Tutorial #7 - Let's practice a bit more

Habib Ghaffari

<2022-11-14 Mon>

Contents

1 Introduction

In the first part of this tutorial, we will work on the exercises from the last tutorial. I will explain how you need to work on each question and justify my solutions.

In the second part, we will have an introduction to python classes, how we can define them and how they work. I will also introduce python constructors with some examples related to `__init__` and `__eq__`, which are two handy constructor methods.

2 Questions:

1. Consider the following JavaScript skeletal program:

```
// The main program
var x;
```

```
  // Defined inside main
  function sub1(){
      var x;
      // Defined inside sub1
      function sub2(){
          . . .
      }

  }
// Defined inside main
function sub3(){
  . . .
 }
```

Assume that the execution of this program is in the following unit order:

- `main` calls `sub1`
- `sub1` calls `sub2`
- `sub2` calls `sub3`

Assuming static scoping, in the following, which declaration of `x` is the correct one for a reference to `x`?

- Since the scope is static. The place we deifne a function is become important. Becuase it is going to show us the refernce to the `x`

  i. sub1 (Because in static scope the search is going to happen locally first.) ii. sub1 (There is not local definition of x inside sub2, but sub2 degfined inside sub1 which has a local definiton for x) iii. main (We defined the function sub3 on main so the reference for x is going to be main)

Repeat part a, but assume dynamic scoping:

- In dynamic scope the sequence of calling functions is important.

  i. sub1 ii. sub1 iii. sub1
- Let's write a piece of code in `JavaScript`:

```
var x;
// This is inside main ( The reference here still main)
function sub1() {
 document.write("x = " + x + "");
}
// this is inside main
function sub2() {
 var x;
 // this is inside sub 2
 x = 10;
 sub1();
}
x = 5;
sub2();
```

- What value of `x` is displayed if we use static-scope rules? Why?

  x = 5 (The scope is static. The place we define sub1 becomes important)

- What if we use dynamic-scope rules? Why?

  x = 10 (We find the refernce based on the sequence of calling fucntions)

- Let's write another piece of code in `JavaScript`:

```
var x, y, z;
function sub1() {
var a, y, z;
  function sub2() {
   var a, b, z;
   . . .
  }
  . . .
}
function sub3() {
  var a, x, w;
  . . .
}
```

List all the variables, along with the program units where they are declared, that are visible in the bodies of `sub1`, `sub2`, and `sub3`, assuming static scoping is used. Justify your answer:

- In sub1:
- In sub2:
- In sub3:

1. Consider the following `C` program:

    - C is a static programing language.

```
void fun(void) {
int a, b, c; /* definition 1 */
. . .
while (. . .) {
int b, c, d; /*definition 2 */
. . . <------------- 1
  while (. . .) {
    /* Whatever happening inside a block is going to stay inside it.
    int c, d, e; /* definition 3 */
    . . . <------------- 2
  }
. . . <-------------- 3
}
. . . <---------------- 4
}
```

For each of the four marked points in this function, list each visible variable, along with the number of the definition statement that defines it. Justify your answer?

- Point 1: (Because b c d are already defined localy in `fun`, so when we are redefing the inside `while` block. The local definitions of the variables going to be hidden. a 1 b 2 c 2 d 2
- Point 2: a 1 b 2 c 3 d 3 e 3
- Point 3:
    - (Exactly like point1)
- Pint 4:
    - a 1 b 1 c 1

1. Consider the following skeletal `C` program:

```
void fun1(void); /* prototype */
void fun2(void); /* prototype */
void fun3(void); /* prototype */
void main() {
 int a, b, c;
 . . .
}
void fun1(void) {
 int b, c, d;
 . . .
}
void fun2(void) {
 int c, d, e;
 . . .
}
void fun3(void) {
 int d, e, f;
 . . .
}
```

Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last function called? Include with each visible variable the name of the function in which it was defined. Justify your answer?

(Sequence of function call)

a. `main` calls `fun1`; `fun1` calls `fun2`; `fun2` calls `fun3`. b. `main` calls `fun1`; `fun1` calls `fun3`. c. `main` calls `fun2`; `fun2` calls `fun3`; `fun3` calls `fun1`. d. `main` calls `fun3`; `fun3` calls `fun1`. e. `main` calls `fun1`; `fun1` calls `fun3`; `fun3` calls `fun2`. f. `main` calls `fun3`; `fun3` calls `fun2`; `fun2` calls `fun1`.

Question	Variables	Where Declared
a)	d e f	fun3
	c	fun2
	b	fun1
	a	main
b)		
c)		
d)		
e)		
f)		

1. Consider the following program, written in `JavaScript`-like

syntax:

```
// main program
var x, y, z;
function sub1() {
var a, y, z;
. . .
}
function sub2() {
var a, b, z;
. . .
}
function sub3() {
var a, x, w;
. . .
}
```

Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last subprogram activated? Include with each visible variable the name of the unit where it is declared.

a. `main` calls `sub1`; `sub1` calls `sub2`; `sub2` calls `sub3`. b. `main` calls `sub1`; `sub1` calls `sub3`. c. `main` calls `sub2`; `sub2` calls `sub3`; `sub3` calls `sub1`. d. `main` calls `sub3`; `sub3` calls `sub1`. e. `main` calls `sub1`; `sub1` calls `sub3`; `sub3` calls `sub2`. f. `main` calls `sub3`; `sub3` calls `sub2`; `sub2` calls `sub1`.

Question	Variables	Where Declared
a)		
b)		
c)		
d)		
e)		
f)		

3 Coding Together

`Python` is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.

Like function, definitions begin with the `def` keyword in Python, class definitions begin with a class keyword. The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended. Here is a simple class definition:

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also special attributes in it that begin with double underscores _~. For example, `__doc__` gives us the docstring of that class. Let's test it out. For this purpose, we need to start with defining a class. Let's consider we are trying to define a class that holds information of students.

```
class Student:
    '''
    Class for defining students in our system
    '''
    pass
```

This class should have attributes that show some part of student's identities as well as a method for us to access these data.

```
print(Student.__doc__)
```

Classes are also able to keep attributes. Let's add an attribute for age of student:

```
class Student:
    '''
    Class for defining students in our system
    '''

    # This is for keeping the age of the student
    age = 10

print(Student.age)
```

Or we can define a function inside a class that does some group of processes:

```
class Student:
    '''
    Class for defining students in our system
    '''

    # This is for keeping the age of the student
    age = 10
    def greet(self):
        print('Hello')
```

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
print(Student.greet)
```

3.1 Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a `function` call.

```
habib = Student()
```

This will create a new object instance named `habib`. We can access the attributes of objects using the object name prefix.

```
class Student:
    '''
    Class for defining students in our system
    '''

    # This is for keeping the age of the student
    age = 10

    def greet(self):
        print('Hello')

habib = Student()

print(Student.greet)

print(habib.greet)

print(habib.greet())
```

You may have noticed the `self` parameter in function definition inside the class but we called the method simply as `habib.greet()` without any arguments. It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, `habib.greet()` translates into `Student.greet(habib)`.

In general, calling a method with a list of `n` arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called `self`.

3.2 Constructors in Python

Class functions that begin with double underscore `__` are called special functions as they have special meanings.

Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called `constructors` in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class Student:
    '''
    Class for defining students in our system
    '''

    # This is for keeping the age of the student
    def __init__(self, first_name,last_name, mac_id, age=10):
        self.first_name = first_name
        self.last_name = last_name
        self.mac_id = mac_id
        self.age=10

    def get_data(self):
        print(f'You are looking at {self.first_name} {self.last_name} information with
```

In the above example, we defined a new class to represent a student. It has two functions, `__init__()` to initialize the variables and `get_data()` to display the student's data. Observer that age already has a default value and there is no need to pass any value for it.

Now let's look at another constructor. `__eq__` that can be used to compare two objects by their values. Python automatically calls the `__eq__` method of a class when you use the == operator to compare the instances of the class. By default, Python uses the `is` operator if you don't provide a specific implementation for the `__eq__` method. The following shows how to implement the `__eq__` method in the Student class

```
class Student:
    '''
    Class for defining students in our system
    '''

    # This is for keeping the age of the student
    def __init__(self, first_name,last_name, mac_id, age=10):
        self.first_name = first_name
        self.last_name = last_name
        self.mac_id = mac_id
        self.age=10

    def get_data(self):
        print(f'You are looking at {self.first_name} {self.last_name} information with

    def __eq__(self, other):
        '''
        Self addressing the current instance of the object while the other address the
        '''
        return self.age == other.age

habib = Student(self, 'Habib', 'Ghaffari', 'ghaffh1')
hadi = Student(self, 'Hadi', 'Daniali', 'danialih', 20)

if habib == hadi:
  print('These two persons are the same')
else:
 print('These two persons are different')
```

But how this function is working? Since Python does not provide left/right versions of its comparison operators, how does it decide which function to call? Let's make some changes in the function to figure out how it makes the comparison:

```
class A(object):
    def __eq__(self, other):
        print("A __eq__ called: %r == %r ?" % (self, other))
        return self.value == other
class B(object):
    def __eq__(self, other):
        print("B __eq__ called: %r == %r ?" % (self, other))
        return self.value == other

a = A()
a.value = 3
b = B()
b.value = 4
a == b
```

Explanation: The `a==b` expression invokes `A.__eq__`, since it exists. Its code includes `self.value == other`. Since int's don't know how to compare themselves to B's, Python tries invoking `B.__eq__` to see if it knows how to compare itself to an int. So it will print this:

```
A __eq__ called: <__main__.A object at 0x013BA070> == <__main__.B object at 0x013BA090
B __eq__ called: <__main__.B object at 0x013BA090> == 3 ?
```

Now we have clear vision of how the decision tree for `__eq__` constructor works.

4 Exercises

1. Try to define a subclass called `EngStudent`. This class going to address the engineering facility students and has an extra attribute called `faculity`.
2. Implement the `__init__()`, `__eq__`, and `__ne__` constructors for `EngStudent`.
3. Justify how decision tree for `__eq__` and `__ne__` going to work? What would happen if we do not implement `__eq__`?