Structured data

- Parallel arrays aren't a natural fit for *heterogeneous* rows of data
  - One set of names, one set of positions, one set of scores
- What we have is structured data
  - Name, position, score for each employee
  - One set of employees
- For a single employee we could do:

```
name = "Tom"
position = "Scapegoat"
review_score = 1
```

  - Allocates memory space for 2 strings and 1 int
    - Keeps a named reference to each location

Using classes

- It would be better to create one object that holds all three pieces of information
  - Could use a list, emp = [‘Tom’, ‘Scapegoat’, 1]
  - But lists are a better fit for *homogenous* data
- Python provides *classes* to group related, heterogeneous data together in a more sensible way

```
emp = employee()
emp.name = “Tom”
emp.position = “Scapegoat”
emp.review_score = 1
```

  - The first statement creates a new employee class *object*
  - The rest use the *member access operator* (.) to work with specific parts (*data members*) in that object

Using classes

- But where did the class employee come from?
  - We need to define it!

```
class employee:
    def __init__(self):
      self.name = “”
      self.position = “”
      self.review_score = 0
```

- This is a *class definition*
  - Acts as a blueprint for creating objects
  - First, we give the class a name
  - Next, we define a function called __init__ inside the class
  - __init__ is called the *constructor*
    - (Don’t worry about the odd name yet)

Using classes

```
class employee:
    def __init__(self):
        self.name = “”
        self.position = “”
        self.review_score = 0
```

- The constructor is a function that is executed whenever you create a new object of this class
  - i.e. when you say emp = employee()
  - Like any function, it can do anything you want
  - But generally, it creates and assigns default values to the data members
- *self* is a reference to the object being created
  - So *self.name* is the variable name inside the new object

More specific details

- Defining the class creates a blueprint
  - No memory is allocated yet
  - The class is a new data type (like integers, strings, etc):
    - To create a new integer:

```
x = 9
```

    - To create a new employee:

```
emp = employee()
```

- This variable declaration:
  - Allocates memory space for an *instance* of the class
    - Contains 2 string variables and 1 integer variable
  - Keeps a named reference (emp) to that memory space
  - A class instance is also called an *object*

More specific details

- With lists, we talked about accessing particular elements in this list
  - Using the subscript operator []
  - E.g. `this_list[15]`
- With class objects, you can access particular data members
  - The *member access operator* (.) indicates part of an object
  - The parts are used like any other variable

```
emp.name = “peter”
emp.position = input(“Position for {}?”.format(emp.name))
emp.review_score = emp.review_score + 1
```

A list of objects

- Now that we've defined a class for employee
  - We use it like any other data type
  - We can have one employee, or a list of employees

```
emps = [employee(), employee(), employee()]
```

  - Creates a list object and three employee objects
    - Each employee object has 2 strings and 1 int in it

- Combine subscript and member access operators
  - The 2nd employee's name:
    - emps[1].name
  - the first employee's review score:
    - emps[0].review_score

Exercise: arrays of objects

- Define a class to hold a point (x, y)
  - Like you would use to specify points on the screen
- Write a statements to:
  - Create a point
  - Set its data members to (1,4)
    - That is x is 1, y is 4
  - Create a list of 100 points
  - Set the second point data members to (5, 3)
  - Print the values of all 100 points to the screen