

CSCI3180 Assignment 2 Report

Nikunj Taneja (SID: 1155123371)

8 Mar 2022

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.

This can be demonstrated using a simple example of adding two integers or concatenating two strings. In a statically typed language like Java or C/C++, it will not be possible to have a single function that supports both these functionalities even though the function body would be the same. Whereas in a dynamically typed language like Python, we can easily write such a function since we don't have to write the return type or the argument type.



```
1 #include <iostream>
2 using namespace std;
3
4 string addStrings(string a, string b) {
5     return a + b;
6 }
7
8 int addInts(int a, int b) {
9     return a + b;
10 }
11
12 int main() {
13     cout << addInts(2, 3) << endl;
14     cout << addStrings("2", "3") << endl;
15     return 0;
16 }
17
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
5
23
>
```

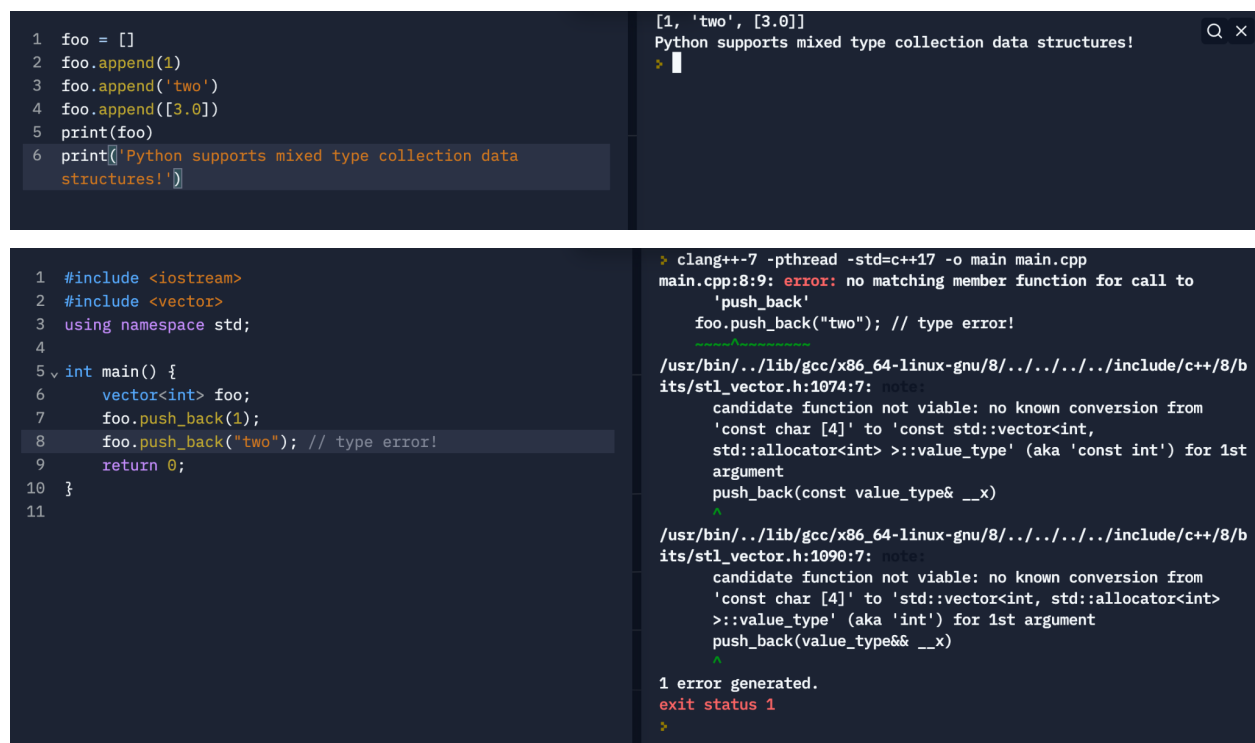
```
1 def add(a, b):
2     return a + b
3
4 print(add(2, 3))
5 print(add("2", "3"))
6
```

```
5
23
>
```

Figure 1. Functions with arguments of different types

2. Possibilities of mixed type collection data structures.

The simplest example to demonstrate this advantage is the Python list data structure. It behaves almost like the `std::vector` in C++ except the fact that it supports heterogeneous types in its elements. So, while it is possible to have an array or list that has elements of different types in a dynamically typed language like Python, it's not possible in a statically typed language like C++ since, for example, in C++ one has to specify what type of objects will be in the `std::vector` during its declaration.



The screenshot displays two code snippets side-by-side. The left snippet is Python code that creates a list 'foo' and appends elements of different types (integer, string, and list), then prints the list. The right snippet shows the output of the Python code: '[1, 'two', [3.0]]' and a confirmation message. Below this, the left snippet shows C++ code attempting to use a `std::vector<int>` with `push_back` on a string and an integer, which results in compilation errors. The right snippet shows the corresponding C++ compiler error messages, highlighting the type mismatch and the lack of a matching member function for the calls.

```
1 foo = []
2 foo.append(1)
3 foo.append('two')
4 foo.append([3.0])
5 print(foo)
6 print('Python supports mixed type collection data structures!')
```

```
[1, 'two', [3.0]]
Python supports mixed type collection data structures!
>
```

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> foo;
7     foo.push_back(1);
8     foo.push_back("two"); // type error!
9     return 0;
10 }
11
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
main.cpp:8:9: error: no matching member function for call to 'push_back'
    foo.push_back("two"); // type error!
        ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/8/../../../../include/c++/8/bits/stl_vector.h:1074:7: note:
candidate function not viable: no known conversion from 'const char [4]' to 'const std::vector<int,
std::allocator<int> >::value_type' (aka 'const int') for 1st argument
push_back(const value_type& __x)
^
/usr/bin/../lib/gcc/x86_64-linux-gnu/8/../../../../include/c++/8/bits/stl_vector.h:1090:7: note:
candidate function not viable: no known conversion from 'const char [4]' to 'std::vector<int, std::allocator<int>
>::value_type' (aka 'int') for 1st argument
push_back(value_type&& __x)
^
1 error generated.
exit status 1
>
```

Figure 2. Mixed type collection data structures.

Task 4: Demonstrating Advantages of Duck Typing

Scenario 1: run() method of Engine (NewEngine in Python)

Since duck typing is used here (see Figure 3a and 3b below), there is no need to check the type of obj in the Python implementation whereas, in the Java implementation, we need to typecheck each actor and call the respective methods, even though the methods are the same, namely the active getter method and the act method. Hence, this makes the code more flexible, easier to maintain and considerably reduces the

development time since the number of lines of code that need to be written is lesser.

```
96     public void run() {
97         this.printInfo();
98         while (this.state() == 0) {
99             for (int i = 0; i < this.actors.size(); ++i) {
100                 if (this.actors.get(i) instanceof GameCharacter) {
101                     if (((GameCharacter)this.actors.get(i)).getActive()) {
102                         ((GameCharacter)this.actors.get(i)).act(this.map);
103                     }
104                 } else if (this.actors.get(i) instanceof Volcano) {
105                     if (((Volcano)this.actors.get(i)).getActive()) {
106                         ((Volcano)this.actors.get(i)).act(this.map);
107                     }
108                 }
109             }
110             this.printInfo();
111             this.cleanUp();
112         }
113         this.printResult();
114     }
```

Figure 3a. Java implementation of Engine's run method (extension)

```
97     def run(self):
98         # main routine of the game
99         self.print_info()
100         while not self.state():
101             for obj in self._actors:
102                 if obj.active:
103                     obj.act(self._map)
104             self.print_info()
105             self.clean_up()
106         self.print_result()
107
```

Figure 3b. Python implementation of NewEngine's run method

Scenario two:

Similarly, in Cell's setOccupant method in Java (see Figure 4a below), we need to typecheck the cell's occupant before calling its interactWith method and eventually setting the occupant. There is no need to do this in the Python implementation (see Figure 4b below) because duck typing allows us to call the interact_with method for both GameCharacter and Trap types, as both the types have that method. This makes the

code much more flexible, easier to write. For example, if we were to add support for another class/type in this method, the Python implementation would need no change while the Java one would need to have another else-if branch to check for the new class type.

```
23
24     public boolean setOccupant(Object obj) {
25         if (this.getOccupant() == null) {
26             this.occupant = obj;
27             return true;
28         } else {
29             if (this.getOccupant() instanceof GameCharacter) {
30                 if (((GameCharacter)this.getOccupant()).interactWith(obj)) {
31                     this.occupant = obj;
32                     return true;
33                 } else {
34                     return false;
35                 }
36             } else if (this.getOccupant() instanceof Trap) {
37                 if (((Trap)this.getOccupant()).interactWith(obj)) {
38                     this.occupant = obj;
39                     return true;
40                 } else {
41                     return false;
42                 }
43             } else {
44                 return false;
45             }
46         }
47     }
```

Figure 4a. Java implementation of Cell's setOccupant method

```
26
27     def set_occupant(self, obj):
28         # TODO: set occupant for the Plain cell
29         #     return whether success or not
30         if self._occupant is None:
31             self._occupant = obj
32             return True
33         else:
34             if self._occupant.interact_with(obj):
35                 self._occupant = obj
36                 return True
37             else:
38                 return False
39         # END TODO
40
```

Figure 4b. Python implementation of Cell's set_occupant method