CMU MSP 36602 Advanced Python: Classes 22 Apr 2019

1) **You are already used to using Python classes**

   a. E.g., `x = [1, 5, 7]` creates an object of class "list".

   b. Then `x.pop()` uses the pop **method** on x to remove and return the last value.

   c. `type(x)` **shows the class** of x

   d. `dir(x)` **shows the attributes and methods** of a "list" (if x is a list)

   e. `dir(list)` works identically to `dir(x)` when `type(x)` is "list".

   f. You may need to use, e.g., `type(x.pop)` to see if "pop" is a **method vs. data attribute**

   g. Note that assignment may well not be needed or appropriate when altering class data with a class method, e.g., `x = [1, 2]; x.clear(); x` shows an empty list.

   h. ***New for us***: The "double underscore" methods correspond to special "magic methods" related to the implementations of the built-in functions for the class. E.g., a class may define a `__len__()` method (sometimes called "dunder len"). This allows the len() function to work (according to user specifications) on the class.

      E.g., `dir(list)` show us that lists implement len(), "<" (using "__lt__"), etc. Some built-in functions are implemented via other dunders, e.g., iteration requires `__iter__` and `__next__` and `max()` requires iteration and `__lt__` (see below).

2) Writing your own Python classes

a. **Planning is key.** Break larger programming tasks into separate pieces, including classes, based on DRY, reusability, separation/encapsulation, and clarity.

b. The basic Python 3 code to **start a class definition** is: `class Spam(object):`

   i. The "(object)" is optional. Anything inside of parentheses indicates that this class "inherits" functionality from another class, possibly a "virtual class", i.e., one that can only be inherited from and not instantiated. If omitted, "(object)" is assumed, and 'object' is a simple class that makes simple definitions of the comparison operator methods and the defines __str__() as __repr__().

   ii. The convention is UpperCamelCase for the class name.

c. To **create an instance of a class** that you have defined, use code like `spam = Eggs()` to create an instance of the class "Eggs" called "spam". Then `ham = Eggs()` creates a second instance. You can also initialize the instantiation with arguments, e.g., `bacon = Eggs(with_meat=TRUE, slices=4)` (see below). If you are a vegan use, e.g., `breakfast = MealSansAnimalProducts(oatmeal=True, fruit="apple")`.

d. **Classes have access to three types of variables**: global variables (rarely appropriate), the automatic variables created from the arguments to each method defined in the class (but only while that method is running), and the special "**self**" variable that corresponds to variables defined separately for each instantiation of the class. E.g., you may have many instantiations of an Employee class, and each will have a different value for self.name, but they would all share a common method (function) called self.get_name(). In addition, you do have access to global built-in functions, and the global classes and functions you have imported. (Note that the variable name "self" is only a very strong convention, but not a keyword or requirement.)

e. To **define a method for a class** you use the usual `def spam(foo):` syntax, but the first argument is required and special, and should be called "self". This is used to ***give your function (method) access to the contents of the variables in the current instantiation of the class***. When you access a method such as `increment_salary()` for an instantiation of a class such as "Employee" named "howard" in code outside your class, you typically use a form like howard.increment_salary(5000). This is actually translated by Python into `Employee.increment_salary(howard, 5000)`, which explains why every function definition uses "self" as its first argument.

f. **A first class definition**: creating a class with some extra string functionality (similar to the "string" module)

In file "StringPlus.py":

```
class StringPlus(object):
    """ Functions to get a list of digits, lower case letters and
        upper case letters.  Also, set and get your name.
    """
    def digits(self):
        """ Return a list of the 10 digits """
        return list("0123456789")

    def lower_letters(self, count):
        """ A list of the first 'count' lower case letters """
        return list("abcdefghijklmnopqrstuvwxyz"[:count])

    def upper_letters(self, count):
        """ A list of the first 'count' upper case letters """
        return list("ABCDEFGHIJKLMNOPQRSTUVWXYZ"[:count])

    def set_name(self, name):
        """ Set your name for later retrieval """
        self.my_name = name
        return self

    def get_name(self):
        """ Retrieve your name (fails if not first set!) """
        return self.my_name
```

In another file (testStringPlus.py):

```
from StringPlus import StringPlus
sp = StringPlus()
print("isinstance(x1, StringPlus):", isinstance(sp, StringPlus))
print("".join(sp.digits()))
print(sp.lower_letters(5) + sp.upper_letters(2))
print(sp.get_name())  # fails!
sp.set_name("Howard")
print(sp.get_name())
print(sp.my_name)
sp.set_name("Seltman").get_name()
help(sp)
help(sp.get_name)
```

Important note:

If you change the code for a module, just re-running `import myModule` will not activate the new code. Instead, when developing a module, including a new class definition, you should run `import importlib`, and then use `importlib.reload(myModule)` to activate the new code.

g. Adding functionality to the class with magic methods

i. An `__init__(self[, argument[, …]])` method is used to **initialize any per-instantiation data variables, as well as to allow arguments so that each instantiation starts differently**. E.g., a Book class might use arguments to set the "title", "ISBN", "author" and "year", but it will always set "condition" to "new" and "checked_out" to False. Because you don't know the order that the methods will be called, it is good to set appropriate defaults for every class level variable. E.g., the StringPlus class should have had in `__init__(self)` method that included `self.my_name = None`.

ii. The default `__repr__(self)` method is the string "`<class 'foo'>`". The intent is that you provide a method that shows how to recreate `foo`, if possible and reasonable. This method is used if you write `repr(foo)` or just `foo`.

iii. The `__str__(self)` method is provided to implement **`str(foo)`**, which is intended to be a "human" oriented representation of the object. By default, `repr()` is used, but you should write your own `__str__` if something better than `repr(foo)` is possible. You must return a `str`.

iv. A `__len__(self)` method is needed to allow **`len(foo)`** to work on `foo` instantiations of your object. Note that there is no general idea of length for all possible class objects.

v. A `__getitem__(self, key)` method should be included if you want users to be able to use `foo[index]` on a `foo` instantiation of your object.

h. Breakout: Make a class that implements the Fibonacci series. Initiate with the length (>=2) and in the initiation, compute and store the series as a list element called "sequence". Implement _repr_(), __str__(), __len__(), __getitem__().

i. Adding more functionality to the class with magic methods

i. You may want your class to **support iteration**, e.g., be the object after "in" in `for index in object:`. To do this you must implement `__iter__(self)` in your class. This method ***initializes the iteration***, e.g., sets an internal indexing variable to 0. It must ***return a class object that contains a `__next__()` method***. Usually this is just "self" with `__next__()` in your class.

What happens "under the hood": The "for loop" (and other related functionalities) work by first calling `iter(foo)` when you write `for index in foo:`, which actually calls `foo.__iter__()`. The return value has a __next__() method, and this method is called repeatedly and `index` is set to `__next__()`'s return value (repeatedly). The whole thing stops when `__next__()` throws the `StopIteration` exception.

What you must implement: `__next__()` should return a single value (typically based on an index set to 0 by `__iter__()`, then it usually increments the index. Usually there is an `if` statement at the beginning of the method that throws `StopIteration` if there are no more values to return.

An alternative to providing `__iter__()` and `__next__()`is to just implement `__getitem__()`.

ii. `__eq__()` and `__ne__()` define equality and inequality. They are used when you have code like `x==y` or `x!=y`. The defaults just compare the memory address of the objects! Usually you want to compare some or all of the instance variables and the class of the second object. If you run `x == y`, and x is of class "foo", then by definition, you are running `foo.__eq__(self, other=y)`. It is possible that y is of a different class and the it has the same attributes as x and the same values, but you probably don't want to call x and y as equal. Usually you need to check `isinstance(y, self.__class__)` as part of your equality test.

iii. To support **set()s** of your class objects and **dictionaries** that use your class objects as the key, you need to implement the `__eq__(self, other)` method (which also defines `foo == spam`) and the `__hash__(self)` method which supports the `hash(foo)` function. [Discussion of hashing.] Your object's data usually consists of some collection of objects that are likely to have their own hash() function pre-defined, so the task is really to write a method that combines several hash values. See the example below for one good method.

iv. To support **`min()`** and **`max()`** and to allow use of **`foo1 < foo2`**, define `__lt__`(self, other) which defines the meaning of less than in the form of `self < other`.

v. If you have `lt`, you will probably also want **`gt`**, **`le`**, **`ge`**, **`eq`**, and **`ne`**.

vi. You can implement **addition** (and thus, **`sum()`**) with `__add__(self, other)`and `__radd__(self, other)`. You can implement += with `__iadd__(self, other)`; The `__radd__()` function is tried when you run `x+y`, but the class of x does not have and `_add_()` function.

j. Example: Class definition with magic methods

```
"""
Rectangle is a test class for 36602, H. Seltman, April. 2019
Show basic methods desired for most classes.
"""


class Rectangle(object):
    """ A class that holds width and length, computes area,
        and implements math by constructing a square with the
        appropriate area.
    """
    def __init__(self, length, width):
        """ Class is initialized with length and width """
        self.set_length(length)
        self.set_width(width)

    def get_length(self):
        return self.length

    def set_length(self, length):
        if not isinstance(length, (int, float)) or length <= 0:
            raise ValueError
        self.length = length

    def get_width(self):
        return self.width

    def set_width(self, width):
        if not isinstance(width, (int, float)) or width <= 0:
            raise ValueError
        self.width = width

    def get_area(self):
        """ Compute the rectangle's area """
        return self.length * self.width

    def __repr__(self):
        """ Standard representation of a Rectangle class object """
        return "Rectangle(length=" + str(self.length) + \
            ", width=" + str(self.width) + ")"

    def __str__(self):
        """ Human representation of a Rectangle class object """
        return str(self.length) + " x " + str(self.width) + \
            " rectangle"

    def __len__(self):
        """ I choose to define length as floor(perimeter) """
        return int(2.0 * (self.length + self.width))
```

```
def __getitem__(self, key):
    """ Implement indexing as foo[L/W] (flexibly) """
    if not isinstance(key, str):
        raise TypeError
    key = key[:1].upper()
    if key not in ('L', 'W'):
        raise IndexError
    if key == 'L':
        return self.length
    else:
        return self.width

def __iter__(self):
    """ Silly implementation of the first half of iteration """
    self.__index = 0
    return self

def __next__(self):
    """ Silly implementation of the second half of iteration """
    if self.__index >= self.get_area():
        raise StopIteration
    self.__index = self.__index + 1
    return self.__index

def __key(self):
    """ 'Private' method to define a key as a tuple """
    return (self.length, self.width)

def __eq__(self, other):
    """ Define equality as same length and width """
    return isinstance(other, self.__class__) and \
           self.__key() == other.__key()

def __hash__(self):
    """ Hash based on length and width """
    # simplest method, perhaps inefficient
    return hash(self.__key())

def __lt__(self, other):
    """ Area based '<' """
    return self.get_area() < other.get_area()

def __ne__(self, other):
    """ Inequality based on different length or width """
    return not isinstance(other, self.__class) or \
           self.__key() != other.__key()

def __gt__(self, other):
    """ Area based '>' """
    return self.get_area() > other.get_area()

def __le__(self, other):
    """ Area based '<=' """
    return self.get_area() <= other.get_area()
```

```
def __ge__(self, other):
    """ Area based '>=' """
    return self.get_area() >= other.get_area()

def __add__(self, other):
    """ Add means make a square with area equal to their sum """
    area = self.get_area() + other.get_area()
    new = Rectangle(1, 1)  # arguments are arbitrary
    new.length = area**0.5
    new.width = new.length
    return new

def __radd__(self, other):
    """ Reverse add adds a Rectangle and either a numeric or
        another Rectangle
    """
    if isinstance(other, (int, float)):
        new = Rectangle(1, 1)  # arguments are arbitrary
        new.length = (other + self.get_area())**0.5
        new.width = new.length
        return new
    else:
        return self + other

def __sub__(self, other):
    """ Subtract returns a square Rectangle with the area
        equal to the difference of areads """
    if other.get_area() >= self.get_area():
        return ValueError
    new = Rectangle(1, 1)  # arguments are arbitrary
    new.length = (self.get_area() - other.get_area())**0.5
    new.width = new.length
    return new

def __mul__(self, other):
    """ Multiply returns a square Rectangle with the area
        equal to the product of areas """
    new = Rectangle(1, 1)  # arguments are arbitrary
    new.length = (self.get_area() * other.get_area())**0.5
    new.width = new.length
    return new

def __truediv__(self, other):
    """ Divide returns a square Rectangle with the area
        equal to the ratio of areas """
    new = Rectangle(1, 1)  # arguments are arbitrary
    new.length = (self.get_area() / other.get_area())**0.5
    new.width = new.length
    return new

def copy(self):
    return Rectangle(self.length, self.width)
```

```
if __name__ == "__main__":
    # Test through get_area()
    try:
        x1 = Rectangle()  # fails!
    except TypeError:
        print("TypeError: __init__() missing 2 required positional"
              + "arguments: 'length' and 'width'")
    try:
        x1 = Rectangle(0, 2)  # fails!
    except ValueError:
        print("ValueError")
    try:
        x1 = Rectangle(1, -2)  # fails!
    except ValueError:
        print("ValueError")
    x1 = Rectangle(1.5, 2)
    print("x1.get_length():", x1.get_length())
    print("x1.get_width():", x1.get_width())
    print("x1.get_area():", x1.get_area())
    x1.set_width(5.5)
    print("x1.get_width():", x1.get_width())
    print("x1.get_area():", x1.get_area())

    # Result of overriding default for __repr__()
    print("repr(x1):", repr(x1))  # same as x1 at the prompt

    # Result of overriding default for __str__()
    print("str(x1):", str(x1))
    print("x1:", x1)

    # Result of supplying __len__()
    print("len(x1)):", len(x1))

    # Result of supplying __getitem__()
    print("x1['L']:", x1['L'])
    print("x1['width']:", x1['width'])
    try:
        print("x1[0]:", x1[0])
    except TypeError:
        print("TypeError")
    try:
        print("x1['A']:", x1['A'])
    except IndexError:
        print("IndexError")

    # Result of implementing __iter__ and __next__
    x2 = Rectangle(width=4, length=2)
    print("x2:", x2, " area =", x2.get_area())
    print("Using x2 in a for loop:")
    for i in x2:
        print(i)
    print("tuple(x2):", tuple(x2))
```

```
# Result of implementing __eq__ and __hash__
print("x2 == Rectangle(width=2, length=4):",
      x2 == Rectangle(width=2, length=4))
print("x2 != Rectangle(width=2, length=4):",
      x2 != Rectangle(width=2, length=4))
print("set([x1, x2]):", set([x1, x2]))
d = {x1: 'first', x2: 'second'}
print("d[x2]:", d[x2])

# Result of implementing __lt__
x3 = Rectangle(2, 3)
print("x3:", x3)
print("area(x2) =", x2.get_area(), " area(x3) =", x3.get_area())
print("x2 < x3:", x2 < x3)
print("x2 > x3:", x2 > x3)
print("max(x1, x2, x3):", max(x1, x2, x3))
print("x2.__lt__(x1):", x2.__lt__(x1))

# Result of implementing additional comparisons
print("x2.__le__(x1):", x2.__le__(x1))
print("x2 <= x3:", x2 <= x3)
print("x2 >= x3:", x2 >= x3)

# Result of implementing __add__
x4 = Rectangle(1, 1)
print("x2:", x2)
print("x4:", x4)
print("x2 + x4:", x2 + x4)

# Results of implementing __radd__
print("sum([x4, x4, x4, x4]):", sum([x4, x4, x4, x4]))

# Result of implementing __sub__, __mul__, and __div__
print("Rectangle(1, 11) - Rectangle(1, 2):",
      Rectangle(1, 11) - Rectangle(1, 2))
print("Rectangle(1, 2) * Rectangle(1, 8):",
      Rectangle(1, 2) * Rectangle(1, 8))
print("Rectangle(2, 9) / Rectangle(1, 2):",
      Rectangle(2, 9) / Rectangle(1, 2))

# Result of implement copy()
x5 = x4
x4.set_length(11)
print("x5.get_length():", x5.get_length())
```

3) Summary of writing classes

a. Consider what data to store and how
b. Consider what methods to provide
c. Use set/get functions to hide implementation details
d. Write documentation strings as you go
e. Write an `__init__` method
f. Write `__str__` and perhaps `__repr__`
g. Write `__getitem__` if you want to support indexing
h. Write `__len__` if you want to support `len()`
i. Write `__iter__` and `__next__` if you want to support iteration
j. Write `__eq__` and `__hash__` if you want to support set() and dictionary keys
k. Write `__lt__` if you want to support `min()`, `max()` and "<"
l. Write `__add__` if you want to support "+"
m. Write `__radd__` if you want to support `sum()`
n. Write `__sub__`, `__mul__`, `__truediv__` if needed
o. Write `copy()` and `deepcopy()` if you want that functionality
p. Write each method along with test code