Module 25

Advanced Error Handling

Describe Error Types

- Error messages contain a lot of information
  - Stack trace is the complete call stack at crash
  - Final thing is the error message
  - But something right before the message…
- **Examples**
  - **ZeroDivisionError:** division by zero
  - **ValueError:** invalid literal for int() with base 10
  - **TypeError:** 'int' object is not iterable
- This value is the **error type**

Error Types in Python

```
def foo():
    assert 1 == 2, 'My error'
    …
```

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

```
>>> foo()
AssertionError: My error
```

```
>>> foo()
ZeroDivisionError: integer
 division or modulo by zero
```

Class Names

Error Types in Python

```
def foo():
    assert 1 == 2, 'My error'
    ...
```

d

Information about an error is stored inside an **object**. The error type is the **class** of the error object.

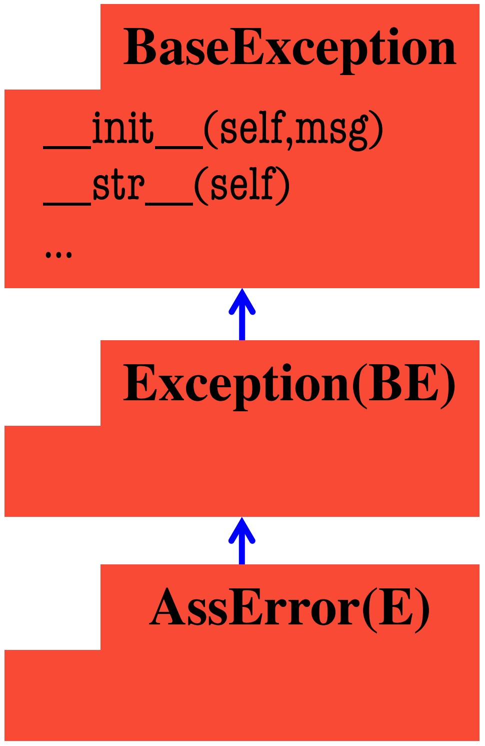
```
>>> foo()
AssertionError: My error
```

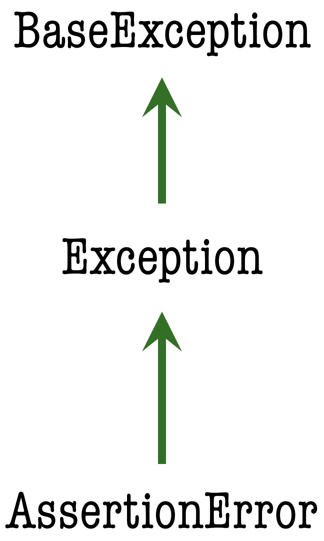
```
>>> foo()
ZeroDivisionError: integer division or modulo by zero
```

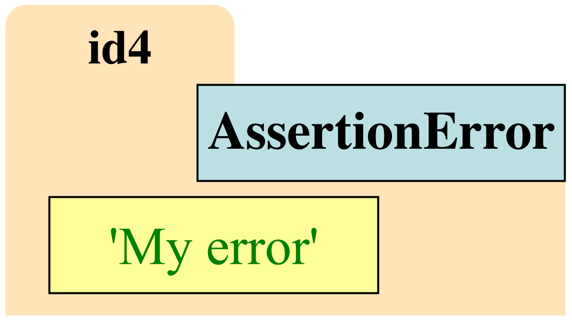
Class Names

Error Types in Python

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



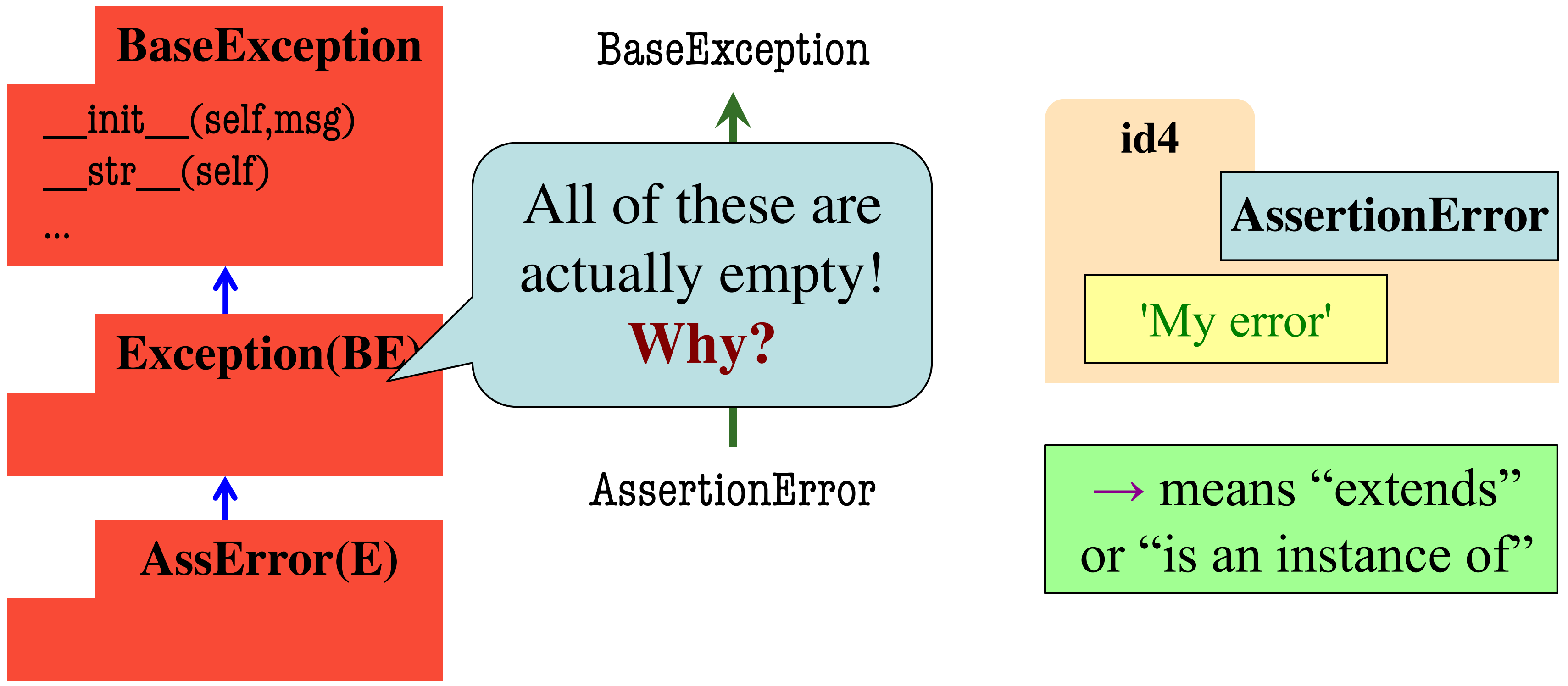




→ means "extends"
or "is an instance of"

Error Types in Python

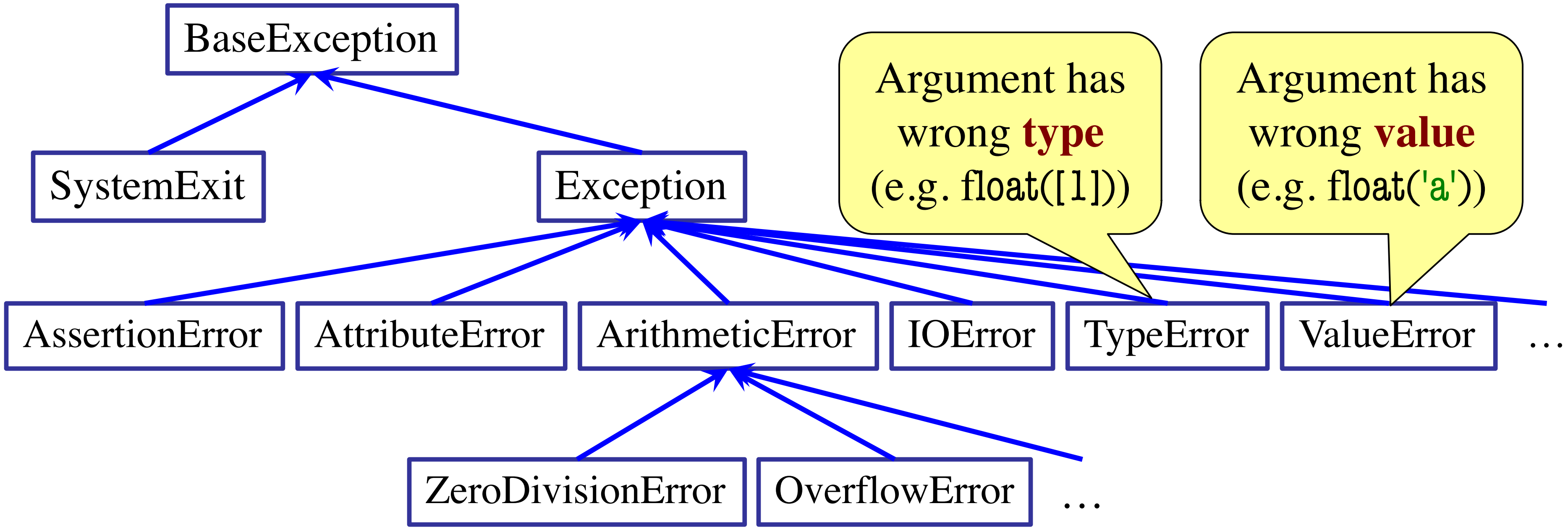
- All errors are instances of class BaseException
- This allows us to organize them in a hierarchy



All of these are actually empty! **Why?**

→ means "extends" or "is an instance of"

Python Error Type Hierarchy



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

Why so many error types?

Recall: Recovering from Errors

- try-except blocks allow us to recover from errors
  - Do the code that is in the try-block
  - Once an error occurs, jump to the catch
- **Example**:

```
try:
    val = input()         # get number from user
    x = float(val)        # convert string to float
    print('The next number is '+str(x+1))
except:
    print('Hey! That is not a number!')
```

might have an error

executes if have an error

Handling Errors by Type

- try-except blocks can be restricted to **specific** errors
  - Doe except if error is **an instance** of that type
  - If error not an instance, do not recover
- **Example**:

```
try:
    val = input()          # get number from user
    x = float(val)         # convert string to float
    print('The next number is '+str(x+1))
except ValueError:
    print('Hey! That is not a number!')
```

May have IOError

May have ValueError

Only recovers ValueError. Other errors ignored.

Handling Errors by Type

- try-except blocks can be restricted to **specific** errors
  - Doe except if error is **an instance** of that type
  - If error not an instance, do not recover
- **Example**:

```
try:
    val = input()          # get number from user
    x = float(val)         # convert string to float
    print('The next number is '+str(x+1))
except IOError:
    print('Check your keyboard!')
```

May have IOError

May have ValueError

Only recovers IOError.
Other errors ignored.

This Allows for Multiple Excepts

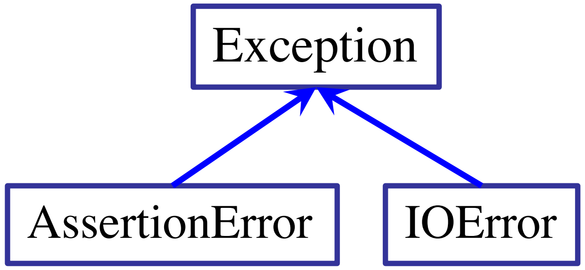
```
try:
    val = input()           # get number from user
    x = float(val)          # convert string to float
    print('The next number is '+str(x+1))
except ValueError:
    print('Hey! That is not a number!')
except IOError:
    print('Check your keyboard!')
```

This works just like elif!

Except Matches with isintance

```
try:
    val = input()          # get number from user
    x = float(val)         # convert string to float
    print('The next number is '+str(x+1))
except Exception:
    print('Something bad just happened')
```

This recovers all errors



Recall: Try-Except and the Call Stack

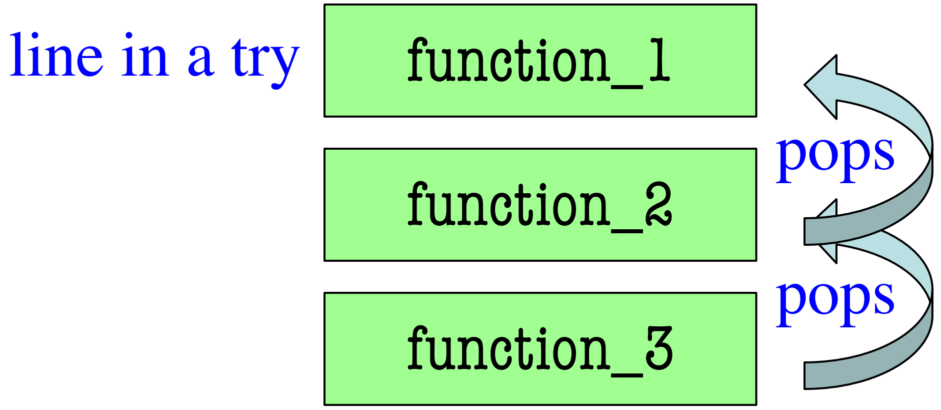
```
# recover.py

def function_1(x,y):
    try:
        return function_2(x,y)
    except:
        return float('inf')

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y # crash here
```

- Error "pops" frames off stack
  - Starts from the stack bottom
  - Continues until it sees that current line is in a try-block
  - Jumps to except, and then proceeds as if no error



Tracing Control Flow

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

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

```
def third(x):
    print('Starting third.')
    if i == 1:
        pass
    if i == 2:
        y = 5/0
    if i == 3:
        assert False, 'Intentional Error'
    print('Ending third.')
```

What is the output of first(2)?

Tracing Control Flow

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

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

```
def third(x):
    print('Starting third.’)
    if i == 1:
        pass
    if i == 2:
        y = 5/0
    if i == 3:
        assert False, 'Intentional Error’
    print('Ending third.')
```

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

Tracing Control Flow

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

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

```
def third(x):
    print('Starting third.’)
    if i == 1:
        pass
    if i == 2:
        y = 5/0
    if i == 3:
        assert False, 'Intentional Error’
    print('Ending third.')
```

What is the output of first(3)?

Tracing Control Flow

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

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

```
def third(x):
    print('Starting third.’)
    if i == 1:
        pass
    if i == 2:
        y = 5/0
    if i == 3:
        assert False, 'Intentional Error’
    print('Ending third.')
```

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

Creating Errors in Python

- Create errors with `raise`
  - **Usage**: `raise <exp>`
  - `exp` evaluates to an object
  - An instance of Exception
- Tailor your error types
  - **ValueError**: Bad value
  - **TypeError**: Bad type
- Still prefer **asserts** for preconditions, however
  - Compact and easy to read

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

Identical

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

Creating Errors in Python

- Create errors with raise
  - **Usage**: raise <exp>
  - exp evaluates to an object
  - An instance of Exception
- Tailor your error types
  - **ValueError**: Bad value
  - **TypeError**: Bad type
- Still prefer **asserts** for preconditions, however
  - Compact and easy to read

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

Identical

```
def foo(x):
    if x >= 2:
        m = 'My error'
        err = ValueError(m)
        raise err
```

Creating Your Own Exceptions

```
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.

Accessing Error Attributes

- try-except can put the error in a variable
- **Example**:

```
try:
    val = input()           # get number from user
    x = float(val)          # convert string to float
    print('The next number is '+str(x+1))
except ValueError as e:
    print(e.args[0])
    print('Hey! That is not a number!')
```

Some Error subclasses have more attributes

Repacking Errors

Error Type

```python
class CustomError(Exception):
    """A custom exception"""

    def __init__(self,value):
        """Creates error for value"""
        super().__init__('Bad value')
        self.value = value
```

Need this to set message

Repackaging

```python
try:
    val = input()
    x = float(val)
    print('Next is '+str(x+1))
except ValueError as e:
    msg = e.args[0]
    val = extract_value(msg)
    raise CustomError(val)
```

Helper

Repacking Errors

Error Type

```
class CustomError(Exception):
    """A cu
    
    def __i
        """Cr
        supe
        self.value = value
```

Need this to set message

Repackaging

```
try:
                                                      ))
    
    val = extract_value(msg)
    raise CustomError(val)
```

Helper

Repackaging

Converting from one error type to another (more useful) error type