National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2024/2025

Additional:
OOP Practices

Background

This document serves as an unofficial guide to reinforce students' understanding of Object-Oriented Programming (OOP) principles prepared by Zhu Ming. Participation in this practice is optional, and students are encouraged to work through the exercises at their own pace to deepen their understanding of OOP concepts.

Recall

To wrap up the takeaways from CS1010S, a primary theme is the how to solve problem using computer code and how to solve it properly following best practices. Best Practices includes functional abstraction, data encapsulation, and the establishment of an abstraction barrier.

Functional Abstraction and Abstract Data Types (ADTs)

Before learning Object-Oriented Programming (OOP), we achieve abstraction through Abstract Data Types (ADTs), which rely on functions to encapsulate and manage data. ADTs emphasize what an object does, rather than how it does it, focusing on a clear interface and hiding the details of the implementation.

In an ADT, we typically have three main types of functions:

- Constructors: Create a new instance of the ADT.
- Accessor/Mutator Functions: These allow controlled interaction with the ADT, following defined rules. Accessors retrieve data, while mutators modify it.
- Operation Functions: These perform the intended tasks of the ADT, using and modifying the encapsulated data in meaningful ways.

While functional approach of ADTs are a good step toward encapsulation, they have notable drawbacks when implemented in a purely functional style:

- Weak Data Encapsulation: Since data in ADTs isn't tightly bound to the ADT itself, it's exposed to the environment. This can lead to accidental misuse or manipulation, as the data isn't strictly hidden from the rest of the codebase.
- Loose Coupling: In an ADT implemented through functions, the accessor, mutator, and operation functions are only loosely coupled to the data itself. ADTs can be used in a variety of ways, but this flexibility can lead to unintended side effects or errors. The lack of a strict interface can cause user to misuse the ADT, increasing the risk of mismanagement or unintended side effects.

Object-Oriented Programming (OOP): A Practical Solution to Encapsulation and Complexity

Object-Oriented Programming (OOP) addresses these limitations by tightly coupling data and behavior within a class. In OOP, a class defines an object's attributes (data properties) and methods (functions representing behaviors). By doing this, OOP achieves two main objectives:

- Encapsulation: OOP hides data within an object, protecting it from outside interference and misuse.
- Clear Boundaries through Abstraction: Each object acts as a self-contained unit, with a well-defined interface for interacting with it, thereby establishing an abstraction barrier.

When a user initializes an object from a class, they receive an object instance, which can be thought of as a "black box" containing both data and behavior. The user interacts with this black box only through the methods provided by the class, without needing to know or access its internal state.

In stricter languages like Java, this encapsulation can be enforced rigorously by making attributes private or protected, preventing direct access. It will throw an error if the user tries to access the attribute directly!!! This is sometimes less enforced in Python, but the principle should be followed: an object instance is a secure, self-contained unit that contains its own data (attributes) and exposes only the methods intended for external interaction (NO BREAKING ABSTRACTION IS ALLOWED).

This is the essence of OOP! Encapsulation and Abstraction.

If you are confused, don't worry! We will go through some examples to illustrate these concepts. You can come back to this part after you finish the examples to have a better understanding of the concepts.

What is Class? What is Object/Instance?

Class: A class is a blueprint for creating objects (a particular data structure). A class doesn't represent any specific item but instead defines the properties and behaviors that similar items, or objects, will share.

Object/Instance: An object/instance is an instance of a class. When a class is defined, no memory is allocated (it is just code). When an object of the class is created/initializes, memory is allocated (it is a specific and real entity).

Here is a more intuitive way to understand the concept of class and object.

The idea of **Human** is a class, it is just a concept, a blueprint. No real person exists yet. Suppose I refer to a person named **Zhu Ming**, then this person Zhu Ming is a instance of the class **Human**. This person has a name, age, etc. These are the attributes of the object **Zhu Ming**. The object **Zhu Ming** can also perform actions like moveing, talking, eating, etc. These are the methods of the object **Zhu Ming**. This Zhu Ming is a real entity, an specific instance of Human.

How to write a class in Python?

```
class Human:
    # __init__ is the CONSTRUCTORS
    def __init__(self, name, age):
        # ATTRIBUTES
        self.name = name
        self.age = age

    # METHODs
    def move(self):
        print(f"{self.name} is walking.")

    def talk(self):
        print(f"{self.name} is talking.")
```

If you wonder why there is alwyas a `self` ***in the method/function parameters, you can think of it as syntax of Python for class method definition. Short explanation is it is because when you call a method of an object, the object itself is passed as the first argument.***
Here is a more details explanation:
stackoverflow.com/questions/2709821/what-is-the-purpose-of-the-self-parameter-why-is-it-needed

What does the `self.name` ***do? What if i define my move method like the following?***

```
    def move():
        print(name + " is walking.")
```

NO, this is wrong. If you just refer to `name`***, Python will think that you are referring to a local variable in the*** `move` ***method which doesn't exists. You should always refer to the attribute of the object by using*** `self.name`***. The*** `self` ***is a reference to the object itself and the*** `.name` ***part is referring to the attribute in the defined in the constructor.***

How to call a method when writing the class?

Suppose we would like to define an additional method `move_and_talk` in the class `Human`.

```
class Human:
    # ... code from previous part

    def move_and_talk(self):
        # Embrassing the idea of reusability
        # We should call the move and talk method we defined above
        self.move()
        self.talk()
```

What does the `self.move()` do? Can I just call `move()`?
NO. Similar to above argument. If you just call `move()`, Python will think that you are calling a local function (rigorously speaking it should be variable, not function) named `move` in which doesn't exists in the `move_and_talk` function.

How to create an object/instance of a class in Python?

```
# create an object of the class Human
zhu_ming = Human("Zhu Ming", 20, 180, 70)
# what python does when you run this code is
# 1. Find the Human class
# 2. Then find the __init__ method in the Human class
# 3. Create a object with the input you passed in
# 4. Return the Huamn object
```

How to call a method of an object?

```
# call the move method of the object zhu_ming
zhu_ming.move()
# what python does when you run this code is
# 1. From the object zhu_ming, find the class it belongs to (which is Human)
# 2. Find the move method in the Human class
# 3. Call the move method
```

If you manage to followed until here, you have a good understanding of the basic concept of class and object in Python.

Inheritance and Polymorphism: additional properties we wish to have in OOP

When computer scientists first come up with the idea of OOP, they were trying to mimic the real world and simplify the complexity of the real world using the notion of object where object hide complexity. Notice how after we defined a class, we can just intuitive understand the object and its methods without caring about the implementation details.

Inheritance

Sooner and later, people realize classes might share similar properties and behaviors. For example, `Student` class and `Teacher` class might share similar properties like name, age, height, weight, etc. and similar behaviors like move, talk, etc.

Imageine the world, where Inheritance doesn't exist :(. We have to define redundant code for each class.

```
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def move(self):
        print(f"{self.name} is walking.")

    def talk(self):
        print(f"{self.name} is talking.")

    # ... talk_and_move

class Student:
    def __init__(self, name, age, height, weight, student_id):
        self.name = name
        self.age = age
        self.height = height
        self.weight = weight
        self.student_id = student_id

    def move(self):
        print(f"{self.name} is walking.")

    def talk(self):
        print(f"{self.name} is talking.")

class Teacher:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def move(self):
        print(f"{self.name} is walking.")

    def talk(self):
        print(f"{self.name} is talking.")
```

We can see that the code is redundant. We have to define the same attributes and methods for each class. This is where Inheritance comes in. We can define a parent class `Human` and let the

`Student` and `Teacher` class inherit the properties and behaviors of the `Human` class.

```
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def move(self):
        print(f"{self.name} is walking.")

    def talk(self):
        print(f"{self.name} is talking.")

# You can do this
class Student(Human):
    def __init__(self, name, age, height, weight):
        # call the constructor of the parent class
        super().__init__(name, age, height, weight)

# OR you can do this
class Teacher(Human):
    pass
    # This work too in this case because it will inherit all the
    # properties and methods of the parent class
    # Meaning it will inherit the __init__, move, and talk method
```

Elegant! Life is much easier now. We can define the `Student` and `Teacher` class with much less code.

What does the* `super().__init__()` *do?
Previously,* `self` *refer to the object itself. Now, the* `super()` *refer to the parent class. The* `super().__init__()` *call the constructor of the parent class.

How to determine whether a class is a subclass of another class?

We can use the idea of check whether a class **is_a** subclass of another class. It should be logically make sense that a `Student` is a `Human` and a `Teacher` is a `Human`.

Logical sense is important in OOP. Suppose we now define a class `Cat`. Although, `Cat` might have similar properties like name and age, similar method like move and talk?? It doesn't make sense that `Cat` is a `Human`. Your code might still work, but it logically make no sense. So inheritance doesn't make sense here.

How to check whether a object is a subclass of another class?

Use the function `isinstance(object, class)` to check whether an object is an instance of a class. If you use `type` function, it will not work because it will only return the class of the object, it will not check whether the object is an instance of the class.

Polymorphism

Polymorphism is the ability of an object to take on many forms. Confusing? Let's see an example.

`Student` and `Teacher` are both `Human`. However, they might have:

- addition attributes like `student_id` for `Student` and `teacher_id` for `Teacher`
- addition behaviours like `study` for `Student` and `teach` for `Teacher`
- behave slightly differently. For example, `Student` might print might run and walk when tired when move.

```
class Student(Human):
    def __init__(self, name, age, student_id):
        super().__init__(name, age) # should use super().__init__
        self.student_id = student_id # new attribute

    def move(self): # same method move but behave differently
        print(f"{self.name} is running.")
        print(f"tired")
        super().move() # call the move method of the parent class

    def study(self): # additional method
        print(f"{self.name} is studying.")

class Teacher(Human):
    def __init__(self, name, age, teacher_id):
        super().__init__(name, age)
        self.teacher_id = teacher_id # new attribute

    def teach(self): # additional method
        print(f"{self.name} is teaching.")
```

Hopefully, you can see the idea of polymorphism here. The `Student` and `Teacher` class have the same method `move` but they behave differently. The `Student` class has an additional method `study` and the `Teacher` class has an additional method `teach`. This is the idea of polymorphism. The same method can behave differently.

That is all needed to know about OOP. This serve as a brief introduction to OOP.

Practice

Point Class Implement a class `Point` that represents a point in 2D space. Implement getter method which `get_x` and `get_y` which return the x and y coordinate of the point. Implement a `distance` method which takes in another point and return the distance between the two points, you must ensure that the input is a `Point` object.

Circle Class Implement a class `Circle` that represents a circle in 2D space. Implement a constructor that takes in a point and a radius. Implement a `get_center` method which return the center of the circle. Implement a `get_radius` method which return the radius of the circle. Implement `area` method which return the area of the circle. Implement a `circumference` method which return the circumference of the circle. Implement a `is_same` method which takes in another circle and return whether the two circles are the same (same value for center and radius).

ColouredCircle Class Implement a class `ColouredCircle` that represents a coloured circle in 2D space. Implement a constructor that takes in a point, a radius, and a colour. Implement a `get_color` method which return the colour of the circle. Implement a `get_center` method which return the center of the circle. Implement a `get_radius` method which return the radius of the circle. Implement `area` method which return the area of the circle. Implement a `circumference` method which return the circumference of the circle. Implement a `is_same` method which takes in another circle and return whether the two circles are the same (same value for center and radius and colour).

Answer will only be provided after you submit your attempt to me, or you join the consultaiton. (or ask chatGPT)