CMSC 143: Introduction to Object-Oriented Programming with Robots

# Lab 7: Defensive Programming
# Due October 26, 2009

Submit a copy of your python program (cmsc143_lab7_LASTNAME_FIRSTNAME.py) on moodle. Your program should have your name, email, and the date at the top of the file as a comment.

This lab is about defensive programming – making your programs robust. Up to now, we've assumed the users of our programs have been relatively well-behaved. When users gave us input, the input could be converted to the right type; when our functions were called, we were passed the right types – but that isn't always true.

```
def do_division():
        x = int(raw_input("Enter the numerator: "))
        y = int(raw_input("Enter the denominator: "))
        q = x / y
        speak(str(x) + " divided by " + str(y) + " is " + str(q))
```

Consider the above program. What happens if the user enters cat as the numerator, or 3.5 as the numerator, or 0 as the denominator? Errors happen, that's what! To be accurate – an exception is thrown.

## Exceptions

When Python detects a run-time error, something called an Not only do exceptions notify us of the problem, we can **catch** exceptions, and handle the errors appropriately. Exceptions, like loops, conditionals, and function calls, modify the flow of execution of our programs. There are different types of exceptions[1] allowing us to handle them differently. `ZeroDivisionError` is one type of exception that is thrown when a division by zero happens, `ValueError` is thrown when you use the wrong type of value, `IndexError` is raised when your sequence index is out of range. The program below improves upon `do_division()` to handle the different error modes. Specifically, it prints out an informative error message if the user enters zero for the denominator, or a non-integer type for either value, then re-calls itself.

```
def do_defensive_division():
    try:
        x = int(raw_input("Enter the numerator"))
        y = int(raw_input("Enter the denominator"))
        q = x / y
        speak(str(x) + " divided by " + str(y) + " is " + str(q))
    except ZeroDivisionError:
        print "Can not divide by zero"
        do_defensive_division()
    except ValueError:
        print "You must enter a integer value"
        do_defensive_division()
```

[1] http://python.org/doc/2.4.1/lib/module-exceptions.html

## Pre-conditions and Post-Conditions

When we write functions, we have a clear, although often implicit idea of what the function does, and what it expects from its parameters or global variables. Pre- and post- conditions make these implicit assumptions explicit. We encode these assumptions as expressions we expect to be true about the world before the function starts executing (pre-conditions) and things that will be true about the world after the function is finished (post-conditions). Preconditions allow us to safeguard ourselves against bogus parameters, and postconditions allow us to sanity check our calculations

Not only can we catch exceptions to handle error conditions, we can raise (or throw) exceptions when we have detected an error condition. We can use if statements to test that our pre- and post-conditions are true, and raise exceptions if they are false.

For instance, we could write pre- and post-conditions for an algorithm for computing square roots. The number we are passed should be non-negative and a number. The number we compute and return, when multiplied by itself, should be close to the original parameter.

```
def sqrt(num):

    #Pre-conditions: num must be a non-zero float or integer
    if type(num) != int or type(num) != float:
         raise ValueError, "num must be a number"

    if num < 0:
         raise ValueError, "num must be non-negative"

    x = num

    while True:
        y = (x + num/x) / 2.0
        if abs(y-x) < 0.00001:
            break
        x = y

    #Post-condition: sqrt * sqrt should be close to our original num
    if abs(y**2 - num) > 0.0001:
         raise RuntimeError, "sqrt is not working correctly"

    return y
```

## Unit Testing

Similar to post-conditions, another useful tool in defensive programming is writing a complete suite of tests. Even before writing your functions or program, it is often convenient to write some test cases for how the function or program **should** behave. Testing each function you write is called unit testing. At the heart of a unit testing is finding the right test conditions. You want to test your function on common values, but more importantly, on **corner cases**, those values where the function is about to stop working, or perhaps shouldn't work at all. There are two techniques for designing your test suite. Using **black-box testing** we are only concerned with what the function **should** do, and design our test-cases around that. Another technique is to look at the code directly and try to exercise that particular implementation, this is known as **white-box testing**.

For example, we could write the following unit-test for our `sqrt()` function to give us confidence it is working correctly.

```
def close(v1, v2):
    if abs(v1 - v2) < 0.01:
        return True
    else:
        return False

def test_sqrt():
    # test common cases (squares)
    for i in range(1, 10):
        if close(sqrt(i*i), i):
            print "unit test passed for", i*i
        else:
            print "unit test FAILED for", i*i

    # test some corner cases
    # non square numbers
    if close(sqrt(5)**2, 5):
        print "unit test passed for", 5
    else:
        print "unit test failed for", 5


    # negative numbers should raise exception
    try:
        v = sqrt(-1)
        print "unit test failed for", -1
    except:
        print "unit test passed for", -1

if __name__ == "__main__":
    test_sqrt()
```

## Learning Objectives

- Program Defensively.
- Handle exceptions.
- Use Pre- and Post-conditions

## Deliverables

Below are a few different programs we have worked with throughout the semester. They are a bit fragile. Our task in this lab is to make them more robust by using unit-tests and defensive programming.

**Part One:** Write unit tests for each function. Break each function below by using strange input (user input or parameter values). Note what you did and the exception that occurred.

**Part Two:** Modify the functions to be more robust. Add pre- and post-conditions to functions that take parameters. When interacting with the user catch any exceptions and validate the input then act appropriately.

**Part Three:** Exchange unit tests with a classmate. Did your unit-test find any bugs in their program? Did their unit-test find any bugs in your program?

```
def factorial(x):
  if x == 0:
       return 1
  else:
       return x * factorial(x-1)

def pigLatin(word):
   return word[1:] + word[0] + "ay"

def translate():
   word = raw_input("Enter your word:")
   speak(pigLatin(word))

def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]

def isPalindrome(s):
    if len(s) < 2:
        return True
    else:
        return first(s) == last(s) and isPalindrome(middle(s))

def divide(dividend, divisor):
    # division can be seen as just repeated subtraction. Keep dividing the dividend
    # until there is only a remainder less than the divisor left.
    if divisor <= 0 or dividend < 0 or not type(int):
        raise ValueError
    quotient, remainder = 0, dividend
    # FILL THIS IN
    if dividend != (divisor * quotient + remainder):
        raise RuntimeError, "divide is not working correctly"
    return (quotient, remainder)
```