Introduction to Python Programming (BPLCK105B)

Module 5

CHAPTER 1

CLASSES AND OBJECTS

1. Programmer-defined types

- We have used many of Python's built-in types; now we are going to define a new type. As anexample, we will create a type called Point that represents a point in two-dimensional space.
- In mathematical notation, points are often written in parentheses with a comma separating thecoordinates.
- For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and yunits up from the origin.
- There are several ways we might represent points in Python:
  - We could store the coordinates separately in two variables, x and y.
  - We could store the coordinates as elements in a list or tuple.
  - We could create a new type to represent points as objects.
- Creating a new type is more complicated than the other options, but it has advantages that willbe apparent soon.
- A programmer-defined type is also called a class. A class definition looks like this:

```
class Point:
"""Represents a point in 2-D space."""
```

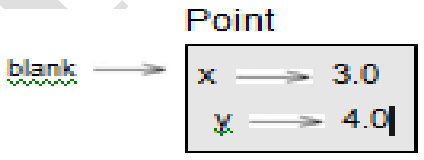


Figure 15.1: Object diagram.

- The header indicates that the new class is called Point. The body is a docstring that ex-plains what the class is for. You can define variable and methods inside a class definition, but we will get back to that later.
- Defining a class named Point creates a class object.

```
>>> Point
                <class '__main__.Point'>
```

- Because Point is defined at the top level, its "full name" is __main__.Point.
- The class object is like a factory for creating objects. To create a Point, you call Point as if itwere a function.

```
blank =
Point()
blank
<__main__.Point object at 0xb7e9d3ac>
```

- The return value is a reference to a Point object, which we assign to blank.
- Creating a new object is called instantiation, and the object is an instance of the class.
- When you print an instance, Python tells you what class it belongs to and where it is stored inmemory (the prefix 0x means that the following number is in hexadecimal)

Attributes

- You can assign values to an instance using dot notation:

```
blank.x = 3.0
blank.y = 4.0
```

- This syntax is similar to the syntax for selecting a variable from a module, such as math.pi orstring.whitespace .
- In this case, though, we are assigning values to named elements of an object. These elements arecalled attributes.
- A state diagram that shows an object and its attributes is called an object diagram; see Figure15.1.
- The variable blank refers to a Point object, which contains two attributes. Each attribute refers toa floating-point number.
- You can read the value of an attribute using the same syntax:

```
Blank.y
4.0
```

```
x=blank.x
x
3.0
```

- The expression blank.x means, "Go to the object blank refers to and get the value of x." In the example, we assign that value to a variable named x. There is no conflict between the variable xand the attribute x.

- You can use dot notation as part of any expression. For example:

```
'(%g, %g)' % (blank.x, blank.y)
 '(3.0, 4.0)'
```

```
distance = math.sqrt(blank.x**2 + blank.y**2)
distance
5.0
```

- You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
print('(%g, %g)' % (p.x, p.y))
```

- print_point takes a point as an argument and displays it in mathematical notation. To invoke it,you can pass blank as an argument:

```
print_point(blank) (3.0, 4.0)
```

- Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

2. Rectangles

- Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions.
- For example, imagine you are designing a class to represent rectangles. What attributes wouldyou use to specify the location and size of a rectangle?
- You can ignore angle; to keep things simple, assume that the rectangle is either vertical orhorizontal.
- There are at least two possibilities:
    1. You could specify one corner of the rectangle (or the center), the width, and the height.
    2. You could specify two opposing corners.
- At this point it is hard to say whether either is better than the other, so we'll implement the firstone, just as an example.
- Here is the class definition:

```
class Rectangle:
"""Represents a rectangle.
attributes: width, height, corner."""
```

- The docstring lists the attributes: width and height are numbers; corner is a Point object thatspecifies the lower-left corner.
- To represent a rectangle, you have to instantiate a Rectangle object and assign values to theattributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

- The expression box.corner.x means, "Go to the object box refers to and select the attributenamed corner; then go to that object and select the attribute named x."
- Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

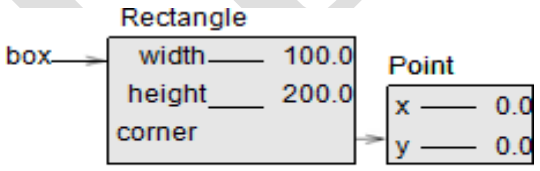


Figure 15.2: Object diagram.

3. Instances as return values

- Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle

```
def find_center(rect):
p = Point()
p.x = rect.corner.x + rect.width/2
p.y = rect.corner.y +
rect.height/2 return p
```

- Here is an example that passes box as an argument and assigns the resulting Point to center.

```
center = find_center(box)
print_point(center)
(50, 100)
```

4. Objects are mutable

- You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.height + 100
```

- You can also write functions that modify objects.
- For example, grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight,and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
rect.width += dwidth
rect.height += dheight
```

- Here is an example that demonstrates the effect:

```
box.width, box.height (150.0, 300.0)
grow_rectangle(box, 50, 100)
box.width, box.height
(200.0, 400.0)
```

- Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

5. Copying

- Aliasing can make a program difficult to read because changes in one place might haveunexpected effects in another place.
- It is hard to keep track of all the variables that might refer to a given object.
- Copying an object is often an alternative to aliasing. The copy module contains a function calledcopy that can duplicate any object:

```
p1 = Point()
p1.x = 3.0
p1.y = 4.0
```

```
import copy
p2 = copy.copy(p1)
```

- p1 and p2 contain the same data, but they are not the same Point.

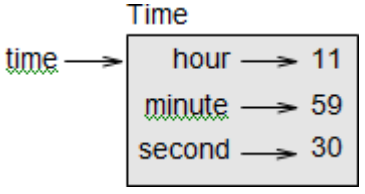
```
print_point(p1)
(3, 4)
```

```
print_point(p2)
(3, 4)
```

```
p1 is p2
False
```

```
p1 == p2
False
```

- The is operator indicates that p1 and p2 are not the same object, which is what we expected. Butyou might have expected == to yield True because these points contain the same data.
- In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence.



- That's because for programmer-defined types, Python doesn't know what should be consideredequivalent. At least, not yet

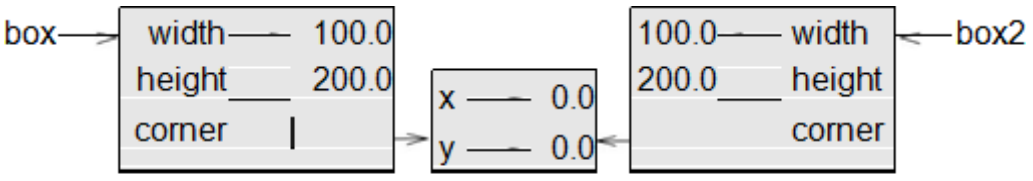


Figure 15.3: Object diagram.

- If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle objectbut not the embedded Point.

```
box2 = copy.copy(box)
box2 is box
False
box2.corner is box.corner
True
```

- Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.
- For most applications, this is not what you want.
- In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking move_rectangle on either would affect both! This behavior is confusing and error- prone.
- Fortunately, the copy module provides a method named deepcopy that copies not only the objectbut also the objects it refers to, and the objects they refer to, and so on. You will not be surprisedto learn that this operation is called a **deep copy.**

```
box3 = copy.deepcopy(box)
box3 is box
False
box3.corner is box.corner
False
```

- box3 and box are completely separate object

CHAPTER 02
CLASSES AND FUNCTIONS

1. Time

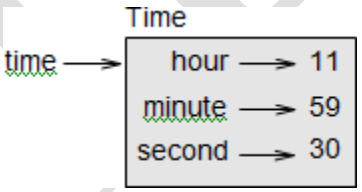
- As another example of a programmer-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second """
```

- We can create a new Time object and assign attributes for hours, minutes, and seconds:

```
time = Time( )
time.hour = 11
time.minute  = 59
time.second  = 30
```

- The state diagram for the Time object looks like Figure below.



2. Pure functions

- In the next few sections, we'll write two functions that add time values.
- They demonstrate two kinds of functions: pure functions and modifiers.
- They also demonstrate a development plan I'll call prototype and patch, which is a way of tackling acomplex problem by starting with a simple prototype and incrementally dealing with the complications.

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

- Here is a simple prototype of add_time:

- The function creates a new Time object, initializes its attributes, and returns a reference to the new object
- This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value
- To test this function, let us create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one hour 35 minutes.
- add_time figures out when the movie will be done

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

1. The result, 10:80:00 might not be what you were hoping for.
2. The problem is that this function does not deal with cases where the number of seconds orminutes adds up to more than sixty.

- When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.
- Here's an improved version:

```
def add_time(t1, t2):
sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

if sum.second >= 60:
    sum.second -= 60
    sum.minute += 1

if sum.minute >= 60:
    sum.minute -= 60
    sum.hour += 1

return sum
```

3. Modifiers

- Sometimes it is useful for a function to modify the objects it gets as parameters
- In that case, the changes are visible to the caller. Functions that work this way are called modifiers.
- increment, which adds a given number of seconds to a Time object, can be written naturally as amodifier. Here is a rough draft:

```
def increment(time, seconds):
time.second += seconds

if time.second >= 60:
    time.second -= 60
    time.minute += 1

if time.minute >= 60:
    time.minute -= 60
    time.hour += 1
```

- The first line performs the basic operation; the remainder deals with the special cases we saw before.
- Is this function correct? What happens if seconds is much greater than sixty?
- In that case, it is not enough to carry once; we have to keep doing it until time.second is less thansixty.
- One solution is to replace the if statements with while statements. That would make the functioncorrect, but not very efficient.
- Anything that can be done with modifiers can also be done with pure functions.

4. Prototyping versus planning

- The development plan, i.e. demonstrating is called "prototype and patch". For each function, we wrotea prototype that performed the basic calculation and then tested it, patching errors along the way.
- This approach can be effective, especially if you don't yet have a deep understanding of theproblem.
- But incremental corrections can generate code that is unnecessarily complicated—since it deals withmany special cases—and unreliable—since it is hard to know if you have found all the errors.

- Here is a function that converts Times to integers:

```
def time_to_int(time):
minutes = time.hour * 60 +time.minute
seconds = minutes * 60 + time.second return seconds
```

- And here is a function that converts an integer to a Time (recall that div mod divides the firstargument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
 time = Time()
minutes, time.second = divmod(seconds, 60)
time.hour, time.minute = divmod(minutes, 60)
return time
```

- Once we are convinced they are correct, you can use them to rewrite:

```
def add_time(t1, t2):
seconds = time_to_int(t1) + time_to_int(t2)return
int_to_time(seconds)
```

- This version is shorter than the original, and easier to verify

CHAPTER 03
CLASSES AND METHODS

Object-Oriented Features

- Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:
  - Programs include class and method definitions.
  - Most of the computation is expressed in terms of operations on objects.
  - Objects often represent things in the real world, and methods often correspond to the ways things in thereal world interact.
- A method is a function that is associated with a particular class.
- Methods are semantically the same as functions, but there are two syntactic differences:
  - Methods are defined inside a class definition in order to make the relationship between the class and themethod explicit.
  - The syntax for invoking a method is different from the syntax for calling a function

2. Printing Objects

- We already defined- a class named and also wrote a function named print_time:

```
class Time:
"""Represents the time of day."""
def print_time(time):
print('%.2d:%.2d:%.2d'  %  (time.hour,  time.minute,  time.second))
```

- To call this function, we have to pass a Time object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>>> print_time(start)
09:45:00
```

- To make print_time a method, all we have to do is move the function definition inside the class definition.Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

- Now there are two ways to call print_time. The first (and less common) way is to use functionsyntax:

```
>>>Time.print_time(start)
09:45:00
```

- In this use of dot notation, Time is the name of the class, and
- print_time is the name of themethod. start is passed parameter.
- The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

- In this use of dot notation, print_time is the name of the method (again), and start is the object the methodis invoked on, which is called the subject.

- Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is whatthe method is about.
- Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.
- By convention, the first parameter of a method is called self, so it would be more common to writeprint_time like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

- The reason for this convention is an implicit metaphor:
  - The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, "Hey print_time! Here's an object for you to print."
  - In object-oriented programming, the objects are the active agents. method invocation like start.print_time() says "Hey start! Please print yourself."

3, Another Example

- Here's a version of increment rewritten as a method:

```
# inside class Time:

def increment(self, seconds):
    seconds += self.time_to_int()
    return int_to_time(seconds)
```

- This version assumes that time_to_int is written as a method. Also, note that it is a pure function, not amodifier.
- Here's how you would invoke increment:

```
>>> start.print_time()

09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

- The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to thesecond parameter, seconds.

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

- This mechanism can be confusing, especially if you make an error. For example, if you invokeincrement with two arguments, you get:
- The error message is initially confusing, because there are only two argument in parentheses. But thesubject is also considered an argument, so all together that's three.
- By the way, a positional argument is an argument that doesn't have a parameter name; that is, it is not akeyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

- parrot and cage are positional, and dead is a keyword argument.

4. A More Complicated Example

- Rewriting is_after is slightly more complicated because it takes two Time objects as parameters.
- In this case it is conventional to name the first parameter self and the second parameter other:

```
# inside class Time:

def is_after(self, other):
return self.time_to_int() > other.time_to_int()
```

```
>>> end.is_after(start)
True
```

- To use this method, you have to invoke it on one object and pass the other as an argument:

5. The init Method

- The init method (short for "initialization") is a special method that gets invoked when an object isinstantiated.

- Its full name is __init__(two underscore characters, followed by init, and then two more underscores).
- An init method for the Time class might look like this:

```
# inside class Time:
def __init__(self, hour=0,
    minute=0, second=0):
    self.hour = hour
    self.minute = minute
    self.second = second
```

- It is common for the parameters of __init to have the same names as the attributes.The statement

```
self.hour = hour
```

- stores the value of the parameter hour as an attribute of self.
- The parameters are optional, so if you call Time with no arguments, you get the default values:

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

- If we provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

- If we provide two arguments, they override hour and minute.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

- And if we provide three arguments, they override all three default values

6. The _str_ Method

- str___is a special method, like _init , that is supposed to return a string representa- tion of anobject.
- For example, here is a str method for Time object

```
# inside class Time:
def__str_(self):
        return  '%.2d:%.2d:%.2d' %  (self.hour,  self.minute,  self.second)
```

- When you print an object, Python invokes the str method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

7. Operator Overloading

- By defining other special methods, you can specify the behavior of operators on programmer-defined types.
- For example, if we define a method named _add_____for the Time class, you can use the + operator on Timeobjects.
- Here is what the definition might look like:

```
def _add_(self,other):
    seconds=self.time_to_int()+other.time_to_int()
      return int_to_time(seconds)
```

- And here is how we could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
```

```
>>> print(start + duration)
11:20:00
```

- When you apply the + operator to Time objects, Python invokes __add__.
- When you print the result, Python invokes __str__. So there is a lot happening behind the scenes!
- Changing the behavior of an operator so that it works with programmer-defined types is called operatoroverloading.
- For every operator in Python there is a corresponding special method, like __add__.

8. Type-Based Dispatch

- The following is the version of _add_ that checks the type of other and invokes either add_time orincrement:

```
def__add__(self,other):
    if isintance(other, Time):
        return self.add_time(other)
    else:
        return self.increment(other)

def add_time(self, other):
    seconds = self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)

def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

- The built-in function isinstance takes a value and a class object, and returns True if the value is an instanceof the class.
- If other is a Time object, _add____invokes add_time. Otherwise it assumes that the parameter is a numberand invokes increment.

- This operation is called a type-based dispatch because it dispatches the computation to different methods based on the type of the arguments
- Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

- Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

- The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add aTime object, and it doesn't know how.
- But there is a clever solution for this problem: the special method _____radd_, which stands for "right-sideadd".
- This method is invoked when a Time object appears on the right side of the + operator. Here's thedefinition:

```
# inside class Time:
def __radd__(self, other):
    return self.__add__(other)


>>> print(1337 + start)
10:07:17
```

8. Polymorphism

- Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often youcan avoid it by writing functions that work correctly for arguments with different types.
- Many of the functions we wrote for strings also work for other sequence types. For example, we usedhistogram to count the number of times each letter appears in a word.

```
def
histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

- This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, sothey can be used as keys in d:

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

- Functions that work with several types are called polymorphic. Polymorphism can facilitate code reuse.
- For example, the built-in function sum, which adds the elements of a sequence, works as long as theelements of the sequence support addition.

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 31)
>>> t3 = Time(7, 37)
>>> total = sum(t1, t2, t3)
>>> print(total)
23:01:00
```

- In general, if all of the operations inside a function work with a given type, the function works with thattype.
- The best kind of polymorphism is the unintentional kind, where you discover that a function you alreadywrote can be applied to a type you never planned for.