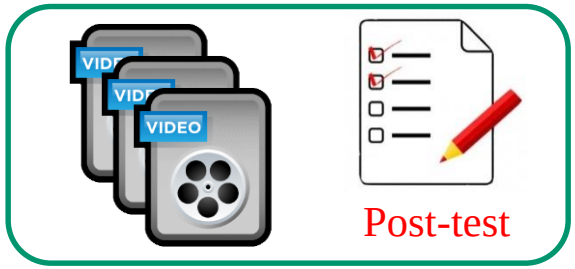
CMPUT 175
Introduction to Foundations of Computing

Exceptions and handling Errors



You should view the vignettes:

Call Stack
Handling Exceptions
Exceptions Metaphor

Objectives

In this lecture we will learn how to use Python **Exceptions** to handle unusual program conditions.

Outline

- When things go wrong
- Exceptions
- Catching exceptions
- Raising exceptions
- Defining your own exception classes
- Assertions

When things go wrong…

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

Reacting to errors

- What if
  - the user enters a string instead of an integer?
  - We try to pop an empty stack?
  - 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?

Handling Unusual Situations

- In many languages, the programmer is responsible for dealing with errors – no help from the language itself
- Code must be written to help identify kinds of errors and how to deal with them – convoluted code!
- Often, methods will return an unusual value such as –1 or null to indicate a failure – often difficult to interpret! [A common practice with C/C++]
- 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 are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions.
- An exception is a Python object that represents an error.

What is an Exception?

- 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 typical 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.
- Referencing a non-existent variable will raise a NameError exception.
- Mixing datatypes without coercion will raise a TypeError exception.

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

The 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')
except IOError :
    print("I/O error: 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
- except *Exception1*: catches Exception 1
- An except clause may name multiple exceptions as a parenthesized tuple, for example:

  except (RuntimeError, TypeError, NameError):

- An except clause without explicit exception will catch all remaining exception

```
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.

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError :
    print ("I/O error: 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

The **try** statement works as follows.

1. The *try clause* (the statement(s) between the **try** and **except** keywords) is executed.
2. If no exception occurs, the *except clause* is skipped and execution of the **try** statement is finished.
3. If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the **except** keyword, the except clause is executed, and then execution continues after the **try** statement.
4. If an exception occurs which does not match the exception named in the except clause, it is passed on to outer **try** statements; if no handler is found, it is an *unhandled exception* and execution stops with a message.

Exception handling

- When we write a piece of code that we know might produce an exception and we want to handle the 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.
- 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*

What is the output ?

```
try:
    alist = 5 * [0]
    last_item = alist[len(alist)]
    print('Done')
except IndexError:
    print('Invalid index')
```

Invalid index

```
try:
    alist = 5 * [s]
    last_item = alist[len(alist)]
    print('Done')
except (IndexError,NameError):
    print('Invalid index')
```

Invalid index

```
try:
    alist = 5 * [s]
    last_item = alist[len(alist)]
    print('Done')
except IndexError:
    print('Invalid index')
except NameError:
    print('Identifier not defined')
print('End of program')
```

Identifier not defined
End of program

```
try:
    alist = 5 * [s]
    last_item = alist[len(alist)]
    print('Done')
except ValueError:
    print('Incorrect Value')
print('End of program')
```

```
Traceback (most recent call last):
  Python Shell, prompt 3, line 2
builtins.NameError: name 's' is not defined
```

What is the output ?

```
def main():
    try:
        foo()
        print('After function call')
    except ZeroDivisionError:
        print('Denominator is zero')
    except:
        print('Some error')

def foo():
    print(1/0)
main()
```

Denominator is zero

The except clause without the Exception Name is a default clause. It is always specified after all except clauses have been specified.

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

Example of propagation of Exception to Caller - What is the output ?

```
def main(): # CALLER
    try:
        foo()
        print('After function call')
    except TypeError: # Exception CAUGHT IN MAIN
        print('Exception handled in main')
        print('A string object is immutable')
    except:
        print('Some error occurred')

def foo(): # CALLEE
    try:
        astring = 'hello'
        astring[0] = 'j'
    except NameError: # Exception not caught-propagated to CALLER
        print('Incorrect Name')
main()
```

Exception handled in main
A string object is immutable

Handling and Propagating by Raising Exception

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

Raising Exception To Caller -1

Exception handled in main
A string object is immutable

```
def main():
    try:
        foo()
        print('After function call')
    except TypeError: # Exception CAUGHT AND HANDLED IN CALLER
        print('Exception handled in main - A string object is immutable')
    except:
        print('Some error occurred')

def foo():
    try:
        astring = 'hello'
        astring[0] = 'j'
    except TypeError:
        raise # Using raise will raise the exception caught in the exception clause
              # It is a TypeError Exception in this case
main()
```

Using Exception Objects

- An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception.

```
try:
   some statements
except ExceptionType as instance:
   You can print value of Argument here using the args property
   of  the instance of the exception. args is a tuple
```

- This argument receives the value of the exception mostly containing the cause of the exception.

```
def prn_convert(var):
    try:
        print (int(var))
    except ValueError as inst:
        print ("The argument does not contain numbers\n", inst.args)
```

```
>>> prn_convert("CMPUT")
The argument does not contain numbers
 ("invalid literal for int() with base 10: 'CMPUT'",)
```

Raising Exception To Caller -2

```
def main():
    try:
        foo()
        print('After function call')
    except TypeError as e:    # Caught as object e
        print(e.args)   # prints a tuple – try printing e.args[0]
        print('Exception handled in main')
        print('A string object is immutable')

def foo():
    try:
        astring = 'hello'
        astring[0] = 'j'
    except TypeError:
        raise TypeError('foo cannot handle this') # propagating to caller

main()
```

```
('foo cannot handle this',)
Exception handled in main
A string object is immutable
```

Raising Exceptions

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

```
>>> try:
...         print("Raising an exception")
...         raise Exception('CMPUT', '175')
...     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', '175')
X=CMPUT Y=175
```

Python Standard Exceptions

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.

EXCEPTION NAME	DESCRIPTION
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Handling errors

- If you can solve a problem locally, use an if-statement
- If a function throws an exception, handle it
- If you cannot solve the problem locally, throw an exception (or let it propagate further) and handle it where you know what to do about it.

```
top = myStack.pop()
```

```
try:
    top = myStack.pop()
Except IndexError :
    print("Stack is empty")
```

```
if not myStack.isEmpty():
    top = myStack.pop()
else:
    print("Stack is empty")
```

```
try:
    top = myStack.pop()
Except :
    print("Stack is empty")
```

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')
except IOError :
    print ("I/O error: 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)
- It is optional – you don't have to have one.

Why use finally?

- The finally clause is 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("thank for dividing")
```

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

```
>>> divide(2, 1)
result is 2.0
thank you for dividing
>>> divide(2, 0)
division by zero!
thank you for dividing
>>> divide("CMPUT", "175")
thank you 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'
```

finally clause is executed in any event.

- no exception
- division by 0
- type error (not handled here)

Full Semantics of try-except-finally

- If an expression in the try block raises an exception:
  - the rest of the code in the try block is skipped.
  - If the exception "matches" the first except clause
    - the code in its except block is run
    - the code in the finally clause is run.
    - the rest of the except clauses are ignored.
    - the rest of the code in the method is run.
  - If the exception does not "match" the first except clause, similar actions are taken for the first "matching" except clause.

Full Semantics of try-except-finally

- If the exception does not "match" any except clause
  - the code in the finally clause is run
  - all other code in the method is abandoned
  - the exception is raised to the calling method – start looking for some other portion of code that will catch and handle the exception
- If no exception is raised at all
  - all code in the try block is executed
  - The code in the else clause is run
  - the code in the finally clause is run
  - normal code in the method following finally is run

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

Example of try-except-else-finally

```
def foo(a):
    try:
        number = int(a)
        print('Lucky Number')
        size = len(a)
        print(size)
    except ValueError:
        print('Number Error')
    else:
        print(number * number)
    finally:
        print('All done')
    print('Goodbye')
```

What is the output for foo('10')?

```
Lucky Number
2
100
All done
Goodbye
```

What is the output for foo(10)?

```
Lucky Number
All done
Traceback (most recent call last):
  Python Shell, prompt 2, line 1
  File "exceptions2.py", line 12, in <module>
    print('All done')
builtins.TypeError: object of type 'int' has no len()
```

User-defined Exception

- Python also allows you to create your own Exception subclasses by deriving classes from the standard built-in exceptions.
- This allows you to conditionally match only the Exceptions that you want to match.
- This is useful when you need to display more specific information when an exception is caught.

Example related to *RuntimeError*

- A class is created that is subclassed from *RuntimeError* to display more specific information.

```
class Networkerror(RuntimeError):
    def __init__(self, arg):
        self.args = arg
```

- Once the class is defined, the exception can be raised

```
try:
    raise Networkerror("Bad hostname")
except Networkerror as e:
    print (e.args)
```

Create Own Exceptions

- You may name your own exceptions by creating a new exception derived from the Exception class

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

The default __init__() of Exception has been overridden. The new behavior simply creates the *value* attribute replacing the default behavior of creating the *args* attribute.

```
>>>try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
...
My exception occurred, value: 4
```

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 Examples

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

print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

```
32.0
451
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print KelvinToFahrenheit(-5)
  File "test.py", line 2, in KelvinToFahrenheit
    assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

```
try:
    KelvinToFahrenheit("-23")
except AssertionError:
    print ("Incorrect  temp")
except TypeError:
    print ("Not a number")

Not a number
```

Asserting valid input

```
from types import *

class MyDB:
        ...
def addRecord(self, id, name):
    assert type(id) is IntType, "id is not an integer: %r" % id
    assert type(name) is StringType, "name is not a string: %r" % name
    ...
```