Python for Astronomers

Inheritance

Operator overloading

Exercise: Iterated Prisoner's dilemma

In the iterated Prisoner's dilemma, the police vist the prisoners repeatedly. The prisoners know the previous testimonies of both themselves and the other prisoner, and adjust their reaction to that.

Modify the existing classes in the following form: The police visits each pair of prisoners nRepeat times. When visiting a prisoner, the police let him know what the other prisoner did when visited last time. The visited prisoner adjusts his probability to testify in the following way: When the other prisoner testified, his (the visited prisoner's) probability is increased by 5%; when the other prisoner did not testify, the probability is decreased by 5%. The new probability is remembered and increased or decreased during the next visit.

Of course, for a new pair of prisoners the initial probabilities must be at their original value.

Keep the modifications of the code at a minimum!

Classes and Instances Summary

Class	Instance
Type	Object with identity, state (data members) and behaviour (methods)
Formalized concept	Concrete representation of concept

Inheritance example

```
>>> # In Polynomial1.py:
>>> class Polynomial:
...   def __init__(self, coeffs):
...     self._coeffs = coeffs

...   def eval(self, x):
...     val = self._coeffs[-1]
...     for i in range(2, len(self._coeffs) + 1):
...       val = val * x + self._coeffs[-i]
...     return val

...   def info(self):
...     return "coefficients: " + str(self._coeffs)
```

Inheritance example

```
>>> class Legendre(Polynomial):
...   def __init__(self, n):
...     coeffs = {0: [1.], 1: [0., 1.],
...               2: [1./2.,0.,3./2.]}
...     self._n = n
...     Polynomial.__init__(self, coeffs[n])

...   def info(self):
...     return "degree: %d; coefficients: %s" \
...           % (self._n, str(self._coeffs))

...   def getDegree(self):
...     return self._n
```

Inheritance example

```
>>> import Polynomial1
>>> # Experiments with Polynomial instance:

>>> p = Polynomial1.Polynomial([1, 2, 3])

>>> p.info()            # Polynomial.info
>>> p.eval(-2.)         # Polynomial.eval
>>> p._coeffs           # Polynomial._coeffs

>>> # Methods and data members of subclass:
>>> p.getDegree()       # Not available
>>> p._n                # Not available
```

Inheritance example

```
>>> # Experiments with Legendre instance:

>>> l = Polynomial1.Legendre(2)

>>> l.info()            # Legendre.info
>>> l.getDegree()       # Legendre.getDegree
>>> l._n                # Legendre._n

>>> # Methods and data members of superclass:
>>> l.eval(-2.)         # Polynomial.eval
>>> l._coeffs           # Polynomial._coeffs

>>> # Check also with l.<TAB>
```

Inheritance example

```
>>> class Legendre(Polynomial):
...     # Legendre is subclass of Polynomial

...     def __init__(self, n):
...       # Constructor extends constructor of Polynomial:
...       coeffs = {0: [1.], 1: [0., 1.],
...                 2: [1./2.,0.,3./2.]}
...       # Data member specific to Legendre:
...       self._n = n
...       # Class constructor of superclass Polynomial:
...       Polynomial.__init__(self, coeffs[n])
```

Inheritance example

```
...     # Method eval not supplied, request transferred to
...     # Polynomial.eval instead

...     def info(self):
...       # Replaces Polynomial.info for Legendre objects
...       return "degree: %d; coefficients: %s" \
...              % (self._n, str(self._coeffs))
...       #   from Legendre,      from Polynomial

...     def getDegree(self):
...       # New method, specific for Legendre, added
...       return self._n
```

Inheritance – Key ideas

- Subclasses can be derived from existing classes with `class Sub(Super):`
- Subclasses inherit data members and methods from all their superclasses.
- Subclasses can add new data members and methods, and can modify existing ones.
- Each `object.attribute` reference invokes a search up the inheritance tree.
- This is also true for references from within methods with `self.attribute`
- Inside a method, methods of superclasses can be accessed with `Superclass.method(self, ...)`

Making use of inheritance

```
>>> # Somewhere deep, deep in your program:
>>> def usePolynomial(p):
...     # Works for any Polynomial:
...     print "Polynomial info: ", p.info()
...     print "Value is: ", p.eval(42)

>>> p = Polynomial1.Polynomial([1,2,3])
>>> usePolynomial(p)        # ok

>>> l = Polynomial1.Legendre(2)
>>> usePolynomial(l)        # also ok!
```

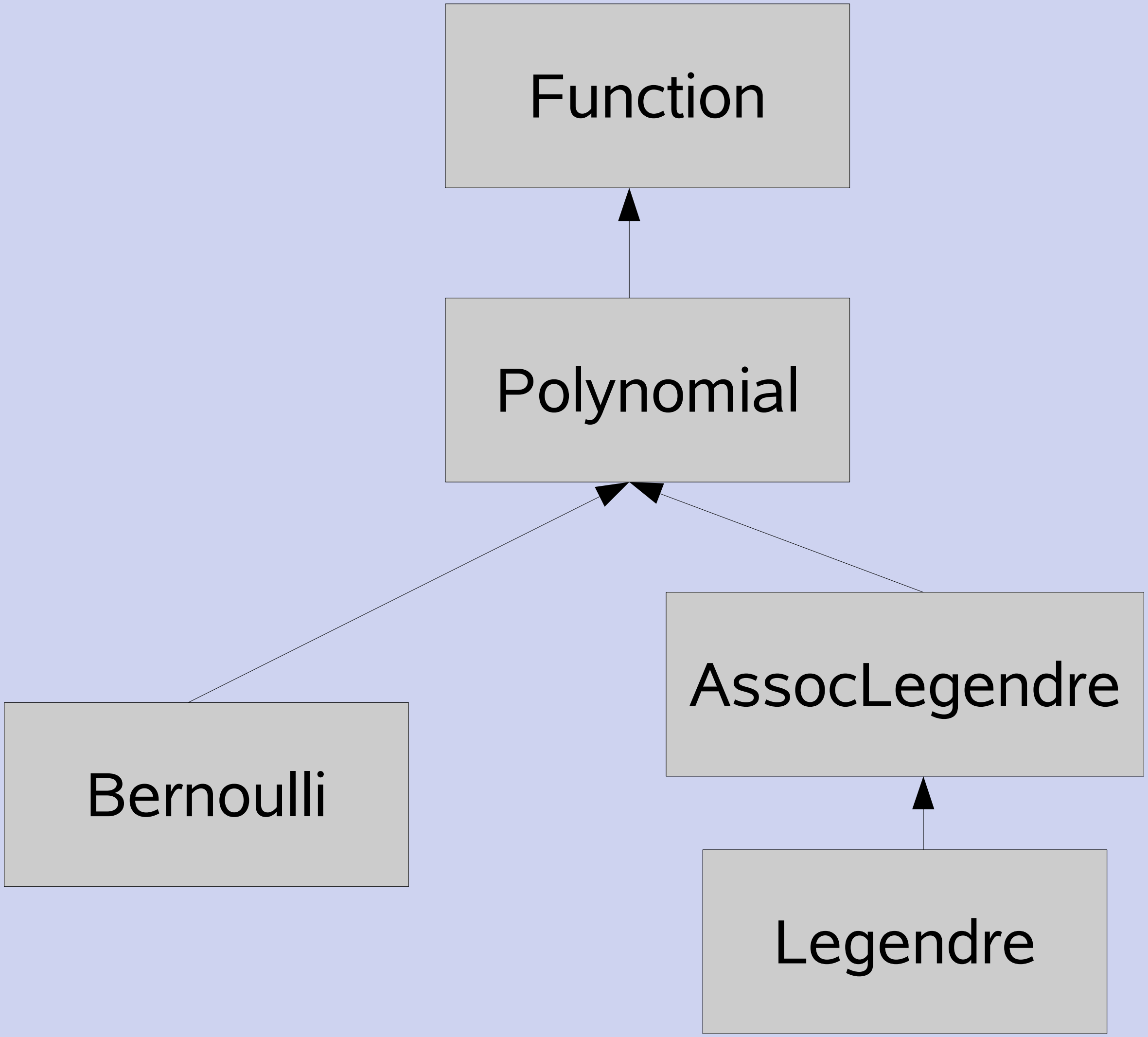
Making use of inheritance

```
>>> # usePolynomial will also work for instances
>>> # of new subclasses of Polynomial:

>>> class Bernoulli(Polynomial):
...     .....

>>> b = Polynomial1.Bernoulli(2)
>>> usePolynomial(b)        # ok
```

Liskov Substitution Principle:

It must always be possible to use an instance of a subclass in place of an instance of its superclass.

Inheritance trees



Abstraction - Generalization

- Abstraction / Generalization is another fundamental principle of OOP
- The related syntactical mechanism is inheritance, that allows subclassing existing classes
- Superclasses implement more general concepts than superclasses
- Python supports generalization fully
- Python does not support abstraction on a syntactical level (no abstract classes)

Operator overloading

```
>>> # In Polynomial2.py:
>>> class Polynomial:
...   def __init__(self, coeffs):
...     self._coeffs = coeffs

...   def __call__(self, x):
...     # Replaces eval
...     val = self._coeffs[-1]
...     for i in range(2, len(self._coeffs) + 1):
...       val = val * x + self._coeffs[-i]
...     return val
```

Operator overloading

```
...   def __str__(self):
...     # Replaces info
...     return "coefficients: " + str(self._coeffs)

...   def __len__(self):
...     """Returns length of Polynomial"""
...     return len(self._coeffs)

...   def __add__(self, p):
...     """Adds Polynomial p to self"""
...     ...
```

Operator overloading

```
>>> import Polynomial2
>>> p1 = Polynomial2.Polynomial([1, 2])
>>> p2 = Polynomial2.Polynomial([1, 2, 3])

>>> p1(2)                      # p1.__call__(2)
>>> len(p1)                    # p1.__len__()
>>> print p1                   # p1.__str__()
>>> p = p1 + p2                # p1.__add__(p2)
>>> print p
>>> p += p1                    # p = p.__add__(p1)
```

Some operator overloading methods

__init__	Class()
__del__	del instance
__add__	x + y, x += y
__radd__	1 + x
__sub__	x – y, x -= y
__mul__	x * y, x *= y
...	
__eq__	x == y
__lt__	x < y
...	

__and__	x & y
__or__	x | y
...	
__call__	x()
__getitem__	x[i]
__setitem__	x[i] = value
...	
__str__	str(x)
__repr__	str(x)

http://docs.python.org/reference/datamodel.html#special-method-names

Things left out

```
>>> # More advanced topics related with classes:
>>> # Static methods
>>> # Old-style and new-style classes
```

Things left out

```
>>> # while and for loops have optional else block
>>> # that are executed if loop is not left with
>>> # break statement:
>>> y = 43
>>> x = y / 2
>>> while x > 1:
...   if y % x == 0:
...     print y, 'has factor', x
...     break
...   x -= 1
... else:
...   print y, 'is prime'
```

Things left out

```
>>> # The continue statement continues with the next
>>> # iteration of a loop

>>> # The pass statement does nothing:
>>> def doNothing():
...     # Something has to be here:
...     pass
```

Things left out

```
>>> # List comprehension is an alternative way
>>> # to write (short) loops:

>>> l = [x**2 for x in range(10)]

>>> # is equivalent to:

>>> l = []
>>> for x in range(10):
...     l.append(x**2)
```

Things left out

```
>>> # The zip function allows you e.g. to visit
>>> # multiple sequences in a single loop:

>>> l = range(5)
>>> s = "abc"

>>> for i, j in zip(l, s):
...   print i, j
```

Things left out

```
>>> # The datatype set provides an unordered
>>> # sequence of unique elements together with
>>> # related methods

>>> s1 = set([1,2,3,2])
>>> len(s1)
>>> s2 = set([2,4,6])
>>> s1.intersection(s2)
```

Things left out

```
>>> # Everything about functional programming,
>>> # including anonymous functions, filters etc:
>>> # lambda, apply, map, filter, reduce
```

Things left out

```
> # Python's options

> python -c "print 'Hello world'"
```