CS 106AX Programming Methodologies in JavaScript and Python Autumn 2026

Lecture 17: Python Classes and Objects

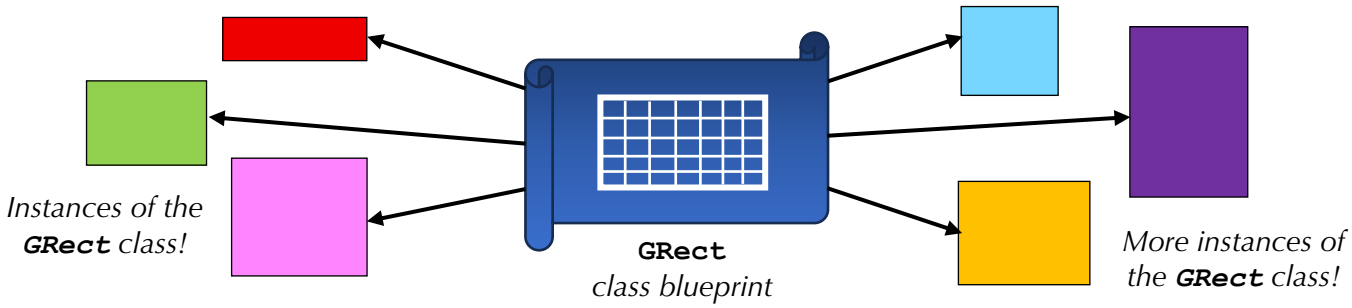
Written by Ben Yan, from materials by Jerry Cain

Classes and Objects

In object-oriented programming, we carefully emphasize the distinction between ***classes*** and ***objects***.

- A ***class*** is a blueprint that defines the structure and behavior of values of a particular type.
- An ***object*** is a value that belongs to a class. A single ***class*** template can be used to create and fashion any number of ***objects***, each of which is an instance of that class.

For instance, the **`JSGraphics`** library—the crown jewel of the JavaScript arc of this course—defines a **`GRect`** class as a blueprint for a rectangle. In your *Breakout!* game, you created many many instances of the **`GRect`** class: one for each colorful brick, and one for the paddle, all of which are separate **objects**.



As a taxonomical metaphor, one can think of ***classes*** as a species name, e.g., *homo sapiens*, and ***objects*** as individual living organisms of that species, e.g., *Jerry Cain*, *Ben Yan*, *Eugene Francisco, Sabrina Yen-Ko*. Alternatively, a class could be *toys*, and objects would include *Buzz Lightyear*, *Sheriff Woody*, etc.

Classes as Templates: Scrooge and Marley

While lecturing on JavaScript aggregates and objects, Jerry Cain conceived a narrative about the tiny bookkeeping company that employs Ebenezer Scrooge and Bob Cratchit from Dicken's *Christmas Carol*.

At the same time, Ben Yan conceived a narrative about the employees of *Ben & Jerrys*—no relation to the ice cream parlor—featuring Jerry Scrooge and his enthusiastic, underpaid assistant Ben Cratchit ☺.



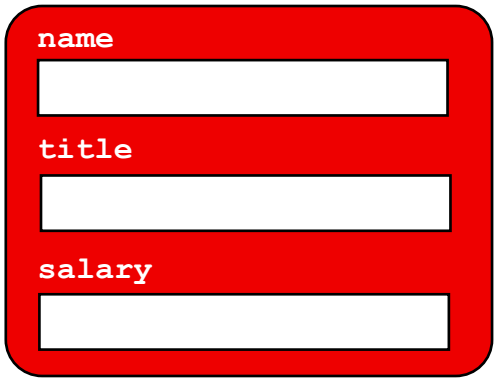
name	title	salary
Jerry Scrooge	Professor	£1000

name	title	salary
Ben Cratchit	Teaching Assistant	£25



While we could use dictionaries to encapsulate this data, we can be even *classier* with, well, **classes** ☺.

The two objects on the preceding page are both instances of the same Python class, which for the moment implies they share the same attributes. **This `Employee` class can be visually defined by the template on the right**—which looks like a name sticker you put on your shirt at a party or awkward social event, as you're forced to answer the usual questions...



- *What's your name?* Ben
- *What's your major?* Idk
- *What's your year?* Lol none of your business
- *What are you from?* Not here
- *What are you doing after graduation?* Alright goodbye

Anyway, class definitions in Python are sufficiently complex that, from a learning perspective, it's better to start with training wheels ☺. That is, **we'll introduce an empty template—that can be used to create blank-slate instances or objects**. We populate those blank-slate objects by filling in the necessary fields.

Defining a Blank-Slate Class

Class definitions in Python start with a header line consisting of the keyword **`class`**, followed by your chosen class name. The body of this class will later glitter with definitions of internal attributes and methods, though for the time being, one can define a blank-slate **`Employee`** class by leaving it empty.

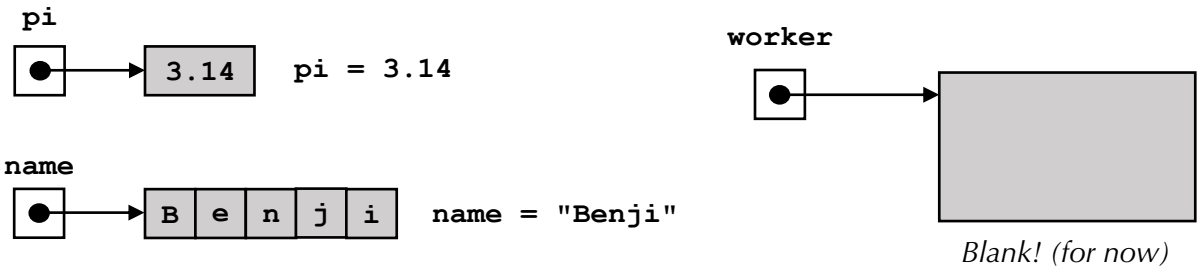
```
class Employee():
    """
    This class defines an empty record type that clients
    can use to store information about employees.
    """
```

Syntactically, a truly blank and empty body is verboten in Python, but you can cheat slightly by using a docstring—as in the example above—or by using the keyword **`pass`**, as a placeholder for future code.

Once you've defined the **`Employee`** class, you can create an empty **`Employee`** object as follows:

```
worker = Employee()
```

You'll want to keep in mind that **objects—like all values in Python—are stored as references**. One can think of the variable as a name tag that's physically separate from the data value as stored in memory.



As such, any code that has access to an object's reference can manipulate the internal contents of the object, such as **getting and setting the contents of existing attributes**, or to **create new attributes inside**.

Creating an Employee by Assignment

The previous statement ominously foreshadowed what we're going to do now ☺, which is to create a blank-slate **`Employee`** object, and manually create and assign each of its attributes. The syntax to do this has form **`objectName.attribute = value`**, much like setting attributes in JavaScript objects.

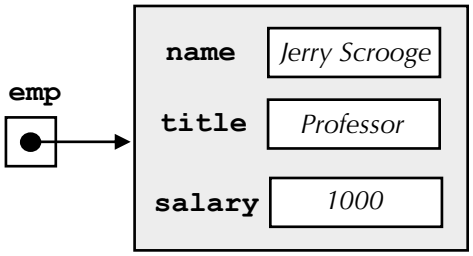
The factory function below accepts three parameters (**`name`**, **`title`**, **`salary`**) as input, and manufactures and returns an **`Employee`** object with those precise attributes and values.



```
def createEmployee(name, title, salary):
      emp = Employee()

      emp.name = name
      emp.title = title
      emp.salary = salary

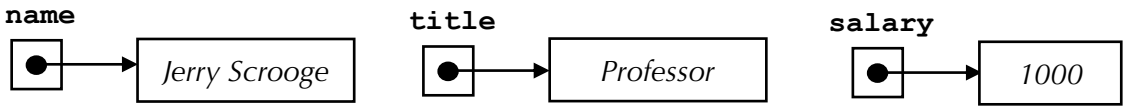
      return emp
```



This is definitely not a commentary on modern capitalism. Anyway, as an example, the function call

```
professor = createEmployee("Jerry Scrooge", "Professor", 1000)
```

first assigns the data values **`"Jerry Scrooge"`**, **`"Professor"`**, and **`1000`** to the local variables **`name`**, **`title`**, and **`salary`**, respectively, which are not yet apart of the blank-slate **`Employee`** object. We then assign each as an attribute of the **`Employee`** object, whose guts are spilled above on the right.



While this strategy does work from a functionality perspective—that is, it does create an **`Employee`** object with the correct contents—making the client responsible for handling the internal data design **violates the spirit of object-oriented programming**, as this should be the implementor's responsibility.

A better strategy for creating a new object from the **`Employee`** class template is to define a method called a **`constructor`**, which is responsible for initializing the internal attributes to the object. Below, we'll discuss how to implement a constructor, as well as other methods hosted in the body of a class!

Defining an `Employee` Class

In Python, **you define a constructor by implementing a special method nested inside the body of the class you're creating**. This method is named **`__init__`**, and is automatically invoked when a client uses the class name as a function, e.g., for the **`Employee`** class, a call like **`Employee(`**…*inputs*…**`)`**.

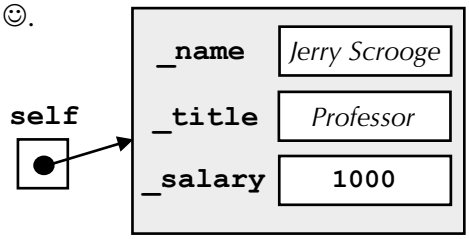
```
class ClassName():
      def __init__(self, other input parameters):
            """ Body of the constructor method, which often initializes the internal attributes """
```

The first parameter is called **`self`**, and contains a reference to the new object itself that we're creating. It's like looking at yourself in a mirror. Any other arguments are passed in as additional parameters.

Here's an implementation of a constructor for our **`Employee`** class ☺.

```
class Employee():

      def __init__(self, name, title, salary):
            self._name = name
            self._title = title
            self._salary = salary
```



Note that to assign values to an object, we mirror the syntax of **`objectName.attribute = value`**, except that we use **`self`** in place of the object's name. Here, **`self`** is a reference to the ***receiver***, which is the **`Employee`** object to which this method is applied—and we use **`self`** because we're initializing attributes ***internally***, from within the **`class`** definition, as opposed to from the outside as we did earlier.

It's customary to precede each attribute name with an underscore, e.g., **`_name`**, **`_title`**, **`_salary`**. That way, someone reading your code understands those attributes should not be accessed from outside.

Alongside the constructor, most classes define an armada of other methods, which can be grouped into:

- ***Getter Methods***: **Retrieve the value of an attribute**, without changing the value as stored inside.
- ***Setter Methods***: **Set an attribute to a new value**, thus changing the internal data representation.

Methods typically declare the parameter **`self`** at the beginning of the parameter list, just as the constructor does, and other parameters are declared after. A method definition thus appears as:

```
class ClassName():
      def methodName(self, other parameters):
            ... body of the method ...
```

The inclusion of **`self`** allows an **`Employee`** object to call this method, e.g., **`emp.methodName(args)`**, and **`self`**, as with the constructor, is initialized to be a reference to this specific **`Employee`** object.

The following code defines **three getter methods**—one for each attribute—and **one setter** for the salary.

```
class Employee():
      """ Assume constructor is defined here! (omitted for conciseness) """

      def getName(self):
            return self._name

      def getTitle(self):
            return self._title

      def getSalary(self):
            return self._salary

      def setSalary(self, newSalary):
            self._salary = newSalary # hopefully a pay raise, but you never know
                                     e.g., my former company is having an uh turbulent week
```



Print Payroll

Because Python lists can contain values of any type, **the elements of a list can be objects**, as constructed from a class. For example, a list of all the employees at ***Ben and Jerry's*** can be initialized as follows.

```
def createStaff():
      roster = [ ]
      roster.append(Employee("Jerry Cain", "Professor", 1000))
      roster.append(Employee("Ben Yan", "Teaching Assistant", 25))
      roster.append(Employee("Eugene Francisco", "Section Leader", 10))
      roster.append(Employee("Sabrina Yen-Ko", "Section Leader", 10))
      return roster
```

The function below then prints the payroll for this employee roster, using our ***getter methods*** to fetch each employee's **`name`**, **`title`**, and **`salary`**. Code such as **`emp._name`**, while legal, is frowned upon.

```
def printPayroll(roster):
      for emp in roster:
            print(emp.getName() + " (" + emp.getTitle() + "): $"
                        + str(emp.getSalary()))

if __name__ == "__main__":
      roster = createStaff()
      printPayroll(roster)
```

Print Payroll

```
Jerry Cain (Professor): $1000
Ben Yan (TA): $25
Eugene Francisco (Section Leader): $10
Sabrina Yen-Ko (Section Leader): $10
```

The pay seems super fair to me… (*for legal reasons, this is a joke, and I actually do get paid very well…*)

Also, note that if you try **`print(emp)`** directly, it will output a long hex value called a **memory address**, e.g., **`<Employee object at 0x123456>`**. This is probably not what you want.



Converting Objects to Strings

If you do want `print(emp)` to work, however, it's good practice to include a `__str__` method in the class you write. This method dictates how the object can be converted to a string, such as when we explicitly cast it to one, e.g., **`str(emp)`**, or when you **`print`**, and comes in handy when debugging ☺.

```
class Employee():
      def __str__(self):
            return emp.getName() + " (" + emp.getTitle() + "): "
                              + str(emp.getSalary())
```

The enables you to simplify the **`printPayroll`** function to this almost insultingly concise form.

```
def printPayroll(roster):
      for emp in roster:
            print(emp)
```

So that you can view all these treasures on one page, here's my full writeup of the **`Employee`** class ☺.

```
class Employee():
      """
      Objects of this class can be used to store information about employees.
      """

      def __init__(self, name, title, salary):
            self._name = name
            self._title = title
            self._salary = salary

      def getName(self):
            return self._name

      def getTitle(self):
            return self._title

      def getSalary(self):
            return self._salary

      def setSalary(self, newSalary):
            self._salary = newSalary

      def __str__(self):
            return emp.getName() + " (" + emp.getTitle() + "): "
                        + str(emp.getSalary())
```