

Review 3

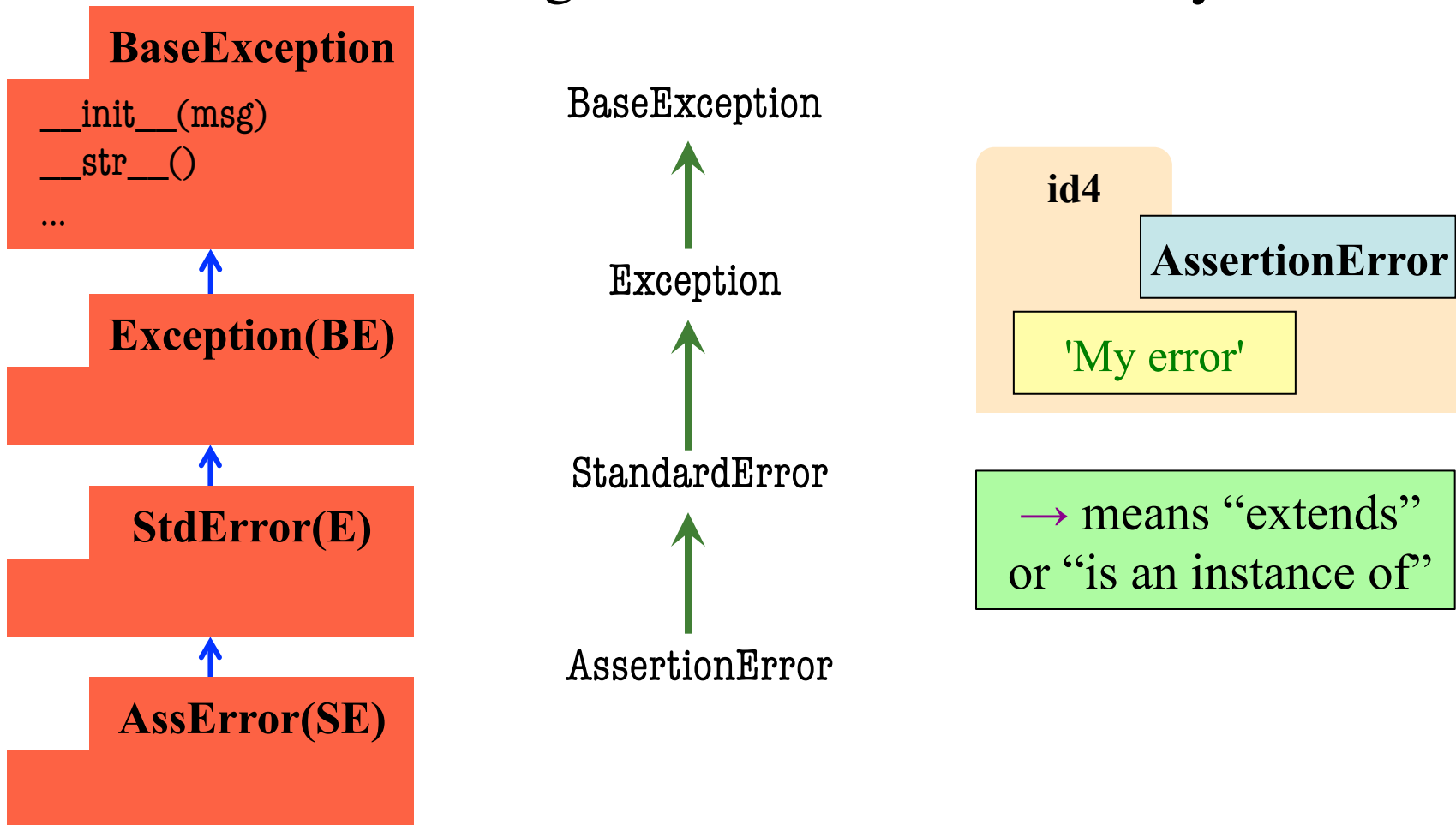
Exceptions and Try-Except Blocks

What Might You Be Asked

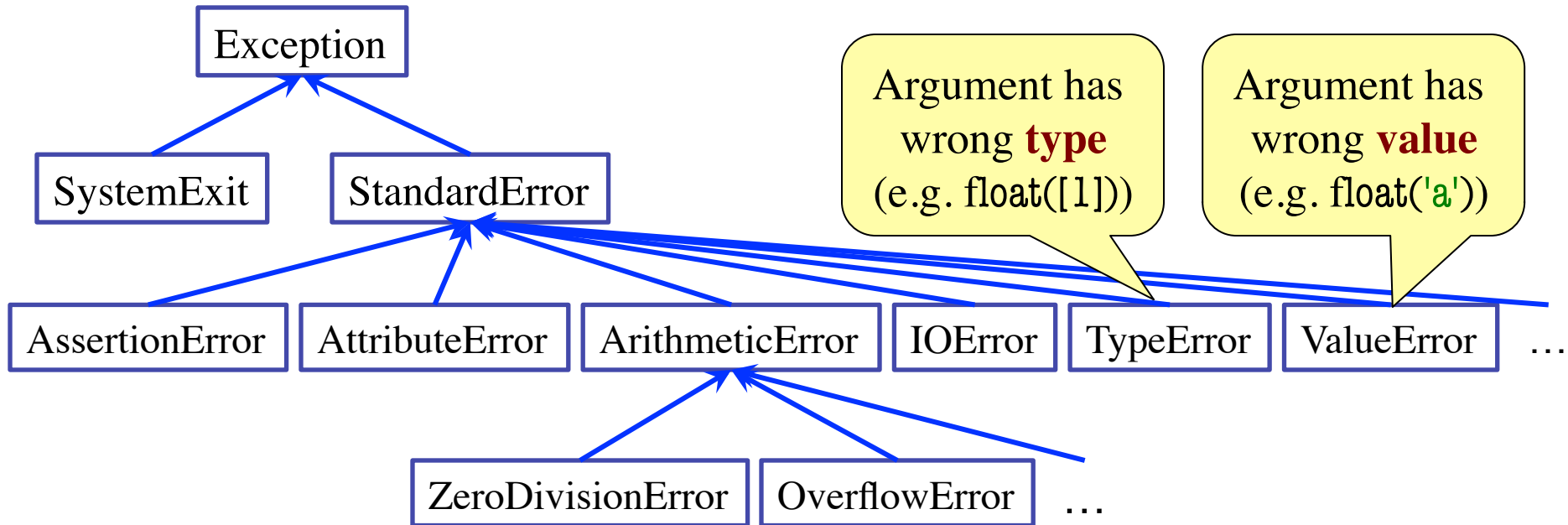
- Create your own Exception class
- Write code to throw an exception
- Follow the path of a thrown exception
 - Requires **understanding** of try-except blocks
 - Similar what you were asked in Prelim 1
 - But now with new except syntax
- Write a simple try-except code fragment
 - Look at the sample code `read.py` from Lecture 20

Error Types in Python

- All errors are instances of class `BaseException`
- This allows us to organize them in a hierarchy



Python Error Type Hierarchy



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

Why so many error types?

Creating Your Own Exceptions

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

This is all you need

- No extra fields
- No extra methods
- No constructors

Inherit everything

Only issue is choice of parent Exception class. Use StandardError if you are unsure what.

When Do Exceptions Happen?

Automatically Created

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

Python creates Exception
for you automatically

Manually Created

```
def void 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

```
def foo(x):
```

```
    assert x < 2, 'My error'
```

```
    ...
```

Identical

```
def foo(x):
```

```
    if x >= 2:
```

```
        m = 'My error'
```

```
        raise AssertionError(m)
```

```
    ...
```

Try-Except: Possible Exam Question

```
def foo():
```

```
    x = 1
```

```
    try:
```

```
        x = 2
```

```
        raise StandardError()
```

```
        x = x+5
```

```
    except StandardError:
```

```
        x = x+10
```

```
    return x
```

What does foo() evaluate to?

Try-Except: Possible Exam Question

```
def foo():
```

```
    x = 1
```



executes this line normally

```
    try:
```

```
        x = 2
```



executes this line normally

```
        raise StandardError()
```

```
        x = x+5
```



never reaches this line

```
    except StandardError:
```

```
        x = x+10
```



but does execute this line

```
    return x
```



and executes this line

Try-Catch: Possible Exam Question

```
def foo():
```

```
    x = 1
```

```
    try:
```

```
        x = 2
```

```
        raise StandardError()
```

```
        x = x+5
```

```
    except StandardError:
```

```
        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)?

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)?

```
'Starting first.'  
'Starting second.'  
'Starting third.'  
'Caught at second'  
'Ending second'  
'Ending first'
```

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(0)?

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(0)?

```
'Starting first.'  
'Starting second.'  
'Starting third.'  
'Ending third'  
'Ending second'  
'Ending first'
```

Exceptions and Dispatch-On-Type

```
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)?

Exceptions and Dispatch-On-Type

```
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.
```


Exceptions and Dispatch-On-Type

```
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)?

Exceptions and Dispatch-On-Type

```
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.
```

Programming With Try-Except

```
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

Programming With Try-Except

```
def isfloat(s):
```

```
    """Returns: True if string  
    s represents a float.
```

```
    False otherwise"""
```

```
    try:
```

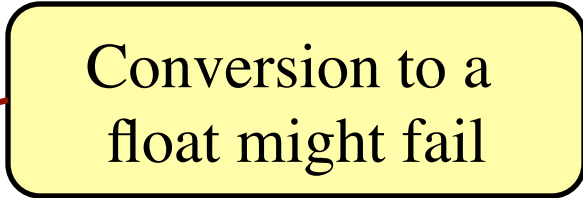
```
        x = float(s)
```

```
        return True
```

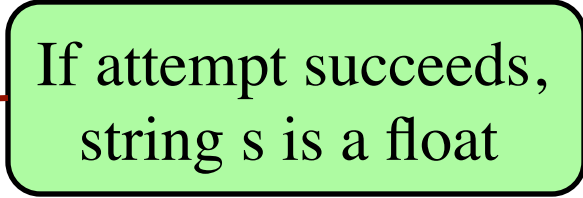
```
    except:
```

```
        return False
```

Conversion to a
float might fail



If attempt succeeds,
string s is a float



Otherwise, it is not



Example from Older Version of A7

```
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
```

Example from Older Version of A7

```
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
```

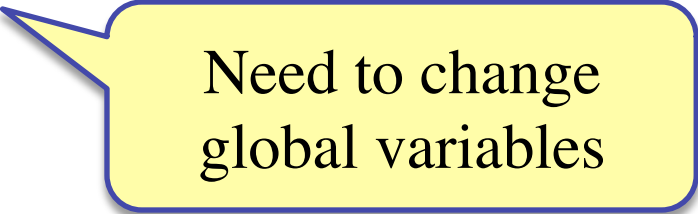
```
        BRICKS_IN_ROW = b_in_row;
```

```
        BRICK_ROWS = b_rows;
```

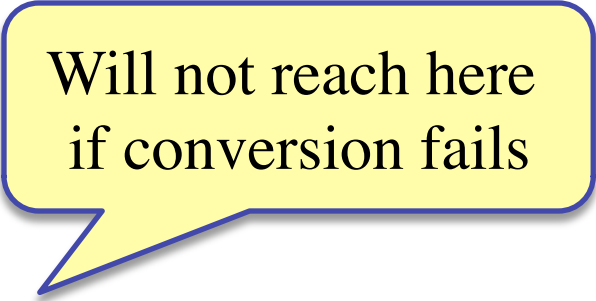
```
        BRICK_WIDTH = GAME_WIDTH / b_in_row - BRICK_SEP_H
```

```
    except:
```

```
        pass
```



Need to change
global variables



Will not reach here
if conversion fails