Object Oriented Programming

1. Object Oriented Programming
   - objects, classes, and hierarchies
2. Evaluating Boolean Expressions
   - the class Boolean
   - callable objects
   - arguments of different types
3. Inheritance
   - the class Vector inherits from array
   - operator overloading

MCS 275 Lecture 4
Programming Tools and File Management
Jan Verschelde, 20 January 2016

Object Oriented Programming

1. Object Oriented Programming
   - objects, classes, and hierarchies
2. Evaluating Boolean Expressions
   - the class Boolean
   - callable objects
   - arguments of different types
3. Inheritance
   - the class Vector inherits from array
   - operator overloading

Object Oriented Programming – a definition

Object oriented programming is a method of implementation in which

1. programs are organized as cooperative collections of ***objects***,
2. each of which represents an instance of some ***class***,
3. and whose classes are all members of a ***hierarchy*** of classes united via inheritance relationships.

Objects — not algorithms — are the building blocks.

Applications:

- development of larger programs; and
- graphical user interfaces.

Object Oriented Programming

1. Object Oriented Programming
   - objects, classes, and hierarchies

2. Evaluating Boolean Expressions
   - the class Boolean
   - callable objects
   - arguments of different types

3. Inheritance
   - the class Vector inherits from array
   - operator overloading

the class Boolean

A Boolean expression is defined by

- a string with a valid logical expression; and
- a list of strings: the variables in the expression.

The script `boolean.py` exports the class `Boolean`.

```
>>> from boolean import Boolean
>>> bxp = Boolean('(x or y) and not z', ['x', 'y', 'z'])
```

The variable `bxp` refers to an instance of the class `Boolean`.

The data attributes are `form` and `vars`, accessible as

```
>>> bxp.form
'(x or y) and not z'
>>> bxp.vars
['x', 'y', 'z']
```

defining a class

The data attributes in a class are defined by the constructor, overriding the `__init__()` method.

```
class Boolean(object):
    """
    Defines Boolean expressions as callable objects.
    """
    def __init__(self, formula, variables):
        """
        Stores a formula as a string and a list of
        variables that appear in the formula.
        """
        self.form = formula
        self.vars = variables
```

The `self` refers to the instance, to `bxp` in the example:

```
>>> bxp = Boolean('(x or y) and not z', ['x', 'y', 'z'])
```

the string representation

The method `__str__()` defines the string representation of an object.
The string representation defines the outcome of `print`:

```
>>> print(bxp)
The variables in '(x or y) and not z' are ['x', 'y', 'z'].
```

If we want to obtain the string that represents the object,
then we apply the method `__str__()` to the object.

```
>>> str(bxp)
"The variables in '(x or y) and not z' are ['x', 'y', 'z']."
```

The representation of an object is defined by `__repr__()`
which for our example equals the string representation:

```
>>> bxp
The variables in '(x or y) and not z' are ['x', 'y', 'z'].
```

the methods `__str__()` and `__repr__()`

```
def __str__(self):
    """
    Returns the string representation of the expression.
    """
    result = 'The variables in \'' + self.form + '\' are '
    result = result + str(self.vars) + '.'
    return result

def __repr__(self):
    """
    Returns the representation of the expression.
    """
    return str(self)
```

Object Oriented Programming

1. Object Oriented Programming
   - objects, classes, and hierarchies
2. Evaluating Boolean Expressions
   - the class Boolean
   - callable objects
   - arguments of different types
3. Inheritance
   - the class Vector inherits from array
   - operator overloading

callable objects

An instance such as `bxp` of the class `Boolean` is *callable* if we can evaluate the object, for example:

```
>>> bxp(6)
1
>>> bxp([1, 1, 0])
1
```

The argument `6` is short for `[1, 1, 0]`, as $6_{10} = 110_2$.

The list of bits provides values for the variables:

```
>>> (x, y, z) = [1, 1, 0]
>>> (x, y, z)
(1, 1, 0)
```

Then we can evaluate a Boolean expression with `eval`:

```
>>> e = '(x or y) and not z'
>>> eval(e)
True
```

the built-in function `locals()`

Problem: we do not know the names of the variables.

If `var` is a string, then we cannot assign directly to the variable with name defined by the string `var`.

The function `locals()` returns a dictionary

- the keys are the names of the local variables; and
- the values are the corresponding variables.

Consider:

```
>>> var = 'x'
```

After `var = 'x'`, the dictionary `locals()` contains `var: 'x'`.

```
>>> locals()
{ ... , 'var': 'x', ... }
```

indirect assignments

To make an assignment to `'x'`, we can now do

```
>>> locals()[var] = 1
>>> x
1
>>> locals()[var]
1
>>> var
'x'
```

The variable `var` still refers to `'x'`.

```
>>> locals()[var] = 0
>>> x
0
>>> locals()[var]
0
>>> var
'x'
```

defining the evaluation

An object is callable if it is an instance of a class with a defined `__call__()` method.

```
def __call__(self, values):
    """
    Evaluates the expression at the list of values.
    The length of the list must equal the number
    of variables.
    """
    for k in range(len(values)):
        var = self.vars[k]
        locals()[var] = values[k]
    result = eval(self.form)
    return int(result)
```

In the loop, `var` refers to a name of a variable which occurs in the Boolean expression, as stored as a data attribute, in the list `vars`.

Object Oriented Programming

using the `isinstance()`

Recall that we want to evaluate `bxp` in two ways:

```
>>> bxp(6)
1
>>> bxp([1, 1, 0])
1
```

We can use the `isinstance()`

```
>>> L = [1, 1, 0]
>>> isinstance(L, list)
True
>>> n = 6
>>> isinstance(n, list)
False
```

extending the evaluation

```
def __call__(self, values):
    """
    Evaluates the expression at the list of values.
    The length of the list must equal the number
    of variables.
    """
    if isinstance(values, list):
        ...
    elif isinstance(values, int):
        ...
    else:
        print('argument must be list or number')
        return None
    result = eval(self.form)
    return int(result)
```

running the main test

```
$ python boolean.py
Give an expression : (x or y) and not z
Give names of variables : x y z
The variables in '(x or y) and not z' are ['x', 'y', 'z'].
The truth table of '(x or y) and not z' is
[0, 0, 0] 0
[0, 0, 1] 0
[0, 1, 0] 1
[0, 1, 1] 0
[1, 0, 0] 1
[1, 0, 1] 0
[1, 1, 0] 1
[1, 1, 1] 0
$
```

application: print the truth table

```
def main():
    """
    Prints the truth table of a boolean expression.
    """
    exstr = input('Give an expression : ')
    names = input('Give names of variables : ')
    nvars = names.split(' ')
    bxp = Boolean(exstr, nvars)
    print(bxp)
    print('The truth table of \'%s\' is' % bxp.form)
    for k in range(2**len(nvars)):
        print(bxp.bits(k), bxp(k))

if __name__ == "__main__":
    main()
```

Object Oriented Programming

1. Object Oriented Programming
   - objects, classes, and hierarchies

2. Evaluating Boolean Expressions
   - the class Boolean
   - callable objects
   - arguments of different types

3. Inheritance
   - the class Vector inherits from array
   - operator overloading

Inheritance

base classes and derived classes

We can create new classes from existing classes.
These new classes are *derived* from *base* classes.

The derived class *inherits* the attributes of the base class and usually contains additional attributes.
Inheritance is a powerful mechanism to reuse software.
To control complexity, we add extra features later.

We distinguish between single and multiple inheritance:

- **single** a derived class inherits from only *one* class;
- **multiple** derivation from *multiple* different classes.

Multiple inheritance may lead to name clashes,
in case the parents have methods with same name.

the Vector class inherits from array

A vector is

- an array of numbers, the data;
- with operations, for example: addition.

Inheriting from the class `array`, the vector class

- comes equipped with an efficient representation of a sequence of basic numerical values;
- operations like indexing are inherited, so we can do `a = v[k]` and `v[k] = a` for any vector `v` and value `a` of the same type of entries as `v`;
- we can wrap the complicated `buffer_info()`, to obtain the size of a vector.

the class Vector

```
from array import array as Array

class Vector(Array):
    """
    Defines a Vector of numbers,
    using an object from the class array.
    """
    def __init__(self, datatype, *data):
        """
        The datatype is a one letter string,
        of one of the characters supported
        by the class array.
        The optional data can be a list of
        elements to initialize the vector with.
        """
        Array.__init__(datatype, data)
```

wrapping or encapsulation

Recall the `buffer_info()` method on an array object.

```
def dimension(self):
    """
    Returns the dimension of the vector,
    wrapping the complicated buffer_info().
    """
    return self.buffer_info()[1]
```

Because the Vector class inherits from array, we can also apply the `buffer_info` on an object instantiated from Vector.

The new `dimension()` method hides the complicated indexing of `buffer_info()`.

the string representation

```
def __str__(self):
    """
    Returns the string representation.
    """
    dim = self.dimension()
    result = '['
    for k in range(dim):
        result += '%3d' % self[k]
        if k < dim-1:
            result += ', '
    result += ']'
    return result
```

Observe the use of the `dimension()` method and the indexing `self[k]`.

Object Oriented Programming

operator overloading

```
def __add__(self, other):
    """
    Defines the addition of two vectors,
    both must be of the same dimension,
    and must be of the same data type.
    """
    dim = self.dimension()
    result = Vector(self.typecode)
    for k in range(dim):
        result.append(self[k] + other[k])
    return result
```

comparison operators and methods

comparison operation	operator	method
equality	`==`	`__eq__`
inequality	`!=`	`__ne__`
less than	`<`	`__lt__`
greater than	`>`	`__gt__`
less or equal	`<=`	`__le__`
greater or equal	`>=`	`__ge__`

arithmetical operators and methods

arithmetical operation	operator	method
negation	-	`__neg__`
addition	+	`__add__`
inplace addition	+=	`__iadd__`
reflected addition	+	`__radd__`
subtraction	-	`__sub__`
inplace subtraction	-=	`__isub__`
reflected subtraction	-=	`__rsub__`
multiplication	*	`__mul__`
inplace multiplication	*=	`__imul__`
reflected multiplication	*	`__rmul__`
division	/	`__div__`
inplace division	/=	`__idiv__`
reflected division	/=	`__rdiv__`
invert	~	`__invert__`
power	**	`__pow__`

testing the constructor

```
def random_vector(dim):
    """
    Returns a vector of the given dimension dim,
    with 2-digit random integers.
    """
    from random import randint
    items = [randint(10, 99) for _ in range(dim)]
    return Vector('l', items)
```

testing the indexing

```
def test_indexing(vec):
    """
    Interactive test on the indexing,
    for a given vector vec.
    Because Vector inherits from array,
    we inherits the indexing operator,
    both for selecting and assigning.
    """
    dim = vec.dimension()
    idx = int(input('Give an index < %d :' % dim))
    val = int(input('Give a value : '))
    vec[idx] = val
    print('vec[%d] : %d' % (idx, val))
    print('the vector :', vec)
```

testing the addition

```
def test_addition(dim):
    """
    Tests the addition of two random vectors,
    of the given dimension dim.
    """
    aaa = random_vector(dim)
    bbb = random_vector(dim)
    ccc = aaa + bbb
    print('  A =', aaa)
    print('  B =', bbb)
    print('A+B =', ccc)
```

the main program

```
def main():
    """
    Prompts the user for the dimension
    and tests the operations in Vector.
    """
    dim = int(input("Give the number of elements : "))
    vec = random_vector(dim)
    print('dimension :', vec.dimension())
    print('typecode :', vec.typecode)
    print('the vector :', vec)
    test_indexing(vec)
    test_addition(dim)

if __name__ == "__main__":
    main()
```

exercises

1. Instead of an explicit loop to compute the binary expansion of a positive integer number, consider `bin()`. Write a function that uses a number as input argument and that returns a list of zeros and ones, which represents the binary expansion of the input argument. Apply `bin()` and list comprehensions.
2. The multiplication of two vectors x and y is defined by the inner product: $x \star y = x_0 y_0 + x_1 y_1 + \cdots + x_{n-1} y_{n-1}$. Add the inner product operation to the vector class, overriding `__mul__()`.
3. Inheriting from the class array, define a class `Poly` to represent a polynomial in one variable. The array stores the coefficients of the polynomial. Write the constructor for `Poly`.
4. Extend the previous exercise so any object `p` of the class `Poly` is callable, that is: for a number `z`, the expression `p(z)` returns the value of the polynomial `p` evaluated at `z`.
5. Describe the design of a class `Matrix`, which stores a matrix as a list of arrays. Write the constructor for this class `Matrix`.