Classes and Objects

You will learn how to define new types of variables.

Composite Types: Review

- Ones that you should be familiar with now:
  - Strings
  - Lists
  - Tuples
  - Dictionaries
- Lists can be used to track relatively simple information e.g., grades, text-based virtual worlds.
- It is less effective at storing more complex information (e.g., client list) – as you will see.

Composite Types: Review (2)

•Previous example: tracking client information

```
firstClient = ["James Tam"
               "(403)210-9455",
               "tamj@cpsc.ucalgary.ca",
               0]

secondClient = ["Peter Griffin"
                "(708)123-4567",
                "griffinp@familyguy.com",
                100]
```

•If a large number of composite types need to be tracked (e.g., many clients) then you can employ lists of lists.

•(This means that each list element consists of another list).

Example: List Of Lists

•Name of the online example: list_of_lists.py

```
MAX = 4

def initialize (myClients):
    for i in range (0, MAX, 1):
        temp = [(i+1),
                "default name",
                "(111)111-1111",
                "foo@bar.com",
                0]
        myClients.append(temp)
```

Example: Lists Of Lists (2)

```
def display (myClients):
    for i in range (0, MAX, 1):
        print (myClients[i])

# MAIN
def main ():
    myClients = []
    initialize (myClients)
    display(myClients)

main ()
```


Some Drawbacks Of Using A List

- Which field contains what type of information? This isn't immediately clear from looking at the program statements.

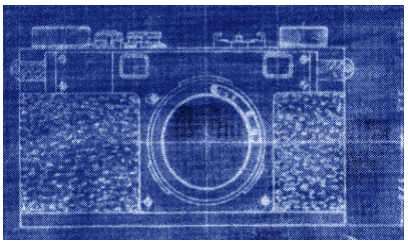
```
temp = [(i+1),
        "default name",
        "(111)111-1111",
        "foo@bar.com",
        0]
```

What is this?

- Is there any way to specify rules about the type of information to be stored in a field e.g., a data entry error could allow alphabetic information (e.g., 1-800-BUY-NOWW) to be entered in the phone number field.

Classes

- Can be used to define a generic template for a new non-homogeneous composite type.
- It can label and define more complex entities than a list.
- This template defines what an instance (example) of this new composite type would consist of but it doesn't create an instance.



Defining A Class

Note the convention: The first letter is capitalized.

- **Format:**

  class <*Name of the class*>:
  name of first field = <*default value*>
  name of second field = <*default value*>

- **Example:**

```
class Client:
    name = "default"
    phone = "(123)456-7890"
    email = "foo@bar.com"
    purchases = 0
```

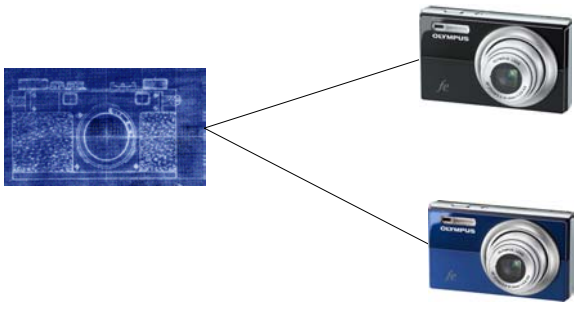
Describes what information that would be tracked by a "Client" but doesn't actually create a client in memory

Creating An Instance Of A Class

- Creating an actual instance (instance = object) is referred to as instantiation.
- **Format:**
  <reference name> = *<name of class>* ()
- **Example:**
  firstClient = Client ()

Defining A Class Vs. Creating An Instance Of That Class

- Defining a class
  - A template that describes that class: how many fields, what type of information will be stored by each field, what default information will be stored in a field.
- Creating a class
  - Instances of that class (during instantiation) which can take on different forms.



Accessing And Changing The Fields

- **Format:**

  <reference name>.<field name> # Accessing value
  <reference name>.<field name> = *<value>* # Changing value

- **Example:**

  aClient.name = "James"

The Client List Example Implemented Using Classes

- Name of the online example: client.py

```
class Client:
   name = "default"
   phone = "(123)456-7890"
   email = "foo@bar.com"
   purchases = 0
```

The Client List Example Implemented Using Classes (2)

```
def main():
  firstClient = Client ()
  firstClient.name = "James Tam"
  firstClient.email = "tamj@cpsc.ucalgary.ca"
  print(firstClient.name)
  print(firstClient.phone)
  print(firstClient.email)
  print(firstClient.purchases)

main()
```


What Is The Benefit Of Defining A Class

- It allows new types of variables to be declared.
- The new type can model information about most any arbitrary entity:
  - Car
  - Movie
  - Your pet
  - A biological entity in a simulation
  - A 'critter' (e.g., monster, computer-controlled player) a video game
  - An 'object' (e.g., sword, ray gun, food, treasure) in a video game
  - Etc.

What Is The Benefit Of Defining A Class (2)

•Unlike creating a composite type by using a list a predetermined number of fields can be specified and those fields can be named.

```
class Client:
    name = "default"
    phone = "(123)456-7890"
    email = "foo@bar.com"
    purchases = 0

firstClient = Client ()
print(firstClient.middleName)
```


What Is The Benefit Of Defining A Class (2)

•Unlike creating a composite type by using a list a predetermined number of fields can be specified and those fields can be named.

```
class Client:
    name = "default"
    phone = "(123)456-7890"
    email = "foo@bar.com"
    purchases = 0

firstClient = Client ()
print(firstClient.middleName)
```

There is no field by this name

Class Methods

- Somewhat similar to the other composite types, classes can have functions associated with them.
  - E.g.,

```
filename = "foo.txt"
name, suffix = filename.split('.')
```

- Unlike these pre-created functions, the ones that you associate with classes can be customized to do anything that a regular function can.
- Functions that are associated with classes are referred to as *methods*.

Defining Class Methods

Format:

```
class <classname>:
    def <method name> (self, <other parameters>):
        <method body>
```

Example:

```
class Person:
    name = "I have no name :("
    def sayName (self):
        print ("My name is...", self.name)
```

Unlike functions, every method of a class must have the 'self' parameter (more on this later)

When the attributes are being accessed inside the methods of a class they MUST be preceeded by the suffix ".self"

Defining Class Methods: Full Example

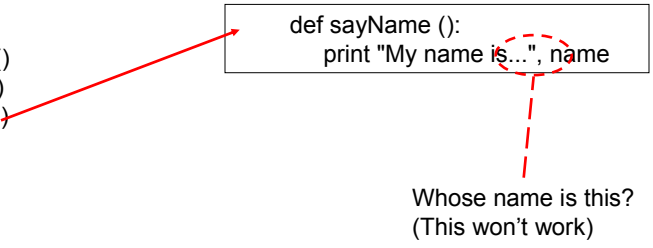
- Name of the online example: person.py

```
class Person:
   name = "I have no name :("
   def sayName (self):
      print ("My name is...", self.name)

def main ():
   aPerson = Person ()
   aPerson.sayName ()
   aPerson.name = "Big Smiley :D"
   aPerson.sayName ()

main ()
```


What Is The 'Self' Parameter

- Reminder: When defining/call methods of a class there is always at least one parameter.
- This parameter is called the 'self' reference which allows an object to access it's attributes inside its methods.
- It's needed to distinguish the attributes of different objects of the same class.
- Example:

```
bart = Person ()
lisa = Person ()
lisa.sayName ()
```



The Self Parameter: A Complete Example

•Name of the online example: person2.py

```
class Person:
   name = "I have no name :("
   def sayName (self):
      print ("My name is...", self.name)

def main ():
   lisa = Person ()
   lisa.name = "Lisa Simpson, pleased to meet you."
   bart = Person ()
   bart.name = "I'm Bart Simpson, who the h*ck are
 you???!!!"
```


Initializing The Attributes Of A Class

•Classes have a special method that can be used to initialize the starting values of a class to some specific values.

•This method is automatically called whenever an object is created.

No spaces here

•**Format**:

```
class <Class name>:
def __init__ (self, <other parameters>):
    <body of the method>
```

•**Example**:

```
class Person:
   name = ""
   def __init__ (self):
      self.name = "No name"
```

Initializing The Attributes Of A Class

- Because the 'init' method is a method it can also be called with parameters which are then used to initialize the attributes.
- Example:

```
- # Attribute is set to a default in the class definition and then the attribute
- # can be set to a non-default value in the init method. (More common
- # approach)
class Person
    name = "Default name"
    def __init___(self, aName):
        self.name = aName
- OR
- # Create the attribute in the init method. (Approach often used in Python).
class Person
    def __init___(self, aName):
        self.name = aName
```


Full Example: Using The "Init" Method

- The name of the online example: init_method1.py

```
class Person:
  name = "I am the nameless bard"

  def __init__ (self, aName):
     self.name = aName

def main ():
  aPerson = Person ("Finder Wyvernspur")
  print (aPerson.name)

main ()
```

Constructor: A Special Method

- Constructor method: a special method that is used when defining a class and it is automatically called when an object of that class has been created.
  - E.g., aPerson = Person () # This calls the constructor
- In Python this method is named 'init'.
- Other languages may require a different name for the syntax but it serves the same purpose (initializing the fields of an objects as it's being created).
- This method should never have a return statement.

Default Parameters

- Similar to other methods, 'init' can be defined so that if parameters aren't passed into them then default values can be assigned.
- Example:

```
def __init__ (self, name = "I have no name"):
```

This method can be called either when a personalized name is given or if the name is left out.

- Method calls (to 'init'), both will work

```
smiley = Person ()
jt = Person ("James")
```

Default Parameters: Full Example

•Name of the online example: init_method2.py

```
class Person:
  name = ""
  def __init__ (self, name = "I have no name"):
     self.name = name

def main ():
  smiley = Person ()
  print ("My name is...", smiley.name)
  jt = Person ("James")
  print ("My name is...", jt.name)

main ()
```


Lists Of References To Objects

•You have already seen examples of composite types which are composed of other composite types.
- E.g., list of strings, each element of the list consists of a string, each string consists of a series of characters. aList = [“james”, “stacey”]

•One important combination of composite types occurs with lists and objects.
- Each element in the list is a reference to an object.
- Example: in the client example (covered earlier in this section) a better implementation would employ a list of clients.

Past approach

```
client1 = Client()
Client2 = Client()
```

Better approach

```
clients = []
for i in range (0, MAX_CLIENTS,1):
  clients[i].append (Client())
```

Example: List Of References To Objects

•Name of the online example: people.py

```
SIZE = 4

class Person:
      name = ""
      age = -1

      def __init__(self,aName,anAge):
         self.name = aName
         self.age = anAge

      def display (self):
            print("My name is...%s" %self.name)
            print("My age is...%d" %self.age)
```


Example: List Of References To Objects (2)

```
def main ():
   people  = []
   for i in range (0,SIZE,1):
      tempName = "Person #" + str(i+1)
      people.append(Person(tempName,i))

   for i in range (0,SIZE,1):
      people[i].display()
      print()

main()
```

Modules: Dividing Up A Large Program

- In Python a module contains a part of a program in a separate file (module name matches the file name).
- In order to access a part of a program that resides in another file you must 'import' it.
- Example:

File: fun.py

```
def fun ():
    print "I'm fun!"
```

File: main.py

```
from fun import *1

def main ():
  fun ()

main ()
```

1 Import syntax:

From <file name> import <function names>

OR

import <file name>

Modules: Complete Example

- Name of the online example: modules1.zip
- Extract both files into the same folder/directory and run the 'main' method (type: "python main.py")

<< **In file main.py** >>

```
from file1 import fun1, fun2
import file2

def main ():
  fun1 ()
  fun2 ()
  file2.fun3()

main ()
```

Note the difference in how fun1 & fun2 vs. fun3 are called

Modules: Complete Example (2)

<< **In module file1.py** >>

```
def fun1 ():
  print ("I'm fun1!")

def fun2 ():
  print ("I'm fun2!")
```

<< In module file2.py >>

```
def fun3 ():
   print("I'm fun3!")
```


Modules And Classes

- Class definitions are frequently contained in their own module.
- A common convention is to have the module (file) name match the name of the class.

Filename: Person.py

```
class Person:
    def fun1 (self):
      print “fun1”

    def fun2 (self):
      print “fun2”
```

Modules And Classes: Complete Example

- The name of the online example: modules2.zip
- Extract both files into the same folder/directory and run the 'main' method which is in the file called "Driver.py" (type: "python Driver.py")

<< **File Driver.py** >>

```
from Greetings import *

def main ():
   aGreeting = Greeting ()
   aGreeting.sayGreeting ()

main ()
```

When importing modules containing class definitions the syntax is:

From <*filename*> import <*classes to be used in this module*>

Modules And Classes: Complete Example (2)

<< **File Greetings.py** >>

```
class Greetings:
   def sayGreeting (self):
      print ("Hello! Hallo! Sup?! Guten tag/morgen/aben! Buenos!
 Wei!  \
               Konichiwa! Shalom! Bonjour! Salaam alikum!
 Kamostaka?")
```

Calling A Classes' Method Inside Another Method Of The Same Class

- Similar to how attributes must be preceded by the keyword 'self' before they can be accessed so must the classes' methods:
- Example:

```
class Bar:
    x = 1
    def fun1(self):
        print (self.x)

    def fun2 (self):
        self.fun1()
```


Complete Example: Accessing Attributes And Methods

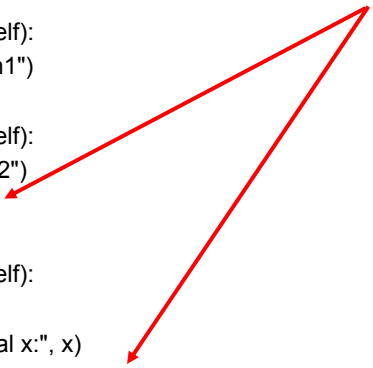
- Name of the online example: modules3.zip
- To run the program extract both files into the same directory and run the "Driver.py" file, at the command line type "python Driver.py"

```
<< Driver.py >>
from Foo import *
def main ():
    aFoo = Foo()
    aFoo.fun2()
    aFoo.fun3()
    print(aFoo.x)


main()
```

- **Access to the methods and attributes of a class outside that classes' methods requires a reference and an object to be created.**
- **This allows access to the attributes and methods using the dot-operator via that reference**

Complete Example: Accessing Attributes And Methods (2)

```
class Foo:
  x = 1

  def fun1 (self):
    print ("fun1")

  def fun2 (self):
    print ("fun2")
    self.fun1()

  def fun3 (self):
    x = 2
    print("Local x:", x)
    print("Attribute x:", self.x)
```



Access to the methods and attributes of a class inside that classes' methods requires the use of the 'self' keyword and the dot-operator

Important Recap: Accessing Attributes And Methods

- Outside of a class the attribute or method MUST be preceded by the name of the reference to the object:
- **Format**:
  <Reference name>.<method or attribute name>
- **Example**:

```
aFoo.fun2()
aFoo.x
```

Important Recap: Accessing Attributes And Methods (2)

- Inside the methods of a class the attribute or method MUST be preceded by the keyword 'self':
- **Format**:
  <self>.<method or attribute name>
- **Example**:

```
self.fun1()
self.x
```


After This Section You Should Now Know

- How to define an arbitrary composite type using a class
- What are the benefits of defining a composite type by using a class definition over using a list
- How to create instances of a class (instantiate)
- How to access and change the attributes (fields) of a class
- How to define methods/call methods of a class
- What is a 'self' parameter and why is it needed
- What is a constructor (__init__ in Python), when it is used and why is it used
- How to write a method with default parameters
- The benefits and the process of creating a list of references to objects
- How to divide your program into different modules