Week 4b: Errors; Program testing and debugging; Exception handling

Professor Aaron Quigley

Thanks to Chun Tung Chou

and Ashesh Mahidadia

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 code to run but the code doesn't do what you intend 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:
  - Misspelling 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 instructions

  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 and 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 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 improved 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
- 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

Debugging

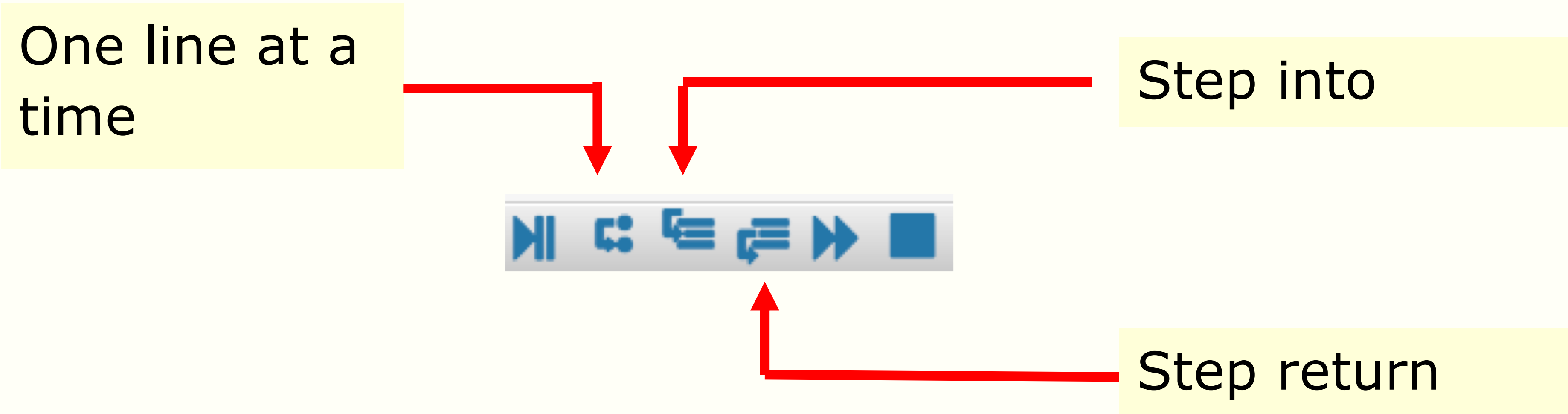
- Now the test has revealed that the program does not do what you intend it to do, you need to debug the program
- Sometimes you may be able to deduce the error by looking at the test cases that the program passes and fails
- Another way is to trace the program
  - This is similar to the web visualisation tool that we have been using
- Spyder has a debugging tool
- These buttons are for debugging

Start debugger

Stop debugger

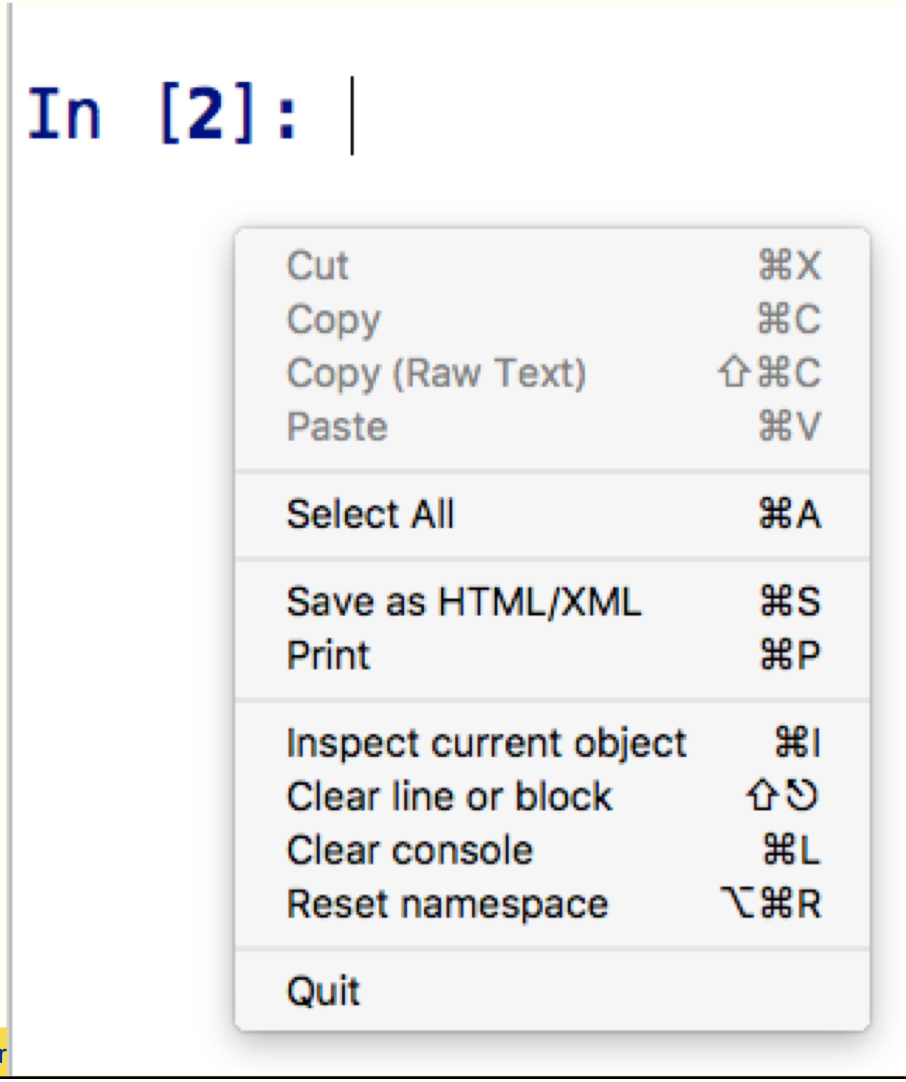
Spyder debugger

- 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
- If a line contains a function call, there are two options
  - The "step into" option: Stepping through the lines of the function
  - The "step return" 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
- You may wish to clear the variables so that the existing variables won't confuse the debugging process
  - Right click on the blank space in the console
  - From the pop-up menu, choose "Reset Namespace"



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
  - Step return

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, double click on the grey space to the left of the line number
  - A solid red circle indicates a breakpoint
  - Double click on the red circle to remove the breakpoint
- Demonstration

```
def my_max(list):
    max_so_far = 0

```

Run till breakpoint

Error handling

- Let us consider the program for calculating the roots of quadratic equation
  - You did one in Week 1
  - Code in quadratic_v1.py

```
# 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)
```

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

Avoiding errors

- You can try to avoid programs running into problem by considering possible errors in your program
- The code is in quadratic_v2.py

```
# 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
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

End

Week 4b: Errors; Program testing and debugging; Exception handling