CSCI 204 – Introduction to Computer Science II

Lab 2 – Class Design, Inheritance, and Exceptions

1. Objectives

In this lab, you will learn the following:

- Designing and implementing classes;
- Using inheritance to abstract classes at different levels;
- Importing Python packages of your own;
- Using exceptions to handle errors.

2. Preparation

You will need to create and go into the directory for today's lab, assuming you have followed the instructions in lab 01 to create a `csci204` directory and stored your `lab01.txt` in that directory.

First start a terminal window. Then create the `labs` and `lab01` directory and move the `lab01.txt` into the lab01 directory.

```
cd ~/
cd csci204/
mkdir labs
mkdir lab01
mv lab01.txt lab01/
```

If you are not sure how to open a terminal window or what the above commands mean, please review what you did in last lab.

Now assuming you are in `~/csci204/labs` directory, create a directory for today's lab and go into that directory.

```
mkdir lab02
cd lab02
```

You will be creating some programs while using other existing programs for testing in this lab. Please copy the given testing files from the course Linux directory using the following Linux command. (Some of these programs are repeated in the lab description.)

```
cp ~csci204/2017-fall/student/labs/lab02/* .
```

Note that there is a dot at the end of the command, which means copying the files to the current place and keeping the file names. You should see a collection of six test programs.

3. Introduction to Class Inheritance

The **inheritance** feature in Python allows us to define objects at different levels of abstraction with common features. Doing so makes programs easier to use and maintain. Let's look at the example from

our textbook. All forms of *Publication* share some common features such as title and author. *Book* is a child-class of *Publication* which may contain information such as publisher and chapter titles in addition to the features in *Publication* while the *Article* child-class may contain journal title, journal volume, and such. The following illustration is from our textbook [ref textbook].

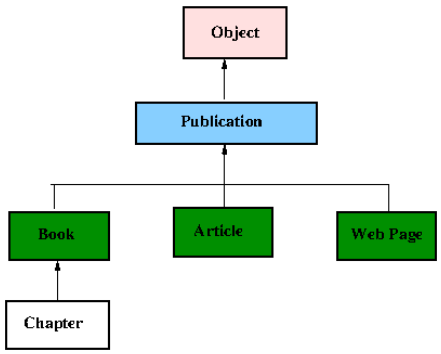

Figure 1: Class hierarchy of publications

In this design, the common features in the class *Publication* such as title and author do not have to be repeated in the inherited classes such as *Book* or *Article*, which simplifies class design a great deal. In this example, the code for the class *Publication* and *Book* may look as follows.

```
class Publication:
    """ Base class: Publication"""
    def __init__(self, title, author):
        self._title = title
        self._author = author

    def __str__(self):
        return '[' + self._title + '] by ' + self._author + '\n'

class Book(Publication):
    """ Derived class: Book"""
    def __init__(self, title, author, publisher, pub_date):
        super().__init__(title, author)   # Invoke parent constructor
        self._publisher = publisher       # Attributes in Book only
        self._pub_date  = pub_date

    def __str__(self):
        s = super().__str__()             # Invoke parent __str__()
        s += 'Publisher: ' + self._publisher + '\n'\
             'Date: ' + self._pub_date + '\n'
```

Figure 2: Partial implementation of Publication and Book class

In this example, the *Book* class **inherits** all features from the *Publication* class besides its own attributes of **publisher** and **pub_data**. The contents in the *Publication* class such as **author** and **title** need not to be repeated in the *Book* child-class, thus simplifying programming and maintenance.

The syntax of Python inheritance is very simple. The child-class, in this case, the *Book* class, needs to just include the name of parent class, *Publication* in its class name definition. In class methods, including the constructor, a child-class can invoke the parent class methods by having the keyword *super()* preceding the method name. If both the parent and child classes have a method with the same name, in our example, the constructor and the string method, typically you'd invoke the parent's method first, followed by additional, optional code in the child class.

Read the above code in Figure 2 carefully and make sure you understand how it works. **You can copy the code into your IDLE and try it out, if you are not 100 percent sure of the meaning of the program.** You will learn and practice the design and implementation of classes with inheritance in this lab.

4. Exercises with Inheritance: A Simple Pet Class

You are going to create a simple *Pet* class that will model a pet. Start thinking about pets. What types of attributes and behaviors do pets have? You might be able to come up with many. And, you may discover some attributes and behaviors that are highly dependent on type of the pet. Consider walking your pet. Clearly this is not a universal activity for all pets. Would you take your fish for a walk? On the other hand, some attributes ARE universal. For example, all pets would have a name, and an age. Designing a class hierarchy involves carefully picking common attributes for all objects of this type, e.g., all features common to pets, and deciding what attributes are specific for child-classes of objects, e.g., features that are only for fish, or dog.

Assume, for the sake of simplicity, that all pets that we are interested in have a **name** and an **age**. They can **eat**, **sleep**, and **walk**. Additionally, we'll use an extra attribute called **activity** to keep track of what our pet is currently doing, such as eating, sleeping, or walking. So a text description for a generic pet class would look as follows.

Table 1: Pet class design

Pet	
Attributes	name
	age
	activity
Methods	walk()
	eat()
	sleep()
	__init__()
	__str__()

4.1 Implement the Pet class

This seems like a reasonable start for a generic pet class. Go to IDLE, or using your favorite text editor to create this *Pet* class. Name your file `pet.py`. You should follow the guidance below.

- Create some class-wide constants to represent the current activity of the pet, walking, eating, and sleeping. Remember Python constants should use all upper-case letters with underscores as their names. For consistency, let's name these constants **WALKING** with a value of 1, **EATING**, 2,

and **SLEEPING**, 3. In addition, define a constant called **UNKNOWN** to have a value of zero so we can handle erroneous cases.

- Define a constructor with two parameters, a **name** and an **age**. Assign these parameters to proper class attributes. The **activity** attribute of the object should be initialized to **UNKNOWN**.
- Define a string representation of the object using the `__str__()` method which should print the name, age, and the activity of the pet.
- For each of the methods, `walk()`, `eat()`, and `sleep()`, all you need to do is to set the current activity to the appropriate value.

If you implemented your Pet class properly, run the following test program.

```
'''
A test program for class Pet
Converted from the Java programs developed by Brian King for the Java version
of CSCI 204.
Xiannong Meng
2017-07-26
'''
from pet import *

def test_pet( this_pet ):
    print(this_pet)
    pet_name = this_pet.name
    print('Taking ' + pet_name + ' for a walk')
    this_pet.walk()
    print(this_pet)
    print('Feeding ' + pet_name)
    this_pet.eat()
    print(this_pet)
    print('Sending ' + pet_name + ' to bed.')
    this_pet.sleep()
    print(this_pet)
def main():
    my_pets = [Pet('Garfield', 4),
               Pet('Sleepy', 8)]
    for pet in my_pets:
        print('--- begin ---')
        test_pet(pet)
        print('--- end ---')

main()
```

Figure 3: Test pets program (`test_pets.py`)

which should generate a result similar to the following.

```
--- begin ---
Garfield (age: 4) is doing UNKNOWN
Taking Garfield for a walk
Garfield (age: 4) is WALKING
Feeding Garfield
```

```
Garfield (age: 4) is EATING
Sending Garfield to bed.
Garfield (age: 4) is SLEEPING
--- end ---
--- begin ---
Sleepy (age: 8) is doing UNKNOWN
Taking Sleepy for a walk
Sleepy (age: 8) is WALKING
Feeding Sleepy
Sleepy (age: 8) is EATING
Sending Sleepy to bed.
Sleepy (age: 8) is SLEEPING
--- end ---
```

Figure 4: Result of running the program `test_pets.py`

Please read the test program and the output and make sure you understand what is going on. In particular, note that the `test_pet.py` program IMPORTS the *Pet* class by the line

```
from pet import *
```

This line means that we are importing everything from the file named `pet.py` that was created by your, just like we'd import any other Python packages.

4.2 Implement the Dog and Cat child-classes

Note that at this point, the objects in the *Pet* class have the same generic features and behaviors. You are now to implement child-classes *Dog* and *Cat*, which are derived from the parent class *Pet*.

Follow the example earlier in the lab description about the classes *Publication* and *Book* to implement the child-class *Dog* and *Cat*. Here are a few notes.

- Both the *Dog* and *Cat* classes need to implement the three methods, **eat()**, **walk()** and **sleep()** as we can imagine the behavior of these activities by a dog may differ that of a cat. In our implementation, we will simply call (a.k.a. invoke) the corresponding method from the super class *Pet*. The different behavior of a dog and a cat is simply represented by printing a different message in addition to calling the method in its parent class.
- Neither the *Dog* nor *Cat* class in our implementation has any extra attributes besides the **name** and the **age**. So the constructors of these two child-classes will simply call the constructor of its parent class.
- In our *Publication* and *Book* example, both the parent class and the child-class are implemented in the same file, say, `pub.py`. So when we use them in an application program
  ```
  from pub import *
  ```
  both the parent class and the child-class are available to the application program. In the *Pet* and its child-classes, you are asked to implement each class in a separate file, i.e., the *Pet* class in `pet.py`, the *Dog* class in `dog.py`, the *Cat* class in `cat.py`. This means in your `dog.py` and `cat.py` files, you have to have the line
  ```
  from pet import *
  ```
  in order to use the *Pet* class from the *Dog* class and the *Cat* class.

If you implement the *Dog* and *Cat* classes properly, run the following program, note the two import statements,

```
'''
A test program for classes Cat and Dog.
Converted from the Java programs developed by Brian King for the Java
version
of CSCI 204.
Xiannong Meng
2017-07-26
'''
from dog import *
from cat import *

def test_pet( this_pet ):
    print(this_pet)
    pet_name = this_pet.name
    print('Taking ' + pet_name + ' for a walk')
    this_pet.walk()
    print(this_pet)
    print('Feeding ' + pet_name)
    this_pet.eat()
    print(this_pet)
    print('Sending ' + pet_name + ' to bed.')
    this_pet.sleep()
    print(this_pet)

def main():
    my_pets = [Cat('Garfield', 4),
               Dog('Sleepy', 8)]
    for pet in my_pets:
        print('--- begin ---')
        test_pet(pet)
        print('--- end ---')

main()
```

Figure 5: Test program for the Dog and Cat classes (`test_dogs_cats.py`)

which should generate a result similar to the following.

```
--- begin ---
Garfield (age: 4) is doing UNKNOWN
Taking Garfield for a walk
Walk? Dude, seriously?
Garfield (age: 4) is WALKING
Feeding Garfield
Lasagna, please.
Garfield (age: 4) is EATING
Sending Garfield to bed.
Yes. I need 23 hours of this each day!
```

```
Garfield (age: 4) is SLEEPING
--- end ---
--- begin ---
Sleepy (age: 8) is doing UNKNOWN
Taking Sleepy for a walk
Walk?!?! Oh boy oh boy!!! Pant! Pant! Pant!
Sleepy (age: 8) is WALKING
Feeding Sleepy
Begging for food... kibbles and bits please.
Sleepy (age: 8) is EATING
Sending Sleepy to bed.
Zzzzzz (drooling)...
Sleepy (age: 8) is SLEEPING
--- end ---
```

Figure 6: Result of executing `test_dogs_cats.py`

Can you notice the differences between the program `test_dogs_cats.py` and `test_pets.py`? If you read carefully both programs, the differences are really very minimal, in `test_pets.py` the objects are defined as *Pet*, in `test_dogs_cats.py` the objects are defined as *Dog* or *Cat*, a child-class of Pet. The output of the test programs are very different because the behavior of a *Cat* is very different from that of a *Dog*. As a side note, Linux has a very useful command that can compare and tell the difference between two files. Try the following.

```
diff  test_dogs_cats.py test_pets.py
```

Observe the output. Convince yourself that you understand that is going on in using the `diff` command.

Make sure your programs `pet.py`, `cat.py`, and `dog.py` work properly, and save them before proceeding.

5. Exceptions

Another feature we will learn in this lab is **Exception**. We often encounter situations in which we know there are errors in data or user input. Python, like many other modern programming languages, allow the programmers to design programs to anticipate and handle these error conditions. The key mechanism of handling errors is the Python `try-except` structure. The idea is that you'd **try** some actions, if something is wrong, the errors should be caught by the **except** statement.

Python has a collection of pre-defined exceptions that the programmers can use. The following example illustrate the concept [ref:https://en.wikibooks.org/wiki/Python_Programming/Exceptions]

```
import random

class CustomValueError(ValueError):
  def __init__(self, message = 'Custom value error'):
    super().__init__(message)
    print('Exception : ' + message)

try:
  ri = random.randint(0, 3)
```

```
  print('randome value : ' + str(ri))
  if ri == 0:
    infinity = 1/0
    raise ZeroDivisionError
  elif ri == 1:
    raise ValueError("Message")
  elif ri == 2:
    raise ValueError # Without message
  elif ri == 3:
    raise CustomValueError
except ZeroDivisionError:
  print('Divided by zero')
except ValueError as valerr:
  print("Value error: " + str(valerr))
except CustomValueError:
  print('Custom value error')
except: # Any other exception
  print('Unknow error')
finally: # Optional
  pass # Clean up
```

Figure 7: The `test_exception.py` program that illustrate the concept of `try-except` clause

Copy and save this program as `test_exception.py`, then run the program multiple times, trying to observe the cases where all five different values are generated. The program generates a random number between 0 and 4, inclusive. For each value, the program invokes some form of exception. The number generation and the invoking exceptions are contained in a `try-except` clause. If something is worthy to cause exceptions (values 0 through 3), an exception is raised and caught in the proper `except` clause when the programming statements in the `except` will be executed.

In this example, two exceptions, `ZeroDivisionError` and `ValueError` are defined by Python. One exception, the `CustomValueError` class is defined by the programmer. Note the class definition of `CustomValueError` is very similar to any inherited class. It calls constructor of its parent class (`ValueError`) first, before adding any of its own action. The `ValueError` exception class actually is a child-class of a more general class `Exception`. See this document (https://docs.python.org/3/library/exceptions.html) for a complete description and hierarchy of Python exceptions.

In the rest of the lab, you are asked to implement a `Counter` class hierarchy using inheritance along with appropriate exception handling.

5.1 Implementing the Counter class hierarchy

Now that you have some experiences working with class design and implementation, we will give you some text description of what the classes should do, you will then design and implement the classes accordingly. So please read the description carefully and think about how you would proceed.

The *Counter* class hierarchy can be illustrated in the following diagram.

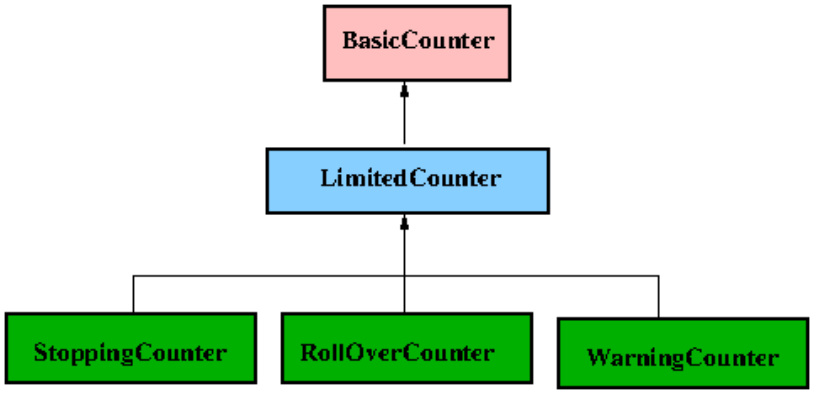

Figure 8: The Counter class hierarchy

The top-most class is the *BasicCounter* which contains two attributes, **counted** and **the_initial_count** along with five methods, `count()`, `un_count()`, `reset()`, `set_count_to()`, and `get_count_value()`. The meaning of the attributes and the behavior of the methods are as follows.

- The attribute **counted** keeps the current value of the counter which could be any integer, positive, negative, or zero;
- The attribute **the_initial_count** stores the initial value of the counter which can be used to reset the counter when needed;
- The `count()` and `un_count()` methods increment or decrement the counter value by one, respectively;
- The `reset()` method resets the counter value to **the_initial_count**;
- The `set_count_to()` and `get_count_value()` methods set and retrieve the counter value, respectively.

The *LimitedCounter* class inherits all features from the *BasicCounter* class. In addition, it has two class-wide constants, **LIMIT_MIN** and **LIMIT_MAX** which specify the upper and lower limit of the counter value. The idea is that this counter contains limits for the counter value. Accordingly, the *LimitedCounter* class has four method related to the limits, `is_at_min()` and `is_at_max()` which checks to see if the counter value is at its limits, as well as `get_min()` and `get_max()` which returns the limit value. Note that we are not allowed to change the limits once initialized. Also note that the *LimitedCounter* object does not take any actions if the value reaches limits. In another words, if the counter reaches its limits, for example, **LIMIT_MAX** and the counter value is incremented one more time, the operation will succeed.

The counters at next level, the *StoppingCounter*, the *RollOverCounter*, and the *WarningCounter* are all derived from *LimitedCounter*. The *StoppingCounter* will stop incrementing or decrementing the counter value once reaching the limits; the *RollOverCounter* will "roll over" the counter value after reaching the limits; the *WarningCounter* will raise exception and stop counting if the counter value reaches its limits.

Your task now is to **implement the first four counters** first, *BasicCounter*, *LimitedCounter*, *StoppingCounter*, and *RollOverCounter*. Unlike the first part of the lab where the parent class and child-classes are stored in separate files, we ask you to implement these counters in one file called `counter.py`. You will be asked to implement the *WarningCounter* a bit later.

Make sure you test often as you progress. Here is a sample program to test the *RollOverCounter* class.

```
from counter import *

print('Testing RollOverCounter...')
my_count = RollOverCounter(10, 12)

print('Its min value should be 10 ... It is ' + str(my_count.get_min()))
print('The count value should be at minimum ... ' +
str(my_count.is_at_min()))
print('The count value should not be at maximum ... ' +
str(my_count.is_at_max()))

my_count.count()
my_count.count()
print('Increment twice, the count value should be 12 now ... ' +
str(my_count.get_count_value()))

print('The count value should be at maximum ... ' +
str(my_count.is_at_max()))
print('The count value should not be at minimum ... ' +
str(my_count.is_at_min()))

my_count.count()
print('Increment one more time, the count value should rollover to 10 ... ' +
str(my_count.get_count_value()))

my_count.un_count()
print('Decrement three times, the cout value should rollover to 12 ... ' +
str(my_count.get_count_value()))
```

Figure 9: A sample test program for RollOverCounter (`test_rollover_counter.py`)

5.2 Design and implement CounterException and WarningCounter class

Following the description and the example at the beginning of Section 4 for Python exceptions, you are to implement a *CounterException* class and use it in the *WarningCounter* class. The idea is that if the counter value in a *WarningCounter* object has reached its limit (upper limit or lower limit), the counter value will not change, but raise a *CounterException*. The *CounterException* constructor will call the constructor of its parent class *Exception* first, then print its own error message. Note that in the random number example in Figure 7, the parent class of the *CustomValueError* is *ValueError*, you could use the *ValueError* as the parent class of *CounterException*, but you can also use *Exception* directly.

Implement both the *WarningCounter* and *CounterException* in the file `counter.py`.

After implementing the *WarningCounter* and *CounterException*, try the following program.

```
from counter import *

def readLimit(prompt):

    v = 0
    while True:
        try:
            v = int(input(prompt))
            if v < LimitedCounter.DEFAULT_MIN:
                raise ValueTooSmallError
            elif v > LimitedCounter.DEFAULT_MAX:
                raise ValueTooLargeError
            break
        except ValueError:
            print('Number error, try again!')
        except ValueTooSmallError:
            print("This value is too small, try again!")
            print()
        except ValueTooLargeError:
            print("This value is too large, try again!")
            print()
    return v

print('Testing WarningCounter...')
my_count = WarningCounter(10, 12)

print('Its min value should be 10 ... It is ' + str(my_count.get_min()))
print('The count value should be at minimum ... ' +
str(my_count.is_at_min()))
print('The count value should not be at maximum ... ' +
str(my_count.is_at_max()))

my_count.count()
my_count.count()
print('Increment twice, the count value should be 12 now ... ' +
str(my_count.get_count_value()))

print('The count value should be at maximum ... ' +
str(my_count.is_at_max()))
print('The count value should not be at minimum ... ' +
str(my_count.is_at_min()))

'''
print('Increment one more time, result in exception ... ')
my_count.count()
'''

my_count.un_count()
my_count.un_count()
```

```
print('The count value should be at minimum ... ' +
str(my_count.is_at_min()))
print('The count value should not be at maximum ... ' +
str(my_count.is_at_max()))

'''
print('Decrement one more time, result in exception ... ')
my_count.un_count()
'''

lo = LimitedCounter.DEFAULT_MIN
hi = LimitedCounter.DEFAULT_MAX
while True:
    try:
        lo = readLimit('Enter minimum: ')
        hi = readLimit('Enter maximum: ')
        if lo >= hi:
            raise ValueIncorrectError
        break
    except ValueIncorrectError:
        print('The values of ' + str(lo) + ' and ' + str(hi) + ' are wrong.')

print('lo ' + str(lo) + ' hi ' + str(hi))
```

Figure 10: Program to test WarningCounter (`test_warning_counter.py`)

6. Prepare for submission

Review your programs and make sure your programs follow proper conventions including naming and comments. Remove extra printing statements you may have put in place during the development of the program. Format the programs properly.

Make sure at the top of the each program you include a global comment section following the sample below that indicate the lab assignment, your name, your lab section, and your professor's name.

```
“””CSCI 204 Lab 02 Class Design, Implementation, and Exceptions
Lab section: CSCI 204.L61, Tuesday 10-11:50
Student name: Sam Snoopy
Instructor name: Professor Garfield”””
```

7. Submission

Submit the following files individually to the course Moodle site. Yes, please submit the files we give you as well to make grading a bit easier.

```
cat.py, dog.py, pet.py, test_dogs_cats.py, test_pets.py,

counter.py, test_rollover_counter.py, test_warning_counter.py,
test_limited_counter.py, test_stop_counter.py.
```