Lecture 22

Advanced Error Handling
Announcements for This Lecture

Prelim 2

- **Prelim, Thurs. at 7:30**
  - A–L in Bailey 101
  - M–Z in Uris G01

- **Material up to Today**
  - Recursion + Loops + Classes
  - Study guide is now posted

- **Conflict with Prelim?**
  - Submit conflict in CMS

Assignments

- **A4 is now graded**
  - Mean: 91  Median: 88.3
  - Mean: 9.1 hrs  SDev: 5 hrs

- **A5 is also graded**
  - Mean: 47.7  Median: 49
  - A: 47 (82%), B: 40 (14%)

- **Keep working on A6**
  - Should be done with Task 1
A Problem with Subclasses

```python
class Fraction(object):
    """Instances are normal fractions n/d""
    # INSTANCE ATTRIBUTES
    # _numerator:  int
    # _denominator: int > 0

class FractionalLength(Fraction):
    """Instances are fractions with units ""
    # INSTANCE ATTRIBUTES same but
    # _unit: one of 'in', 'ft', 'yd'
    def __init__(self,n,d,unit):
        """Make length of given units""
        assert unit in ['in', 'ft', 'yd']
        super().__init__(n,d)
        self._unit = unit

>>> p = Fraction(1,2)
>>> q = FractionalLength(1,2,'ft')
>>> r = p*q
    # ERROR
    Python converts to

>>> r = p.__mul__(q)  # ERROR

__mul__ has precondition
    type(q) == Fraction
```

11/8/22 Advanced Error Handling 3
The `isinstance` Function

- `isinstance(<obj>,<class>)`
  - True if `<obj>`’s class is same as or a subclass of `<class>`
  - False otherwise
- **Example:**
  - `isinstance(e,Executive)` is True
  - `isinstance(e,Employee)` is True
  - `isinstance(e,object)` is True
  - `isinstance(e,str)` is False
- Generally preferable to `type`
  - Works with base types too!

11/8/22
**isinstance and Subclasses**

```python
>>> e = Employee('Bob', 2012)
>>> isinstance(e, Executive)
???
```

**A:** True  
**B:** False  
**C:** Error  
**D:** I don’t know
isinstance and Subclasses

>>> e = Employee('Bob', 2011)
>>> isinstance(e, Executive)

A: True  B: False  C: Error  D: I don’t know

→ means “extends” or “is an instance of”
class Fraction(object):
    """Instances are fractions n/d"""
    # _numerator:  int
    # _denominator: int > 0

def __mul__(self, q):
    """Returns: Product of self, q
    Makes a new Fraction; does not modify contents of self or q
    Precondition: q a Fraction"""
    assert isinstance(q, Fraction)
    top = self.numerator * q.numerator
    bot = self.denominator * q.denominator
    return Fraction(top, bot)

>>> p = Fraction(1,2)
>>> q = FractionalLength(1,2,'ft')
>>> r = p*q

Python converts to

>>> r = p.__mul__(q)  # OKAY

Can multiply so long as it has numerator, denominator
def foo():
    assert 1 == 2, 'My error'
...

>>> foo()
AssertionError: My error

def foo():
    x = 5 / 0
...

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

Class Names
Error Types in Python

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

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

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

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

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

id4
  'My error'

→ 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

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

Exception(BE)

AssError(E)
```

All of these are actually empty! Why?

```
AssertionError

id4

AssertionError

'My error'

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

BaseException

SystemExit
Exception

AssertionError
AttributeError
ArithmeticError
IOError
TypeError
ValueError

ZeroDivisionError
OverflowError

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

Why so many error types?

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

```python
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!')
```
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:**

  ```python
  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!')
  ```

  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:**

```python
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.
Handling Errors by Type

• try-except can put the error in a variable

• **Example:**

```python
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
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

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

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

**Identical**
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'
    ...
```

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

Identical
def foo():
    x = 0
    try:
        raise Exception()
        x = 2
    except Exception:
        x = 3
    return x

- The value of foo()?  

A: 0  
B: 2  
C: 3  
D: No value. It stops!  
E: I don’t know
def foo():
    x = 0
    try:
        raise Exception()
        x = 2
    except Exception:
        x = 3
    return x

• The value of foo()?

A: 0
B: 2
C: 3  Correct
D: No value. It stops!
E: I don’t know
Raising and Try-Except

def foo():
    x = 0
    try:
        raise Exception()
        x = 2
    except BaseException:
        x = 3
    return x

• The value of foo()?

A: 0
B: 2
C: 3
D: No value. It stops!
E: I don’t know
Raising and Try-Except

def foo():
    x = 0
    try:
        raise Exception()
        x = 2
    except BaseException:
        x = 3
    return x

• The value of foo()?

A: 0
B: 2
C: 3 Correct
D: No value. It stops!
E: I don’t know
Raising and Try-Except

```python
def foo():
    x = 0
    try:
        raise Exception()
        x = 2
    except AssertionError:
        x = 3
    return x
```

- The value of `foo()`?

A: 0  
B: 2  
C: 3  
D: No value. It stops!  
E: I don’t know
def foo():
    x = 0
    try:
        raise Exception()
        x = 2
    except AssertionError:
        x = 3
    return x

• The value of foo()?

A: 0
B: 2
C: 3
D: No value. Correct
E: I don’t know

Python uses isinstance to match Error types
Creating Your Own Exceptions

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

    pass
```

This is all you need!
- No extra attributes
- No extra methods
- No constructors

Inherit everything

Only issues is choice of parent error class. Use Exception if you are unsure what.
Case Study: Files

• Can read the contents of any file with `open()`
  ▪ Returns a file object with method `read()`
  ▪ Method `read()` returns contents as a string
  ▪ Remember to `close()` file when done

• There are **SO many** errors that can happen
  ▪ `FileNotFoundError`: File does not exit
  ▪ `PermissionError`: You are not allowed to read it
  ▪ Other errors possible when processing data
Recall: JSON Files

```
{
    "wind": {
        "speed": 13.0,
        "crosswind": 5.0
    },
    "sky": [
        {
            "cover": "clouds",
            "type": "broken",
            "height": 1200.0
        },
        {
            "type": "overcast",
            "height": 1800.0
        }
    ]
}
```

- Look like a nested dict
  - But read in as a string
  - You have to convert it

- Python module json
  - Function `loads()`
    Converts str -> dict
  - Function `dumps()`
    Convert dict -> str

- Conversion is sensitive
  - Stray commas crash it
def read_json(fname):
    try:
        file = open(fname)
        data = file.read()
        file.close()
        result = json.loads(data)
        return result
    except FileNotFoundError:
        print(fname + ' not found')
    except JsonDecodeError:
        print(fname + ' is invalid')
    return None

Note that we can chain excepts like an if-elif statement

Close file when done

Open file with name

Could not find file

JSON contents are not valid

If failed

Advanced Error Handling
def read_foo(fname):
    try:
        file = open(fname)
        data = file.read()
        file.close()
        result = convert(data)
    return result
except FileNotFoundError:
    print(fname + ' not found')
except MyConversionError:
    print(fname + ' is invalid')
return None

Advanced Error Handling
Aside: Pathnames

- Files obey the same rule as other modules
  - To read a file, it must be in the same folder
  - Otherwise, you must use a pathname for file
- **Relative path**: directions from current folder
  - **macOS**: `../../lec22/file.txt`
  - **Windows**: `..\..\lec22\file.txt`
- **Absolute path**: directions that work anywhere
  - **macOS**: `/Users/white/cs1110/lect22/file.txt`
  - **Windows**: `C:\Users\white\cs1110\lect22\file.txt`
Aside: Pathnames

• Files obey the same rule as other modules
  ▪ To read a file, it must be in the same folder
  ▪ Otherwise, you must use a pathname for the file

• Relative path: directions from current folder
  ▪ macOS: ' ../../lec22/file.txt'
  ▪ Windows: ' .. \ .. \lec22\file.txt'

• Absolute path: directions that work anywhere
  ▪ macOS: '/Users/white/cs1110/lect22/file.txt'
  ▪ Windows: 'C: \ Users \ white \ cs1110 \ lect22 \ file.txt'

Note the change in slash direction

Like navigating command shell
Pathnames are OS Specific

- This makes reading files harder
  - May work on Windows but crash on macOS!
  - Yet another error message we need to handle

- **Solution**: Use the module `os.path`
  - Builds a pathname string for current os

- **Example**: `os.path('..', 'cs1110', 'lec22', 'file.txt')`
  - macOS: `../cs1110/lec22/file.txt`
  - Windows: `..\cs1110\lec22\file.txt`

- Absolute paths are a little trickier, but similar
Final Word on Error Handling

- Versions of **try-except** exist in most languages
  - Java, C++, C#, Objective-C all have it
- But those languages try to **minimize** its use
  - Give application a way to crash “nicely”
  - Because processing a try-except it quite slow
- Python has a very **different** philosophy
  - Python is sort-of slow; exceptions are not slower
  - It is okay to use **try-except** all the time
  - Encourages its use as much as **if**-statements
Final Word on Error Handling

• Versions of try-except exist in most languages
  ▪ Java, C++, C#, Objective-C all have it
• But those languages try to minimize its use
  ▪ Give application a way to crash “nicely”
  ▪ Because processing a try-except it quite slow
• Python has a different philosophy
  ▪ Python is sort-of slow; exceptions are not slower
  ▪ It is okay to use try-except all the time
  ▪ Encourages its use as much as if-statements

Developers refer to coding styles unique to python as pythonic programming