Lab 10: Classes

CSE/IT 107

NMT Department of Computer Science and Engineering

"Who are you? How did you get in my house?"

— Donald Knuth

---

1 Introduction

This week, we will have a brief introduction to Object Oriented Programming by way of Python's classes.

2 Classes and Objects

In Python, classes are a way to group together functions and attributes that are closely related to one another. A `class` is defined similarly to a function, though it uses the `class` keyword rather than `def`.

```
>>> class Test:
...     i = 5
...     def hi():
...         print('Hello world.')
...
>>> Test.i
5
>>> Test.hi()
Hello world.
```

Once a class has been declared, an *Object* or *Instance* of that class can be created. An object is an independent copy of the class, with all of the same attributes and functions that were declared with the class that the object was created from. This is the primary use for classes, as it allows for you to easily create a group of objects that store the same kind of data.

When doing this, it is important to declare a special function called `__init__` inside the class. This function is called when creating the new class and can be used to initialize values for the new object:

```
1  class Test:
2      def __init__(self):
3          self.x = 10
4
5      def set(self, x):
6          self.x = x
7
8  a = Test()
9  b = Test()
10 print(a.x, b.x) #10 10
11 a.set(20)
12 print(a.x, b.x) #20 10
```

Note the use of the variable `self` in the function declarations. This variable should be first in any function declared that needs to be used on a function level. This variable refers to the object that the function is being called on. For example, in line 11, `self` would refer to `a`. Note that `a.set(20)` is equivalent to `Test.set(a, 20)`. The former is preferred because it is shorter and more readable. Additionally, a function that refers operates on an instance of the class (that is, has `self` as the first argument) are called methods rather than functions.A ny variable referred to with `self.variablename` will stay with the object, while other variables will stay local to the function.

`__init__` can be passed additional arguments in addition to `self` so that the new objects can be initialized differently.

The special method `__str__(self)` can be defined to override how the object is displayed when printed. By default, printing an object will display something like this:

```
>>> class Test:
...     pass
...
>>> a = Test()
>>> print(a)
<__main__.Test object at 0x7f2b806cc9e8>
```

However, if we define the method, it will instead print whatever we want it to. `__str__` should always return a string.

```
class Test:
    def __init__(self):
        self.x = 5

    def __str__(self):
        return 'x = {}'.format(self.x)

a = Test()
print(a)
```

Try running the above code. It should print out `x = 5`. Overriding `__str__` makes our output code far simpler when printing objects.

3 Inheritance

A class can be built on another class. This is commonly done when multiple classes should logically be different but share common functionality. A class that inherits from another class is considered to be a subclass of its parent class. The subclass gains all the functions and attributes of its parent class.

An example might be that both Books and Articles could be considered Publications. Both would have a title and an author, but only the book would have a chapter count.

```
class Publication:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return '{} by {}'.format(self.title, self.author)

class Book(Publication):
    def __init__(self, title, author, chapters):
        Publication.__init__(self, title, author)
        self.chapters = chapters

class Article(Publication):
    def __init__(self, title, author, magazine):
```

```
        Publication.__init__(self, title, author)
        self.magazine = magazine

    def __str__(self):
        return '{} by {}, published in {}'.format(self.title, self.author,
            self.magazine)

b = Book('Title', 'Author', 100)
a = Article('Other Title', 'Other Author', 'Some Magazine')

print(b) #Title by Author
print(a) #Other Title by Other Author, published in Some Magazine
```

Note that `Article` creates its own version of the `__str__` method, while `Book` does not. This means that `Book` simply uses the method that it inherits from `Publication`.

Also note that each of the subclasses call the constructors of the parent class. This is required to properly initialize the attributes of the object. Though it is not strictly required, doing otherwise would require duplicating the contents of the parent class's constructor in the subclass. A class can inherit from multiple other classes. In that case, it would be expected to call the constructors for both of its parent classes.

Sometimes you will want to check if an object belongs to a specific class. This can be done with the `isinstance` function. This function will return `True` if the given object is an instance of the given class or of a subclass of that class.

```
class A:
    pass
class B:
    pass
class C(A):
    pass
class D(C, B):
    pass

print(isinstance(A(), A)) #True
print(isinstance(B(), A)) #False
print(isinstance(C(), A)) #True
print(isinstance(D(), B)) #True
```

4 Iteration

By defining a couple of special methods, we can make an object of any class that we create iterable. These two functions are `__iter__(self)` and `__next__(self)`. These functions are used by anything that uses iterables (such as `for` loops). `__iter__` is called when the loop starts and is expected to return an object with `__next__` defined. Thus, we can usually just `return self` if we've defined it ourselves. Then, `__next__` is called repeatedly until it raises an exception called `StopIteration`. Using this knowledge we can write our own version of `range()`:

```python
class MyRange:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        self.current = None
        return self

    def __next__(self):
        if self.current is None:
            self.current = self.start
        else:
            self.current += 1

        if self.current >= self.end:
            raise StopIteration
        else:
            return self.current

r = MyRange(1, 10)
for i in r:
    print(i) #prints 1, 2, 3, ..., 9
```

With a bit more complex logic in our `__next__` we can do some pretty complicated things with `for` loops!

5 Exercises

Exercise 5.1 (library.py).
Create a module called `library`. This module should include two classes as well as methods for each. The provided file `library_test.py` will use the contents of `library.py`. Do not edit `library_test.py` for your submission, though you are free to comment out lines using parts of `library.py` that you have not implemented yet. The contents of `library.py` should be as follows:

class Library A `Library` object stores multiple `Book` objects and has methods to return statistics about those books. When printed, the object should list all of the books it contains in alphabetical order by author (as listed. You do not need to split into first and last names).

- **def add_book(self, book)** This method takes in either a line of text or a book object. If the input is a line of text, then it creates a new `Book` object out of it. The line of text will be formatted as are the lines in `library.txt` and should be parsed as needed to form a `Book` object. The object (whether given or created) is then stored inside the library.
- **def get_authors(self)** This method returns a list of all the authors who have works contained in the library. No author should appear more than once.
- **def get_books_per_author(self)** This method returns a dictionary. Each author should be a key in the dictionary. The value for each author should be the number of books by that author in the library.

class Book A `Book` object stores the author and title of a book. When printed, the object should display both the title and author of the book. The constructor should take in two arguments: the title of the book and the author of the book.

You are free (and encouraged) to include any additional methods that you may need to accomplish the tasks given.

Exercise 5.2 (myturtle.py).
Write a class that inherits from `turtle.Turtle`. Overwrite the `forward`, `backward`, `left`, and `right` methods so that, if given a negative value, they cause no movement to occur. Add a new method `regular_polygon` that takes as arguments a number of sides anda length for the sides, then draws out the regular polygon fulfilling those requirements.

Write a program using your replacement `Turtle` class that creates multiple objects of that class and tests out each of the methods you have written.

Exercise 5.3 (tree.py).
Write a class that implements a simple binary tree data structure. That is a tree that consists of a group of nodes, each of which has an optional "left" and "right" child. The class needs to include the following:

class Tree The class used to create objects representing trees. Each object will also represent one of the nodes of the tree. The constructor should take in one argument: the contents of that node. The contents can be anything and are not important for the scope of this exercise. The constructor should have two optional arguments: left and right trees. If given, these will be the left and right children of the new node. This allows for the joining of two trees into a single one with a new parent node.

self.left The left child of the node. May be `None` or a `Tree` object.

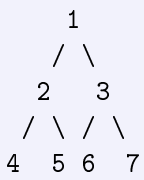
self.right The right child of the node. May be `None` or a `Tree` object.

self.datum The contents of the node.

def height(self) Returns the height of the tree that this node is the root of. This is equal to one more than the largest height of the children trees. If a child tree is `None` then its height is 0.

def add_item(self, item) Create a new `Tree` object and add it to the tree. The new object should be added to the child tree with the smallest height (remember that a `None` tree has a height of 0!). If there is a tie, favor the left tree.

Additionally, you should implement `__iter__(self)` and `__next__(self)` such that the tree can be used in a for loop. When used in a for loop, the nodes of the tree should be accessed in-order. This means that they should be accessed from left to right. That is, if we have a tree that looks like this:



Then the nodes should be accessed in the order 4, 2, 5, 1, 6, 3, 7. To accomplish this, you will need to keep track of where in the traversal the loop currently is, as well as `raise StopIteration` when necessary.

Hint: You may find using a stack to track where you are in the tree to be useful!

Once you have it using, you should be able to use `list` to convert your tree into a list.

6 Submitting

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

`cse107_firstname_lastname_lab10.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit