Cheatography

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('in_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_fe b_march = df.ilo c[:3]
feb_ma rch _april = df.ilo -
c[1:4]
may_june = df.ilo c[-2:]
# Select Rows with Logic
january = df[df.m onth==
'January']
-> <, >, <=, >=, !=, ==
march_ april = df[(df.month ==
'March') | (df.month ==
'April')]
-> &, |
januar y_f ebr uar y_march =
df[df.month.isin(['Jan uary',
'Febru ary', 'March'])]
-> column _na me.i si n([ " ", "
"])
```

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, 'screw dri ver', 2.50,
3.00]
],
     col umn s=[ 'Pr oduct ID',
'Descr ipt ion', 'Cost to
Manufa cture', '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 calcul ation
df['Re venue'] = df['Pr ice'] -
df['Cost to Manufa cture']
```

Performing Column Operation

```
df = pd.DataFrame([
    ['JOHN SMITH', 'john.s mi -
th@ gma il.c om'],
    ['Jane Doe', 'jdoe@ yah -
oo.c om'],
    ['joe schmo', 'joesc hmo -
@ho tma il.c om']
],
column s=[ 'Name', 'Email'])
# Changing a column with an
Operation
df['Name'] = df.Name.apply( -
lower)
```

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([
    ["Ap ple ", 1.00, " No"],
    ["Mi lk", 4.20, " No"],
    ["Paper Towels ", 5.00, " -
Yes "],
    ["Light Bulbs", 3.75, " -
Yes "],
],
    col umn s=[ " Ite m", " -
Pri ce", "Is taxed? "])
# Lambda Function
df['Price with Tax'] = df.app -
ly( lambda row:
          row ['P rice'] * 1.075
         if row['Is taxed?'] ==
'Yes'
         else row['P rice'],
     axis=1
)
```

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

Cheatography

Pandas Cheat Sheet

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

Rename Columns

```
# Method 1
df.columns = ['NewN ame_1',
'NewNa me_2, 'NewNa me_3',
'...']
# Method 2
df.rename(columns={
        'Ol dNa me_1': 'NewNa -
me_1',
        'Ol dNa me_2': 'NewNa -
me_2'
}, inplac e=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.groupb y([ 'col1',
'col2']).col3
.measur ement().reset_ index()
# -> group by column1 and
column2, calculate values of
column3
# Percentile
high_e arners = df.gro upb y(' -
cat ego ry' ).wage
     apply( lambdax: np.per -
cen tile(x, 75))
     reset_ index()
# np.per centile can calculate
any percentile over an array of
values
```

Don't forget reset.index()

Pivot Tables

```
orders =
pd.read_csv('orders.csv')
shoe_c ounts = orders.
groupb y([ 'sh oe_ type',
'shoe_ col or']).
id.cou nt( ).r ese t_i ndex()
shoe_c oun ts_ pivot = shoe_c -
oun ts.p ivot(
index = 'shoe_ type',
columns = 'shoe_ color',
values = 'id').r es et_ 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.rea d_c sv( 'ta -
rge ts.c sv')
men_women = pd.rea d_c sv( 'me -
n_w ome n_s ale s.csv')
# Method 1
sales_ targets = pdmerge(sales,
targets, how=" ")
# how: " inn er"( def ault), " -
out er", " lef t", " rig ht"
#Method 2 (Method Chaining)
all_data =
sales.merge(targets).merge(men_w
omen)
```

Inner Merge (Different Column Name)

```
orders =
pd.read_csv('orders.csv')
products = pd.rea d_c sv( 'pr -
odu cts.csv')
# Method 1: Rename Columns
orders _pr oducts =
pd.merge(orders,
products.rename(colum ns= {'i -
d': 'pr odu ct_ id'}), how=" ")
.reset _in dex()
# how: " inn er"( def ault), " -
out er", " lef t", " rig ht"
# Method 2:
orders _pr oducts =
pd.merge(orders, products,
                left_on="pr -
odu ct_ id",
                right_on="id ",
                suffixes=["_ -
ord ers " ,"_p rod uct s"])
```

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.rea d_c sv( 'ic -
e_c rea m.csv')
menu = pd.concat([bakery,
ice_cr eam])
```

Melt

```
pandas.me lt( Dat aFrame, id_vars, value_ vars, var_name, value_ nam e=
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 gapmin der.co unt ry.d -
types == np.object
# Test if year is of type int64
assert gapmin der.ye ar.d types
== np.int64
# Test if life_e xpe ctancy is
of type float64
assert gapmin der.li fe_ exp -
ect anc y.d types == np.float64
# Assert that country does not
contain any missing values
assert pd.not nul l(g apm ind -
er.c ou ntr y).a ll()
# Assert that year does not
contain any missing values
assert pd.not nul l(g apm ind -
er.y ea r).a ll()
```