Lab 4

Object-Oriented Programming

Lab Objective: *Python is a class-based language. A* class *is a blueprint for an object that binds together specified variables and routines. Creating and using custom classes is often a good way to clean and speed up a program. In this lab we learn how to define and use Python classes. In subsequents labs, we will create customized classes often for use in algorithms.*

Python Classes

A Python *class* is a code block that defines a custom object and determines its behavior. The `class` key word defines and names a new class. Other statements follow, indented below the class name, to determine the behavior of objects instantiated by the class.

A class needs a method called a *constructor* that is called whenever the class instantiates a new object. The constructor specifies the initial state of the object. In Python, a class's constructor is always named `__init__()`. For example, the following code defines a class for storing information about backpacks.

```
class Backpack(object):
    """A Backpack object class. Has a name and a list of contents.

    Attributes:
        name (str): the name of the backpack's owner.
        contents (list): the contents of the backpack.
    """
    def __init__(self, name):           # This function is the constructor.
        """Set the name and initialize an empty contents list.

        Inputs:
            name (str): the name of the backpack's owner.

        Returns:
            A Backpack object wth no contents.
        """
        self.name = name                # Initialize some attributes.
        self.contents = []
```

This `Backpack` class has two *attributes*: `name` and `contents`. Attributes are variables stored within the class object. In the body of the class definition, attributes are accessed via the name `self`. This name refers to the object internally once it has been created.

Instantiation

The `class` code block above only defines a blueprint for backpack objects. To create a backpack object, we "call" the class like a function. An object created by a class is called an *instance* of the class. It is also said that a class *instantiates* objects.

Classes may be imported in the same way as modules. In the following code, we import the `Backpack` class and instantiate a `Backpack` object.

```
# Import the Backpack class and instantiate an object called 'my_backpack'.
>>> from oop import Backpack
>>> my_backpack = Backpack("Fred")

# Access the object's attributes with a period and the attribute name.
>>> my_backpack.name
'Fred'
>>> my_backpack.contents
[]

# The object's attributes can be modified dynamically.
>>> my_backpack.name = "George"
>>> print(my_backpack.name)
George
```

> **Note**
>
> Many programming languages distinguish between *public* and *private* attributes. In Python, all attributes are automatically public. However, an attribute can be hidden from the user in IPython by starting the name with an underscore.

Methods

In addition to storing variables as attributes, classes can have functions attached to them. A function that belongs to a specific class is called a *method*. Below we define two simple methods in the `Backpack` class.

```
class Backpack(object):
    # ...
    def put(self, item):
        """Add 'item' to the backpack's list of contents."""
        self.contents.append(item)

    def take(self, item):
        """Remove 'item' from the backpack's list of contents."""
        self.contents.remove(item)
```

The first argument of each method must be `self`, to give the method access to the attributes and other methods of the class. The `self` argument is only included in the declaration of the class methods, **not** when calling the methods.

```
# Add some items to the backpack object.
>>> my_backpack.put("notebook")
>>> my_backpack.put("pencils")
>>> my_backpack.contents
['notebook', 'pencils']

# Remove an item from the backpack.
>>> my_backpack.take("pencils")
>>> my_backpack.contents
['notebook']
```

Problem 1. Expand the `Backpack` class to match the following specifications.

1. Modify the constructor so that it accepts a name, a color, and a maximum size (in that order). Make `max_size` a default argument with default value 5. Store each input as an attribute.
2. Modify the `put()` method to ensure that the backpack does not go over capacity. If the user tries to put in more than `max_size` items, print "No Room!" and do not add the item to the contents list.
3. Add a new method called `dump()` that resets the contents of the backpack to an empty list. This method should not receive any arguments (except `self`).

To ensure that your class works properly, consider writing a test function outside outside of the `Backpack` class that instantiates and analyzes a `Backpack` object. Your function may look something like this:

```
def test_backpack():
    testpack = Backpack("Barry", "black")       # Instantiate the object.
    if testpack.max_size != 5:                  # Test an attribute.
        print("Wrong default max_size!")
    for item in ["pencil", "pen", "paper", "computer"]:
        testpack.put(item)                      # Test a method.
    print(testpack.contents)
```

Inheritance

Inheritance is an object-oriented programming tool for code reuse and organization. To create a new class that is similar to one that already exists, it is often better to *inherit* the already existing methods and attributes rather than create a new class from scratch. This is done by including the existing class as an argument in the class definition (where the word `object` is in the definition of the `Backpack` class).

This creates a *class hierarchy*: a class that inherits from another class is called a *subclass*, and the class that a subclass inherits from is called a *superclass*.

For example, since a knapsack is a kind of backpack (but not all backpacks are knapsacks), we create a special `Knapsack` subclass that inherits the structure and behaviors of the `Backpack` class, and adds some extra functionality.

```
# Inherit from the Backpack class in the class definition.
class Knapsack(Backpack):
    """A Knapsack object class. Inherits from the Backpack class.
    A knapsack is smaller than a backpack and can be tied closed.

    Attributes:
        name (str): the name of the knapsack's owner.
        color (str): the color of the knapsack.
        max_size (int): the maximum number of items that can fit
            in the knapsack.
        contents (list): the contents of the backpack.
        closed (bool): whether or not the knapsack is tied shut.
    """
    def __init__(self, name, color, max_size=3):
        """Use the Backpack constructor to initialize the name, color,
        and max_size attributes. A knapsack only holds 3 item by default
        instead of 5.

        Inputs:
            name (str): the name of the knapsack's owner.
            color (str): the color of the knapsack.
            max_size (int): the maximum number of items that can fit
                in the knapsack. Defaults to 3.

        Returns:
            A Knapsack object with no contents.
        """
        Backpack.__init__(self, name, color, max_size)
        self.closed = True
```

A subclass may have new attributes and methods that are unavailable to the superclass, such as the `closed` attribute in the `Knapsack` class. If methods in the new class need to be changed, they are overwritten as is the case of the constructor in the `Knapsack` class. New methods can be included normally. As an example, we modify the `put()` and `take()` methods in `Knapsack` to check if the knapsack is shut.

```
class Knapsack(Backpack):
    # ...
    def put(self, item):
        """If the knapsack is untied, use the Backpack.put() method."""
        if self.closed:
            print("I'm closed!")
        else:
            Backpack.put(self, item)

    def take(self, item):
        """If the knapsack is untied, use the Backpack.take() method."""
        if self.closed:
            print("I'm closed!")
        else:
            Backpack.take(self, item)
```

Since `Knapsack` inherits from `Backpack`, a knapsack object is a backpack object. All methods defined in the `Backpack` class are available to instances of the `Knapsack` class. For example, the `dump()` method is available even though it is not defined explicitly in the `Knapsack` class.

```
>>> from oop import Knapsack
>>> my_knapsack = Knapsack("Brady", "brown")
>>> isinstance(my_knapsack, Backpack)       # A Knapsack is a Backpack.
True

# The put() and take() method now require the knapsack to be open.
>>> my_knapsack.put('compass')
I'm closed!

# Open the knapsack and put in some items.
>>> my_knapsack.closed = False
>>> my_knapsack.put("compass")
>>> my_knapsack.put("pocket knife")
>>> my_knapsack.contents
['compass', 'pocket knife']

# The dump method is inherited from the Backpack class, and
# can be used even though it is not defined in the Knapsack class.
>>> my_knapsack.dump()
>>> my_knapsack.contents
[]
```

Problem 2. Create a `Jetpack` class that inherits from the `Backpack` class.

1. Overwrite the constructor so that in addition to a name, color, and maximum size, it also accepts an amount of fuel. Change the default value of `max_size` to 2, and set the default value of fuel to 10. Store the fuel as an attribute.
2. Add a `fly()` method that accepts an amount of fuel to be burned and decrements the fuel attribute by that amount. If the user tries to burn more fuel than remains, print "Not enough fuel!" and do not decrement the fuel.
3. Overwrite the `dump()` method so that both the contents and the fuel tank are emptied.

Magic Methods

In Python, a *magic method* is a special method used to make an object behave like a built-in data type. Magic methods begins and ends with two underscores, like the constructor `__init__()`. Every Python object is automatically endowed with several magic methods, but they are normally hidden from IPython's object introspection because they begin with an underscore. To see an object's magic methods, type an underscore before pressing tab.

```
In [1]: run oop.py                  # Load the names from oop.py.

In [2]: b = Backpack("Oscar", "green")

In [3]: b.      # Press 'tab' to see standard methods and attributes.
b.name      b.contents  b.put       b.take

In [4]: b.put?
Signature: b.put(item)
Docstring: Add 'item' to the backpack's content list.
File:      ~/Downloads/Packs.py
Type:      instancemethod

In [5]: b.__    # Now press 'tab' to see magic methods.
b.__add__           b.__getattribute__  b.__reduce_ex__
b.__class__         b.__hash__          b.__repr__
b.__delattr__       b.__init__          b.__setattr__
b.__dict__          b.__lt__            b.__sizeof__
b.__doc__           b.__module__        b.__str__
b.__eq__            b.__new__           b.__subclasshook__
b.__format__        b.__reduce__        b.__weakref__

In [6]: b?
Type:           Backpack
File:           ~/Downloads/Packs.py
Docstring:
A Backpack object class. Has a name and a list of contents.

Attributes:
    name (str): the name of the backpack's owner.
    contents (list): the contents of the backpack.
Init docstring:
Set the name and initialize an empty contents list.

Inputs:
    name (str): the name of the backpack's owner.

Returns:
    A backpack object wth no contents.
```

Note

In all of the preceding examples, the comments enclosed by sets of three double quotes are the object's *docstring*, stored as the `__doc__` attribute. A good docstring typically includes a summary of the class or function, information about the inputs and returns, and other notes. Modules also have a `__doc__` attribute for describing the purpose of the file. Writing detailed docstrings is critical so that others can utilize your code correctly (and so that you don't forget how to use your own code!).

Now, suppose we wanted to add two backpacks together. How should addition be defined for backpacks? A simple option is to add the number of contents. Then if backpack A has 3 items and backpack B has 5 items, $A + B$ should return 8.

```
class Backpack(object):
    # ...
    def __add__(self, other):
        """Add the number of contents of each Backpack."""
        return len(self.contents) + len(other.contents)
```

Using the + binary operator on two `Backpack` objects calls the class's `__add__()` method. The object on the left side of the + is passed in to `__add__()` as `self` and the object on the right side of the + is passed in as `other`.

```
>>> pack1 = Backpack("Rose", "red")
>>> pack2 = Backpack("Carly", "cyan")

# Put some items in the backpacks.
>>> pack1.put("textbook")
>>> pack2.put("water bottle")
>>> pack2.put("snacks")

# Now add the backpacks like numbers
>>> pack1 + pack2                  # Equivalent to pack1.__add__(pack2).
3
```

Subtraction, multiplication, division, and other standard operations may be similary defined with their corresponding magic methods (see Table 4.1).

Comparisons

Magic methods also facilitate object comparisons. For example, the `__lt__()` method corresponds to the < operator. Suppose one backpack is considered "less" than another if it has fewer items in its list of contents.

```
class Backpack(object)
    # ...
    def __lt__(self, other):
        """Compare two backpacks. If 'self' has fewer contents
        than 'other', return True. Otherwise, return False.
        """
        return len(self.contents) < len(other.contents)
```

Now using the < binary operator on two `Backpack` objects calls `__lt__()`. As with addition, the object on the left side of the < operator is passed to `__lt__()` as `self`, and the object on the right is passed in as `other`.

```
>>> pack1 = Backpack("Maggy", "magenta")
>>> pack2 = Backpack("Yolanda", "yellow")

>>> pack1.put('book')
>>> pack2.put('water bottle')
>>> pack1 < pack2
False

>>> pack2.put('pencils')
>>> pack1 < pack2                  # Equivalent to pack1.__lt__(pack2).
True
```

Other standard comparison operators also have corresponding magic methods and should be implemented similarly (see Table 4.1). Note that comparison methods should return either `True` or `False`, while arithmetic methods like `__add__()` might return a numerical value or another kind of object.

Method	Operation	Operator
`__add__()`	Addition	`+`
`__sub__()`	Subtraction	`-`
`__mul__()`	Multiplication	`*`
`__div__()`	Division	`/`
`__lt__()`	Less than	`<`
`__le__()`	Less than or equal to	`<=`
`__gt__()`	Greater than	`>`
`__ge__()`	Greater than or equal to	`>=`
`__eq__()`	Equal	`==`
`__ne__()`	Not equal	`!=`

Table 4.1: Common magic methods for arithmetic and comparisons. What each of these operations do, or should do, is up to the programmer and should be carefully documented. See `https://docs.python.org/2/reference/datamodel.html#special-method-names` for more methods and details.

Problem 3. Endow the `Backpack` class with two additional magic methods:

1. The `__eq__()` magic method is used to determine if two objects are equal, and is invoked by the `==` operator. Implement the `__eq__()` magic method for the `Backpack` class so that two `Backpack` objects are equal if and only if they have the same name, color, and number of contents.

2. The `__str__()` magic method is used to produce the string representation of an object. This method is invoked when an object is cast as a string with the `str()` function, or when using the `print` statement. Implement the `__str__()` method in the `Backpack` class so that printing a `Backpack` object yields the following output:

```
Owner:      <name>
Color:      <color>
Size:       <number of items in contents>
Max Size:   <max_size>
Contents:   [<item1>, <item2>, ...]
```

   (Hint: Use the tab and newline characters `'\t'` and `'\n'` to help align output nicely.)

Achtung!

Comparison operators are not automatically related. For example, for two backpacks `A` and `B`, if `A==B` is `True`, it does not automatically imply that `A!=B` is `False`. Accordingly, when defining `__eq__()`, one should also define `__ne__()` so that the operators will behave as expected, perhaps by calling `__eq__()` itself:

```
def __ne__(self, other):
    return not self == other
```

Problem 4. Create your own `ComplexNumber` class that supports the basic operations of complex numbers.

1. Complex numbers have a real and an imaginary part. The constructor should therefore accept two numbers. Store the first as `self.real` and the second as `self.imag`.
2. Implement a `conjugate()` method that returns the object's complex conjugate (as a new `ComplexNumber` object). Recall that $\overline{a+bi} = a-bi$.
3. Add the following magic methods:
   (a) The `__abs__()` magic method determines the output of the built-in `abs()` function (absolute value). Implement `__abs__()` so that it returns the magnitude of the complex number. Recall that $|a+bi| = \sqrt{a^2+b^2}$.
   (b) Implement `__lt__()` and `__gt__()` so that `ComplexNumber` objects can be compared by their magnitudes. That is, $(a+bi) < (c+di)$ if and only if $|a+bi| < |c+di|$, and so on.
   (c) Implement `__eq__()` and `__ne__()` so that two `ComplexNumber` objects are equal if and only if they have the same real and imaginary parts.
   (d) Implement `__add__()`, `__sub__()`, `__mul__()`, and `__div__()` appropriately. Each of these should return a new `ComplexNumber` object.
   (Hint: use the complex conjugate to implement division).

Compare your class to Python's built-in `complex` type. How can your class be improved to act more like true complex numbers?

Additional Material

Static Attributes

Attributes that are accessed through `self` are called *instance* attributes because they are bound to a particular instance of the class. In contrast, a *static* attribute is one that is shared between all instances of the class. To make an attribute static, declare it inside of the `class` block but outside of any of the class's functions, and do not use `self`. Since the attribute is not tied to a specific instance of the class, it should be accessed or changed via the class name.

For example, suppose our Backpacks all have the same brand name.

```
class Backpack(object):
    # ...
    brand = "Adidas"
```

Changing the brand name changes it on every backpack instance.

```
>>> pack1 = Backpack("Bill", "blue")
>>> pack2 = Backpack("William", "white")
>>> print pack1.brand, pack2.brand
Adidas Adidas

# Change the brand name for the class to change it for all class instances.
>>> print Backpack.brand
Adidas
>>> Backpack.brand = "Nike"
>>> print pack1.brand, pack2.brand
Nike Nike
```

Static Methods

Individual class methods can also be static. A static method can have no dependence on the attributes of individual instances of the class, so there can be no references to `self` inside the body of the method and `self` is **not** listed as an argument in the function definition. Thus only static attributes and other static methods are available within the body of a static method. Include the tag `@staticmethod` above the function definition to designate a method as static.

```
class Backpack(object):
    # ...
    @staticmethod
    def origin():
        print "Manufactured by " + Backpack.brand + ", inc."
```

```
# We can call static methods before instantiating the class.
Backpack.origin()
Manufactured by Nike, inc.

# The method can also be accessed through class instances.
>>> pack = Backpack("Larry", "lime")
>>> pack.origin()
Manufactured by Nike, inc.
```

To practice these principles, consider adding a static attribute to the `Backpack` class to serve as a counter for a unique ID. In the constructor for the `Backpack` class, add an instance variable called `self.ID`. Set this ID based on the static ID variable, then increment the static ID so that the next `Backpack` object will have a new ID.

More Magic Methods

Explore the following magic methods and consider how they could be implemented for the `Backpack` class.

Method	Operation	Trigger Function
`__repr__()`	Object representation	`repr()`
`__nonzero__()`	Truth value	`bool()`
`__len__()`	Object length or size	`len()`
`__getitem__()`	Indexing and slicing	`self[index]`
`__setitem__()`	Assignment via indexing	`self[index] = x`
`__iter__()`	Iteration over the object	`iter()`
`__reversed__()`	Reverse iteration over the object	`reversed()`
`__contains__()`	Membership testing	`in`

See `https://docs.python.org/2/reference/datamodel.html` for more details and documentation on all magic methods.

Hashing

A *hash value* is an integer that identifies an object. The built-in `hash()` function calculates an object's hash value by calling its `__hash__()` magic method.

In Python, the built-in `set` and `dict` structures use hash values to store and retrieve objects in memory quickly. Thus if an object is unhashable, it cannot be put in a set or be used as a key in a dictionary.

```
# Get the hash value of a hashable object.
>>> hash("math")
-8321016616855971138

# Create a dictionary and attempt to get its hash value.
>>> example = {"math": 320}
>>> hash(example)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

If the `__hash__()` method is not defined, the default hash value is the object's memory address (accessible via the built-in function `id()`) divided by 16, rounded down to the nearest integer. However, two objects that compare as equal via the `__eq__()` magic method must have the same hash value. A simple `__hash__()` method for the `Backpack` class that conforms to this rule and returns an integer might be the following.

```
class Backpack(object):
    # ...
    def __hash__(self):
        return hash(self.name) ^ hash(self.color) ^ hash(len(self.contents))
```

The caret operator `^` is a bitwise XOR (exclusive or). The bitwise AND operator `&` and the bitwise OR operator `|` are also good choices to use.

Achtung!

If a class does not have an `__eq__()` method it should **not** have a `__hash__()` method either. Furthermore, if a class defines `__eq__()` but not `__hash__()`, its instances may be usable in hashed collections like sets and dictionaries (because of the default hash value), but two instances that compare as equal may or may not be recognized as distinct in the collection.