ENGG1811 Computing for Engineers

Week 4B:

Errors; Program testing and debugging; Exception handling

- Errors, program testing and debugging
- Exception handling

Introduction

- By now, you have been writing programs for a number of weeks, you probably have experienced:
  - Getting programs to run ☺
  - Getting error messages ☹
  - Getting the code to run but it doesn’t do what you intended it to do ☹

Error types

- There are two types of errors that stop your program from running:
  - Syntax error
  - Runtime error

- Language analogy
  - A syntax error is analogous to a grammatically incorrect sentence, e.g., sentences with spelling and/or punctuation error
  - A runtime error is analogous to a grammatically correct instruction that cannot be carried out. For example,
    - Fly to the centre of the sun and come back

Syntax errors

- Syntax errors violate the rules of how Python statements are written

- Some examples:
  - Misspelt keywords
  - Forget to use colon with if/else/elif/for/while/def
  - Wrong usage of or missing (), [], {}
  - Improper indentation

- The Spyder editor catches many of these errors

Quiz

- Can you tell what the syntax errors are?

```
Print(Good day mate!)
```

```
i_am_a_list = [2 3 5]
```

Runtime errors

- Terminology: Runtime is the time from the beginning of executing a program till the program terminates
- Runtime errors mean the computer is unable to execute the instructions
- Examples:
  - Forgot to import libraries
  - Used the wrong data types
  - Forgot to initialise a variable before using it
- Quiz: What are the errors in the following code?

```
b = 21
b = a + b
```

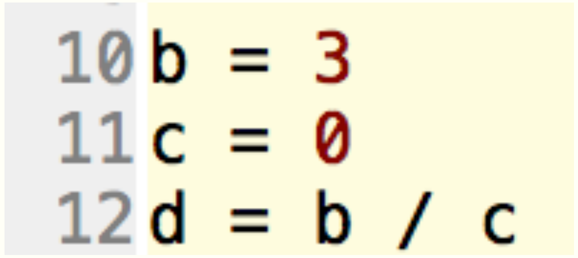
```
c = [4, 10, 17]
d = c[2] + c[3]
```

More runtime errors

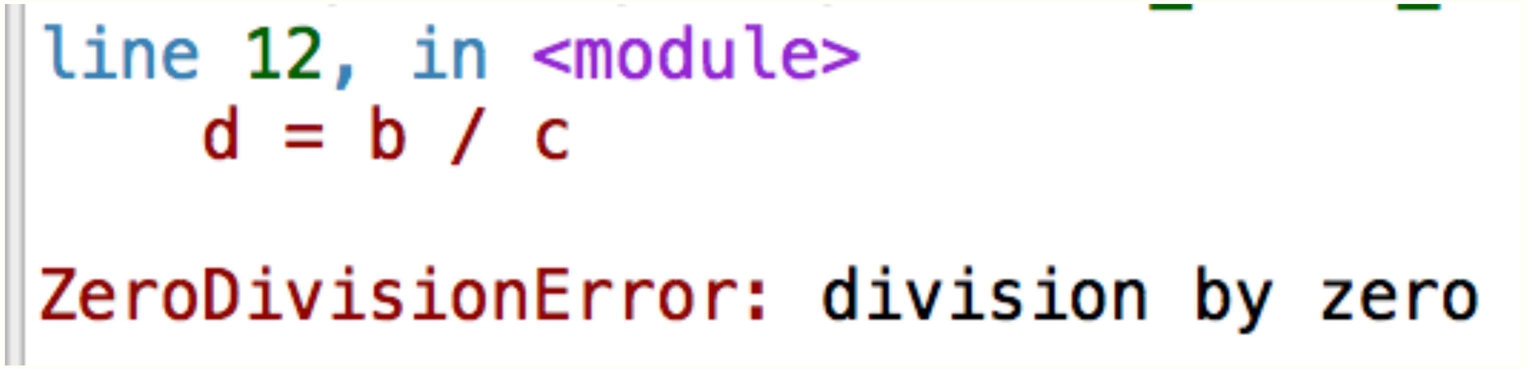
- You can also get run-time errors from doing operations that are not permitted
- Let us look at an example in runtime_error_ex.py

Runtime error

- When you see an error message, don’t panic
- There are two important pieces of information
  - Where the error occurs
  - What the error is



```
10 b = 3
11 c = 0
12 d = b / c
```



```
line 12, in <module>
    d = b / c

ZeroDivisionError: division by zero
```

Now your program runs ..

- A program that runs **doesn't** mean that your program is correct
- The instructions you give to the computer may not solve the problem you intend it to solve
- A real-life analogy: The room is really hot and you want to cool the room down. You issue the instruction:

  "Turn the heater on"

  - The instruction is grammatically correct = No syntax errors
  - The instruction can be executed = No runtime errors
  - But the instruction does not solve your problem

Program testing

- This is to test whether your program is doing what you intend the program to do
- We will first discuss a number of concepts
  - Unit testing
  - Black-box testing
  - Glass-box testing

Unit testing

- This refers to testing of the various components of a piece of software
- You may have written a program with a number of different functions
- You want to test all these functions to ensure that each function works properly

- Recall that we talk about incremental development
  - You should develop, test, develop, test

An example testing procedure

- Let us assume that you have developed a function to compute the maximum value in a Python list of numbers
- You can come out with a number of test cases that you know the expected answers

Test cases	Expected output
[2, 5, 8]	8
[3, 17, 19, 24]	24
[23, 1, 51, 19, 107, 123]	123

- You can write a testing program

```
For each test case
        Does the function output match the expected output?
```

An example testing procedure (cont’d)

- The function to be tested is in my_max.py
- The testing program is in the file test_my_max.py
- Let us go through the testing program and run it
  - We won’t open my_max.py
- A few remarks:
  - You may be surprised to see that we are writing a program to test another program. Yes, this is additional work but it is absolutely necessary.
  - If you write a test program, you can re-use the test cases for the future versions of the software if needed

Different test methods

- The method that we were using is known as black-box testing
  - We didn't look at the code
  - We simply applied the test cases and compared against the expected output

- There is also glass-box testing where tests are derived by looking at the code

Choosing test cases (1)

- It’s important that you choose test cases in a diverse way to cover as many possibilities as possible
- What limitations do you see in the test cases we’ve used? How can you improve it?

Test cases	**Expected output**
[2, 5, 8]	8
[3, 17, 19, 24]	24
[23, 1, 51, 19, 107, 123]	123

Choosing test cases (1)

Test cases	Expected output
[2, 5, 8]	8
[3, 17, 19, 24]	24
[23, 1, 51, 19, 107, 123]	123

Choosing test cases (2)

- It is important that you choose tests as diverse as possible

- In order to test how robust a software is, you may also want to consider
  - Empty list
  - Lists with a mixture of numerals and non-numeral types

- Many software companies test their software with random inputs in addition to using the expected inputs

Using different test cases

- Some new test cases are now included in the file test_my_max_v2.py. We now use:
- Let us run it and see
- It failed one test ☹
- There is a logic error in the code
  - This example came from a past ENGG1811 assignment submission

© UNSW, CRICOS Provider No: 00098G

Debugging

- Now the test has revealed that the program does not do what you intended it to do, you need to debug the program

- Sometimes you may be able to deduce the error by looking at the test cases that the program passes and fails

- “Code does not do what you intended” errors are harder because they are caused by a mismatch between:
    - What we want the program to do
    - The instructions we gave the computers

- Computers faithfully execute the instructions we give them
    - The instructions we gave to the computers are incorrect (swallow our pride 😔) and adjust our mindset

© UNSW, CRICOS Provider No: 00098G

Some debugging strategies

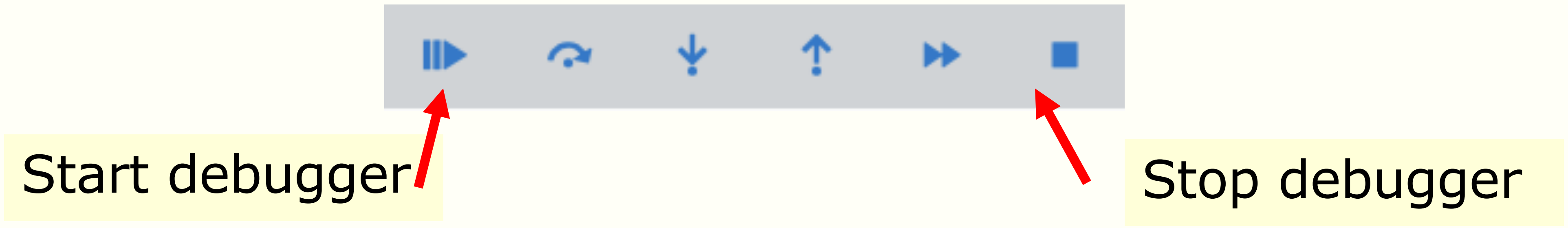
- Isolate the error
  - What is the smallest input that produces the error?
  - What is the smallest code that produces the error?

- Isolate the part of the code that produces the error
  - Selectively commenting out part of the code to locate the part of the code that produces the error

- Use print() statements to help you
  - Place print() in your code near where you think the error occurs
  - Check the values of the variables are what you expect at that point of the program
  - Don’t forget to remove these print() statements once you have fixed the errors

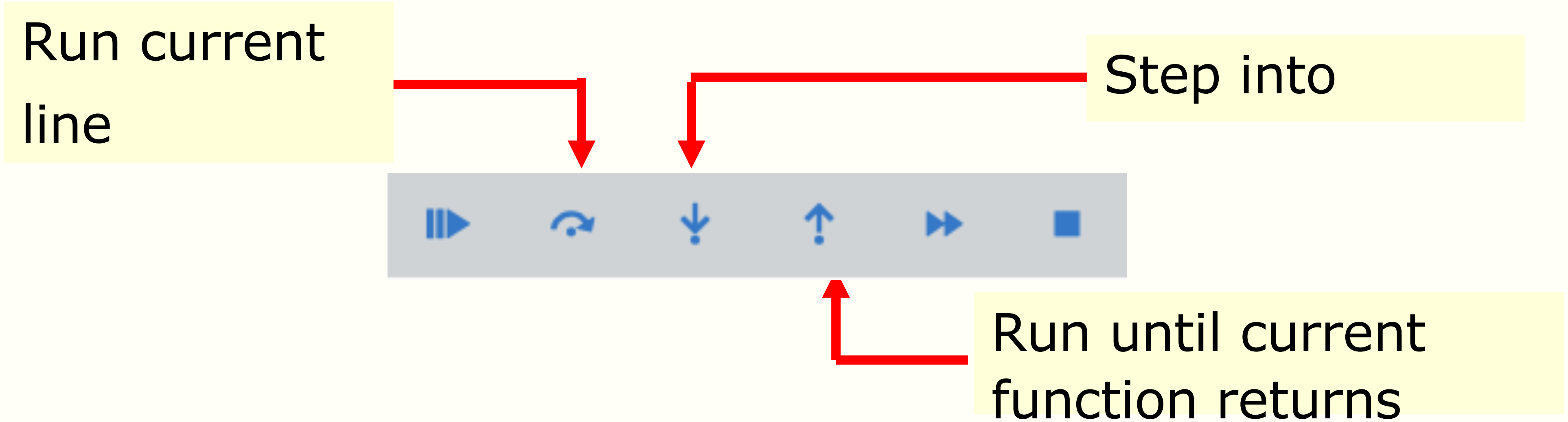
Debugger to trace programs

- Another debugging strategy is to trace the program
  - This is similar to the web visualisation tool that we have been using
- Spyder has a debugging tool
- These buttons are for debugging



Spyder debugger

- The Spyder debugger allows you to step through your program in a few different ways
- You can step through the program one line at a time by running the current line
- If a line contains a function call, there are two options
  - The "step into" option: Stepping through the lines of the function
  - The "run until … returns " option: Execute the function without stepping through the lines of the function



Variable explorer

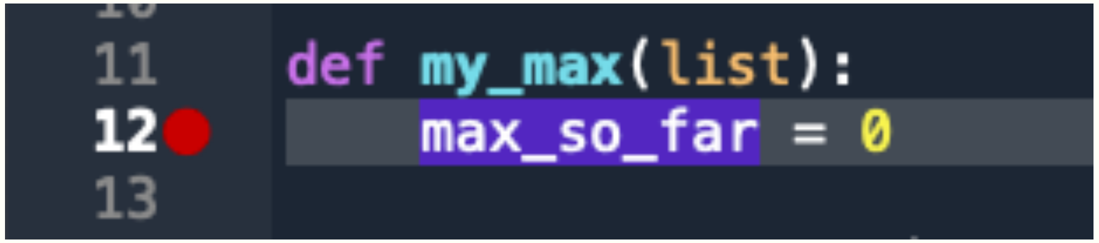
- The debugger is very often used in conjunction with the variable explorer so that you can observe changes in variables
- It’s best to clear the variables so that the existing variables won’t confuse the debugging process
  - Spyder automatically clears the variables when the debugger is invoked

Demonstration

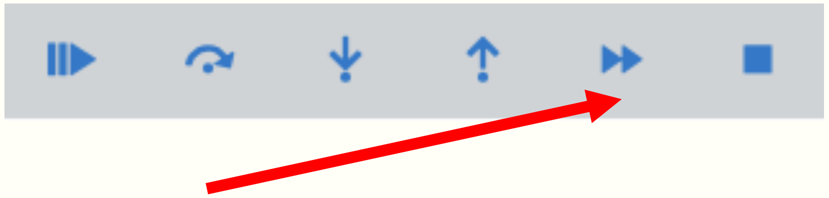
- We will use the files debug_my_max.py and my_max.py to demonstrate these functions
- We will try
  - Stepping through one line at a time
  - Step into
  - Run until the current function returns

Breakpoint

- Very often you want to skip a block of code instead of running a line at a time
- You can set breakpoint and run the program until the line before the breakpoint has been executed
- To set a breakpoint at a line, click on the space immediately to the right of the line number
  - A solid red circle indicates a breakpoint
  - Click on the red circle to remove the breakpoint
- Demonstration



```
11  def my_max(list):
12●     max_so_far = 0
13
```



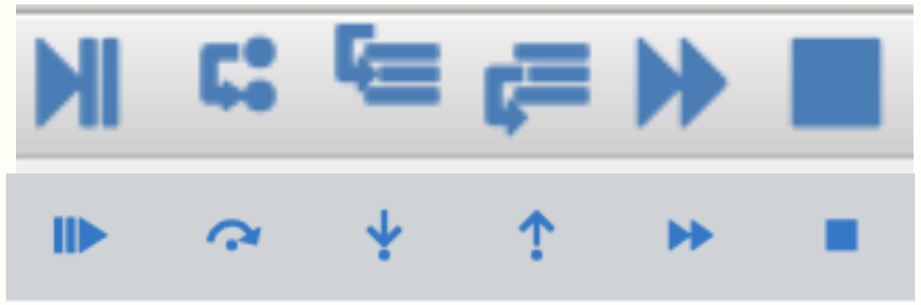
Run till breakpoint

Quiz: poor choice of variable name

- The program my_max.py has a poor choice of variable name. What is it?

Spyder debugger buttons (older version)

- The CSE labs has an older version of Spyder. The debugger buttons looked slightly different but they are arranged in the same order



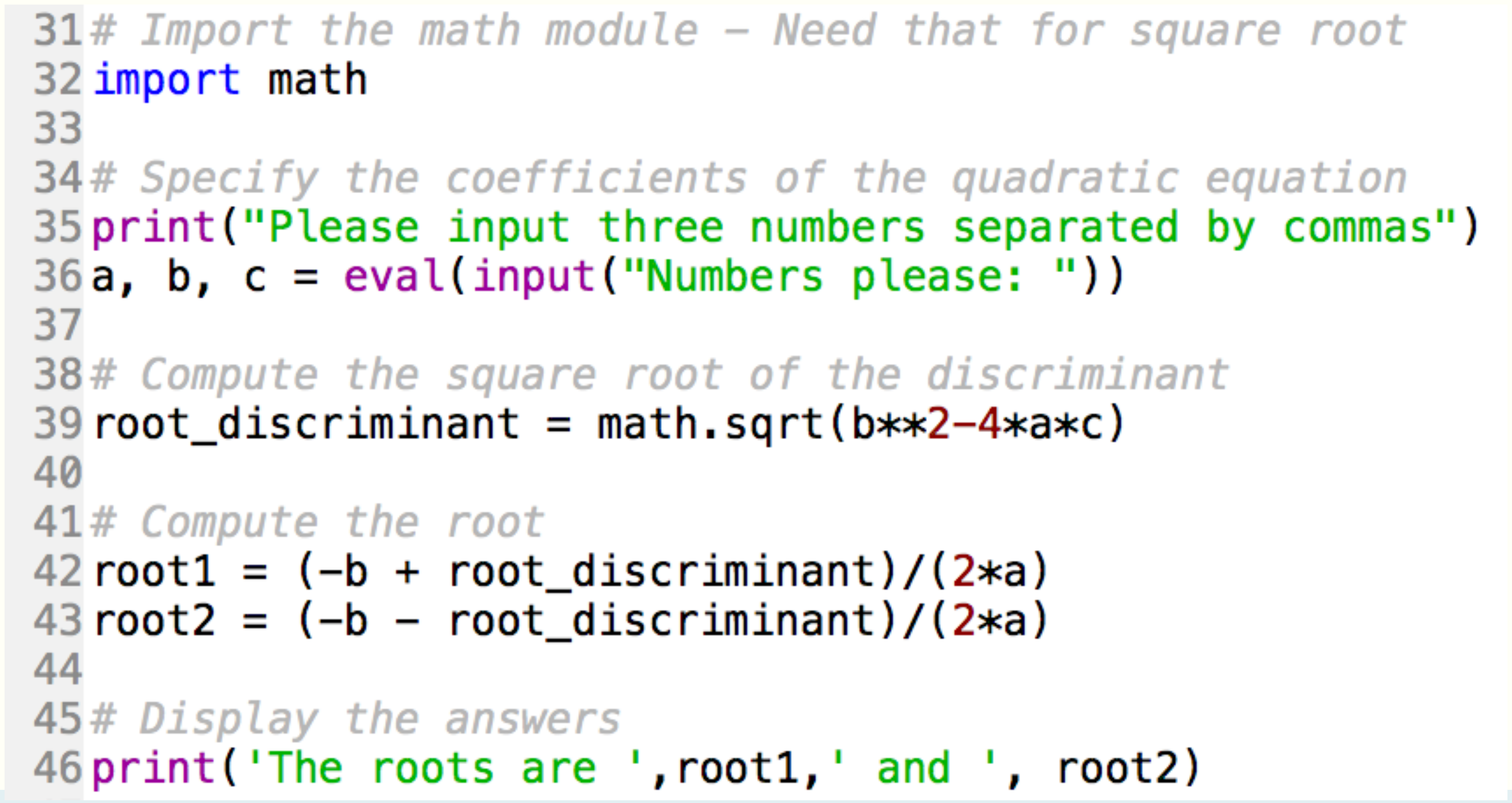
Updated:
It is now possible to run Spyder 5.3.3 on the CSE system. The instructions are provided on the course forum. We haven't thoroughly tested so if there are any problems, please let us know.

- From left to right
  - Start debugging
  - One line at a time / Run current line
  - Step into
  - Step return / Run until current function returns
  - Run until breakpoints
  - Stop debugging

- Errors, program testing and debugging
- Exception handling

Error handling

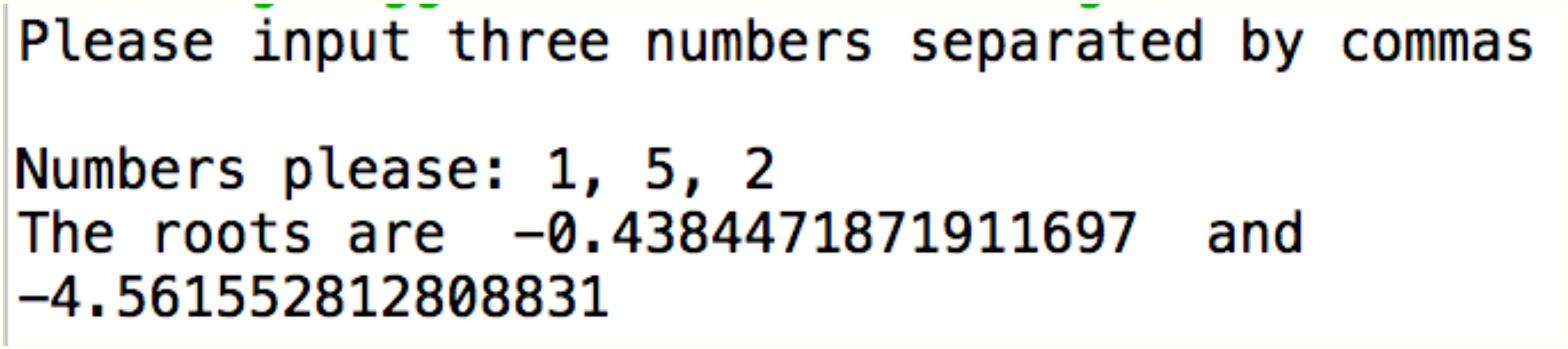
- Let us consider the program for calculating the roots of quadratic equation
  - We used that in Week 1
  - Code in quadratic_v1.py



```python
# Import the math module - Need that for square root
import math

# Specify the coefficients of the quadratic equation
print("Please input three numbers separated by commas")
a, b, c = eval(input("Numbers please: "))

# Compute the square root of the discriminant
root_discriminant = math.sqrt(b**2-4*a*c)

# Compute the root
root1 = (-b + root_discriminant)/(2*a)
root2 = (-b - root_discriminant)/(2*a)

# Display the answers
print('The roots are ',root1,' and ', root2)
```

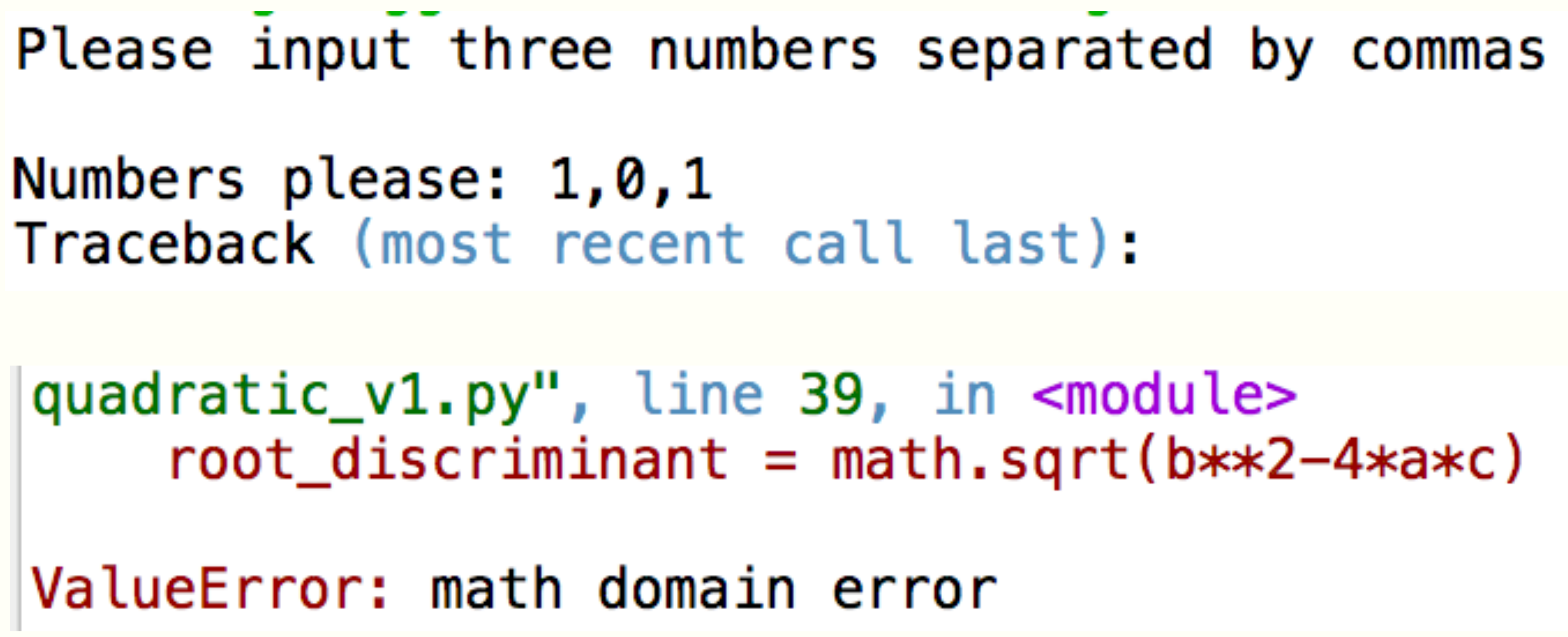
© UNSW, CRICOS Provider No: 00098G

Expected usage



```
Please input three numbers separated by commas

Numbers please: 1, 5, 2
The roots are  -0.4384471871911697  and
-4.561552812808831
```

Unexpected usage



```
Please input three numbers separated by commas

Numbers please: 1,0,1
Traceback (most recent call last):

quadratic_v1.py", line 39, in <module>
    root_discriminant = math.sqrt(b**2-4*a*c)

ValueError: math domain error
```

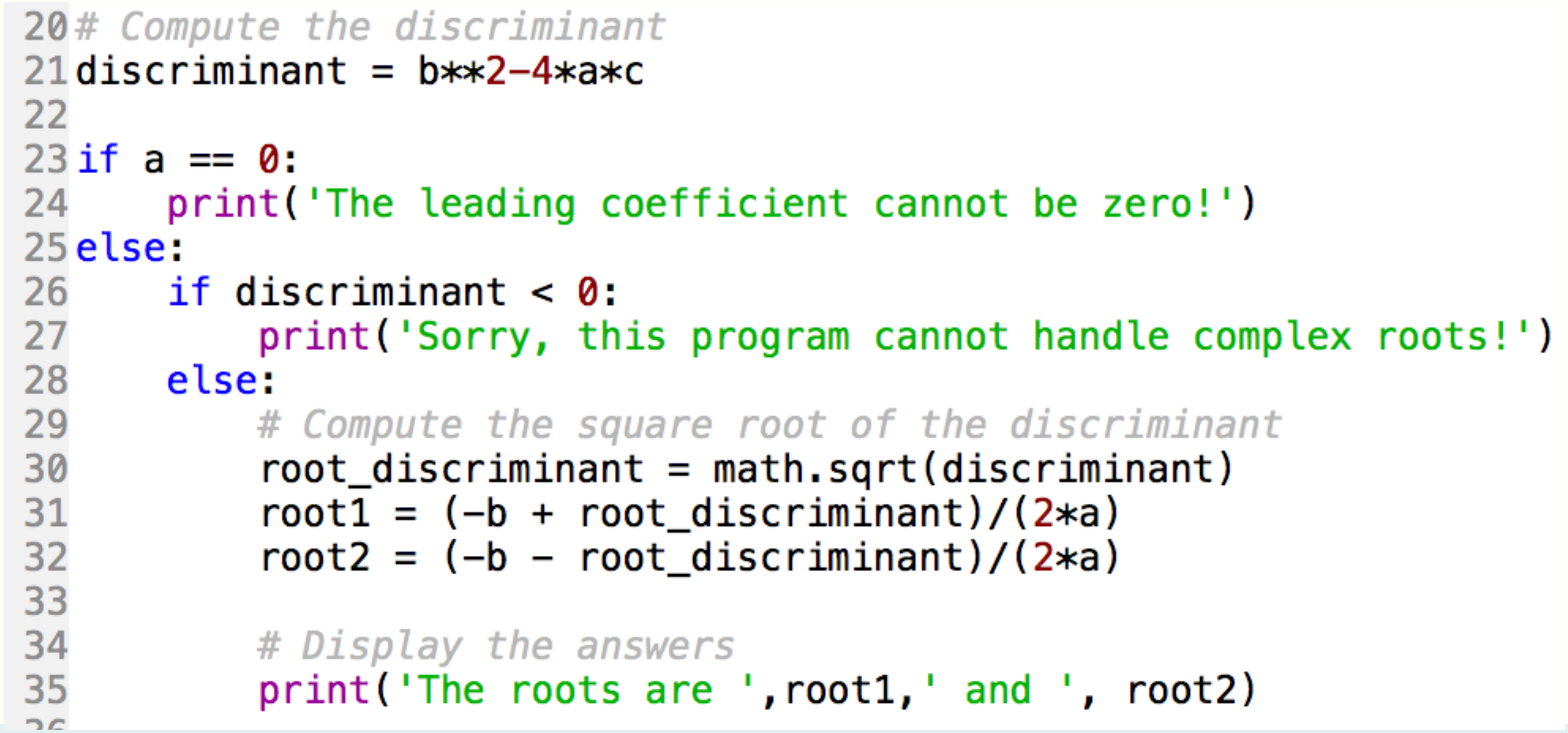
- You can’t always expect the users to know the limitation of your software

Error or exception handling

- You want to make sure that your program can handle unexpected user input
- Two methods:
  - if/else if/elif/else
  - try   except

Avoiding errors

- You can try to avoid programs running into problem by considering possible errors in your program
- The modified code is in quadratic_v2.py. It checks:
  - The leading coefficient is not zero
  - The discriminant is not negative



```python
# Compute the discriminant
discriminant = b**2-4*a*c

if a == 0:
    print('The leading coefficient cannot be zero!')
else:
    if discriminant < 0:
        print('Sorry, this program cannot handle complex roots!')
    else:
        # Compute the square root of the discriminant
        root_discriminant = math.sqrt(discriminant)
        root1 = (-b + root_discriminant)/(2*a)
        root2 = (-b - root_discriminant)/(2*a)

        # Display the answers
        print('The roots are ',root1,' and ', root2)
```

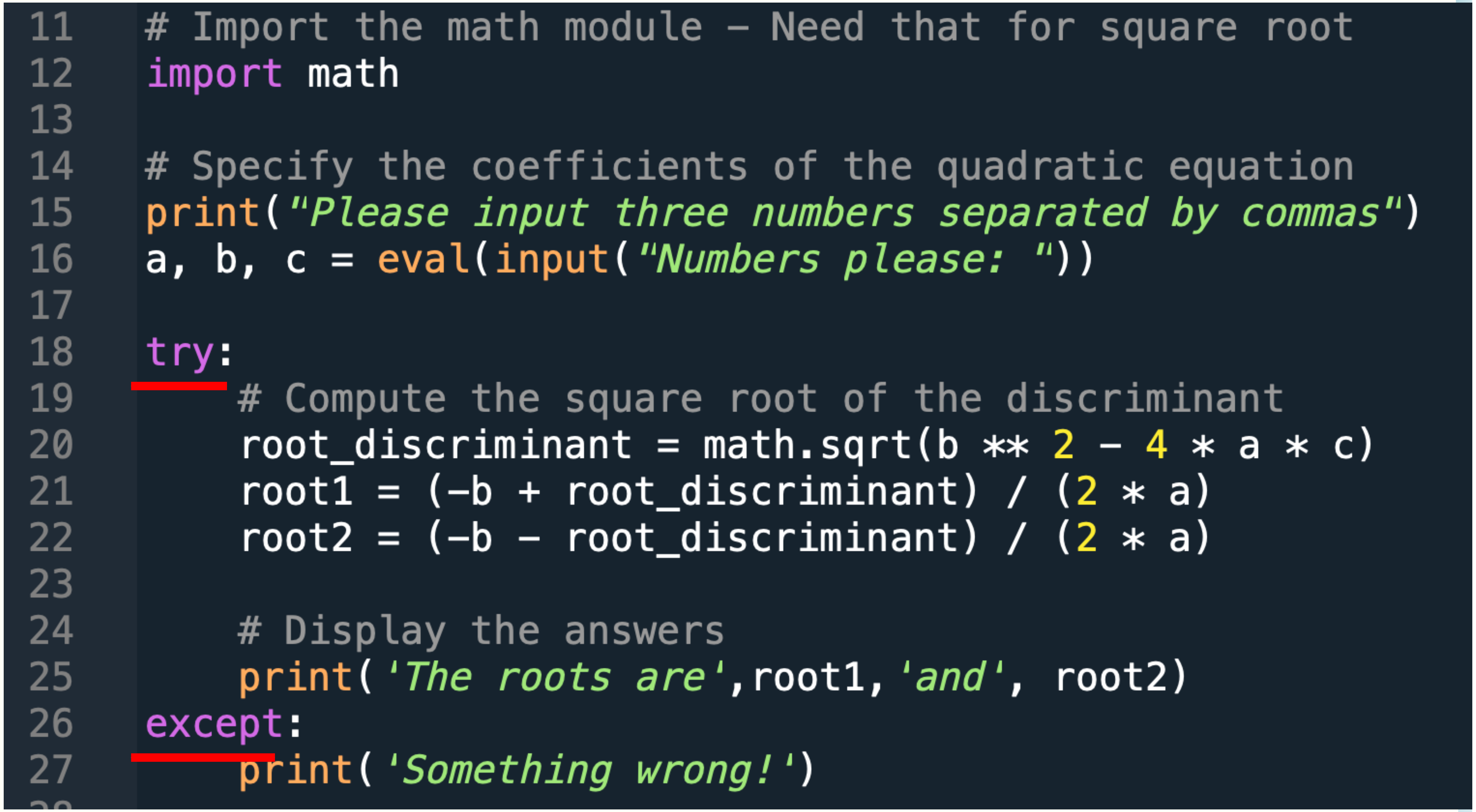
Python try … except

- Instead of using if/elif/else to handle the special cases, you can also use try … except
- The code under try will be run first, if it results in an error, the code under except will be run

```
try:
    # Code under the try block
    #
except:
    # Code under the except block
    #
```

try … except

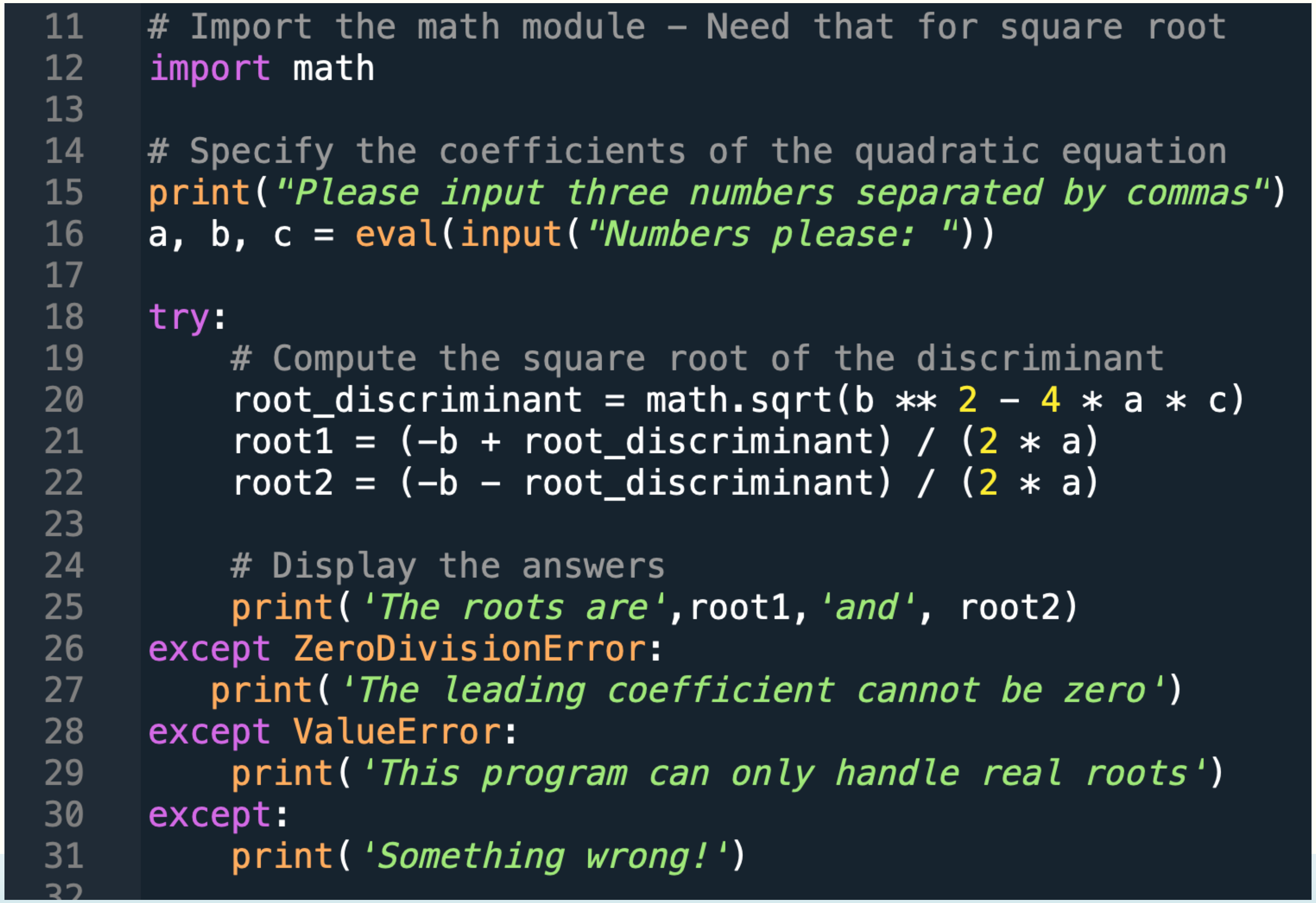
- Example: Modify quadratic_try_v1_prelim.py to:



```python
# Import the math module – Need that for square root
import math

# Specify the coefficients of the quadratic equation
print(“Please input three numbers separated by commas”)
a, b, c = eval(input(“Numbers please: ”))

try:
    # Compute the square root of the discriminant
    root_discriminant = math.sqrt(b ** 2 - 4 * a * c)
    root1 = (-b + root_discriminant) / (2 * a)
    root2 = (-b - root_discriminant) / (2 * a)

    # Display the answers
    print(‘The roots are’,root1,‘and’, root2)
except:
    print(‘Something wrong!’)
```

Python try … except

- You can make the exception handling more precise by handling each type of exception
- Code in quadratic_try_v2.py



```python
# Import the math module - Need that for square root
import math

# Specify the coefficients of the quadratic equation
print("Please input three numbers separated by commas")
a, b, c = eval(input("Numbers please: "))

try:
    # Compute the square root of the discriminant
    root_discriminant = math.sqrt(b ** 2 - 4 * a * c)
    root1 = (-b + root_discriminant) / (2 * a)
    root2 = (-b - root_discriminant) / (2 * a)

    # Display the answers
    print('The roots are',root1,'and', root2)
except ZeroDivisionError:
   print('The leading coefficient cannot be zero')
except ValueError:
    print('This program can only handle real roots')
except:
    print('Something wrong!')
```

Summary

- Running code does not mean correct code
- Test, test, test
- Writing test code
- Debugging
  - Useful skills for your assignment
- Exception handling