# CSCI 3180 HW 2 Report

ZHANG Jingze (SID: 1155107857)

## Task 2

### 1. Advantages:

Dynamic typing can be extremely useful if we need to efficiently contsruct the code that can work in various conditions. The efficency of dynamic typing heavily depends on feature that the defined function can return and take parameters of various types without incresing the code complexity a lot and that the entity in data structures can be of all possible types (or classes). Those features can help to remarkably simplify the code.

- Example 1:

```
def printInFormat(obj):
  print("Name: " + obj.name + ", Value: " + str(obj.value))
```

In the code above, we can fully use the dynamic typing of Python to construct a function that can print the instance variables in format for the all objects with attributes "name" and "value". In static typing languages, we need to specify the class type for "obj" and have to define distinct function for distinct classes. However, in Python, we can define only one function for all the classes as long as it has attributes "name" and "value".

- Example 2:

Assume we have defined 3 distinct classes as "Sword", "Shield" and "Elixir", with dynamic typing we can use a list in Python to store all the items collected by the soldier, the example code is like:

```
items = [Sword("Ice Sword"), Sword("Fire Sword"), Shield("Normal Shield"), Elixir(name =
"Life Elixir", healing = 100)]
```

In such case, we do not need to consider inheritance where we need to unify the superclass to store all the objects in a static list. The dynamic typing allows the list to be added with a newly defined object where we need not make it of the same superclass as other items.

### 2. Disadvantages:

The dynamic typing also makes it difficult to do compiling time type checking for Python. The dynamic feature can cause various bugs in some conditions. Example code is like:

```
def printInCell(strVar):
  print("[" + strVar + "]")  # print the strVar inside a cell
```

This function will print the parameter "strVar" inside a cell. If the type of "strVar" is string, the function works well. However, if we mistakenly call the function with "strVar" as an integer, an error will be raised and a bug was generated.

## Task 3

### Scenario 1

```
// Java code in task 3, SaveTheTribe.java : start()
if (occupiedObject instanceof Monster) {
  ((Monster)occupiedObject).actionOnSoldier(this.soldier);
} else if (occupiedObject instanceof Spring) {
  ((Spring)occupiedObject).actionOnSoldier(this.soldier);
} else {
  // some command here ...
}
```

```python
# Python code in task 3, SaveTheTribe.py : start()
if occupiedObject is not None:
  occupiedObject.actionOnSoldier(self.soldier)
else:
  pass # some command here ...
```

In Java code, if we need to call the method of an object, we need to force type conversion for the object before we call the method. Also, we have to use the inheritance feature to allow the type coversion for the object. However, in Python, we can call the method of an object as long as the object has the method. This duck typing feature of Python allows us to remarkably simplify the content and structure of our code.

**Scenario 2**

```java
// Java code in task 3, SaveTheTribe.java : SaveTheTribe()
this.monsters = new Monster[7];
```

```python
# Python code in task 3, SaveTheTribe.py : __init__()
self.monsters = [None for _i_ in range(7)]
```

In Java code, if we need to initialize an instance variable, we need to specify the class type of the instance variable. And if we need to adjust the code to suit more objects for the variable, we have to modify many parts of the code. In Python, we can initialize the instance variable without speccifying the class type for the variable. And if we need to assign an object of a new type, we can directly intialize with that new type object.

**Task 4**

In Java code, if we need to reuse the code after changing the class type of some relative objects or adding a new class object, we need to modify the type conversion in some branches or add another branch in the "if-else" expression. For example, if we need to assign the variable "occupiedObject" with a new object (such as "Task4Monster" or "Task4Merchant"), we need to change the type conversion from "Monster" to "Task4Monster" or add another branch for the new class "Task4Merchant". The original code is shown in Task 3 scenario 1 and the changed example code is like:

```java
if (occupiedObject instanceof Task4Monster) {
  ((Task4Monster)occupiedObject).actionOnSoldier(this.soldier);
} else if (occupiedObject instanceof Spring) {
  ((Spring)occupiedObject).actionOnSoldier(this.soldier);
} else if (occupiedObject instanceof Task4Merchant) {
  ((Task4Merchant)occupiedObject).actionOnSoldier(this.soldier);
} else {
  // some command here ...
}
```

However, for Python, the code can remain the same as long as the newly added object has the method "actionOnSoldier". The code for Python in this part is the same as the Python code in Task 3 scenario 1. In this case, passing object in Python does not require to specify the class type for the object, thus we can reuse the Python code after we added another class and eliminate our effort to keep the class type consistent if we need to change the class type of some objects. Generally, the duck typing feature of Python could help to reduce the requirement on the inheritance while handling obejcts of various classes compared to programming languages like Java.