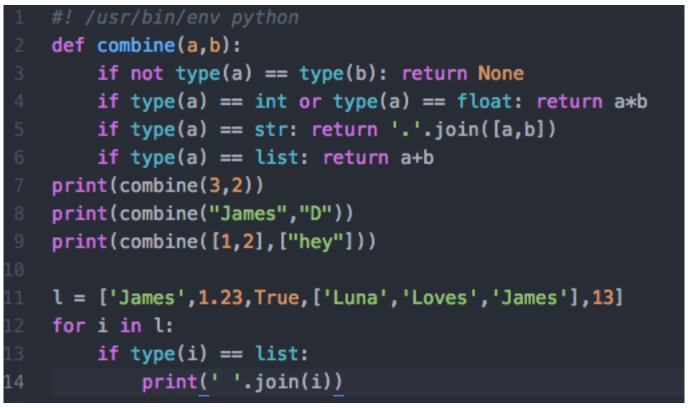
CSCI3180 Report for assignment 2

DENZG Zihao 1155077093

March 9, 2018

Task 2: Advantage of Dynamic Typing



```python
#! /usr/bin/env python
def combine(a,b):
    if not type(a) == type(b): return None
    if type(a) == int or type(a) == float: return a*b
    if type(a) == str: return '.'.join([a,b])
    if type(a) == list: return a+b
print(combine(3,2))
print(combine("James","D"))
print(combine([1,2],["hey"]))

l = ['James',1.23,True,['Luna','Loves','James'],13]
for i in l:
    if type(i) == list:
        print(' '.join(i))
```

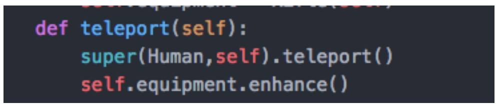
As shown in the code above:

(1) Functions can be applied on different types of arguments: Combine() is a function that takes two parameters a,b as input. If they are both integer or floating numbers, it will return the multiplication of them. If they are both strings, it will use single dot as delimiter to join them. If they are both lists, it will return the combined list of them.

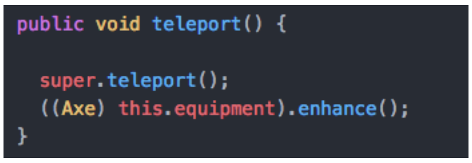
(2) Possibilities of mixed type collection data structures. l is a collection of string, float, boolean, and integer. The following for loop examines each element inside l and do some operations on those elements of list type.

Disadvantage: There is no type checking during compile time, which means that type error cannot be caught before execution. This will weaken the security of program because the program will terminate unexpectedly when there is a type error during run time. Also it takes more time to run a program written in dynamically typed language because all type checking are done during the execution.

Task 3: Survival Game–Python vs Java

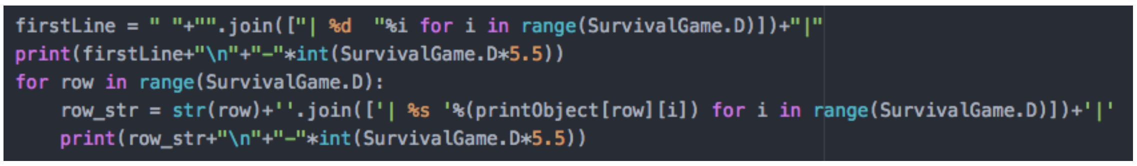


```python
def teleport(self):
    super(Human,self).teleport()
    self.equipment.enhance()
```



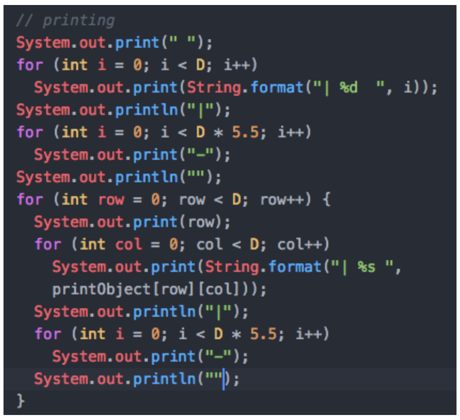
```java
public void teleport() {

  super.teleport();
  ((Axe) this.equipment).enhance();
}
```

(1) Python implements dynamic typing that supports duck typing. We don't have to declare data types for both class variables and instance variables; and whenever a object has some attributes or methods, we can use them through the object without knowing what class it exactly belongs to. For example in above code, Player.equipment can be either of class Rifle or of class Axe. In Java, we have to first declare an Weapon class object for Player.equipment and when we implement teleport() for sub-classes of Player, we have to cast the equipment to either class Rifle or class Axe. But in Python, equipment is dynamically typed and we can always call equipment.enhance() without knowing what class of equipment it is, because both Rifle and Axe has method enhance().

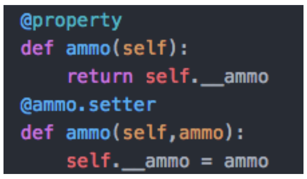


```python
firstLine = " "+"".join(["| %d  "%i for i in range(SurvivalGame.D)])+"|"
print(firstLine+"\n"+"-"*int(SurvivalGame.D*5.5))
for row in range(SurvivalGame.D):
    row_str = str(row)+''.join(['| %s '%(printObject[row][i]) for i in range(SurvivalGame.D)])+'|'
    print(row_str+"\n"+"-"*int(SurvivalGame.D*5.5))
```

(2) Python provides many powerful functions and operands for string manipulation. For example we can use substr * n in python to generate a string in which sub-string substr appears n times; we can also use delimiter.join(list) to build a string with all strings in list, separated by delimiter. Java don't support those methods, so we have to explicitly use loops to achieve above task, which is a lot more tedious and inconvenient. In above code, Python uses 5 lines to print game board while Java uses 17 lines including 3 loops as shown below.



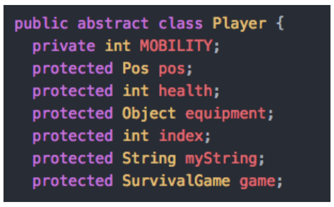
```java
// printing
System.out.print(" ");
for (int i = 0; i < D; i++)
  System.out.print(String.format("| %d  ", i));
System.out.println("|");
for (int i = 0; i < D * 5.5; i++)
  System.out.print("-");
System.out.println("");
for (int row = 0; row < D; row++) {
  System.out.print(row);
  for (int col = 0; col < D; col++)
    System.out.print(String.format("| %s ",
    printObject[row][col]));
  System.out.println("|");
  for (int i = 0; i < D * 5.5; i++)
    System.out.print("-");
  System.out.println("");
}
```



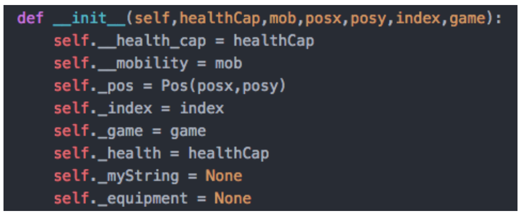
```python
@property
def ammo(self):
    return self.__ammo
@ammo.setter
def ammo(self,ammo):
    self.__ammo = ammo
```

(3) Python allows users to use property decorators. In above code, __ammo is a private instance variable of Rifle class. In Java, to access and modify such private variables, we need to implement some public methods like getAmmo() and setAmmo(); but by using property and setter decorators in Python, we can directly call object.ammo to access and modify such private variables. This improves the readability of programs.

Task 4: More on Dynamic Typing and Duck Typing

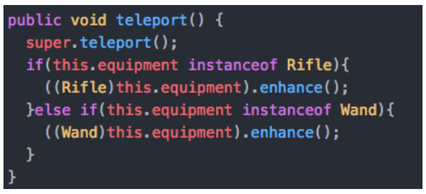


```java
public abstract class Player {
  private int MOBILITY;
  protected Pos pos;
  protected int health;
  protected Object equipment;
  protected int index;
  protected String myString;
  protected SurvivalGame game;
```

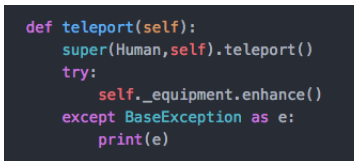


```python
def __init__(self,healthCap,mob,posx,posy,index,game):
    self.__health_cap = healthCap
    self.__mobility = mob
    self._pos = Pos(posx,posy)
    self._index = index
    self._game = game
    self._health = healthCap
    self._myString = None
    self._equipment = None
```

Dynamic typing helps when we extend functionality to achieve Task 4. In Task 3, Player.equipment is of class Weapon, but in Task4 it can be of class Wand. In such case, we need to check potential type problems in Java. For example, we need to change the original 'protected Weapon equipment' to 'protected Object equipment', otherwise in compile time there will be type error when we assign a Wand to the last one of each race. This checking process can be difficult, time-consuming, and needs to be done again and again when we extend functionality of previous program. Python helps us avoid this trouble because its variable/object can be of any type/class during run time. In above codes, Python didn't care whether we implement further classes, because _equipment can be any class during run time; once we put it into the __init__ of Player, there is no more modification needed for future development.



```java
public void teleport() {
  super.teleport();
  if(this.equipment instanceof Rifle){
    ((Rifle)this.equipment).enhance();
  }else if(this.equipment instanceof Wand){
    ((Wand)this.equipment).enhance();
  }
}
```



```python
def teleport(self):
    super(Human,self).teleport()
    try:
        self._equipment.enhance()
    except BaseException as e:
        print(e)
```

For example in above code, Human.equipment can be either of class Rifle or of class Wand. In Java, when we implement teleport() for Human, we have to determine the class of equipment before we call equipment.enhance(). But in Python, equipment is dynamically typed and we can always call equipment.enhance() without knowing what class of equipment it is, because both class Rifle and class Wand has implemented its own method enhance(). Duck typing also helps when we call Player.equipWand() for the last one in Chark and Human: because both Chark and Human has method equipWand(), so we don't need to do class casting like in Java, we just straightly call the method.

To dig into deeper about duck typing in Python, it is in coherence with the EAFP principle of Python, which stands for easier to ask forgiveness than permission. Instead of LBYL principle (look before you leap) in Java, EAFP don't care whether a object has certain method or attribute; it just tries directly to call the method or attribute. If the call fails, which implies the object doesn't have the attribute or method, we just catch the exception. As we see, compared to LBYL, EAFP saves time for condition checking (like above Java code) and type/class casting. Duck typing helps programmers to adopt EAFP principle and achieve greater efficiency.