HAB 619 Introduction to Scientific Computing in Sports Science

#4

De Motu Animalium G.Borelli (1680)

SERDAR ARITAN

serdar.aritan@hacettepe.edu.tr

Biyomekanik Araştırma Grubu
www.biomech.hacettepe.edu.tr
Spor Bilimleri Fakültesi
www.sbt.hacettepe.edu.tr
Hacettepe Universitesi, Ankara, Türkiye
www.hacettepe.edu.tr

Functions and Classes

- **Functions**
- Classes

Functions and Classes

Functions in Python are first-class objects. Programming language theorists define a “first-class object” as a program entity that can be:

- Created at runtime
- Assigned to a variable or element in a data structure
- Passed as an argument to a function
- Returned as the result of a function

Functions and Classes

When to Use a Function

- Only one purpose: A function should be the encapsulation of a single, identifiable operation.
- Readable: A function should be readable.
- Not too long: A function shouldn’t be too long.
- Reusable: A function should be reusable in contexts other than the program it was written for originally.
- Complete: A function should be complete, in that it works in all potential situations. If you write a function to perform one thing, you should make sure that all the cases where itmight be used are taken into account.
- Able to be refactored: Refactoring is the process of taking existing code and modifying it such that its structure is somehow improved but the functionality of the code remains the same.

Functions and Classes

```
def name(arg1, arg2,... argN):
        statements
```

As with all compound Python statements, def consists of a header line followed by a block of statements, usually indented (or a simple statement after the colon).

def Executes at Runtime

The Python **def** is a true executable statement: when it runs, it creates a new function object and assigns it to a name. Because it's a statement, a **def** can appear anywhere a statement

```
if test:
      def func():          # Define func this way
      ...
else
      def                  # Or else this way
      ...

func()       # Call the version selected and built
```

Functions and Classes

Just like a value can be associated with a name, a piece of logic can also be associated with a name by defining a function.

```
>>> def square(x):
        return x * x
>>> square(5)
25
```

The body of the function is indented. Indentation is the Python's way of grouping statements.

The functions can be used in any expressions.

```
def square(x):
    return x * x
```

```
>>> square(2) + square(3)
 13
>>> square(square(3))
 81
```

Functions and Classes

We can even create more functions using the existing ones.

```
>>> def sum_of_squares
        return square(x) + square(y)
>>> sum_of_squares(2, 3)
13
```

Functions are just like other values, they can assigned, passed as arguments to other functions etc.

```
>>> f = square
>>> f(4)
16
>>> def fxy(f, x, y):
        return f(x) + f(y)

>>> fxy(square, 2, 3)
13
```

Functions and Classes

```
>>>def cube(x):
            return x * x * x
>>> fxy(cube, 2, 3)
35
```

Python supports simple anonymous functions through the **lambda** form.

```
>>>forthpower = lambda x: x ** 4
>>> fxy(forthpower, 2, 3)
97
>>> fxy(lambda x: x ** 5, 2, 3)
275
```

lambda

The **lambda** operator becomes handy when writing small functions to be passed as arguments etc.

Functions and Classes

Lambda functions are anonymous functions that are not defined in the namespace. Roughly speaking, they are functions without names, intended for single use.

```
lambda <arguments> : <return expression>
```

A lambda function can have one or multiple arguments, separated by commas.

```
>>> print((lambda x: x + 3)(3))
6
>>> print((lambda x, y: x + y)(3, 4))
7
```

Functions and Classes

Generally, `defs` are not evaluated until they are reached and run, and the code inside `defs` is not evaluated until the functions are later called. Because function definition happens at runtime, there's nothing special about the function name. What's important is the object to which it refers:

```
othername = func   # Assign function object
othername()        # Call func again
```

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just objects; they are recorded explicitly in memory at program execution time.

```
def func(): ...    # Create function object
func()             # Call object
func.attr = value        # Attach attributes
```

Functions and Classes

What will be the output of the following programs?

```
x = 1
def f():
      x = 2
      return x
 print(x) ??
 print(f()) ??

 <CRTL>F6
x = 1
def f():
      y = x
      x = 2
      return x + y
 print (x) ??
 print (f()) ??
```

Functions and Classes

Functions can be called with keyword arguments.

```
>>> def difference(x, y):
      return x - y

>>> difference(5, 2)
3
>>> difference(x=5, y=2)
3
>>> difference(5, y=2)
3
>>> difference(y=2, x=5)
3
```

Functions and Classes

And some arguments can have default values.

```
>>> def increment(x, amount=1):
      return x + amount
>>> increment(10)
1
>>> increment(10, 5)
15
>>> increment(10, amount=2)
12
```

Functions and Classes

How to unpack more than one variable when functions return multiple values.

```python
def get_stats(numbers):
      minimum = min(numbers)
      maximum = max(numbers)
      return minimum, maximum


lengths = [63, 73, 72, 60, 67, 66, 71, 61, 72, 70]

# Two return values
minimum, maximum = get_stats(lengths)
print(f'Min: {minimum}, Max: {maximum}')
```

Functions and Classes

Keyword Arguments

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name}.")

describe_pet('dog', ‘Bruno')
describe_pet(‘Bruno', 'dog')
```

```
I have a dog.
My dog's name is Bruno.

I have a bruno.
My bruno's name is Dog.
```

Functions and Classes

Keyword Arguments

When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.

```
describe_pet(animal_type=‘dog', pet_name=‘Bruno')
describe_pet(pet_name=‘Bruno', animal_type=‘dog')
```

```
I have a dog.
My dog's name is Bruno.

I have a dog.
My dog's name is Bruno.
```

Functions and Classes

Default Values

```
def describe_pet(pet_name, animal_type=‘dog’):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name}.")

describe_pet(pet_name = ‘Bruno')
```

```
I have a dog.
My dog's name is Bruno.

I have a bruno.
My bruno's name is Dog.
```

Functions and Classes

Making an Argument Optional

```
def get_formatted_name(first_name, middle_name, last_name):
 """Return a full name, neatly formatted."""
 full_name = f"{first_name} {middle_name} {last_name}"
 return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

```
John Lee Hooker
```

Functions and Classes

Making an Argument Optional

```
def get_formatted_name(first_name, last_name, middle_name=‘ ‘ ):
  """Return a full name, neatly formatted."""
  if middle_name:
     full_name = f"{first_name} {middle_name} {last_name}“
  else:
     full_name = f"{first_name} {last_name}“
  return full_name.title()

musician = get_formatted_name('jimi', ‘hendrix')
print(musician)
Jimi Hendrix

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
John Lee Hooker
```

Functions and Classes

```
def drawBox():
      print("**********")
      print("*        *")
      print("*        *")
      print("**********")
```

The `drawBox` function works correctly. It draws the particular box that it was intended to draw, but it is not flexible, and as a result, it is not as useful as it could be.

Functions and Classes

```
def drawBox(width, height):
        # A box that is smaller than 2x2 cannot be
        # drawn by this function
        if width < 2 or height < 2:
          print("Error: The width or height is too small.")
          quit()
        # Draw the top of the box
        print("*" * width)
        # Draw the sides of the box
        for i in range(height - 2):
          print("*" + " " * (width - 2) + "*")
        # Draw the bottom of the box
        print("*" * width)

drawBox(15, 4)
```

Functions and Classes

Write a function drawBox that takes four arguments and draw a box.

```
def drawBox(width, height, outline="*", fill=" "):

drawBox(14, 5, "@", ".")

@@@@@@@@@@@@@@
@............@
@............@
@............@
@@@@@@@@@@@@@@
```

Functions and Classes

Define a function **`max_of_three()`** that takes three numbers as arguments and returns the largest of them.

Write a function **`max_in_list()`** that takes a list of numbers and returns the largest one.

Write a function called **`FtoC`** (ftoc.py) to convert Fahrenheit temperatures into Celsius. Make sure the program has a title comment. Test from the command window with:

```
FtoC(96)
```

Write a function (roll2dice.py) to roll 2 dice, returning two individual variables: d1 and d2. For example:

```
d1, d2 = roll2dice
```

Functions and Classes

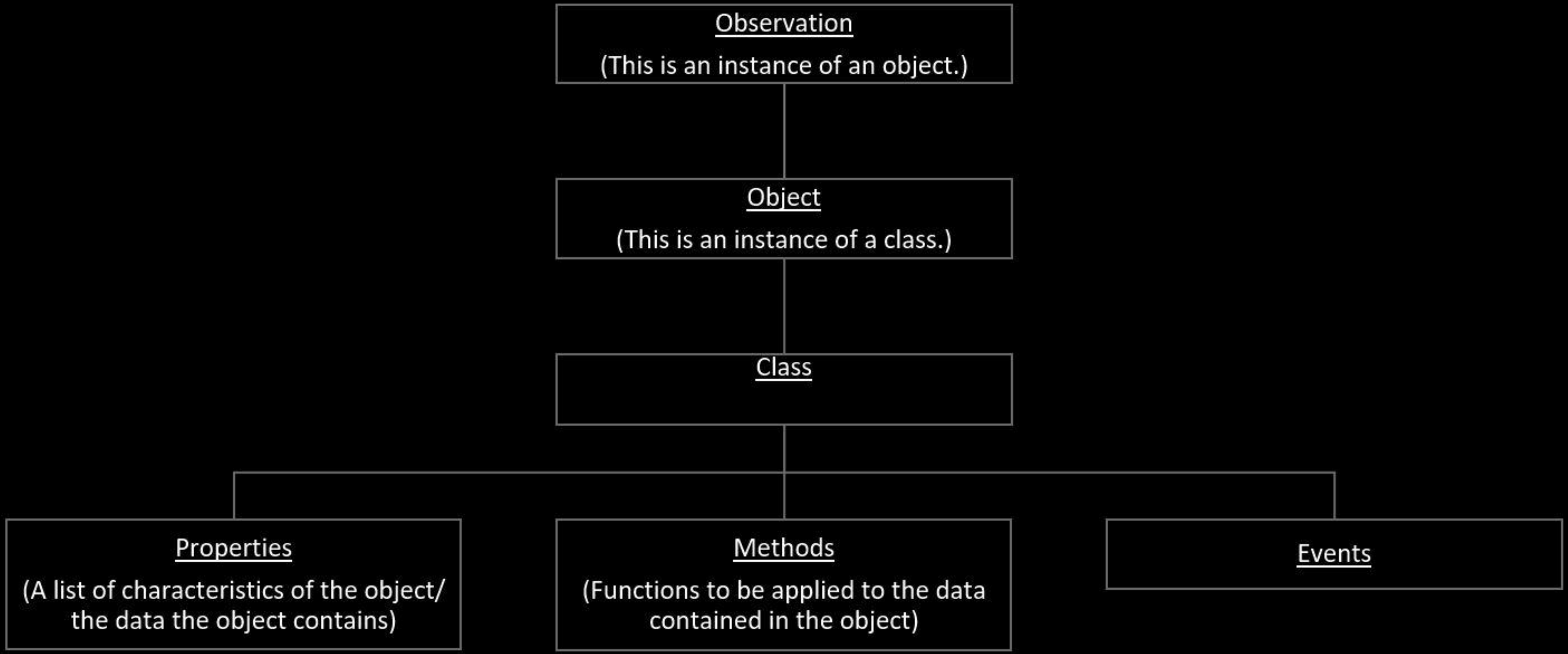
- Functions
- Classes

Functions and Classes

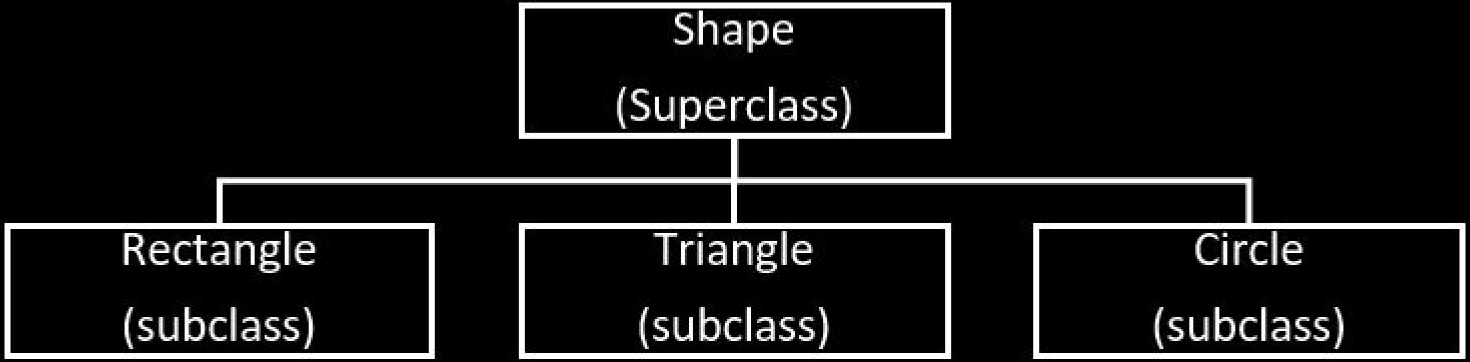
Simple programming tasks are easily implemented as simple functions, but as the magnitude and complexity of your tasks increase, functions become more complex and difficult to manage. As **`functions`** become too large, you might break them into smaller functions and pass data from one to the other. However, as the number of functions becomes large, designing and managing the data passed to functions becomes difficult and error prone. At this point, you should consider moving your programming tasks to **`object-oriented`** designs.

In the simplest sense, objects are **`data structures`** that encapsulate some internal state, which you access via its **`methods`**. When you invoke a method, it is the object that determines exactly what code to execute. In fact, two objects of the same class might execute different code paths for the same method invocation because their internal state is different. The internal workings of the object need not be of concern to your program — you simply use the interface the object provides.

Functions and Classes

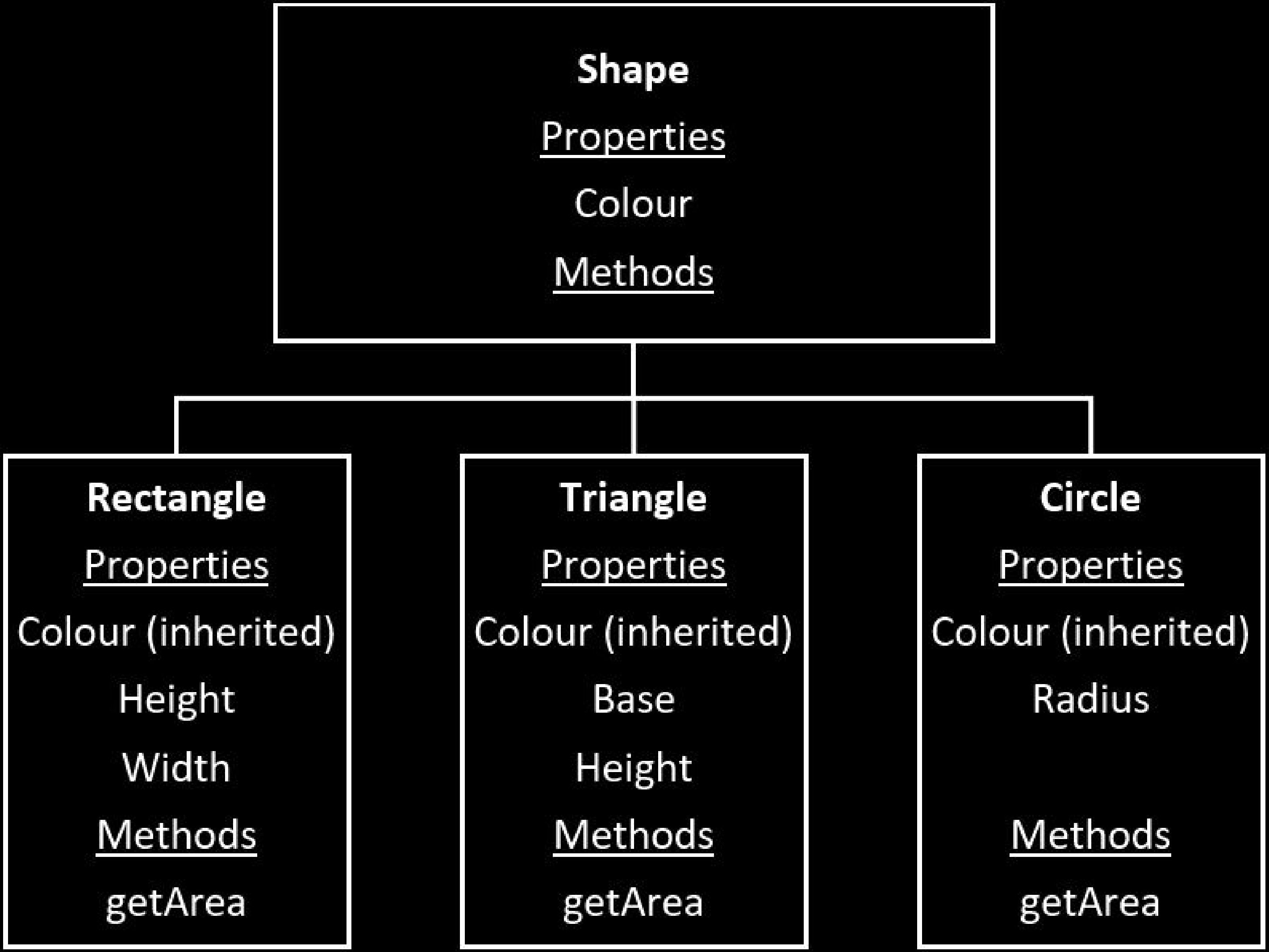

Observation
(This is an instance of an object.)
Object
(This is an instance of a class.)
Class
Properties
(A list of characteristics of the object/ the data the object contains)
Methods
(Functions to be applied to the data contained in the object)
Events

Functions and Classes



In the example of a rectangle class, a superclass namely Shape can be defined, where the superclass provides a more broadly defined class for a shape, and rectangle a more narrowly defined class for a specific type of shape. We can extend our example, by including the creation of Triangle and Circle classes, which are also subclasses of the superclass Shape.

Functions and Classes



Functions and Classes

Operator Overloading

Every use of a Matlab operator, such as + - .* * ./ .\ / \ .^ ^ < > <= >= == ~ ~= & | && || : ' .' [] [;] (). is actually short hand for a call to a named function like `plus()`, `minus()`, `times()`, `power()`, `lt()`, `eq()`, `not()`.

We can define custom behavior for any of these operators by witting class methods by the same name. Since class methods are dynamically dispatched, our own versions of these functions will execute when we use the corresponding operators with our objects. We could write our own `plus()` method in the date class, for example, to add dates together and then call the function with `d1 + d2`. Or, we could write our own `lt()` function, (for less than) to compare dates, calling it with `d1 < d2`. Such calls get converted automatically to `plus(d1,d2)` and `lt(d1,d2)`, and our own implementations of these functions are then invoked.

Functions and Classes

A class in Python is effectively a data type. All the data types built into Python are classes, and Python gives you powerful tools to manipulate every aspect of a class's behavior. You define a class with the class statement:

```
class MyClass:
      body
```

body is a list of Python statements, typically variable assignments and function definitions. No assignments or function definitions are required. The body can be just a single pass statement.

By convention, class identifiers are in CapCase — that is, the first letter of each component word is capitalized, to make them stand out.

Functions and Classes

Class instances can be used as structures or records. Unlike MATLAB, the fields of an instance don’t need to be declared ahead of time but can be created on the fly. The following short example defines a class called Circle, creates a Circle instance, assigns to the radius field of the circle, and then uses that field to calculate the circumference of the circle:

```
>>> class Circle:
...   pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>>
print('CemberinAlanı:',3.14*my_class.radius**2))
78.5
```

Functions and Classes

Like many other languages, the fields of an instance / structure are accessed and assigned to by using dot notation. You can initialize fields of an instance automatically by including an __init__ initialization method in the class body. This function is run every time an instance of the class is created, with that new instance as its first argument. The __init__ method is similar to a constructor, but it doesn't really construct anything—it initializes fields of the class. This example creates circles with a radius of 1 by default:

```
class Circle:
      def __init__(self):
            self.radius = 1

my_circle = Circle()
print(2 * 3.14 * my_circle.radius)
my_circle.radius = 5
print(2 * 3.14 * my_circle.radius)
```

Functions and Classes

Instance variables are the most basic feature of OOP. Take a look at the Circle class again:

```
class Circle:
      def __init__(self):
            self.radius = 1
```

radius is an instance variable of Circle instances. That is, each instance of the Circle class has its own copy of radius, and the value stored in that copy may be different from the values stored in the radius variable in other instances. In Python, you can create instance variables as necessary by assigning to a field of a class instance:

```
instance.variable = value
```

Functions and Classes

A METHOD is a function associated with a particular class. You've already seen the special __init__ method, which is called on a new instance when that instance is first created. In the following example, we define another method, area, for the Circle class, which can be used to calculate and return the area for any Circle instance. Like most user-defined methods, area is called with a method invocation syntax that resembles instance variable access:

```
class Circle:
    def __init__(self):
        self.radius = 1
    def area(self):
        return self.radius **2 * 3.14159
```

Functions and Classes

```
class Circle:
    def __init__(self):
        self.radius = 1
    def area(self):
        return self.radius **2 * 3.14159
```

Method invocation syntax consists of an instance, followed by a period, followed by the method to be invoked on the instance.

```
>>> my_circle = Circle()
>>> print(2 * 3.14 * my_circle.radius)
>>> my_circle.radius = 5
>>> print(2 * 3.14 * my_circle.radius)
>>> print(my_circle.area())
```

Write a Method that calculates the circumference of the circle

Functions and Classes

Methods can be invoked with arguments, if the method definitions accept those arguments. This version of Circle adds an argument to the __init__ method, so that we can create circles of a given radius without needing to set the radius after a circle is created:

local function variable

```
class Circle
      def __init__(self, radius):
            self.radius = radius
      def area(self):
            return self.radius ** 2 * 3.14159
```

Note the two uses of radius here. self.radius is the instance variable called radius. radius by itself is the local function variable called radius. The two aren't the same! In practice, we'd probably call the local function variable something like r or rad, to avoid any possibility of confusion.

Functions and Classes

All the standard Python function features—default argument values, extra arguments, keyword arguments, and so forth—can be used with methods. For example, we could have defined the first line of `__init__` to be

```
def __init__(self, radius=1):
```

Then, calls to circle would work with or without an extra argument; `Circle()` would return a circle of `radius 1`, and `Circle(3)` would return a circle of radius 3.

Functions and Classes

A class variable is a variable associated with a class, not an instance of a class, and is accessed by all instances of the class, in order to keep track of some class-level information, such as how many instances of the class have been created at any point in time. A class variable is created by an assignment in the class body, not in the `__init__` function; after it has been created, it can be seen by all instances of the class.

```
class Circle:
      pi = 3.14159
      def __init__(self, radius):
            self.radius = radius
      def area(self):
            return self.radius **2 * Circle.pi
```

Functions and Classes

Static methods even though no instance of that class has been created, although you can call them using a class instance. To create a static method, use the @staticmethod decorator.

```
"""circle module: contains the Circle class."""
class Circle:
    """Circle class"""
    all_circles = []
    pi = 3.14159

    def __init__(self, r=1):
        """Create a Circle with the given radius"""
        self.radius = r
        self.__class__.all_circles.append(self)

    def area(self):
        """determine the area of the Circle"""
        return self.__class__.pi * self.radius * self.radius

    @staticmethod
    def total_area():
        total = 0
        for c in Circle.all_circles:
            total = total + c.area()
        return total
```

Functions and Classes

```
>>> os.getcwd()
'C:\\Python33'
>>> os.chdir('F:\Lectures\Python')
>>> os.getcwd()
'F:\\Lectures\\Python'
>>> import circle
>>> c1 = circle.Circle(1)
>>> c2 = circle.Circle(2)
>>> c1.area()
3.14159
>>> c2.area()
12.56636
>>> c1.area() + c2.area()
15.70795
>>> circle.Circle.total_area()
15.70795
```

Functions and Classes

Class methods are similar to static methods in that they can be invoked before an object of the class has been instantiated or by using an instance of the class. But class methods are implicitly passed the class they belong to as their first parameter, so you can code them more simply.

```
"""circle module: contains the Circle class."""
class Circle:
        """Circle class"""
        all_circles = []
        pi = 3.14159
        def __init__(
                """Create a Circle with the given radius"""
                self.radius = r
                self.__class__.all_circles.append(self)
        def area(self):
                """determine the area of the Circle"""
                return self.__class__.pi * self.radius * self.radius
        @classmethod
        def total_area(cls):
                total = 0
                for c in cls.all_circles:
                        total = total + c.area()
                return total
```

The class parameter is traditionally `cls`

Functions and Classes

```
>>> import circle_cm
>>> c1 = circle_cm.Circle(1)
>>> c2 = circle_cm.Circle(2)
>>> circle_cm.Circle.total_area()
15.70795
>>> c2.radius = 3
>>> circle_cm.Circle.total_area()
31.4159
>>> c1.total_area()
31.4159
>>> c2.total_area()
31.4159
```

By using a class method instead of a static method, we don't have to hardcode the class name into total_area. That means any subclasses of Circle can still call `total_area` and refer to their own members, not those in `Circle`.

Functions and Classes

Inheritance in Python is easier and more flexible than inheritance in compiled languages such as Java and C++ because the dynamic nature of Python doesn't force as many restrictions on the language.

```
class Square:
      def __init__(self, side=1, x=0, y=0):
            self.side = side
            self.x = x
            self.y = y
class Circle:
      def __init__(self, radius=1, x=0, y=0):
            self.radius = radius
            self.x = x
            self.y = y
```

Functions and Classes

Instead of defining the x and y variables in each shape class, abstract them out into a general **Shape** class, and have each class defining an actual shape inherit from that general class.

```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    class Square(Shape):
        def __init__(self, side=1, x=0, y=0):
            super().__init__(x, y)
            self.side = side
    class Circle(Shape):
        def __init__(self, r=1, x=0, y=0):
            super().__init__(x, y)
            self.radius = r
```

Says Square inherits from Shape

Must call __init__ method of Shape

Functions and Classes

There are (generally) two requirements in using an inherited class in Python. The first requirement is defining the inheritance hierarchy, which you do by giving the classes inherited from, in parentheses, immediately after the name of the class being defined with the class keyword. In the previous code, `Circle` and `Square` both inherit from Shape. The second and more subtle element is the necessity to explicitly call the `__init__` method of inherited classes. Python doesn't automatically do this for you, but you can use the super function to have Python figure out which inherited class to use. This is accomplished in the example code by the `super().__init__(x,y)` lines.

Instead of using super, we could call Shape's `__init__` by explicitly naming the inherited class using `Shape.__init__(self, x, y)`, which would also call the `Shape` initialization function with the instance being initialized.

Functions and Classes

Inheritance comes into effect when you attempt to use a method that isn't defined in the base classes but is defined in the superclass. To see this, let's define another method in the Shape class called **move**, which will move a shape by a given displacement.

```
class Shape:
    def __init__ self, x, y):
        self.x = x
        self.y = y

    def move(self, delta_x, delta_y):
        self.x = self.x + delta_x
        self.y = self.y + delta_y
```

Functions and Classes

```
>>> ========================= RESTART =========================
>>> class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, delta_x, delta_y):
        self.x = self.x + delta_x
        self.y = self.y + delta_y

>>> class Circle(Shape):
        def __init__(self, r=1, x=0, y=0):
            super().__init__(x, y)
            self.radius = r

>>> c = Circle(1)
>>> c.move(3, 4)
>>> c.x
3
>>> c.y
4
```