Math 246 Unit 7: Exception handling in Python

Brenton LeMesurier, October 22-24, 2015

Updated after the lab with some notes on handling the message from an exception

Aside. How to use these notebooks

Either run this code example below in the notebook, or copy them into to IPython comman window (within Spyder or stand-alone.)

Introduction

Ideally, when write a computer program for a mathematical task, we will 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, and 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.

We start by considering a very basic code for our favorite example, solving quadratic equations.

But first, a reminder of the title-author-date info. that should go at the top of every file!

In [ ]:

```
'''Handling exceptions when solving (quadratic) equations.
Author: Brenton LeMesurier <lemesurierb@cofc.edu>
Date: October 22, 2015
'''
```

We start with "version 0" which works fine for a case like `(a, b, c) = (2, -10, 8)`, but can fail in many ways if the "wrong" input is given.

Try it repeatedly, with some "destructive testing": seek input choices that will cause various problems.

In [ ]:

```
'''Version 0
Note: we must use the basic Python module math here;
the sqrt function from the module numpy or pylab behaves differently.
'''

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)
```

Let me know what problems you found; we will work as a class to detect and handle all of them.

Some messages I get ended with these lines, whose meaning we will soon explore:

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

Catching any "exceptional" situation and handling it specially

Here is a minimal way to catch all problems, and at least apologize for failing to solve the equation:

In [ ]:

```
'''Version 1: detect any exception, and apologise.'''

from math import sqrt

print("Let's solve a quadratic equation a*x**2 + b*x + c = 0")
try:
    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)
except:
    print("Something went wrong; sorry!")
```

This try/except structure does two things:

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

Catching and displaying the exception error message

One thing has been lost though: the messages like "float division by zero" as seen above, which say what sort of exception occured.

We can regain some of that, by having `except` statement save that message into a variable:

In [ ]:

```
'''Version 2: detect any exception, display its message, and apologise.'''

from math import sqrt

print ("Let's solve a quadratic equation a*x**2 + b*x + c = 0.")
try:
    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)
except Exception as what_just_happened:
    print("Something went wrong; sorry!")
    print("The exception message is: ", what_just_happened)
```

Another improvement would be to tell the user what went wrong, and give an opportunity to try again.

In fact I recommend that almost any program with interactive input, especially one still being developed, use a "try, try again" loop like this:

In [ ]:

```
'''Version 3: detect any exception, display its message, and repeat until we suc
ceed.'''

from math import sqrt

print("Let's solve a quadratic equation a*x**2 + b*x + c = 0.")
are_we_done_yet = False
while not are_we_done_yet:
    try:
        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("Success!")
        print("The roots are %g and %g" % roots)
        are_we_done_yet = True  # so we can exit the while loop
    except Exception as what_just_happened:
        print("Something went wrong; sorry!")
        print("The error message is: ", what_just_happened)
        print("Please try again.")
print('\nThank you; please come again soon!')
```

This version detects every possible exception and handles them all in the same way, whether it be a problem in arithmetic (like the dreaded division by zero) or the user making a typing error in the input of the coefficients. Try answering "one" when asked for a coefficient!

Handling specific exception types

Python divides exceptions into many types, and the statement `except` can be given the name of an exception type, so that it then handles only that type of exception.

For example, in the case of division be zero, where we originally got a message

```
ZeroDivisionError: float division by zero
```

we can catch that particular exception and handle it specially:

In [ ]:

```
'''Version 4: As above, but with special handing for division by zero.'''
from math import sqrt

print("Let's solve a quadratic equation a*x**2 + b*x + c = 0.")
are_we_done_yet = False
while not are_we_done_yet:
    try:
        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("Success!")
        print("The roots are %g and %g" % roots)
        are_we_done_yet = True  # so we can exit the while loop
    except ZeroDivisionError as message:  # Note the "CamelCase": capitalization
of each word
        print("Division by zero; the first coefficient cannot be zero!")
        print("Please try again.")
print('\nThank you; please come again soon!')
```

Handling multiple exception types

However, this still crashes with other errors, lke typos in the input. To detect several types of exception, and handle each in an appropriate way, there can be a list of `except` statements, each with a block of code to run when that exception is detected. The type `Exception` was already seen above; it is just the totally generic case, so can be used as a catch-all after a list of exception types have been handled.

In [ ]:

```
'''Version 5: As above for division by zero, but also catching other exceptions.
'''

from math import sqrt

print("Let's solve a quadratic equation a*x**2 + b*x + c = 0.")
are_we_done_yet = False
while not are_we_done_yet:
    try:
        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("Success!")
        print("The roots are %g and %g" % roots)
        are_we_done_yet = True  # so we can exit the while loop
    except ZeroDivisionError as message:
        print("ZeroDivisionError, message: ", message)
        print("The first coefficient cannot be zero!")
        print("Please try again.")
    except Exception as message:  # Any other exceptions
        print("Some exception that I have not planned for, with message:", messa
ge)
        print("Please try again.")
print('\nThank you; please come again soon!')
```

Experiment a bit, and you will see how these multiple `except` statements are used:

- the first `except` clause that matches is used, and any later ones are ignored;
- only if none matches does the code go back to simply "crashing", as with version 0 above.

Summary: a while-try-except pattern for interactive programs

For programs with interactive input, a useful pattern for robustly handling errors or surprises in the input is a while-try-except pattern, with a form like:

```
try_again = True
while try_again:
    try:
        Get input
        Do stuff with it
        try_again = False
    except Exception as message:
        print("Exception", message, " occurred; please try again.")
        Maybe actually fix the problem, and if successful:
            try_again = False
```

One can refine this by adding `except` clauses for as many specific exception types as are relevant, with more specific handling for each.

Exercise 1. Add handling for multiple types of exception

Copy my latest version here into a file named `Unit7_quadratic_solver.py` and augment it with multiple `except` clauses to handle all exceptions that we can get to occur.

First, read about the possibilities, for example in Section 5 of the official Python 3 Standard Library Reference Manual (https://docs.python.org/3/library/)

at https://docs.python.org/3/library/exceptions.html (https://docs.python.org/3/library/exceptions.html) or other sources that you can find.

Two exceptions of particular importance for us are `ValueError` and `ArithmeticError`, and sub-types of the latter like `ZeroDivisionError`. (Note the "CamelCase"; capitalization of each word in an exception name: it is essential to get this right, since Python is case-sensitive.)

Aside: If you find a source on Python exceptions that you prefer to that manual, please let us all know!

Update: converting that message to a string of text, and examing parts of it

Two topics came up in the lab today.

Firstly, it might be useful to have that `message` got from an exception as a string: that is, a list of characters, or "a piece of text". You can convert the exception's message to a string by using the function `str` as follows:

```
message = str(message)
```

Secondly, you might wish to examine just the initial part of this message string. One way to do this is to note that you can get the first n characters of a text string with indexing, just as for lists, arrays, and so on:

```
message_starts_with = message[:n]
```

However, there is another nice Python function for checking the start of a text string; the Python *method* `startswith()`. Here is an example of its usage:

In [2]:

```
message = 'Division by zero'
print(message[:8])
print(message.startswith('Division'))
print(message.startswith('division'))
```

```
Division
True
False
```

Note that this test is case sensitive, as always with Python.

Note also how a "method" is used: roughly, methods are a special type of function that act on a quantity by appending with a period, plus possible additional arguments in parentheses afterword.

Ther are many other method for working with strings described at https://docs.python.org/3.4/library/stdtypes.html#string-methods (https://docs.python.org/3.4/library/stdtypes.html#string-methods) and other methods too, as we will see later. Here is one example:

In [3]:

```
print('The original message:\n')
print(message)
print('\nConvert a string to all lower-case with the method lower():\n')
print(message.lower())
print("\nThis comparison is false, due to the upper-case 'D':\n")
print(message.startswith('division'))
print('\nBut we can compose methods, joined with periods from left to right.')
print('Here we first convert the message string to lower case,')
print('and then check what it starts with:\n')
print(message.lower().startswith('division'))
```

```
The original message:

Division by zero

Convert a string to all lower-case with the method lower():

division by zero

This comparison is false, due to the upper-case 'D':

False

But we can compose methods, joined with periods from left to right.
Here we first convert the message string to lower case,
and then check what it starts with:

True
```

Exercise 2. Handling division by zero Newton's method

Newton's method can be implemented with a function as follows:

In [ ]:

```
def newton(f, Df, x0, error_tolerance, maximum_steps):
    x = x0
    for step in range(maximum_steps):
        x_new = x - f(x)/Df(x)
        error_estimate = abs(x_new - x)
        x = x_new
        if error_estimate <= error_tolerance:
            # We are finshed early
            return x, error_estimate

# Demonstration
from math import cos, sin
def f(x): return x - cos(x)
def Df(x): return 1 + sin(x)

root, estimated_error = newton(f, Df, x0=0, error_tolerance=1e-10, maximum_steps
=100)
print('The solution is %.16g with estimated error of %.2g' % (root, estimated_er
ror))
```

However, division by zero can occur with some choices of the function `f` and the inital value `x0`, due to `Df(x) == 0` arising at some stage. So create a file `newton.py` based on the above, adding appropriate handling of this possibility. This can go beyond simply printing a message to actually trying to fix the problem and continue.