 Exceptions
Software Quality

Motivation
• Things can (and do) go wrong in a program
• Crashing the program is too drastic
• Exceptions and exception handling enables a program to deal with anomalous conditions gracefully
• Exceptions are errors that occur at runtime
• Allows the separation of error detection and error handling

Exceptions
• Syntax:
  try: <body>
  except: <handler>
• <body> contains code that may raise an exception
• <handler> is code to deal with the exception

What is printed?
try:
  print('Hi')
  x = 3 / 1
  print(x)
except:
  print('Error')
• No error in try block
• except block is completely skipped

What is printed?
try:
  lst = 10 * [0]
x = lst[10]
  print('Done')
except IndexError:
  print('Index out of bounds')
• If an error occurs:
  • No statements after the error generating code are executed
  • Statements in the exception block are executed
Multiple Exceptions

```
try:
    lst = [1, 3, 5]
    u = lst[2] / 0
    x = lst[4]
    print('Done')
except IndexError:
    print('Index out of bounds')
except ZeroDivisionError:
    print("Silly! That's undefined!")
print('Whew!')
```

Common Exceptions

- **Exception**
  - **AttributeError**
  - ** IndexError**
  - **NameError**
  - **OS**
    - **Error**
  - **FileNotFoundError**
  - **SyntaxError**
  - **TypeError**
  - **ValueError**
  - **ArithmeticError**
    - **ZeroDivisionError**

Common Exceptions (at least for us)

- **NameError**
  - Occurs when you attempt to use a variable before it has a value
- **TypeError**
  - Occurs when you combine two objects that are not compatible (e.g. 3 * 'hi')
- **ValueError**
  - Occurs when you pass a parameter to a function and the function is expecting a certain type, but you pass it a different type

Exception Heirarchy

```
try:
    lst = [1, 3, 5]
    u = lst[2] / 0
    print('Done')
except ArithmeticError:
    print('No good!')
except ZeroDivisionError:
    print("Silly! That's undefined!")
print('Whew!')
```

- **except** clauses are like **elif**:
  - Only one clause is executed
  - First match is used!!

Example

- Write a program to get a positive integer from the user. The user should be repeatedly prompted until they enter a valid value, and should be told what went wrong if they enter an invalid value.

```
while True:
    try:
        x = int(input("Please enter a positive integer: '))
        if 0 < x:
            break
        else:
            print('Your value is not positive.')
    except ValueError:
        print('That was not an integer.')
```
Signal an Anomalous Condition

• Syntax:
  ```python
  raise <ExceptionType>(<message>)
  ```

• `<ExceptionType>` is a valid error type
• `<message>` is information about what went wrong

Instead of crashing the program...

```python
class Fraction:
  def __init__(self, numerator = 0, denominator = 1):
    if denominator == 0:
      print('This is a not a valid denominator', file = sys.stderr)
      sys.exit()
    else:
      self._numerator = numerator
      self._denominator = denominator
      self._simplify()
```

In the calling program...

• Original:
  ```python
  f1 = Fraction(3, 0)
  ```
  • Crashed the program

• With exceptions:
  ```python
  try:
    f1 = Fraction(3, 0)
    print(f1)
    print(f1.get_denominator())
  except ValueError as excpt:
    print(excpt)
  ```

What is the output?

```python
try:
  value = 35
  if value < 40:
    raise ValueError('Too small!')
  print('Heya!')
except ValueError as err:
  print(err)
print('Pickle')
```

Example

• Write a program that creates a list of 100 random numbers (integers between -100 and 100). The program should then prompt the user for an index in the list, and display the value at the index.
  • Should handle non-integer inputs gracefully
  • Should handle invalid indices gracefully

Example Solution

```python
import random
x = []
for i in range(100):
  x.append(random.randint(-100, 100))
try:
  index = int(input("Please enter an index:"))
  print(f'The value is', x[index])
except ValueError:
  print('That was not an appropriate index')
except IndexError:
  print('Valid indices are between -100 and 99 inclusive')
```
What is the output?

```python
def func1(p):
    print('value of p:', p)
    return p * 0
x = input('Enter a value for x: ')
p = int(x)
print(func1(p))
print('done')
```

Enter a value for x: 4

done

What is the output?

```python
def func1(p):
    print('value of p:', p)
    return p * 0
x = input('Enter a value for x: ')
p = int(x)
print(func1(p))
print('done')
```

Enter a value for x: 5

Update main code to handle any exceptions

```python
def speed (time, distance):
    if (time <= 0):
        raise ValueError('Impossible to go back in time')
    return distance / time
t = float(input('Enter time: '))
dist = float(input('Enter distance: '))
print('Your speed is:', speed(t, dist))
```

```python
def speed (time, distance):
    if (time <= 0):
        raise ValueError('Impossible to go back in time')
    return distance / time
t = float(input('Enter time: '))
dist = float(input('Enter distance: '))
print('Your speed is:', speed(t, dist))
```

```python
except ValueError as v:
    print(v)
except TypeError as t:
    print('This only works for numbers')
```

finally and else

- `finally` allows code to be executed regardless of any exceptions that may or may not have been raised
  - must be last clause
- `else` is executed only if no exceptions were raised
Example

```python
try:
infile = open('sales.txt')
for line in infile:
    total += float(line)
infile.close()
except Exception as err:
    print(err)
else:
    print('total:', total)
finally:
    print('off to the races')
```

[Errno 2] No such file or directory: 'sales.txt'
off to the races

Quadratic Equation

- Write a function to compute the roots of a quadratic equation.
  - Given: coefficients a, b and c
  - Return: A tuple of the roots
  - Raise an exception if there are no real roots

- Write a program that prompts for the coefficients and then uses the function to compute the roots
  - Ensure your program never crashes

New Exception Types

- Exceptions are objects
- Define a new class to be a kind of exception
- Syntax:

  ```python
class NewExceptionError(Exception):
    def __init__(self, msg):
        self._message = msg
    def __str__(self):
        return self._message
    ```

- Can add accessor, mutator, etc. as appropriate

Testing

- Testing is a much maligned, but critically important aspect of software development.
- Principles:
  - Test all aspects of the application.
  - You can’t test every possible case.
  - Test rigorously.
  - Test early and test often.

Varieties of Testing

- Functional testing
  - Levels of functional testing:
    - Unit testing
    - System testing

- Non-functional testing
  - Usability testing

A common mistake people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

- Douglas Adams, Mostly Harmless, 1992

Functional Testing Variations

- Print statements
- assert statements
- unittest module
Simplest Approach: `print`

```python
if __name__ == '__main__':
    tv1 = Television()
    print(tv1)
    tv1.toggle_power()
    tv1.set_channel(43)
    print(tv1)
    tv1.toggle_power()
    tv1.set_channel(43)
    print(tv1)
```

Testing Exceptions with `print`

```python
def func(x):
    if x < 0:
        raise ValueError('No negatives allowed: ' + str(x))
    else:
        return x + 1
```

• How do we test this??
  • 2 cases: good value, bad value

**Good value test**
```python
> y = func(3)
```
```python
try:
    y = func(3)
    print(y)  # Visual check
except:
    print('ERROR!')
```

**Bad value test**
```python
> y = func(-3)
```
```python
try:
    y = func(-3)
    print(y)  # Visual check
except:
    print('ERROR!')
```

Testing with `assert`

• Motivation:
  • Visual checks are error prone
  • Not fully automated

• Assertions are a systematic way to check that the state of a program is as the programmed intended.

```python
assert y == 4  # Check the value
except:
    assert False  # problem will be signaled
```

Still need two cases:

**Good value test**
```python
try:
    y = func(3)
    assert y == 4  # Check the value
except:
    assert False  # problem will be signaled
```

**Bad value test**
```python
try:
    y = func(-3)
except Exception as e:
    assert isinstance(e, ValueError)
```

Testing with `unittest`

```python
import unittest
import math

class FuncTest(unittest.TestCase):
    def testSin(self):
        assert math.sin(0) == 0
    def testCos(self):
        assert math.cos(0) == 1
```

```python
if __name__ == '__main__':
    unittest.main()
```
Run As... Unit Test (PyUnit)

```python
from unittest import 
import mock

class FunTest(unittest.TestCase):
    def testTrue(self):
        mock.Mock(string) == True
        mock.Mock(string) == True

if __name__ == '__main__':
    unittest.main()
```

Run as ... unit test (console)

```bash
$ python main.py
```

Run 2 tests in 0.001s

```python
from unittest import 
import mock

class FunTest(unittest.TestCase):
    def testTrue(self):
        mock.Mock(string) == True
        mock.Mock(string) == True

if __name__ == '__main__':
    unittest.main()
```