Defining classes and using objects

Read: *Sections 10.1 – 10.5 from Chapter 10 in the textbook*

- We can view a **type** like int, float, str, list etc. as a *mold*, out of which we can produce as many items as we want, all with the same 'properties'. For example, when we type

  ```
  >>> x = “John”
  >>> y = “Doe”
  ```

  we construct two items (called **instances**) x and y out of the same type(mold) string. We have studied some of the properties of all these types (e.g., strings come equipped with a full set of functions (we call them **methods –** we'll see why later) that perform all sorts of operations (e.g., split()). In this unit, we will learn how to build our own molds (called **classes**) out of which we can mass-produce items called **objects**. The 'properties' a class endows its objects with are

  1. a set of internal variables
  2. a set of methods

  The general form of a class definition is the following:

  ```
  class <class-name>
      <method-definitions>
  ```

  Let's look at the definition of a class (slightly modified from what is found in module msdie.py):

  ```
  #  msdie.py
  #  Class definition for an n-sided die.

  from random import randrange

  class MSDie:

      def __init__(self, s):
          self.sides = s
          self.value = 1

      def roll(self):
          self.value = randrange(1,self.sides+1)

      def getValue(self):
          return self.value
  ```

```
    def setValue(self, v):
        self.value = v
```

- For now, don't worry about what the objects of this class are doing. We are going to use randrange(), so we import it. The name of the class is MSDie, and it has 4 methods.
- The method __init__ is a special method we will find in any class definition, called the **constructor**. It takes two parameters, the first called self and the second called sides. Notice that the first parameter in all 4 methods are all named the same by the author (self); we start suspecting that this first parameter is used in the same way in all methods. Indeed, the first parameter[1] is a reference (pointer) to the object itself; it allows a method to know who is its "owner" object, so that if it needs to access anything from the "owner" it knows how to ask for it (or, more accurately, whom to ask it from). You can see this usage of the self-pointing reference in the constructor (__init__), where the internal variables of the object are initialized. Here there are two internal variables: sides and value. But whose sides and values (remember, there may be many many objects of the class MSDie)? Well, the object's own sides and values and not anybody else's. Hence the self.sides and self.values! The first is initialized with the value passed in the second parameter of the constructor (s), and the second is always initialized to 1. So the constructor initializes the internal variables; that's why it runs automatically whenever a new object of this class is created.
- Notice the use of internal variables in the other methods.
- Let's look at how an object of this class is created. After you import msdie.py, type:

```
>>> die1 = MSDie(12)
```

Notice that we create the object die1 by invoking the name of the *class* and passing to it a value (12) for parameter s of the constructor. So, when an object is created, immediately the constructor is called, with parameter values:

```
    self = die1
    s = 12
```

Note that we didn't pass the value for self; Python knows that self is the object itself die1. When the object is created, we can access its internal variables:

```
>>> die1.sides
12
>>> die1.value
1
```

In general, the creation of an object obj happens by a call to its class constructor as follows:

```
obj = <class-name>(<param1>, <param2>, ...)
```

1You don't have to name it self; you can name it this (like Java), or Bob, or Sandra, but self is more indicative of what it does.

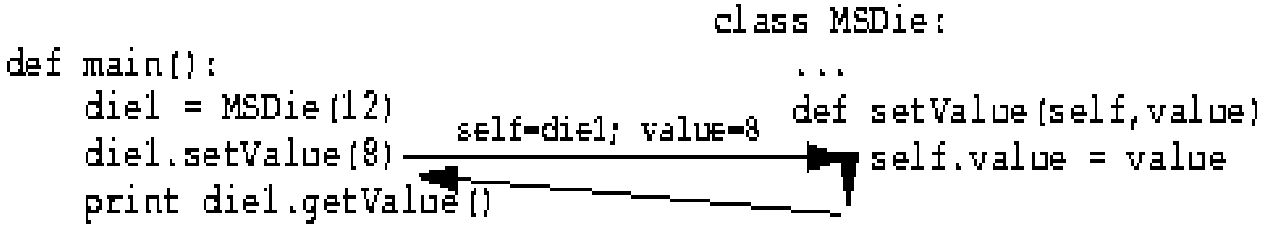
- We can also access the object methods:

```
>>> die1.getValue()
1
>>> die1.setValue(19)
>>> die1.value
19
>>> die1.roll()
 >>> die1.getValue()
```

In general, an object method is called as follows:

<object>.<method-name>(<param1>, <param2>, ...)

- Here is an example of using objects in a program:



The picture above shows what happens when the method die1.setValue is called.

- **Example:** In the text you can find a simple application that calculates the distance a cannonball travels, given the initial conditions. In Sections 10.2.1 and 10.2.2 you can find the specifications of the problem, as well as a top-down design of the program. The final product cball1.py doesn't use functions, although it is pretty clear that certain pieces of the code can be bundled into a function (as you would expect from the top-down design approach).

  Practice problem: Write again cball1.py but now use functions as you see fit (and natural). Actually, the book itself proposes a way of splitting the code into function in the main() of Section 10.2.3, p. 303, but feel free to follow your own design.

  So far, we haven't seen anything more than what we've already known since the last lab. But now we know about classes, objects, and object-oriented programming (i.e., design programs by designing objects). The book proposes the creation of a class named Projectile so that the main() looks like p. 304.

  Practice problem: Implement the object class Projectile. (You may or may not have noticed that the book does exactly that in Section 10.3.2; *please ignore it!* After you are done, compare your implementation with the one in the book.)
  Practice problem: Even if you skipped directly to the book implementation in Sec. 10.3.2, please do Problem 9 in p. 335 (it's very similar to the example). Please do it even if you have

just finished implementing Projectile.

- A common structure in programming languages is a *record,* i.e., a type that contains more than one fields of simpler types. For example, the following is how a record type would look like:

```
record student {
    string name;
    int hours;
    float qpoints;
} s1, s2, s3;
```

  In Python, we can define such a type as an object class, whose internal variables play the role of the fields. Then we can create objects s1,s2,s3 of this class. An example of how this can be done is the Student class in Section 10.4 (we have already seen that a simpler implementation of records is through lists with one list element per record field; but lists don't even come near to the power of objects when it comes to manipulating records).
- A main goal behind object-oriented programming is to push the implementation details to the background, so that a user of a class needs only to know the variables and how the methods are called and what they do. This is called **encapsulation**.
- We can write documentation for modules, classes, functions by writing comments between “””” (three double quotes). Then these comments (called *docstrings*) can be displayed by the user of the module, class, function, by using attribute __doc__ . Read how you can write and use docstrings in Section 10.5.3.

  Practice problem: Equip class Projectile with docstrings.
  Practice problem: Change your comments for your blackjack project into docstrintgs.

- The particular values of the object internal variables are sometimes called the **state** of the object. A method that changes one or more of these values is sometimes called a **mutator**.

  Practice problem: Write a class that handles a list. It has one internal variable (the list), and methods that:
  - append a new item at the end of the list
  - return the current list
  - search the list for an item and returns the position or -1 if not in the list
  - delete the last item of the list

  The constructor of the class must set the list to be the empty list.
  Practice problem: Do all problems (except those related to graphics) from the textbook.