method Chaining)
all_data = sales.merge(targets).merge(men_women)
```

Inner Merge (Different Column Name)

```
orders = pd.read_csv('orders.csv')
products = pd.read_csv('products.csv')
# Method 1: Rename Columns
orders_products = pd.merge(orders, products.rename(columns={'id':'product_id'}), how=" ").reset_index()
# how: "inner"(default), "outer", "left", "right"
# Method 2:
orders_products = pd.merge(orders, products,
                  left_on="product_id",
                  right_on="id",
                  suffixes=["_orders","_products"])
```

Method 2:
If we use this syntax, we'll end up with **two columns called id.**
Pandas won't let you have two columns with the same name, so it will change them to id_x and id_y.
We can help make them more useful by using the keyword **suffixes.**

Concatenate

```
bakery = pd.read_csv('bakery.csv')
ice_cream = pd.read_csv('ice_cream.csv')
menu = pd.concat([bakery, ice_cream])
```

Melt

```
pandas.melt(DataFrame, id_vars, value_vars, var_name, value_name='value')
```

id_vars: Column(s) to use as identifier variables.
value_vars: Column(s) to unpivot. If not specified, uses all columns that are not set as id_vars.
var_name: Name to use for the 'variable' column.
value_name: Name to use for the 'value' column.

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

Assert Statements

```
# Test if country is of type object
assert gapminder.country.dtypes == np.object
# Test if year is of type int64
assert gapminder.year.dtypes == np.int64
# Test if life_expectancy is of type float64
assert gapminder.life_expectancy.dtypes == np.float64
# Assert that country does not contain any missing values
assert pd.notnull(gapminder.country).all()
# Assert that year does not contain any missing values
assert pd.notnull(gapminder.year).all()
```