Review 3

Exceptions and Try-Except Blocks
What Might You Be Asked

• Create your own Exception class
• Write code to throw an exception
• Follow the path of a thrown exception
  ▪ Requires **understanding** of try-except blocks
  ▪ Simply give us the trace (print statement) results
• Write a simple try-except code fragment
  ▪ Will only confine it to a single function/fragment
  ▪ Look at the sample code `read.py` from Lecture 21
Error Types in Python

• All errors are instances of class BaseException
• This allows us to organize them in a hierarchy
Python Error Type Hierarchy

- BaseException
  - SystemExit
  - Exception
    - AssertionError
    - AttributeError
    - ArithmeticError
    - IOError
      - Argument has wrong type (e.g. float([1]))
    - TypeError
      - Argument has wrong value (e.g. float('a'))
    - ZeroDivisionError
    - OverflowError

http://docs.python.org/library/exceptions.html

You will NOT have to memorize this on exam.
Creating Your Own Exceptions

```python
class CustomError(Exception):
    """An instance is a custom exception""
    pass
```

This is all you need

- No extra fields
- No extra methods
- No constructors

Inherit everything

Only issues is choice of parent error class. Use Exception if you are unsure what.
# When Do Exceptions Happen?

## Automatically Created

```python
def foo():
    x = 5 / 0
```

**Python creates Exception for you automatically**

## Manually Created

```python
def foo():
    raise Exception('I threw it')
```

**You create Exception manually by raising it**
Raising Errors in Python

• **Usage:** `raise <exp>`
  - `exp` evaluates to an object
  - An instance of Exception

• Tailor your error types
  - **ValueError**: Bad value
  - **TypeError**: Bad type

• **Examples:**
  - `raise ValueError('not in 0..23')`
  - `raise TypeError('not an int')`

• Only issue is the type

```python
def foo(x):
    assert x < 2, 'My error'
    ...
```

```python
def foo(x):
    if x >= 2:
        m = 'My error'
        raise AssertionError(m)
    ...
```

**Identical**
def foo():
    x = 1
    try:
        x = 2
        raise Exception()
        x = x+5
    except Exception:
        x = x+10
    return x

What does foo() evaluate to?
def foo():
    x = 1  
    try:
        x = 2  
        raise Exception()  
        x = x+5  
    except Exception:
        x = x+10  
    return x
def foo():
    x = 1
    try:
        x = 2
        raise Exception()
        x = x+5
    except Exception:
        x = x+10
    return x

What does foo() evaluate to?

Answer: 12 (2+10)
More Exception Tracing

def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')

What is the output of first(2)?
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third')

What is the output of first(2)?

'Starting first.'
'Starting second.'
'Starting third.'
'Caught at second'
'Ending second'
'Ending first'
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third')

What is the output of first(0)?
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third')

What is the output of first(0)?

'Starting first.'
'Starting second.'
'Starting third.'
'Ending third'
'Ending second'
'Ending first'
def first(x):
    print('Starting first.')
    try:
        second(x)
    except IOError:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except AssertionError:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    print('Ending third')

What is the output of first(-1)?
def first(x):
    print('Starting first.')
    try:
        second(x)
    except IOError:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except AssertionError:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    print('Ending third')

What is the output of first(-1)?

Starting first.
Starting second.
Starting third.
Caught at first.
Ending first.
def first(x):
    print('Starting first."
    try:
        second(x)
    except IOError:
        print('Caught at first"
    print('Ending first"

What is the output of first(1)?

def second(x):
    print('Starting second."
    try:
        third(x)
    except AssertionError:
        print('Caught at second"
    print('Ending second"

def third(x):
    print('Starting third."
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    print('Ending third."

def first(x):
    print('Starting first.')
    try:
        second(x)
    except IOError:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except AssertionError:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    print('Ending third')

What is the output of first(1)?

Starting first.
Starting second.
Starting third.
Caught at second.
Ending second.
Ending first.
Exceptions and Call Frames

>>> y = first(-1)

Diagram this call
def isFloat(s):
    """Returns: True if string s represents a float. False otherwise"""
    # Implement Me

    float(s) returns an error if s does not represent a float
def isFloat(s):
    """Returns: True if string s represents a float.
    False otherwise"""
    try:
        x = float(s)
        return True
    except:
        return False

Conversion to a float might fail
If attempt succeeds, string s is a float
Otherwise, it is not
def isFloat(s):
    """Returns: True if string s represents a float.
    False otherwise"""
try:
    x = float(s)
    return True
except ValueError as e:
    print(e)
    return False
Conversion to a float might fail
If attempt succeeds, string s is a float
Otherwise, it is not
def fix_bricks(args):
    """Changes constants BRICKS_IN_ROW, BRICK_ROWS, and BRICK_WIDTH to match command line arguments

    If args does not have exactly 2 elements, or they do not represent positive integers, DON'T DO ANYTHING.

    If args has exactly two elements, AND they represent positive integers:
        1. Convert the second element to an int and store it in BRICKS_IN_ROW.
        2. Convert the third element to an int and store it in BRICK_ROWS.
        3. Recompute BRICK_WIDTH formula

    Precondition: args is a list of strings.""
    pass

Examples:

>>> fix_bricks(['3', '4'])    # okay
>>> fix_bricks(['3'])         # error
>>> fix_bricks(['3','4','5']) # error
>>> fix_bricks(['a', '1'])    # error
def fix_bricks(args):
    """Change constants BRICKS_IN_ROW, BRICK_ROWS, and BRICK_WIDTH"""
    global BRICKS_IN_ROW, BRICK_ROWS
    global BRICK_WIDTH
    if len(args) != 2:
        return
    try:
        b_in_row = int(args[0])
        b_rows   = int(args[1])
        if (b_in_row <= 0 or b_rows <= 0):
            return
        BRICKS_IN_ROW = b_in_row;
        BRICK_ROWS   = b_rows;
        BRICK_WIDTH = (GAME_WIDTH - BRICK_SEP_H * (b_in_row+1)) / b_in_row
    except:
        pass
    print("Need to change global variables")
    print("Will not reach here if conversion fails")