3 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 write clean, efficient, well-designed programs. In this lab we learn how to define and use Python classes. In subsequents labs, we will often create customized classes for use in algorithms.*

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:
    """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 list of contents.

        Parameters:
            name (str): the name of the backpack's owner.
        """
        self.name = name                # Initialize some attributes.
        self.contents = []
```

An *attribute* is a variable stored within an object. This `Backpack` class has two attributes: `name` and `contents`. In the body of the class definition, attributes are assigned and 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 an actual backpack object, "call" the class name like a function. This triggers the constructor and returns a new *instance* of the class, an object whose type is the class.

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

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

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

> **Note**
>
> Every object in Python has some built-in attributes. For example, modules have a `__name__` attribute that identifies the scope in which it is being executed. If the module is being run directly, not imported, `__name__` is set to `"__main__"`. Therefore, any commands under an `if __name__ == "__main__":` clause are ignored when the module is imported.

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*.

```
class Backpack:
    # ...
    def put(self, item):
        """Add 'item' to the backpack's list of contents."""
        self.contents.append(item)  # Use 'self.contents', not just 'contents'.

    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 on an instantiation of the class.

```
# Add some items to the backpack object.
>>> my_backpack.put("notebook")         # my_backpack is passed implicitly to
>>> my_backpack.put("pencils")          # Backpack.put() as the first argument.
>>> my_backpack.contents
['notebook', 'pencils']

# Remove an item from the backpack.     # This is equivalent to
>>> my_backpack.take("pencils")         # Backpack.take(my_backpack, "pencils")
>>> my_backpack.contents
['notebook']
```

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

1. Modify the constructor so that it accepts three total arguments: `name`, `color`, and `max_size` (in that order). Make `max_size` a keyword argument that defaults to 5. Store each input as an attribute.

2. Modify the `put()` method to check that the backpack does not go over capacity. If there are already `max_size` items or more, print "No Room!" and do not add the item to the contents list.

3. Write 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`).

4. Documentation is especially important in classes so that the user knows what an object's attributes represent and how to use methods appropriately. Update (or write) the docstrings for the `__init__()`, `put()`, and `dump()` methods, as well as the actual class docstring (under `class` but before `__init__()`) to reflect the changes from parts 1-3 of this problem.

To ensure that your class works properly, write a test function outside outside of the `Backpack` class that instantiates and analyzes a `Backpack` object.

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

Inheritance

To create a new class that is similar to one that already exists, it is often better to *inherit* the methods and attributes from an existing class rather than create a new class from scratch. 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*. To define a subclass, add the name of the superclass as an "argument" at the end of the `class` declaration.

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 inside.
        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.

        Parameters:
            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 inside.
        """
        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 from the superclass need to be changed for the subclass, they can be overridden by defining them again in the subclass. New methods can be included normally.

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

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

    def weight(self):               # Define a new method just for knapsacks.
        """Calculate the weight of the knapsack by counting the length of the
        string representations of each item in the contents list.
        """
        return sum([len(str(item)) for item in self.contents])
```

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.

The built-in function `issubclass()` shows whether or not one class is derived from another. Similarly, `isinstance()` indicates whether or not an object belongs to a specified class hierarchy. Finally, `hasattr()` shows whether or not a class or object **has** a specified attribute or method.

```
>>> from object_oriented import Knapsack
>>> my_knapsack = Knapsack("Brady", "brown")

# A Knapsack is a Backpack, but a Backpack is not a Knapsack.
>>> print(issubclass(Knapsack, Backpack), issubclass(Backpack, Knapsack))
True False
>>> isinstance(my_knapsack, Knapsack) and isinstance(my_knapsack, 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 Knapsack class has a weight() method, but the Backpack class does not.
>>> print(hasattr(my_knapsack, 'weight'), hasattr(my_backpack, 'weight'))
True False

# The dump method is inherited from the Backpack class.
>>> my_knapsack.dump()
>>> my_knapsack.contents
[]
```

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

1. Override 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. Override the `dump()` method so that both the contents and the fuel tank are emptied.
4. Write clear, detailed docstrings for the class and each of its methods.

NOTE

All classes are subclasses of the built-in `object` class, even if no parent class is specified in the class definition. In fact, the syntax "`class ClassName(object):`" is not uncommon (or incorrect) for the class declaration, and is equivalent to the simpler "`class ClassName:`".

Magic Methods

A *magic method* is a special method used to make an object behave like a built-in data type. Magic methods begin and end with two underscores, like the constructor `__init__()`. Every Python object is automatically endowed with several magic methods, which can be revealed through IPython.

```
In [1]: run object_oriented.py

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

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

In [3]: b.__        # Press 'tab' to see magic methods and hidden attributes.
          __add__()           __getattribute__    __new__()
          __class__           __gt__              __reduce__()
          __delattr__         __hash__            __reduce_ex__()
          __dict__            __init__()          __repr__
          __dir__()           __init_subclass__() __setattr__
          __doc__             __le__              __sizeof__()
          __eq__              __lt__()            __str__
          __format__()        __module__          __subclasshook__()
          __ge__              __ne__              __weakref__
```

> **Note**
>
> Many programming languages distinguish between *public* and *private* variables. In Python, all attributes are public, period. However, attributes that start with an underscore are hidden from the user, which is why magic methods do not show up at first in the preceding code box.

The more common magic methods define how an object behaves with respect to addition and other binary operations. For example, 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. To incorporate this idea, implement the `__add__()` magic method.

```
class Backpack:
    # ...
    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")

# Add the backpacks together.
>>> pack1 + pack2                    # Equivalent to pack1.__add__(pack2).
3
```

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):
        """If 'self' has fewer contents than 'other', return True.
        Otherwise, return False.
        """
        return len(self.contents) < len(other.contents)
```

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

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

Comparison methods should return either `True` or `False`, while methods like `__add__()` might return a numerical value or another kind of object.

Method	Arithmetic Operator	Method	Comparison Operator
`__add__()`	`+`	`__lt__()`	`<`
`__sub__()`	`-`	`__le__()`	`<=`
`__mul__()`	`*`	`__gt__()`	`>`
`__pow__()`	`**`	`__ge__()`	`>=`
`__truediv__()`	`/`	`__eq__()`	`==`
`__floordiv__()`	`//`	`__ne__()`	`!=`

Table 3.1: 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. For more methods and details, see `https://docs.python.org/3/reference/datamodel.html#special-method-names`.

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 returns the string representation of an object. This method is invoked by `str()` and used by `print()`. Implement the `__str__()` method in the `Backpack` class so that printing a `Backpack` object yields the following output (that is, construct and return the following string).

   ```
   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 align output nicely.)

Achtung!

Magic methods for comparison are **not** automatically related. For example, even though the `Backpack` class implements the magic methods for `<` and `==`, two `Backpack` objects cannot respond to the `<=` operator unless `__le__()` is explicitly defined. The exception to this rule is the `!=` operator: as long as `__eq__()` is defined, `A!=B` is `False` if and only if `A==B` is `True`.

Problem 4. Write a `ComplexNumber` class from scratch.

1. Complex numbers are denoted $a+bi$ where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}$. Write the constructor so it accepts two numbers. Store the first as `self.real` and the second as `self.imag`.
2. The *complex conjugate* of $a+bi$ is defined as $\overline{a+bi} = a-bi$. Write a `conjugate()` method that returns the object's complex conjugate as a new `ComplexNumber` object.
3. Add the following magic methods:
   (a) Implement `__str__()` so that $a+bi$ is printed out as `(a+bj)` for $b \geq 0$ and `(a-bj)` for $b < 0$.
   (b) The *magnitude* of $a+bi$ is $|a+bi| = \sqrt{a^2+b^2}$. 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.
   (c) Implement `__eq__()` 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 `__truediv__()` appropriately. Each of these should return a new `ComplexNumber` object.

Write a function to test your class by comparing it to Python's built-in `complex` type.

```
def test_ComplexNumber(a, b):
    py_cnum, my_cnum = complex(a, b), ComplexNumber(a, b)

    # Validate the constructor.
    if my_cnum.real != a or my_cnum.imag != b:
        print("__init__() set self.real and self.imag incorrectly")

    # Validate conjugate() by checking the new number's imag attribute.
    if py_cnum.conjugate().imag != my_cnum.conjugate().imag:
        print("conjugate() failed for", py_cnum)

    # Validate __str__().
    if str(py_cnum) != str(my_cnum):
        print("__str__() failed for", py_cnum)
    # ...
```

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 methods, and do not use `self`. Since the attribute is not tied to a specific instance of the class, it may be accessed or changed via the class name without even instantiating the class at all.

```
class Backpack:
    # ...
    brand = "Adidas"                # Backpack.brand is a static attribute.
```

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

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

Static Methods

Individual class methods can also be static. A static method cannot be dependent 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 static methods only have access to static attributes and other static methods. Include the tag `@staticmethod` above the function definition to designate a method as static.

```
class Backpack:
    # ...
    @staticmethod
    def origin():                   # Do not use 'self' as a parameter.
        print("Manufactured by " + Backpack.brand + ", inc.")
```

```
# Static methods can be called without instantiating the class.
>>> Backpack.origin()
Manufactured by Nike, inc.

# The method can also be accessed by individual 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 different ID.

More Magic Methods

Consider how the following methods might be implemented for the `Backpack` class. These methods are particularly important for custom data structure classes.

Method	Operation	Trigger Function
`__bool__()`	Truth value	`bool()`
`__len__()`	Object length or size	`len()`
`__repr__()`	Object representation	`repr()`
`__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/3/reference/datamodel.html#special-method-names` for more details and documentation on all magic methods.

Hashing

A *hash value* is an integer that uniquely 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. If an object is *unhashable*, it cannot be put in a set or be used as a key in a dictionary. See `https://docs.python.org/3/glossary.html#term-hashable` for details.

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:
    # ...
    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.

See `https://docs.python.org/3/reference/datamodel.html#object.__hash__` for more on hashing.