COMPSCI 105 S1 2017
Principles of Computer Science

Exceptions 2

The else clause

- Executed only if the try clause completes with no errors
  - It is useful for code that must be executed if the try clause does not raise an exception.

```
try:
    statement block here
except:
    more statements here (undo operations)
else:
    more statements here (close operations)
```

Examples

Example07.py

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
```

```
Please enter your age: 4
I see that you are 4
years old.
```

```
Please enter your age: a
Hey, that wasn't a
number!
```

Exercise 1

- What is the output of the following code fragment?

```
try:
        my_list = [1, 2, 3]
        num = int(input('Enter an index: '))
        value = my_list[num]
except IndexError:
        print("Invalid index!")
else:
        print(value)
print("DONE")
```

- Cases:
  - Enter an index: 1
  - Enter an index: 6

The Finally clause

- The finally block is optional, and is not frequently used
- Executed after the try and except blocks, but before the entire try-except ends
- Code within a finally block is **guaranteed** to be executed if any part of the associated try block is executed regardless of an exception being thrown or not
  - It allows for cleanup of actions that occurred in the try block but may remain undone if an exception is caught
  - Often used with files to close the file

```
try:
    statement block here
except:
    more statements here (undo operations)
finally:
    more statements here (close operations)
```

Example

Example08.py

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        result = 'Divided by zero'
    else:
        print("result is", result)
    finally:
        print("finally clause")
    return result
```

- Case 1:
  - No error

```
x = divide(2, 1)
print(x)
```

```
result is 2.0
finally clause
2.0
```

Example

Example08.py

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        result = 'Divided by zero'
    else:
        print("result is", result)
    finally:
        print("finally clause")
    return result
```

- Case 2:
  - Divided by zero

```
x = divide(2, 0)
print(x)
```

```
finally clause
Divided by zero
```

Example

Example08.py

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        result = 'Divided by zero'
    else:
        print("result is", result)
    finally:
        print("finally clause")
    return result
```

- Case 3:
  - Other error

```
x = divide('2', '1')
print(x)
```

```
finally clause
Traceback (most ...
TypeError: unsupported operand type(s) ...
```

Exercise 2

- What is the output of the following code fragment?

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
finally:
    print("It was really nice talking to you.  Goodbye!")
```

- Cases:
  - Please enter your age: a
  - Please enter your age: -1
  - Please enter your age: 4

FileNotFoundError & 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.
- Example:

```
input_file = open ("numbers1.txt", "r")

print ("Reading from file numbers.txt")

one_line = input_file.readline()
print(one_line)

print ("Completed reading of file input.txt")
input_file.close()
```

- FileNotFoundError: ..No such file or directory: 'numbers1.txt"

Handling With Exceptions for FileIO

- Basic structure of handling exceptions

```
try:
     Attempt something where exception error may happen
      (i.e. open a file and read the content)

except IOERROR
     React to the error

else:
     What to do if no error is encountered
    (i.e. close the file)

finally:
     Actions that must always be performed
```

Exceptions: File Example

- Consider the following code:

```
try:
       inputFileName = input("Enter name of input file: ")
       input_file = open (inputFileName, "r")
       one_line = input_file.readline()
except IOError:
      print("File", inputFileName, "could not be opened")
else:
       print(one_line)
       input_file.close()
       print ("Closed file", inputFileName)
```

```
Enter name of input file: numbers.txt
43 34

Closed file numbers.txt
```

- Case 1:
- Case 2:

```
Enter name of input file: test.txt
File test.txt could not be opened
```

Raising an exception:

- You can create an exception by using the raise statement

```
raise Error('Error message goes here')
```

  - The program stops immediately after the raise statement; and any subsequent statements are not executed.
  - It is normally used in testing and debugging purpose
- Example:

```
def checkLevel(level):
    if level < 1:
        raise ValueError('Invalid level!')
    else:
        print (level)
```

```
Traceback (most recent call last):
  ...
  raise ValueError('Invalid level!')
ValueError: Invalid level!
```

Example09.py

Handling Exceptions

- Put code that might create a runtime error is enclosed in a try block

```
def checkLevel(level):
    try:
        if level < 1:
            raise ValueError('Invalid level!')
        else:
            print (level)
        print ('This print statement will not be reached.')
    except ValueError as x:
        print ('Problem: {0}'.format(x))
```

```
Problem: Invalid level!
```

```
def checkLevel(level):
    try:
        if level < 1:
            raise ValueError('Invalid level!')
        ...
    except ValueError as x:
        pass
```

Using Exceptions

- When to use try catch blocks?
  - If you are executing statements that you know are unsafe and you want the code to continue running anyway.
- When to raise an exception?
  - When there is a problem that you can't deal with at that point in the code, and you want to "pass the buck" so the problem can be dealt with elsewhere.

Exercise 3

- Modify the following function that calculates the mean value of a list of numbers to ensure that the function generates an informative exception when input is unexpected

```
def mean(data):
    sum = 0
    for element in data:
        sum += element
    mean = sum / len(data)
    return mean
```

Summary

- Exceptions alter the flow of control
  - When an exception is raised, execution stops
  - When the exception is caught, execution starts again
- try… except blocks are used to handle problem code
  - Can ensure that code fails gracefully
  - Can ensure input is acceptable
- finally
  - Executes code after the exception handling code

Appendix

- TypeErrors are caused by combining the wrong type of objects, or calling a function with the wrong type of object.
  - This happens when someone tries to do an operation with different kinds of incompatible data types. A common example is to do addition of Integers and a string.
  - print (1 + "a")
- A ValueError is used when a function receives a value that has the right type but an invalid value
  - value = int('a')
  - value = float ('a')