Python: Object-Oriented Programming & General Tips

Klein
carlj.klein@gmail.com

1 Acknowledgments

The majority of these notes follow closely with the Data Science Nanodegree created by Udacity.

2 Learning Objectives

Preferably starting with some background in Python coding, the idea is to examine professional coding practices such as writing clean code and testing code, and then move into some object-oriented programming and creating web applications and dashboards.

1. Software Engineering Practices Pt. 1
2. Software Engineering Practices Pt. 2
3. Introduction to Object-Oriented Programming
4. Introduction to Web Development

3 Software Engineering Practices Pt. 1

3.1 Learning Targets

- Write clean and modular mode
- Improve code efficiency
- Add effective documentation
- Use version control

3.2 Clean and Modular Code

- Production code: software being used by live users
- Clean code: code that is readable, simple, and precise. Crucial for collaboration
- Modular code: code that is logically broken up into functions and modules
- Module: a file that allows code to be reused

3.3 Refactoring Code

- Refactoring: restructuring the code to improve its internal structure without changing its external functionality
- Usually done after the code runs properly
- Easier to maintain, collaborate and reuse

3.4 Tips on Writing Clean Code

- Use meaningful names
- Be descriptive and imply type
  - Verbs for functions
  - Nouns for variables
- Use arbitrary naming conventions for variables in functions (i.e. a column of temperature data could be "temperature_array")
- Use white space properly

3.5 Writing Modular Code

- DRY: Don't Repeat Yourself
- Abstract out logic to improve readability
- Aim to minimize number of entities (functions, classes, modules, etc.), but also keep balance in mind
- Functions should do ONE thing
- Use arbitrary variable names
- Try for fewer (at most about 3) arguments per functions

3.6 Efficient Code

- Reducing run time
- Reducing space in memory
- Tip: Use vector operations (vectorization) over loops hwne possible (think NumPy and Pandas functions)

3.7 Documentation

- Clarify complex parts of code
- Navigate code easily
- Convey how and why code works
- Line Level: inline comments to clarify
- Function or Module Level: Docstrings to clarify
- Project Level: README to clarify

3.8 Version Control (git)

- Scenario 1: Switching between module work, merging, and returning
- Scenario 2: Using detailed git commit lines, return to the faster running code and see what changed
- Scenario 3: Another user has been working on a separate branch while you were working on your branches

4 Software Engineering Practices Pt. 2

4.1 Learning Targets

- Testing
- Logging
- Code Reviews

4.2 Testing

- Essential before deployment
- Many data scientist's code goes into production without testing, and can be an annoyance to the software developers

- Test Driven Development (TDD): development process in which tests are written prior to code, itself
- Unit Test: a test covering a unit of code, usually a single function

4.3 Unit Tests Basics

- Useful when testing hundreds of functions repeatedly while tracking the outcomes
- Reduces manual tests
- Repeatable and Automated!
- Isolated from the rest of program, thus no dependencies involved
- A renowned testing tool: pytest
- TDD: write unit tests as programming for errors that are likely occur

4.4 Logging

- Logging: Recording of Errors / Error Messages
- Be professional and clear
- Be concise and use normal capitalization
- Choose appropriate level for logging:
  - Debug: use this "level" for anything that happens in the program
  - Error: use for any errors that occur
  - Info: records all actions that are user-driven or system-specific (regularly scheduled operations)
  - Provide any useful information

4.5 Code Reviews

- Catch errors
- Ensure readability
- Check standards are met
- Share knowledge (personal, professional, team, etc.)
- Questions to ask when conducting a code review:
  - Can I understand the code easily?
  - Does it use meaningful names and whitespace?

  - Is there duplicated code?
  - Can I provide another layer of abstraction?
  - Is each function and module necessary?
  - Is each function or module the right length?
- Is the code efficient?
  - Are there loops or other steps I can vectorize?
  - Can I use better data structures to optimize any steps?
  - Can I use generators or multiprocessing to optimize any steps?
- Is the documentation effective?
  - Are inline comments concise and meaningful?
  - Is there complex code that missing documentation?
  - Do functions use effective docstrings?
  - Is the necessary project documentation provided?
- Is the code well tested?
  - Does the code have test coverage?
  - Do tests check for interesting cases?
  - Are the tests readable?
  - Can the tests be made more efficient?
- Is the logging effective?
  - Are the log messages clear, concise, and professional?
  - Do they include all relevant and useful information?
  - Do they use the appropriate logging level?
- Tips for code review:
  - Use a code linter: linters like pylint can check for coding standard and PEP8 guidelines
  - Agree on a style guide as a team
  - Explain issues and make suggestions
  - Keep comments objective (avoid "I" and "you")

5 Introduction to Object-Oriented Programming

5.1 Learning Targets

- Syntax of Object-Oriented Programming
  - Procedural vs. Object-Oriented Programming
  - Classes, Objects, Methods, and Attributes
  - Coding a Class
  - Magic Methods
  - Inheritance
- Build a Python Package that Analyzes Distributions

5.2 Procedural vs. OOP

- Object: Specific instance of a class in which attributes (and sometimes methods) change
  - Characteristics (attributes)
  - Actions (methods)
- Class: generic version of an object, almost a blueprint, which specifies attributes and methods
- Magic methods: overwrite default python behavior

5.3 Inheritance

- Imagine a Clothing parent class which provides a blueprint for any type of clothing
- The general class helps to create more modular code and follow DRY
- Inheritance helps organize code with a more general version of a class which translates to more specified children classes
- Inheritance can make OOP more efficient to write
- Updates to a parent class automatically trickle down to its children!

5.4 OOP Example: Clothing

We'll be walking through the process of using object-oriented programming, starting with basic objects and then working our way to a generic parent class. Example 1: Create an object named Shirt

- Attributes

  - color
  - size
  - style
  - price

- Methods
  - change_price: set the price a the shirt to a new value
  - discount: returns the price of a shirt with a discount applied

```
class Shirt:
  def __init__(self, shirt_color, shirt_size, shirt_style,
                                    shirt_price):
    self.color = shirt_color
    self.size = shirt_size
    self.style = shirt_style
    self.price = shirt_price

  def change_price(self, new_price):
    self.price = new_price

  def discount(self, discount):
    return self.price * (1 - discount)

"""
Now, create a Shirt type object and test out the attributes and
                                    methods.
"""
new_shirt = Shirt('red', 'S', 'short sleeve', 15)

print(new_shirt.color)
print(new_shirt.size)
print(new_shirt.style)
print(new_shirt.price)
"""
red
S
short sleeve
15
"""

new_shirt.change_price(10)
print(new_shirt.price)
# 10

print(new_shirt.discount(0.2))
# 8.0
```

Similar to the above, write a Pants class. This class will have waist size and length instead of shirt size and style, respectively.

```
class Pants:
  def __init__(self, color, waist_size, length, price):
    """Method for initializing a Pants object
```

```
    Args:
    color (str)
    waist_size (int)
    length (int)
    price (float)

    Attributes:
    color (str): color of a pants object
    waist_size (str): waist size of a pants object
    length (str): length of a pants object
    price (float): price of a pants object
    """

    self.color = color
    self.waist_size = waist_size
    self.length = length
    self.price = price

  def change_price(self, new_price):
    """The change_price method changes the price attribute of a
                                pants object

    Args:
    new_price (float): the new price of the pants object

    Returns: None

    """
    self.price = new_price

  def discount(self, discount):
    """The discount method outputs a discounted price of a pants
                                object

    Args:
    percentage (float): a decimal representing the amount to
                                discount

    Returns:
    float: the discounted price
    """
    return self.price * (1 - discount)
```

Now, let's see how we can combine and use the objects we create. Make a SalesPerson class with information and methods pertaining to selling pants.

```
class SalesPerson:
  def __init__(self, first_name, last_name, employee_id, salary):
    self.first_name = first_name
    self.last_name = last_name
    self.employee_id = employee_id
    self.salary = salary
    self.pants_sold = []
    self.total_sales = 0

  def sell_pants(self, pants_object):
    self.pants_sold.append(pants_object)
```

```
  def calculate_sales(self):
    self.total_sales = sum([pants.price for pants in self.
                            pants_sold])
    return self.total_sales

  def display_sales(self):
    for pants in self.pants_sold:
    print(f'color: {pants.color}, length: {pants.length}, price:
                            {pants.price}, waist size: {pants
                            .waist_size}')

  def calculate_commission(self, commission):
    sales_total = self.calculate_sales()
    return sales_total * commission
```

Something notable about SalesPerson class is the fact we set attributes to predefined values, such as the blank list and value of 0. Additionally, note that we were able to call Pants class attributes after they had been passed in.

Moving into the more general parent class we discussed while describing inheritance, let's create that general Clothing class.

```
class Clothing:
  def __init__(self, color, size, style, price):
    self.color = color
    self.size = size
    self.style = style
    self.price = price

  def change_price(self, price):
    self.price = price

  def calculate_discount(self, discount):
    return self.price * (1 - discount)

  def calculate_shipping(self, weight, rate):
    return weight * rate
"""
Children classes will inherit Clothing's attributes and methods,
but can also be added to and overwritten.
See Below for some examples:
"""

# Shirt
class Shirt(Clothing):
  def __init__(self, color, size, style, price, long_or_short):
    Clothing.__init__(self, color, size, style, price)
    self.long_or_short = long_or_short

  def double_price(self):
  self.price = 2 * self.price

# Pants
class Pants(Clothing):
  def __init__(self, color, size, style, price, waist):
```

```
    Clothing.__init__(self, color, size, style, price)
    self.waist = waist

  def calculate_discount(self, discount):
    return self.price * (discount / 2)

# Blouse
class Blouse(Clothing):
  def __init__(self, color, size, style, price, country_of_origin
                                ):
    Clothing.__init__(self, color, size, style, price)
    self.country_of_origin = country_of_origin

  def triple_price(self):
    return self.price * 3
```

5.5 OOP Example: Distributions

We'll sidestep the learning order and exploration phase somewhat in these notes. It wouldn't be unusual to build out a specific class for a distribution, and then given a more diverse need, build out a parent class for the distributions. However, we'll present a parent Distribution class, and then build a few child distributions through inheritance.

```
class Distribution:
  def __init__(self, mu=0, sigma=1):
  """ Generic distribution class for calculating and
  visualizing a probability distribution.

    Attributes:
    mean (float) representing the mean value of the distribution
    stdev (float) representing the standard deviation of the
                                distribution
    data_list (list of floats) a list of floats extracted from
                                the data file
    """
    self.mean = mu
    self.stdev = sigma
    self.data = []

  def read_data_file(self, file_name):

    """Function to read in data from a txt file. The txt file
                                should have
    one number (float) per line. The numbers are stored in the
                                data   attribute.

    Args:
    file_name (string): name of a file to read from

    Returns:
    None

    """
```

```
        with open(file_name) as file:
        data_list = []
        line = file.readline()
        while line:
        data_list.append(int(line))
        line = file.readline()
        file.close()

        self.data = data_list
```

Note a new method with this class is the use of reading in an outside file. It may be useful to consider files methods that will read in or output files of different types.

```
class Gaussian(Distribution):
  """ Gaussian distribution class for calculating and
  visualizing a Gaussian distribution.

  Attributes:
  mean (float) representing the mean value of the distribution
  stdev (float) representing the standard deviation of the
                                distribution
  data_list (list of floats) a list of floats extracted from the
                                data file

  """
  def __init__(self, mu=0, sigma=1):
    Distribution.__init__(self, mu, sigma)

  def calculate_mean(self):
    """Function to calculate the mean of the data set.

    Args:
    None

    Returns:
    float: mean of the data set

    """
    avg = 1.0 * sum(self.data) / len(self.data)

    self.mean = avg

    return self.mean

  def calculate_stdev(self, sample=True):
    """Function to calculate the standard deviation of the data
                                set.

    Args:
    sample (bool): whether the data represents a sample or
                                population

    Returns:
    float: standard deviation of the data set
```

```
        """

        if sample:
        n = len(self.data) - 1
        else:
        n = len(self.data)

        mean = self.calculate_mean()

        sigma = 0

        for d in self.data:
        sigma += (d - mean) ** 2

        sigma = math.sqrt(sigma / n)

        self.stdev = sigma

        return self.stdev

    def plot_histogram(self):
        """Function to output a histogram of the instance variable
                                    data using
        matplotlib pyplot library.

        Args:
        None

        Returns:
        None
        """
        plt.hist(self.data)
        plt.title('Histogram of Data')
        plt.xlabel('data')
        plt.ylabel('count')

    def pdf(self, x):
        """Probability density function calculator for the gaussian
                                    distribution.

        Args:
        x (float): point for calculating the probability density
                                    function


        Returns:
        float: probability density function output
        """

        return (1.0 / (self.stdev * math.sqrt(2*math.pi))) * math.exp
                                    (-0.5*((x - self.mean) / self.
                                    stdev) ** 2)

    def plot_histogram_pdf(self, n_spaces = 50):
        """Function to plot the normalized histogram of the data and
                                    a plot of  the
```

```
        probability density function along the same range

        Args:
        n_spaces (int): number of data points

        Returns:
        list: x values for the pdf plot
        list: y values for the pdf plot

        """

        mu = self.mean
        sigma = self.stdev

        min_range = min(self.data)
        max_range = max(self.data)

        # calculates the interval between x values
        interval = 1.0 * (max_range - min_range) / n_spaces

        x = []
        y = []

        # calculate the x values to visualize
        for i in range(n_spaces):
        tmp = min_range + interval*i
        x.append(tmp)
        y.append(self.pdf(tmp))

        # make the plots
        fig, axes = plt.subplots(2,sharex=True)
        fig.subplots_adjust(hspace=.5)
        axes[0].hist(self.data, density=True)
        axes[0].set_title('Normed Histogram of Data')
        axes[0].set_ylabel('Density')

        axes[1].plot(x, y)
        axes[1].set_title('Normal Distribution for \n Sample Mean and
                                    Sample Standard Deviation')
        axes[0].set_ylabel('Density')
        plt.show()

        return x, y

    def __add__(self, other):
        """Function to add together two Gaussian distributions

        Args:
        other (Gaussian): Gaussian instance

        Returns:
        Gaussian: Gaussian distribution

        """
        result = Gaussian()
        result.mean = self.mean + other.mean
        result.stdev = math.sqrt(self.stdev ** 2 + other.stdev ** 2)
```

```
        return result

    def __repr__(self):
        """Function to output the characteristics of the Gaussian
                                    instance

        Args:
        None

        Returns:
        string: characteristics of the Gaussian

        """

        return "mean {}, standard deviation {}".format(self.mean,
                                    self.stdev)
```

Note that in the build of child Gaussian class we feature a method to produce a visual, and we introduce our first magic methods.

Magic methods override the main functionality of core Python methods, and have the telltale sign of the double underscores on each side (dunders, if you will).

Next up, we add more variability with the Binomial distribution.

```
class Binomial(Distribution):
    """ Binomial distribution class for calculating and
    visualizing a Binomial distribution.

    Attributes:
    mean (float) representing the mean value of the distribution
    stdev (float) representing the standard deviation of the
                                    distribution
    data_list (list of floats) a list of floats to be extracted
                                    from the data   file
    p (float) representing the probability of an event occurring
    n (int) the total number of trials


    TODO: Fill out all TODOs in the functions below

    """

    #       A binomial distribution is defined by two variables:
    #           the probability of getting a positive outcome
    #           the number of trials

    #       If you know these two values, you can calculate the
                                    mean and the  standard deviation
    #
    #       For example, if you flip a fair coin 25 times, p = 0.5
                                    and n = 25
    #       You can then calculate the mean and standard deviation
                                    with the   following formula:
```

```
#           mean = p * n
#           standard deviation = sqrt(n * p * (1 - p))

#

def __init__(self, mu = 0, sigma = 1, prob = 0.5, size = 20):
  Distribution.__init__(self, mu, sigma)
  self.prob = prob
  self.size = size

# TODO: store the probability of the distribution in an
                              instance variable p
# TODO: store the size of the distribution in an instance
                              variable n

# TODO: Now that you know p and n, you can calculate the mean
                              and standard  deviation
#       Use the calculate_mean() and calculate_stdev() methods
                              to calculate   the
#       distribution mean and standard deviation
#
#       Then use the init function from the Distribution class
                              to   initialize the
#       mean and the standard deviation of the distribution
#
#       Hint: You need to define the calculate_mean() and
                              calculate_stdev()   methods
#               farther down in the code starting in line 55.
#               The init function can get access to these
                              methods via the   self
#               variable.

def calculate_mean(self):
  """Function to calculate the mean from p and n

  Args:
  None

  Returns:
  float: mean of the data set

  """

  # TODO: calculate the mean of the Binomial distribution.
                              Store the mean
  #       via the self variable and also return the new mean
                              value

  self.mu = self.prob * self.size
  return self.mu

def calculate_stdev(self):
  """Function to calculate the standard deviation from p and n.

  Args:
  None
```

```
        Returns:
        float: standard deviation of the data set

        """
        # TODO: calculate the standard deviation of the Binomial
                                    distribution.  Store
        #       the result in the self standard deviation attribute.
                                    Return the   value
        #       of the standard deviation.
        # sqrt(n * p * (1 - p))
        self.sigma = math.sqrt(self.size * self.prob * (1 - self.prob
                                    ))
        return self.sigma

    def replace_stats_with_data(self):
        """Function to calculate p and n from the data set

        Args:
        None

        Returns:
        float: the p value
        float: the n value

        """
        # TODO: The read_data_file() from the Generaldistribution
                                    class can   read in a data
        #       file. Because the Binomaildistribution class inherits
                                     from the  Generaldistribution
                                    class,
        #       you don't need to re-write this method. However,  the
                                     method
        #       doesn't update the mean or standard deviation of
        #       a distribution. Hence you are going to write a method
                                     that  calculates n, p, mean and
        #       standard deviation from a data set and then updates
                                    the n, p,   mean and stdev
                                    attributes.
        #       Assume that the data is a list of zeros and ones like
                                     [0 1 0 1  1 0 1].
        #
        #       Write code that:
        #           updates the n attribute of the binomial
                                    distribution
        #           updates the p value of the binomial distribution
                                    by   calculating the
        #               number of positive trials divided by the
                                    total trials
        #           updates the mean attribute
        #           updates the standard deviation attribute
        #
        #       Hint: You can use the calculate_mean() and
                                    calculate_stdev()  methods
        #           defined previously.
        self.size = len(self.data)
        self.calculate_mean()
        self.calculate_stdev()
```

```
def plot_bar(self):
    """Function to output a histogram of the instance variable
                                data using
    matplotlib pyplot library.

    Args:
    None

    Returns:
    None
    """

    # TODO: Use the matplotlib package to plot a bar chart of the
                                 data
    #       The x-axis should have the value zero or one
    #       The y-axis should have the count of results for each
                                case
    #
    #       For example, say you have a coin where heads = 1 and
                                tails = 0.
    #       If you flipped a coin 35 times, and the coin landed
                                on
    #       heads 20 times and tails 15 times, the bar chart
                                would have two   bars:
    #       0 on the x-axis and 15 on the y-axis
    #       1 on the x-axis and 20 on the y-axis
    #       Make sure to label the chart with a title, x-axis
                                label and   y-axis label
    dat = np.array(self.data)
    zero = len(dat[dat==0])
    one = len(dat[dat==1])
    plt.bar(plt.bar(['zero','one'],[zero, one]))
    plt.xlabel('Result')
    plt.ylabel('Count')
    plt.title('Counts of Binomial Data')

def pdf(self, k):
    """Probability density function calculator for the gaussian
                                 distribution.

    Args:
    k (float): point for calculating the probability density
                                function

    Returns:
    float: probability density function output
    """
    # TODO: Calculate the probability density function for a
                                binomial   distribution
    #  For a binomial distribution with n trials and probability
                                p,
    #  the probability density function calculates the likelihood
                                 of getting
    #   k positive outcomes.
    #
    #   For example, if you flip a coin n = 60 times, with p = .5
```

```
                                ,
    #   what's the likelihood that the coin lands on heads 40 out
                                 of 60   times?
    n_choose_k = math.factorial(self.size) / (math.factorial(k) *
                                   (math.factorial(self.size - k)
                                ))
    test_prob = n_choose_k * (self.prob ** k) * ((1 - self.prob)
                                **   (self.size - k))
    return test_prob

  def plot_bar_pdf(self):
    """Function to plot the pdf of the binomial distribution

    Args:
    None

    Returns:
    list: x values for the pdf plot
    list: y values for the pdf plot
    """
    # TODO: Use a bar chart to plot the probability density
                                function from
    # k = 0 to k = n

    #   Hint: You'll need to use the pdf() method defined above
                                to  calculate the
    #   density function for every value of k.

    #   Be sure to label the bar chart with a title, x label and
                                y label

    #   This method should also return the x and y values used to
                                 make the  chart
    #   The x and y values should be stored in separate lists
    k_values = np.arange(self.size + 1)
    k_probs = [self.pdf(k) for k in k_values]
    plt.bar(k_values, k_probs)
    plt.xlabel('Successful Outcomes')
    plt.ylabel('Probability')
    plt.title('Probability  of Successful Outcomes')

  def __add__(self, other):
    """Function to add together two Binomial distributions with
                                equal p

    Args:
    other (Binomial): Binomial instance

    Returns:
    Binomial: Binomial distribution

    """
    try:
    assert self.p == other.p, 'p values are not equal'
    except AssertionError as error:
    raise
```

```
        # TODO: Define addition for two binomial distributions.
                                    Assume that the
        # p values of the two distributions are the same. The formula
                                     for
        # summing two binomial distributions with different p values
                                    is more  complicated,
        # so you are only expected to implement the case for two
                                    distributions with equal p.

        # the try, except statement above will raise an exception if
                                    the p  values are not equal

        # Hint: You need to instantiate a new binomial object with
                                    the correct  n, p,
        #   mean and standard deviation values. The __add__ method
                                    should   return this
        #   new binomial object.

        #   When adding two binomial distributions, the p value
                                    remains the same
        #   The new n value is the sum of the n values of the two
                                    distributions.
        new_distr = Binomial()
        new_distr.prob = self.prob
        new_distr.size = self.size + other.size
        new_distr.calculate_mean()
        new_distr.calculate_stdev()

    def __repr__(self):

        """Function to output the characteristics of the Binomial
                                    instance

        Args:
        None

        Returns:
        string: characteristics of the Gaussian

        """
        # TODO: Define the representation method so that the output
                                    looks like
        #       mean 5, standard deviation 4.5, p .8, n 20
        #
        #       with the values replaced by whatever the actual
                                    distributions   values are
        #       The method should return a string in the expected
                                    format
        return f'mean {self.mu}, standard deviation {self.stdev}, p
                                     {self.prob}, n {self.size}'
```

5.6 Uploading a Package to PyPI

- Recommended to create a virtual environment (almost like a silo) to test installing packages (won't affect main Python installation)

- Virtual environments:
  - Conda: manages packages, manages environments
  - conda create –name environmentname
  - source activate environmentname
  - conda install numpy
  - Pip and Venv: Pip is a package manager, Venv is an environment manager that comes preinstalled with Python 3
  - Recommended to create a Conda environment and install Pip simultaneously
    - conda create -–name environmentname pip
  - Pip with Venv work as expected, used for generic software development projects including web development
    - python -m venv venv_name
    - source venv_name/bin/activate (activates virtual environment)

6 Introduction to Web Development