9.1 Classes: Introduction

A large program thought of as thousands of variables and functions is hard to understand. A higher level approach is needed to organize a program in a more understandable way.

In the physical world, we are surrounded by basic items made from wood, metal, plastic, etc. But to keep the world understandable, we think at a higher level, in terms of *objects* like an oven. The oven allows us to perform a few specific operations, like put an item in the oven, or set the temperature.

Thinking in terms of objects can be powerful when designing programs. Suppose a program should record time and distance for various runners, such as a runner ran 1.5 miles in 500 seconds, and should compute speed. A programmer might think of an "object" called RunnerInfo. The RunnerInfo object supports operations like setting distance, setting time, and computing speed. In a program, an ***object*** consists of some internal data items plus operations that can be performed on that data.

PARTICIPATION ACTIVITY 9.1.1: Grouping variables and functions into objects keeps programs understandable.

Animation captions:

1. A program with many variables and functions can be hard to understand
2. Grouping related items into objects keeps programs understandable

Python automatically creates certain objects, known as *built-ins*, like ints, strings, and functions. Take for example a string object created with `mystr = 'Hello!'`. The value of the string 'Hello!' is one part of the object, as are methods like mystr.isdigit() and mystr.lower().

Creating a program as a collection of objects can lead to a more understandable, manageable, and properly-executing program.

PARTICIPATION ACTIVITY 9.1.2: Objects.

Some of the variables and functions for a used-car inventory program are to be grouped into an object type named CarOnLot. Select True if the item should become part of the CarOnLot object type, and False otherwise.

1) car_sticker_price

- ○ True

- False

2) todays_temperature
   - True
   - False

3) days_on_lot
   - True
   - False

4) orig_purchase_price
   - True
   - False

5) num_sales_people
   - True
   - False

6) increment_car_days_on_lot()
   - True
   - False

7) decrease_sticker_price()
   - True
   - False

8) determine_top_salesman()
   - True
   - False

©zyBooks 11/11/19 20:35 569464
Diep Vu
UNIONCSC103OrhanFall2019

9.2 Classes: Grouping data

Multiple variables are frequently closely related and should thus be treated as one variable with multiple parts. For example, two variables called hours and minutes might be grouped together in a single variable called time. The ***class*** keyword can be used to create a user-defined type of object containing groups of related variables and functions.

Construct 9.2.1: The class keyword.

```
class ClassName:
    # Statement-1
    # Statement-2
    # ...
    # Statement-N
```


A class defines a new type that can group data and functions to form an object. The object maintains a set of ***attributes*** that determines the data and behavior of the class. For example, the following code defines a new class containing two attributes, hours and minutes, whose values are initially 0:

Figure 9.2.1: Defining a new class object with two data attributes.

```
class Time:
    """ A class that represents a time of day """
    def __init__(self):
        self.hours = 0
        self.minutes = 0
```

The programmer can then use instantiation to define a new Time class variable and access that variable's attributes. An ***instantiation*** operation is performed by "calling" the class, using parentheses like a function call as in `my_time = Time()`. An instantiation operation creates an ***instance***, which is an individual object of the given class. An instantiation operation automatically calls the __init__ method defined in the class definition. A ***method*** is a function defined within a class. The ***__init__*** method, commonly known as a ***constructor***, is responsible for setting up the initial state of the new instance. In the example above, the __init__ method creates two new attributes, hours and minutes, and assigns default values of 0.

The __init__ method has a single parameter "self", that automatically references the instance being created. A programmer writes an expression such as `self.hours = 0` within the __init__ method to create a new attribute hours.

Figure 9.2.2: Using instantiation to create a variable using the Time class.

```
7 hours and 15 minutes
```

```
class Time:
    """ A class that represents a time of day """
    def __init__(self):
        self.hours = 0
        self.minutes = 0


my_time = Time()
my_time.hours = 7
my_time.minutes = 15

print('%d hours' % my_time.hours, end=' ')
print('and %d minutes' % my_time.minutes)
```


Attributes can be accessed using the ***attribute reference operator*** "." (sometimes called the ***member operator*** or ***dot notation***).

PARTICIPATION ACTIVITY 9.2.1: Using classes and attribute reference.

Animation captions:

1. The Time() function creates a time object, time1, and initializes time1.hours and time1.minutes to 0.
2. Attributes can be accessed using the "." attribute reference operator.

A programmer can create multiple instances of a class in a program, with each instance having different attribute values.

Figure 9.2.3: Multiple instances of a class.

```
class Time:
    """ A class that represents a time of day """
    def __init__(self):
        self.hours = 0
        self.minutes = 0


time1 = Time()  # Create an instance of the Time class called
time1
time1.hours = 7
time1.minutes = 30

time2 = Time()  # Create a second instance called time2
time2.hours = 12
time2.minutes = 45
```

```
7 hours and 30 minutes
12 hours and 45
minutes
```

```
print('%d hours and %d minutes' % (time1.hours, time1.minutes))
print('%d hours and %d minutes' % (time2.hours, time2.minutes))
```

Good practice is to use initial capitalization for class names. Thus, appropriate names might include LunchMenu, CoinAmounts, or PDFFileContents.

PARTICIPATION ACTIVITY 9.2.2: Class terms.

class **self** **instance** **__init__** **attribute**

	A name following a "." symbol.
	A method parameter that refers to the class instance.
	An instantiation of a class.
	A constructor method that initializes a class instance.
	A group of related variables and functions.

Reset

PARTICIPATION ACTIVITY 9.2.3: Classes.

1) A class can be used to group related variables together.

- True
- False

2) The __init__ method is called automatically.

- True
-

False

3) Following the statement `t = Time()`, t references an instance of the Time class.

- True
- False

PARTICIPATION ACTIVITY 9.2.4: Classes.

1) Given the above definition of the Time class, what is the value of `time1.hours` after the following code executes?

```
time1 = Time()
```

Check Show answer

2) Given the above definition of the Time class, what is the value of `time1.hours` after the following code executes?

```
time1 = Time()
time1.hours = 7
```

Check Show answer

3) Given the above definition of the Time class, what is the value of `time2.hours` after the following code executes?

```
time1 = Time()
time1.hours = 7

time2 = time1
```

Check Show answer

CHALLENGE ACTIVITY 9.2.1: Declaring a class.

Declare a class named PatientData that contains two attributes named height_inches and weight_pounds.

Sample output for the given program with inputs: 63 115


```
Patient data (before): 0 in, 0 lbs
Patient data (after): 63 in, 115 lbs
```

```
1
2 ''' Your solution goes here '''
3
4 patient = PatientData()
5 print('Patient data (before):', end=' ')
6 print(patient.height_inches, 'in,', end=' ')
7 print(patient.weight_pounds, 'lbs')
8
9
10 patient.height_inches = int(input())
11 patient.weight_pounds = int(input())
12
13 print('Patient data (after):', end=' ')
14 print(patient.height_inches, 'in,', end=' ')
15 print(patient.weight_pounds, 'lbs')
```

Run

CHALLENGE ACTIVITY 9.2.2: Access a class' attributes.

Print the attributes of the InventoryTag object red_sweater.

Sample output for the given program with inputs: 314 500


```
ID: 314
Qty: 500
```

```
1 class InventoryTag:
2     def __init__(self):
```

```
        self.item_id = 0
        self.quantity_remaining = 0

red_sweater = InventoryTag()
red_sweater.item_id = int(input())
red_sweater.quantity_remaining = int(input())

''' Your solution goes here '''
```


Run

9.3 Instance methods

A function defined within a class is known as an ***instance method***. The Python object that encapsulates the method's code is thus called a ***method object*** (as opposed to a pure function object). A method object is an attribute of a class, and can be referenced using dot notation. The following example illustrates:

Figure 9.3.1: A class definition may include user-defined functions.

```
class Time:
    def __init__(self):
        self.hours = 0
        self.minutes = 0

    def print_time(self):
        print('Hours:', self.hours, end=' ')
        print('Minutes:', self.minutes)


time1 = Time()
time1.hours = 7
time1.minutes = 15
time1.print_time()
```

```
Hours: 7 Minutes: 15
```

The definition of print_time() has a parameter "self" that provides a reference to the class instance. In the program above "self" is bound to time1 when time1.print_time() is called. A programmer does not specify an argument for "self" when calling the function; (the argument list in time1.print_time() is empty.) The method's code can use "self" to access other attributes or methods of the instance; for example, the print_time method uses "self.hours" and "self.minutes" to get the value of the time1 instance data attributes.

PARTICIPATION ACTIVITY 9.3.1: Methods.

Consider the following class definition:

```
class Animal:
    def __init__(self):
        # ...

    def noise(self, sound):
        # ...
```

1) Write a statement that creates an instance of Animal called "cat".

Check Show answer

2) Write a statement that calls the noise method of the cat instance with the argument "meow".

Check Show answer

3) What should the first item in the parameter list of every method be?

Check Show answer

zyDE 9.3.1: Adding methods to a class.

Add a method calculate_pay() to the Employee class. The method should return th
pay the employee by multiplying the employee's wage and number of hours worke

Load default template... Run

```
1
2  class Employee:
3      def __init__(self):
4          self.wage = 0
5          self.hours_worked = 0
6
7  #   def ... Add new method here ...
8  #         ...
9
10 alice = Employee()
11 alice.wage = 9.25
12 alice.hours_worked = 35
13 print('Alice:\n Net pay: %f' % alice.calculat
14
15 bob = Employee()
16 bob.wage = 11.50
17 bob.hours_worked = 20
18 print('Bob:\n Net pay: %f' % bob.calculate_pa
19
```


Note that __init__ is also a method of the Time class; however, __init__ is a ***special method name***, indicating that the method implements some special behavior of the class. In the case of __init__, that special behavior is the initialization of new instances. Special methods can always be identified by the double underscores __ that appear before and after an identifier. Good practice is to avoid using double underscores in identifiers to prevent collisions with special method names, which the Python interpreter recognizes and may handle differently. Later sections discuss special method names in more detail.

A common error for new programmers is to omit the self argument as the first parameter of a method. In such cases, calling the method produces an error indicating too many arguments to the method were given by the programmer, because a method call automatically inserts an instance reference as the first argument:

Figure 9.3.2: Accidentally forgetting the self parameter of a method generates an error when calling the method.

```
class Employee:
    def __init__(self):
        self.wage = 0
        self.hours_worked = 0

    def calculate_pay():  # Programmer forgot self parameter
        return self.wage * self.hours_worked


alice = Employee()
alice.wage = 9.25
alice.hours_worked = 35
print('Alice earned %f' % alice.calculate_pay())
```



```
Traceback (most recent call last):
```

```
  File "<stdin>", line 13, in <module>
TypeError: calculate_pay() takes 0 positional arguments but 1 was given
```

PARTICIPATION ACTIVITY 9.3.2: Method definitions.

1) Write the definition of a method "add" that has no parameters.

```
class MyClass:
    # ...
    def 
:
        return self.x +
self.y
```

Check Show answer

2) Write the definition of a method "print_time" that has a single parameter "gmt".

```
class Time:
    # ...
    def 
:
        if gmt:
            print('Time is:
%d:%d GMT'
                  %
(self.hours-8,
self.minutes))
        else:
            print('Time is:
%d:%d'
                  %
(self.hours, self.minutes))
```


Check Show answer

CHALLENGE ACTIVITY 9.3.1: Creating a method object.

Define the method object inc_num_kids() for PersonInfo. inc_num_kids increments the member data num_kids

member data num_kids.

Sample output for the given program with one call to inc_num_kids():

```
Kids: 0
New baby, kids now: 1
```



```
class PersonInfo:
    def __init__(self):
        self.num_kids = 0

    # FIXME: Write inc_num_kids(self)

    ''' Your solution goes here '''

person1 = PersonInfo()

print('Kids:', person1.num_kids)
person1.inc_num_kids()
print('New baby, kids now:', person1.num_kids)
```

Run

9.4 Class and instance object types

A program with user-defined classes contains two additional types of objects: class objects and instance objects. A ***class object*** acts as a *factory* that creates instance objects. When created by the class object, an _instance object_ is initialized via the __init__ method. The following tool demonstrates how the __init__ method of the Time class object is used to initialize two new Time instance objects:

PARTICIPATION ACTIVITY 9.4.1: Class Time's init method initializes two new Time instance objects.

```
1  class Time:
       def __init__(self):
```

```
2
3           self.hours = 0
4           self.minutes = 0
5
6   time1 = Time()
7   time1.hours = 5
8   time1.minutes = 30
9
10  time2 = Time()
11  time2.hours = 7
12  time2.minutes = 45
```


<< First < Back Step 1 of 15 Forward > L

line that has just executed
next line to execute

Frames Objects

Class and instance objects are namespaces used to group data and functions together.

- A ***class attribute*** is shared amongst all of the instances of that class. Class attributes are defined within the scope of a class.

Figure 9.4.1: A class attribute is shared between all instances of that class.

```
class MarathonRunner:
    race_distance = 42.195  # Marathon distance in Kilometers

    def __init__(self):
        # ...

    def get_speed(self):
        # ...

runner1 = MarathonRunner()
runner2 = MarathonRunner()

print(MarathonRunner.race_distance)  # Look in class namespace
print(runner1.race_distance)  # Look in instance namespace
print(runner2.race_distance)
```

```
42.195
42.195
42.195
```


- An ***instance attribute*** can be unique to each instance.

Figure 9.4.2: An instance attribute can be different between instances of a class.

```
class MarathonRunner:
    race_distance = 42.195  # Marathon distance in Kilometers

    def __init__(self):
        self.speed = 0
        # ...

    def get_speed(self):
        # ...

runner1 = MarathonRunner()
runner1.speed = 7.5

runner2 = MarathonRunner()
runner2.speed = 8.0

print('Runner 1 speed:', runner1.speed)
print('Runner 2 speed:', runner2.speed)
```

```
Runner 1 speed: 7.5
Runner 2 speed: 8.0
```


Instance attributes are created using dot notation, as in `self.speed = 7.5` within a method, or `runner1.speed = 7.5` from outside of the class' scope.

Instance and class namespaces are linked to each other. If a name is not found in an instance namespace, then the class namespace is searched.

PARTICIPATION ACTIVITY 9.4.2: Class and instance namespaces.

Animation captions:

1. Class namespace contains all class attributes
2. Instance attributes added to each instance's namespace only
3. Using dot notation initiates a search that first looks in the instance namespace, then the class namespace.

Besides methods, typical class attributes are constants required only by instances of the class. Placing such constants in the class' scope helps to reduce possible collisions with other variables or functions in the global scope.

Figure 9.4.3: Changing the gmt_offset class attribute affects behavior of all instances.

```
class Time:
    gmt_offset = 0  # Class attribute. Changing alters
print_time output

    def __init__(self):  # Methods are a class attribute too
        self.hours = 0  # Instance attribute
        self.minutes = 0  # Instance attribute

    def print_time(self):  # Methods are a class attribute
too
        offset_hours = self.hours + self.gmt_offset  # Local
variable
        print('Time -- %d:%d' % (offset_hours,
self.minutes))


time1 = Time()
time1.hours = 10
time1.minutes = 15

time2 = Time()
time2.hours = 12
time2.minutes = 45

print ('Greenwich Mean Time (GMT):')
time1.print_time()
time2.print_time()

Time.gmt_offset = -8  # Change to PST time (-8 GMT)

print('\nPacific Standard Time (PST):')
time1.print_time()
time2.print_time()
```

```
Greenwich Mean Time
(GMT):
Time -- 10:15
Time -- 12:45

Pacific Standard Time
(PST):
Time -- 2:15
Time -- 4:45
```


PARTICIPATION ACTIVITY 9.4.3: Class and instance objects.

Class object **Instance methods** **Class attribute** **Instance object**

Instance attribute

	A factory for creating new class instances.
	Represents a single instance of a class.

	Functions that are also class attributes.
	A variable that exists in a single instance.
	A variable shared with all instances of a class.

Reset

Note that even though class and instance attributes have unique namespaces, a programmer can use the "self" parameter to reference either type. For example, `self.hours` finds the instance attribute hours, and `self.gmt_offset` finds the class attribute gmt_offset. Thus, if a class and instance both have an attribute with the same name, the instance attribute will always be referenced. Good practice is to avoid name collisions between class and instance attributes.

PARTICIPATION ACTIVITY 9.4.4: Identifying class and instance attributes.

1) What type of attribute is number?

```
class PhoneNumber:
    def __init__(self):
        self.number = '805-555-2231'
```

- Class attribute
- Instance attribute

2) What type of attribute is number?

```
class PhoneNumber:
    def __init__(self):
        self.number = None

garrett = PhoneNumber()
garrett.number = '805-555-2231'
```

- Class attribute
- Instance attribute

3) What type of attribute is area_code?

```
class PhoneNumber:
    area_code = '805'
    def __init__(self):
        self.number = '555-2231'
```

- Class attribute
- Instance attribute

9.5 Class example: Seat reservation system

zyDE 9.5.1: Using classes to implement an airline seat reservation system.

The following example implements an airline seat reservations system using class instance data members and methods. Ultimately, the use of classes should lead to that are easier to understand and maintain.

Load default

```
1  class Seat:
2      def __init__(self):
3          self.first_name = ''
4          self.last_name = ''
5          self.paid = 0.0
6
7      def reserve(self, fn, ln, pd):
8          self.first_name = fn
9          self.last_name = ln
10         self.paid = pd
11
12     def make_empty(self):
13         self.first_name = ''
14         self.last_name = ''
15         self.paid = 0.0
16
17     def is_empty(self):
18         return self.first_name == ''
19
20     def print_seat(self):
21         print('%s %s  Paid: %.2f' % (self.first_name, self.last_name, self.paid))
```

```
r
1
Hank
```

Run

9.6 Class constructors

A class instance is commonly initialized to a specific state. The __init__ method constructor can be customized with additional parameters, as shown below:

Figure 9.6.1: Adding parameters to a constructor.

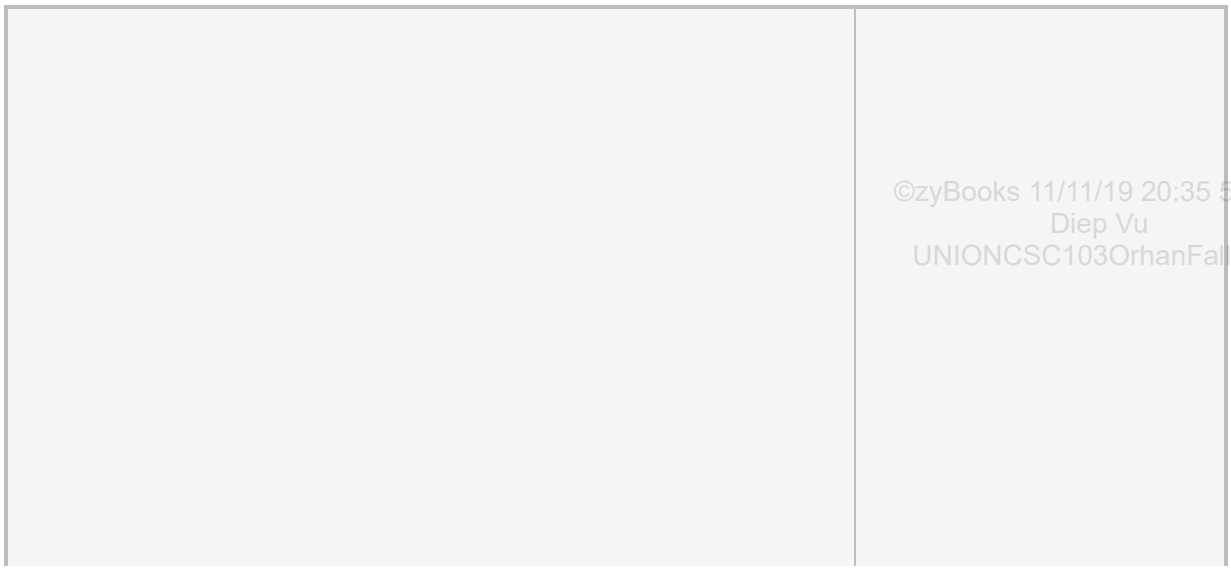
```
class RaceTime:
    def __init__(self, start_time, end_time, distance):
        """
         start_time: Race start time. String w/ format 'hours:minutes'.
         end_time: Race end time. String w/ format 'hours:minutes'.
         distance: Distance of race in miles.
        """
    # ...

# The race times of marathon contestants
time_jason = RaceTime('3:15', '7:45', 26.21875)
time_bobby = RaceTime('3:15', '6:30', 26.21875)
```

The above constructor has three parameters, *start_time*, *end_time*, and *distance*. When instantiating a new instance of RaceTime, arguments must be passed to the constructor, e.g., `RaceTime('3:15', '7:45', 26.21875)`.

Consider the example below, which fully implements the RaceTime class, adding methods to print the time taken to complete the race and average pace.

Figure 9.6.2: Additional parameters can be added to a class constructor.



```
class RaceTime:

    def __init__(self, start_hrs, start_mins, end_hrs,
end_mins, dist):
        self.start_hrs = start_hrs
        self.start_mins = start_mins
        self.end_hrs = end_hrs
        self.end_mins = end_mins
        self.distance = dist

    def print_time(self):
        if self.end_mins >= self.start_mins:
            minutes = self.end_mins - self.start_mins
            hours = self.end_hrs - self.start_hrs
        else:
            minutes = 60 - self.start_mins + self.end_mins
            hours = self.end_hrs - self.start_hrs - 1

        print('Time to complete race: %d:%d' % (hours,
minutes))

    def print_pace(self):
        if self.end_mins >= self.start_mins:
            minutes = self.end_mins - self.start_mins
            hours = self.end_hrs - self.start_hrs
        else:
            minutes = 60 - self.start_mins + self.end_mins
            hours = self.end_hrs - self.start_hrs - 1

        total_minutes = hours*60 + minutes
        print('Avg pace (mins/mile): %.2f' % (total_minutes
/ self.distance))

distance = 5.0

start_hrs = int(input('Enter starting time hours: '))
start_mins = int(input('Enter starting time minutes: '))
end_hrs = int(input('Enter ending time hours: '))
end_mins = int(input('Enter ending time minutes: '))

race_time = RaceTime(start_hrs, start_mins, end_hrs,
end_mins, distance)

race_time.print_time()
race_time.print_pace()
```

```
Enter starting time
hours: 5
Enter starting time
minutes: 30
Enter ending time
hours: 7
Enter ending time
minutes: 00
Time to complete race:
1:30
Avg pace (mins/mile):
18.00

...

Enter starting time
hours: 5
Enter starting time
minutes: 30
Enter ending time
hours: 6
Enter ending time
minutes: 24
Time to complete race:
0:54
Avg pace (mins/mile):
10.80
```


The arguments passed into the constructor are saved as instance attributes using the automatically added "self" parameter and dot notation, as in `self.distance = distance`. Creation of such instance attributes allows methods to later access the values passed as arguments; for example, print_time() uses self.start and self.end, and print_pace() uses self.distance.

PARTICIPATION ACTIVITY 9.6.1: Method parameters.

1) Write the definition of an __init__ method that requires the parameters x and y.

Check Show answer

2) Complete the statement to create a new instance of Widget with p1=15 and p2=5.

```
class Widget:
    def __init__(self, p1,
p2):
        # ...

widg =
```

Check Show answer

Constructor parameters can have default values like any other function, using the name=value syntax. Default parameter values may indicate the default state of an instance. A programmer might then use constructor arguments only to modify the default state if necessary. For example, the Employee class constructor in the program below uses default values that represent a typical new employee's wage and scheduled hours per week.

Figure 9.6.3: Constructor default parameters.

```
class Employee:
    def __init__(self, name, wage=8.25, hours=20):
        """
        Default employee is part time (20 hours/week)
        and earns minimum wage
        """
        self.name = name
        self.wage = wage
        self.hours = hours

    # ...


todd = Employee('Todd')  # Typical part-time employee
jason = Employee('Jason')  # Typical part-time employee
tricia = Employee('Tricia', wage=12.50, hours=40)  #
Manager employee

employees = [todd, jason, tricia]

for e in employees:
    print ('%s earns %.2f per week' % (e.name,
e.wage*e.hours))
```

```
Todd earns 165.00 per
week
Jason earns 165.00 per
week
Tricia earns 500.00 per
week
```

Similar to calling functions, default parameter values can be mixed with positional and name-mapped arguments in an instantiation operation. Arguments without default values are required, must come first, and must be in order. The following arguments with default values are optional, and can appear in any order.

PARTICIPATION ACTIVITY 9.6.2: Default constructor parameters.

Consider the class definition below. Match the instantiations of Student to the matching list of attributes.

```
class Student:
    def __init__(self, name, grade=9, honors=False, athletics=False):
        self.name = name
        self.grade = grade
        self.honors = honors
        self.athletics = athletics

    # ...
```

Student('Tommy')

Student('Johnny', grade=11, athletics=False)

Student('Johnny', grade=11, honors=True)

Student('Tommy', grade=9, honors=True, athletics=True)

	self.name: 'Tommy', self.grade: 9, self.honors: False, self.athletics: False
	self.name: 'Tommy', self.grade: 9, self.honors: True, self.athletics: True
	self.name: Johnny, self.grade:11, self.honors: False, self.athletics: False
	self.name: Johnny, self.grade: 11,

self.honors: True, self.athletics: False

Reset

CHALLENGE ACTIVITY 9.6.1: Defining a class constructor.

Write a constructor with parameters self, num_mins and num_messages. num_mins and num_messages should have a default value of 0.

Sample output with one plan created with input: 200 300, one plan created with no input, and one plan created with input: 500

```
My plan... Mins: 200 Messages: 300
Dad's plan... Mins: 0 Messages: 0
Mom's plan... Mins: 500 Messages: 0
```

```
class PhonePlan:
    # FIXME add constructor

    ''' Your solution goes here '''

    def print_plan(self):
        print('Mins:', self.num_mins, end=' ')
        print('Messages:', self.num_messages)


my_plan = PhonePlan(int(input()), int(input()))
dads_plan = PhonePlan()
moms_plan = PhonePlan(int(input()))

print('My plan...', end=' ')
my_plan.print_plan()

print('Dad\'s plan...', end=' ')
dads_plan.print_plan()

print('Mom\'s plan...', end=' ')
```

Run

9.7 Class interfaces

A class usually contains a set of methods that a programmer interacts with. For example, the class RaceTime might contain the instance methods print_time() and print_pace() that a programmer calls to print some output. A ***class interface*** consists of the methods that a programmer calls to create, modify, or access a class instance. The figure below shows the class interface of the RaceTime class, which consists of the __init__ constructor and the print_time() and print_pace() methods.

Figure 9.7.1: A class interface consists of methods to interact with an instance.

```
class RaceTime:
    def __init__(self, start_time, end_time, distance):
        # ...

    def print_time(self):
        # ...

    def print_pace(self):
        # ...
```

A class may also contain methods used internally that a user of the class need not access. For example, consider if the RaceTime class contains a separate method _diff_time() used by print_time() and print_pace() to find the total number of minutes to complete the race. A programmer using the RaceTime class does not need to use the _diff_time() function directly; thus, _diff_time() does not need to be a part of the class interface. Good practice is to prepend an underscore to methods only used internally by a class. The underscore is a widely recognized convention, but otherwise has no special syntactic meaning. A programmer could still call the method, e.g. `time1._diff_time()`, though such usage should be unnecessary if the class interface is well-designed.

Figure 9.7.2: Internal instance methods.

RaceTime class with internal instance method usage and definition highlighted.

```
class RaceTime:
    def __init__(self, start_hrs, start_mins, end_hrs,
end_mins, dist):
        self.start_hrs = start_hrs
        self.start_mins = start_mins
        self.end_hrs = end_hrs
        self.end_mins = end_mins
        self.distance = dist

    def print_time(self):
        total_time = self._diff_time()
        print('Time to complete race: %d:%d' %
(total_time[0], total_time[1]))

    def print_pace(self):
        total_time = self._diff_time()
        total_minutes = total_time[0]*60 + total_time[1]
        print('Avg pace (mins/mile): %.2f' %
(total_minutes / self.distance))

    def _diff_time(self):
        """Calculate total race time. Returns a 2-tuple
(hours, minutes)"""
        if self.end_mins >= self.start_mins:
            minutes = self.end_mins - self.start_mins
            hours = self.end_hrs - self.start_hrs
        else:
            minutes = 60 - self.start_mins + self.end_mins
            hours = self.end_hrs - self.start_hrs - 1

        return (hours, minutes)


distance = 5.0

start_hrs = int(input('Enter starting time hours: '))
start_mins = int(input('Enter starting time minutes: '))
end_hrs = int(input('Enter ending time hours: '))
end_mins = int(input('Enter ending time minutes: '))

race_time = RaceTime(start_hrs, start_mins, end_hrs,
end_mins, distance)

race_time.print_time()
race_time.print_pace()
```

```
Enter starting time
hours: 5
Enter starting time
minutes: 30
Enter ending time hours:
7
Enter ending time
minutes: 0
Time to complete race:
1:30
Average pace
(minutes/mile): 18.00
...
Enter starting time
hours: 9
Enter starting time
minutes: 30
Enter ending time hours:
10
Enter ending time
minutes: 3
Time to complete race:
0:33
Avg pace (mins/mile):
6.60
```

A class can be used to implement the computing concept known as an ***abstract data type (ADT)***, which is a data type whose creation and update are constrained to specific, well-defined operations (the class interface). A key aspect of an ADT is that the internal implementation of the data and operations are hidden from the ADT user, a concept known as *information hiding*, which allows the ADT user to be more productive by focusing on higher-level concepts. Information hiding also allows the ADT developer to improve the internal implementation without requiring changes to programs using the ADT. In the previous example, a RaceTime ADT was defined that captured the number of hours and minutes to complete a race, and that

presents a well-defined set of operations to create (via __init__) and view (via print_time and print_pace) the data.

Programmers commonly refer to separating an object's *interface* from its *implementation* (internal methods and variables); the user of an object need only know the object's interface.

Python lacks the ability to truly hide information from a user of the ADT, because all attributes of a class are always accessible from the outside. Many other computing languages require methods and variables to be marked as either *public* (part of a class interface) or *private* (internal), and attempting to access private methods and variables results in an error. Python on the other hand, is a more "trusting" language. A user of an ADT can always inspect, and if desired, utilize private variables and methods in ways unexpected by the ADT developer.

PARTICIPATION ACTIVITY 9.7.1: Class interfaces.

1) A class interface consists of the methods that a programmer should use to modify or access the class
   - True
   - False

2) Internal methods used by the class should start with an underscore in their name.
   - True
   - False

3) Internal methods can not be called; e.g., my_instance._calc() results in an error.
   - True
   - False

4) A well-designed class separates its *interface* from its *implementation*.
   - True
   - False

9.8 Class customization

9.8 Class customization

Class customization is the process of defining how a class should behave for some common operations. Such operations might include printing, accessing attributes, or how instances of that class are compared to each other. To customize a class, a programmer implements instance methods with ***special method names*** that the Python interpreter recognizes. For example, to change how a class instance object is printed, the special __str__()_ method can be defined, as illustrated below.

Figure 9.8.1: Implementing __str__() alters how the class is printed.

Normal printing

```
class Toy:
    def __init__(self, name,
price, min_age):
        self.name = name
        self.price = price
        self.min_age = min_age


truck = Toy('Monster Truck
XX', 14.99, 5)
print(truck)
```

```
<__main__.Toy instance at
0xb74cb98c>
```

Customized printing

```
class Toy:
    def __init__(self, name, price, min_age):
        self.name = name
        self.price = price
        self.min_age = min_age

    def __str__(self):
        return ('%s costs only $%.2f. Not for
children under %d!' %
                (self.name, self.price,
self.min_age))

truck = Toy('Monster Truck XX', 14.99, 5)
print(truck)
```

```
Monster Truck XX costs only $14.99. Not for
children under 5!
```

The left program prints a default string for the class instance. The right program implements __str__(), generating a custom message using some instance attributes.

Run the tool below, which visualizes the execution of the above example. When `print(truck)` is evaluated, the __str__() method is called.

PARTICIPATION ACTIVITY 9.8.1: Implementing __str__() alters how the class is printed.

```
1 class Toy:
2     def __init__(self, name, price, min_age):
3         self.name = name
4         self.price = price
```

```
5           self.min_age = min_age
6
7       def __str__(self):
8           return ('%s costs only $%.2f.\nNot for children unde
9                   (self.name, self.price, self.min_age))
10
11  truck = Toy('Monster Truck XX', 14.99, 5)
12  print(truck)
```


<< First < Back Step 1 of 12 Forward > L

line that has just executed
next line to execute

Frames Objects

zyDE 9.8.1: Customization of printing a class instance.

The following class represents a vehicle for sale in a used-car lot. Add a __str__() m
that printing an instance of Car displays a string in the following format:

```
1989 Chevrolet Blazer:
  Mileage: 115000
  Sticker price: $3250
```

Load default template... Run

```
1
2   class Car:
3       def __init__(self, make, model, year, m
4           self.make = make
5           self.model = model
6           self.year = year
7           self.miles = miles
8           self.price = price
9
10      def __str__(self):
11          # ... This line will cause error un
12
13  cars = []
14  cars.append(Car('Ford', 'Mustang', 2013, 25
15  cars.append(Car('Nissan', 'Xterra', 2004, 8
16  cars.append(Car('Nissan', 'Mazda', 2012, 25
17
18  for car in cars:
19      print(car)
20
```

Class customization can redefine the functionality of built-in operators like <, >=, +, -, and * when used with class instances, a technique known as ***operator overloading***.

The below code shows overloading of the less-than (<) operator of the Time class by defining a method with the __lt__ special method name.

Figure 9.8.2: Overloading the less-than operator of the Time class allows for comparison of instances.

```
class Time:
    def __init__(self, hours, minutes):
        self.hours = hours
        self.minutes = minutes

    def __str__(self):
        return '%d:%d' % (self.hours, self.minutes)

    def __lt__(self, other):
        if self.hours < other.hours:
            return True
        elif self.hours == other.hours:
            if self.minutes < other.minutes:
                return True
        return False

num_times = 3
times = []

# Obtain times from user input
for i in range(num_times):
    user_input = input('Enter time (Hrs:Mins): ')
    tokens = user_input.split(':')
    times.append(Time(int(tokens[0]), int(tokens[1])))

min_time = times[0]
for t in times:
    if t < min_time :
        min_time = t

print('\nEarliest time is', min_time)
```

```
Enter time (Hrs:Mins): 10:40
Enter time (Hrs:Mins): 12:15
Enter time (Hrs:Mins): 9:15

Earliest time is 9:15
```

In the above program, the Time class contains a definition for the __lt__ method, which overloads the < operator. When the comparison `t < min_time` is evaluated, the __lt__ method is automatically called. The self parameter of __lt__ is bound to the left operand, t, and the other parameter is bound to the right operand, min_time. Returning True indicates that t is indeed less-than min_time, and returning False indicates that t equal-to or greater-than min_time.

Methods like ***__lt__*** above are known as ***rich comparison methods***. The following table describes rich comparison methods and the corresponding relational operator that is overloaded.

Table 9.8.1: Rich comparison methods.

Rich comparison method	Overloaded operator
__lt__(self, other)	less-than (<)
__le__(self, other)	less-than or equal-to (<=)
__gt__(self, other)	greater-than (>)
__ge__(self, other)	greater-than or equal-to (>=)
__eq__(self, other)	equal to (==)
__ne__(self, other)	not-equal to (!=)

zyDE 9.8.2: Rich comparisons for a quarterback class.

Complete the __gt__ method. A quarterback is considered greater than another on quarterback has both more wins and a higher quarterback passer rating.

Once __gt__ is complete, compare Tom Brady's 2007 stats as well (yards: 4806, TD completions: 398, attempts: 578, interceptions: 8, wins: 16).

Load default

```
1
2  class Quarterback:
3      def __init__(self, yrds, tds, cmps, atts, ints, wins):
4          self.wins = wins
5
6          # Calculate quarterback passer rating (NCAA)
7          self.rating = ((8.4*yrds) + (330*tds) + (100*cmps) - (200 * ints))/atts
8
9      def __le__(self, other):
10         if (self.rating <= other.rating) or (self.wins <= other.wins):
11             return True
12         return False
13
14     def __gt__(self, other):
15         return True
16         # Complete the method...
17
18 peyton = Quarterback(yrds=4700, atts=679, cmps=450, tds=33, ints=17, wins=10)
19 eli = Quarterback(yrds=4002, atts=539, cmps=339, tds=31, ints=25, wins=9)
20
21 if peyton > eli:
```


Run

More advanced usage of class customization is possible, such as customizing how a class accesses or sets its attributes. Such advanced topics are out of scope for this material; however, the reader is encouraged to explore the links at the end of the section for a complete list of class customizations and special method names.

PARTICIPATION ACTIVITY 9.8.2: Rich comparison methods.

Consider the following class:

```
class UsedCar:
    def __init__(self, price, condition):
        self.price = price
        self.condition = condition  # integer between 0-5; 0=poor condition, 5=new
condition
```

Fill in the missing code as described in each question to complete the rich comparison methods.

1) A car is less than another if the price is lower.

```
def __lt__(self, other):
    if                    :
        return True
    return False
```

Check Show answer

2) A car is less than or equal-to another if the price is at most the same.

```
def __le__(self, other):
    if                    :
        return True
    return False
```


Check Show answer

3) A car is greater than another if the condition is better.

```
def __gt__(self, other):
    if                    :
        return True
    return False
```

Check Show answer

4) Two cars are not equivalent if either the prices or conditions don't match.

```
def __ne__(self, other):
    if                    :
        return True
    return False
```

Check Show answer

CHALLENGE ACTIVITY 9.8.1: Defining __str__.

Write the special method __str__() for CarRecord.

Sample output with input: 2009 'ABC321'

```
Year: 2009, VIN: ABC321
```

```
1  class CarRecord:
2      def __init__(self):
3          self.year_made = 0
4          self.car_vin = ''
5
6      # FIXME add __str__()
7
8      ''' Your solution goes here '''
9
10 my_car = CarRecord()
11 my_car.year_made = int(input())
12 my_car.car_vin = input()
13
14 print(my_car)
```


Run

Exploring further:

- Wikipedia: Operator overloading
- Python documentation: Class customization

9.9 More operator overloading: Classes as numeric types

Numeric operators such as +, -, *, and / can be overloaded using class customization techniques. Thus, a user-defined class can be treated as a numeric type of object wherein instances of that class can be added together, multiplied, etc. Consider the example example, which represents a 24-hour clock time.

Figure 9.9.1: Extending the time class with overloaded subtraction operator.

```
Enter time1 (hours:minutes):
5:00
Enter time2 (hours:minutes):
3:30
Time difference: 01:30
...
Enter time1 (hours:minutes):
22:30
Enter time2 (hours:minutes):
2:40
Time difference: 04:10
```

```
class Time24:
    def __init__(self, hours, minutes):
        self.hours = hours
        self.minutes = minutes

    def __str__(self):
        return '%02d:%02d' % (self.hours, self.minutes)

    def __gt__(self, other):
        if self.hours > other.hours:
            return True
        else:
            if self.hours == other.hours:
                if self.minutes > other.minutes:
                    return True
        return False

    def __sub__(self, other):
        """ Calculate absolute distance between two times
"""
        if self > other:
            larger = self
            smaller = other
        else:
            larger = other
            smaller = self

        hrs = larger.hours - smaller.hours
        mins = larger.minutes - smaller.minutes
        if mins < 0:
            mins += 60
            hrs -=1

        # Check if times wrap to new day
        if hrs > 12:
            hrs = 24 - (hrs + 1)
            mins = 60 - mins

        # Return new Time24 instance
        return Time24(hrs, mins)

t1 = input('Enter time1 (hours:minutes): ')
tokens = t1.split(':')
time1 = Time24(int(tokens[0]), int(tokens[1]))

t2 = input('Enter time2 (hours:minutes): ')
tokens = t2.split(':')
time2 = Time24(int(tokens[0]), int(tokens[1]))

print('Time difference:', time1 - time2)
```


©zyBooks 11/11/19 20:35 569464
Diep Vu
UNIONCSC103OrhanFall2019

The above program adds a definition of the __sub__method to the Time24 class that is called when an expression like **`time1 - time2`** is evaluated. The method calculates the absolute difference between the two times, and returns a new instance of Time24 containing the result.

The overloaded method will be called whenever the left operand is an instance Time24. Thus, an expression like `time1 - 1` will also cause the overloaded method to be called. Such an expression would cause an error because the __sub__ method would attempt to access the attribute other.minutes, but the integer 1 does not contain a minutes attribute. The error occurs because the behavior is undefined; does `time1 - 1` mean to subtract one hour or one minute?

To handle subtraction of arbitrary object types, the built-in ***isinstance()*** function can be used. The isinstance() function returns a True or False Boolean depending on whether a given variable matches a given type. The __sub__ function is modified below to first check the type of the right operand, and subtract an hour if the right operand is an integer, or find the time difference if the right operand is another Time24 instance:

Figure 9.9.2: The isinstance() built-in function.

```
def __sub__(self, other):
        if isinstance(other, int): # right op
is integer
            return Time24(self.hours - other,
self.minutes)

        if isinstance(other, Time24):  #
right op is Time24
            if self > other:
                larger = self
                smaller = other
            else:
                larger = other
                smaller = self

            hrs = larger.hours -
smaller.hours
            mins = larger.minutes -
smaller.minutes
            if mins < 0:
                mins += 60
                hrs -=1

            # Check if times wrap to new day
            if hrs > 12:
                hrs = 24 - (hrs + 1)
                mins = 60 - mins

            # Return new Time24 instance
            return Time24(hrs, mins)

        print('%s unsupported' %
(type(other)))
        raise NotImplementedError
```

Operation	Result
t1 - t2	Difference between t1, t2
t1 - 5	t1 minus 5 hours.
t1 - 5.75	"float unsupported"
t1 - <other_type>	"<other_type> unsupported"

Every operator in Python can be overloaded. The table below lists some of the most common methods. A full list is available at the bottom of the section.

Table 9.9.1: Methods for emulating numeric

Table 9.9.1: Methods for emulating numeric types.

Method	Description
__add__(self, other)	Add (+)
__sub__(self, other)	Subtract (-)
__mul__(self, other)	Multiply (*)
__truediv__(self, other)	Divide (/)
__floordiv__(self, other)	Floored division (//)
__mod__(self, other)	Modulus (%)
__pow__(self, other)	Exponentiation (**)
__and__(self, other)	"and" logical operator
__or__(self, other)	"or" logical operator
__abs__(self)	Absolute value (abs())
__int__(self)	Convert to integer (int())
__float__(self)	Convert to floating point (float())

The table above lists common operators such as addition, subtraction, multiplication, division, and so on. Sometimes a class also needs to be able to handle being passed as arguments to built-in functions like abs(), int(), float(), etc. Defining the methods like __abs__(), __int__(), and __float__() will automatically cause those methods to be called when an instance of that class is passed to the corresponding function. The methods should return an appropriate object for each method, i.e., an integer value for __int__() and a floating-point value for __float__(). Note that not all such methods need to be implemented for a class; their usage is generally optional, but can provide for cleaner and more elegant code. Not defining a method will simply cause an error if that method is needed but not found, which indicates to the programmer that additional functionality must be implemented.

PARTICIPATION ACTIVITY 9.9.1: Emulating numeric types with operating overloading.

Assume that the following class is defined. Fill in the missing statements in the most direct possible way to complete the described method.

```
class LooseChange:
    def __init__(self, value):
        self.value = value  # integer representing total number of cents.

    # ...
```


1) Adding two LooseChange instances `lc1 + lc2` returns a new LooseChange instance with the summed value of lc1 and lc2.

```
def __add__(self, other):

      new_value =

      return
LooseChange(new_value)
```

Check Show answer

2) Executing the code:

```
lc1  = LooseChange(135)
print(float(lc1))
```

yields the output

```
1.35
```

```
def __float__(self):

      fp_value =

      return fp_value
```


Check Show answer

Exploring further:

- List of numeric special method names

9.10 Memory allocation and garbage collection

Memory allocation

The process of an application requesting and being granted memory is known as ***memory allocation***. Memory used by a Python application must be granted to the application by the operating system. When an application requests a specific amount of memory from the operating system, the operating system can then choose to grant or deny the request.

While some languages require the programmer to write memory allocating code, the Python runtime handles memory allocation for the programmer. Ex: Creating a list in Python and then appending 100 items means that memory for the 100 items must be allocated. The Python runtime allocates memory for lists and other objects as necessary.

PARTICIPATION ACTIVITY 9.10.1: Memory allocation in Python.

Animation content:

undefined

Animation captions:

1. System memory is partitioned into segments and managed by the operating system.
2. A Python application creates an array with 100 items. The Python runtime has allocated space for this array.
3. Other applications may use other areas of allocated memory.
4. Memory allocation is usually transparent to the programmer, since the allocation is done by the Python runtime.

PARTICIPATION ACTIVITY 9.10.2: Memory allocation in Python.

1) The Python runtime requests memory from the operating system.

True

False

2) Certain objects in a Python may reside in memory that has not been allocated.

True

False

3) All programming languages perform all memory allocation on behalf of the programmer.

True

False

Garbage collection

Python is a managed language, meaning objects are deallocated automatically by the Python runtime, and not by the programmer's code. When an object is no longer referenced by any variables, the object becomes a candidate for deallocation.

A ***reference count*** is an integer counter that represents how many variables reference an object. When an object's reference count is 0, that object is no longer referenced. Python's garbage collector will deallocate objects with a reference count of 0. However, the time between an object's reference count becoming 0 and that object being deallocated may differ across different Python runtime implementations.

PARTICIPATION ACTIVITY 9.10.3: Python's garbage collection.

Animation content:

undefined

Animation captions:

1. The variable string1 is a reference to the "Python" string object. string2 and string3 both reference the "Computer Science" string object.
2. The "Python" string object is referenced by 1 variable and therefore has a reference count (RC) of 1. The "Computer science" string object's RC is 2.
3. Reference counts > 0 imply that neither object can be deallocated.
4. When string1 is reassigned to reference the "zyBooks" string, the "Python" string object is no longer referenced and can be deallocated.

5. After assigning string2 with "zyBooks", "Computer Science" is still referenced by string3 and cannot be deallocated.
6. The Python garbage collector will eventually deallocate objects that are no longer referenced.

PARTICIPATION ACTIVITY 9.10.4: Reference counts and garbage collection.

1) An object with a reference count of 0 can be deallocated by the garbage collector.
   - ○ True
   - ○ False

2) Immediately after an object's reference count is decremented from 1 to 0, the garbage collector deallocates the object.
   - ○ True
   - ○ False

3) Swapping variables string1 and string2 with the code below is potentially problematic, because a moment exists when the "zyBooks" string has a reference count of 0.

```
string1 = "zyBooks"
string2 = "Computer science"

temp = string1
string1 = string2
string2 = temp
```

   - ○ True
   - ○ False

©zyBooks 11/11/19 20:35 569464
Diep Vu
UNIONCSC103OrhanFall2019