Lecture 21

Programming with Subclasses
Announcements for Today

### Reading
- Today: See reading online
- Tuesday: Chapter 7

### Assignments
- A4 is still being graded
  - Will be done tomorrow
- But I looked at surveys
  - People generally liked it
  - **Avg Time:** 8.8 hrs
  - **Median:** 8, **STDev:** 4.6
- A5 is due tonight at midnight
- Continue working on A6
  - Finish Task 3 by Sunday

**Prelim, Nov 9th 7:30-9:00**
- Material up to **Today**
- Review has been posted
- Recursion + Loops + Classes

**S/U Students are exempt**

**Conflict with Prelim time?**
- **LAST DAY TO SUBMIT**
About `super()`

- `super()` is very limited
  - Can only go one level
  - **BAD**: `super().super()`
- Need arguments for more
  - `super(class, self)`

The subclass

Object in the method
About `super()`

- `super()` is very limited
  - Can only go one level
  - **BAD**: `super().super()`

- Need arguments for more
  - `super(class, self)`

The subclass

Object in the method

p

id2

Rect

p.__str__()
About `super()`

- `super()` is very limited
  - Can only go one level
  - **BAD**: `super().super()`

- Need arguments for more
  - `super(class,self)`

The subclass

Object in the method

```
Rect __str__()
Poly __str__()
Shape __str__()
p.__str__()
super().__str__()
```
About `super()`

- `super()` is very limited
  - Can only go one level
  - **BAD**: `super().super()`

- Need arguments for more
  - `super(class,self)`

11/2/17

Programming with Subclasses

The subclass

Object in the method

```
p.id2
```

```
Rect
  __str__()

Poly
  __str__()

Shape
  __str__()
```
A Problem with Subclasses

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
    super().__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!
isinstance and Subclasses

>>> 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)
```
Fixing Multiplication

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

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

>>> foo()
AssertionError: My error

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

Class Names

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

AssertionError(SE)
```

```
BaseException
| 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

<table>
<thead>
<tr>
<th>BaseException</th>
<th>Exception (BE)</th>
<th>AssError (SE)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(msg)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>__str__()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All of these are actually empty! Why?

```
BaseException
AssertionError

id4
AssertionError
'My error'

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

BaseException

SystemExit

Exception

Argument has wrong **type** (e.g. float([1]))

Argument has wrong **value** (e.g. float('a'))

AssertionError

AttributeError

ArithmeticError

IOError

TypeError

ValueError

ZeroDivisionError

OverflowError

...
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!')
```

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
  - Does 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!')
```

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'
        err = AssertionError(m)
        raise err
```

11/2/17
Programming with Subclasses 20
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 = TypeError(m)
        raise err
```
Raising and Try-Except

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()
    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 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
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()`?

Options:

- 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 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 fields
- No extra methods
- No constructors

Inherit everything

Only issues is choice of parent error class. Use Exception if you are unsure what.
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
Accessing Attributes with Strings

- `hasattr(<obj>,<name>)`
  - Checks if attribute exists
- `getattr(<obj>,<name>)`
  - Reads contents of attribute
- `delattr(<obj>,<name>)`
  - Deletes the given attribute
- `setattr(<obj>,<name>,<val>)`
  - Sets the attribute value
- `<obj>.__dict__`
  - List all attributes of object
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 [int]: top
    denominator [int > 0]: bottom"
    ...

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
    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 [int]: top
denominator [int > 0]: bottom"

... 

def __eq__(self,q):
    """Returns: True if self, q equal,
False if not, or q not a Fraction"
    if (not (hasattr(q,'numerator') and
             hasattr(q,'denomenator'))):
        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 [int]: top
denominator [int > 0]: bottom"

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

Compares **anything** with `numerator` & `denominator`
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`
    
    ```python
    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

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

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

11/2/17
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 Employee(object):
    """An Employee with a salary""
    ...
    def __eq__(self, other):
        if (not (hasattr(other,'name')
                 and
                 hasattr(other,'start')
                 and
                 hasattr(other,'salary')))
            return False
        return (self.name == other.name
                and
                self.start == other.start
                and
                self.salary == other.salary)
```