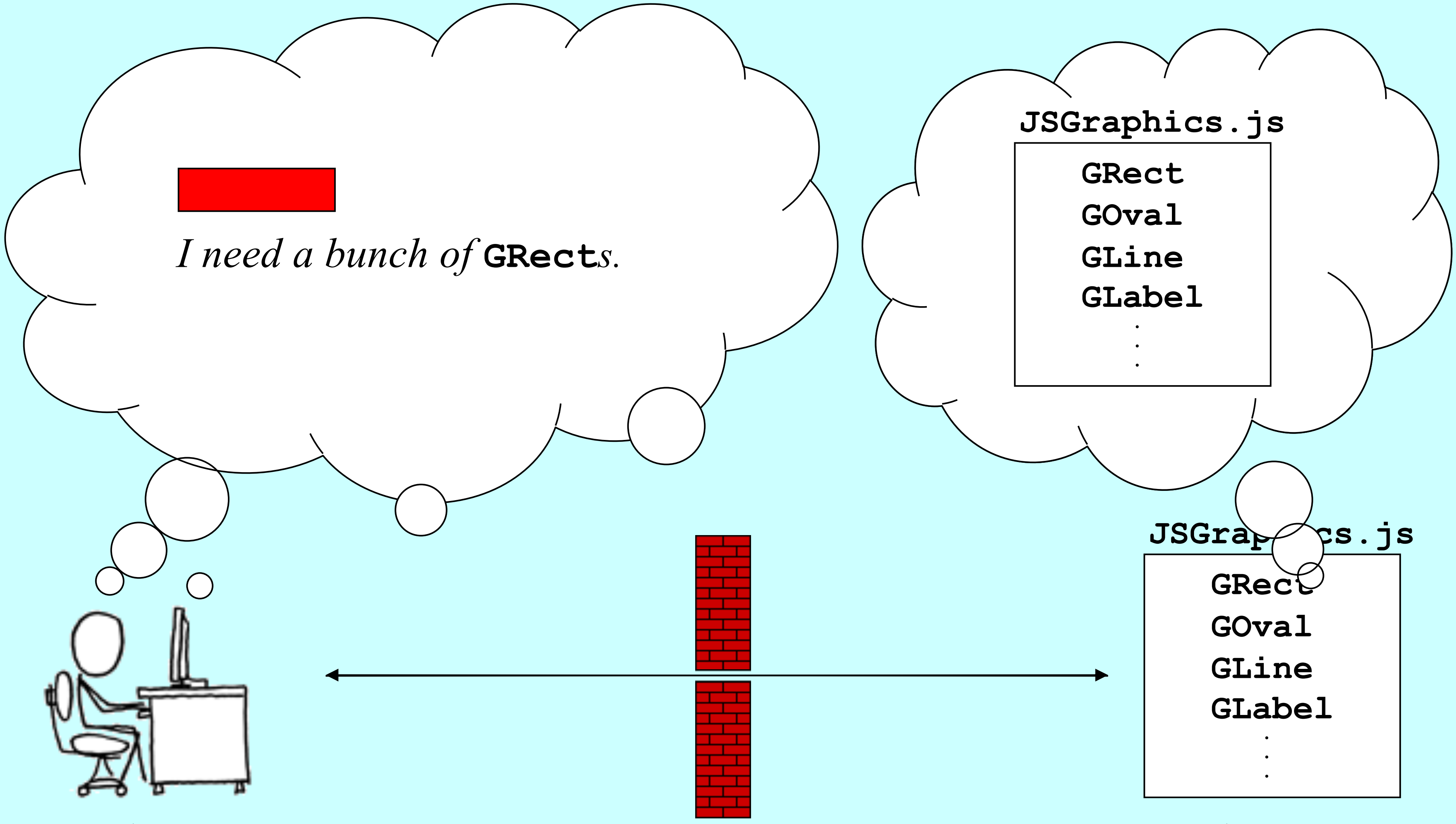
Python and Objects

Jerry Cain
CS 106AX
October 30th, 2024
slides are leveraged from those previously constructed by Eric Roberts

Review: Classes and Objects

- While studying JavaScript, classes, and object orientation, we emphasized the distinction between **classes** and **objects**:
  - A ***class*** is a pattern 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 can be used to create any number of objects, each of which is an ***instance*** of that class.
- The **`JSGraphics`** library, for example, defines a **`GRect`** **class**. In your Breakout assignment, you created many instances of the **`GRect`** **class**—one for each brick and one for the paddle—each of which was a separate **object**.

Thinking About Objects



client *abstraction boundary* *implementation*

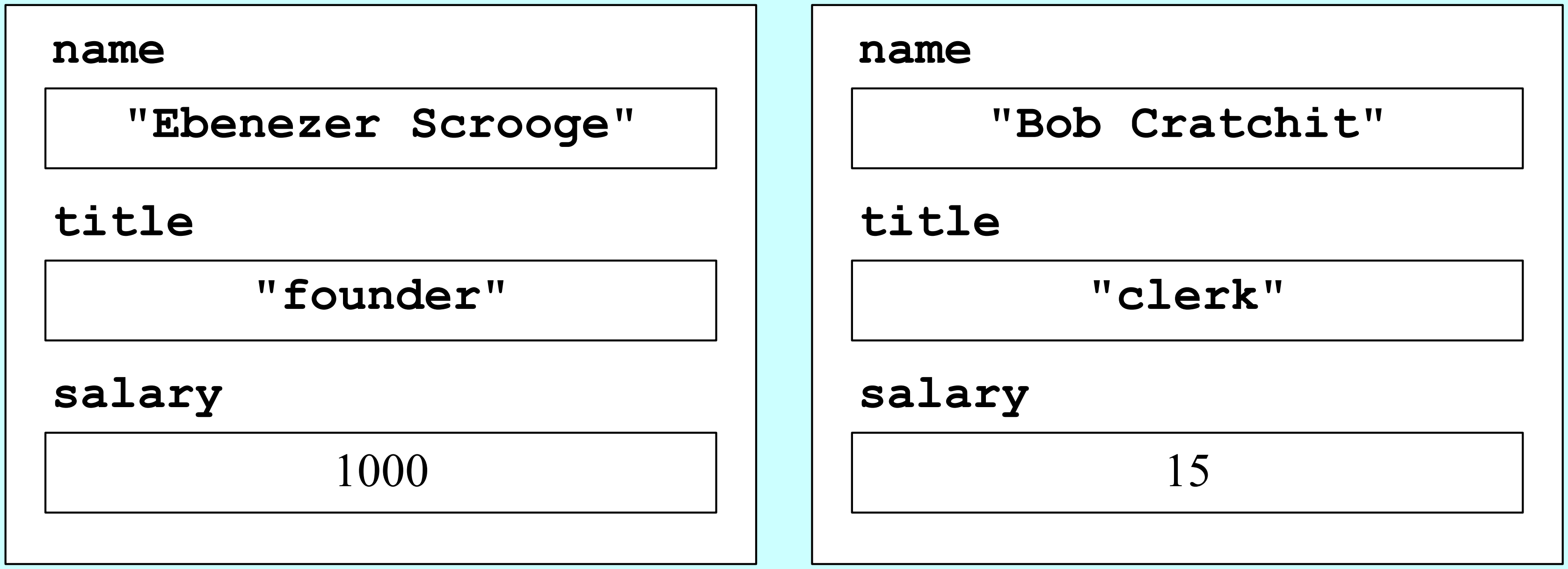
with thanks to Randall Monroe at **xkcd.com**

The Purposes of Objects

- Python uses the concepts of objects and classes to achieve at least three different goals:
  - ***Aggregation***. Objects make it possible to represent collections of independent data values as a single unit. In Python, such collections are traditionally called ***records***.
  - ***Encapsulation***. Classes make it possible to store data values together with the operations that manipulate them. In Python, the data values are called ***attributes***, and the operations are called ***methods***.
  - ***Inheritance***. Class hierarchies make it possible for a class that shares some attributes and methods from a previously defined class to ***inherit*** those definitions without explicitly repeating the definitions. We won't speak about inheritance much more this quarter, but it's a hallmark feature of OOP and studied extensively in later courses.

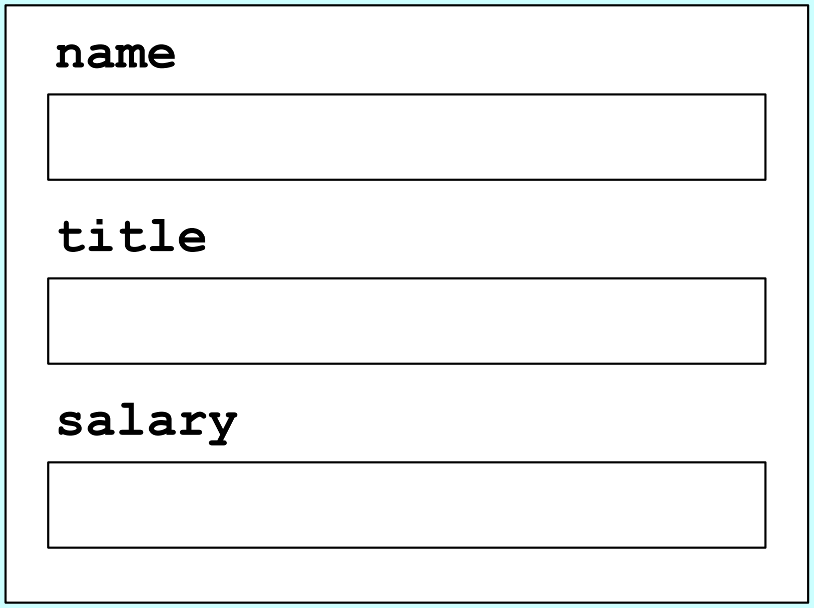
Scrooge and Marley in Python

- While learning about aggregates during the JavaScript segment, we contrived a narrative about a tiny company employing Ebenezer Scrooge and Bob Cratchit. We'll revisit the example and work to promote our aggregates—or records, as they're called in Python—to classes.



Classes as Templates

- The objects on the preceding slide are both instances of the same class, which for the moment means they share the same attributes. The class itself is visually defined by the template:



- Class definitions in Python, however, are sufficiently complex that it helps to start by using an empty template that creates blank-slate objects and then 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 the class name.
- Although the body of a class will later contain definitions of attributes and methods, it is possible to define a blank-slate version of the **`Employee`** class by leaving the body empty:

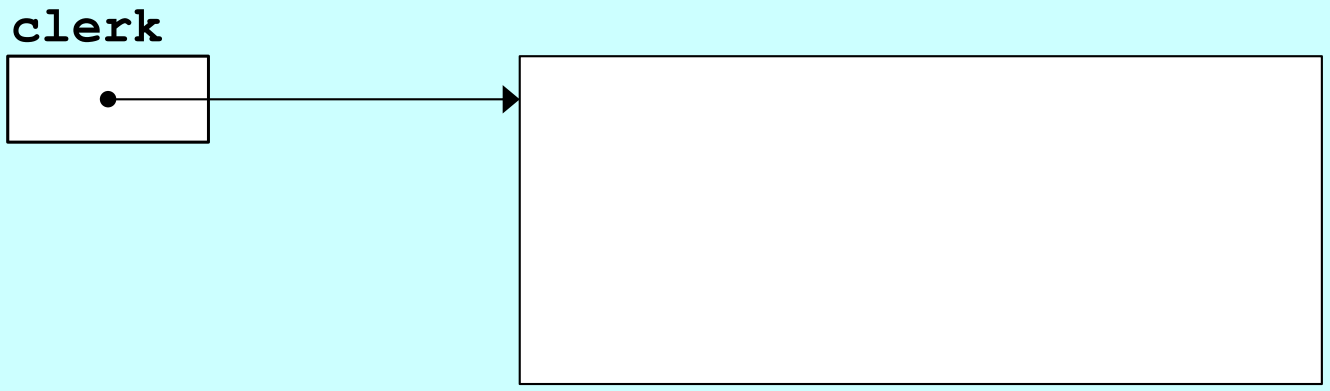
```
class Employee:
   """This class has an empty body"""
```

- Python's syntactic rules do not allow an empty body. You can either use a docstring as in this example or the keyword **`pass`**.
- Once you have defined the **`Employee`** class, you can create an empty **`Employee`** object like this:

```
clerk = Employee()
```

Object Values are References

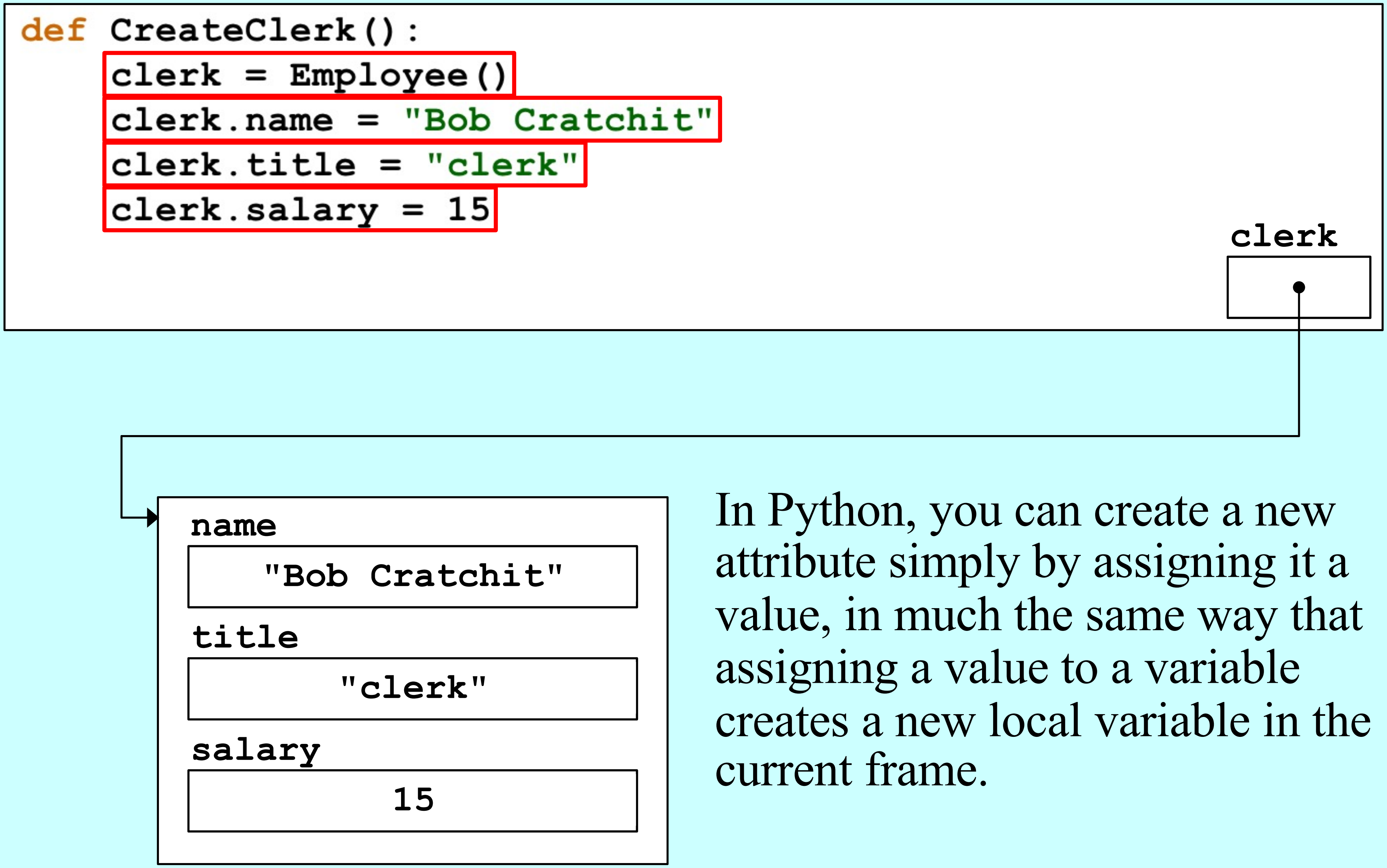
- It is important to keep in mind that objects—like all values in Python—are stored as ***references***. The blank-slate template created by the preceding slide therefore looks like this:



- Any code that has access to this reference can manipulate the contents of the object. In particular, the reference allows code to get and set the contents of existing attributes or to create new ones.

Creating an Employee by Assignment

```
def CreateClerk():
    clerk = Employee()
    clerk.name = "Bob Cratchit"
    clerk.title = "clerk"
    clerk.salary = 15
```

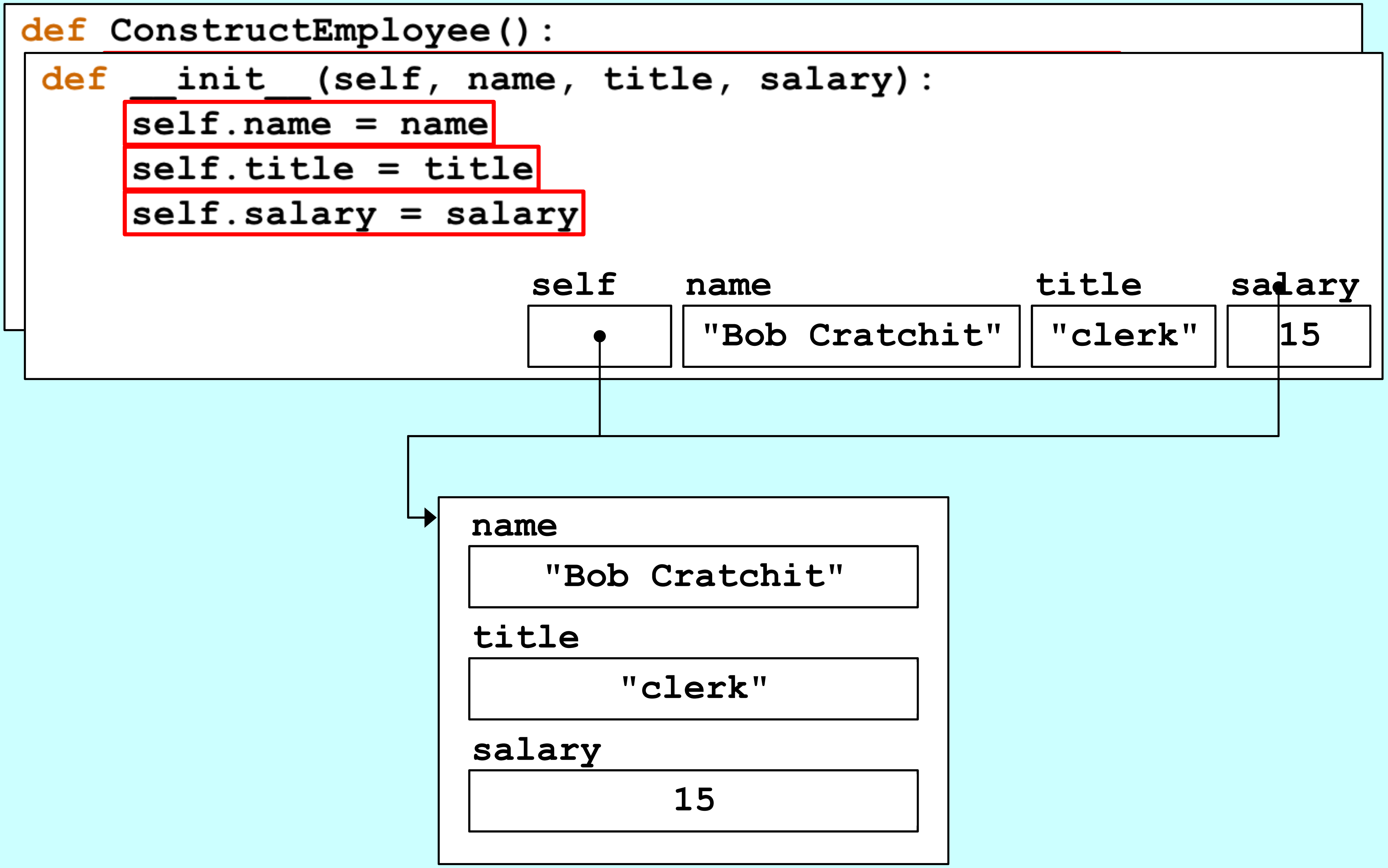


In Python, you can create a new attribute simply by assigning it a value, in much the same way that assigning a value to a variable creates a new local variable in the current frame.

Constructors

- Although the strategy from the preceding slide creates an **`Employee`** object with the correct contents, making the client responsible for creating the attributes violates the spirit of object-oriented programming. The details of the data structure are the implementor's responsibility, not the client's.
- A better strategy for creating a new **`Employee`** object is to define a method called a ***constructor***, which is responsible for initializing the attributes to the object.
- In Python, you define a constructor by implementing a special method called **`__init__`**, which is automatically called when a client uses the class name as a function.
- The first parameter to the **`__init__`** method is called **`self`** and contains a reference to the new object. Any other arguments provided by the client are passed as additional parameters.

A Constructor for the **`Employee`** Class



```
def ConstructEmployee():
```

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

Defining Additional Methods

- In addition to the constructor, most classes define additional methods that allow clients to read or update attributes of the object or to manipulate the object in some way.
- Methods always declare an explicit parameter `self` at the beginning of the parameter list, just as the constructor does. A method definition therefore looks like this:

```
def name(self, other parameters):
    . . . body of the method . . .
```

- Whenever the client calls a method on an object, Python initializes `self` to be a reference to the ***receiver***, which is the object to which the method is applied.

Getters and Setters

- The simplest methods to describe are those that retrieve the value of an attribute, which are called ***getters***, and those that set an attribute to a new value, which are called ***setters***.
- The following definitions show the getter and setter for the salary attribute of an **`Employee`** object:

```
def getSalary(self):
    return self.salary

def setSalary(self, salary):
    self.salary = salary
```

- Getters are much more common than setters. You need to think carefully before providing a setter as to whether you want clients to be able to change the attribute.

Lists of Objects

- Because lists can contain values of any type, the elements of a list can be objects. For example, a list of the employees at Scrooge and Marley can be initialized like this:

```
SCROOGE_AND_MARLEY = [
   Employee("Ebenezer Scrooge", "founder", 1000),
   Employee("Bob Cratchit", "clerk", 15)
]
```

- The following function prints the payroll for the roster of employees supplied as an argument *using our getter methods*:

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

Converting Objects to Strings

- If for no other reason than that doing so simplifies debugging, it is good practice to include a **`__str__`** method in each class to convert an object to a string.
- The **`__str__`** method for the **`Employee`** class looks like this:

```
def __str__(self):
   return self.name + " (" + self.title + "): " +
          str(self.salary)
```

- This definition allows you to simplify the **`printPayroll`** function to this much shorter form:

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

The End