PIC 16, Winter 2018 – Preparation 4F

Assigned 1/31/2018. To be completed by class 2/2/2018.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- determine (programmatically) whether a script is the main program or not using the global variable `__name__`;
- write subclasses that inherit variables and methods from their superclasses;
- create new-style classes and use the built-in `super` function to access methods of the superclass;

Tasks

- ☐ First, download the example scripts referred to in the following. Direct links are provided for each, but you may want to just open them Spyder to start.
- ☐ Before starting GUIs, we need to revisit some things we brushed over earlier. Read or review Executing Modules as Scripts.
- ☐ We'll see the Python idiom `if __name__ == "__main__"` a lot in GUI examples, so it's good to understand what's going on. Watch Main Function.
- ☐ Reading `main1.py` and predict what it will print to the console when run.
- ☐ Now run `main1.py` in Spyder or by calling `python main1.py` at the command line. Only `main1 is running` prints to the console, not `main() ran`. That's because the code to print `main() ran` is in a method called `main`, and there's nothing special about a method with the name `main` in Python (whereas it's quite special in C++ and Java). If we don't call the `main` method explicitly, it doesn't run.
- ☐ Now run `main2.py`. It prints `main1 is running` *and* `main() ran` because we called `main()` explicitly in the script.
- ☐ Next, run `main3.py`. Note that it's `__name__` variable has automatically been (created and) initialized to `__main__` because this is the file you ran from Spyder (or the command line), and thus it is the "main" module.
- ☐ Finally, run `main4.py` from the same directory as `main3.py`. Let's look at what is going on:
  - `import main3` causes `main3.py` to run as a script.
  - The messages printed to the console indicate that the `__name__` variable in the `main3` script is no longer `__main__` as in the last example because it is not the "main" module; it is not the one you chose to run from Spyder (or the command line). It was just imported, so instead its `__name__` is `main3`, the name of the file. Consequently, its `main` method does not run.
  - Now the code in `main4.py` is executed. You can see that its `__name__` is `__main__` because it is the "main" module, the one you chose to run.
- ☐ The only thing that's unusual here is how the `__name__` variable works. It takes on a different value depending on whether it appears in the script you chose to run or in an imported module. In the script you chose to run, `__name__` takes on the value `__main__`. In an imported module, `__name__` is the filename of the module. If this was not clear, please go back and/or do some tests of your own to understand its behavior.
- ☐ The purpose of the `if __name__ == "__main__"` idiom is to enable certain code to run (often test code) when the user chooses to run the file as a script but to prevent that same code from running if the file is imported as a module by another script.
- ☐ Read/reread Class Inheritance and Overloading Methods

- ☐ Try running `inheritance1.py`. Here we have a superclass `ClassA` and a subclass `ClassB`; we indicate that `ClassB` extends (subclasses) `ClassA` by including `ClassA` in parentheses after the name of `ClassB`. Both have initializers that print something when they run. `ClassA` has an class variable `static_var` (named because it's very similar to a static field in Java), and its initializer defines an instance variable `instance_var`. In the `main` function, we create a `ClassB` object and see that indeed, the class itself and an instance of that class have inherited the class variable `static_var` from `ClassA`. However, `b` does *not* have an `instance_var` because the superclass initializer *doesn't* run automatically when we create an instance of the subclass. **If we think of `__init__` as a "constructor", this would seem quite different from C++ and Java. On the other hand, if we recall that a different method `__new__` is really the "constructor", `__init__` is really just the *initializer*, and consider that `__new__` *does run* automatically before `__init__`, it makes a little more sense that we can access the superclass class variables (and methods, if they exist) even though the superclass `__init__` hasn't run.**
- ☐ Now run `inheritance2.py`. The difference here is that on line 12 we `try` to call the superclass' initializer explicitly (using the Python equivalent of `super` from Java). We're trying to explicitly initialize the superclass object within the instance of the subclass so we can inherit its instance variables. Calling `super(ClassB,self)` means something like "give me a reference to the instance of the superclass of `ClassB` that lives within `self`", or "give me a reference to the superclass part of the present object", then `.__init__()` calls the initializer of that superclass object. It doesn't work in this script… but with one simple change it will.
- ☐ Run `inheritance4.py`. The *only* difference between this and `inheritance2.py` is that `ClassA` subclasses `object`. This is all it takes to make the built-in `super` function work properly. Now `b` has the `instance_var` inherited from the `ClassA` object that "lives within" it. Why did having `ClassA` subclass `object` make this work? Apparently a class that doesn't explicitly subclass `object` is an "old-style" class and those that do are "new-style" classes. You're welcome to read about this online, but it doesn't really matter to us because we'll typically use "new-style" classes when we need to use inheritance. All you need to know is that you can refer to the "superclass object within `self`" or "superclass part of the present object" by calling `super(ClassB,self)`, provided the superclass explicitly subclasses `object`.
- ☐ Run `inheritance5.py`. Now we've added `my_method` to `ClassA`, and we can invoke it on an instance of `ClassB` because `ClassB` subclasses `ClassA`.
- ☐ Run `inheritance6.py`. Here we *override* `my_method` in `ClassB`. Now, invoking `my_method` on `b` causes the `ClassB` version of the method to run.
- ☐ Run `inheritance7.py`. Now we've added `my_super_method` to `ClassB` that exists only to call the superclass version of `my_method()`. Note that we've used `super(ClassB,self)` again to get a reference to the superclass object within, and invoke its `my_method`.
- ☐ Run `inheritance8.py`. Now we've called `my_method()` from within the initializer of `ClassA`. Perhaps surprisingly, the *subclass* version of the method is the one that actually gets run! That's the way things are supposed to work. Subclass methods completely override superclass methods unless we use something like `super(ClassB,self)` to explicitly refer to the superclass object and invoke the method on that. If we really, really want to call the `ClassA` version of `my_method`, we can call `super(ClassB,self).my_method()` or, perhaps more logically here, `ClassA.my_method(self)`, but this is atypical.