Additional Object-Oriented Techniques

Overview

In this lab, you'll enhance the `Employee` class from the previous lab...

First, you'll add various operator methods that will allow `Employee` objects to be compared with each other, based on their salary. Then you'll define a new class named `Programmer`, which will inherit from `Employee` and support some new and improved features.

Source folders

Student folder :	…`\Student\07-MoreOOP`
Solution folder:	…`\Solutions\07-MoreOOP`

Roadmap

There are 5 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Improving stringification
2. Implementing operator methods
3. Defining a `Programmer` class
4. Overriding superclass behavior
5. Additional suggestions (if time permits)

Exercise 1: Improving stringification

In the *Student* folder, open `company.py` in the text editor. This is the solution code from the previous lab, and it defines a simple `Employee` class. Take a moment to familiarize yourself with the code.

Note that the class currently has a `toString()` method, which returns a textual representation of the `Employee` object. This might look fine, especially if you're coming from a Java or C++ background, but it's not really the Python way to do it… In Python, the method should be named `__str__()` rather than `toString()`, so rename the method now. The implementation can stay the same.

Now open `clientcode.py`. The code creates a couple of `Employee` objects and prints them in string format. Note that the code currently calls `toString()` explicitly, to get the string representation for the `Employee` objects. You don't need to do this now – Python automatically calls `__str__()` whenever it needs to convert an object to a string, e.g. when you pass the object to `print()`. Therefore, replace the following statements:

```
print(emp1.toString())
print(emp2.toString())
```

…with the following simplified statements:

```
print(emp1)
print(emp2)
```

Run the code in `clientcode.py`. It should all still work nicely.

Exercise 2: Implementing operator methods

Go back to `company.py` and add methods to support the following relational operators for `Employee` objects (these methods should compare `Employee` objects based on their salary):

- `==`
- `!=`
- `<`
- `>`
- `<=`
- `>=`

Add code in `clientcode.py` to compare the two employees, to test all the above operators.

Exercise 3: Defining a `Programmer` class

Go back to `company.py` and define a new class named `Programmer`. A programmer is a kind of employee, so inheritance is appropriate here.

Suggestions and requirements for the `Programmer` class:

- A programmer is a kind of employee, as we just said.
- A programmer has all the basic characteristics of a regular employee.
- A programmer also has a collection of skills, i.e. the programming languages that he/she knows. Implement this as a dictionary, where the key is the name of a programming language, and the value is the programmer's current skill level in that language (1 to 5, where 1 means novice and 5 means guru). Initially, this skills dictionary should be empty. (You might want to take a look back in the "Data Structures" chapter to remind yourself about dictionaries in Python).
- A programmer can add a new language to his/her skillset. You must specify the name of the language and the level of proficiency.
- A programmer can improve his/her skill level in a particular language.
- You should be able to ask a programmer for his/her skill level in a particular language. It's OK for the programmer to say "none"!

Once you're happy with your `Programmer` class, go to `clientcode.py` and add some code to create a `Programmer` object and exercise its capabilities.

Exercise 4: Overriding superclass behavior

Go back to `company.py` and enhance the `Programmer` class so that it overrides the following methods:

- `payBonus()`
  For a programmer, the bonus calculation is different than for regular employees…

  A programmer gets the regular bonus payment as would any employee, but they also get an additional bonus of £100 for every programming language in their skillset. This is to encourage programmers to broaden their skillset (although it doesn't necessarily encourage them to *deepen* their skillsets!).

  Note that you'll probably need to access the salary attribute defined in the superclass, in order to add the bonus to the salary. This will present you with a problem, because the salary attribute is currently named `__salary` (which means its private to the base class). To overcome this problem, rename `__salary` to `_salary` everywhere in the base class. A single underscore is a Python convention that means "don't access this variable from the client code, but it's OK to access it from subclasses. It's a bit like "protected" in some other OO languages.

- `__str__()`
  For a programmer, you should output the number of languages in his/her skillset (as well as all the regular employee-related information, of course).

When you're done, go to `clientcode.py` and exercise the new bonus rules and string formatting capabilities.

Exercise 5: Additional suggestions (if time permits)

- In the `Programmer` class, enhance the `__str__()` method so that it displays all the details for the programmer's skillset, rather than just the number of languages.
- Explore the use of multiple inheritance in Python. Think of a scenario where multiple inheritance would be the correct approach, and then implement it.