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

```
BaseException
  __init__(self, msg)
  __str__(self)
  ...

Exception
  
assertionError
  'My error'

id4

→ means “extends” or “is an instance of”
```
Python Error Type Hierarchy

BaseException

SystemExit

Exception

AssertionError
AttributeError
ArithmeticError
IOError
TypeError
ValueError

Argument has wrong type (e.g. float([1]))
Argument has wrong value (e.g. float('a'))

ZeroDivisionError
OverflowError

You will NOT have to memorize this on exam.

http://docs.python.org/library/exceptions.html
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.
<table>
<thead>
<tr>
<th>Automatically Created</th>
<th>Manually Created</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def foo():</code></td>
<td><code>def foo():</code></td>
</tr>
<tr>
<td><code>x = 5 / 0</code></td>
<td><code>raise Exception('I threw it')</code></td>
</tr>
</tbody>
</table>

Python creates Exception for you automatically

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?
```python
def foo():
    x = 1  # executes this line normally

    try:
        x = 2  # executes this line normally
        raise Exception()

    except Exception:
        x = x+5  # never reaches this line

    x = x+10  # but does execute this line

    return x  # and executes this line
```
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)
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)?
More Exception Tracing

```python
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)?
More Exception Tracing

```python
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')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except AssertionError:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    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:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    print('Ending second')

def third(x):
    print('Starting third.')
    if x < 0:
        raise IOError()
    else:
        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.
```python
1. def first(x):
2.     try:
3.         y = second(x)
4.     except IOError:
5.         y = 1
6.     return y
7. def second(x):
8.     try:
9.         y = third(x)
10.    except AssertionError:
11.        y = 2
12.    return y
13. def third(x):
14.     if x < 0:
15.         raise IOError()
16.     elif x > 0:
17.         raise AssertionError()
18.     return 3

>>> y = first(-1)
Diagram this call
```
def first(x):
    try:
        y = second(x)
    except IOError:
        y = 1
    return y

def second(x):
    try:
        y = third(x)
    except AssertionError:
        y = 2
    return y

def third(x):
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    return 3

>>> y = first(-1)

first  3
x  -1
```python
1. def first(x):
2.     try:
3.         y = second(x)
4.     except IOError:
5.         y = 1
6.     return y
7. def second(x):
8.     try:
9.         y = third(x)
10.    except AssertionError:
11.        y = 2
12.    return y
13. def third(x):
14.     if x < 0:
15.        raise IOError()
16.     elif x > 0:
17.        raise AssertionError()
18.    return 3

>>> y = first(-1)
```
```python
1. def first(x):
2.     try:
3.         y = second(x)
4.     except IOError:
5.         y = 1
6.     return y
7. def second(x):
8.     try:
9.         y = third(x)
10.    except AssertionError:
11.         y = 2
12.    return y
13. def third(x):
14.     if x < 0:
15.         raise IOError()
16.     elif x > 0:
17.         raise AssertionError()
18.     return 3

>>> y = first(-1)
```

```none
>>> y = first(-1)
```

```
first     3
         x  -1

second    9
         x  -1

third     14
         x  -1
```
```python
def first(x):
    try:
        y = second(x)
    except IOError:
        y = 1
    return y

def second(x):
    try:
        y = third(x)
    except AssertionError:
        y = 2
    return y

def third(x):
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    return 3

>>> y = first(-1)

>>> y = first(-1)
```
Exceptions and Call Frames

```python
def first(x):
    try:
        y = second(x)
    except IOError:
        y = 1
    return y

def second(x):
    try:
        y = third(x)
    except AssertionError:
        y = 2
    return y

def third(x):
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    return 3

>>> y = first(-1)
```

When an error is created
### Exceptions and Call Frames

1. def first(x):
   2.     try:
   3.         y = second(x)
   4.     except IOError:
   5.         y = 1
   6.     return y
2. def second(x):
   3.     try:
   4.         y = third(x)
   5.     except AssertionError:
   6.         y = 2
   7.     return y
3. def third(x):
   4.     if x < 0:
   5.         raise IOError()
   6.     elif x > 0:
   7.         raise AssertionError()
   8.     return 3

```python
>>> y = first(-1)
```

#### Call Frame

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>3</td>
</tr>
<tr>
<td>second</td>
<td>None</td>
</tr>
<tr>
<td>third</td>
<td>None</td>
</tr>
</tbody>
</table>

When an error is created, it is not caught.
 Exceptions and Call Frames

1. `def first(x):
2.    try:
3.        y = second(x)
4.    except IOError:
5.        y = 1
6.    return y
7. `def second(x):
8.    try:
9.        y = third(x)
10.    except AssertionError:
11.        y = 2
12.    return y
13. `def third(x):
14.    if x < 0:
15.        raise IOError()
16.    elif x > 0:
17.        raise AssertionError()
18.    return 3

```python
>>> y = first(-1)
```

Recovers!
def first(x):
    try:
        y = second(x)
    except IOError:
        y = 1
    return y

def second(x):
    try:
        y = third(x)
    except AssertionError:
        y = 2
    return y

def third(x):
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    return 3

>>> y = first(-1)
Exceptions and Call Frames

```python
1. def first(x):
2.     try:
3.         y = second(x)
4.     except IOError:
5.         y = 1
6.     return y
7. def second(x):
8.     try:
9.         y = third(x)
10.    except AssertionError:
11.         y = 2
12.     return y
13. def third(x):
14.     if x < 0:
15.         raise IOError()
16.     elif x > 0:
17.         raise AssertionError()
18.     return 3

>>> y = first(-1)
```

```
first

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

6
```
```python
1. def first(x):
2.     try:
3.         y = second(x)
4.     except IOError:
5.         y = 1
6.     return y
7. def second(x):
8.     try:
9.         y = third(x)
10.    except AssertionError:
11.         y = 2
12.     return y
13. def third(x):
14.     if x < 0:
15.         raise IOError()
16.     elif x > 0:
17.         raise AssertionError()
18.     return 3

>>> y = first(-1)
```

The code defines three functions: `first`, `second`, and `third`. `first` calls `second` with `x` as an argument. `second` calls `third` with `x` as an argument. If `x` is less than 0, `third` raises an `IOError`. If `x` is greater than 0, `third` raises an `AssertionError`. Otherwise, `third` returns 3.

When `first(-1)` is called, it enters the `try` block of `first`, which calls `second(-1)`. `second(-1)` enters the `try` block, which calls `third(-1)`. Since `x < 0`, `third(-1)` raises an `IOError`, which is caught in the `except` block of `second(-1)`. `second(-1)` then returns 1.

The output is `1`.
```python
>>> y = first(-1)
```

```
def first(x):
    try:
        y = second(x)
    except IOError:
        y = 1
    return y

def second(x):
    try:
        y = third(x)
    except AssertionError:
        y = 2
    return y

def third(x):
    if x < 0:
        raise IOError()
    elif x > 0:
        raise AssertionError()
    return 3
```
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