Classes and Objects
Built-in objects

• We’ve already seen several kinds of objects: strings, lists, sets, tuples, and dictionaries

• An object has two aspects:
  – Some fields (or instance variables) containing data, such as numbers, booleans, or other objects; these describe the state of the object
  – Some methods that provide means of examining or manipulating the object

• In other words, an object bundles together data, and methods for working with that data

• Examples:
  – the list [1, 2, 3] is an object
  – If s = [1, 2, 3], then s is a name for the object, but it isn’t part of the object
  – append is a method you can use with lists: If s is as defined above, s.append(4) changes the named list to have the value [1, 2, 3, 4]

• Objects are sometimes called instances, or instances of (some class)
Classes and objects

- A **class** is a recipe for creating objects
- Classes define the **fields** (or **instance variables**) and methods that each object of the class will have
  - The methods are shared by all objects of that class
  - The fields are **not** shared; every object has its own
- A class is sometimes described as a “blueprint,” or as a “cookie cutter,” since the primary purpose of a class is to describe objects
- Everything in Python is an object
  - Hence, classes are themselves objects – more on this later
Defining a class

- Syntax:

```python
class NameOfClass:
    """Documentation string (optional)"""

Method definitions
- By convention, class names start with a capital letter, and are CamelCase
- Every method definition has the word `self` as its first parameter
  - Exceptions to this rule will be covered later
- There is almost always one special method, called `__init__`, used to construct new objects of this class
  - Known as a constructor in java
- IDLE example: personClass.py
Functions and methods

• **Functions** are independent of objects, and “stand alone”
  – We call functions to get a result
  – Examples:
    – `len(list)` tells the number of things in the list
    – `len(string)` tells the number of characters in the string

• **Methods** are associated with objects
  – “Calling” a method is best thought of as talking to the object
  – The technical jargon is sending a message to the object
  – If `s` is a list, then `s.append(4)` is saying, “s, append 4 to yourself”

• Methods can:
  – Ask the object to tell us something (usually about itself)
  – Tell the object to modify its state in some way
  – Tell the object to give us a modified copy of itself
Parameters and arguments

• You define a method like this:
  
  ```python
  def methodname(self, par1, ..., parN)
  ```

• But you call it like `object.methodname(arg1, ..., argN)`

• How do arguments match up to parameters?
  – They match up by position

• IDLE example: personClass.py
Dot notation

• Dots (periods) are used in two very similar ways:
  1. When you “talk to” an object, you name the object, put a dot, and then the message
     – Example: `my_list.append(another_element)`
     – In this example the “message” contains additional information (`another_element`)
     – The object changes itself (by adding an element)
     – Example: `low_string = my_string.lower()`
     – In this example, no additional information is required
     – The object is unchanged, but returns a new, similar object
  2. When you send a message to a module, you name the module, put a dot, and then the message (the function you want the module to execute)
     ```python
     import copy
     new_list = copy.deepcopy(my_list)
     ```
Initializing an object

• Almost every class you write will have an __init__ method
  – Also known as a constructor in java

• The purpose of the __init__ method is to initialize some instance variables of the object, usually based on the parameters

• Although you define the __init__ method in your class, you don’t call it!

• IDLE example: personClass.py
Creating an object

• To create an object, you use the name of the class, followed by some arguments in parentheses

• Can directly print the object’s fields
  – This is questionable style (more later)

• When we created the object, the __init__ method was automatically called with the given parameters and the new object (self)

• IDLE example: personClass.py
Using an object’s methods

• To use the instance variables or instance methods of an object, you name the object, put a dot, and then the name of the variable or method

• But the object refers to itself by using the name “self”
  – Example: get_older() method
  – What actually happens is that jenny, although listed separately from the other arguments, is an argument, and it gets passed into the self parameter

• IDLE example: personClass.py
Call by value vs. reference revisited

- Everything in Python is an object
  - Even ints and floats are, although they don’t fit the standard notion
  - This is not true in Java, although Java is “more object-oriented”
- Technically speaking, Python has neither “call by value” nor “call by reference”
  - It has “call by object”
  - All function arguments are passed by reference
  - But if you try to modify an argument, then a local variable is created
    - By “modify” I mean change the actual value (e.g., change the int or change the pointer of a collection), not change an element in a list
    - In other words, the function call starts off as “call by reference” and may change to “call by value” if you modify the variable
- **Bottom line**: remembering the earlier distinction of call by value vs. reference is sufficient
- **IDLE example**: callByValue.py
Special Syntax

• For convenience, all the built-in objects have quite a bit of special syntax
  – For example, while you can do things like `my_list.append(an_element)` (usual object syntax), you can also do things like `my_list + my_other_list` (special syntax)
  – `my_list[index]` is yet more special syntax

• There is so much special syntax associated with numbers and booleans that we almost never use the standard object notation
  – Example: `f.is_integer()` returns `True` if the floating point number `f` has an integral value (like 2.0)
Special Methods

• `__init__` is a special function; if you define it, Python will use it
• Another special function is `__str__`, which is used by the `str` and `print` methods to provide a string useful for printing
• Another special function is `__repr__`, whose purpose is to provide a representation of the object that could be used by `eval` to recreate the object
• IDLE example: `personClass.py`
Special Variables

- The documentation string of a function can be retrieved with the `__doc__` special variable.

- A module’s `__name__` is set equal to `['__main__']` when read from standard input, a script, or from an interactive prompt.
  - If this file is being imported from another module, `__name__` will be set to the name of that module.
  - As a result, we have this common idiom:
    ```python
    if __name__ == '__main__':
        Call to the function that starts the program
    ```

- IDLE example: `personClass.py`
A class without `__init__`

- Can initialize class without `__init__`
- Can add instance variables in other methods as well
- Such objects not necessarily useless
  - In python you can add instance variables dynamically anyway
  - Depends on the application
- IDLE example: boring.py
Subclasses

• A new class can **extend** a previously-defined class and add new instance variables and methods
  – Such a class is called a **subclass** of the earlier class
  – To create a subclass, put the name of the **superclass** in parentheses after the name of the subclass

• The subclass **inherits** the variables and methods defined in the superclass

• The **type** of the new object is the subclass type

• IDLE example: employee.py
Creating an instance of a subclass

• A subclass **inherits** the variables and methods of its superclass
• A subclass can (and usually does) **extend** the superclass with additional variables and methods
• To extend a **Person** object, we must first have a **Person** object
  – In a subclass, we can refer to the methods of the superclass with `super()`
  – The first thing to do is to explicitly call `super().__init__`
  – Then we can add instance variables (**role**) or modify existing ones (**age**)
• IDLE example: Employee.py
Overriding

• When we have the same method in a subclass as in a superclass, a subclass instance will use its own version
  – This is called **overriding** a method
  – Instances (objects) of the superclass will continue to use the method defined there
• **Note**: Don’t forget the keyword **super**!
• IDLE example: Employee.py
Classes are objects, too

- Classes can have **attributes**, or **class variables**, and can have class methods
  - These are the same for every object of that class
  - Called static variables/methods in Java (parallel not 100% accurate)
- IDLE example: Employee.py