Math 260: Python programming in math

Classes and objects:
creating classes, rules and syntax

Classes

- So far, we've used only the built-in types like `float`, `list` and so on
- It is essential to be able to create new objects - this is done with **classes**

Suppose we want to make an object describing a pet with properties:

- `name` (the name of the pet)
- `species` (a string)
- `hunger` (number of hours until it needs food)
- `parents` (a tuple of two pets)

Here's the minimal code:

```
class Pet:  # style: capitalize class names
    def __init__(self, name, species, parent1=None, parent2=None):
        self.name = name
        self.species = species
        self.hunger = 0  # (always starts hungry)
        self.parents = (parent1, parent2)

m = Pet('mittens', 'cat')
print('{} is a {}'.format(m.name, m.species))  # mittens is a cat
```

Classes: initialization

Creating an object:

- An object of a certain class is called an '**instance**'
- An instance is created by calling the **constructor**: `obj = Pet(...)`
- The constructor is specified as the special function `__init__`
- Each instance contains its own 'instance variables' (initialized by `__init__`)

```
def __init__(self, name, species, parent1=None, parent2=None):
    self.name = name
    self.species = species
    self.hunger = 0  # (always starts hungry)
    self.parents = (parent1, parent2)
```

Variables in classes:

- In the class block, `self` refers to the **instance** calling that function
- For code **inside** the class scope, `self.var` refers to the instance variable
- For an instance `obj`, use `obj.var` (e.g. `m.species`)

```
m = Pet('mittens', 'cat')
f = Pet('bandit', 'ferret')
# m.name is 'mittens' and f.name is 'bandit'
```

Classes: functions and class variables

Class variables:

- variables defined in the class block itself (no `self.`)
- Each instance is initialized with these
- Typically used for variables shared between all instances or properties common to all things in the class

Methods:

- functions in classes are called **member functions** or **methods**
- `self` is always the first argument
- These live in the class scope
- `self` gives the method access to the calling instance's data
- When calling, `self` input is implied by the `obj.` syntax, e.g. `obj.eat(2)` means `eat(obj,2)`

```
class Pet:
    full = -2

    def __init__(self,name,species):
        self.name = name
        self.species = species
        self.hunger = 0

    def eat(self, amount):
        self.hunger -= amount
        if self.hunger < self.full:
            self.nap()

    def nap(self):
        self.hunger += 1
```

```
m = Pet('java','cat')
m.eat(3)
m.nap() #m.hunger is now -1
```

Classes: Class variables

Important note: **mutable** class variables are initialized only once, when the first instance is created.

Thus, mutable class variables are **shared by all instances**.

```python
class Blob:
    population = [0] #only set when the first Blob is made

    def __init__(self, mass):
        self.mass = mass  # not shared
        self.population[0] += 1  # shared!

family = []
for k in range(10):
    family.append(Blob(k))
```

- Intent: each Blob instance can see a global count of the number of Blobs.
- All blobs have references to the same `.population` list.
- Thus, `family[k].population` is [10] for each *k*
- Each blob can see the count!

Classes: special methods

There are ‘special methods’ that define object behavior (such as `__init__`).
Special functions are denoted with double underscores like `__init__`

- `__repr__` defines how `print(obj)` behaves (a string representation)

```
def __repr__(self):
    return "Name: " + self.name + ", a " + self.species
```

  Always good to define for debugging (so you can print the object)

- `__del__` (the **destructor**) is called when the object is destroyed

  - for instance: `obj` has a file open; then `del` closes the file

Important point: You **cannot** change how assignment (=) behaves.

- `a=b` assigns by reference (`a` and `b` point to the same object)
- objects are ‘**passed by reference**’, not by value:

```
m = Pet('mittens','cat')
def tranquilize(animal):
    animal.nap()
tranquilize(m)
```

Operator overloading

Built-in operators like + and [] can be (re)-defined for classes.
This is called **operator overloading**.

Example: Suppose we want pets to be comparable by their name.

- We need comparison operators ==, >, etc.
- Python needs == and one other (it then fills in the rest!)
- The special names for == and > are `__eq__` and `__gt__`

```
class Pet:
    ...
    def __eq__(self, r):
        return self.name == r.name

    def __gt__(self, r):  #... or some better ordering...
        return self.name > r.name
```

Now the object is 'comparable' - all the operators $<$, $<=$ etc. work.

```
pet1 = Pet('felix','cat')
pet2 = Pet('mittens','cat')
pet1 < pet2 # True
```

You can now plug lists of Pets into search functions (e.g. binary search)!

Operator overloading: when (not) to use it

- Other operators to overload include all arithmetic (*, *=, + etc.)
- For instance, we could define + for pets...

```python
def __add__(self, r): #(not really a good idea)
    return Pet("", self.species, parent1 = self.name, parent2 = r)
```

Best practices:

Only overload an operator when it is obvious what it should do. The above is an example of a bad usage. It's better to define a named function in most cases.
Exceptions include math objects like vectors (where + is obvious) - see HW.

Operator overloading: more options

Here are a few useful special methods:

- `__add__(self,r)` and `__sub__(self,r)` are $+$ and $-$

  ```
  a+b         means        a.__add__(b)
  ```

- `__getitem__`(self, key) defines 'getting' with `obj[key]`
- `__setitem__`(self, key, val) defines 'setting' with `obj[key]=`$\cdots$

  ```
  obj[key]        means        obj.__getitem__(key)

  obj[key]=v      means        obj.__setitem__(key, v)
  ```

- `__call__`(self,var) defines 'function call' (parentheses)

  ```
  obj(var)       means        obj.__call__(var)
  ```

Many others have reasonable default behavior - don't override unless you have a very good reason. For instance:

- `__setattr__` and `__getattr__` define the `obj.var` behavior (period)

  ```
  obj.var      means        obj.__getattr__(var)
  obj.var=v    means        obj.__setattr__(var, v)
  ```

Style notes and scope

Classes are important for organizing code. Good style helps.

- Use objects to give structure to your code. This style of programming is called **object oriented programming** (OO). Don't go overboard - not everything needs an object when existing types will do (more to come!).

- Never use the double underscore, except defining the special methods.

- It's standard to use the same-name convention for initialization, e.g.

  ```python
  class Pet:
      def __init__(self, name, species):
          self.name = name
          self.species = species
  ```

  which is allowed since `self.species` is in the class scope,
  and `species` is a local variable in the function.

Shallow and deep copies

We do often need to define what it means to ‘copy’ an object, something like:

```
a = Array([1, 2, 3])
b = a.copy()
```

- A **shallow copy** creates a **new object**, then sets its member variables equal to the old ones (in the a=b sense)

  The original/copy will have references to the same (mutable) data!

```
class Thing:
   __init__(self):
    self.x = 1
    self.arr = [1, 2]
```

```
shallowcopy(self):
    obj = Thing()
    obj.x = self.x
    obj.arr = self.arr
    return obj
```

Shallow and deep copies

We do often need to define what it means to 'copy' an object, something like:

```
a = Array([1, 2, 3])
b = a.copy()
```

- A **deep copy** creates a new object, then copies its member variables by creating true copies (same values, new data)
- In both cases, *immutable* member variables are truly copied

```
class Thing:
   __init__(self):
    self.x = 1
    self.arr = [1, 2]
```

```
deepcopy(self):
    obj = Thing()
    obj.x = self.x
    obj.arr[:] = self.arr[:]  # !
    return obj
```

Example: an array

A **numeric array** class `Vector` can represent a real vector $x \in \mathbb{R}^n$.
It has properties like:

- The vector can be initialized from a list, or with a repeated fixed value
- Elements can be set and accessed by index: `arr[k]` and `arr[k] = v`
- Vectors can be added/subtracted (etc.),
- The vector can be (nicely) displayed

---

constructor and indexing...

```
def __init__(self, r):  # constructor
    if type(r) == int:
        self.arr = [0 for k in range(r)]
    else:
         self.arr = r[:]

def __getitem__(self, k): # arr[k]
    return self.arr[k]

def __setitem__(self, k, val): # arr[k] = v
    self.arr[k] = val
```

```
x = Vector(3, 0)  # x = [0, 0, 0]
y = Vector([4, 5, 6])

x[0] = 1  #calls __setitem__

print(x[0]+y[1]) # calls __getitem__
```

Example: an array

A **numeric array** class `Vector` can represent a real vector $x \in \mathbb{R}^n$.
It has properties like:

- The vector can be initialized from a list, or with a repeated fixed value
- Elements can be set and accessed by index: `arr[k]` and `arr[k] = v`
- Vectors can be added/subtracted (etc.),
- The vector can be (nicely) displayed

---

printing and addition...

```python
def __repr__(self):
    return str(self.arr)

def __add__(self, y):  # z <- x + y
    n = len(self.arr)
    z = Vector(n)
    for k in range(n):
        z[k] = self.arr[k] + y[k]
    return z
```

```python
print(x)  # calls __repr__

x = Vector(3, 0)  # x = [0, 0, 0]
y = Vector([4, 5, 6])

z = x + y
```

Example: an array

mult. of a vector with a scalar on the left:

```
def __repr__(self):
    return str(self.arr)

def __rmul__(self, y):  # y*arr
    n = len(self.arr)
    z = Vector(n)
    for k in range(n):
        z[k] = y*self.arr[k]
    return z

def __mul__(self, y):  # arr*y
    # ...
```

```
x = Vector([1, 2, 3])
y = Vector([4, 5, 6])

z = y*5  # calls y.__mul__
w = 2*x + 3*y  # calls __rmul__

x*y  # also calls x.__mul__(y)
```

What's missing? Many more features could be added...

- Error handling: incompatible input types for +...
- Case work: scalar + vector (`3 + Vector([1,2])`) and more
- More vector functions (dot product, ...)
- A nicer print function
- Much more!