Lecture 20

Classes and Types
Announcements for Today

Reading

• Today: See reading online
• Tuesday: Chapter 7

Assignments

• A4 is now graded
  - Mean: 90, Median: 94
  - CMS is skewed by auditors
  - Recursion generally good
  - Most points lost on loops

• Keep working on A5
  - Finish Part D by Sunday
  - Material in Part I on exam
  - Might adjust Part II a bit

Prelim, Nov 14th 7:30-9:30
- Material up to Today
- Review has been posted
- Recursion + Loops + Classes

Conflict with Prelim time?
- Submit to Prelim 2 Conflict assignment on CMS
- LAST DAY TO SUBMIT
class Fraction(object):
    '''Instance attributes:
    numerator: top [int]
    denominator: bottom [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 type(q) == Fraction
    top = self.numerator * q.numerator
    bot = self.denominator * q.denominator
    return Fraction(top, bot)

>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p*q

Python converts to

>>> r = p.__mul__(q)

Operator overloading uses method in object on left.
Recall: Overloading Multiplication

```python
class Fraction(object):
    """Instance attributes:
    numerator: top    [int]
    denominator: bottom [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 type(q) == Fraction
        top = self.numerator*q.numerator
        bot = self.denominator*q.denominator
        return Fraction(top,bot)
```

```python
>>> p = Fraction(1,2)
>>> q = 2 # an int
>>> r = p*q
>>> r = p.__mul__(q) # ERROR

Python converts to

```
>>> r = p.__mul__(q) # ERROR
Can only multiply fractions. But ints “make sense” too.
```
Dispatch on Type

- Types determine behavior
  - Diff types = diff behavior
  - **Example**: + (plus)
    - Addition for numbers
    - Concatenation for strings
- Can implement with ifs
  - Main method checks type
  - “Dispatches” to right helper
- **How all operators work**
  - Checks (class) type on left
  - Dispatches to that method

```python
class Fraction(object):
    ...
    def __mul__(self,q):
        """Returns: Product of self, q
        Precondition: q a Fraction or int""
        if type(q) == Fraction:
            return self._mulFrac(q)
        elif type(q) == int:
            return self._mulInt(q)
    ...
    def _mulInt(self,q): # Hidden method
        return Fraction(self.numerator*q,
                        self.denominator)
```
Dispatch on Type

- Types determine behavior
  - Diff types = diff behavior
  - **Example**: + (plus)
    - Addition for numbers
    - Concatenation for strings
- Can implement with ifs
  - Main method checks type
  - “Dispatches” to right helper
- **How all operators work**
  - Checks (class) type on left
  - Dispatches to that method

```python
class Fraction(object):
    ...
    def __mul__(self, q):
        """Returns: Product of self, q
        Precondition: q a Fraction or int""
        if type(q) == Fraction:
            return self._mulFrac(q)
        elif type(q) == int:
            return self._mulInt(q)
    ...
    def _mulInt(self, q):
        # Hidden method
        return Fraction(self.numerator * q, self.denominator)
```

Classes are main way to handle “dispatch on type” in Python. Other languages have other ways to support this (e.g. Java)
Another Problem: Subclasses

```python
class Fraction(object):
    """Instances are normal fractions n/d
    Instance attributes:
    numerator:    top [int]
    denominator: bottom [int > 0]"

class BinaryFraction(Fraction):
    """Instances are fractions k/2^n
    Instance attributes are same, BUT:
    numerator:    top [int]
    denominator: bottom [= 2^n, n ≥ 0]"

def __init__(self, k, n):
    """Make fraction k/2^n"
    assert type(n) == int and n >= 0
    Fraction.__init__(k, 2 ** n)

>>> p = Fraction(1, 2)
>>> q = BinaryFraction(1, 2)  # 1/4
>>> r = p * q

Python converts to

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

__mul__ has precondition

type(q) == Fraction
```
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!
>>> e = Employee('Bob', 2011)
>>> isinstance(e, Executive)

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

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

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

→ means “extends” or “is an instance of”
Fixing Multiplication

```python
class Fraction(object):
    """Instance attributes:
    numerator:   top    [int]
    denominator: bottom [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)
```

```python
>>> p = Fraction(1, 2)
>>> q = BinaryFraction(1, 2) # 1/4
>>> r = p * q
Python converts to

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

Can multiply so long as it has numerator, denominator
```
Error Types in Python

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

11/7/13
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
```

Class Names

11/7/13

Classes and Types
Error Types in Python

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

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

Exception(BE)

StdError(E)

AssError(SE)
```

```
BaseException
  Exception

StandardError

AssertionError

id4

AssertionError

'My error'

→ means “extends”
or “is an instance of”

Classes and Types
Error Types in Python

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

```python
BaseException
  __init__(msg)
  __str__()
...

Exception(BE)

StdError(E)

AssError(SE)
```

All of these are actually empty! Why?

→ means “extends” or “is an instance of”

Classes and Types
Python Error Type Hierarchy

```
Exception
  - SystemExit
  - StandardError
    - ArgumentError
    - AttributeError
    - ArithmeticError
    - ValueError
    - TypeError
    - IOError
    - ... 
    - 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](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:
    input = raw_input()  # get number from user
    x = float(input)     # 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
Errors and Dispatch on 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:
    input = raw_input()  # get number from user
    x = float(input)     # 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.
```

May have IOError

May have ValueError

Only recovers ValueError. Other errors ignored.
Errors and Dispatch on 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:
      input = raw_input()  # get number from user
      x = float(input)      # convert string to float
      print 'The next number is ' + str(x+1)
  except IOError:
      print 'Check your keyboard!'  
  ```
  Only recovers IOError. Other errors ignored.
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'
    ...
```

Identical

```python
def foo(x):
    if x >= 2:
        m = 'My error'
        raise AssertionError(m)
    ...
```
Raising and Try-Except

def foo():
    x = 0
    try:
        raise StandardError()
        x = 2
    except StandardError:
        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 StandardError()
    except StandardError:
        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
def foo():
    x = 0
    try:
        raise StandardError()
        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
Raising and Try-Except

```python
def foo():
    x = 0
    try:
        raise StandardError()
        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
def foo():
    x = 0
    try:
        raise StandardError()
        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 StandardError()
    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(StandardError):
    """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 Exception class. Use `StandardError` if you are unsure what.
Errors and Dispatch on Type

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

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

Some Error subclasses have more attributes
Typing Philosophy in Python

- **Duck Typing:**
  - “Type” object is determined by its methods and properties
  - Not the same as `type()` value
  - Preferred by Python experts

- Implement with `hasattr()`
  - `hasattr(<object>,<string>)`
  - Returns true if object has an attribute/method of that name

- This has many problems
  - The name tells you nothing about its specification

```python
class Fraction(object):
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"

    def __eq__(self,q):
        """Returns: True if self, q equal, False if not, or q not a Fraction"
        if type(q) != Fraction:
            return False
        left = self.numerator*q.denominator
        right = self.denominator*q.numerator
        return left == right
```

Typing Philosophy in Python

• **Duck Typing:**
  - “Type” object is determined by its methods and properties
  - Not the same as `type()` value
  - Preferred by Python experts

• Implement with `hasattr()`
  - `hasattr(<object>,<string>)`
  - Returns true if object has an attribute/method of that name

• This has many problems
  - The name tells you nothing about its specification

```python
class Fraction(object):
    """Instance attributes:
    numerator:    top       [int]
    denominator: bottom [int > 0]"

    ...

def __eq__(self,q):
    """Returns: True if self, q equal, False if not, or q not a Fraction"
    if (not (hasattr(other,'numerator') and
             hasattr(other,'denominator'))):
        return False
    left = self.numerator*q.denominator
    rght = self.denominator*q.numerator
    return left == rght
```
Typing Philosophy in Python

- **Duck Typing:**
  - “Type” object is determined by its methods and properties
  - Not the same as `type()` value
  - Preferred by Python experts
  - Implement with `hasattr`
    - `hasattr(<object>, <string>)`
    - Returns true if object has an attribute/method of that name
  - This has many problems
    - The name tells you nothing about its specification

```python
class Fraction(object):
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"

    def __eq__(self, q):
        """Returns: True if self, q equal,
        False if not, or q not a Fraction"
        if (not (hasattr(other, 'numerator')
                 and hasattr(other, 'denominator'))):
            return False
        left = self.numerator * q.denominator
        rght = self.denominator * q.numerator
        return left == rght
```

Compares **anything** with **numerator** & **denominator**

11/7/13
Typing Philosophy in Python

• **Duck Typing:**
  - “Type” object is determined by its methods and properties
  - Not the same as `type()` value
  - Preferred by Python experts
  - Implement with `hasattr()`
    - `hasattr(<object>, <string>)`
      - Returns true if object has an attribute/method of that name

• This has many problems
  - The name tells you nothing about its specification

11/7/13

How to properly implement/use typing is a major debate in language design

• What we really care about is **specifications** (and **invariants**)

• Types are a “shorthand” for this

Different typing styles trade ease-of-use with overall program robustness/safety

```
class Fraction(object):
    """Instance attributes:"
    numerator:    top       [int]
    denominator: bottom [int > 0]"

    def __eq__(self, q):
        """Returns: True if self, q equal, False if not, or q not a Fraction"
        if (not (hasattr(other, 'numerator')) and
            hasattr(other, 'denominator')):
            return False

        left = self.numerator * q.denominator
        right = self.denominator * q.numerator
        return left == right
```
Typing Philosophy in Python

- **Duck Typing:**
  - “Type” object is determined by its methods and properties
  - Not the same as `type()` value
  - Preferred by Python experts
  - Implement with `hasattr()`
    - `hasattr(<object>, <string>)`
    - Returns true if object has an attribute/method of that name
  - This has many problems
    - The name tells you nothing about its specification

How to properly implement/use typing is a major debate in language design

- What we really care about is **specifications** (and **invariants**)
- Types are a “shorthand” for this

Different typing styles trade ease-of-use with overall program robustness/safety