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

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