Python and Datascience Workshop

Python Tutorial

Author: Mohammad Akradi [1]

[1] Institute of Medical Science and Technology, Shahid Beheshti University, Tehran, Iran

Session 2a

In this session, we will focus on clean code concepts and data manipulation. so, I will teach you what is object-oriented programming and how to implement that in our project. then, we will go through pandas library to learn deal with tabular data.

---

1. Clean Code Concepts

Reference

Code is clean if it can be understood easily – by everyone on the team. Clean code can be read and enhanced by a developer other than its original author. With understandability comes readability, changeability, extensibility and maintainability.

General rules

1. Follow standard conventions.
2. Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
3. Boy scout rule. Leave the campground cleaner than you found it.
4. Always find root cause. Always look for the root cause of a problem.

Design rules

1. Keep configurable data at high levels.
2. Prefer polymorphism to if/else or switch/case.
3. Separate multi-threading code.
4. Prevent over-configurability.
5. Use dependency injection.
6. Follow Law of Demeter. A class should know only its direct dependencies.

Understandability tips

1. Be consistent. If you do something a certain way, do all similar things in the same way.

2. Use explanatory variables.
3. Encapsulate boundary conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place.
4. Prefer dedicated value objects to primitive type.
5. Avoid logical dependency. Don't write methods which works correctly depending on something else in the same class.
6. Avoid negative conditionals.

Names rules

1. Choose descriptive and unambiguous names.
2. Make meaningful distinction.
3. Use pronounceable names.
4. Use searchable names.
5. Replace magic numbers with named constants.
6. Avoid encodings. Don't append prefixes or type information.

Functions rules

1. Small.
2. Do one thing.
3. Use descriptive names.
4. Prefer fewer arguments.
5. Have no side effects.
6. Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

Comments rules

1. Always try to explain yourself in code.
2. Don't be redundant.
3. Don't add obvious noise.
4. Don't use closing brace comments.
5. Don't comment out code. Just remove.
6. Use as explanation of intent.
7. Use as clarification of code.
8. Use as warning of consequences.

Source code structure

1. Separate concepts vertically.
2. Related code should appear vertically dense.
3. Declare variables close to their usage.
4. Dependent functions should be close.

5. Similar functions should be close.
6. Place functions in the downward direction.
7. Keep lines short.
8. Don't use horizontal alignment.
9. Use white space to associate related things and disassociate weakly related.
10. Don't break indentation.

Objects and data structures

1. Hide internal structure.
2. Prefer data structures.
3. Avoid hybrids structures (half object and half data).
4. Should be small.
5. Do one thing.
6. Small number of instance variables.
7. Base class should know nothing about their derivatives.
8. Better to have many functions than to pass some code into a function to select a behavior.
9. Prefer non-static methods to static methods.

Tests

1. One assert per test.
2. Readable.
3. Fast.
4. Independent.
5. Repeatable.

Code smells

1. Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
2. Fragility. The software breaks in many places due to a single change.
3. Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
4. Needless Complexity.
5. Needless Repetition.
6. Opacity. The code is hard to understand.

---

2. Classes

classes are pillar of `object oriented programming (OOP)`. OOP is highly concerned with code organization, reusability, and encapsulation; it means, to have a clean code, we should use OOP.

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have `attributes` attached to it for maintaining its state. Class instances can also have `methods` (defined by its class) for modifying its state.

In the first example, we define a class named Employee without any attributes and methods. then we create some instances of this class.

- Example #1:

```
class Employee:
    pass

emp_1 = Employee()
emp_2 = Employee()

print(emp_1)
print(emp_2)
```

although our class have no attributes, but we can add attributes to the objects of Employee type.

- Example #2:

```
class Employee:
    pass

emp_1 = Employee()

emp_1.first_name = 'Mohammad'
emp_1.last_name = 'Akradi'
emp_1.email = 'mohammad.akradi96@gmail.com'

print(emp_1.email)
```

but it's not optimized. attributes' name may differ between different instances of Employee class, so, it looks better if we setup these attributes within the class. each class should have an `__init__` method that gets main attributes.

- Example #3:

```
class Employee:

    def __init__(self, first, last, email):        # self argument gets
the instance name automatically
        '''
         initialize your instance with his/her first name, last name and
email
         "first": gets first name
         "last": gets last name
         "email": gets email
        '''

        self.first_name = first
        self.last_name = last
        self.email = email
        self.name = first + ' ' + last


emp_1 = Employee('Mohammad', 'Akradi', 'mohammad.akradi96@gmail.com')

print(emp_1.name)
```

know we want to define a method for full name displaying:

- Example #4:

```
class Employee:

    def __init__(self, first, last, email):        # self argument gets
the instance name automatically
        '''
         initialize your instance with his/her first name, last name and
email
         "first": gets first name
         "last": gets last name
         "email": gets email
        '''

        self.first_name = first
        self.last_name = last
        self.email = email

    def full_name(self):
        return self.first_name + ' ' + self.last_name


emp_1 = Employee('Mohammad', 'Akradi', 'mohammad.akradi96@gmail.com')
```

```
print(emp_1.full_name()
```

know we want to define another method which gets extra information about our instance:

- Example #5:

```
class Employee:

    def __init__(self, first, last, email):        # self argument gets
the instance name automatically
        '''
         initialize your instance with his/her first name, last name and
email
         "first": gets first name
         "last": gets last name
         "email": gets email
        '''

        self.first_name = first
        self.last_name = last
        self.email = email

    def full_name(self):
        return self.first_name + ' ' + self.last_name

    def birth_info(self, location, birth_year, birth_month, birth_day):
        self.location = location
        self.birth_year = birth_year
        self.birth_month = birth_month
        self.birth_day = birth_day

        self.birth_date = str(birth_year) + "/" + str(birth_month) + "/" +
str(birth_day)


emp_1 = Employee('Mohammad', 'Akradi', 'mohammad.akradi96@gmail.com')
emp_1.birth_info("Sanandaj", 1996, "8", "fourth")

print(emp_1.birth_date)
```

to avoid adding information in different formats, you should add discriptions to the method.
know for better understanding, we continue with another class example:

- Example #6:

```
class Dog:

    def __init__(self, name):
        self.name = name

    def bark(self):
        return "Bark! Bark!"

    # maybe we want to reset name of an instance of a class:

    def set_name(self, name):
        self.name = name

dog_1 = Dog("Joe")
dog_1.set_name("Hapoo")
print(dog_1.bark)
```

sometimes, we have to define some classes. for example, imaginge a high school, we would like to define a class for students and another one for courses.

- Example #7:

```
class Student:

    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def get_grade(self):
        return self.grade

class Course:

    def __init__(self, name, max_students):
        self.name = name
        self.max_students = max_students
        self.students = []

    def add_student(self, student):
        if len(self.students) < self.max_students:
            self.students.append(student)
            return True
        return False

    def get_average_grade(self):
        value = 0
        for student in self.students:
```

```
            value += student.get_grade()

        return value / len(self.students)

student_1 = Student("Mohammad", 26, 95)
student_2 = Student("Ali", 24, 100)
student_3 = Student("Fateme", 27, 96)
student_4 = Student("Babak", 31, 15)

course_1 = Course("Python", 3)
course_1.add_student(student_1)
course_1.add_student(student_2)
course_1.add_student(student_3)
course_1.add_student(student_4)
```