Supplemental Intro to Python Material

John Strickler

Version 1.0, September 2023

Table of Contents

Chapter 1: IPython and Jupyter

Objectives

- Learn the basics of IPython
- Apply magics
- List and replay commands
- Run external commands
- Create profiles
- Use Jupyter notebooks

About IPython

- Enhanced python interpreter
- Great for "playing around" with Python
- Saves running entire script
- Not intended for application development
- Embedded in Jupyter notebooks

IPython is an enhanced interpreter for Python. It provides a large number of "creature comforts" for the user, such as name completion and improved help features.

It is very handy for quickly trying out Python features or for casual data analysis.

Command line interface

When started from a command line, starts a read-execute-print loop (REPL), also known as an interactive interpreter.

Ipython uses different colors for variables, functions, strings, comments, and so forth.

Jupyter notebook

The most flexible and powerful way to run IPython is embedded in a Jupyter notebook. This mode starts a dedicated web server and begins a session using your default web browser. From the home page of Jupyter, you can create a Jupyter notebook containing Python code.

Starting IPython

- Type `ipython` at the command line
- Huge number of options

To get started with IPython

- Type `ipython` at the command line

OR

- Double-click the IPython icon from Windows explorer.

IPython works like the normal interactive Python interpreter, but with many more features.

There is a huge number of options. To see them all, invoke IPython with the --help-all option:

```
ipython help-all
```

TIP Use the `--colors=NoColor` option to turn off syntax highlighting and other colorized features.

Getting Help

- ? basic help
- %quickref quick reference
- help standard Python help
- thing? help on thing

IPython provides help in several ways.

Typing `?` at the prompt will display an introduction to IPython and a feature overview.

For a quick reference card, type `%quickref`.

To start Python's normal help system, type `help`.

For help on any Python object, type `object?` or `?object`. This is similar to saying `help("object")` in the default interpreter, but is "smarter".

TIP For more help, add a second question mark. This does not work for all objects, however, and sometimes it displays the source code of the module containing the object definition.

IPython features

- Name completion (variables, modules, methods, folders, files, etc.)
- Enhanced help system
- Autoindent
- Syntax highlighting
- 'Magic' commands for controlling IPython itself
- Easy access to shell commands
- Dynamic introspection (dir() on steroids)
- Search namespaces with wildcards
- Commands are numbered (and persistent) for recall
- Aliasing system for interpreter commands
- Simplified (and lightweight) persistence
- Session logging (can be saved as scripts)
- Detailed tracebacks when errors occur
- Session restoring (playback log to specific state)
- Flexible configuration system
- Easy access to Python debugger
- Simple profiling
- Interactive parallel computing (if supported by hardware)
- Background execution in separate thread
- Auto-parentheses ('sin 3' becomes 'sin(3)'
- Auto-quoting (',foo a b' becomes 'foo("a","b")'

Tab Completion

- Press `Tab` to complete
  - keywords
  - variables
  - modules
  - methods and attributes
  - parameters to functions
  - file and directory names

Pressing `Tab` will invoke **tab completion**, AKA **autocomplete**. If there is only one possible completion, it will be expanded. If there is more than one completion that will match, IPython will display a list of possible completions.

Autocomplete works on keywords, functions, classes, methods, and object attributes, as well as paths from your file system.

Magic Commands

- Start with `%` (line magic) or `%%` (cell magic)
- Simplify common tasks
- Use `%lsmagic` to list all magic commands

One of the enhancements in IPython is the set of "magic" commands. These are meta-commands (macros) that help you manipulate the IPython environment.

Normal magics apply to a single line. Cell magics apply to a cell (a group of lines).

For instance, `%history` will list previous commands.

Type `lsmagic` for a list of all magics

TIP If the magic command is not the same as a name in your Python code, you can leave off the leading `%` or `%%`.

Loading and running Python scripts

- Run script in current session
- `%run` runs script
- `%load` loads script source code into IPython

IPython provides two magics to run scripts — one to run directly, and one to run indirectly. Both will run the script in the context of the current IPython session.

Running scripts directly

The `%run` magic just takes a script name, and runs it. This method does not allow IPython magics to be executed as part of a script.

```
In [1]: %run ../EXAMPLES/my_vars.py
```

```
In [2]:  user_name
Out[2]: 'Susan'
```

```
In [3]: snake
Out[3]: 'Eastern Racer'
```


Running scripts indirectly

The `%load` magic takes a script name, and loads the contents of the script so it can then be executed.

This method allows IPython magics to be executed as part of a script.

This also useful if you want to run a script, but edit the script before it is run.

```
In [4]: %load imports.py
```

```
In [5]: # %load imports.py
...: import numpy as np
...: import scipy as sp
...: import pandas as pd
...: import matplotlib.pyplot as plt
...: import matplotlib as mpl
...: %matplotlib inline
...: import seaborn as sns
...: sns.set()
...:
...:
```

External commands

- Precede command with !
- Can assign output to variable

Any OS command can be run by starting it with a !.

The resulting output is returned as a list of strings (stripping the trailing \n characters). The result can be assigned to a variable.

Windows

```
In [3]: !dir DATA\*.csv
 Volume in drive Z is Shared Folders
 Volume Serial Number is 0000-0064

 Directory of Z:\Desktop\py2forsci\DATA

02/20/2014  01:53 PM             5,511 airport_boardings.csv
02/20/2014  01:53 PM             2,182 energy_use_quad.csv
02/20/2014  01:53 PM             4,993 parasite_data.csv
               3 File(s)         12,686 bytes
               0 Dir(s)  352,625,324,032 bytes free

In [4]:
```

Non-Windows (Linux, OS X, etc)

```
In [2]: !ls -l DATA/*.csv
-rwxr-xr-x  1 jstrick  staff  5511 Jan 27 19:44 DATA/airport_boardings.csv
-rwxr-xr-x  1 jstrick  staff  2182 Jan 27 19:44 DATA/energy_use_quad.csv
-rwxr-xr-x  1 jstrick  staff  4642 Jan 27 19:44 DATA/parasite_data.csv

In [3]:
```


Using history

- use %history magic
- `history` *list commands*
- `history -n` *list commands with numbers*
- `hist` *shortcut for "history"*

The `%history` magic will list previous commands. Use `-n` to list commands with their numbers.

Selecting commands

You can select a single command or a range of commands separated by a dash.

```
history 5
history 6-10
```

Use `~N/`, where N is 1 or greater, to select commands from previous sessions.

```
history ~2/3  third command in second previous session
```

To select more than one range or individual command, separate them by spaces.

```
history 4-6 9 12-16
```

TIP The same syntax can be used with `%edit`, `%rerun`, `%recall`, `%macro`, `%save` and `%pastebin`.

Recalling commands

The `%recall` magic will recall a previous command by number. It will leave the cursor at the end of the command so you can edit it.

```
recall 12
recall 4-7
```

Rerunning commands

`%rerun` will re-run a previous command without waiting for you to press `Enter`.

Saving sessions

- Save commands to Python script
- Specify one or more commands
- Use `%save` magic

It is easy to save a command, a range of commands, or any combination of commands to a Python script using the `%save` magic.

The syntax is

```
%save filename selected commands
```

.py will be appended to the filename.

Using Pastebin

- Online "clipboard"
- Use `%pastebin` command

Pastebin is a free online service that accepts pasted text and provides a link to access the text. It can be used to share code snippets with other programmers.

The `%pastebin` magic will paste selected commands to **Pastebin** and return a link that can be used to retrieve them. The link provided will expire in 7 days.

Use `-d` to specify a title for the pasted code.

```
link = %pastebin -d "my code" 10-15  write commands 10 through 15 to Pastebin and get link
```

TIP Add ".txt" to the link to retrieve the plain text that you pasted. This can be done with `requests`:

```
import requests
link = %pastebin -d "my code" 10-15
pasted_text = requests.get(link + '.txt').text
```


Benchmarking

- Use %timeit

IPython has a handy magic for benchmarking.

```
In [1]: color_values = { 'red':66, 'green':85, 'blue':77 }

In [2]: %timeit red_value = color_values['red']
10000000 loops, best of 3: 54.5 ns per loop

In [3]: %timeit red_value = color_values.get('red')
10000000 loops, best of 3: 115 ns per loop
```

%timeit will benchmark whatever code comes after it on the same line. %%timeit will benchmark contents of a notebook cell

Profiles

- Stored in `.ipython` folder in home folder
- Contains profiles and other configuration
- Can have multiple profiles
- `ipython profile` subcommands
  - `list`
  - `create`
  - `locate`

IPython supports *profiles* for storing custom configurations and startup scripts. There is a default profile, and any number of custom profiles can be created.

Each profile is a separate subfolder under the `.ipython` folder in a users's home folder.

Creating profiles

Use `ipython profile create name` to create a new named profile. If `name` is omitted, this will create the default profile (if it does not already exist)

Listing profiles

Use `ipython profile list` to list all profiles

Finding profiles

`ipython profile locate name` will display the path to the specified profile. As with creating, omitting the name shows the path to the default profile.

```
.ipython
├── cython
│   └── Users
│       └── mikedev
├── extensions
├── nbextensions
├── profile_default
│   ├── db
│   ├── ipython_config.py
│   ├── ipython_kernel_config.py
│   ├── log
│   ├── pid
│   ├── security
│   ├── startup
│   │   ├── 00_imports.py
│   │   └── 10_macros.py
│   └── static
│       └── custom
└── profile_science
    ├── db
    ├── ipython_config.py
    ├── ipython_kernel_config.py
    ├── log
    ├── pid
    ├── security
    └── startup
        └── 00_imports.py
```

Configuration

IPython has many configuration settings. You can change these settings by creating or editing the script named `ipython_config.py` in a profile folder.

Within this script you can use the global config object, named `c`.

For instance, the line

```
c.InteractiveShellApp.pylab_import_all = False
```

Will change how the `%pylab` magic works. When true, it will populate the user namespace with the contents of `numpy` and `pylab` as though you had entered `from numpy import *` and `from pylab import *`

When false, it will just import `numpy` as `np` and `pylab` as `pylab`.

Link to all IPython settings:

https://ipython.org/ipython-doc/3/config/options/index.html

Note that there are four groups of settings.

> **TIP** When you create a profile, this config script is created with some commented code to get you started.

Startup

Startup scripts allow you to execute frequently used code, especially imports, when starting IPython.

Startup scripts go in the `startup` folder of the profile folder. All Python scripts in this folder will be executed, in lexicographical (sorted) order.

The scripts will be executed in the context of the IPython session, so all imports, variables, functions, classes, and other definitions will be available in the session.

TIP It is convenient to prefix the startup scripts with "00", "10", "20", and so forth, to set the order of execution.

Jupyter notebooks

- Extension of IPython
- Puts the interpreter in a web browser
- Code is grouped into "cells"
- Cells can be edited, repeated, etc.

In 2015, the developers of IPython pulled the notebook feature out of IPython to make a separate product called Jupyter. It is still invoked via the `jupyter notebook` command, and now supports over 130 language kernels in addition to Python.

A Jupyter notebook is a journal-like python interpreter that lives in a browser window. Code is grouped into cells, which can contain multiple statements. Cells can be edited, repeated, rearranged, and otherwise manipulated.

A notebook (i.e, a set of cells, can be saved, and reopened). Notebooks can be shared among members of a team via the notebook server which is built into Jupyter.

Jupyter Notebook Demo

At this point please start the Jupyter notebook server and follow along with a demo of Jupyter notebooks as directed by the instructor

Open an Anaconda prompt and navigate to the top folder of the student files, then

```
cd NOTEBOOKS
jupyter notebook
```


For more information

- https://ipythonbook.com

Chapter 2: Introduction to Pandas

Objectives

- Understand what the pandas module provides
- Load data from CSV and other files
- Access data tables
- Extract rows and columns using conditions
- Calculate statistics for rows or columns

About pandas

- Reads data from file, database, or other sources
- Deals with real-life issues such as invalid data
- Powerful selecting and indexing tools
- Builtin statistical functions
- Munge, clean, analyze, and model data
- Works with numpy and matplotlib

pandas is a package designed to make it easy to get, organize, and analyze large datasets. Its strengths lie in its ability to read from many different data sources, and to deal with real-life issues, such as missing, incomplete, or invalid data.

pandas also contains functions for calculating means, sums and other kinds of analysis.

For selecting desired data, pandas has many ways to select and filter rows and columns.

It is easy to integrate pandas with NumPy, Matplotlib, and other scientific packages.

While pandas can handle three (or higher) dimensional data, it is generally used with two-dimensional (row/column) data, which can be visualized like a spreadsheet.

pandas provides powerful split-apply-combine operations — **groupby** enables transformations, aggregations, and easy-access to plotting functions. It is easy to emulate R's plyr package via pandas.

NOTE pandas gets its name from *pan*el *da*ta *s*ystem

Tidy data

- Tidy data is neatly grouped
- Data
  - *Value* = "observation"
  - *Column* = "variable"
  - *Row* = "related observations"
- Pandas best with tidy data

A dataset contains *values*. Those values can be either numbers or strings. Values are grouped into *variables*, which are usually represented as *columns*. For instance, a column might contain "unit price" or "percentage of NaCL". A group of related values is called an *observation*. A *row* represents an observation. Every combination of row and column is a single value.

When data is arranged this way, it is said to be "tidy". Pandas is designed to work best with tidy data.

For instance,

```
Product    SalesYTD
oranges    5000
bananas    1000
grapefruit 10000
```

is tidy data. The variables are "Product" and "SalesYTD", and the observations are the names of the fruits and the sales figures.

The following dataset is NOT tidy:

```
Fruit    oranges bananas grapefruit
SalesYTD 5000    1000    10000
```

To make selecting data easy, Pandas dataframes always have variable labels (columns) and observation labels (row indexes). A row index could be something simple like increasing integers, but it could also be a time series, or any set of strings, including a column pulled from the data set.

TIP variables could be called "features" and observations could be called "samples"

NOTE See https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html for a detailed discussion of tidy data.

pandas architecture

- Two main structures: Series and DataFrame
- Series – one-dimensional
- DataFrame – two-dimensional

The two main data structures in pandas are the **Series** and the **DataFrame**. A series is a one-dimensional indexed list of values, something like an ordered dictionary. A DataFrame is is a two-dimensional grid, with both row and column indexes (like the rows and columns of a spreadsheet, but more flexible).

You can specify the indexes, or pandas will use successive integers. Each row or column of a DataFrame is a Series.

NOTE pandas used to support the **Panel** type, which is more more or less a collection of DataFrames, but Panel has been deprecated in favor of hierarchical indexing.

Series

- Indexed list of values
- Similar to a dictionary, but ordered
- Can get sum(), mean(), etc.
- Use index to get individual values
- indexes are not positional

A Series is an indexed sequence of values. Each item in the sequence has an index. The default index is a set of increasing integer values, but any set of values can be used.

For example, you can create a series with the values 5, 10, and 15 as follows:

```
s1 = pd.Series([5,10,15])
```

This will create a Series indexed by [0, 1, 2]. To provide index values, add a second list:

```
s2 = pd.Series([5,10,15], ['a','b','c'])
```

This specifies the indexes as 'a', 'b', and 'c'.

You can also create a Series from a dictionary. pandas will put the index values in order:

```
s3 = pd.Series({'b':10, 'a':5, 'c':15})
```

There are many methods that can be called on a Series, and Series can be indexed in many flexible ways.

Example

pandas_series.py

```
from numpy.random import default_rng
import pandas as pd

NUM_DATA_POINTS = 10
index = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

rng = default_rng()
data = rng.standard_normal(NUM_DATA_POINTS)

s1 = pd.Series(data, index=index)  # create series with specified index
s2 = pd.Series(data)  # create series with auto-generated index (0, 1, 2, 3, ...)

print("s1:", s1, "\n")
print("s2:", s2, "\n")

print("selecting elements")
print(s1[['h', 'b']], "\n")  # select items from series

print(s1[['a', 'b', 'c']], "\n")  # select items from series

print("slice of elements")
print(s1['b':'d'], "\n")  # select slice of elements

print("sum(), mean(), min(), max():")
print(s1.sum(), s1.mean(), s1.min(), s1.max(), "\n")  # get stats on series

print("cumsum(), cumprod():")
print(s1.cumsum(), s1.cumprod(), "\n")  # get stats on series

print('a' in s1)  # test for existence of label
print('m' in s1)  # test for existence of label
print()

s3 = s1 * 10  # create new series with every element of s1 multiplied by 10
print("s3 (which is s1 * 10)")
print(s3, "\n")

s1['e'] *= 5

print("boolean mask where s3 > 0:")
print(s3 > 0, "\n")  # create boolean mask from series

print("assign -1 where mask is true")
```

```
s3[s3 < 5] = -1  # set element to -1 where mask is True
print(s3, "\n")

s4 = pd.Series([-0.204708, 0.478943, -0.519439])  # create new series
print("s4.max(), .min(), etc.")
print(s4.max(), s4.min(), s4.max() - s4.min(), '\n')  # print stats

s = pd.Series([5, 10, 15], ['a', 'b', 'c'])  # create new series with index
print("creating series with index")
print(s)
```


pandas_series.py

```
s1: a    0.893738
b    0.753999
c    2.335929
d   -0.320981
e    1.545986
f    1.871491
g   -0.343049
h   -1.158003
i    0.353271
j    1.537565
dtype: float64

s2: 0    0.893738
1    0.753999
2    2.335929
3   -0.320981
4    1.545986
5    1.871491
6   -0.343049
7   -1.158003
8    0.353271
9    1.537565
dtype: float64

selecting elements
h   -1.158003
b    0.753999
dtype: float64

a    0.893738
b    0.753999
c    2.335929
dtype: float64

slice of elements
b    0.753999
c    2.335929
d   -0.320981
dtype: float64

sum(), mean(), min(), max():
7.469944930767108 0.7469944930767107 -1.1580031739916454 2.33592854066975

cumsum(), cumprod():
a    0.893738
```

```
b    1.647736
c    3.983665
d    3.662683
e    5.208670
f    7.080161
g    6.737112
h    5.579109
i    5.932380
j    7.469945
dtype: float64 a    0.893738
b    0.673877
c    1.574129
d   -0.505266
e   -0.781134
f   -1.461886
g    0.501498
h   -0.580737
i   -0.205157
j   -0.315442
dtype: float64

True
False

s3 (which is s1 * 10)
a     8.937378
b     7.539985
c    23.359285
d    -3.209814
e    15.459865
f    18.714911
g    -3.430488
h   -11.580032
i     3.532706
j    15.375652
dtype: float64

boolean mask where s3 > 0:
a     True
b     True
c     True
d    False
e     True
f     True
g    False
h    False
i     True
j     True
```



```
dtype: bool

assign -1 where mask is true
a     8.937378
b     7.539985
c    23.359285
d    -1.000000
e    15.459865
f    18.714911
g    -1.000000
h    -1.000000
i    -1.000000
j    15.375652
dtype: float64

s4.max(), .min(), etc.
0.478943 -0.519439 0.998382

creating series with index
a     5
b    10
c    15
dtype: int64
```

DataFrames

- Two-dimensional grid of values
- Row and column labels (indexes)
- Rich set of methods
- Powerful indexing

A DataFrame is the workhorse of pandas. It represents a two-dimensional grid of values, containing indexed rows and columns, something like a spreadsheet.

There are many ways to create a DataFrame. They can be modified to add or remove rows/columns. Missing or invalid data can be eliminated or normalized.

DataFrames can be initialized from many kinds of data. See the table on the next page for a list of possibilities.

NOTE	The panda DataFrame is modeled after R's data.frame

Table 1. DataFrame Initializers

Initializer	Description
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are union-ed together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are union-ed to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

IMPORTANT Most, if not all, of the time you will create Series and Dataframes by reading data.

Example

pandas_simple_dataframe.py

```python
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column names
indices = ['a', 'b', 'c', 'd', 'e', 'f']  # row names

values = [  # sample data
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('cols')
print(cols, '\n')

print_header('indices')
print(indices, '\n')

print_header('values')
print(values, '\n')

df = pd.DataFrame(values, index=indices, columns=cols)  # create dataframe with row and column names
print_header('DataFrame df')
print(df, '\n')

print_header("df['gamma']")
print(df['gamma'])  # select column 'gamma'
```


pandas_simple_dataframe.py

```
==================================================
=                       cols                     =
==================================================
['alpha', 'beta', 'gamma', 'delta', 'epsilon']

==================================================
=                     indices                    =
==================================================
['a', 'b', 'c', 'd', 'e', 'f']

==================================================
=                      values                    =
==================================================
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340], [400,
410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]

==================================================
=                   DataFrame df                 =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                   df['gamma']                  =
==================================================
a    120
b    220
c    320
d    420
e    520
f    620
Name: gamma, dtype: int64
```

Reading Data

- Supports many data formats
- Reads headings to create column indexes
- Auto-creates indexes as needed
- Can used specified column as row index

Pandas supports many different input formats. It will read file headings and use them to create column indexes. By default, it will use integers for row indexes, but you can specify a column to use as the index, or provide a list of index values.

The **read_...()** functions have many options for controlling and parsing input. For instance, if large integers in the file contain commas, the thousands options let you set the separator as comma (in the US), so it will ignore them.

read_csv() is the most frequently used function, and has many options. It can also be used to read generic flat-file formats. **read_table** is similar to **read_csv()**, but doesn't assume CSV format.

There are corresponding **to_...()** functions for many of the read functions. `to_csv()` and `to_ndarray()` are very useful.

NOTE

See **Jupyter** notebook **pandas_Input_Demo** (in the **NOTEBOOKS** folder) for examples of reading most types of input.

See https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html?highlight=output#io-html for details on the I/O functions.

Example

pandas_read_csv.py

```
import pandas as pd

df = pd.read_csv('../DATA/sales_records.csv')  # Read CSV data into dataframe. Pandas
automatically uses the first row as column names

print(df.describe())  # Get statistics on the numeric columns (use
`df.describe(include='O')` for text columns)
print()

print(df.info())  # Get information on all the columns ('object' means text/string)
print()

print(df.head(5))  # Display first 5 rows of the dataframe (`df.describe(__n__)` displays
n rows)

df['total_sales'] = df['Units Sold'] * df['Unit Price']
print(df)

print(df.info())
print(df.describe())
```

pandas_read_csv.py

```
            Order ID   Units Sold   Unit Price    Unit Cost
count  5.000000e+03  5000.000000  5000.000000  5000.000000
mean   5.486447e+08  5030.698200   265.745564   187.494144
std    2.594671e+08  2914.515427   218.716695   176.416280
min    1.000909e+08     2.000000     9.330000     6.920000
25%    3.201042e+08  2453.000000    81.730000    35.840000
50%    5.523150e+08  5123.000000   154.060000    97.440000
75%    7.687709e+08  7576.250000   437.200000   263.330000
max    9.998797e+08  9999.000000   668.270000   524.960000

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Region          5000 non-null   object
 1   Country         5000 non-null   object
 2   Item Type       5000 non-null   object
 3   Sales Channel   5000 non-null   object
```

```
 4   Order Priority  5000 non-null   object
 5   Order Date      5000 non-null   object
 6   Order ID        5000 non-null   int64
 7   Ship Date       5000 non-null   object
 8   Units Sold      5000 non-null   int64
 9   Unit Price      5000 non-null   float64
 10  Unit Cost       5000 non-null   float64
dtypes: float64(2), int64(2), object(7)
memory usage: 429.8+ KB
None

                              Region  ... Unit Cost
0  Central America and the Caribbean  ...    159.42
1  Central America and the Caribbean  ...     97.44
2                             Europe  ...     31.79
3                               Asia  ...    117.11
4                               Asia  ...     97.44

[5 rows x 11 columns]
                                 Region  ... total_sales
0     Central America and the Caribbean  ...   140914.56
1     Central America and the Caribbean  ...   330640.86
2                                Europe  ...   226716.10
3                                  Asia  ...  1854591.20
4                                  Asia  ...  1150758.36
...                                 ...  ...         ...
4995              Australia and Oceania  ...  3545172.35
4996       Middle East and North Africa  ...   117694.56
4997                               Asia  ...  1328477.12
4998                             Europe  ...  1028324.80
4999                 Sub-Saharan Africa  ...   377447.00

[5000 rows x 12 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Region          5000 non-null   object
 1   Country         5000 non-null   object
 2   Item Type       5000 non-null   object
 3   Sales Channel   5000 non-null   object
 4   Order Priority  5000 non-null   object
 5   Order Date      5000 non-null   object
 6   Order ID        5000 non-null   int64
 7   Ship Date       5000 non-null   object
 8   Units Sold      5000 non-null   int64
 9   Unit Price      5000 non-null   float64
```

```
 10  Unit Cost       5000 non-null   float64
 11  total_sales     5000 non-null   float64
dtypes: float64(3), int64(2), object(7)
memory usage: 468.9+ KB
None
           Order ID   Units Sold   Unit Price    Unit Cost   total_sales
count  5.000000e+03  5000.000000  5000.000000  5000.000000  5.000000e+03
mean   5.486447e+08  5030.698200   265.745564   187.494144  1.325738e+06
std    2.594671e+08  2914.515427   218.716695   176.416280  1.475375e+06
min    1.000909e+08     2.000000     9.330000     6.920000  6.531000e+01
25%    3.201042e+08  2453.000000    81.730000    35.840000  2.574168e+05
50%    5.523150e+08  5123.000000   154.060000    97.440000  7.794095e+05
75%    7.687709e+08  7576.250000   437.200000   263.330000  1.839975e+06
max    9.998797e+08  9999.000000   668.270000   524.960000  6.672676e+06
```

Table 2. pandas I/O functions

Format	Input function	Output function
CSV	`read_csv()`	`to_csv()`
Delimited file (generic)	`read_table()`	`to_csv()`
Excel worksheet	`read_excel()`	`to_excel()`
File with fixed-width fields	`read_fwf()`	
Google BigQuery	`read_gbq()`	`to_gbq()`
HDF5	`read_hdf()`	`to_hdf()`
HTML table	`read_html()`	`to_html()`
JSON	`read_json()`	`to_json()`
OS clipboard data	`read_clipboard()`	`to_clipboard()`
Parquet	`read_parquet()`	`to_parquet()`
pickle	`read_pickle()`	`to_pickle()`
SAS	`read_sas()`	
SQL query	`read_sql()`	`to_sql()`

NOTE All **read_...()** functions return a new **DataFrame**, except **read_html()**, which returns a list of **DataFrames**

Data summaries

- `describe()` *basic statistical details*
- `info()` *per-column details (shallow memory use)*
- `info(memory_usage='deep')` *actual memory use*

You can call the `describe()` and `info()` methods on a dataframe to get summaries of the kind of data contained.

The `describe()` method, by default, shows statistics on all numeric columns. Add `include='int'` or `include='float'` to restrict the output to those types. `include='all'` will show all types, including "objects" (AKA text).

To show just objects (strings), use `include='O'`. This will show all text columns. You can compare the **count** and **unique** values to check the *cardinality* of the column, or how many distinct values there are. Columns with few unique values are said to have low cardinality, and are candidates for saving space by using the `Categorical` data type.

The `info()` method will show the names and types of each column, as well as the count of non-null values. Adding `memory_usage='deep'` will display the total memory actually used by the dataframe. (Otherwise, it's only the memory used by the top-level data structures).

Example

pandas_data_summaries.py

```
import pandas as pd
from printheader import print_header

df = pd.read_csv('../DATA/airport_boardings.csv', thousands=',', index_col=1)

print_header('df.head()')
print(df.head())
print()

print_header('df.describe()')
print(df.describe())

print_header("df.describe(include='int')")
print(df.describe(include='int'))

print_header("df.describe(include='all')")
print(df.describe(include='all'))

print_header("df.info()")
print(df.info())
```

pandas_data_summaries.py

```
==================================================
=                   df.head()                    =
==================================================
                                                Airport  ...  Percent change 2010-2011
Code                                                     ...
ATL   Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...                     -22.6
ORD          Chicago, IL (Chicago O'Hare International)  ...                     -25.5
DFW        Dallas, TX (Dallas/Fort Worth International)  ...                     -23.7
DEN                   Denver, CO (Denver International)  ...                     -23.1
LAX         Los Angeles, CA (Los Angeles International)  ...                     -19.6

[5 rows x 9 columns]

==================================================
=                 df.describe()                  =
==================================================
       2001 Rank  ...  Percent change 2010-2011
count  50.000000  ...                 50.000000
mean   26.460000  ...                -23.758000
```

```
std    15.761242  ...                2.435963
min     1.000000  ...              -32.200000
25%    13.250000  ...              -25.275000
50%    26.500000  ...              -23.650000
75%    38.750000  ...              -22.075000
max    59.000000  ...              -19.500000

[8 rows x 8 columns]
==================================================
=           df.describe(include='int')           =
==================================================
       2001 Rank    2001 Total  ...  2011 Rank         Total
count  50.000000  5.000000e+01  ...   50.00000  5.000000e+01
mean   26.460000  9.848488e+06  ...   25.50000  8.558513e+06
std    15.761242  7.042127e+06  ...   14.57738  6.348691e+06
min     1.000000  2.503843e+06  ...    1.00000  2.750105e+06
25%    13.250000  4.708718e+06  ...   13.25000  3.300611e+06
50%    26.500000  7.626439e+06  ...   25.50000  6.716353e+06
75%    38.750000  1.282468e+07  ...   37.75000  1.195822e+07
max    59.000000  3.638426e+07  ...   50.00000  3.303479e+07

[8 rows x 6 columns]
==================================================
=           df.describe(include='all')           =
==================================================
                                                  Airport  ...  Percent change 2010-2011
count                                                  50  ...                 50.000000
unique                                                 50  ...                       NaN
top     Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...                       NaN
freq                                                    1  ...                       NaN
mean                                                  NaN  ...                -23.758000
std                                                   NaN  ...                  2.435963
min                                                   NaN  ...                -32.200000
25%                                                   NaN  ...                -25.275000
50%                                                   NaN  ...                -23.650000
75%                                                   NaN  ...                -22.075000
max                                                   NaN  ...                -19.500000

[11 rows x 9 columns]
==================================================
=                   df.info()                    =
==================================================
<class 'pandas.core.frame.DataFrame'>
Index: 50 entries, ATL to IND
Data columns (total 9 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Airport                  50 non-null     object
```

```
 1   2001 Rank                 50 non-null     int64
 2   2001 Total                50 non-null     int64
 3   2010 Rank                 50 non-null     int64
 4   2010 Total                50 non-null     int64
 5   2011 Rank                 50 non-null     int64
 6    Total                    50 non-null     int64
 7   Percent change 2001-2011  50 non-null     float64
 8   Percent change 2010-2011  50 non-null     float64
dtypes: float64(2), int64(6), object(1)
memory usage: 3.9+ KB
None
```


Basic Indexing

- Similar to normal Python or numpy
- Slices select rows

One of the real strengths of pandas is the ability to easily select desired rows and columns. This can be done with simple subscripting, like normal Python, or extended subscripting, similar to numpy. In addition, pandas has special methods and attributes for selecting data.

For selecting columns, use the column name as the subscript value. This selects the entire column. To select multiple columns, use a sequence (list, tuple, etc.) of column names.

For selecting rows, use slice notation. This may not map to similar tasks in normal python. That is, dataframe[x:y] selects rows x through y, but dataframe[x] selects column x.

Example

pandas_selecting.py

```
import pandas as pd
from printheader import print_header

columns = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column labels
index = ['a', 'b', 'c', 'd', 'e', 'f']  # row labels

values = [  # sample data
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=columns)  # create dataframe with data,
row labels, and column labels
print_header('DataFrame df')
print(df, '\n')

print_header("df['alpha']")
print(df['alpha'], '\n')  # select column 'alpha' -- single value selects column by name

print_header("df.beta")
print(df.beta, '\n')  # same, but alternate syntax (only works if column name is letters,
digits, and underscores)

print_header("df[['alpha','epsilon','beta']]")
print(df[['alpha', 'epsilon', 'beta']])  # select columns -- note index is an iterable
print()

print_header("df['b':'e']")
print(df['b':'e'], '\n')  # select rows 'b' through 'e' using slice of row labels

print_header("df['b':'b']")
print(df['b':'b'], '\n')  # select row 'b' only using slice of row labels (returns
dataframe)

print_header("df[['alpha','epsilon','beta']]['b':'e']")
print(df[['alpha', 'epsilon', 'beta']]['b':'e'])  # select columns AND slice rows
print()
```


pandas_selecting.py

```
==================================================
=                 DataFrame df                   =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                 df['alpha']                    =
==================================================
a    100
b    200
c    300
d    400
e    500
f    600
Name: alpha, dtype: int64

==================================================
=                   df.beta                      =
==================================================
a    110
b    210
c    310
d    410
e    510
f    610
Name: beta, dtype: int64

==================================================
=         df[['alpha','epsilon','beta']]         =
==================================================
   alpha  epsilon  beta
a    100      140   110
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
f    600      640   610

==================================================
```

```
=                  df['b':'e']                  =
=================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540

=================================================
=                  df['b':'b']                  =
=================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240

=================================================
=    df[['alpha','epsilon','beta']]['b':'e']     =
=================================================
   alpha  epsilon  beta
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
```

Saner indexing

- `loc[row-spec,col-spec]` for names (strings or numbers)
- `.iloc[row-spec,col-spec]` for 0-based position (integers only)
- `.loc[]` row or column specs can be
  - single name
  - iterable of names
  - range (inclusive) of names
- `.iloc[]` row or column specs can be
  - single number
  - iterable of numbers
  - range (exclusive) of numbers
- `.at[]` single value

The `.loc` and `.iloc` indexers provide more extensive and consistent selecting of rows and columns for dataframes. They both work exactly the same way, but `.loc` uses only row and column *names*, and `.iloc` uses only *positions*.

Both indexers use the *getitem* operator `[]`, with the syntax `[row-specifier, column-specifier]`.

For `.loc[]`, the specifier can be either a single name, an iterable of names, or a range of names. The end of a range is inclusive.

For `.iloc[]`, the specifier can be either a single numeric index (0-based), iterable of indexes, or a range of indexes. The end of a range is exclusive.

To select all rows, or all columns, use `:`.

The `.at[]` property can be used to select a single value at a given row and column: `df.at[47, "color"]`.

NOTE The column specifier can be omitted, which will select all columns for those rows.

Example

pandas_loc.py

```python
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
indices = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=indices, columns=cols)
print_header('DataFrame df')
print(df, '\n')

print_header("df.loc['b', 'delta']")  # one value
print(df.loc['b', 'delta'], "\n")


print_header("df.loc['b']")  # one row
print(df.loc['b'], '\n')

print_header("df.loc[:,'delta']")  # one column
print(df.loc[:,'delta'], '\n')


print_header("df.loc['b': 'd']")  # range of rows
print(df.loc['b':'d', :], '\n')
print(df.loc['b':'d'], '\n')   # shorter version

print_header("df.loc[:,'beta':'delta'")  # range of columns
print(df.loc[:, 'beta':'delta'], "\n")

print_header("df.loc['b':'d', 'beta':'delta']")  # ranges of rows and columns
print(df.loc['b':'d', 'beta':'delta'], '\n')

print_header("df.loc[['b', 'e', 'a']]")  # iterable of rows
print(df.loc[['b', 'e', 'a']], "\n")
```

```
print_header("df.loc[:, ['gamma', 'alpha', 'epsilon']]")  # iterable of columns
print(df.loc[:, ['gamma', 'alpha', 'epsilon']], "\n")

print_header("df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']]")  # iterables of
rows and columns
print(df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']], "\n")
```

pandas_loc.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=              df.loc['b', 'delta']              =
==================================================
230

==================================================
=                  df.loc['b']                   =
==================================================
alpha      200
beta       210
gamma      220
delta      230
epsilon    240
Name: b, dtype: int64

==================================================
=               df.loc[:,'delta']                =
==================================================
a    130
b    230
c    330
d    430
e    530
f    630
Name: delta, dtype: int64
```

```
==================================================
=                 df.loc['b': 'd']               =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440

   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440

==================================================
=             df.loc[:,'beta':'delta'            =
==================================================
   beta  gamma  delta
a   110    120    130
b   210    220    230
c   310    320    330
d   410    420    430
e   510    520    530
f   610    620    630

==================================================
=         df.loc['b':'d', 'beta':'delta']        =
==================================================
   beta  gamma  delta
b   210    220    230
c   310    320    330
d   410    420    430

==================================================
=             df.loc[['b', 'e', 'a']]            =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
e    500   510    520    530      540
a    100   110    120    130      140

==================================================
=     df.loc[:, ['gamma', 'alpha', 'epsilon']]   =
==================================================
   gamma  alpha  epsilon
a    120    100      140
b    220    200      240
c    320    300      340
```



```
d    420    400      440
e    520    500      540
f    620    600      640

==================================================
df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']]
==================================================
   gamma  alpha  epsilon
b    220    200      240
e    520    500      540
a    120    100      140
```

Example

pandas_iloc.py

```
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
indices = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=indices, columns=cols)
print_header('DataFrame df')
print(df, '\n')

print_header("df.iloc[1, 3]")  # one value
print(df.iloc[1, 3], "\n")


print_header("df.iloc[1]")  # one row
print(df.iloc[1], '\n')

print_header("df.iloc[:,3]")  # one column
print(df.iloc[:, 3], '\n')


print_header("df.iloc[1: 3]")  # range of rows
print(df.iloc[1:3, :], '\n')
print(df.iloc[1:3], '\n')   # shorter version

print_header("df.iloc[:,1:3]")  # range of columns
print(df.iloc[:, 1:3], "\n")

print_header("df.iloc[1:3, 1:3]")  # ranges of rows and columns
print(df.iloc[1:3, 1:3], '\n')

print_header("df.iloc[[1, 4, 0]]")  # iterable of rows
print(df.iloc[[1, 4, 0]], "\n")
```



```
print_header("df.iloc[:, [2, 0, 4]]")  # iterable of columns
print(df.iloc[:, [2, 0, 4]], "\n")

print_header("df.iloc[[1, 4, 0], [2, 0, 4]]")  # iterables of rows and columns
print(df.iloc[[1, 4, 0], [2, 0, 4]], "\n")
```

pandas_iloc.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                 df.iloc[1, 3]                  =
==================================================
230

==================================================
=                   df.iloc[1]                   =
==================================================
alpha      200
beta       210
gamma      220
delta      230
epsilon    240
Name: b, dtype: int64

==================================================
=                  df.iloc[:,3]                  =
==================================================
a    130
b    230
c    330
d    430
e    530
f    630
Name: delta, dtype: int64
```

```
==================================================
=                 df.iloc[1: 3]                  =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340

   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340

==================================================
=                 df.iloc[:,1:3]                 =
==================================================
   beta  gamma
a   110    120
b   210    220
c   310    320
d   410    420
e   510    520
f   610    620

==================================================
=               df.iloc[1:3, 1:3]                =
==================================================
   beta  gamma
b   210    220
c   310    320

==================================================
=               df.iloc[[1, 4, 0]]               =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
e    500   510    520    530      540
a    100   110    120    130      140

==================================================
=             df.iloc[:, [2, 0, 4]]              =
==================================================
   gamma  alpha  epsilon
a    120    100      140
b    220    200      240
c    320    300      340
d    420    400      440
e    520    500      540
f    620    600      640
```

```
==================================================
=         df.iloc[[1, 4, 0], [2, 0, 4]]          =
==================================================
   gamma  alpha  epsilon
b    220    200      240
e    520    500      540
a    120    100      140
```

Broadcasting

- Operation is applied across rows and columns
- Can be restricted to selected rows/columns
- Sometimes called vectorization
- Use apply() for more complex operations

If you multiply a dataframe by some number, the operation is broadcast, or vectorized, across all values. This is true for all basic math operations.

The operation can be restricted to selected columns.

For more complex operations, the apply() method will apply a function that selects elements. You can use the name of an existing function, or supply a lambda (anonymous) function.

Example

pandas_broadcasting.py

```
import pandas as pd
from printheader import print_header

column_labels = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column labels
row_labels = pd.date_range('2013-01-01 00:00:00', periods=6, freq='D')  # date range to
be used as row indexes

print(row_labels, "\n")

values = [  # sample data
    [100, 110, 120, 930, 140],
    [250, 210, 120, 130, 840],
    [300, 310, 520, 430, 340],
    [275, 410, 420, 330, 777],
    [300, 510, 120, 730, 540],
    [150, 610, 320, 690, 640],
]

df = pd.DataFrame(values, row_labels, column_labels)  # create dataframe from data
print_header("Basic DataFrame:")
print(df)
print()

print_header("Triple each value")
print(df * 3)
print()  # multiply every value by 3

print_header("Multiply column gamma by 1.5")
df['gamma'] *= 1.5  # multiply values in column 'gamma' by 1.
print(df)
print()
```

pandas_broadcasting.py

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

==================================================
=                Basic DataFrame:                =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110    120    930      140
2013-01-02    250   210    120    130      840
2013-01-03    300   310    520    430      340
2013-01-04    275   410    420    330      777
2013-01-05    300   510    120    730      540
2013-01-06    150   610    320    690      640

==================================================
=               Triple each value                =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    300   330    360   2790      420
2013-01-02    750   630    360    390     2520
2013-01-03    900   930   1560   1290     1020
2013-01-04    825  1230   1260    990     2331
2013-01-05    900  1530    360   2190     1620
2013-01-06    450  1830    960   2070     1920

==================================================
=          Multiply column gamma by 1.5          =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110  180.0    930      140
2013-01-02    250   210  180.0    130      840
2013-01-03    300   310  780.0    430      340
2013-01-04    275   410  630.0    330      777
2013-01-05    300   510  180.0    730      540
2013-01-06    150   610  480.0    690      640
```


Counting unique occurrences

- Use `.value_counts()`
- Called from column

To count the unique occurrences within a column, call the method `value_counts()` on the column. It returns a `Series` object with the column values and their counts.

Example

pandas_unique.py

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_excel('https://qrc.depaul.edu/Excel_Files/Presidents.xlsx',
sheet_name='Master',
                  na_values='NA()')
df.index = range(1,len(df)+1)

print(df.head())
print(df.loc[1])
party_counts = df['Political Party'].value_counts()
print(party_counts)
# plot the data
plt.figure(figsize=(20.0,8.0))
party_counts.plot(kind='barh')
plt.show()
```

Creating new columns

- Assign to column with new name
- Use normal operators with other columns

For simple cases, it's easy to create new columns. Just assign a Series-like object to a new column name. The easy way to do this is to combine other columns with an operator or function.

Example

pandas_new_columns.py

```
import pandas as pd

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=cols)

def times_ten(x):
    return x * 10

df['zeta'] = df['delta'] * df['epsilon'] # product of two columns
df['eta'] = times_ten(df.alpha) # user-defined function
df['theta'] = df.sum(axis=1)  # sum each row
df['iota'] = df.mean(axis=1)  # avg of each row
df['kappa'] = df.loc[:,'alpha':'epsilon'].mean(axis=1)
# column kappa is avg of selected columns

print(df)
```


pandas_new_columns.py

```
   alpha  beta  gamma  delta  epsilon    zeta   eta   theta      iota  kappa
a    100   110    120    130      140   18200  1000   19800    4950.0  120.0
b    200   210    220    230      240   55200  2000   58300   14575.0  220.0
c    300   310    320    330      340  112200  3000  116800   29200.0  320.0
d    400   410    420    430      440  189200  4000  195300   48825.0  420.0
e    500   510    520    530      540  286200  5000  293800   73450.0  520.0
f    600   610    620    630      640  403200  6000  412300  103075.0  620.0
```

Removing entries

- Remove rows or columns
- Use drop() method

To remove columns or rows, use the `drop()` method, with the appropriate labels. Use `axis=1` to drop columns, or axis=0 to drop rows.

Example

pandas_drop.py

```
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('values:')
print(values, '\n\n')

df = pd.DataFrame(values, index=index, columns=cols)  # create dataframe
print_header('DataFrame df')
print(df, '\n')

df2 = df.drop(['beta', 'delta'], axis=1)  # drop columns beta and delta (axes: 0=rows,
1=columns)
print_header("After dropping beta and delta:")
print(df2, '\n')

print_header("After dropping rows b, c, and e")
df3 = df.drop(['b', 'c', 'e'])  # drop rows b, c, and e
print(df3)

print_header(" In-place drop")
df.drop(['beta', 'gamma'], axis=1, inplace=True)
print(df)
df.drop(['b', 'c'], inplace=True)
print(df)
```

pandas_drop.py

```
==================================================
=                    values:                     =
==================================================
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340], [400,
410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]



==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=         After dropping beta and delta:         =
==================================================
   alpha  gamma  epsilon
a    100    120      140
b    200    220      240
c    300    320      340
d    400    420      440
e    500    520      540
f    600    620      640

==================================================
=        After dropping rows b, c, and e         =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
d    400   410    420    430      440
f    600   610    620    630      640
==================================================
=                  In-place drop                 =
==================================================
   alpha  delta  epsilon
a    100    130      140
b    200    230      240
c    300    330      340
d    400    430      440
e    500    530      540
```



```
f    600    630      640
   alpha  delta  epsilon
a    100    130      140
d    400    430      440
e    500    530      540
f    600    630      640
```

Useful pandas methods

Table 3. Methods and attributes for fetching DataFrame/Series data

Method	Description
`DF.columns()`	Get or set column labels
`DF.shape()` `S.shape()`	Get or set shape (length of each axis)
`DF.head(n)` `DF.tail(n)`	Return n items (default 5) from beginning or end
`DF.describe()` `S.describe()`	Display statistics for dataframe
`DF.info()`	Display column attributes
`DF.values` `S.values`	Get the actual values from a data structure
`DF.loc[row_indexer`[1]`, col_indexer]`	Multi-axis indexing by label (not by position)
`DF.iloc[row_indexer`[2]`, col_indexer]`	Multi-axis indexing by position (not by labels)

[1] Indexers can be label, slice of labels, or iterable of labels.

[2] Indexers can be numeric index (0-based), slice of indexes, or iterable of indexes.

Table 4. Methods for Computations/Descriptive Stats (called from pandas)

Method	Returns
`abs()`	absolute values
`corr()`	pairwise correlations
`count()`	number of values
`cov()`	Pairwise covariance
`cumsum()`	cumulative sums
`cumprod()`	cumulative products
`cummin(), cummax()`	cumulative minimum, maximum
`kurt()`	unbiased kurtosis
`median()`	median
`min(), max()`	minimum, maximum values
`prod()`	products
`quantile()`	values at given quantile
`skew()`	unbiased skewness
`std()`	standard deviation
`var()`	variance

NOTE these methods return Series or DataFrames, as appropriate, and can be computed over rows (axis=0) or columns (axis=1). They generally skip NA/null values.

Even more pandas…

At this point, please load the following Jupyter notebooks for more pandas exploration:

- pandas_Demo.ipynb
- pandas_Input_Demo.ipynb
- pandas_Selection_Demo.ipynb

NOTE The instructor will explain how to start the Jupyter server.

Chapter 2 Exercises

Exercise 2-1 (add_columns.py)

Read in the file **sales_records.csv** as shown in the early part of the chapter. Add three new columns to the dataframe:

- Total Revenue (*units sold x unit price*)
- Total Cost (*units sold x unit cost*)
- Total Profit (*total revenue - total cost*)

Exercise 2-2 (parasites.py))

The file parasite_data.csv, in the DATA folder, has some results from analysis on some intestinal parasites (not that it matters for this exercise…). Read parasite_data.csv into a DataFrame. Print out all rows where the Shannon Diversity is >= 1.0.

Chapter 3: Serializing Data

Objectives

- Have a good understanding of the XML format
- Know which modules are available to process XML
- Use lxml ElementTree to create a new XML file
- Parse an existing XML file with ElementTree
- Using XPath for searching XML nodes
- Load JSON data from strings or files
- Write JSON data to strings or files
- Read and write CSV data
- Read and write YAML data

Which XML module to use?

- Bewildering array of XML modules
- Some are SAX, some are DOM
- Use xml.etree.ElementTree

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, use **lxml.etree**, which is based on **ElementTree** with some nice extra features, such as pretty-printing. While not part of the core Python library, it is provided by the Anaconda bundle.

If `lxml.etree` is not available, you can use **xml.etree.ElementTree** from the core library.

Getting Started With ElementTree

- Import xml.etree.ElementTree (or lxml.etree) as ET for convenience
- Parse XML or create empty ElementTree

ElementTree is part of the Python standard library; lxml is included with the Anaconda distribution.

Since putting "xml.etree.ElementTree" in front of its methods requires a lot of extra typing , it is typical to alias xml.etree.ElementTree to just ET when importing it: import xml.etree.ElementTree as ET

You can check the version of ElementTree via the VERSION attribute:

```
import xml.etree.ElementTree as ET
print(ET.VERSION)
```


How ElementTree Works

- ElementTree contains root Element
- Document is tree of Elements

In ElementTree, an XML document consists of a nested tree of Element objects. Each Element corresponds to an XML tag.

An ElementTree object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use ElementTree.parse(); this creates the ElementTree wrapper and the tree of Elements. You can then navigate to, or search for, Elements within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

```
element = root.find('sometag')
for subelement in element:
    print(subelement.tag)
print(element.get('someattribute'))
```

Elements

- Element has
  - Tag name
  - Attributes (implemented as a dictionary)
  - Text
  - Tail
  - Child elements (implemented as a list) (if any)
- SubElement creates child of Element

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** class is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the get() method).

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is any text following the element, before the next element.

The **SubElement** class is a convenient way to add children to an existing Element.

TIP Only the tag property of an Element is required; other properties are optional.

Table 5. Element methods and properties

Method/Property	Description
`append(element)`	Add a subelement element to end of subelements
`attrib`	Dictionary of element's attributes
`clear()`	Remove all subelements
`find(path)`	Find first subelement matching path
`findall(path)`	Find all subelements matching path
`findtext(path)`	Shortcut for `find(path).text`
`get(attr)`	Get an attribute; Shortcut for `attrib.get()`
`getiterator()`	Returns an iterator over all descendants
`getiterator(path)`	Returns an iterator over all descendants matching path
`insert(pos,element)`	Insert subelement element at position pos
`items()`	Get all attribute values; Shortcut for `attrib.items()`
`keys()`	Get all attribute names; Shortcut for `attrib.keys()`
`remove(element)`	Remove subelement element
`set(attrib,value)`	Set an attribute value; shortcut for `attr[attrib] = value`
`tag`	The element's tag
`tail`	Text following the element
`text`	Text contained within the element

Table 6. ElementTree methods and properties

Property	Description
`find(path)`	Finds the first toplevel element with given tag; shortcut for getroot().find(path).
`findall(path)`	Finds all toplevel elements with the given tag; shortcut for getroot().findall(path).
`findtext(path)`	Finds element text for first toplevel element with given tag; shortcut for getroot().findtext(path).
`getiterator(path)`	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
`getroot()`	Return the root node of the document
`parse(filename)` `parse(fileobj)`	Parse an XML source (filename or file-like object)
`write(filename,encoding)`	Writes XML document to filename, using encoding (Default us-ascii).

Creating a New XML Document

- Create root element
- Add descendants via SubElement
- Use keyword arguments for attributes
- Add text after element created
- Create ElementTree for import/export

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once the root element is created, use SubElement to add elements to the root element, and then nested Elements as needed. SubElement returns the new element, so you can assign the contents of the tag to the **text** attribute.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text. From the ElementTree object, call write.

To output an XML string from your elements, call ET.tostring(), passing the root of the element tree as a parameter. It will return a bytes object (pure ASCII), so use .decode() to convert it to a normal Python string.

For an example of creating an XML document from a data file, see **xml_create_knights.py** in the EXAMPLES folder

Example

xml_create_movies.py

```
# from xml.etree import ElementTree as ET
import lxml.etree as ET

movie_data = [
    ('Jaws', 'Spielberg, Stephen'),
    ('Vertigo', 'Alfred Hitchcock'),
    ('Blazing Saddles', 'Brooks, Mel'),
    ('Princess Bride', 'Reiner, Rob'),
    ('Avatar', 'Cameron, James'),
]

movies = ET.Element('movies')

for name, director in movie_data:
    movie = ET.SubElement(movies, 'movie', name=name)
    ET.SubElement(movie, 'director').text = director

print(ET.tostring(movies, pretty_print=True).decode())

doc = ET.ElementTree(movies)

doc.write('movies.xml')
```


xml_create_movies.py

```
<movies>
  <movie name="Jaws">
    <director>Spielberg, Stephen</director>
  </movie>
  <movie name="Vertigo">
    <director>Alfred Hitchcock</director>
  </movie>
  <movie name="Blazing Saddles">
    <director>Brooks, Mel</director>
  </movie>
  <movie name="Princess Bride">
    <director>Reiner, Rob</director>
  </movie>
  <movie name="Avatar">
    <director>Cameron, James</director>
  </movie>
</movies>
```

Parsing An XML Document

- Use ElementTree.parse()
- returns an ElementTree object
- Use get* or find* methods to select an element

Use the parse() method to parse an existing XML document. It returns an ElementTree object, from which you can find the root, or any other element within the document.

To get the root element, use the getroot() method.

Example

```
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

root = doc.getroot()
```


Navigating the XML Document

- Use find() or findall()
- Element is iterable of it children
- findtext() retrieves text from element

To find the first child element with a given tag, use find('tag'). This will return the first matching element. The findtext('tag') method is the same, but returns the text within the tag.

To get all child elements with a given tag, use the findall('tag') method, which returns a list of elements.

to see whether a node was found, say

```
if node is None:
```

but to check for existence of child elements, say

```
if len(node) > 0:
```

A node with no children tests as false because it is an empty list, but it is not None.

TIP The ElementTree object also supports the find() and findall() methods of the Element object, searching from the root object.

Example

xml_planets_nav.py

```
'''Use etree navigation to extract planets from solar.xml'''
import lxml.etree as ET


def main():
    '''Program entry point'''
    doc = ET.parse('../DATA/solar.xml')

    solar_system = doc.getroot()

    print(solar_system)
    print()

    inner = solar_system.find('innerplanets')
    print('Inner:')

    for planet in inner:
        if planet.tag == 'planet':
            print('\t', planet.get("planetname", "NO NAME"))

    outer = solar_system.find('outerplanets')
    print('Outer:')

    for planet in outer:
        print('\t', planet.get("planetname"))

    plutoids = solar_system.find('dwarfplanets')
    print('Dwarf:')

    for planet in plutoids:
        print('\t', planet.get("planetname"))


if __name__ == '__main__':
    main()
```


xml_planets_nav.py

```
<Element solarsystem at 0x7fb928034d00>

Inner:
     Mercury
     Venus
     Earth
     Mars
Outer:
     Jupiter
     Saturn
     Uranus
     Neptune
Dwarf:
     Pluto
```

Example

xml_read_movies.py

```
# import xml.etree.ElementTree as ET
import lxml.etree as ET

movies_doc = ET.parse('movies.xml')  # read and parse the XML file

movies = movies_doc.getroot()  # get the root element (<movies>)

for movie in movies:  # loop through children of root element
    print('{} by {}'.format(
        movie.get('name'),  # get 'name' attribute of movie element
        movie.findtext('director'),  # get 'director' attribute of movie element
    )
    )
```

xml_read_movies.py

```
Jaws by Spielberg, Stephen
Vertigo by Alfred Hitchcock
Blazing Saddles by Brooks, Mel
Princess Bride by Reiner, Rob
Avatar by Cameron, James
```


Using XPath

- Use simple XPath patterns Works with find* methods

When a simple tag is specified, the find* methods only search for subelements of the current element. For more flexible searching, the find* methods work with simplified **XPath** patterns. To find all tags named 'spam', for instance, use `.//spam`.

```
.//movie
presidents/president/name/last
```

Example

xml_planets_xpath1.py

```python
# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')  # parse XML file

inner_nodes = doc.findall('innerplanets/planet')  # find all elements (relative to root element) with tag "planet" under "innerplanets" element

outer_nodes = doc.findall('outerplanets/planet')  # find all elements with tag "planet" under "outerplanets" element

print('Inner:')
for planet in inner_nodes:  # loop through search results
    print('\t', planet.get("planetname"))  # print "name" attribute of planet element

print('Outer:')
for planet in outer_nodes:  # loop through search results
    print('\t', planet.get("planetname"))  # print "name" attribute of planet element
```

xml_planets_xpath1.py

```
Inner:
     Mercury
     Venus
     Earth
     Mars
Outer:
     Jupiter
     Saturn
     Uranus
     Neptune
```

Example

xml_planets_xpath2.py

```
# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')

jupiter = doc.find('.//planet[@planetname="Jupiter"]')

if jupiter is not None:
    for moon in jupiter:
        print(moon.text)  # grab attribute
```

xml_planets_xpath2.py

```
Metis
Adrastea
Amalthea
Thebe
Io
Europa
Gannymede
Callista
Themisto
Himalia
Lysithea
Elara
```


Table 7. ElementTree XPath Summary

Syntax	Meaning
`tag`	Selects all child elements with the given tag. For example, “spam” selects all child elements named “spam”, “spam/egg” selects all grandchildren named “egg” in all child elements named “spam”. You can use universal names (“{url}local”) as tags.
`*`	Selects all child elements. For example, “*/egg” selects all grandchildren named “egg”.
`.`	Select the current node. This is mostly useful at the beginning of a path, to indicate that it’s a relative path.
`//`	Selects all subelements, on all levels beneath the current element (search the entire subtree). For example, “.//egg” selects all “egg” elements in the entire tree.
`..`	Selects the parent element.
`[@attrib]`	Selects all elements that have the given attribute. For example, “.//a[@href]” selects all “a” elements in the tree that has a “href” attribute.
`[@attrib='value']`	Selects all elements for which the given attribute has the given value. For example, “.//div[@class=’sidebar’]” selects all “div” elements in the tree that has the class “sidebar”. In the current release, the value cannot contain quotes.
`parent_tag[child_tag]`	Selects all parent elements that has a child element named *child_tag*. In the current version, only a single tag can be used (i.e. only immediate children are supported). Parent tag can be *****.

About JSON

- Lightweight, human-friendly format for data
- Contains dictionaries and lists
- Stands for JavaScript Object Notation
- Looks like Python
- Basic types: Number, String, Boolean, Array, Object
- White space is ignored
- Stricter rules than Python

JSON is a lightweight and human-friendly format for sharing or storing data. It was developed and popularized by Douglas Crockford starting in 2001.

A JSON file contains objects and arrays, which correspond exactly to Python dictionaries and lists.

White space is ignored, so JSON may be formatted for readability.

Data types are Number, String, and Boolean. Strings are enclosed in double quotes (only); numbers look like integers or floats; Booleans are represented by true or false; null (None in Python) is represented by null.

Reading JSON

- json module in standard library
- json.load() parse from file-like object
- json.loads() parse from string
- Both methods return Python dict or list

To read a JSON file, import the json module. Use json.loads() to parse a string containing valid JSON. Use json.load() to read JSON from a file-like object0.

Both methods return a Python dictionary containing all the data from the JSON file.

Example

json_read.py

```
from pprint import pprint
import json

# json.loads(STRING)     load from string
# json.load(FILE_OBJECT) load from file-like object

with open('../DATA/solar.json') as solar_in:  # open JSON file for reading
    solar = json.load(solar_in)  # load from file object and convert to Python data
structure

# uncomment to see raw Python data
# print('-' * 60)
# pprint(solar)
# print('-' * 60)
# print('\n\n')

print(solar['innerplanets'])  # solar is just a Python dictionary
print('*' * 60)
print(solar['innerplanets'][0]['name'])
print('*' * 60)
for planet in solar['innerplanets'] + solar['outerplanets']:
    print(planet['name'])

print("*" * 60)
for group in solar:
    if group.endswith('planets'):
        for planet in solar[group]:
            print(planet['name'])
```


json_read.py

```
[{'name': 'Mercury', 'moons': None}, {'name': 'Venus', 'moons': None}, {'name': 'Earth',
'moons': ['Moon']}, {'name': 'Mars', 'moons': ['Deimos', 'Phobos']}]
************************************************************
Mercury
************************************************************
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
************************************************************
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto
```

Writing JSON

- Use json.dumps() or json.dump()

To output JSON to a string, use json.dumps(). To output JSON to a file, pass a file-like object to json.dump(). In both cases, pass a Python data structure as the data to be output.

Example

json_write.py

```
import json

george = [
    {
        'num': 1,
        'lname': 'Washington',
        'fname': 'George',
        'dstart': [1789, 4, 30],
        'dend': [1797, 3, 4],
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'dbirth': [1732, 2, 22],
        'ddeath': [1799, 12, 14],
        'assassinated': False,
        'party': None,
    },
    {
        'spam': 'ham',
        'eggs': [1.2, 2.3, 3.4],
        'toast': {'a': 5, 'm': 9, 'c': 4},
    }
]  # Python data structure

js = json.dumps(george, indent=4)  # dump structure to JSON string
print(js)

with open('george.json', 'w') as george_out:  # open file for writing
    json.dump(george, george_out, indent=4)  # dump structure to JSON file using open
file object
```


json_write.py

```
[
    {
        "num": 1,
        "lname": "Washington",
        "fname": "George",
        "dstart": [
            1789,
            4,
            30
        ],
        "dend": [
            1797,
            3,
            4
        ],
        "birthplace": "Westmoreland County",
        "birthstate": "Virginia",
        "dbirth": [
            1732,
            2,
            22
        ],
        "ddeath": [
            1799,
            12,
            14
        ],
        "assassinated": false,
        "party": null
    },
    {
        "spam": "ham",
        "eggs": [
            1.2,
            2.3,
            3.4
        ],
        "toast": {
            "a": 5,
            "m": 9,
            "c": 4
        }
    }
]
```

Customizing JSON

- JSON data types limited
- simple cases — dump dict
- create custom encoders

The JSON spec only supports a limited number of datatypes. If you try to dump a data structure contains dates, user-defined classes, or many other types, the json encoder will not be able to handle it.

You can a custom encoder for various data types. To do this, write a function that expects one Python object, and returns some object that JSON can parse, such as a string or dictionary. The function can be called anything. Specify the function with the **default** parameter to json.dump().

The function should check the type of the object. If it is a type that needs special handling, return a JSON-friendly version, otherwise just return the original object.

Table 8. Python types that JSON can encode

Python	JSON
`dict`	object
`list`	array
`str`	string
`int`	number (int)
`float`	number (real)
`True`	true
`False`	false
`None`	null

NOTE see the file **json_custom_singledispatch.py** in EXAMPLES for how to use the **singledispatch** decorator (in the **functools** module) to handle multiple data types.

Example

json_custom_encoding.py

```
import json
from datetime import date

class Parrot():  # sample user-defined class (not JSON-serializable)
    def __init__(self, name, color):
        self._name = name
        self._color = color

    @property
    def name(self):  # JSON does not understand arbitrary properties
        return self._name

    @property
    def color(self):
        return self._color

parrots = [  # list of Parrot objects
    Parrot('Polly', 'green'),  #
    Parrot('Peggy', 'blue'),
    Parrot('Roger', 'red'),
]

def encode(obj):  # custom JSON encoder function
    if isinstance(obj, date):  # check for date object
        return obj.ctime()  # convert date to string
    elif isinstance(obj, Parrot):  # check for Parrot object
        return {'name': obj.name, 'color': obj.color}  # convert Parrot to dictionary
    return obj  # if not processed, return object for JSON to parse with default parser


data = {  # dictionary of arbitrary data
    'spam': [1, 2, 3],
    'ham': ('a', 'b', 'c'),
    'toast': date(2014, 8, 1),
    'parrots': parrots,
}

# convert Python data to JSON data;
# 'default' parameter specifies function for custom encoding;
# 'indent' parameter says to indent and add newlines for readability
print(json.dumps(data, default=encode, indent=4))
```

json_custom_encoding.py

```
{
    "spam": [
        1,
        2,
        3
    ],
    "ham": [
        "a",
        "b",
        "c"
    ],
    "toast": "Fri Aug  1 00:00:00 2014",
    "parrots": [
        {
            "name": "Polly",
            "color": "green"
        },
        {
            "name": "Peggy",
            "color": "blue"
        },
        {
            "name": "Roger",
            "color": "red"
        }
    ]
}
```


Reading and writing YAML

- yaml module from PYPI
- syntax like **json** module
- yaml.load(), dump() parse from/to file-like object
- yaml.loads(), dumps() parse from/to string

YAML is a structured data format which is a superset of JSON. However, YAML allows for a more compact and readable format.

Reading and writing YAML uses the same syntax as JSON, other than using the **yaml** module, which is NOT in the standard library. To install the **yaml** module:

```
pip install pyyaml
```

To read a YAML file (or string) into a Python data structure, use `yaml.load(file_object)` or `yaml.loads(string)`.

To write a data structure to a YAML file or string, use `yaml.dump(data, file_object)` or `yaml.dumps(data)`.

You can also write custom YAML processors.

NOTE YAML parsers will parse JSON data

Example

yaml_read_solar.py

```
import yaml

PLANET_SECTIONS = "inner outer plutoid".split()

with open('../DATA/solar.yaml') as solar_in:
    solar_data = yaml.load(solar_in, Loader=yaml.FullLoader)

star = solar_data['star']
print("Our star is {}\n".format(star))

for section in PLANET_SECTIONS:
    for planet in solar_data[section]:
        print(planet['name'])
        for moon in planet['moons']:
            print("\t{}".format(moon))
```

yaml_read_solar.py

```
Our star is Sun

Mercury
    None
Venus
    None
Earth
    Moon
Mars
    Deimos
    Phobos
    Metis
Jupiter
    Adrastea
    Amalthea
    Thebe
    Io
    Europa
    Gannymede
    Callista
    Themisto
    Himalia
    Lysithea
    Elara
Saturn
    Rhea
    Hyperion
    Titan
    Iapetus
    Mimas
```

…

Example

yaml_create_file.py

```
import sys
from datetime import date
import yaml

potus = {
    'presidents': [
         {
            'lastname': 'Washington',
            'firstname': 'George',
            'dob': date(1732, 2, 22),
            'dod': date(1799, 12, 14),
            'birthplace': 'Westmoreland County',
            'birthstate': 'Virginia',
            'term': [ date(1789, 4, 30), date(1797, 3, 4) ],
            'assassinated': False,
            'party': None,
        },
        {
            'lastname': 'Adams',
            'firstname': 'John',
            'dob': date(1735, 10, 30),
            'dod': date(1826, 7, 4),
            'birthplace': 'Braintree, Norfolk',
            'birthstate': 'Massachusetts',
            'term': [date(1797, 3, 4), date(1801, 3, 4)],
            'assassinated': False,
            'party': 'Federalist',

        }
    ]
}

with open('potus.yaml', 'w') as potus_out:
    yaml.dump(potus, potus_out)

yaml.dump(potus, sys.stdout)
```


yaml_create_file.py


```
presidents:
- assassinated: false
  birthplace: Westmoreland County
  birthstate: Virginia
  dob: 1732-02-22
  dod: 1799-12-14
  firstname: George
  lastname: Washington
  party: null
  term:
  - 1789-04-30
  - 1797-03-04
- assassinated: false
  birthplace: Braintree, Norfolk
  birthstate: Massachusetts
  dob: 1735-10-30
  dod: 1826-07-04
  firstname: John
  lastname: Adams
  party: Federalist
  term:
  - 1797-03-04
  - 1801-03-04
```

Reading CSV data

- Use `csv` module
- Create a reader with file object or any iterable
- Iterate through reader to get rows as lists of columns

To read CSV data, create an instance of the `reader` class from the `csv` module. Pass in an iterable – typically, but not necessarily, a file object. (A file object is the object returned by `open()`).

TIP You can pass in parameters to customize the input or output data.

Example

csv_read.py

```
import csv

with open('../DATA/knights.csv') as knights_in:
    rdr = csv.reader(knights_in)  # create CSV reader
    for name, title, color, quest, comment, number, ladies in rdr:  # Read and unpack
records one at a time; each record is a list
        print('{:4s} {:9s} {}'.format(
            title, name, quest
        ))
```

csv_read.py

```
King Arthur    The Grail
Sir  Lancelot  The Grail
Sir  Robin     Not Sure
Sir  Bedevere  The Grail
Sir  Gawain    The Grail
```

...

Customizing CSV readers and writers

- Variations in how CSV data is written
- Most common alternate is for Excel
- Add parameters to reader/writer

You can customize how the CSV parser and generator work by passing extra parameters to `csv.reader()` or `csv.writer()`. You can change the field and row delimiters, the escape character, and for output, what level of quoting. This can be used for any text file, not just CSV formats.

You can also specify a *dialect*, which is a custom set of CSV parameters. TO create a custom dialect, use `csv.register_dialect()`.

Example

csv_dialects.py

```
import csv

csv.register_dialect('colon-sep', delimiter=":")

with open('../DATA/knights.txt') as knights_in:
    reader = csv.reader(knights_in, dialect="colon-sep")
    for row in reader:
        print(row)
print()

with open('../DATA/primeministers.txt') as pm_in:
    reader = csv.reader(pm_in, dialect="colon-sep")
    for row in reader:
        print(row)
```

csv_dialects.py

```
['Arthur', 'King', 'blue', 'The Grail', 'King of the Britons']
['Galahad', 'Sir', 'red', 'The Grail', "'I could handle some more peril'"]
['Lancelot', 'Sir', 'blue', 'The Grail', "It's too perilous!"]
['Robin', 'Sir', 'yellow', 'Not Sure', 'He boldly ran away']
['Bedevere', 'Sir', 'red, no blue!', 'The Grail', 'AARRRRRRRGGGGHH']
['Gawain', 'Sir', 'blue', 'The Grail', 'none']

['1', 'Sir John A.', 'Macdonald', '1867-7-1', '1873-11-5', 'Glasgow, Scotland', '1867-07-
01', '1873-11-05', 'Liberal-Conservative']
['2', 'Alexander', 'Mackenzie', '1873-11-7', '1878-10-8', 'Logierait, Scotland', '1873-
11-07', '1878-10-08', 'Liberal']
['3', 'Sir John A.', 'Macdonald', '1878-10-17', '1891-6-6', 'Glasgow, Scotland', '1878-
10-17', '1891-06-06', 'Liberal-Conservative']
['4', 'Sir John', 'Albott', '1891-6-16', '1892-11-24', "Saint-Andre-d'Argenteuil,
Quebec", '1891-06-16', '1892-11-24', 'Liberal-Conservative']
['5', 'Thompson', 'Sir John', '1892-12-5', '1894-12-12', 'Halifax, Nova Scotia', '1892-
12-05', '1894-12-12', 'Conservative']
['6', 'Sir Mackenzie', 'Bowell', '1894-12-21', '1896-4-27', 'Rickinghall, England',
'1894-12-21', '1896-04-27', 'Conservative']
['7', 'Sir Charles', 'Tupper', '1896-5-1', '1896-7-8', 'Amherst, Nova Scotia', '1896-05-
01', '1896-07-08', 'Conservative']
['8', 'Sir Wilfred', 'Laurier', '1896-7-11', '1911-10-6', 'Saint-Lin-Laurentides,
Quebec', '1886-07-11', '1911-10-06', 'Liberal']
['9', 'Sir Robert', 'Borden', '1911-10-10', '1917-10-12', 'Grand-Pre, Nova Scotia',
'1911-10-10', '1917-10-11', 'Conservative']
['10', 'Sir Robert', 'Borden', '1917-10-12', '1920-7-10', 'Grand-Pre, Nova Scotia',
'1917-10-12', '1920-07-10', 'Unionist']
['11', 'Arthur', 'Meighen', '1920-7-10', '1921-12-29', 'Perth South, Ontario', '1920-07-
10', '1921-12-29', 'NLC']
['12', 'William Lyon Mackenzie', 'King', '1921-12-29', '1926-6-29', 'Kitchener, Ontario',
'1921-12-29', '1926-06-28', 'Liberal']
['13', 'Arthur', 'Meighen', '1926-6-29', '1926-9-25', 'Perth South, Ontario', '1926-06-
29', '1926-09-25', 'Conservative']
```


Example

csv_nonstandard.py

```
import csv

with open('../DATA/computer_people.txt') as computer_people_in:
    rdr = csv.reader(computer_people_in, delimiter=';')  # specify alternate field
delimiter

    # iterate over rows of data -- csv reader is an iterator

    for first_name, last_name, known_for, birth_date in rdr:
        print('{}: {}'.format(last_name, known_for))
```

csv_nonstandard.py

```
Gates: Gates Foundation
Jobs: Apple
Wall: Perl
Allen: Microsoft
Ellison: Oracle
van Rossum: Python
Kurtz: BASIC
Hopper: COBOL
Gates: Microsoft
Zuckerberg: Facebook
Brin: Google
van Rossum: Python
Lovelace:
Page: Google
Torvalds: Linux
```

Table 9. CSV reader/writer Parameters

Parameter	Meaning
`quotechar`	One-character string to use as quoting character (default: `'"'`)
`delimiter`	One-character string to use as field separator (default: `','`)
`skipinitialspace`	If True, skip white space after field separator (default: `False`)
`lineterminator`	The character sequence which terminates rows (default: depends on OS)
`quoting`	When should quotes be generated when writing CSV `csv.QUOTE_MINIMAL` – only when needed (default) `csv.QUOTE_ALL` – quote all fields `csv.QUOTE_NONNUMERIC` – quote all fields that are not numbers `csv.QUOTE_NONE` – never put quotes around fields
`escapechar`	One-character string to escape delimiter when quoting is set to `csv.QUOTE_NONE`
`doublequote`	Control quote handling inside fields. When `True`, two consecutive quotes are read as one, and one quote is written as two. (default: `True`)
`dialect`	string representing registered dialect name, such as "excel"

Using csv.DictReader

- Returns each row as dictionary
- Keys are field names
- Use header or specify

Instead of the normal reader, you can create a dictionary-based reader by using the DictReader class.

If the CSV file has a header, it will parse the header line and use it as the field names. Otherwise, you can specify a list of field names with the **fieldnames** parameter. For each row, you can look up a field by name, rather than position.

Example

csv_dictreader.py

```
import csv

field_names = ['term', 'firstname', 'lastname', 'birthplace', 'state', 'party'] # field
names, which will become dictionary keys on each row

with open('../DATA/presidents.csv') as presidents_in:
    rdr = csv.DictReader(presidents_in, fieldnames=field_names)  # create reader, passing
in field names (if not specified, uses first row as field names)
    for row in rdr:  # iterate over rows in file
        print('{:25s} {:12s} {}'.format(row['firstname'], row['lastname'], row['party']))
# print results with formatting
        # string .format can use keywords from an unpacked dict as well:
        # print('{firstname:25s} {lastname:12s} {party}'.format(**row))
```

csv_dictreader.py

```
George                    Washington   no party
John                      Adams        Federalist
Thomas                    Jefferson    Democratic - Republican
James                     Madison      Democratic - Republican
James                     Monroe       Democratic - Republican
John Quincy               Adams        Democratic - Republican
Andrew                    Jackson      Democratic
Martin                    Van Buren    Democratic
William Henry             Harrison     Whig
John                      Tyler        Whig
James Knox                Polk         Democratic
Zachary                   Taylor       Whig
Millard                   Fillmore     Whig
Franklin                  Pierce       Democratic
James                     Buchanan     Democratic
Abraham                   Lincoln      Republican
Andrew                    Johnson      Republican
Ulysses Simpson           Grant        Republican
Rutherford Birchard       Hayes        Republican
James Abram               Garfield     Republican
```

...

Writing CSV Data

- Use csv.writer()
- Parameter is file-like object (must implement write() method)
- Can specify parameters to writer constructor
- Use writerow() or writerows() to output CSV data

To output data in CSV format, first create a writer using csv.writer(). Pass in a file-like object.

For each row to write, call the writerow() method of the writer, passing in an iterable with the values for that row.

To modify how data is written out, pass parameters to the writer.

TIP On Windows, to prevent double-spaced output, add `lineterminator='\n'` when creating a CSV writer.

Example

csv_write.py

```
import sys
import csv

chicago_data = [
    ['Name', 'Position Title', 'Department', 'Employee Annual Salary'],
    ['BONADUCE,  MICHAEL J', 'POLICE OFFICER', 'POLICE', '$80724.00'],
    ['MELLON,  MATTHEW J "Matt"', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FIERI,  JOHN J', 'FIREFIGHTER-EMT', 'FIRE', '$75342.00'],
    ['GALAHAD,  MERLE S', 'CLERK III', 'BUSINESS AFFAIRS', '$45828.00'],
    ['ORCATTI,  JENNIFER L', 'FIRE COMMUNICATIONS OPERATOR I', 'OEMC', '$63121.68'],
    ['ASHE,  JOHN W', 'FOREMAN OF MACHINISTS', 'AVIATION', '$96553.60'],
    ['SADINSKY BLAKE,  MICHAEL G', 'POLICE OFFICER', 'POLICE', '$78012.00'],
    ['GRANT,  CRAIG A', 'SANITATION LABORER', 'STREETS & SAN', '$69576.00'],
    ['MILLER,  JONATHAN D', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FRANK,  ARTHUR R','POLICE OFFICER/EXPLSV DETECT, K9 HNDLR','POLICE','$87918.00'],
    ['POVOTTI,  JAMES S "Jimmy P"', 'TRAFFIC CONTROL AIDE-HOURLY', 'OEMC', '$19167.20'],
    ['TRAWLER,  DANIEL J', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SCUBA,  ANDREW G', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SWINE,  MATTHEW W', 'SERGEANT', 'POLICE', '$99756.00'],
    ['''RYDER,  MYRTA T "Lil'Myrt"''', 'POLICE OFFICER', 'POLICE', '$83706.00'],
    ['KORSHAK,  ROMAN', 'PARAMEDIC', 'FIRE', '$75372.00']
]

with open('../TEMP/chi_data.csv', 'w') as chi_out:
    #  On Windows, output line terminator must be set to '\n'.
    #  While it's not needed on Linux/Mac, it doesn't cause any problems,
    #  so this keeps the code portable.
    wtr = csv.writer(chi_out, lineterminator='\n') # create CSV writer from file object that is opened
    for data_row in chicago_data:  # iterate over records from file
        data_row[0] = data_row[0].title()  # make first field title case rather than all uppercase
        data_row[-1] = data_row[-1].lstrip('$')  # strip leading $ from last field
        wtr.writerow(data_row) # write one row (of iterables) to output file
```

Pickle

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Python uses the pickle module for data serialization.

To create pickled data, use either pickle.dump() or pickle.dumps(). Both functions take a data structure as the first argument. dumps() returns the pickled data as a string. dump () writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use pickle.load(), which takes a file-like object that has been open for writing, or pickle.loads() which reads from a string. Both functions return the original data structure that had been pickled.

NOTE The syntax of the **json** module is based on the **pickle** module.

Example

pickling.py

```
import pickle
from pprint import pprint

# some data structures
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

values = [
    3/7, 1/9, 14.5
]

data = [  # list of data structures
    colors,
    airports,
    values,
]

with open('../TEMP/pickled_data.pic', 'wb') as pic_out:  # open pickle file for writing
in binary mode
    pickle.dump(data, pic_out)  # serialize data structures to pickle file

with open('../TEMP/pickled_data.pic', 'rb') as pic_in:  # open pickle file for reading in
binary mode
    pickled_data = pickle.load(pic_in)  # de-serialize pickle file back into data
structures

pprint(pickled_data)  # view data structures
```

pickling.py

```
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
```

Chapter 3 Exercises

Exercise 3-1 (xwords.py)

Using ElementTree, create a new XML file containing all the words that start with 'x' from words.txt. The root tag should be named 'words', and each word should be contained in a 'word' tag. The finished file should look like this:

```
<words>
   <word>xanthan</word>
   <word>xanthans</words>
   and so forth
</words>
```

Exercise 3-2 (xpresidents.py)

Use ElementTree to parse presidents.xml. Loop through and print out each president's first and last names and their state of birth.

Exercise 3-3 (jpresidents.py)

Rewrite xpresidents.py to parse presidents.json using the json module.

Exercise 3-4 (cpresidents.py)

Rewrite xpresidents.py to parse presidents.csv using the csv module.

Exercise 3-5 (pickle_potus.py)

Write a script which reads the data from presidents.csv into an dictionary where the key is the term number, and the value is another dictionary of data for one president.

Using the pickle module, Write the entire dictionary out to a file named presidents.pic.

Exercise 3-6 (unpickle_potus.py)

Write a script to open presidents.pic, and restore the data back into a dictionary.

Then loop through the array and print out each president's first name, last name, and party.

Index