Pandas Reference Sheet

POWERED BY THE SCIENTISTS AT THE DATA INCUBATOR

PRAGMATIC INSTITUTE

Loading/exporting a data set

path_to_file: string indicating the path to the file, e.g., ‘data/results.csv’

`df = pd.read_csv(path_to_file)`—read a CSV file

`df = pd.read_excel(path_to_file)`—read an Excel file

`df = pd.read_html(path_to_file)`—parses HTML to find all tables

`df.to_csv(path_to_file)`—creates CSV of the data frame

Examining the data

`df.head(n)`—returns first n rows

`df.tail(n)`—returns last n rows

`df.describe()`—returns summary statistics for each numerical column

`df[‘State’].unique()`—returns unique values for the column

`df.columns`—returns column names

`df.shape`—returns the number of rows and columns

Selecting and filtering

SELECTING COLUMNS

`df[‘State’]`—selects ‘State’ column

`df[[‘State’, ‘Population’]]`—selects ‘State’ and ‘Population’ column

SELECTING BY LABEL

`df.loc[‘a’]`—selects row by index label

`df.loc[‘a’, ‘State’]`—selects single value of row ‘a’ and column ‘State’

SELECTING BY POSITION

`df.iloc[0]`—selects rows in position 0

`df.iloc[0, 0]`—selects single value by position at row 0 and column 0

FILTERING

`df[df[‘Population’] > 20000000]]`—filter out rows not meeting the condition

`df.query(“Population > 20000000”)`—filter out rows not meeting the condition

	State	Capital	Population
a	Texas	Austin	28700000
b	New York	Albany	19540000
c	Washington	Olympia	7536000

Example data frame

Statistical operations

can be applied to both data frames and series/column

`df[‘Population’].sum()`—sum of all values of a column

`df.sum()`—sum for all numerical columns

`df.mean()`—mean

`df.std()`—standard deviation

`df.min()`— minimum value

`df.count()`—count of values, excludes missing values

`df.max()`—maximum value

`df[‘Population’].apply(func)`—apply func to each value of column

Data cleaning and modifications

`df[‘State’].isnull()`—returns True/False for rows with missing values

`df.dropna(axis=0)`—drop rows containing missing values

`df.dropna(axis=1)`—drop columns containing missing values

`df.fillna(0)`—fill in missing values, here filled with 0

`df.sort_values(‘Population’, ascending=True)`—sort rows by a column’s values

`df.set_index(‘State’)`—changes index to a specified column

`df.reset_index()`—makes the current index a column

`df.rename(columns={‘Population’=’Pop.’})`—renames columns

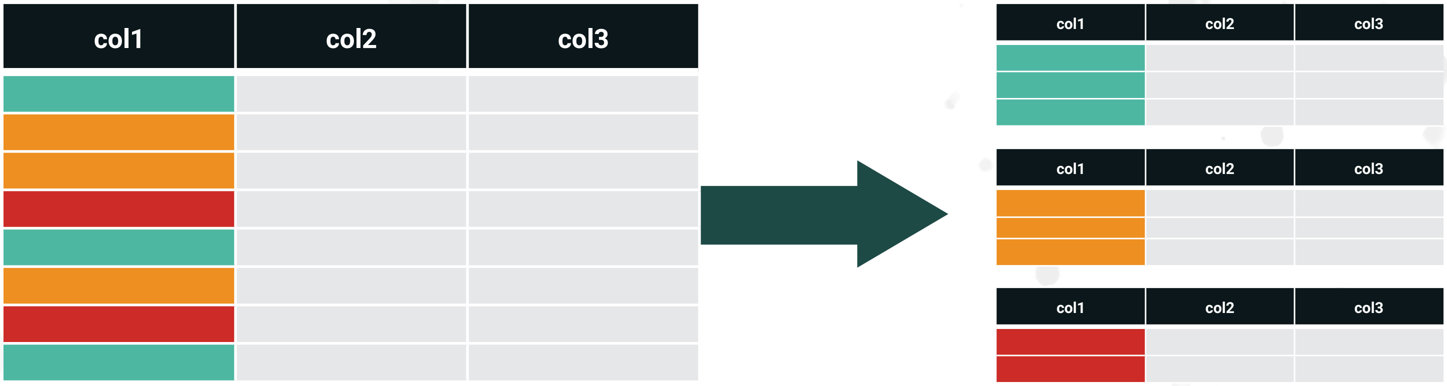
Grouping and aggregation

`grouped = df.groupby(by=‘col1’)`—create grouped by object

`grouped[‘col2’].mean()`—mean value of ‘col2’ for each group

`grouped.agg({‘col2’: np.mean, ‘col3’: [np.mean, np.std]})`—apply different functions to different columns

`grouped.apply(func)`—apply func to each group



Merging data frames

There are several ways to merge two data frames, depending on the value of `method`. The resulting indices are integers starting with zero.

`df1.merge(df2, how=method, on=‘State’)`

	State	Capital	Population
a	Texas	Austin	28700000
b	New York	Albany	19540000
c	Washington	Olympia	7536000

Data frame `df1`

+

	State	Highest Point
x	Washington	Mount Rainier
y	New York	Mount Marcy
z	Nebraska	Panorama Point

Data frame `df2`

	State	Capital	Population	Highest Point
0	Texas	Austin	28700000	NaN
1	New York	Albany	19540000	Mount Marcy
2	Washington	Olympia	7536000	Mount Rainier

`how=‘left’`

	State	Capital	Population	Highest Point
0	New York	Albany	19540000	Mount Marcy
1	Washington	Olympia	7536000	Mount Rainier

`how=‘inner’`

	State	Capital	Population	Highest Point
0	New York	Albany	19540000	Mount Marcy
1	Washington	Olympia	7536000	Mount Rainier
2	Nebraska	NaN	NaN	Panorama Point

`how=‘right’`

	State	Capital	Population	Highest Point
0	Texas	Austin	28700000	NaN
1	New York	Albany	19540000	Mount Marcy
2	Washington	Olympia	7536000	Mount Rainier
3	Nebraska	NaN	NaN	Panorama Point

`how=‘outer’`

scikit learn Reference Sheet

PRAGMATIC
INSTITUTE

The data

Your data needs to be contained in a two-dimensional feature matrix and, in the case of supervised learning, a one-dimensional label vector. The data has to be numeric (NumPy array, SciPy sparse matrix, pandas DataFrame).

Transformers: preprocessing the data

EXAMPLE

`ex_transf = ExampleTransformer()`—creates a new instance

`ex_transf.fit(X_train)`—fits transformer on training data

`transf_X = ex_transf.transform(X_train)`—transforms training data

`transf_X_test = ex_transf.transform(X_test)`—transforms test data

STANDARDIZE FEATURES (ZERO MEAN, UNIT VARIANCE)

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

SCALE EACH FEATURE BY ITS MAX ABS VALUE

```
from sklearn.preprocessing import MaxAbsScaler
max_scaler = MaxAbsScaler()
```

GENERATE POLYNOMIAL FEATURES

```
from sklearn.preprocessing import PolynomialFeatures
poly_transform = PolynomialFeatures(degree=n)
```

ONE-HOT ENCODE CATEGORICAL FEATURES

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
```

PRINCIPAL COMPONENT ANALYSIS

```
from sklearn.decomposition import PCA
pca = PCA(n_components=n)
```

Splitting into training data and test data

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test  = train_test_split(X, y)
```

Predictors: supervised learning

EXAMPLE

`ex_predictor = ExamplePredictor()`—creates a new instance

`ex_predictor.fit(X_train, y_train)`—fits model on training data

`y_pred = ex_predictor.predict(X_train)`—predicts on training data

`y_pred_probs = ex_predictor.predict_proba(X_train)`—classifiers only, predicts class probabilities on training data

LINEAR REGRESSION

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
```

DECISION TREE REGRESSION MODEL

```
from sklearn.tree import DecisionTreeRegressor
tree = DecisionTreeRegressor(max_depth=n)
```

RANDOM FOREST REGRESSION MODEL

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor()
```

LOGISTIC REGRESSION

```
from sklearn.linear_model import LogisticRegression
logr = LogisticRegression()
```

RANDOM FOREST CLASSIFICATION MODEL

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
```

Predictors: unsupervised learning

EXAMPLE

`ex_predictor = ExamplePredictor()`—creates a new instance

`ex_predictor.fit(X_train)`—fits model on training data

`y_pred = ex_predictor.predict(X_train)`—predicts on training data

K-MEANS CLUSTERING

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=n)
```

Evaluating model performance

```
from sklearn import metrics
```

REGRESSION METRICS

`metrics.mean_absolute_error(y_true, y_pred)`—Mean absolute error

`metrics.mean_squared_error(y_true, y_pred)`—Mean squared error

`metrics.r2_score(y_true, y_pred)`—R^2 score

CLASSIFICATION METRICS

`metrics.accuracy_score(y_true, y_pred)`—Accuracy score

`metrics.precision_score(y_true, y_pred)`—Precision score

`metrics.recall_score(y_true, y_pred)`—Recall score

`metrics.classification_report(y_true, y_pred)`—Classification report

`metrics.roc_auc_score(y_true, y_pred_probs)`—ROC AUC score

`metrics.log_loss(y_true, y_pred_probs)`—Cross-entropy loss

CLUSTERING METRICS

`metrics.silhouette_score(X_train, y_pred)`—Silhouette score

CROSS-VALIDATION

```
from sklearn.model_selection import cross_val_score
cross_val_score(lr, X_train, y_train, cv=5)
```

Pipeline

EXAMPLE

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([(‘feature scaling’, StandardScaler()),
                 (‘linear regression’, LinearRegression())])
```

`pipe.fit(X_train, y_train)`—fits model on training data

`y_pred = pipe.predict(X_train)`—predicts on training data

`y_pred_test = pipe.predict(X_test)`—predicts on test data

```
scaler = pipe.named_steps[‘feature scaling’]
lr = pipe.named_steps[‘linear regression’]
```

Feature union

EXAMPLE

```
from sklearn.pipeline import FeatureUnion
union = FeatureUnion([(‘transf_1’, ExampleTransformer1()),
                      (‘transf_2’, ExampleTransformer2())])
```

`union.fit(X_train)`—fits on training data

`X_transf = union.transform(X_train)`—transforms training data

Transforming only some features/columns

EXAMPLE

```
from sklearn.compose import ColumnTransformer
example_transf = ColumnTransformer(
   [(transformer_name, transformer, columns_to_transform)])
example_transf.fit(X_train)
X_transf = example_transf.transform(X_train)
```

Optimizing hyperparameters

```
from sklearn.grid_search import GridSearchCV
grid = GridSearchCV(estimator=DecisionTreeRegressor(),
   param_grid={‘max_depth’: range(3, 10)})
grid.fit(X_train, y_train)
```

`print(grid.best_estimator_)`—estimator that was chosen by the search

`print(grid.best_params_)`—parameters that gave the best results

Python Syntax REFERENCE SHEET

POWERED BY THE SCIENTISTS AT THE DATA INCUBATOR

PRAGMATIC
INSTITUTE

SYNTAX

Creating variables

Variables can be created by:

`deg_C = 10.5 # This is a variable`

A variable name can consist of letters, numbers and the underscore character (_) but the variable name may not start with a number. Comments are created with a # and are ignored by the Python interpreter.

Common mathematical operations

2 + 3 - addition
1 – 4 - subtraction
2 * 3 - multiplication
4 / 3 - division
4 // 3 - floor division (round down)
2 ** 3 - raise to the power
a += 1 - compute **a + 1** and assign the result to a
a -= 1 - compute **a – 1** and assign the result to a

Common built-in functions

print(temp_data) - print/display the value of temp_data

len(temp_data) - returns the number of values of the iterable

sum(temp_data) - returns sum of the values of the iterable

min(temp_data) - returns the minimum value of the iterable

max(temp_data) - returns the maximum value of the iterable

sorted(temp_data) - returns a list of the sorted values of **temp_data**

range(start, end, step) - returns an iterable from start to end (exclusive) using a step size of **step** (defaults to 1)

Functions

Functions are a great way to group related lines of code into a single unit that can be called upon. Here, we define a function with two positional arguments a and b and one keyword argument **multiplier** with a default value of 1.

```
def subtract(a, b, multiplier=1):

    “””
    Subtract two numbers and scale the result.
    ”””
    diff = multiplier * (a – b)

    return diff
```

Now, we call the function.

```
In  [1]: subtract(1, 2, multiplier=2)
Out [1]: –2
```

Boolean logic

These operations will return either **True** or **False**,depending on the value of the two variables. They are often used in conjunction with **if**/**elif** statements.

a < b - is **a** less than **b**
a > b - is **a** greater than **b**
a <= b - is **a** less than or equal to to **b**
a >= b - is **a** greater than or equal to **b**
a == b - do **a** and **b** have the same value
a != b - do **a** and **b** not have the same value
a is b - is **a** the same object as **b**

Loops

Loops are a way to repeatedly execute a block of code. There are two types of loops: for and while loops. For loops are used to loop through every value of an iterable, like a list or tuple. While loops are used to continually execute a block of code while a provided condition is still true.

```
for temp in temp_data:
    print(temp)

count = 0
while count < 10:
    print(count)
    count += 1
```

if/elif/else blocks

`if`/`elif`/`else` blocks let us control the behavior of our program based on conditions. For example, what value to assign to a variable based on the value of another variable. At a minimum, you need one condition to test, using `if`. Multiple conditions can be tested using multiple `elif` statements. The code in the `else` block, which is optional, is run when none of the tested conditions are met.

```
if amount < 5:
    rate = 0.1
elif amount <= 5 and amount < 10:
    rate = 0.2
else:
    rate = 0.25
```

DATA STRUCTURES . . . .

Strings

Strings are a sequence of characters and are great when wanting to represent text. They're created using either single or double quotes. They can be indexed but strings are immutable. Strings are iterables, iterating over each of the characters.

```
sentence = ‘The quick brown fox jumped over the lazy dog.’
```

Common operations with strings and usage:

`sentence.lower()` - returns new string with all characters in lowercase

`sentence.upper()` - returns a new string with all characters in uppercase

`sentence.startswith(‘The’)` - returns `True` or `False` if string starts with `‘The’`

`sentence.endswith(‘?’)` - returns `True` or `False` if string ends with `‘?’`

`sentence.split()` - returns a list resulting from splitting the string by a provided separator, defaults to splitting by whitespace if no argument is passed.

`sentence.strip()` - returns a new string with leading and trailing whitespace removed

`‘fox’ in sentence` - returns `True` if `‘fox’` is present in `sentence`.

`‘taco ’ + ‘cat’` - returns a new string from concatenating the two strings

`f”My name is {name} and I’m {age} years old.”` - returns a string with the values of variables `name` and `age` substituted into `{name}` and `{age}`, respectively.

`sentence.replace(“brown”, “red”)` - replace every occurrence of `“brown”` with `“red”`

`len(sentence)` - returns the number of characters of the string

Lists

Lists are an ordered collection of Python objects. The items of the lists do not have to be the same data type. For example, you can store strings and integers inside the same list. Lists are mutable; they can be altered after their creation. Since they are ordered, they can be indexed by position. Note, Python uses zero indexing so the “first” element is index by 0. Lists are created using square brackets `[]`.

```
temp_data = [10.5, 12.2, 5, 8.7, 1]
```

Common operations on lists and example usage:

`temp_data.append(2.5)` - adds 2.5 to the end of the list

`temp_data.sort()` - sorts the elements of the list in ascending order. Use `reverse=True` to sort by descending order.

`temp_data.remove(12.2)` - removes the first occurrence of 12.2 from the list

`temp_data.pop()` - remove and returns the last element of the list

`temp_data[0]` - access value at position 0

`temp_data[:3]` - access the first three values, positions 0 to 3 (inclusive-exclusive)

`temp_data[-1]` - access the the last element

`temp_data[1:4:2]` - access values from position 1 (inclusive) and 4 (exclusive) with a step size of 2

`len(temp_data)` - returns the number of values in the list

`sum(temp_data)` - returns sum of the values of the list

`min(temp_data)` - returns the minimum value of the list

`max(temp_data)` - returns the maximum value of the list

Tuples

Tuples are similar to lists but they are immutable; they cannot be modified. As with lists, they can be indexed in a similar fashion. Tuples are created by using parentheses `()`.

```
array_shape = (100, 20)
```

Sets

Sets are a collection of unique values. They're a great data structure to use when wanting to keep track of only unique values. The members of a set need to be immutable. For example, lists are not allowed but tuples are. A set can be created by passing an iterable to set or directly using curly braces.

```
even_numbers = set([x for x in range(100)
    if x % 2 == 0])
squares = { 1, 1, 2, 4, 2, 9, 16, 25, 36,
    49, 64, 81, 100}
```

`even_numbers.add(100)` - add 100 to the set `even_numbers`

`even_numbers.difference(squares)` - returns a set that is the difference between `even_numbers` and `squares`

`even_numbers.union({1, 3, 5, 7, 9})` - return a set that is the union of the two sets

`squares.intersection(even_numbers)` - return set of common elements

`1 in even_numbers` - returns `True` or `False` if `1` is a member of the set `even_numbers`

Dictionary

Dictionaries store data in key-value pairs. Values can be indexed using the key associated with the value. There's no restriction in what can be values but keys are restricted to immutable types. For example, strings, numerics and tuples can be keys. Dictionaries are created using curly braces `{}` with the key and value pair separated by a colon `:`. Iterating over a dictionary yields the keys.

```
customer_data = {
     ‘name’: ‘Clarissa’,
     ‘account_id’: 100045,
     ‘account_balance’: 4515.76,
     ‘open_account’: True
}
```

`customer_data[‘name’]` - access value associated with ‘name’

`customer_data[‘telephone’] = None` - create new key-value pair `‘telephone’: None`

`customer_data[‘telephone’] = ‘555-1234’` - update value of key `‘telephone’`

`del customer_data[‘telephone’]` - delete key-value for `‘telephone’`

`‘age’ in customer_data` - returns True/False if key `‘age’` is in the dictionary

`customer_data.get(‘age’, -1)` - returns the value of key `‘age’` if it exists, returns `None` otherwise. Optional second argument is returned instead of None if key does not exist.

`customer_data.keys()` - returns an iterable over all keys

`customer_data.items()` - returns an iterable over all key-value pairs

`customer_data.values()` - returns an iterable over all values

* * *

SQL REFERENCE SHEET

POWERED BY THE SCIENTISTS AT THE DATA INCUBATOR

PRAGMATIC INSTITUTE

Query structure

```
SELECT <expressions>
FROM <tables>
WHERE <conditions>
GROUP BY <columns>
HAVING <conditions>
ORDER BY <columns>
LIMIT <number>
```

Only SELECT and FROM are mandatory

LIMIT restricts the number of rows returned

SELECT chooses what to get

```
SELECT customer_id, items*price AS total
FROM transactions;
```

can select columns or expressions, such as product, ratios, etc.

Rename cols or expressions with AS

Get number of rows in table customers:

```
SELECT COUNT(*) FROM customers;
```

Get distinct elements in column state:

```
SELECT DISTINCT state FROM customers;
```

Number of distinct elements:

```
SELECT COUNT(DISTINCT state)
FROM customers;
```

CASE allows if-like behavior

```
SELECT customer_id,
CASE WHEN items < 10 THEN ‘few’
WHEN items > 10 AND items < 100
THEN ‘many’
ELSE ‘lots’;
```

Sample tables

transactions

customer_id	items	price
27	5	12.00
33	25	11.00
60	150	9.00
60	250	9.00

customers

id	customer	state
44	Amy	CA
60	Brian	CA
27	Pat	NY
51	Alex	NULL

WHERE filters results

```
SELECT *
FROM customers
WHERE state = ‘CA’;
```

Select from a list:

```
WHERE customer IN (‘Amy’, ‘Pat’);
```

A pattern, % can be filled in with anything:

```
WHERE customer LIKE ‘A%’;
```

Find missing values:

```
WHERE state IS NULL;
```

Combine filters:

```
WHERE customer_id < 50
    AND state = ‘CA’;

WHERE name = ‘Amy’
    OR NOT state = ‘CA’;
```

Create temporary tables using schemas

```
CREATE TEMP TABLE trans (
    customer_id       INTEGER,
    items             INTEGER,
    price             REAL
);
```

Add to the table:

```
INSERT INTO trans VALUES (27, 5, 12.00),
(33, 25, 11.00);
```

```
CREATE TEMP TABLE custs (
    id          INTEGER PRIMARY KEY,
    customer    TEXT NOT NULL,
    state       TEXT
);
```

Or by saving a query

```
CREATE TEMP TABLE big AS
SELECT * FROM transactions
WHERE items > 100;
```

Replace `TEMP TABLE` with `TEMP VIEW` to get a live-updated `VIEW`.

`GROUP BY` aggregates

```
SELECT state, COUNT(*) AS number
FROM customers
GROUP BY state;
```

Many options for aggregation: `SUM`, `COUNT`, `AVG`, etc.

Everything in the `SELECT` must be either in the `GROUP BY` or in an aggregation.

`HAVING` is for conditioning after aggregation.

```
SELECT state, COUNT(*) AS number
FROM customers
GROUP BY state
HAVING COUNT(*) > 1;
```

`JOIN` combines tables

Use `ON` or `WHERE` to set matching condition. Use table prefix if ambiguous.

```
SELECT customer, items, state
FROM customers JOIN transactions
ON customer_id = id;
```

```
SELECT customer, items, state
FROM customers, transactions
WHERE customer_id = customers.id;
```

`LEFT JOIN` includes unmatched values from the first table.

```
SELECT customer, items, state
FROM customers LEFT JOIN transactions
ON customer_id = customers.id;
```

`RIGHT JOIN` does the same for the second table.

```
SELECT customer, items, state
FROM customers RIGHT JOIN transactions
ON customer_id = id;
```

`FULL JOIN` includes all unmatched rows.

```
SELECT customer, items, state
FROM customers FULL JOIN transactions
ON customer_id = customers.id;
```

Subqueries allow more complex operations

```
SELECT customer, total
FROM customers JOIN
    (SELECT customer_id,
    SUM(items*price) AS total
    FROM transactions
    GROUP BY customer_id) AS orders
ON customer_id = id;
```