# System: Exceptions

## CPSC 231: Introduction to Computer Science for Computer Science Majors I
## Fall 2021

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

**Tuesday, 31 August 2021**



UNIVERSITY OF CALGARY

# Revisiting Errors

- Previously, you learned about the three main types of errors:
  1. Syntax Errors: refers to errors in the structure of a program and the rules about that structure.
  2. Semantic/Logic Errors: refers to errors in the logic of a program
  3. Runtime Errors: refers to errors that occur during program execution

- Runtime Errors are also referred to as ***Exceptions***

# Exceptions

- An **exception** is an event that occurs during the execution of a program, which disrupts its execution.
- Exceptions can rise due to many reasons, including improper use of functions or operators, user input, logic errors, hardware and OS limitations, etc.
- Examples:
  - trying to access a list with an invalid index
  - trying to open a non-existent file
  - trying to parse a string using an invalid character
  - trying to converting a string to an integer
  - ...

# Exceptions - Exception Handing

- Exceptions can be handled is several ways:
  - Using conditionals: the code handles scenarios where errors may occur.
  - Using **try/except** blocks: placing code that may fail within a try/except block.

# Try/Except

# Exceptions – Try/Except Block

Always executed, but may raise exception(s).

```
try:
    <code segment that may cause error>
except:
    <action to take when an exception occurs>
```

executed only if exception(s) occur

catch any exception that may occur within the try block.

# Exceptions - Exception Handing

- Example: open a file.

```
from os import path

if path.exists("no file"):
    print("File exists")
else:
    fileHandler = open("file does not exit")
    print("File does not exist")
```

```
try:
    fileHandler = open("file does not exit")

except:
    print("Oops! Something went wrong.")
```

# Exceptions - Exception Handing

- It is good coding practice to catch potential errors using conditionals instead of try/except blocks.
- However, in some cases, you must try/except.

- Consider the open file example again. What happens if the file exists, but by the time the open() function is called, the file no longer exists or another process had locked it?

# Exceptions - Exception Handing

- The following code will crash if: the file exists, but was removed, renamed, locked, etc. at the **input()** function call.

```
from os import path

filePath = "C:/…/inputFile.txt"

if path.exists(filePath): #File exists
    print("File exists")
    userInput = input("How U doin?") #file deleted
    fileHandler = open("file does not exit") #exception
```

- Try/except will handle the situation

# Exceptions - Exception Handing

- When to use if statements vs. try/except?
  - Use if statements to catch any as many potential errors as possible. However, as you have seen, not all situations can be captured (easily or at all) by if statements.
  - For such situations, use the try/except block.

# Naming Exceptions

# Exceptions – Try/Except Block - Catching Specific Exception

```
try:
      <code segment that may cause error>
except <exception type>:
      <action to take when an exception occurs>
```

catch any exception that may occur within the try block.

# Exceptions – Catching Several Specific Exceptions

- Every exception raised in Python code is of a specific type.
- Example:
  - Division by zero is an instance of *ZeroDivisionError:* `print(10/0)`
  - Passing correct data type to a function, but with the wrong value is an instance of *ValueError:* `int("Hello")`
- For a list of all built-in exception types see:

https://docs.python.org/3/library/exceptions.html

# Try/Except/Except

# Exceptions – Catching Several Specific Exceptions

```
try:
      <code segment that may cause error>
except (<type>, <type>, ...) as <obj name>:
      <action to take when an exception occurs>
except (<type>, <type>, ...) as <obj name>:
      <action to take when an exception occurs>
```

- It is a good coding practice to have one **except** for each exception type.

# Exceptions – Catching Several Specific Exceptions

- Example:

```
try:
    userInput = input(">Enter an int >0: ")
    value = int(userInput)     #<-potential ValueError
    print(1/value)             #<-potential ZeroDivisionError
except ValueError as value_error:
    print("ValueError: %s is not an int.“ % userInput)
except ZeroDivisionError as div_by_zero_except:
    print("ZeroDivisionError - Cannot divide by Zero")
except:
    print("Something else went wrong!")
```

# Try/Except/Else

# Exceptions – Try/Except Block - *else* and *finally* Causes

- *else* is an optional clause that may appear after all the *except* clauses.
  - Code within the *else* clause is executed only if the code under the *try* clause did not raise an exception.
  - It is useful in situations where you don't want some code to be protected by the *try* clause and you want the code to execute only when the code protected by *try* did not raise an exception.

```
try:
    userInput = input(">Enter an int >0: ")
    value = int(userInput)#<- potential ValueError
    print("End of Try clause.")
except ValueError as value_error:
    print(value_error)
else:
    print("In else clause: No exception occurred")
```

```
>Enter an int >0: 10
End of Try clause.
In else clause:…
```

# Try/Except/Finally

# Exceptions – Try/Except Block - *else* and *finally* Causes

- *finally* is also an optional clause that may appear after all the *except* clauses and the *else* clause (if it was present).
  - Code within the *finally* clause always executes even if an exception is raised.
  - Code within the *try* clause stops executing at the line where an exception arise. So, it is important to place any code that must execute in *finally.*
  - Also, if an exception is risen and the except clause ends with a return, the finally clause will execute before the return statement.
    - A consequence of this is that: if *finally* ends with a *return*, then the *return* in the *except* clause will be ignored.

# Exceptions – Try/Except Block - *else* and *finally* Causes

```
def foo():
    try:
        userInput = input(">Enter an int >0: ")
        value = int(userInput)#<- potential ValueError
        return value
    except ValueError as value_error:
        print(value_error)
        return None
    finally:
        print("In finally clause.")
        #return -1

print('foo() returned: %s' % foo())
```

```
>Enter an int >0: 1
In finally clause.
foo() returned: 1
```

```
>Enter an int >0: a
invalid literal…
In finally clause.
foo() returned: None
```

# Exceptions – Try/Except Block - *else* and *finally* Causes

```
def foo():
    try:
        userInput = input(">Enter an int >0: ")
        value = int(userInput)#<- potential ValueError
        return value
    except ValueError as value_error:
        print(value_error)
        return None
    finally:
        print("In finally clause.")
        #return -1

print('foo() returned: %s' % foo())
```

```
>Enter an int >0: 1
In finally clause.
foo() returned: -1
```

```
>Enter an int >0: a
invalid literal…
In finally clause.
foo() returned: -1
```

If this line is uncommented, then the return statements under except and try will not execute.

# Try/Except Summary

# Exceptions – Try/Except Block – Putting It All Together

```
try:
      <code segment that may cause error>
except (<type>, <type>, ...) as <obj name>:
      <action to take when an exception occurs>
else:
      <action to take when no exception occurs>
finally:
      <action to take in any case>
```

optional: can target certain types of exceptions

optional: a named exception object for accessing info on exception

optional: executed only if no exceptions

optional: executed regardless of the code outcome

# exit()

# Error Handling - exit() Function

- *exit(errCode)* is a Python built-in function that, when called, raises a *SystemExit* exception and exists Python.
  - This function is useful when you need to stop a program's execution and indicate that an error (of a certain code) occurred.
  - Other python programs, and functions can catch this exception and retrieve the error code to make decisions.
  - Non-python programs that execute your code can also receive the exit code.
  - Useful to communicate across programs that an error occurred.
  - You can define your own errCodes:
    - Some systems have a convention for assigning specific meanings to specific exit codes; Unix programs generally use 2 for command line syntax errors and 1 for all other errors.
  - Read more about it: https://docs.python.org/3/library/sys.html#sys.exit

# Error Handling - exit() Function

```
import sys

def div(a, b):
    if not isinstance(a, int) or not isinstance(b, int):
        sys.exit(-123)
    if b == 0:
        sys.exit(123)
    return a/b
```

# Examples

# Raising an Exception

```
import sys
def repeatStar(number):
    if not isinstance(number, int):
        raise Exception('%s not an int' % number)
    if number == 0:
        raise ValueError('parameter number is zero')
    return number*"*"
```

Raise generic exception

Raise ValueError

```
try:
    repeatStar(1.5)
    #repeatStar(0)

except (ValueError, Exception) as detail:
    print(detail)
```

```
1.5 is not an int
```

```
Parameter number is zero
```

# Invalid Index

- Accessing a list with an invalid index

```
try:
      filename = sys.argv[1]
except IndexError:
      print('Usage: ... ')
      sys.exit(1)
```

If no argument is passed, then IndexError will be raised.

# Can’t open file

- Trying to open an non-existent file

```
import sys
try:
      inputFile = open(sys.argv[1], 'r')
except (IOError, IndexError) as detail:
      print (detail)
      sys.exit(detail)
```

exit() accepts any data type

The variable detail will be assigned to which ever exception is raised first.

# Error Handling - Examples

```
def reverseLines(inFilename, outFilename):
    try:
        inFile = open(inFilename,'r')
        outFile = open(outFilename, 'w')
        for line in inFile:
            line = line.rstrip()
            outFile.write(line[::-1] + "\n")
    except IOError :
        print("Encountered problem")
    finally:
        inFile.close()
        outFile.close()

reverseLines("names.txt","ReverseNames.txt")
```

*a:b:c* counts in increments of ***c*** from ***a*** to ***b*** or from ***b*** to ***c*** depending on the sign of ***c*** (negative vs. positive).

# Onward to … Classes and Objects.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY