8.4 Subclasses

There are many natural situations where some elements of a class have more or different properties than the other elements. For example, all people have names and addresses. Some people, the students, also have majors and gpas. Other people have job titles and annual salaries. Subclasses give a way to model these situations. A subclass refines a class by adding more instance variables or new methods.

There is a variety of terminology for classes and subclasses. Some people call the class being refined the "super class". Some people call the subclass a "child class" and the class it refines the "parent class". Whatever terminology you prefer, the refining class is generally smaller than the class it refines. Every element of the subclass is also an element of the parent class. The converse of this is false. For example, every student is a person, but not every person is a student. This means that all of the instance variables and all of the methods of the parent class are available to the subclass. We say the subclass *inherits* the data and methods of the parent class. The subclass may alter the definition of the methods of the parent class, or it may use them unaltered.

Why do we use subclasses? One reason is that they allow us to avoid re-implementing the methods of the parent class. Anything that allows us to re-use code helps to avoid errors in the code. Another reason is that this is a good organizational tool. Suppose we have classes that represent various geometric shapes: Square, Circle , and Polygon. If they are all subclasses of a basic Shape class, and if this Shape class has a method Moveto( ) then we know they all have such a method. This means we could put any of these shapes into a list and move them by processing the list with a **for**-loop without worrying which class a particular shape is from.

A subclass definition looks just like a class definition, only it contains the name of the parent class in parentheses, as in

```
class Circle(Shape):
    ...
```

Python allow a class to inherit from several parent classes. Setting up multiple inheritance correctly is not easy and we generally will avoid the problems this creates by only giving classes a single parent.

Subclasses inherit the methods of their parent classes. If the subclass provides a new definition of a method of its parent, the subclass has access to both version of the method, with the parents version accessed as <**class** name>.<method name>(self,) For example, subclasses usually overwrite the constructor method __init__ ( ) of the parent. If class B is a subclass of class A, then inside B we can refer to As constructor as A. __init__ ( )

Here is an example of two classes: Person and Student, with Student a subclass of Person. Note that the Student constructor calls the Person constructor, and also the method SetAge( ) method of class Person. Class Student does not give a new version of the __str__ ( ) method of class Person; it just uses the version inherited from Person.

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.age = 0

    def SetAge(self, a):
        self.age = a

    def __str__(self):
        return "%s, who is %d years old"%(self.name,self.age)

class Student(Person):
    def __init__(self, myName):
        Person.__init__(self, myName)
        self.SetAge(18)
        self.major = " "

    def SetMajor(self, subject):
        self.major = subject

    def Major(self):
        return self.major

def main():
    x = Student("joe")
    x.SetAge(21)
    x.SetMajor("Computer Science" )
    print(x)

main()
```

Program 8.4.1: Class Student is a subclass of Person.

In the next example we model the payroll office of a company that has three kinds of workers: hourly workers, who have an hourly wage and work 40 hours per week, salaried workers who have an annual salary and are paid every fourth week, and interns,who arent paid at all. Each class of workers has a method Pay( ) that takes as argument a week number (the weeks are numbered 0, 1, 2, and so forth). The hourly workers are paid each week. The salaried workers are paid when the week%4 is 3 (i.e, on the last week of each month). The constructor for each class calls the Employee constructor, as that assigns a unique employee number to each staff member. The individual subclass constructors also save their own information, in addition to the work of the Employee constructor.

Here is the top class: Employee:

```
# Class Employee has the code common to all employees,
# There are 3 subclasses that refine Employee with more
# details.

class Employee:
    Count = 0
    def __init__(self, name, jobTitle):
        self.name = name
        self.title = jobTitle
        Employee.Count = Employee.Count + 1
        self.employeeNumber = Employee.Count

    def Print(self):
        print "Name: %s  Employee Number: %d  Title: %s" % \
                  (self.name, self.employeeNumber, self.title)

    def Pay(self, week):
        pass
```

Program 8.4.2: The top-level class.

Note the use of the **pass** command in Employees Pay( ) method. This is one way to have a function in Python that does nothing — the system ignores the **pass** statement. Functions must have something in the body, so some statement that has no effect is necessary.

Next we have the definitions of the three subclasses of Employee: HourlyWorker, SalariedWorker, and Intern. Each of these only needs to implement the portion of their functionality that is not covered in class Employee.

```
class SalariedWorker(Employee):
    # Salaried workders are paid once a month
    # (every 4th pay period).  Then they are paid
    # one twelfth of their annual salary
    def __init__(self, name, jobTitle, annualSalary):
        Employee.__init__(self, name, jobTitle)
        self.salary = annualSalary

    def Print(self):
        print( "%s is a salaried worker with salary $%d" %\
                        (self.name, self.salary))

    def Pay(self, week):
        if week % 4 == 3:
            print( "Pay %s $%.2f" % \
                        (self.name, self.salary/12.0))

class HourlyWorker(Employee):
    # Hourly workers are paid every week.
    # Each period they are paid 40 times their hourly wage.
    def __init__(self, name, jobTitle, hourlyWage):
        Employee.__init__(self, name, jobTitle)
        self.wage = hourlyWage

    def Print(self):
        print( "%s is an hourly worker with wage $%d" % \
                        (self.name, self.wage))

    def Pay(self, week):
        print( "Pay %s $%.2f" % \
                        (self.name, self.wage*40))

class Intern(Employee):
    # Interns aren't paid
    def __init__(self, name):
        Employee.__init__(self, name, "Robot")

    def Print(self):
        print( "%s is an intern" % (self.name))
```

Program 8.4.2: 3 subclasses of Employee.

Finally, we have two utility functions: one for maintaining a list of all employees and one for processing this list to pay all of the employees that need

paying in a given week. Note that we can put employees into the employee list in any order, and we can process this list without worrying about what class each element of the list comes from. All of the objects have a Pay( ) method (even the unpaid interns; they inherit the default Pay( ) method from class Employee). This ability to manage a variety of classes as though there are all the same class is one of the great strengths of subclasses.

```
def Hire(list, employee):
    list.append(employee)

def PayWorkers(list, numWeeks):
    for week in range(0, numWeeks):
        print( "Pay period %d" % week)
        for employee in list:
            employee.Pay(week)

def main():
    Staff = []

    Hire(Staff,SalariedWorker("Suzie","Engineer",50000))
    Hire(Staff,SalariedWorker("Bob","Boss",120000))
    Hire(Staff,HourlyWorker("Fred","Construction Worker",10))
    Hire(Staff,Intern("Mary"))
    Hire(Staff,Intern("Herman"))

    for x in Staff:
        x.Print()

    PayWorkers( Staff, 12)

main()
```

Program 8.4.2: Finishing the program.