```
/*
*CSCI3180 Principles of Programming Languages
*
*--- Declaration ---
*
*I declare that the assignment here submitted is original except for source
*material explicitly acknowledged. I also acknowledge that I am aware of
*University policy and regulations on honesty in academic work, and of the
*disciplinary guidelines and procedures applicable to breaches of such policy
*and regulations, as contained in the website
*http://www.cuhk.edu.hk/policy/academichonesty/
*
*Assignment 2
*Name : YU Si Hong
*Student ID : 1155141630
*Email Addr : shyu0@cse.cuhk.edu.hk
*/
```

Task 2: Demonstrating Advantages of Dynamic Typing

1. More generic code can be written. In other words, functions can be defined to apply to arguments of different types.

- Scenario: To add two target variables together. Then try two examples as follow.
- Code Segment (python).

```python
def add(target1, target2):
    return target1 + target2

print(add("hello ", "world "))
print(add(3.14, 100))
```

- Result.

```
hello world
103.14
```

2. Possibilities of mixed type collection data structures.

- Scenario: Assume list is a 2d array and each row stores the name (string), phone number (integer) and phone balance (float). Then print the first row.
- Code Segment (python).

```python
list = [[] for i in range(100)]
list[0].append("Name")
list[0].append(12345678)
list[0].append(100.00)

print(list[0])
```

- Result.

```
['Name', 12345678, 100.0]
```

3. A Disadvantage: Without type definition in code, the programmer is required to remember all the types of items, or take the time to find its original call. This disadvantage is magnified in complex projects and also shows up in understanding code written by someone else.

- Scenario: To check if a number is positive.

- Code Segment (python).

```python
def is_pos(i):
    if i > 0:
        return True
    else:
        return False
```

- Result in error if you forget i should be a number.

Task 4: Demonstrating Advantages of Duck Typing

Duck Typing between Trap and GameCharacter

- In NewEngine.py that stores Trap class where the GameCharacter class should be stored, namely ***initcell.set_occupant(trap)***, resulting in a chain reaction of duck typing.

```python
for tno in range(num_of_traps):
    ...
    trap = Trap(traprow, trapcol)
    initcell = self._map.get_cell(traprow, trapcol)
    initcell.set_occupant(trap)
    trap.occupying = initcell
```

1. a. In finding the occupants around the volcano of Volcano.py, checking the name of occupant by using ***occ.name*** may use eithor name of Trap or name of GameCharacter, which is a Duck Typing.

```python
def act(self, map):
    ...
        cells = map.get_neighbours(self._row, self._col)
        for obj in cells:
            occ = obj.occupant
            if occ != None:
                if occ.name == "Goblin":
                    occ.active = False
                    occ.occupying.remove_occupant()
                elif occ.name == "Player":
                    occ.hp -= 1
```

1. b. So as the ***obj.occupant.name*** in finding the traps around the player of NewEngine.py.

```python
def print_info(self):
    ...
    cells = self._map.get_neighbours(self._player.row, self._player.col)
    for obj in cells:
        if obj.occupant != None:
            if obj.occupant.name == "Trap":
                num += 1
    print("Oxygen: %d, HP: %d, Trap: %d" % (self._player.oxygen, self._player.hp, 
num))
```

2. In setting occupant in Cell.py, ***self.occupant.interact_with(obj)*** is also Ducking Typing because ***interact_with()*** can from either Trap or GameCharacter.

```
    def set_occupant(self, obj):
        ...
        if self.occupant == None or self.occupant.interact_with(obj):
            ...
        ...
```

3. In displying the cell in Cell.py, ***self.occupant.display()*** is also Duck Typing because ***display()*** can from either Trap or GameCharacter.

```
    def display(self):
        ...
        if self.occupant == None:
            print("%s   \033[0m   " % (self._color), end='')
        else:
            print("%s %s%s \033[0m   " % (self._color, self.occupant.display(), 
self._color, end='')
```

Duck typing makes coding more flexible and convenient.

- Scenario: To call methods with the same name in different classes. e.g. There are two instances, dog and car of two classes, Animal and Vehicle respectively, and I want to print the status of items in the street. The following are python implement and java implement respectively.

```
class Animal:
    def status(self):
        print("An animal runs on the street.")
class Vehicle:
    def status(self):
        print("A vehicle drives on the street.")
def in_the_street(obj):
    obj.status()

dog=Animal()
car=Vehicle()
in_the_street(dog)
in_the_street(car)
```

```java
class Animal {
    public void status() {
        System.out.println("An animal runs on the street.");
    }
}
class Vehicle {
    public void status() {
        System.out.println("A vehicle drives on the street.");
    }
}
public class street {
    public static void in_the_street(Object obj) {
        if (obj instanceof Animal)
            ((Animal) obj).status();
        if (obj instanceof Vehicle)
            ((Vehicle) obj).status();
    }
    public static void main(String[] args) {
        Animal dog = new Animal();
        Vehicle car = new Vehicle();
        in_the_street(dog);
        in_the_street(car);
    }
}
```

- For pyhton, it's more flexible. It has dynamic typing and duck typing that make it possible to switch automatically without forcing items to be the same type. For java, on the other hand, requires us to define each item and switch between different types. Python is superior in terms of code length and simplicity, and it can get things done quickly. But Java looks more cautious.