



Python for Data Science

3

Welcome!



Welcome to the **Python for Data Science** class!

About the Program

Four Part Series, Four Hours Each

- Session 1: Getting Started with Python
- Session 2: Applying Python – The Basics
- **Session 3: Exploring Python Files, Dictionaries, Sets & methods**
- Session 4: Expanding Python – methods, Error Handling, Importing and OO Classes

Quick Logistics: Format, Q&A and Follow-On Materials / Hand-Outs

About Me: Ernesto Lee, Director of Emerging Technologies, Professor of Data Science

Today's Agenda: Session 3

Get started with Python! Learn how to apply dictionaries and sets, files, and methods with analytical use cases. We'll also explore some common use cases that are often used in Data Analysis and Data Science.

Topics We'll Explore Today:

- Dictionaries and Sets
- Files
- Methods
- Pandas Introduction



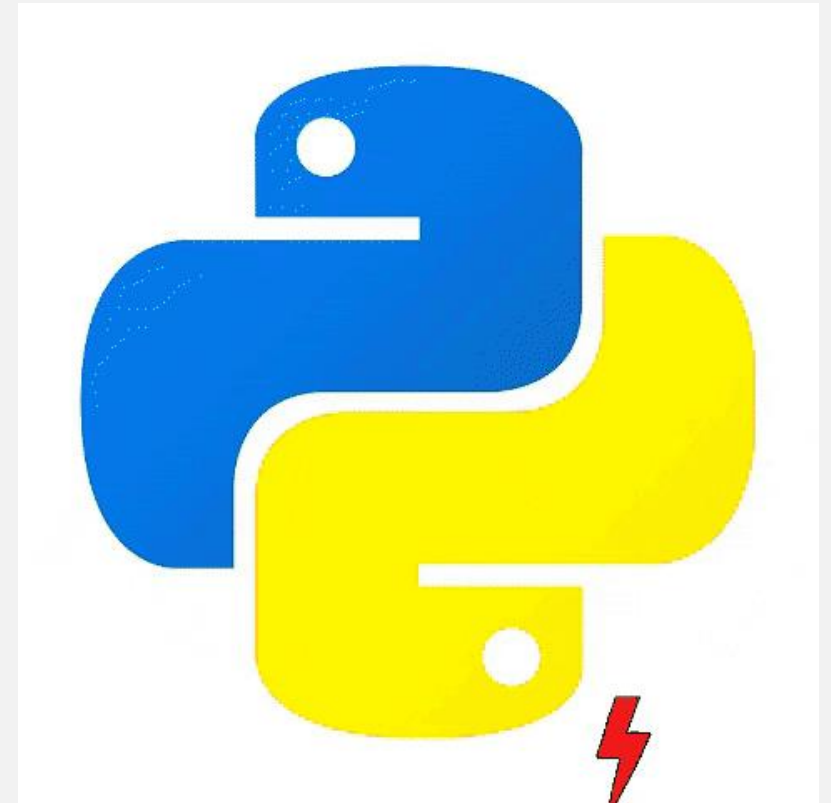


Files and Methods

Overview

In this lesson, you will learn

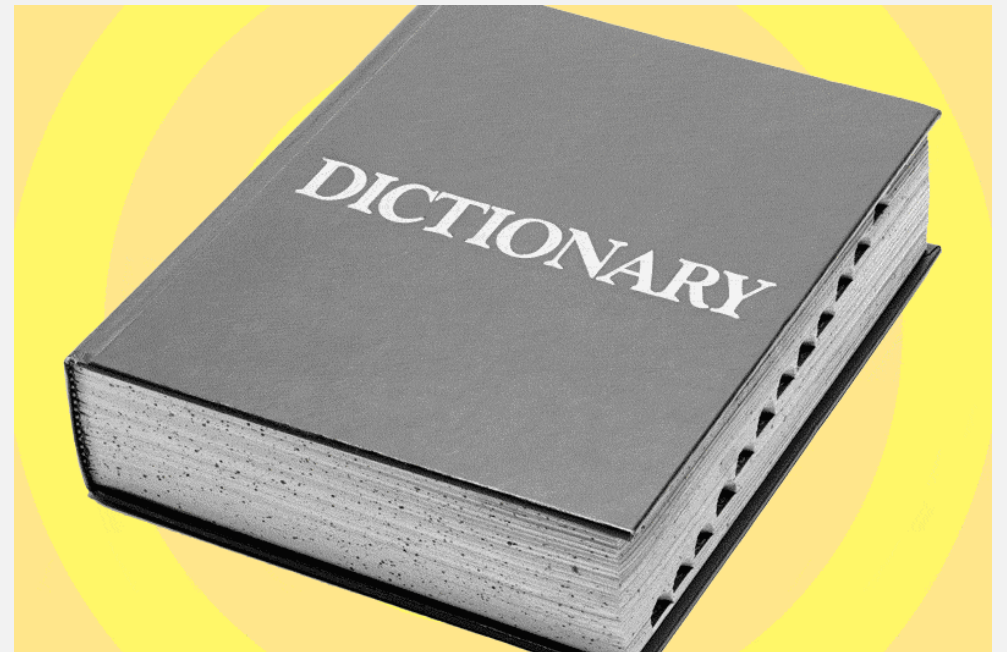
- Dictionaries / Sets
- Files
- Calling Methods
- Defining Methods
- Variable Scope
- Using a main Method
- Method Parameters
- Modules



Dictionary

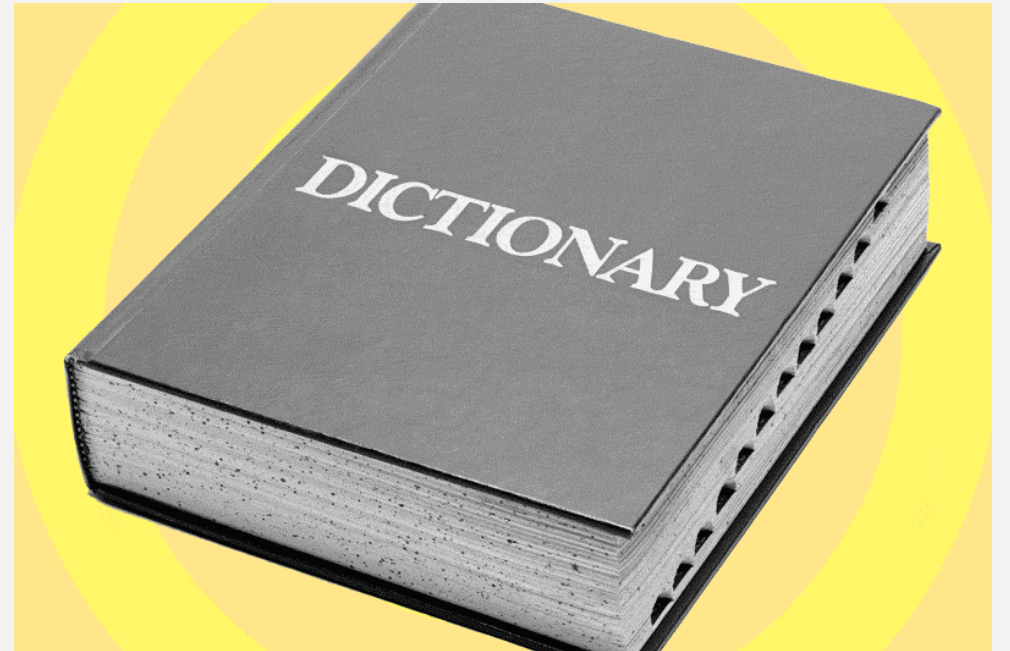
A Python dictionary is an extremely efficient data structure for storing pairs of values in the form key:value.

```
>>> color = {'red' : 1, 'blue' : 2, 'green' : 3}
>>> color
{'blue': 2, 'green': 3, 'red': 1}
>>> color['green']
3
>>> color['red']
1
```



Dictionary

```
>>> color = {'red' : 1, 'blue' : 2, 'green' : 3}
>>> color
{'blue': 2, 'green': 3, 'red': 1}
>>> color['red'] = 0
>>> color
{'blue': 2, 'green': 3, 'red': 0}
```



Dictionary Restrictions

keys are unique within the dictionary:

```
>>> color = {'red' : 1, 'blue' : 2, 'green' : 3, 'red' : 4}
>>> color
{'blue': 2, 'green': 3, 'red': 4}
```

Name	Return Value
<code>d.items()</code>	Returns a view of the (key, value) pairs in <code>d</code>
<code>d.keys()</code>	Returns a view of the keys of <code>d</code>
<code>d.values()</code>	Returns a view of the values in <code>d</code>
<code>d.get(key)</code>	Returns the value associated with <code>key</code>
<code>d.pop(key)</code>	Removes <code>key</code> and returns its corresponding value
<code>d.popitem()</code>	Returns some (key, value) pair from <code>d</code>
<code>d.clear()</code>	Removes all items from <code>d</code>
<code>d.copy()</code>	A copy of <code>d</code>
<code>d.fromkeys(s, t)</code>	Creates a new dictionary with keys taken from <code>s</code> and values taken from <code>t</code>
<code>d.setdefault(key, v)</code>	If <code>key</code> is in <code>d</code> , returns its value; if <code>key</code> is not in <code>d</code> , returns <code>v</code> and adds (<code>key</code> , <code>v</code>) to <code>d</code>
<code>d.update(e)</code>	Adds the (key, value) pairs in <code>e</code> to <code>d</code> ; <code>e</code> may be another dictionary or a sequence of pairs

Dictionaries are immutable

Sets

```
>>> lst = [1, 1, 6, 8, 1, 5, 5, 6, 8, 1, 5]
>>> s = set(lst)
>>> s
{8, 1, 5, 6}
```

Files



- A file is a named collection of bits stored on a secondary storage device, such as a hard disk, USB drive, flash memory stick, and so on.

Folders

- In addition to files, folders (or directories) are used to store files and other folders.
- The folder structure of most file systems is quite large and complex, forming a hierarchical folder structure.



Current Working Directory

- Many programs use the idea of a *current working directory*, or *cwd*.
- This is simply one directory that has been designated as the *default directory*.

Name	Action
<code>os.getcwd()</code>	Returns the name of the current working directory
<code>os.listdir(p)</code>	Returns a list of strings of the names of all the files and folders in the folder specified by path p
<code>os.chdir(p)</code>	Sets the current working directory to be path p
<code>os.path.isfile(p)</code>	Returns True just when path p specifies the name of a file, and False otherwise
<code>os.path.isdir(p)</code>	Returns True just when path p specifies the name of a folder, and False otherwise
<code>os.stat(fname)</code>	Returns information about fname , such as its size in bytes and the last modification time

Current Working Directory

```
# list.py  
def list_cwd():  
    return os.listdir(os.getcwd())
```

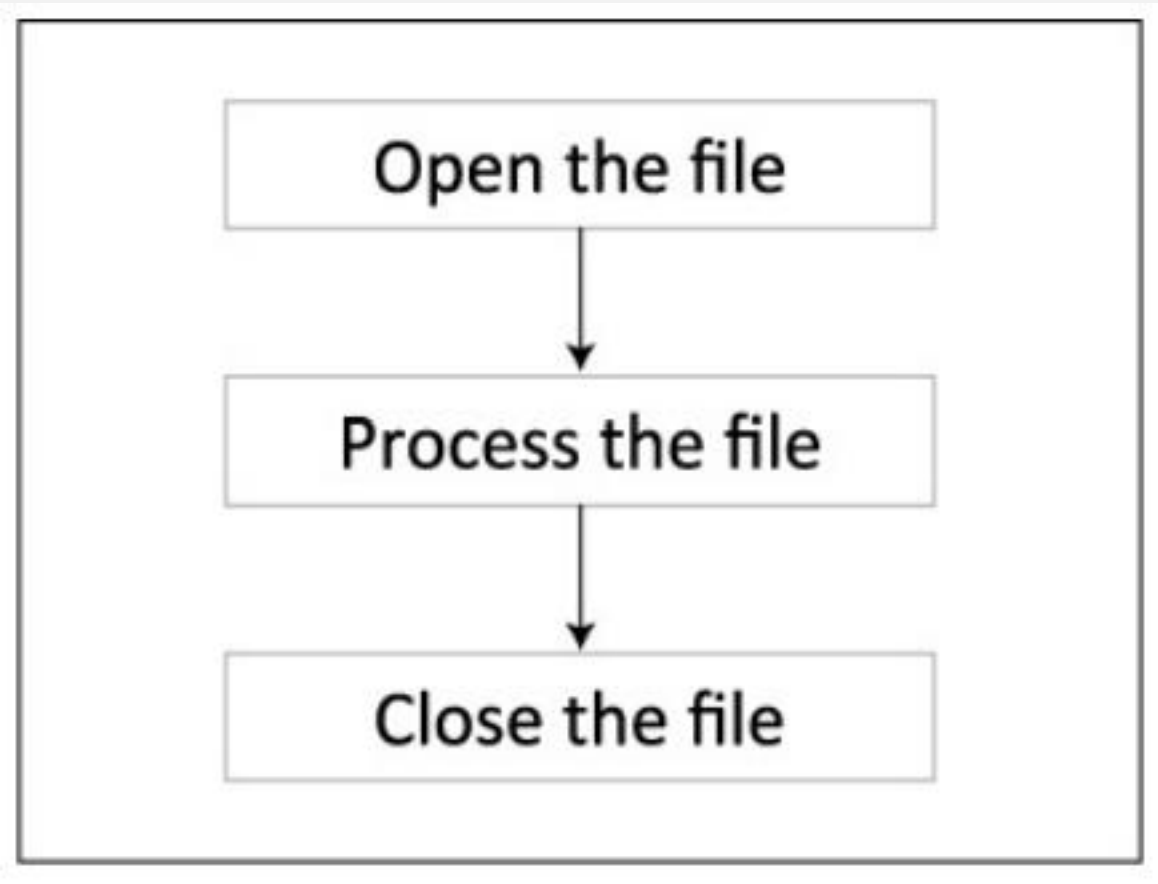
```
# list2.py  
def files_cwd():  
    return [p for p in list_cwd()  
        if os.path.isfile(p)]
```

```
def folders_cwd():  
    return [p for p in list_cwd()  
        if os.path.isdir(p)]
```

Current Working Directory

```
# list.py
def list_py(path = None):
    if path == None:
        path = os.getcwd()
    return [fname for fname in os.listdir(path)
            if os.path.isfile(fname)
            if fname.endswith('.py')]
```


Dealing with Text Files



Dealing with Text Files

```
# printfile.py
def print_file1(fname):
    f = open(fname, 'r')
    for line in f:
        print(line, end = "")
    f.close() # optional
```

Character	Meaning
'r'	Open for reading (default)
'w'	Open for writing
'a'	Open for appending to the end of the file
'b'	Binary mode
't'	Text mode (default)
'+'	Open a file for reading and writing

Dealing with Text Files

```
# printfile.py
def print_file2(fname):
    f = open(fname, 'r')
    print(f.read())
    f.close()
```

Writing to Text Files

```
# write.py
def make_story1():
    f = open('story.txt', 'w')
    f.write('Mary had a little
lamb,\n')
    f.write('and then she had some
more.\n')
```

```
# write2.py
import os
def make_story2():
    if os.path.isfile('story.txt'):
        print('story.txt already exists')
    else:
        f = open('story.txt', 'w')
        f.write('Mary had a little lamb,\n')
        f.write('and then she had some
more.\n')
```

Appending to Text Files

```
def add_to_story(line, fname = 'story.txt'):
    f = open(fname, 'a')
    f.write(line)
```

Appending to Text Files

```
def add_to_story(line, fname = 'story.txt'):
    f = open(fname, 'a')
    f.write(line)
```


Appending to the START of Text Files

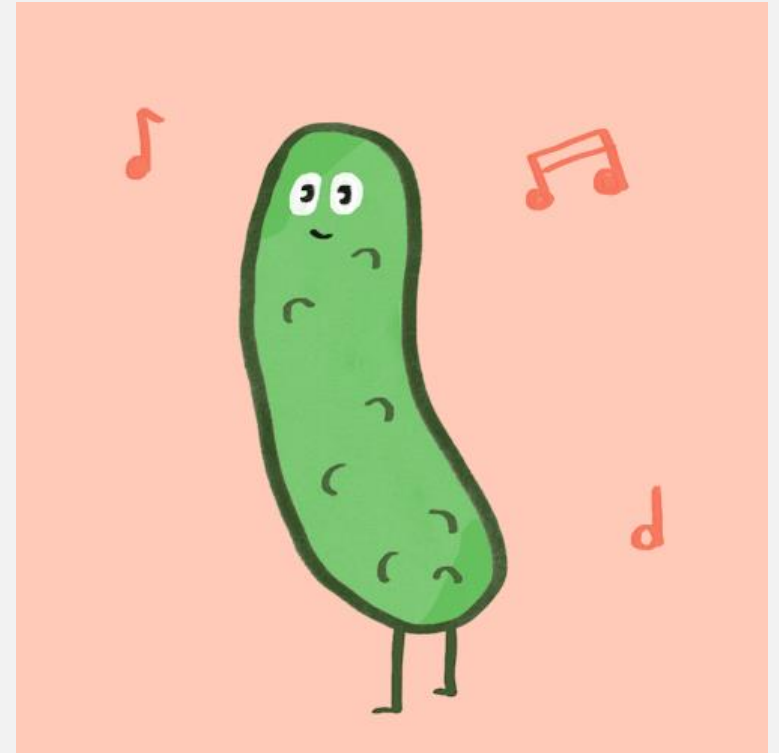
```
def add_to_story(line, fname = 'story.txt'):
    f = open(fname, 'a')
    f.write(line)
    f.seek(0) # reset file pointer
              # to beginning
```

Binary Files

```
def is_gif(fname):  
    f = open(fname, 'br')  
    first4 = tuple(f.read(4))  
    return first4 == (0x47, 0x49, 0x46, 0x38)
```

What is Pickling?

- Pickle is used for serializing and de-serializing Python object structures.
- Serialization refers to the process of converting an object in memory to a byte stream that can be stored on disk or sent over a network.
- Later on, this character stream can then be retrieved and de-serialized back to a Python object.
- Pickling is not to be confused with compression!
 - **Pickling** is the conversion of an object from one representation (data in Random Access Memory (RAM)) to another (text on disk)
 - While the latter is the process of encoding data with fewer bits, in order to save disk space.



What can be done with a Pickle?



Use Cases

- 1) saving a program's state data to disk so that it can carry on where it left off when restarted (persistence)
- 2) sending **python** data over a TCP connection in a multi-core or distributed system (marshalling)
- 3) storing **python** objects in a database
- 4) converting an arbitrary **python** object to a string so that it can be used as a dictionary key (e.g. for caching & memorization).

What can be pickled?

- **Booleans**
- **Integers**
- **Floats**
- **Complex numbers**
- **(normal and Unicode) Strings**
- **Tuples**
- **Lists**
- **Sets**
- **Dictionaries**



What can be pickled?

Pickle a File

```
import pickle
dogs_dict = { 'Ozzy': 3, 'Filou': 8, 'Luna': 5, 'Skippy': 10, 'Barco': 12, 'Balou': 9, 'Laika': 16 }
filename = 'dogs'
outfile = open(filename,'wb')
pickle.dump(dogs_dict,outfile)
outfile.close()
```

UnPickle a File

```
infile = open(filename,'rb')
new_dict = pickle.load(infile)
infile.close()
print(new_dict)
print(new_dict==dogs_dict)
print(type(new_dict))
```



Pickling

```
# picklefile.py
import pickle
def make_pickled_file():
    grades = {'alan' : [4, 8, 10, 10],
              'tom' : [7, 7, 7, 8],
              'dan' : [5, None, 7, 7],
              'may' : [10, 8, 10, 10]}
    outfile = open('grades.dat', 'wb')
    pickle.dump(grades, outfile)

def get_pickled_data():
    infile = open('grades.dat', 'rb')
    grades = pickle.load(infile)
    return grades
```

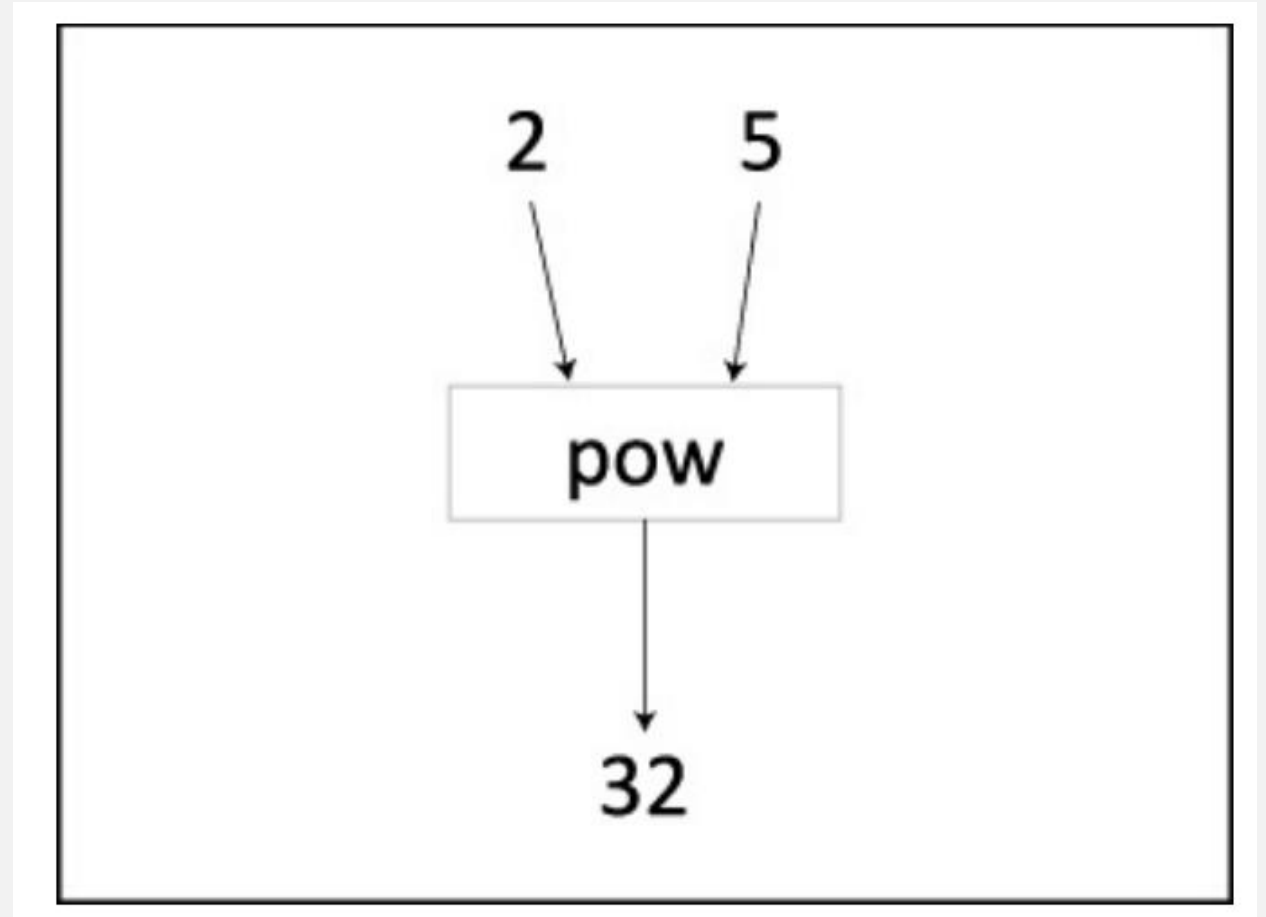
Calling Methods

- We've been calling methods quite a bit so far, so let's take a moment to look a little more carefully at a method call.
- Consider the built-in method `pow(x, y)`, which calculates $x ** y$ —that is to say, x raised to the power y :

```
>>> pow(2, 5)  
32
```

Calling Methods

- Figure gives a high-level overview of a method call



Calling Methods

- When a method takes no input (that is to say, it has zero arguments), you must still include the round brackets () after the method name:

```
>>> dir()  
['__builtins__', '__doc__', '__name__', '__package__']
```

Calling Methods

- The () tells Python to execute the method.
- If you leave off the (), then you get this:

```
>>> dir
```

```
<built-in method dir>
```

Methods that don't return a value

- Some methods, such as `print`, are not meant to return values. Consider:

```
>>> print('hello')
```

```
hello
```

```
>>> x = print('hello')
```

```
hello
```

```
>>> x
```

```
>>> print(x)
```

```
None
```


Reassigning method names

```
>>> dir = 3
```

```
>>> dir
```

```
3
```

```
>>> dir()
```

Traceback (most recent call last):

File "<pyshell#28>", line 1, in <module>

dir()

TypeError: 'int' object is not callable

Defining Methods

```
# area.py
import math
def area(radius):

    """ Returns the area of a circle
    with the given radius.
    For example:
    >>> area(5.5)
    95.033177771091246
    """

    return math.pi * radius ** 2
```

Defining Methods

- If everything is typed correctly, a prompt should appear and nothing else; a method is not executed until you call it.
- To call it, just type the name of the method, with the radius in brackets:

```
>>> area(1)
3.1415926535897931
>>> area(5.5)
95.033177771091246
>>> 2 * (area(3) + area(4))
157.07963267948966
```

Parts of a method

Let's look at each part of the area method.

- The first line, the one that begins with `def`, is called the method header; all the code indented beneath the header is called the method body.
- Method headers always begin with the keyword `def` (short for definition), followed by a space, and then the name of the method (in this case, `area`).
- Method names follow essentially the same rules as names for variables.

Parts of a method

- In the case of the area method, the return statement is the last line of the method, and it simply returns the value of the area of a circle using the standard formula.
- Note that it uses the radius parameter in its calculation; the value for radius is set when the area method is called.

Parts of a method

- A method is not required to have an explicit return statement.
- For example:

```
# hello.py
def say_hello_to(name):
    """ Prints a hello message.
    """
    cap_name = name.capitalize()
    print('Hello ' + cap_name + ', how are you?')
```

Parts of a method

- If you don't put a return anywhere in a method, Python treats the method as if it ended with this line:

`return None`

Variable Scope

- An important detail that methods bring up is the issue of scope.
- The scope of a variable (or method) is where in a program it is accessible, or visible.
- Consider these two methods:

```
# local.py
import math
def dist(x, y, a, b):
    s = (x - a) ** 2 + (y - b) ** 2
    return math.sqrt(s)
def rect_area(x, y, a, b):
    width = abs(x - a)
    height = abs(y - b)
    return width * height
```


Global variables

- Variables declared outside of any method are called global variables, and they are readable anywhere by any method or code within the program

Global variables

- Consider the following:

```
# global_error.py
name = 'Jack'
def say_hello():
    print('Hello ' + name + '!')
def change_name(new_name):
    name = new_name
```

Global variables

- The `say_hello()` method reads the value of `name` and prints it to the screen as you would expect:

```
>>> say_hello()  
Hello Jack!
```

Global variables

- However, things don't work as expected when you call `change_name`:

```
>>> change_name('Piper')
```

```
>>> say_hello()
```

```
Hello Jack!
```

Global variables

- To access the global variable, you must use the global statement:

```
# global_correct.py
name = 'Jack'
def say_hello():
    print('Hello ' + name + '!')
def change_name(new_name):
    global name
    name = new_name
```

Global variables

- Makes all the difference. Both methods now work as expected:

```
>>> say_hello()
```

```
Hello Jack!
```

```
>>> change_name('Piper')
```

```
>>> say_hello()Hello Piper!
```

Using a Main Method

- It is usually a good idea to use at least one method in any Python program you write: `main()`.
- A `main()` method is, by convention, assumed to be the starting point of your program
- Important: The `main()` method is NOT required in Python but it is a good practice.

Using a Main Method

```
# password2.py
def main():
    pwd = input('What is the password? ')
    if pwd == 'apple':
        print('Logging on ...')
    else:
        print('Incorrect password.')
    print('All done!')
```


Method Parameters

Pass by reference

- Python passes parameters to a method using a technique known as pass by reference.
- This means that when you pass parameters, the method refers to the original passed values
- using new names.
- For example, consider this simple program:

```
# reference.py  
def add(a, b):  
    return a + b
```

Method Parameters

- Run the interactive command line and type this:

```
>>> x, y = 3, 4
```

```
>>> add(x, y)
```

```
7
```

Method Parameters

x \longrightarrow **3**

y \longrightarrow **4**

Method Parameters

x → **3** ← **a**

y → **4** ← **b**

An important example

- Passing by reference is simple and efficient, but there are some things it cannot do.
- For example, consider this plausibly named method:

```
# reference.pydef set1(x):  
x = 1
```

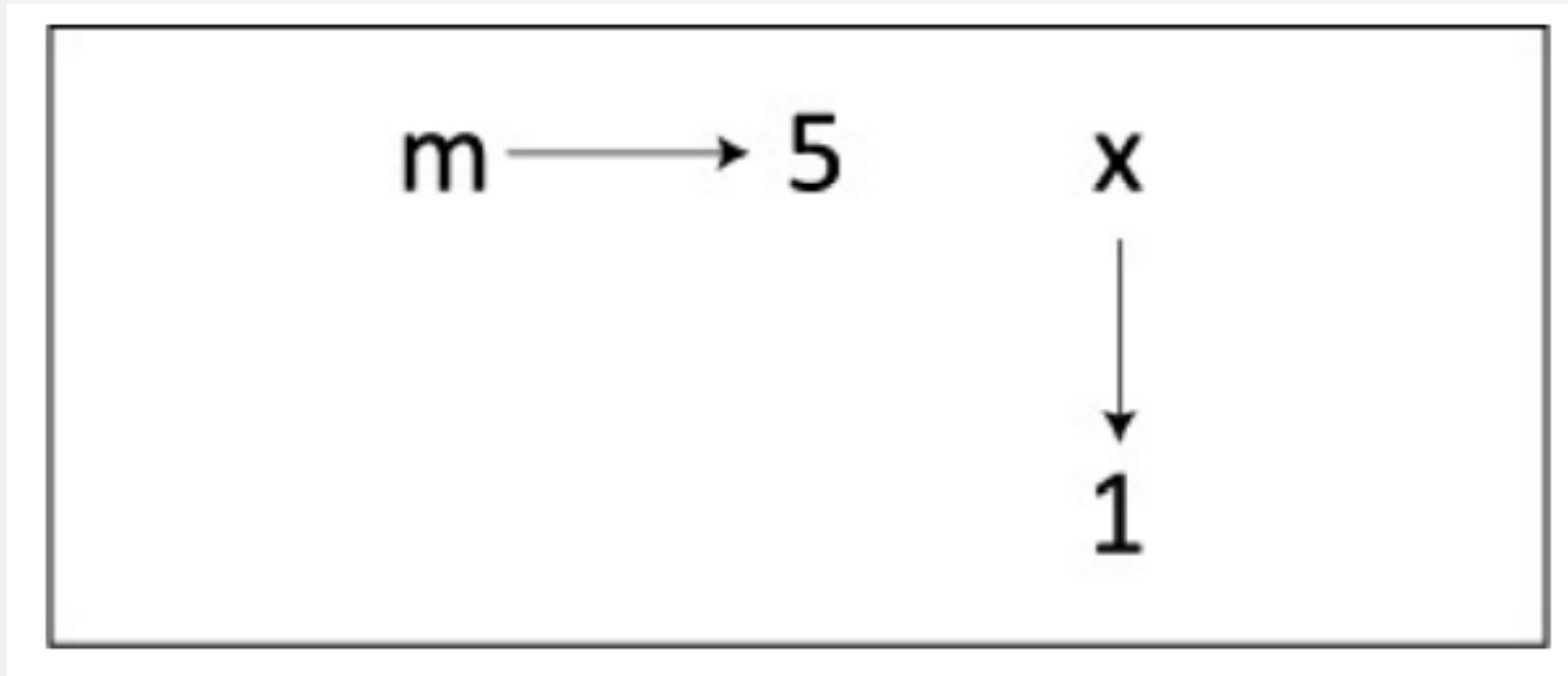
An important example

- The purpose of `set1` is to set the value of the passed-in variable to 1.
- But when you try it, it does not work as expected:

```
>>> m = 5  
>>> set1(m)  
>>> m  
5
```

An important example

- Assign 1 to m. Now the situation is as shown.



Default values

- It's often useful to include a default value with a parameter.
- For example, here we have given the greeting parameter a default value of 'Hello':

```
# greetings.py  
def greet(name, greeting = 'Hello'):  
    print(greeting, name + '!')
```


Default values

- You can now call greet in two distinct ways:

```
>>> greet('Bob')
```

```
Hello Bob!
```

```
>>> greet('Bob', 'Good morning')
```

```
Good morning Bob!
```

Keyword parameters

- Another useful way to specify parameters in Python is by using keywords. For example:

```
# shopping.py
def shop(where = 'store',
         what = 'pasta',
         howmuch = '10 pounds'):
    print('I want you to go to the', where)
    print('and buy', howmuch, 'of', what + '.')
```

Keyword parameters

- To call a method that uses keyword parameters, pass data in the form `param = value`. For example:

```
>>> shop()
I want you to go to the store
and buy 10 pounds of pasta.
>>> shop(what = 'towels')
I want you to go to the store
and buy 10 pounds of towels.
>>> shop(howmuch = 'a ton', what = 'towels')
I want you to go to the store
and buy a ton of towels.
>>> shop(howmuch = 'a ton', what = 'towels', where = 'bakery')
I want you to go to the bakery
and buy a ton of towels.
```

Modules

- A module is collection of related methods and variables.

To create a Python module

- Simple module for printing shapes to the screen:

```
# shapes.py
"""A collection of functions
for printing basic shapes.
"""

CHAR = '*'

def rectangle(height, width):
    """ Prints a rectangle. """
    for row in range(height):
        for col in range(width):
            print(CHAR, end = "")
        print()

def square(side):
    """ Prints a square. """
    rectangle(side, side)

def triangle(height):
    """ Prints a right triangle. """
    for row in range(height):
        for col in range(1, row + 2):
            print(CHAR, end = "")
        print()
```

- To use a module, you simply import it.
- For example:

```
>>> import shapes
>>> dir(shapes)
['CHAR', '__builtins__', '__doc__', '__file__', '__name__', '__package__', 'rectangle']
>>> print(shapes.__doc__)
A collection of functions
for printing basic shapes.
>>> shapes.CHAR
'*'
>>> shapes.square(5)
*****
*****
*****
*****
*****
*****
>>> shapes.triangle(3)
*
**
***
```

To create a Python module

- You can also import everything at once:

```
>>> from shapes import *
```

```
>>> rectangle(3, 8)
```

```
*****
```

```
*****
```

```
*****
```

Namespaces

- A very useful fact about modules is that they form namespaces.
- A namespace is essentially a set of unique variable and method names.
- The names within a module are visible outside the module only when you use an import statement.
- To see why this is important, suppose Jack and Sophie are working together on a large programming project.

Namespaces

- You can still run into name clashes as follows:

```
>>> from jack import *  
>>> from sophie import *
```



The Series Object

This session covers:

- Instantiating Series objects from lists, dictionaries, and more
- Creating a custom index for a Series
- Accessing attributes and invoking methods on a Series
- Performing mathematical operations on a Series
- Passing the Series to Python's built-in functions

Populating the Series with Values

```
In [3] ice_cream_flavors = ["Chocolate", "Vanilla", "Strawberry",  
                             "Rum Raisin"]  
  
      pd.Series(ice_cream_flavors)  
  
Out [3] 0      Chocolate  
        1      Vanilla  
        2    Strawberry  
        3    Rum Raisin  
        dtype: object
```

Customizing the Index

```
In [6] ice_cream_flavors = ["Chocolate", "Vanilla", "Strawberry",  
                             "Rum Raisin"]  
       days_of_week = ("Monday", "Wednesday", "Friday", "Wednesday")  
  
       # The two lines below are equivalent  
       pd.Series(ice_cream_flavors, days_of_week)  
       pd.Series(data = ice_cream_flavors, index = days_of_week)
```

Out [6]

Monday	Chocolate
Wednesday	Vanilla
Friday	Strawberry
Wednesday	Rum Raisin

dtype: object



Series Methods

This lesson covers:


- Importing CSV datasets
- Sorting Series values in ascending and descending order
- Retrieving the largest and smallest values in a Series
- Mutating a Series inplace
- Counting occurrences of unique values in a Series
- Applying an operation to every value in a Series



Importing a Dataset with the read_csv Method

- As always, our first step is to launch a fresh Jupyter Notebook and import pandas.
- Make sure to create the Notebook in the same directory as the CSV files you downloaded for this course.

In [1] import pandas as pd



```
In [2] pd.read_csv(filepath_or_buffer = "pokemon.csv")
pd.read_csv("pokemon.csv")
```

```
Out [2]
```

	Pokemon	Type
0	Bulbasaur	Grass / Poison
1	Ivysaur	Grass / Poison
2	Venusaur	Grass / Poison
3	Charmander	Fire
4	Charmeleon	Fire
...
804	Stakataka	Rock / Steel
805	Blacephalon	Fire / Ghost
806	Zeraora	Electric
807	Meltan	Steel
808	Melmetal	Steel
809	rows x 2 columns	

Importing a Dataset with the read_csv Method

```
In [3] pd.read_csv("pokemon.csv", index_col = "Pokemon")
```

```
Out [3]
```

	Type
Pokemon	
Bulbasaur	Grass / Poison
Ivysaur	Grass / Poison
Venusaur	Grass / Poison
Charmander	Fire
Charmeleon	Fire

Importing a Dataset with the read_csv Method

```
In [4] pd.read_csv("pokemon.csv", index_col = "Pokemon", squeeze = True)
```

```
Out [4] Pokemon
Bulbasaur      Grass / Poison
Ivysaur        Grass / Poison
Venusaur       Grass / Poison
Charmander           Fire
Charmeleon         Fire
...
Stakataka      Rock / Steel
Blacephalon    Fire / Ghost
Zeraora        Electric
Meltan          Steel
Melmetal        Steel
Name: Type, Length: 809, dtype: object
```



Importing a Dataset with the read_csv Method

The last step is to assign the `Series` to a variable so it can be reused throughout the Notebook.

```
In [5] pokemon = pd.read_csv("pokemon.csv", index_col = "Pokemon", squeeze  
    = True)
```

Curious if there are any `NaN` values in the `Series`? The `hasnans` attribute informs us there are no missing values in either the values or the index.

```
In [6] pokemon.hasnans
```

```
Out [6] False
```

```
In [7] pokemon.index.hasnans
```

```
Out [7] False
```

Importing a Dataset with the read_csv Method

```
In [8] pd.read_csv("google_stocks.csv").head()
```

```
Out [8]
```

	Date	Close
0	2004-08-19	49.98
1	2004-08-20	53.95
2	2004-08-23	54.50
3	2004-08-24	52.24
4	2004-08-25	52.80



Importing a Dataset with the read_csv Method

```
In [9] google = pd.read_csv("google_stocks.csv", index_col = "Date",  
    parse_dates = ["Date"], squeeze = True)  
  
    google.head()
```

```
Out [9] Date  
    2004-08-19    49.98  
    2004-08-20    53.95  
    2004-08-23    54.50  
    2004-08-24    52.24  
    2004-08-25    52.80  
    Name: Close, dtype: float64
```

Importing a Dataset with the read_csv Method

```
In [10] pd.read_csv("revolutionary_war.csv").tail()
```

```
Out [10]
```

	Battle	Start Date	State
227	Siege of Fort Henry	9/11/1782	Virginia
228	Grand Assault on Gibraltar	9/13/1782	NaN
229	Action of 18 October 1782	10/18/1782	NaN
230	Action of 6 December 1782	12/6/1782	NaN
231	Action of 22 January 1783	1/22/1783	Virginia

Importing a Dataset with the read_csv Method

```
In [11] pd.read_csv("revolutionary_war.csv",  
                    index_col = "Start Date",  
                    parse_dates = ["Start Date"]).tail()
```

Out [11]

Start Date	Battle	State
1782-09-11	Siege of Fort Henry	Virginia
1782-09-13	Grand Assault on Gibraltar	NaN
1782-10-18	Action of 18 October 1782	NaN
1782-12-06	Action of 6 December 1782	NaN
1783-01-22	Action of 22 January 1783	Virginia

Importing a Dataset with the read_csv Method

```
In [12] pd.read_csv("revolutionary_war.csv",  
                    index_col = "Start Date",  
                    parse_dates = ["Start Date"],  
                    usecols = ["State", "Start Date"],  
                    squeeze = True).tail()
```

```
Out [12] Start Date  
1782-09-11    Virginia  
1782-09-13         NaN  
1782-10-18         NaN  
1782-12-06         NaN  
1783-01-22    Virginia  
Name: State, dtype: object
```

Importing a Dataset with the read_csv Method

```
In [13] pokemon = pd.read_csv("pokemon.csv",
                                index_col = "Pokemon",
                                squeeze = True)

google = pd.read_csv("google_stocks.csv",
                      index_col = "Date",
                      parse_dates = ["Date"],
                      squeeze = True)

battles = pd.read_csv("revolutionary_war.csv",
                      index_col = "Start Date",
                      parse_dates = ["Start Date"],
                      usecols = ["State", "Start Date"],
                      squeeze = True)
```


Sorting a Series

Sorting by Values with the `sort_values` Method

```
In [14] google.sort_values()
```

```
Out [14] Date
```

2004-09-03	49.82
2004-09-01	49.94
2004-08-19	49.98
2004-09-02	50.57
2004-09-07	50.60

...

2019-04-23	1264.55
2019-10-25	1265.13
2018-07-26	1268.33
2019-04-26	1272.18
2019-04-29	1287.58

```
Name: Close, Length: 3824, dtype: float64
```

Sorting a Series

```
In [15] pokemon.sort_values()
```

```
Out [15] Pokemon
          Illumise      Bug
          Silcoon      Bug
          Pinsir      Bug
          Burmy      Bug
          Wurmple      Bug
          ...
          Tirtouga    Water / Rock
          Relicanth    Water / Rock
          Corsola      Water / Rock
          Carracosta    Water / Rock
          Empoleon      Water / Steel
          Name: Type, Length: 809, dtype: object
```

Sorting a Series

- In pandas, as in Python, lowercase characters are sorted after uppercase characters.
- In the example below, the string "adam" is placed after the string "Ben".

```
In [16] pd.Series(data = ["Adam", "adam", "Ben"]).sort_values()

Out [16] 0    Adam
         2    Ben
         1    adam
         dtype: object
```

```
In [17] google.sort_values(ascending = False).head()
```

```
Out [17] Date
2019-04-29      1287.58
2019-04-26      1272.18
2018-07-26      1268.33
2019-10-25      1265.13
2019-04-23      1264.55
Name: Close, dtype: float64
```

```
In [18] pokemon.sort_values(ascending = False).head()
```

```
Out [18] Pokemon
Empoleon      Water / Steel
Carracosta    Water / Rock
Corsola       Water / Rock
Relicanth     Water / Rock
Tirtouga      Water / Rock
Name: Type, dtype: object
```

Sorting a Series

```
In [19] battles.sort_values()
```

```
Out [19] Start Date
          1781-09-06      Connecticut
          1779-07-05      Connecticut
          1777-04-27      Connecticut
          1777-09-03      Delaware
          1777-05-17      Florida
          ...
          1782-08-08      NaN
          1782-08-25      NaN
          1782-09-13      NaN
          1782-10-18      NaN
          1782-12-06      NaN
          Name: State, Length: 232, dtype: object
```

Sorting a Series

```
In [20] battles.sort_values(na_position = "first")
```

```
Out [20] Start Date
```

1775-09-17	NaN
1775-12-31	NaN
1776-03-03	NaN
1776-03-25	NaN
1776-05-18	NaN

...

1781-07-06	Virginia
1781-07-01	Virginia
1781-06-26	Virginia
1781-04-25	Virginia
1783-01-22	Virginia

```
Name: State, Length: 232, dtype: object
```

Sorting a Series

```
In [21] battles.dropna().sort values()
```

```
Out [21] Start Date
```

```
1781-09-06      Connecticut
```

```
1779-07-05      Connecticut
```

```
1777-04-27      Connecticut
```

```
1777-09-03      Delaware
```

```
1777-05-17      Florida
```

```
...
```

```
1782-08-19      Virginia
```

```
1781-03-16      Virginia
```

```
1781-04-25      Virginia
```

```
1778-09-07      Virginia
```

```
1783-01-22      Virginia
```

```
Name: State, Length: 162, dtype: object
```

Sorting by Index with the sort_index Method

```
In [22] pokemon.sort_index()
```

```
Out [22] Pokemon
        Abomasnow      Grass / Ice
        Abra           Psychic
        Absol          Dark
        Accelgor        Bug
        Aegislash       Steel / Ghost
        ...
        Zoroark         Dark
        Zorua           Dark
        Zubat           Poison / Flying
        Zweilous        Dark / Dragon
        Zygarde         Dragon / Ground
        Name: Type, Length: 809, dtype: object
```


Sorting by Index with the sort_index Method

```
In [23] battles.sort_index(ascending = False, na_position = "first")
```

```
Out [23] Start Date
        NaT          New Jersey
        NaT          Virginia
        NaT          NaN
        NaT          NaN
        1783-01-22      Virginia
        ...
        1775-04-20      Virginia
        1775-04-19      Massachusetts
        1775-04-19      Massachusetts
        1774-12-14      New Hampshire
        1774-09-01      Massachusetts
        Name: State, Length: 232, dtype: object
```

Retrieving the Smallest and Largest Values with the nsmallest and nlargest Methods

```
In [24] google.nlargest(n = 5) # is the same as  
        google.nlargest()
```

```
Out [24] Date  
         2019-04-29      1287.58  
         2019-04-26      1272.18  
         2018-07-26      1268.33  
         2019-10-25      1265.13  
         2019-04-23      1264.55  
         Name: Close, dtype: float64
```

Retrieving the Smallest and Largest Values with the nsmallest and nlargest Methods

```
In [25] google.nsmallest(n = 6) # is the same as  
        google.nsmallest(6)
```

```
Out [25] Date  
        2004-09-03      49.82  
        2004-09-01      49.94  
        2004-08-19      49.98  
        2004-09-02      50.57  
        2004-09-07      50.60  
        2004-08-30      50.81  
        Name: Close, dtype: float64
```

Overwriting a Series with the inplace Parameter

```
In [26] battles.head(3)

Out [26] Start Date
         1774-09-01    Massachusetts
         1774-12-14    New Hampshire
         1775-04-19    Massachusetts
         Name: State, dtype: object

In [27] battles.sort_values().head(3)

Out [27] Start Date
         1781-09-06    Connecticut
         1779-07-05    Connecticut
         1777-04-27    Connecticut
         Name: State, dtype: object

In [28] battles.head(3)

Out [28] Start Date
         1774-09-01    Massachusetts
         1774-12-14    New Hampshire
         1775-04-19    Massachusetts
         Name: State, dtype: object
```

Overwriting a Series with the inplace Parameter

```
In [29] battles.head(3)
```

```
Out [29] Start Date  
         1774-09-01      Massachusetts  
         1774-12-14      New Hampshire  
         1775-04-19      Massachusetts  
         Name: State, dtype: object
```

```
In [30] battles.sort_values(inplace = True)
```

```
In [31] battles.head(3)
```

```
Out [31] Start Date  
         1781-09-06      Connecticut  
         1779-07-05      Connecticut  
         1777-04-27      Connecticut  
         Name: State, dtype: object
```

Counting Values with the value_counts Method

```
In [32] pokemon.value_counts()
```

```
Out [32] Normal          65  
         Water          61  
         Grass          38  
         Psychic        35  
         Fire           30  
         ..  
         Fire / Dragon    1  
         Dark / Ghost     1  
         Steel / Ground   1  
         Fire / Psychic   1  
         Dragon / Ice     1  
         Name: Type, Length: 159, dtype: int64
```



Counting Values with the value_counts Method

- The length of the value_counts Series will be equal to the number of unique values from the pokemon Series.
- As a reminder, the nunique method returns this piece of information.

In [33] len(pokemon.value_counts())

Out [33] 159

In [34] pokemon.nunique()

Out [34] 159

Counting Values with the value_counts Method

```
In [35] pokemon.value_counts(ascending = True)
```

```
Out [35] Rock / Poison      1
         Ghost / Dark       1
         Ghost / Dragon     1
         Fighting / Steel   1
         Rock / Fighting    1
         ..
         Fire               30
         Psychic            35
         Grass              38
         Water              61
         Normal             65
```


Counting Values with the value_counts Method

- The normalize parameter can be set to True to return the frequencies of each unique value.
- The frequency represents what portion of the dataset a given value makes up.

```
In [36] pokemon.value_counts(normalize = True)
```

```
Out [36] Normal          0.080346  
         Water          0.075402  
         Grass          0.046972  
         Psychic        0.043263  
         Fire           0.037083
```

Counting Values with the value_counts Method

```
In [37] pokemon.value_counts(normalize = True) * 100
```



```
Out [37]
```

Normal	8.034611
Water	7.540173
Grass	4.697157
Psychic	4.326329
Fire	3.708282

- Normal Pokémon make up 8.03% of the dataset, Water make up 7.54%, and so on.

Counting Values with the value_counts Method

```
In [38] (pokemon.value_counts(normalize = True) * 100).round(2)
```



```
Out [38]
```

Normal	8.03
Water	7.54
Grass	4.70
Psychic	4.33
Fire	3.71
...	
Rock / Fighting	0.12
Fighting / Steel	0.12
Ghost / Dragon	0.12
Ghost / Dark	0.12
Rock / Poison	0.12

Name: Type, Length: 159, dtype: float64

Counting Values with the value_counts Method

- A Series with numeric values will work similarly.
- In the example below, we can see that no stock price appears more than three times in our google dataset.

```
In [39] google.value_counts().head()
```

```
Out [39] 237.04      3  
         288.92      3  
         287.68      3  
         290.41      3  
         194.27      3
```



Counting Values with the value_counts Method

- Let's begin by determining the smallest and largest values within the Series with the max and min methods.
- An alternative solution is to pass the Series into Python's built-in max and min functions.

In [40] google.max()

Out [40] 1287.58

In [41] google.min()

Out [41] 49.82

Counting Values with the value_counts Method

```
In [42] bins = [0, 200, 400, 600, 800, 1000, 1200, 1400]
        google.value_counts(bins = bins)
```

```
Out [42] (200.0, 400.0]      1568
        (-0.001, 200.0]     595
        (400.0, 600.0]      575
        (1000.0, 1200.0]    406
        (600.0, 800.0]      380
        (800.0, 1000.0]     207
        (1200.0, 1400.0]     93
        Name: Close, dtype: int64
```

Counting Values with the value_counts Method

```
In [43] google.value_counts(bins = bins).sort_index()
```



```
Out [43] (-0.001, 200.0]      595  
         (200.0, 400.0]     1568  
         (400.0, 600.0]     575  
         (600.0, 800.0]     380  
         (800.0, 1000.0]    207  
         (1000.0, 1200.0]   406  
         (1200.0, 1400.0]    93  
         Name: Close, dtype: int64
```

Counting Values with the value_counts Method

- An alternative solution is to pass a value of False to the sort parameter of the value_counts method.
- This will yield the same result.

```
In [44] google.value_counts(bins = bins, sort = False)
```

```
Out [44] (-0.001, 200.0]      595  
         (200.0, 400.0]      1568  
         (400.0, 600.0]      575  
         (600.0, 800.0]      380  
         (800.0, 1000.0]     207  
         (1000.0, 1200.0]    406  
         (1200.0, 1400.0]     93  
         Name: Close, dtype: int64
```


Counting Values with the value_counts Method

```
In [45] google.value_counts(bins = 6, sort = False)
```

```
Out [45] (48.581, 256.113]      1204  
         (256.113, 462.407]     1104  
         (462.407, 668.7]       507  
         (668.7, 874.993]       380  
         (874.993, 1081.287]    292  
         (1081.287, 1287.58]    337  
         Name: Close, dtype: int64
```

Counting Values with the value_counts Method

- What about our battles dataset? We can use the value_counts method to see which states had the most battles in the Revolutionary War.

```
In [46] battles.value_counts().head()
```


```
Out [46] South Carolina      31  
         New York           28  
         New Jersey         24  
         Virginia           21  
         Massachusetts      11  
         Name: State, dtype: int64
```

Counting Values with the value_counts Method

- NaN values will be missing from the list by default.
- Pass the dropna parameter an argument of False to count null values as a distinct category.

```
In [47] battles.value_counts(dropna = False).head()
```

```
Out [47] NaN          70  
         South Carolina  31  
         New York       28  
         New Jersey     24  
         Virginia       21  
         Name: State, dtype: int64
```



```
In [48] battles.index
```

```
Out [48]
```

```
DatetimeIndex(['1774-09-01', '1774-12-14', '1775-04-19', '1775-04-19',  
              '1775-04-20', '1775-05-10', '1775-05-27', '1775-06-11',  
              '1775-06-17', '1775-08-08',  
              ...  
              '1782-08-08', '1782-08-15', '1782-08-19', '1782-08-26',  
              '1782-08-25', '1782-09-11', '1782-09-13', '1782-10-18',  
              '1782-12-06', '1783-01-22'],  
              dtype='datetime64[ns]', name='Start Date', length=232,  
              freq=None)
```

```
In [49] battles.index.value_counts()
```

```
Out [49] 1775-04-19      2  
         1781-05-22      2  
         1781-04-15      2  
         1782-01-11      2  
         1780-05-25      2  
         ..  
         1778-05-20      1  
         1776-06-28      1  
         1777-09-19      1  
         1778-08-29      1  
         1777-05-17      1  
         Name: Start Date, Length: 217, dtype: int64
```



Invoking a Function on Every Series Value with the apply Method

- A function can be treated like any other object in Python.
- This is a tricky concept for some because a function is a more abstract data type than a concrete value like an integer.
- But anything that you can do with an object like a number, you can also do with a function.



Invoking a Function on Every Series Value with the apply Method

- The example below declares a funcs list that stores 3 of Python's built-in functions.
- Notice that the functions are not invoked within the list.
- These are references to the functions themselves.
- In analogous terms, we have a cookbook of 3 recipes here, but we haven't started baking anything yet.

In [50] funcs = [len, max, min]

Invoking a Function on Every Series Value with the apply Method

- For each iteration, we invoke the current function being referenced by func and pass in the google Series.
- The output thus includes the length of the Series followed by its maximum and minimum values.

```
In [51] funcs = [len, max, min]

        for func in funcs:
            print(func(google))

Out [51] 3824
         1287.58
         49.82
```

Invoking a Function on Every Series Value with the apply Method

```
In [52] round(99.2)
```

```
Out [52] 99
```

```
In [53] round(99.49)
```

```
Out [53] 99
```

```
In [54] round(99.5)
```

```
Out [54] 100
```


Invoking a Function on Every Series Value with the apply Method

```
In [55] google.apply(func = round)
        google.apply(round)
```

```
Out [55] Date
2004-08-19      50
2004-08-20      54
2004-08-23      54
2004-08-24      52
2004-08-25      53
...
2019-10-21    1246
2019-10-22    1243
2019-10-23    1259
2019-10-24    1261
2019-10-25    1265
Name: Close, Length: 3824, dtype: int64
```

Invoking a Function on Every Series Value with the apply Method

- We can use the in operator to check for the presence of the substring "/" in the input string. If it is found, we'll return the string "Multi". Otherwise, we'll return the string "Single".

```
In [56] def single_or_multi(types):  
        if "/" in types:  
            return "Multi"  
  
        return "Single"
```

Invoking a Function on Every Series Value with the apply Method

- Let's get ready to pass the `single_or_multi` function to the `apply` method. Here's a quick refresher of our pokemon dataset.

```
In [57] pokemon.head(4)
```



```
Out [57] Pokemon
```

Bulbasaur	Grass / Poison
Ivysaur	Grass / Poison
Venusaur	Grass / Poison
Charmander	Fire

```
Name: Type, dtype: object
```

Invoking a Function on Every Series Value with the apply Method

```
In [58] pokemon.apply(single_or_multi)
```


```
Out [58] Pokemon
          Bulbasaur      Multi
          Ivysaur       Multi
          Venusaur      Multi
          Charmander    Single
          Charmeleon    Single
          ...
          Stakataka     Multi
          Blacephalon    Multi
          Zeraora       Single
          Meltan        Single
          Melmetal      Single
          Name: Type, Length: 809, dtype: object
```

Invoking a Function on Every Series Value with the apply Method

- We have a new Series object now! Let's find out how many Pokémon fall into each classification by using `value_counts`.

```
In [59] pokemon.apply(single_or_multi).value_counts()

Out [59] Multi      405
         Single     404
         Name: Type, dtype: int64
```



Demo / Use Case Python with Data Analytics