Programación orientada a objetos

Abstract Data Types

An **abstract data type** is a set of objects and the operations on those objects. These are bound together so that one can pass an object from one part of a program to another, and in doing so provide access not only to the data attributes of the object but also to operations that make it easy to manipulate that data.

The specifications of those operations define an **interface** between the abstract data type and the rest of the program. The interface defines the behavior of the operations- what they do, but not how they do it. The interface thus provides an **abstraction barrier** that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction.

Programming is about managing complexity in a way that facilitates change. There are two powerful mechanisms available for accomplishing this: decomposition and abstraction. **Decomposition** creates structure in a program, and **abstraction** suppresses detail. The key is to suppress the appropriate details. This is where data abstraction hits the mark. One can create domain-specific types that provide a convenient abstraction. Ideally, these types capture concepts that will be relevant over the lifetime of a program. If one starts the programming process by devising types that will be relevant months and even decades later, one has a great leg up in maintaining that software.

Abstraction Example:

```
class IntSet(object):
   """An intSet is a set of integers"""

   # Information about the implementation (not the abstraction)
   # value of the set is represented by a list of ints, self.vals.
   # Each int in the set occurs in self.vals exactly once.

   def __init__(self):
       """Create an empty set of integers"""
       self.vals = []

   def insert(self, e):
       """Assumes e is an integer and inserts e into self"""
       if e not in self.vals:
           self.vals.append(e)

   def member(self, e):
       """Assumes e is an integer
       Returns True if e is in self, and False otherwise"""
       return e in self.vals
```

```
    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + 'not found')

    def get_members(self):
        """Returns a list containing the elements of self.
        Nothing can be assumed about the order of the elements"""
        return self.vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}'  # -1 omits trailing comma
```

When a function definition occurs within a class definition, the defined function is called a **method** and is associated with the class. These methods are sometimes referred to as **method attributes** of the class.

Classes support two kind of operations:

- **Instantiation** is used to create instances of the class. For example, the statement *s = IntSet()* creates a new object of type IntSet. This object is called an **instance** of *IntSet*.
- **Attribute references** use dot notation to access attributes associated with the class. For example, *s.member* refers to the method member associated with the instance *s* of type *IntSet*.

Each class definition begins with the reserved word class followed by the name of the class and some information about how it relates to other classes.

Whenever a class is instantiated, a call is made to the *__init__* method defined in that class. When the line of code

s = IntSet()

is executed, the interpreter will create a new instance of type *IntSet*, and then call *IntSet.__init__* with the newly created object as the actual parameter that is bound to the formal parameter self.

The methods receive one parameter instead of two because the object associated with the expression preceding the dot is implicitly passed as the first parameter to the method.

When data attributes are associated with a class we call them **class variables**. When they are associated with an instance we call them **instance variables**.

Data abstraction achieves representation-independence. Think of the implementation of an abstract type as having several components:

- Implementation of the methods of the type,
- Data structures that together encode values of the type, and
- Conventions about how the implementations of the methods are to use the data structures. A key convention is captured by the representation invariant.

The **representation invariant** defines which values of the data attributes correspond to valid representations of the class instances. The implementation of __*init*__ is responsible for establishing the invariant (which holds on the empty list), and the other methods are responsible for maintaining that invariant.

The __*str*__ method of a class is also invoked when a program converts an instance of that class to a string by calling str.

Designing Programs Using Abstract Data Types

Abstract data types are a big deal. They lead to a different way of thinking about organizing large programs. When we think about the world, we rely on abstractions. In the world of finance people talk about stocks and bonds. In the world of biology people talk about proteins and residues. When trying to understand concepts such as these, we mentally gather together some of the relevant data and features of these kinds of objects into one intellectual package. For example, we think of bonds as having an interest rate and a maturity date as data attributes. We also think of bonds as having operations such as “set price” and “calculate yield to maturity”. Abstract data types allow us to incorporate this kind of organization into the design of programs.

Data abstraction encourages program designers to focus on the centrality of the data objects rather than the functions. Thinking about a program more as a collection of types than as a collection of functions leads to a profoundly different organizing principle. Among other things, it encourages one to think about programming as a process of combining relatively large chunks, since data abstractions typically encompass more functionality than do individual functions. This, in turn, leads us to think of the essence of programming as a process not of writing individual lines of code, but composing abstractions.

Using Classes to Keep Track of Students and Faculty

As an example use of classes, imagine that you are designing a program to help keep track of all students and faculty at university. It is certainly possible to write such a program without using data abstraction. Each student would have a family name, a given name, a home address, a year, some grades, etc. This could all be kept in some combination of list and dictionaries. Keeping track of faculty and staff would require some similar data structures and some different data structures, e.g., data structures to keep track of things like salary history.

Before rushing in to design a bunch of data structures, let's think about some abstractions that might prove useful. Is there an abstraction that covers the common attributes of students, professors, and staff? Some would argue that they are all human.

Person class example:

```
import datetime

class Person(object):

   def __init__(self, name):
        """Create a person"""
        self.name = name

        try:
            lastBlank = name.rindex(' ')
            self.lastName =name[lastBlank+1:]
        except:
            self.lastName = name

        self.birthday = None

        def get_name(self):
            """Return self's full name"""
            return self.name

        def get_last_name(self):
            """Return self's last name"""
            return self.lastName

        def set_birthday(self, birthdate):
            """Assumes birthday is of type datetime.date
            Sets self's birthday to birthdate"""
            self.birthday = birthdate

        def getAge(self):
            """Returns self's current age in days"""
            if self.birthday == None:
                raise ValueError
            return (datetime.date.today() - self.birthday).days

        def __lt__(self, other):
            """Returns True if self precedes other in alphabetical order,
            and False otherwise. comparison is based on last names,
            but if these are the same full names are compared"""
            if self.lastName == other.lastName:
                return self.name < other.name
            return self.lastName < other.lastName
        def __str__(self):
            """Returns self's name"""
            return self.name
```

When instantiating a class we need to look at the specification of the *__init__* function for that class to know what arguments to supply and what properties those arguments should have.

We can access to the lastName of an instance by *instance.lastName* but writing expressions that directly access instance variables is considered poor form, and should be avoided. Similarly, there is no appropriate way for a user of the Person abstraction to extract a person's birthday, despite the fact that the implementation contains an attribute with that value.

The *__lt__* method overloads the < operator. It is of type *str* and is used when we sort a list with elements of *Person* type.

Inheritance

Inheritance provides a convenient mechanism for building groups of related abstractions. It allows programmers to create a type hierarchy in which each type inherits attributes from the types above it in the hierarchy.

The class object is at the top of the hierarchy. This makes sense, since in Python everything that exists at run time is an object. Because *Person* inherits all of the properties of objects, programs can bind a variable to a Person, append a *Person* to a list, etc.

The class MITPerson inherits attributes from its parent class, Person, including all of the attributes that Person inherited from its parent class, object. In the jargon of object-oriented programming, MITPerson is a **subclass** of Person, and therefore **inherits** the attributes of its superclass. In addition to what it inherits, the subclass can:

- Add new attributes.
- Override, i.e., replace attributes of the superclass. When a method has been overridden, the version of the method that is executed is based on the object that is used to invoke the method.If the type of the object is the subclass, the version defined in the subclass will be used. If the type of the object is the superclass, the version in the superclass will be used.

Subclass example:

```
class MITPerson(Person):

   next_id_num = 0 # identification number
```

```
    def __init__(self, name):
        Person.__init__(self, name)
        self.id_num = MITPerson.next_id_num
        MITPerson.next_id_num += 1

    def get_id_num(self):
        return self.id_num

    def __lt__(self, other):
        return self.id_num < other.id_num
```

Multiple Levels of Inheritance

```
class Student(MITPerson):
    pass

class UG(Student):
    def __init__(self, name, class_year):
        MITPerson.__init__(self, name)
        self.year = class_year

    def get_class(self):
        return self.year

class Grad(Student):
    pass
```

By introducing Grad, we gain the ability to create two different kinds of students and use their types to distinguish one ind of object from another. For example

p5 = Grad(‘Buzz aldrin’)
p6 = UG(‘Billy Beaver’, 1984)
print(p5, ‘is a graduate student is ‘, type(p5) == Grad)
print(p5, ‘is an undergraduate student’, type(p6) == UG)

will print

Buzz Aldrin is a graduate student is True
Buzz Aldrin is a undergraduate student is False

Consider going back to class *MITPerson* and adding the method

```
def is_student(self):
        return isinstance(self, Student)
```

Testing the new function

```
p5 = Grad('Buzz aldrin')
p6 = UG('Billy Beaver', 1984)

print(p5.is_student())
```

Notice that the meaning of *isinstance(p5, Student)* is quite different from the meaning of *type(p6)* == *Student*. The object to which p6 is bound is of type UG, not Student, but since UG is a subclass of Student, the object to which p6 is bound is considered to be an instance of class Student (as well as an instance of MITPerson and Person).

The implementation of *is_student* method allow us avoid to edit code when adding different types of students if the method was

```
def is_student(self):
    return type(self) == Grad or type(self) == UG
```

```
class TransferStudent(Student):

   def __init__(self, name, from_school):
       MITPerson.__init__(self, name)
       self.from_school = from_school

   def get_old_school(self):
       return self.from_school
```

The substitution principle

When subclassing is used to define a type hierarchy, the subclasses should be thought of as extending the behavior of their superclasses. We do this by adding new attributes or overriding attributes inherited from a superclass. For example, *TransferStudent* extends *Student* by introducing a former school.

Sometimes, the subclass overrides methods from the superclass, but this must be done with care. In particular, important behaviors of the supertype must be supported by each of its subtypes. If client code works correctly using an instance of the supertype, it should also work correctly when an instance of the subtype is substituted for the instance of the supertype.

Encapsulation and Information Hiding

Example: class to track student's course grades

```
class Grades(object):

   def __init__(self):
       """Create empty grade book"""
       self.students = []
       self.grades = {}
       self.is_sorted = True

   def add_student(self, student):
       """Assumes: student is of type Student
       Add student to the grade book"""
       if student in self.students:
           raise ValueError('Duplicate student')
       self.students.append(student)
       self.grades[student.get_id_num()] = []
       self.is_sorted = False

   def add_grade(self, student, grade):
       """Assumes: grade is a float
       Add grade to the list of grades for student"""
       try:
           self.grades[student.get_id_num()].append(grade)
       except:
           raise ValueError('Student not in mapping')

   def get_grades(self, student):
       """Return a list of grades for student"""
       try:  # return copy of list of student's grades
           return self.grades[student.get_id_num()][:]
       except:
           raise ValueError('Student not in mapping')

   def get_students(self):
       """Return a sorted list of the students in the grade book"""
       if not self.is_sorted:
           self.students.sort()
           self.is_sorted = True
       return self.students[:]  # return copy of list of students
```

Example: function to produce a grade report

```
def grade_report(course):
   """Asumes course is of type Grades"""
   report = ''
   for s in course.get_students():
       tot = 0.0
       num_grades = 0
       for g in course.get_grades(s):
           tot += g
           num_grades += 1
       try:
           average = tot / num_grades
           report = report + '\n' \
                    + str(s) + '\'s mean grade is ' + + str(average)
       except ZeroDivisionError:
           report = report + '\n' \
                    + str(s) + ' has no grades'
       return report
```

There are two important concepts at the heart of object-oriented programming. The first is the idea of **encapsulation**. By this we mean the bundling together of data attributes and the methods for operating on them. For example:

Rafael = MITPerson('Rafael Reif')

We can use dot notation to access attributes such as Rafael's name and identification number.

The second important concept is **information hiding**. This is one of the key to modularity if those part of the program that use a class (i.e., the clients of the class) rely on the specifications of the methods in the class, a programmer implementing the class is free to change the implementation of the class(e.g., to improve efficiency) without worrying that the change will break code that uses the class.

Python 3 uses naming convention to make attributes invisible outside the class. When the name of an attribute starts with __ but does not end with __, that attribute is not visible outside the class.

Example: invisible (private) attributes.

```
class InfoHiding(object):

   def __init__(self):
       self.visible = 'look at me'
       self.__also_visible__ = 'look at me too'
       self.__invisible = 'don\'t look at me directly'

   def print_visible(self):
       print(self.visible)

   def print_invisible(self):
       print(self.__invisible)

   def __print_invisible(self):
        print(self.__invisible)

    def __print_invisible__(self):
        print(self.__invisible)

test = InfoHiding()

print(test.visible)
print(test.__also_visible__)
print(test.__invisible)
```