1 Questions

Question 1. What is the difference between a class object and an instance object?

Question 2. What is the __init__ method used for?

Question 3. How are instances and classes created?

Question 4. What does self mean in a Python class?

Question 5. When might you want to support operator overloading in your classes?

2 Exercises

Exercise 1. Write a definition for a class named Kangaroo with the following methods:

- An __init__ method that initializes an attribute named pouch_contents to an empty list;
• A method named `put_in_pouch` that takes an object of any type and adds it to pouch_contents;

• A `__str__` method that returns a string representation of the Kangaroo object and the contents of the pouch;

Test your code by creating two Kangaroo objects, assigning them to variables named Kanga and roo, and then adding roo to the contents of Kanga's pouch.

**Exercise 2.** Simulate a fast-food ordering scenario by defining four classes:

• Lunch: a container and controller class;

• Customer: the actor who buys food;

• Employee: the actor from whom a customer orders;

• Food: What the customer buys.

To get you started, here are the classes and methods you'll be defining:

```python
class Lunch:
    def __init__(self) # Make/embed Customer and Employee
    def order(self, foodName) # Start a Customer order simulation
    def result(self) # Ask the Customer what Food it has

class Customer:
    def __init__(self) # Initialize my food to None
    def placeOrder(self, foodName, employee) # Place order with an Employee
    def printFood(self) # Print the name of my food

class Employee:
    def takeOrder(self, foodName) # Return a Food, with requested name

class Food:
    def __init__(self, name) # Store food name
```

The order simulation should work as follows:

1. The Lunch class's constructor should make and embed an instance of Customer and an instance of Employee, and it should export a method called order. When called, this order method should ask the Customer to place an order by calling its `placeOrder` method. The Customer's `placeOrder` method should in turn ask the Employee object for a new Food object by calling Employees `takeOrder` method;

   ```python
def order(self, foodName): # Start a Customer order simulation
    self.customer.placeOrder(foodName, self.employee)
```
2. **Food** objects should store a food name string (e.g., burritos), passed down from **Lunch.order**, to **Customer.placeOrder**, to **Employee.takeOrder**, and finally to **Food**s constructor. The top-level **Lunch** class should also export a method called result, which asks the customer to print the name of the food it received from the **Employee** via the order (this can be used to test your simulation).

Note that **Lunch** needs to pass either the **Employee** or itself to the **Customer** to allow the **Customer** to call **Employee** methods. Experiment with your classes interactively by importing the **Lunch** class, calling its order method to run an interaction, and then calling its result method to verify that the **Customer** got what he or she ordered. In this simulation, the **Customer** is the active agent; how would your classes change if **Employee** were the object that initiated customer/employee interaction instead?

**Exercise 3.** Write a class called **Mylist** that shadows (wraps) a Python list: it should overload most list operators and operations, including +, indexing, iteration, slicing, and list methods such as append and sort. Also, provide a constructor for your class that takes an existing list (or a **Mylist** instance) and copies its components into an instance member. Experiment with your class interactively. Things to explore:

- Why is copying the initial value important here?
- Can you use an empty slice (e.g., start[:]) to copy the initial value if its a **Mylist** instance?
- Is there a general way to route list method calls to the wrapped list?
- What type of object should operations like + and slicing return? What about indexing operations?
- If you are working with a more recent Python release (version 2.2 or later), you may implement this sort of wrapper class by embedding a real list in a stand-alone class, or by extending the built-in list type with a subclass. Which is easier, and why?

**Exercise 4.** Write a function called **oops** that explicitly raises an **IndexError** exception when called. Then write another function that calls **oops** inside a try/except statement to catch the error. What happens if you change oops to raise a **KeyError** instead of an **IndexError**? Where do the names **KeyError** and **IndexError** come from? (Hint: recall that all unqualified names come from one of four scopes.)
Excercise 5. Change the \texttt{oops} function you just wrote to raise an exception you define yourself, called \texttt{MyError}. Identify your exception with a class. Then, extend the try statement in the catcher function to catch this exception and its instance in addition to \texttt{IndexError}, and print the instance you catch.

Excercise 6. Write a function called \texttt{safe(func, \ast args)} that runs any function with any number of arguments by using the \texttt{\ast name} arbitrary arguments call syntax, catches any exception raised while the function runs, and prints the exception using the \texttt{exc.info} call in the \texttt{sys} module. Then use your safe function to run your \texttt{oops} function from exercise 1 or 2. Put safe in a module file called \texttt{tools.py}, and pass it the \texttt{oops} function interactively. What kind of error messages do you get? Finally, expand safe to also print a Python stack trace when an error occurs by calling the built-in \texttt{print.exc} function in the standard traceback module (see the Python library reference manual for details).