Classes And Objects

Defining new types of variables that can have custom attributes.

James Tam

Composites

- What you have seen
  - Lists
  - Strings
  - Tuples (depends upon semester)
- What if we need to store information about an entity with multiple attributes and those attributes need to be labeled?
  - Example: Client attributes = name, address, phone, email
- The best option you have seen thus far is a list as it's composite (each field is an attribute) and it doesn't have to be homogenous (attributes can store different types of information)

James Tam

Some Drawbacks Of Using A List

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

```
client = [“xxxxxxxxxxxxxxx",
                “0000000000",
                “xxxxxxxxx",
                0]
```

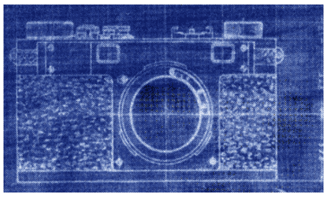
The parts of a composite list can be accessed via [index] but they cannot be labeled (what do these fields store?)

- There isn't a 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.

James Tam

New Term: Class

- Can be used to define a generic template for a new non-homogeneous (elements not always same type) 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.



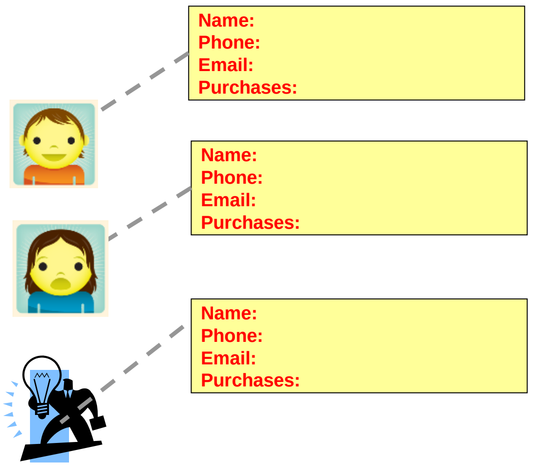
Copyright information unknown

James Tam

New term:
Attribute

Classes Define A Composite Type

- The class definition specifies the type of information (called "**attributes**") that each instance (example) tracks.



James Tam

Defining A Class[1]

Note the convention: The first letter is capitalized.

- **Format:**

```
class <Name of the class>:
      def __init__(self):
           self.name of first field = <default value>
           self.name of second field = <default value>
```

- **Example (attributes are explicitly named):**

```
class Client:
     def __init__(self):
          self.name = "default"
          self.phone = "(123)456-7890
```

Init: Describes what information would be tracked by a "Client" but doesn't yet create a client variable. Analogous to a function definition.

- **Defining a 'client' by using a list (# mapped to a attribute is not self-evident, determined by the index)**

```
client = ["xxxxxxxxxxxxxxx",
          [0]
```

1 It's analogous to defining a function via 'def', the function definition specifies instructions when the function is called. The class definition specifies information to be stored should an instance of the class be declared but doesn't actually create an instance.

James Tam

New terms:
- Instance
- Instantiation
- Object

Creating An Instance Of A Class

- Creating an actual instance (instance = object) is referred to as *instantiation*
  - **Instantiation:** declaring a variable whose type is new type that you defined in the class definition (e.g. creating a new `Client` variable).
- **Object:** it is the variable whose type is the class you defined e.g. `firstClient` is a variable whose type is `Client`.
  - Similar to lists: the creation of an object creates a reference and the actual variable (object)
- **Format:**

```
<reference name> = Name of class>()
```

- **Example:**

```
firstClient = Client()
```

James Tam

Defining A Class Vs. Creating An Instance Of That Class

- **Defining a class** (~List type)
  - 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 (and more…coming later)

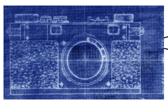


Image copyright unknown

```
Example:
class Client:
    def __init__(self):
         self.name = "default"
         self.phone = "(123)456-7890
```

- **Creating an object** (~creating a new list)
  - Instances of that class (during instantiation) which can take on different forms.

```
Example:
firstClient = Client()
```

James Tam

The Client List Example Implemented Using Classes And Objects

- **Name of the online example**: `1client.py`

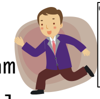
```
class Client:
    def __init__(self):
        self.name = "default"
        self.phone = "(123)456-7890"
        self.email = "foo@bar.com"
        self.purchases = 0
```

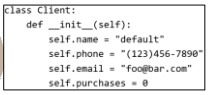
Exactly as-is i.e. no spaces, 2 underscores

James Tam

The Client List Example Implemented Using Classes (2)

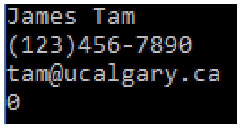
```
def start():
    firstClient = Client()
    firstClient.name = "James Tam"
    firstClient.email = "tam@ucalgary.ca"
    print(firstClient.name)
    print(firstClient.phone)
    print(firstClient.email)
    print(firstClient.purchases)

start()
```





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

```
Changes 2 attributes:
name = "James Tam"
email = "tam@ucalgary.ca"
```



```
James Tam
(123)456-7890
tam@ucalgary.ca
0
```

James Tam

Important Details

- Accessing attributes **inside** the methods of the class
  - MUST preface the attribute with **'self'**

```
class Client:
    def __init__(self):
        self.name = "default"
```

Format (inside eof class):
`self.<attribute name>`

(More on the 'self' keyword later in this section)

- Accessing attributes **outside** the methods in the body of the class (e.g. `start()` function)
  - Must create a **reference** to the object first

```
firstClient = Client()
```

Format (create variable):
`<Ref. name> = <Class name>()`

  - Then **access the object** through that **reference**

```
firstClient.name = "James Tam"
```

Format (access outside of class):
`<Ref. name>.<attribute name>`

```
def start():
    firstClient = Client()
    firstClient.name = "Ja
```

Important Details (2)

- Accessing attributes **inside** the methods of the class.
  - Method MUST have at least 1 parameter: **'self'**

```
class Client:
    def __init__(self):
        self.name = "default"
```

Format (method defined
Must include this parameter

(More on the 'self' keyword later in this section)

- Calling the method outside the body of the class (e.g. `start()` function)
  - **No self reference**

```
firstClient = Client()
```

Format (method call):
Does NOT include the self parameter

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 bacteria or virus in a medical 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
  - A member of a website (e.g., a social network user could have attributes to specify the person's: images, videos, links, comments and other posts associated with the 'profile' object).
  - Etc.

James Tam

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.
  - This provides an **error** prevention mechanism

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

firstClient = Client()
print(firstClient.middleName)  #Error: no such field defined
```

James Tam

New terms:
- `__init__()`
- Constructor

Revisiting A Previous Example: `__init__()`

- Python:
  - `__init__() is used to` *init*ialize the attributes
- Classes have a special function (actually s 'method' – more on this later in this section) called a **constructor** 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 e.g. `bob = Person()`

Automatically calls the `init()` constructor

- **Format**:

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

- **Example**:

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

James Tam

Classes Have Attributes

But Also **Behaviors**

ATTRIBUTES
Name:
Phone:
Email:
Purchases:

BEHAVIORS
Open account
Buy investments
Sell investments
Close account

Image of James courtesy of James Tam

James Tam

New Term: Class Methods ("Behaviors")

- **Functions**: not tied to a composite type or object
  - The call is 'stand alone', just name of function
  - E.g.,
  - `print()`, `input()`
- **Methods**: must be called through an **instance** of a composite[1].
  - E.g.,

List reference

Method operating on the list

```
aList = []
aList.append(0)
```

  - Unlike the above pre-created functions (e.g. append), the methods that you define with your classes can be customized to do anything that a regular function can.
- Functions that are associated with classes (**call through an instance**) are referred to as *methods*.

1 Not all composites have methods e.g., arrays in 'C' are a composite but don't have methods

James Tam

Defining Class Methods

Format:

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

Unlike functions, EVERY python method of a class must have the 'self' parameter (more details later)

Example:

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

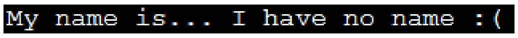
Reminder: When the attributes are accessed inside the methods of a class they MUST be preceded by the suffix ".self"

James Tam

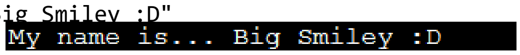
Defining Class Methods: Full Example

- **Name of the online example**: `2personV2.py` (has a method other than just the constructor).

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

def start(): #Access outside class requires a reference
   aPerson = Person()
   aPerson.sayName()
   aPerson.name = "Big Smiley :D"
   aPerson.sayName()

start()
```



```
My name is... I have no name :(
```



```
My name is... Big Smiley :D
```

James Tam

Calling A 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:
    def __init__(self):
         self.x = 0

    def method1(self):
         print(self.x)  #Accessing attribute 'x'

    def method2(self):
         self.method1()  #Calling method 'method1'
```

James Tam

Why Is ‘Self’ Needed

- **Name of the full online example**: `3need_for_self.py`

```
class Person:
   def __init__(self,aName):
       self.name = aName

   def sayFriend(self,myFriend):
       print("Calling object's name %s" %(self.name))
       print("name of friend is %s" %(myFriend.name))

def start():
    stacey = Person("Stacey")
    jamie = Person("Jamie")
    stacey.sayFriend(jamie)

start()
```

James Tam

Whose Method Is Called: Stacey’s Due To Self

Self distinguishes the object whose method is called from other object(s)

```
  def sayFriend(self,myFriend):
      print("Calling object's name %s" %(self.name) ,
        end=",\t")
      print("name of friend is %s" %(myFriend.name))
```

```
Calling object's name is Stacey,        name of Stacey's friend is Jamie
```

```
Calling Stacey's sayFriend() method
```

```
def start():
    print("Calling %s's sayFriend() method" %(stacey.name))
    stacey = Person("Stacey")
    jamie = Person("Jamie")
    stacey.sayFriend(jamie)
```

James Tam

Whose Method Is Called: Jamie's Due To Self

Self distinguishes the object whose method is called from other object(s)

```
def sayFriend(self,myFriend):
     print("Calling object's name %s" %(self.name))
     print("name of friend is %s" %(myFriend.name))
```

```
Calling Jamie's sayFriend() method
Calling object's name is Jamie, name of Jamie's friend is Stacey
```

```
def start():
    stacey = Person("Stacey")
    jamie = Person("Jamie")
    jamie.sayFriend(stacey)
```

James Tam

Self Is Still Needed Even With A Single Object

Error: 'Name' is neither local nor global

- **Reference to the identifier 'name'**
- **Not specified as 'self.name'**
  - **It's not treated as an attribute.**

Check global scope for variable declaration

```
def cannotSay(self):
     print("My name is %s" %(name))

def start():
    stacey.cannotSay()
```

```
  File "F:\work home\217F 2025\www\exa
self.py", line 30, in cannotSay
    print("My name is %s" %(name))
NameError: name 'name' is not defined
```

Check local scope for variable declaration

James Tam

Including Out Of Scope Reference Name Inside Of The Class

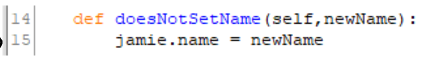
- **Name of the full online example**:
  `4need_for_reference_name.py`
  - **Inappropriately including reference name** in method.

```
class Person:
    def __init__(self,aName):
        self.name = aName
```

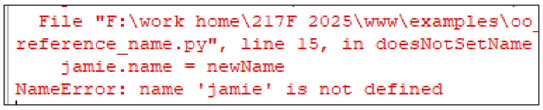
Scope

```
def start():
    stacey = Person("Stacey")
    jamie = Person("Jamie")
    jamie.doesNotSetName
      ("Jamie's new name?")
```

Problem



```
14      def doesNotSetName(self,newName):
15          jamie.name = newName
```



```
  File "F:\work home\217F 2025\www\examples\oo_
reference_name.py", line 15, in doesNotSetName
    jamie.name = newName
NameError: name 'jamie' is not defined
```

James Tam

Excluding The Reference Name

- You wouldn't do **this** now (I hope!)

```
def start():
    aList1 = []
    aList2 = []
    append(321)  #No such 'function'
```

James Tam

Excluding Reference Name Outside Of Class

```
def start():
    stacey = Person("Stacey")
    jamie = Person("Jamie")

    #print("What would the output be? Why?")
    #print(name)
```

```
class Person:
    def __init__(self,aName):
        self.name = aName
```

James Tam

Using 'Self' Outside Of The Class

- **Name of the full online example**: `5mixing_up_self_with_references.py`

```
def start():
    stacey = Person("Stacey")
    jamie = Person("Jamie")

    #self.name = "Jamie's friend"
```

James Tam

Using 'Self' Outside Of The Class

- **Name of the full online example**: `5mixing_up_self_with_references.py`

`Self`: Not declared globally

```
def start():
    stacey = Person("Stacey")
    jamie = Person("Jamie")

    #self.name = "James friend"
```

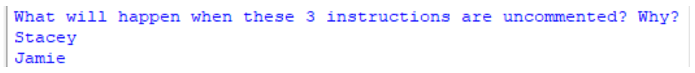
`Self`: Not declared locally

- The identifier '`self`' is not known in this function.
- The same problem if the identifier '`name`' is used without a reference name

James Tam

Previous Example: Follow Up

```
def start():

    #After previous
    name = "James friend"
    print(stacey.name)
    print(jamie.name)
```



```
What will happen when these 3 instructions are uncommented? Why?
Stacey
Jamie
```

James Tam

New Term: Encapsulation

- **Definition 1 for encapsulation**: it's the class definition i.e. the bundling of attributes of methods into the definition encapsulates the **attributes** and methods.

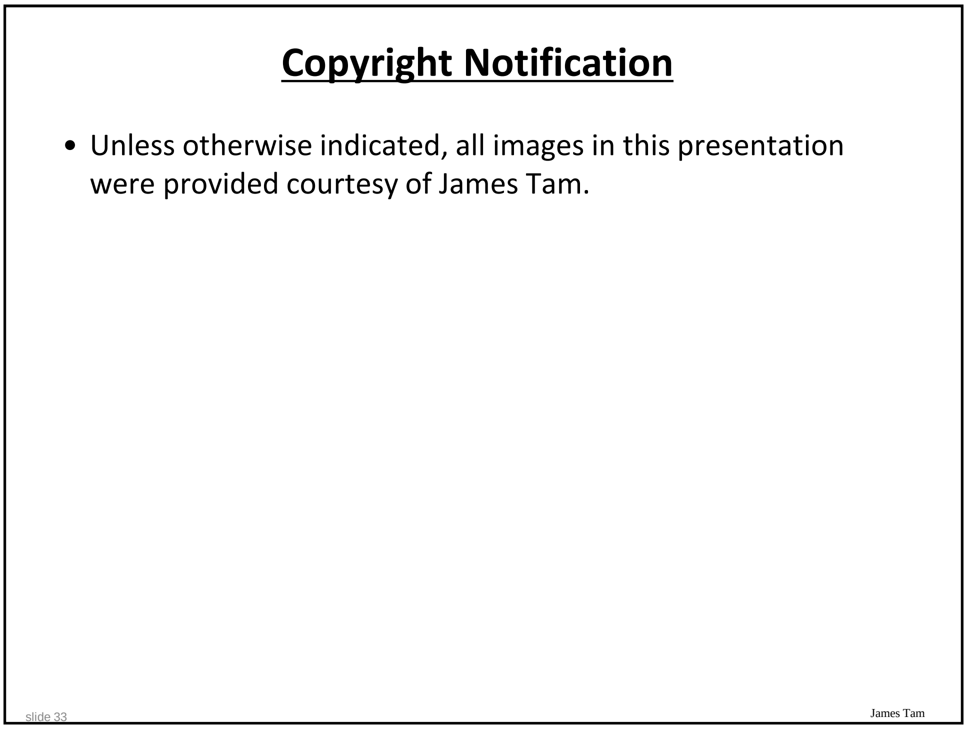
```
class Person:
   def __init__(self,aName):
       self.name = aName
```

James Tam

After This Section You Should Now Know

- How to define an arbitrary composite type using a class.
  - Attributes and methods are bundled with ('encapsulated' into the class definition).
- 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 the '`self`' parameter and why is it needed.
- Why method calls outside of the class must be prefaced by the name of the reference.
- What is a constructor (`__init__` in Python), when it is used and why is it used.

James Tam



Copyright Notification

- Unless otherwise indicated, all images in this presentation were provided courtesy of James Tam.

slide 33

James Tam