4. Object Oriented Programming

4.1. State

Suppose we want to model a bank account with support for `deposit` and `withdraw` operations. One way to do that is by using global state as shown in the following example.

```
balance = 0

def deposit(amount):
    global balance
    balance += amount
    return balance

def withdraw(amount):
    global balance
    balance -= amount
    return balance
```

The above example is good enough only if we want to have just a single account. Things start getting complicated if want to model multiple accounts.

We can solve the problem by making the state local, probably by using a dictionary to store the state.

```
def make_account():
    return {'balance': 0}

def deposit(account, amount):
    account['balance'] += amount
    return account['balance']

def withdraw(account, amount):
    account['balance'] -= amount
    return account['balance']
```

With this it is possible to work with multiple accounts at the same time.

```
>>> a = make_account()
>>> b = make_account()
>>> deposit(a, 100)
100
>>> deposit(b, 50)
50
>>> withdraw(b, 10)
40
>>> withdraw(a, 10)
90
```

4.2. Classes and Objects

```
class BankAccount:
    def __init__(self):
        self.balance = 0

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

>>> a = BankAccount()
>>> b = BankAccount()
>>> a.deposit(100)
100
>>> b.deposit(50)
50
>>> b.withdraw(10)
40
>>> a.withdraw(10)
90
```

4.3. Inheritance

Let us try to create a little more sophisticated account type where the account holder has to maintain a pre-determined minimum balance.

```
class MinimumBalanceAccount(BankAccount):
    def __init__(self, minimum_balance):
        BankAccount.__init__(self)
        self.minimum_balance = minimum_balance

    def withdraw(self, amount):
        if self.balance - amount < self.minimum_balance:
            print 'Sorry, minimum balance must be maintained.'
        else:
            BankAccount.withdraw(self, amount)
```

Problem 1: What will the output of the following program.

```
class A:
    def f(self):
        return self.g()

    def g(self):
        return 'A'

class B(A):
    def g(self):
        return 'B'

a = A()
b = B()
print a.f(), b.f()
print a.g(), b.g()
```

Example: Drawing Shapes

```
class Canvas:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.data = [[' '] * width for i in range(height)]

    def setpixel(self, row, col):
        self.data[row][col] = '*'

    def getpixel(self, row, col):
        return self.data[row][col]

    def display(self):
        print "\n".join(["".join(row) for row in self.data])

class Shape:
    def paint(self, canvas): pass

class Rectangle(Shape):
    def __init__(self, x, y, w, h):
        self.x = x
        self.y = y
        self.w = w
        self.h = h

    def hline(self, x, y, w):
        pass

    def vline(self, x, y, h):
        pass

    def paint(self, canvas):
        hline(self.x, self.y, self.w)
        hline(self.x, self.y + self.h, self.w)
        vline(self.x, self.y, self.h)
        vline(self.x + self.w, self.y, self.h)

class Square(Rectangle):
    def __init__(self, x, y, size):
        Rectangle.__init__(self, x, y, size, size)

class CompoundShape(Shape):
    def __init__(self, shapes):
        self.shapes = shapes

    def paint(self, canvas):
        for s in self.shapes:
            s.paint(canvas)
```

4.4. Special Class Methods

In Python, a class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators.

For example, the `+` operator invokes `__add__` method.

```
>>> a, b = 1, 2
>>> a + b
3
>>> a.__add__(b)
3
```

Just like `__add__` is called for `+` operator, `__sub__`, `__mul__` and `__div__` methods are called for `-`, `*`, and `/` operators.

Example: Rational Numbers

Suppose we want to do arithmetic with rational numbers. We want to be able to add, subtract, multiply, and divide them and to test whether two rational numbers are equal.

We can add, subtract, multiply, divide, and test equality by using the following relations:

```
n1/d1 + n2/d2 = (n1*d2 + n2*d1)/(d1*d2)
n1/d1 - n2/d2 = (n1*d2 - n2*d1)/(d1*d2)
n1/d1 * n2/d2 = (n1*n2)/(d1*d2)
(n1/d1) / (n2/d2) = (n1*d2)/(d1*n2)

n1/d1 == n2/d2 if and only if n1*d2 == n2*d1
```

Lets write the rational number class.

```
class RationalNumber:
    """
    Rational Numbers with support for arthmetic operations.

        >>> a = RationalNumber(1, 2)
```

```
    >>> b = RationalNumber(1, 3)
    >>> a + b
    5/6
    >>> a - b
    1/6
    >>> a * b
    1/6
    >>> a/b
    3/2
"""
def __init__(self, numerator, denominator=1):
    self.n = numerator
    self.d = denominator

def __add__(self, other):
    if not isinstance(other, RationalNumber):
        other = RationalNumber(other)

    n = self.n * other.d + self.d * other.n
    d = self.d * other.d
    return RationalNumber(n, d)

def __sub__(self, other):
    if not isinstance(other, RationalNumber):
        other = RationalNumber(other)

    n1, d1 = self.n, self.d
    n2, d2 = other.n, other.d
    return RationalNumber(n1*d2 - n2*d1, d1*d2)

def __mul__(self, other):
    if not isinstance(other, RationalNumber):
        other = RationalNumber(other)

    n1, d1 = self.n, self.d
    n2, d2 = other.n, other.d
    return RationalNumber(n1*n2, d1*d2)

def __div__(self, other):
    if not isinstance(other, RationalNumber):
        other = RationalNumber(other)

    n1, d1 = self.n, self.d
    n2, d2 = other.n, other.d
    return RationalNumber(n1*d2, d1*n2)

def __str__(self):
    return "%s/%s" % (self.n, self.d)

__repr__ = __str__
```

4.5. Errors and Exceptions

We've already seen exceptions in various places. Python gives `NameError` when we try to use a variable that is not defined.

```
>>> foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
```

try adding a string to an integer:

```
>>> "foo" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

try dividing a number by 0:

```
>>> 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

or, try opening a file that is not there:

```
>>> open("not-there.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'not-there.txt'
```

Python raises exception in case errors. We can write programs to handle such errors. We too can raise exceptions when an error case in encountered.

Exceptions are handled by using the try-except statements.

```
def main():
    filename = sys.argv[1]
    try:
        for row in parse_csv(filename):
            print row
    except IOError:
        print >> sys.stderr, "The given file doesn't exist: ", filename
        sys.exit(1)
```

This above example prints an error message and exits with an error status when an IOError is encountered.

The *except* statement can be written in multiple ways:

```
# catch all exceptions
try:
    ...
except:

# catch just one exception
try:
    ...
except IOError:
    ...

# catch one exception, but provide the exception object
try:
    ...
except IOError, e:
    ...

# catch more than one exception
try:
    ...
except (IOError, ValueError), e:
    ...
```

It is possible to have more than one *except* statements with one *try*.

```
try:
    ...
except IOError, e:
    print >> sys.stderr, "Unable to open the file (%s): %s" % (str(e), filename)
    sys.exit(1)
except FormatError, e:
    print >> sys.stderr, "File is badly formatted (%s): %s" % (str(e), filename)
```

The *try* statement can have an optional *else* clause, which is executed only if no exception is raised in the try-block.

```
try:
    ...
except IOError, e:
    print >> sys.stderr, "Unable to open the file (%s): %s" % (str(e), filename)
    sys.exit(1)
else:
    print "successfully opened the file", filename
```

There can be an optional *else* clause with a *try* statement, which is executed irrespective of whether or not exception has occured.

```
try:
    ...
except IOError, e:
    print >> sys.stderr, "Unable to open the file (%s): %s" % (str(e), filename)
    sys.exit(1)
finally:
    delete_temp_files()
```

Exception is raised using the raised keyword.

```
raise Exception("error message")
```

All the exceptions are extended from the built-in *Exception* class.

```
class ParseError(Exception):
    pass
```

Problem 2: What will be the output of the following program?

```
try:
    print "a"
except:
    print "b"
else:
    print "c"
finally:
    print "d"
```

Problem 3: What will be the output of the following program?

```
try:
    print "a"
    raise Exception("doom")
except:
    print "b"
else:
    print "c"
finally:
    print "d"
```

Problem 4: What will be the output of the following program?

```
def f():
    try:
        print "a"
        return
    except:
        print "b"
    else:
        print "c"
    finally:
        print "d"

f()
```

Previous Next