CSCE 1035 Lab 10 Fall 2021

1. zyBooks Labs

Please follow the link on Canvas to complete the following zyBooks labs:

- 10.7 *LAB: Fat-burning heart rate*
- 10.8 *LAB: Exception handling to detect input string vs. integer*
- 11.9 *LAB: Guess the random number*
- 11.10 *LAB: Quadratic formula*

2. Exception Handling in Python

Ideally, when we write a computer program for a mathematical task, we plan in advance for every possibility and design an algorithm to handle all of them. For example, anticipating the possibility of division by zero and avoiding it is a common issue in making a program robust.

However, this is not always feasible. In particular, while still developing a program, there might be situations that you have not yet anticipated, so it can be useful to write a program that will detect problems that occur while the program is running, and handle them in a reasonable way.

Consider the following Python 3 code for solving quadratic equations:

```
from math import sqrt

print('Let\'s solve a quadratic equation a*x**2 + b*x + c = 0')
a = float(input('a = '))
b = float(input('b = '))
c = float(input('c = '))

discriminant = b**2 - 4*a*c
root_of_discriminant = sqrt(discriminant)
roots = ((-b - root_of_discriminant)/(2*a), (-b + root_of_discriminant)/(2*a))
print('The roots are %g and %g' % roots)
```

Type up this program in as **`Lab10A.py`** and run it to verify that it works for cases where the solution has roots. For example, run the program to show that the equation $x^2 + x - 12$ has roots at $x = 3$ and $x = -4$. Unfortunately, the equation $x^2 - 4x + 4$ has no real roots and results in a `ValueError` exception with the `'math domain error'` error message when this program is run using this input.

We can try to catch any exceptional situation and handle it by apologizing for failing to solve the equation using a `try-except` block as follows:

```
try:
   a = float(input('a = '))
   …
   print('The roots are %g and %g' % roots)
except:
   print('Something went wrong - sorry!')
```

This `try-except` block does two things:

- It first tries to run the code in the indented block introduced by the colon after the `try` statement.
- If anything goes wrong (i.e., if any exception occurs), it gives up on that `try` block runs the code in the block under the `except` statement.

Add the code in red to your existing `Lab10A.py` file, being sure to indent the original code inside the `try` block, and verify that it now catches the error introduced with the $x^2 - 4x + 4$ equation. Although we have now "handled" the error, we lost some detail about why the exception occurred. We can regain this information by having the `except` statement save the error message into a variable as follows:

```
try:
   a = float(input('a = '))
   …
   print('The roots are %g and %g' % roots)
except Exception as errmsg:
   print('The exception message is:', errmsg)
```

Make the changes from the code in red to your existing `Lab10A.py` file and verify that it now provides the `'math domain error'` error message.

There are several ways to "handle" exceptions. If the error is critical such that the program can no longer run, we can exit the program gracefully. However, another option would be to tell the user what went wrong and try again. Let's modify the code again to give the user another chance to solve the quadratic equation again as follows:

```
done = False
while not done:
   try:
      a = float(input('a = '))
      …
      print('The roots are %g and %g' % roots)
      done = True # so we can exit the loop
   except Exception as errmsg:
      print('The exception message is:', errmsg)
      print('Please try again')
print('Hope you enjoyed this program')
```

Make the changes from the code in red to your existing `Lab10A.py` file, being sure to indent the original code inside the `while` block, and verify that it now allows the user to enter new coefficients for the quadratic equation.

Now, let's look at handling multiple specific exception types:

- `ZeroDivisionError: float division by zero`
- `ValueError: math domain error`
- `ValueError: could not convert string to float: 'one'`

To handle the division by zero error (i.e., when our value for `a` is 0), add the following above the `except Exception as errmsg:` line to your `Lab10A.py` file:

```
except ZeroDivisionError as errmsg:
    print('ZeroDivisionError:', errmsg)
    print('The first coefficient cannot be zero!')
```

It is important that you keep the `except Exception as errmsg:` line after any other except blocks since this handles the generic case and is a catch-all after a list of exception types have been handled.

Finally, go ahead and handle the `ValueError` exception in your `Lab10A.py` file that can be caused by two different scenarios. Fortunately, you only need to add one `except` block in this case since it is one type of exception and we can still obtain the specific error message for whatever scenario caused the exception. You do not need an extra `print()` message like what was done for the zero division error.

Note that you will submit this file to Canvas.

3. Custom Exceptions in Python

You can also catch custom exceptions that you define yourself. Consider the following code:

```
import random

def random_errors(p_error):
    if random.random() <= p_error:
        raise RandomError('Error raised with {:0.2f} likelihood'.format(p_error))

random_errors(0.5)
print('No errors occurred!')
```

The `random()` function returns floating-point numbers in the range [0.0, 1.0), so this means that roughly half of the time, the random number will be less than 0.5, thus raising a `RandomError` exception.

Type up this program in as **`Lab10B.py`** and run it several times to verify that it results in a `SyntaxError` some of the time since the `RandomError` exception is not defined and other times it prints the statement `'No errors occurred!'`.

Now, let's go ahead and define a custom exception for the `RandomError` exception and implement a `try-except-finally` block to handle the exception.

A custom exception for the `RandomError` exception can be done by adding a `class` definition at the top of the file, but below the `import` statement as follows:

```
class RandomError(Exception):
    """A custom exception for this code block"""
    pass
```

Note that the text inside the `"""` is a docstring for the `RandomError` class. `pass` is just an empty statement.

Next, add the `try` block, indenting the function call and `print()` statement inside the block. Add the `except` block for `RandomError`, printing the error message generated from the exception.

Finally, add a `finally` block at the end (after the `except` block) with the print statement `'This runs no matter what'`.

Note that you will submit this file to Canvas.

4. Working with Modules in Python

A Python "module" is a single `.py` file that contains function definitions and variable assignment statements. Importing a module will execute these statements, rendering the resulting objects available via the imported module.

Create a Python 3 module called **`my_module.py`** with the following:

- Define a function called `square` that accepts one argument, a number. Inside the function, return the square of that number.
- Define a function called `cube` that accepts one argument, a number. Inside the function, return the cube of that number.
- Add a `print` statement outside of any function that prints the sentence `"My name is <insert your name>!"`, where you would use your first and last name in place of `<insert your name>`.
- Define a list called `some_list` that contains the values `3` and `8`.

Now, run `my_module.py` and verify that it prints out the statement with your name.

Suppose that we are creating another module and want to use the `square` and `cube` functions to compute the area and volume of a square, respectively, as well as values from the `some_list` variable. But you don’t want it to print out your name!

In the same directory as `my_module.py`, create a new Python 3 module called **`Lab10C.py`** that does the following:

- Add an `import` statement to import the `my_module.py` module.
- Suppose that the `some_list` variable in `my_module.py` contains the length of the sides of a square and a cube (i.e., the shape object, not the functions) that you are interested in. In your program, pass these values to the `square` and `cube` functions to compute the area and volume, respectively,

of the two different objects. Be sure to prefix the functions and the list with `my_module` to make sure it works.

Now, run your code to verify that it produces something like this:

```
$ python3 Lab10C.py
My name is Mark Thompson
Area  : 9
Volume: 512
```

Unfortunately, we don't want it to print out your name, every time you run your `Lab10C.py` program, so we will need to modify the `my_module.py` program. To treat `my_module.py` as an importable module, not an executable script, add an `if` statement that only executes the `print` statement if the `__name__` attribute is equal to `'__main__'`.

Before we run the code again, I am sure that it was tedious to add the `my_module` prefix before both the functions and the list, so let's change the `import` statement to a `from` … `import` … statement in `Lab10C.py` to use the wildcard `*` that imports all names from the module. Then, remove the `my_module` prefix from the function calls and list.

Now, run your code again to verify that it produces the expected results (and doesn't print your name).

Note that you will submit both of these files to Canvas.

Now that you have completed this lab, it's time to turn in your results. Once you've moved the files to your windows machine (using **WinSCP**), you may use the browser to submit them to Canvas for the **Lab 10** dropbox.

You should submit the following files:

- **`Lab10A.py`**
- **`Lab10B.py`**
- **`my_module.py`**
- **`Lab10C.py`**
- **(Note that the zyBooks labs are submitted separately through Canvas.)**

Ask your TA to check your results before submission.

Now that you've finished the lab, use any additional time to practice writing simple programs out of the textbook, lectures, or even ones you come up with on your own to gain some more experience.