15.1 Classes: Introduction

This section has been set as optional by your instructor.

Grouping related items into objects

The physical world is made up of material items like wood, metal, plastic, fabric, etc. To keep the world understandable, people think in terms of higher-level objects, like chairs, tables, and TV's. Those objects are groupings of the lower-level items.

Likewise, a program is made up of lower-level items like variables and functions. To keep programs understandable, programmers often deal with higher-level groupings of those items, known as objects. In programming, an ***object*** is a grouping of data (variables) and operations that can be performed on that data (functions or methods).

PARTICIPATION ACTIVITY 15.1.1: The world is viewed not as materials, but rather as objects.

Animation captions:

1. The world consists of items like, wood, metal, fabric, etc.
2. But people think in terms of higher-level objects, like chairs, couches, and drawers.
3. In fact, people think mostly of the operations that can be done with the objects. For a drawer, operations include put stuff in and take stuff out.

PARTICIPATION ACTIVITY 15.1.2: Programs commonly are not viewed as variables and functions/methods, but rather as objects.

Animation content:

```
undefined
```


Animation captions:

1. A program consists of variables and functions/methods. But programmers may prefer to think of higher-level objects like Restaurants and Hotels.
2. In fact, programmers think mostly of the operations that can be done with the object, like giving a Hotel or Restaurant a name or adding a review.

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

PARTICIPATION ACTIVITY 15.1.3: 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

Abstraction / Information hiding

Abstraction occurs when a user interacts with an object at a high-level, allowing lower-level internal details to remain hidden (aka ***information hiding*** or ***encapsulation***). Ex: An oven supports an abstraction of a food compartment and a knob to control heat. An oven's user does not need to interact with the internal parts of an oven.

Objects support abstraction by hiding entire groups of functions and variables and exposing only certain functions to a user.

An ***abstract data type*** (***ADT***) is a data type whose creation and update are constrained to specific well-defined operations. A class can be used to implement an ADT.

PARTICIPATION ACTIVITY 15.1.4: Objects strongly support abstraction / information hiding.

Animation captions:

1. Abstraction simplifies our world. An oven is viewed as having a compartment for food and a knob that can be turned to heat the food.
2. People need not be concerned with an oven's internal workings. Ex: People don't reach inside trying to adjust the flame.
3. Similarly, an object has operations that a user can apply. The object's internal data, and possibly other operations, are hidden from the user.

PARTICIPATION ACTIVITY 15.1.5: Abstraction / information hiding.

1) A car presents an abstraction to a user, including a steering wheel, gas pedal, and brake.

- True
- False

2) A refrigerator presents an abstraction to a user, including freon gas, a compressor, and a fan.

- True
- False

3) A software object is created for a soccer team. A reasonable abstraction allows setting the team's name, adding or deleting players, and printing a roster.

- True
- False

4) A software object is created for a university class. A reasonable abstraction allows viewing and modifying variables for the teacher's name, and variables implementing a list of every student's name as well.

- True
- False

Python built-in objects

Python automatically creates ***built-in*** objects that are provided by the language for a programmer to use, and include the basic data types like integers and strings.

A programmer always interacts with built-in objects when writing Python code. Ex: A string object created with `mystr = 'Hello!'`. The value of the string 'Hello!' is one part of the object, as are functions to operate on that string like `str.isdigit()` and `str.lower()`.

PARTICIPATION ACTIVITY 15.1.6: Built in objects.

Animation content:

```
Defining  a new string or integer variable utilizes the built-in
string and integer data type objects. These objects group together the
value of the variable along with useful functions for operating on
that variable.
```

Animation captions:

1. Defining a new string s1 creates a new built-in str object. The str object groups together the

string value "Hello!!" along with useful functions.
2. Defining a new integer i1 creates a new built-in int object. The implementation of the int object abstracts many details away that a programmer doesn't need.

PARTICIPATION ACTIVITY 15.1.7: Built-in objects.

1) The Python programming language provides a built-in object for strings.
   - True
   - False

2) Built-in objects often include useful functions.
   - True
   - False

3) The built-in string object includes every function a programmer could ever find useful for dealing with strings.
   - True
   - False

Survey

The following questions are part of a zyBooks survey to help us learn about the experiences in programming classes among college students. The survey can be taken anonymously and takes just 5-10 minutes. Please take a short moment to answer by clicking the following link.

Link: Student survey

15.2 Classes: Grouping data

This section has been set as optional by your instructor.

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 15.2.1: 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 15.2.1: 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 15.2.2: Using instantiation to create a variable using the Time class.


```
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('{} hours'.format(my_time.hours), end='
')
print('and {}
minutes'.format(my_time.minutes))
```

```
7 hours and 15
minutes
```

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

PARTICIPATION ACTIVITY 15.2.1: Using classes and attribute reference.

Animation captions:

1. The Time() method 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 15.2.3: 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

print('{} hours and {} minutes'.format(time1.hours,
time1.minutes))
print('{} hours and {} minutes'.format(time2.hours,
time2.minutes))
```

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


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

PARTICIPATION ACTIVITY 15.2.2: Class terms.

Mouse: Drag/drop. Refresh the page if unable to drag and drop.

class **__init__** **instance** **self** **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 15.2.3: 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 15.2.4: 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()
```

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 15.2.1: Enter the output of grouping data.

334598.939404.qx3zqy7

Start

Type the program's output

```
class Person:
   def __init__(self):
      self.name = ''

person1 = Person()
username = 'Jon'

person1.name = username
print('This is ' + person1.name)
```

1 2

Check Next

CHALLENGE ACTIVITY 15.2.2: 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
```


334598.939404.qx3zqy7

```
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 15.2.3: 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
```


334598.939404.qx3zqy7

```
1 class InventoryTag:
2     def __init__(self):
3         self.item_id = 0
4         self.quantity_remaining = 0
```

Run

15.3 Instance methods

This section has been set as optional by your instructor.

A function defined within a class is known as an ***instance method***. An instance method can be referenced using dot notation. The following example illustrates:

Figure 15.3.1: 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 15.3.1: 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 15.3.1: Adding methods to a class.

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

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: {:.2f}'.form
14
15 barbara = Employee()
16 barbara.wage = 11.50
17 barbara.hours_worked = 20
18 print('Barbara:\n Net pay: {:.2f}'.fo
```

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 15.3.2: 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 {:.2f}'.format(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 15.3.2: 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:
{}:{} GMT'

.format(self.hours-8,
self.minutes))
        else:
            print('Time is:
{}:{}'
```


CHALLENGE ACTIVITY 15.3.1: Instance methods.

334598.939404.qx3zqy7

Start

Type the program's output

```
class Person:
   def __init__(self):
      self.first_name = ''

   def print_name(self):
      print('You are', self.first_name)

person1 = Person()
person1.first_name = 'Sam'
person1.print_name()
```

1	2	3

Check Next

CHALLENGE ACTIVITY 15.3.2: Defining an instance method.

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

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

```
Kids: 0
```

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

334598.939404.qx3zqy7

```
1  class PersonInfo:
2      def __init__(self):
3          self.num_kids = 0
4
5      # FIXME: Write inc_num_kids(self)
6
7      ''' Your solution goes here '''
8
9  person1 = PersonInfo()
10
11 print('Kids:', person1.num_kids)
12 person1.inc_num_kids()
13 print('New baby, kids now:', person1.num_kids)
```


Run

15.4 Class and instance object types

This section has been set as optional by your instructor.

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 15.4.1: Class Time's init method initializes two new Time instance 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 15.4.1: 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 15.4.2: 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 15.4.2: 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 variables 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 15.4.3: 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 15.4.3: Class and instance objects.

Mouse: Drag/drop. Refresh the page if unable to drag and drop.

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

Class 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 15.4.4: 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

CHALLENGE ACTIVITY 15.4.1: Classes and instances.

334598.939404.qx3zqy7

Start

Type the program's output

```
class Shape:
    def __init__(self):
        self.color = None

shape1 = Shape()
shape2 = Shape()
shape2.color = 'indigo'

print(shape1.color)
print(shape2.color)
```

1 2 3 4 5

Check Next

15.5 Class example: Seat reservation system

This section has been set as optional by your instructor.

zyDE 15.5.1: Using classes to implement an airline seat reservation system.

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

Load default ter

```
class Seat:
    def __init__(self):
        self.first_name = ''
        self.last_name = ''
        self.paid = 0.0

    def reserve(self, f_name, l_name, amt_paid):
        self.first_name = f_name
        self.last_name = l_name
        self.paid = amt_paid

    def make_empty(self):
        self.first_name = ''
        self.last_name = ''
        self.paid = 0.0

    def is_empty(self):
        return self.first_name == ''
```

r
1
Hank

Run

15.6 Class constructors

This section has been set as optional by your instructor.

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

Figure 15.6.1: 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 15.6.2: 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: {}:{}'.format(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}'.format(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 15.6.1: 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 15.6.3: 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 ('{} earns {:.2f} per week'.format(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 15.6.2: 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

    # ...
```


Mouse: Drag/drop. Refresh the page if unable to drag and drop.

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

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

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

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 15.6.1: Constructor customization.

334598.939404.qx3zqy7

Start

Type the program's output


```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

rectangle1 = Rectangle(1, 9)
rectangle2 = Rectangle(8, 15)

print(rectangle1.width)
print(rectangle2.length)
```

1 | 2 | 3

Check Next

CHALLENGE ACTIVITY 15.6.2: 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
```

334598.939404.qx3zqy7

```
1  class PhonePlan:
2      # FIXME add constructor
3
4      ''' Your solution goes here '''
5
6      def print_plan(self):
7          print('Mins:', self.num_mins, end=' ')
8          print('Messages:', self.num_messages)
9
10
11 my_plan = PhonePlan(int(input()), int(input()))
12 dads_plan = PhonePlan()
13 moms_plan = PhonePlan(int(input()))
14
15 print('My plan...', end=' ')
16 my_plan.print_plan()
17
18 print('Dad\'s plan...', end=' ')
```


Run

15.7 Class interfaces

This section has been set as optional by your instructor.

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 15.7.1: 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 15.7.2: 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:
{}:{}'.format(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}'.format(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 15.7.1: 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

15.8 Class customization

This section has been set as optional by your instructor.

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. Ex: To change how a class instance object is printed, the special __str__() method can be defined, as illustrated below.

Figure 15.8.1: 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 ('{} costs only ${:.2f}. Not for
children under {}!'
                .format(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 15.8.1: Implementing __str__() alters how the class is printed.

zyDE 15.8.1: Customization of printing a class instance.

The following class represents a vehicle for sale in a used-car lot. Add a __str__() met 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, y
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 er
12
13 cars = []
14 cars.append(Car('Ford', 'Mustang', 20
15 cars.append(Car('Nissan', 'Xterra', 2
16 cars.append(Car('Nissan', 'Maxima', 2
17
18 for car in cars:
```

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 15.8.2: 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 '{}:{}'.format(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 15.8.1: 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 15.8.2: Rich comparisons for a quarterback class.

Complete the __gt__ method. A quarterback is considered greater than another only i 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, TDs: completions: 398, attempts: 578, interceptions: 8, wins: 16).

Load default ter

```
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 * int
8
9     def __lt__(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, wi
```

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 15.8.2: 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 15.8.1: Enter the output of the program that has a class with special methods.

334598.939404.qx3zqy7

Start

Type the program's output

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

    def __str__(self):
        minute_string = str(self.minutes)
        if self.minutes < 10:
            minute_string = '0{}'.format(minute_string)
        return '{} mins {} hrs'.format(minute_string, self.hours)

day = Duration(24, 0)
print(day)
```

1 2

Check Next

CHALLENGE ACTIVITY 15.8.2: Defining __str__.

Write the special method __str__() for CarRecord.

Sample output with input: 2009 'ABC321'

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


334598.939404.qx3zqy7

```
1 class CarRecord:
2     def __init__(self):
3         self.year_made = 0
4         self.car_vin = ''
5
```

Run

Exploring further:

- Wikipedia: Operator overloading
- Python documentation: Class customization

15.9 More operator overloading: Classes as numeric types

This section has been set as optional by your instructor.

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, which represents a 24-hour clock time.

Figure 15.9.1: Extending the time class with overloaded subtraction operator.

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

    def __str__(self):
        return '{:02d}:{:02d}'.format(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)
```



```
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
```


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 15.9.2: 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('{}
unsupported'.format(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 15.9.1: 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 15.9.1: 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

CHALLENGE ACTIVITY | 15.9.1: Enter the output of classes as numeric types.

334598.939404.qx3zqy7

Start

Type the program's output


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

    def __add__(self, other):
        total_hours = self.hours + other.hours
        total_minutes = self.minutes + other.minutes
        if total_minutes >= 60:
            total_hours += 1
            total_minutes -= 60
        return Duration(total_hours, total_minutes)

first_trip = Duration(4, 54)
second_trip = Duration(0, 46)

first_time = first_trip + second_trip
second_time = second_trip + second_trip

print(first_time.hours, first_time.minutes)
print(second_time.hours, second_time.minutes)
```

1 | 2 | 3

Check Next

Exploring further:

- List of numeric special method names

15.10 Memory allocation and garbage collection

This section has been set as optional by your instructor.

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 15.10.1: 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 invisible to the programmer, since the allocation is done by the Python runtime.

PARTICIPATION ACTIVITY 15.10.2: Memory allocation in Python.

1) The Python runtime requests memory from the operating system.
   - ○ True
   - ○ False

2) Certain objects in a Python application 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 15.10.3: 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 15.10.4: 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

15.11 LAB: Car value (classes)

This section has been set as optional by your instructor.

Complete the `Car` class by creating an attribute purchase_price (type int) and the method print_info() that outputs the car's information.

Ex: If the input is:

```
2011
18000
2018
```

where 2011 is the car's model year, 18000 is the purchase price, and 2018 is the current year, then print_info() outputs:

```
Car's information:
   Model year: 2011
   Purchase price: 18000
   Current value: 5770
```


Note: print_info() should use three spaces for indentation.

334598.939404.qx3zqy7

LAB ACTIVITY | 15.11.1: LAB: Car value (classes) 0 / 10

main.py Load default template...

```
1  class Car:
2      def __init__(self):
3          self.model_year = 0
4          # TODO: Declare purchase_price attribute
5
6          self.current_value = 0
7
8      def calc_current_value(self, current_year):
9          depreciation_rate = 0.15
10         # Car depreciation formula
11         car_age = current_year - self.model_year
12         self.current_value = round(self.purchase_price * (1 - depreciation_rate)
13
14     # TODO: Define print_info() method to output model_year, purchase_price, and
15
16
17 if __name__ == "__main__":
18     year = int(input())
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```


15.12 LAB: Nutritional information (classes/constructors)

This section has been set as optional by your instructor.

Complete the `FoodItem` class by adding a constructor to initialize a food item. The constructor should initialize the name (a string) to "None" and all other instance attributes to 0.0 by default. If the constructor is called with a food name, grams of fat, grams of carbohydrates, and grams of protein, the constructor should assign each instance attribute with the appropriate parameter value.

The given program accepts as input a food item name, fat, carbs, and protein and the number of servings. The program creates a food item using the constructor parameters' default values and a food item using the input values. The program outputs the nutritional information and calories per serving for both food items.

Ex: If the input is:

```
M&M's
10.0
34.0
2.0
1.0
```


where M&M's is the food name, 10.0 is the grams of fat, 34.0 is the grams of carbohydrates, 2.0 is the grams of protein, and 1.0 is the number of servings, the output is:

```
Nutritional information per serving of None:
   Fat: 0.00 g
```

```
   Carbohydrates: 0.00 g
   Protein: 0.00 g
Number of calories for 1.00 serving(s): 0.00

Nutritional information per serving of M&M's:
   Fat: 10.00 g
   Carbohydrates: 34.00 g
   Protein: 2.00 g
Number of calories for 1.00 serving(s): 234.00
```


334598.939404.qx3zqy7

LAB ACTIVITY | 15.12.1: LAB: Nutritional information (classes/constructors) 0 / 10

main.py

```
1 Loading latest submission...
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

15.13 LAB: Artwork label (classes/constructors)

This section has been set as optional by your instructor.

Define the **Artist** class with a constructor to initialize an artist's information and a print_info() method. The constructor should by default initialize the artist's name to "None" and the years of birth and death to 0. print_info() should display *Artist Name, born XXXX* if the year of death is -1 or *Artist Name (XXXX-YYYY)* otherwise.

Define the **Artwork** class with a constructor to initialize an artwork's information and a print_info() method. The constructor should by default initialize the title to "None", the year created to 0, and the artist to use the **Artist** default constructor parameter values.

Ex: If the input is:

```
Pablo Picasso
1881
1973
Three Musicians
1921
```


the output is:

```
Artist: Pablo Picasso (1881-1973)
Title: Three Musicians, 1921
```

If the input is:

```
Brice Marden
1938
-1
Distant Muses
2000
```

the output is:


```
Artist: Brice Marden, born 1938
Title: Distant Muses, 2000
```

334598.939404.qx3zqy7

LAB ACTIVITY | 15.13.1: LAB: Artwork label (classes/constructors) 0 / 10

main.py Load default template...

```
1  class Artist:
2      # TODO: Define constructor with parameters to initialize instance attributes
3      #       (name, birth_year, death_year)
4
5      # TODO: Define print_info() method. If death_year is -1, only print birth_ye
6
7
8  class Artwork:
9      # TODO: Define constructor with parameters to initialize instance attributes
10     #       (title, year_created, artist)
11
12     # TODO: Define print_info() method
13
14
15 if __name__ == "__main__":
16     user_artist_name = input()
17     user_birth_year = int(input())
18     user_death_year = int(input())
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```


15.14 LAB: Triangle area comparison (classes)

This section has been set as optional by your instructor.

Given class **`Triangle`**, complete the program to read and set the base and height of triangle1 and triangle2, determine which triangle's area is larger, and output the larger triangle's info, making use of **`Triangle`**'s relevant methods.

Ex: If the input is:

```
3.0
4.0
4.0
5.0
```

where 3.0 is triangle1's base, 4.0 is triangle1's height, 4.0 is triangle2's base, and 5.0 is triangle2's height, the output is:

```
Triangle with larger area:
Base: 4.00
Height: 5.00
Area: 10.00
```


334598.939404.qx3zqy7

LAB ACTIVITY 15.14.1: LAB: Triangle area comparison (classes) 0 / 10

main.py

```
1 Loading latest submission...
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

15.15 LAB: Winning team (classes)

This section has been set as optional by your instructor.

Complete the `Team` class implementation. For the instance method get_win_percentage(), the formula is:

```
team_wins / (team_wins + team_losses)
```

Note: Use floating-point division.

Ex: If the input is:

```
Ravens
13
3
```

where Ravens is the team's name, 13 is the number of team wins, and 3 is the number of team losses, the output is:

```
Congratulations, Team Ravens has a winning average!
```

If the input is Angels 80 82, the output is:

```
Team Angels has a losing average.
```

334598.939404.qx3zqy7

LAB ACTIVITY | 15.15.1: LAB: Winning team (classes) 0 / 10

main.py Load default template...

```
1  class Team:
2      def __init__(self):
3          self.team_name = 'none'
4          self.team_wins = 0
5          self.team_losses = 0
6
7      # TODO: Define get_win_percentage()
8
9
10 if __name__ == "__main__":
11
12     team = Team()
13
14     team_name = input()
15     team_wins = int(input())
16     team_losses = int(input())
17
18     team.team_name = team_name
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

15.16 LAB: Vending machine

This section has been set as optional by your instructor.

Given two integers as user inputs that represent the number of drinks to buy and the number of bottles to restock, create a VendingMachine object that performs the following operations:

- Purchases input number of drinks
- Restocks input number of bottles
- Reports inventory

A VendingMachine's initial inventory is 20 drinks.

Ex: If the input is:

```
5
2
```

the output is:

```
Inventory: 17 bottles
```

334598.939404.qx3zqy7

LAB ACTIVITY | 15.16.1: LAB: Vending machine 0 / 10

main.py Load default template...

```
class VendingMachine:
    def __init__(self):
        self.bottles = 20

    def purchase(self, amount):
        self.bottles = self.bottles - amount

    def restock(self, amount):
        self.bottles = self.bottles + amount

    def get_inventory(self):
        return self.bottles

    def report(self):
        print('Inventory: {} bottles'.format(self.bottles))

if __name__ == "__main__":
    # TODO: Create VendingMachine object
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```


©zyBooks 09/19/22 19:37 469702
Steven Cameron
WGUC859v4