CS 1: Intro to CS

Intro to Classes and Object-Oriented Programming

Q3 of today's Exit Ticket:

- What are some ideas of classes/objects you come up with today?

Administrivia

Very great questions lately! We've really been impressed by the quality of student questions and seeing growth in your debugging skills this week compared to Week 1

We hope you are enjoying Matplotlib! Don't forget to utilize the student Discord more!

MP 5 Part C

Take advantage of this creative component! Above-and-beyond applications are eligible for Engagement Opportunity credit based on:

- Sharing on Discord with peers, including your data science question, motivation, and 2-3 sentences reflecting on what you learned ("engagement with peers/collaboration")
- Originality and application to your own interests ("engagement in applications of CS 1")
- Any reflection component (reflection.txt) you'd like to add to your submission, including room for improvement and possible extension

Agenda

Introduction to classes and object-oriented programming

- The **`class`** statement
- Creating our own objects
- Constructors
- Defining new methods

Wednesday:

- More OOP in Python
- Handling errors with Exceptions
    - Using **`raise`** to raise exceptions on invalid arguments
    - Using **`try`**/**`except`** to handle raised exceptions

You can find a helpful Pre-Check on Canvas to review OOP

Object

Since Week 1, we have been working with various **objects**:

- Strings
- Lists
- Dictionaries
- Tuples
- Files
- Figure and Axes in Matplotlib

All of these objects are built in to Python

What is an Object?

We have used the term “object” pretty informally so far

Concisely, it’s something that:

- Has some kind of internal data (**state**)
- We can call **methods** on

As we discussed a while back, a method is like a function, but it calls on an object (with possible other arguments) using the “dot syntax”

```
object.method(arg1, arg2, ...)
```

Objects and Methods

Recall the **append** method on lists:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(42)
```

Here, `lst` is the **object** (a list)

append is the name of the **method**

42 is the **argument** to the method

Examples We've Seen: Lists and Files

Which of the following lines use **functions**? Which of the following lines use **methods**?

Lists

```
1 lst = [1, 2, 3, 4, 5]
2 length = len(lst)
3 lst.reverse()
4 lst.sort()
5 two_index = lst.index(2)
6 lst_sum = sum(lst)
```

Files

```
1 f = open('aqis.txt', 'r')
2 while True:
3     line = f.readline()
4     print(line)
5     if not line:
6         break
7 f.close()
```

Examples We've Seen: Lists and Files

Which of the following lines use **functions**? Which of the following lines use **methods**?

Lists

```
1 lst = [1, 2, 3, 4, 5]
2 length = len(lst)
3 lst.reverse()
4 lst.sort()
5 two_index = lst.index(2)
6 lst_sum = sum(lst)
```

Files

```
1 f = open('aqis.txt', 'r')
2 while True:
3     line = f.readline()
4     print(line)
5     if not line:
6         break
7 f.close()
```

Methods vs. Functions

Why methods instead of functions?

If **append** wasn't a list method, we could turn it into a function:

```
>>> lst = [1, 2, 3, 4, 5]
>>> append(lst, 42)
```

What's are some issues with this?

Methods vs. Functions

```
>>> lst = [1, 2, 3, 4, 5]
>>> append(lst, 42)
```

Issues with the function approach:

1. If **append** was a function, it would have to change the list argument (usually we don't want to do this in order to make functions easier to manage)
2. We might want to use **append** for different objects that append in different ways, and would then have to use a different function name for each
3. It's also generally more readable to see what the main object is being acted upon (left of "`.`"), and any arguments the method is taking in

Methods vs. Functions

So how do we define methods?

Methods are inextricably tied to objects, so before learning how to define methods, we need to talk about defining our own kinds of objects!

Motivating Objects

Often, we want to create certain kinds of objects to represent things with particular kinds of data

We also want to define new methods for the object to interact with its internal data

What are some types of things you can think of that could represent objects with **state** (data) and **methods** (functionality)?

- Dog(breed) -> different functionality
- CaltechStudent(name, id, year)
- Car(make, color, wheels, can_reverse)
- Food(contains_gluten, vegetarian)

Examples

Some student class ideas from last year's Pre-Check:

- `Course`
- `LOLCharacter`
- `Pet`
- `Dog`
- `Plant`
- `Coffee`
- `FoodItem`
- `Laptop`
- `...`

You can find a list of some proposed state and methods in `oop_student_ideas.py`

Recall: *Implicit vs. Explicit Interfaces*

Matplotlib's documentation refers to "implicit" and "explicit" interfaces; two approaches to create plots using pyplot (Axes.plot vs. plt.plot):

Explicit approach: *"Explicitly create Figures and Axes, and call methods on them (the 'object-oriented (OO) style')."*

```
# we use the ax object methods to plot
fig, ax = plt.subplots()
ax.plot(xs, ys, label='genres')
plt.show()
```

Implicit approach: *"Rely on pyplot to implicitly create/manage the Figures/Axes, and use pyplot functions for plotting."*

```
plt.plot(xs, ys, label='genres')
plt.show()
```

Matplotlib's official documentation recommends "the explicit object-oriented API for complex plots". Today, we'll learn more about what this means!

Getting Started with OOP

OOP is very useful whenever you are working with state for entities and need a way to interact with each one. Graphics programs (with interactive widgets like windows, buttons, plots, etc.) and games (entities with state to keep track of, including characters, moves, buffs, levels, etc.) are some of the most common applications of OOP

For today, we'll just start practicing writing our own objects and exploring the "object-oriented" mindset

First, we need to define what a **class** is...

Classes Define Objects

Objects in Python are are all **instances** of some **class**

A class describes what an object is (like a blueprint), including the definition of all of the methods objects of that class have

A class is also a data **type**

Instances of a class can contain internal data

Examples: lists have elements, a dictionary has key/value pairs, **`Axes`** have x and y axis information, etc.

Defining a Class

```
class <name of class>:
    """<docstring>"""
    def __init__(self, arg1, ...):
        ...

    def <method1>(self, arg1, ...):
        ...

    def <method2>(self, arg1, ...):
        ...
```

PEP8 Note: Whereas PEP8 requires 2 lines between functions, class methods are separated with 1 line

A Very Simple Example

```
class Box:
    """Instances of this class store a single value."""
    def __init__(self, value):
        self.value = value

    def get_value(self):
        return self.value

    def set_value(self, new_value):
        self.value = new_value
```

What do you observe about this code? What do you think **`self`** is used for? **`__init__`**?

What is the internal data?

`class` and `def` Keywords

```
class Box:
    """
    A Box instance represents a Box with a value inside.
    """
    def __init__(self, value):
        self.value = value

    def get_value(self):
        return self.value

    def set_value(self, new_value):
        self.value = new_value
```

The **`class`** statement defines the class followed by the class name and **`:`**

Just like functions, methods are defined with **`def`**, but there are a couple of differences...

self

```
class Box:
    """
    A Box instance represents a Box with a value inside.
    """
    def __init__(self, value):
        self.value = value

    def get_value(self):
        return self.value

    def set_value(self, new_value):
        self.value = new_value
```

The first argument to a Python method represents **the object being acted upon**

By convention, this is called **`self`** (though it doesn't have to be)

self

```
def __init__(self, value):
    self.value = value

def get_value(self):
    return self.value

def set_value(self, new_value):
    self.value = new_value
```

The **`self`** object uses dot syntax to get/add/modify values associated with names

These names are called the object's **attributes**

`__init__` Constructor

```
def __init__(self, value):
    self.value = value
```

Recall that names in Python that are surrounded by double-underscores have a special meaning to Python

- e.g. `__name__` is the current module's name

The `__init__` method is called the **constructor method** (or constructor for short) and is called when a new instance of the class is being created

It is responsible for **init**ializing the object in whatever way is required, assigning all of the attributes (often given through method parameters)

`__init__` Constructor

```
def __init__(self, value):
    self.value = value
```

The **`__init__`** method returns the object that has been constructed (even though it doesn't have a **`return`** statement, it is an implicit return)

It's as if it were written as:

```
def __init__(self, value):
    self.value = value
    return self # this would actually give an error though)
```

More Methods

```
def get_value(self):
    return self.value

def set_value(self, new_value):
    self.value = new_value
```

The class contains two more method definitions that look like regular functions

Both have self as their first argument (meaning “this object”)

`get_value` is an example of a “getter” method, returning internal state

`set_value` is an example of a “setter” method, modifying internal state

Creating Objects

Now that we have our class “blueprint” defined, we can create instances of it!

To create a new object, we use the constructor (the class name) as if it were a function name:

```
>>> b1 = Box(42) # b1.value is set to 42
>>> b2 = Box(-1) # b2.value is set to -1
```

New Style Conventions

To distinguish between functions, methods, objects, etc. some important PEP8 Python conventions need to be followed:

- Every class definition should have a docstring underneath the class header describing the class
- Every method should be separated with 1 line, not 2 lines (**`pycodestyle`** will luckily catch this for you!)
- Classes follow PascalCasing (e.g. **`CaltechStudent`**, not **`caltechStudent`** or **`caltech_student`**)
- Every class should have an **`__init__`** method, ideally a **`__str__`** method, sometimes a **`__len__`** method.

Using Objects

Once we have created objects, we can use them, similar to how we have done with built-in objects (strings, lists, files, Axes, etc.)

```
>>> b1_value = b1.get_value()
42
>>> b1.set_value(0)
b1_value
42
>>> b1_value = b1.get_value()
0
>>> Box.get_value() # error - we must call on instances of Box
```

Moving on From Simple Boxes...

Let's explore how we might define some other, more interesting classes!

Some student ideas:

- `Course`
- `LOLCharacter`
- `Pet`
- `Dog`
- `Plant`
- `Coffee`
- `FoodItem`
- `Laptop`
- …

Check Your Understanding

Can you put into your own words what each of object-oriented terms terms mean?

Class

Object

Attributes

`__init__`

`self`

Object-oriented programming

Next Time

More OOP in Python

- More interactive lectures this week! El will take student ideas and build them into class exercises/live-coding

Classes vs. Client Programs

Handling errors with Exceptions

- Using `raise` to raise exceptions on invalid arguments
- Using `try`/`except` to handle raised exceptions