CMPUT 274

Exceptions and Handling Errors

Topics Covered:

- What are exceptions?
- Raising exceptions
- Catching exceptions
- Assertions

When Things Go Wrong...

- We have talked about input validation.
  - Good practice to always check the input that a user provides to make sure it is what is expected in terms of *type*, *value interval*, etc.
  - This avoids some errors.
  - But errors can still happen.
- Programs must be able to handle unusual situations and act appropriately when things go wrong.

Reacting to Errors

- What if:
  - the user enters a string instead of an integer?
  - we try to divide by zero?
  - We try to read from a file which doesn't exist?
  - We try to access a remote location but the network is down?
- Should the program produce bad output, or should it crash, or should it try to recover?
- Python provides some helpful mechanisms to assist us by handling exceptions.

What is an Exception?

- **An exception** is an event which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- Exceptions allow us to handle errors or other exceptional conditions.
- In Python, an exception is an object that represents an error.

Exceptions in Python

- How do we recover from errors, or at least handle them gracefully?
- In general, when a Python script encounters a situation that it can't cope with, it raises an exception at the point where the error is detected.
  - The Python interpreter raises an exception when it detects a run-time error.
  - A Python program can also explicitly raise an exception.

Python Interpreter and Exceptions

```
>>> '2013' + 1
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

```
>>> 175 + cmput*13
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
NameError: name 'cmput' is not defined
```

```
>>> 365 * (12/0)
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Other Common Exceptions

- Accessing a non-existent dictionary key will raise a KeyError exception.
- Searching a list for a non-existent value will raise a ValueError exception.
- Calling a non-existent method will raise an AttributeError exception.
- A hierarchy of all of Python's exceptions: https://docs.python.org/3/library/exceptions.html

Why Use Exceptions?

- The advantages of using exceptions include:
  - separating error-handling code from regular code
  - deferring decisions about how to respond to exceptions
  - providing a mechanism for specifying the different kinds of exceptions that can arise in our program

Exception Handling Blocks

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem

```
try:
    f = open('myfile.txt', 'r')
except IOError:
    print("File does not exist or cannot be read.")
```

- Once an exception has been handled, processing continues normally on the first line after the **try...except** block.

Catching Exceptions

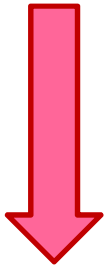
- If you don't catch the exception, your entire program will crash
- An except clause may name one exception, or multiple exceptions as a parenthesized tuple
  - e.g. `except RuntimeError:`
  - e.g. `except (RuntimeError, TypeError, NameError):`
- An except clause without explicit exception will catch all remaining exceptions

```
try:
    some statements
except:
    print("Unexpected error")
```

Multiple Except Clauses

- A **try** statement may have more than one except clause, to specify handlers for different exceptions.
- At most, one handler will be executed.
- Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same **try** statement.

specific



general

```
try:
    f = open('myfile.txt', 'r')
    s = f.readline()
    i = int(s.strip())
except IOError :
    print ("File does not exist or cannot be read.")
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error")
```

The `try` Statement

- When we write a piece of code that we know might produce an exception and we want to handle that exception, we encapsulate that code in a try…except block.
- If no exception is raised by the code within the try block (or the methods that are called within the try block), the code executes normally and all except blocks are skipped.
- If an exception arises in the try block, the execution of the try block terminates execution immediately and an except is sought to handle the exception. Either
    1. *An appropriate except clause is found, in which case it is executed*, or
    2. *The exception is propagated to the calling method/outer try*
    3. No handler is found - it is an *unhandled exception* and execution stops with a message (program crashes)

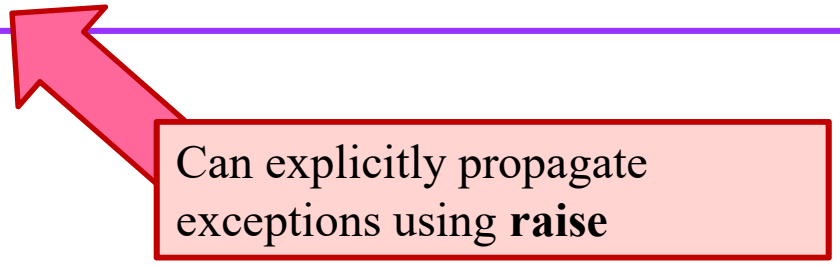
Propagating Exceptions

- An exception will bubble up the call stack until:
  - it reaches a method with a suitable handler, or
  - it propagates through the main program (the first method on the call stack)

- If it is not caught by any method the exception is treated like an error: the stack frames are displayed and the program terminates

Handling and Propagating

```
try:
    f = open('myfile.txt', 'r')
    s = f.readline()
    i = int(s.strip())
except IOError :
    print ("File does not exist or cannot be read.")
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error")
    raise
```



Raising Exceptions

- What can be raised as an exception?
  - Any standard Python Exception
  - A new instance of Exception with custom arguments
  - Instances of our own specialized exception classes

```
try:
    print("Raising an exception")
    raise Exception('CMPUT', '274')
except Exception as inst:    # the exception instance
    print(inst.args) # arguments stored in .args
    x, y = inst.args # unpack args
    print('x =', x,'y =', y)
```

```
Raising an exception
('CMPUT', '274')
x = CMPUT y = 274
```

`else` Clause

- The try…except statement has an optional *else clause*, which, when present, must follow all *except clauses*.
- The code in the else clause must be executed if the try clause does not raise an exception

```
try:
    f = open('myfile.txt', 'r')
except IOError :
    print ("File does not exist or cannot be read.")
else:
    print("the file has", len(f.readlines()), "lines")
    f.close()
```

`finally` Clause

- If you want to execute some code whether an exception was raised or not, you can use the optional *finally clause*
- Guaranteed to be executed, no matter why we leave the try block (i.e. executed under all circumstances)
- Useful if you want to perform some kind of "clean up" operations before exiting the method
  - example: closing a file
- Also avoids duplicating code in each *except* clause

`finally` Example

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("thanks for dividing")
```

finally clause is executed in **any** event

- no exception
- division by 0
- type error

Type error is **re-raised** after the *finally* clause since no *except* exists for it.

```
>>> divide(2, 1)
result is 2.0
thanks for dividing
>>> divide(2, 0)
division by zero!
thanks for dividing
>>> divide("CMPUT", "175")
thanks for dividing
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    File "<stdin>", line 3, in divide
TypeError: unsupported operand
type(s) for /: 'str' and 'str'
```

Summary: Possible Execution Paths

1. No exception occurs
    1. Execute the try block
    2. Execute the else and finally clauses
    3. Execute the rest of the method
2. Exception occurs and is caught
    1. Execute the try block until the first exception occurs
    2. Execute the first except clause that matches the exception
    3. Execute the finally clause
    4. Execute the rest of the method
3. Exception occurs and is **not** caught
    1. Execute the try block until the first exception occurs
    2. Execute the finally clause
    3. Propagate the exception to the calling method

Assertions

- An assertion is a statement that raises an AssertionError Exception if a condition is not met.

```
assert Expression[, Arguments]
```

- If the assertion fails, Python uses the given argument as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception.

- It is good practice to place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

Assertion Example

```
def KelvinToFahrenheit(temperature):
    assert (temperature >= 0),"Colder than absolute zero!"
    return ((temperature-273)*1.8)+32

if __name__ == "__main__":
    try:
        fahrenheit = KelvinToFahrenheit(-23)
        print(fahrenheit)
    except AssertionError as my_error:
        print(my_error.args)
```

```
(Colder than absolute zero!, )
```