

Classes And Objects

- Defining new types of variables that can have custom attributes and capabilities

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.

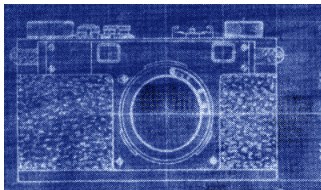
```
client = ["xxxxxxxxxxxxxxxx",
          "0000000000",
          "xxxxxxxx",
          0]
```

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.

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.



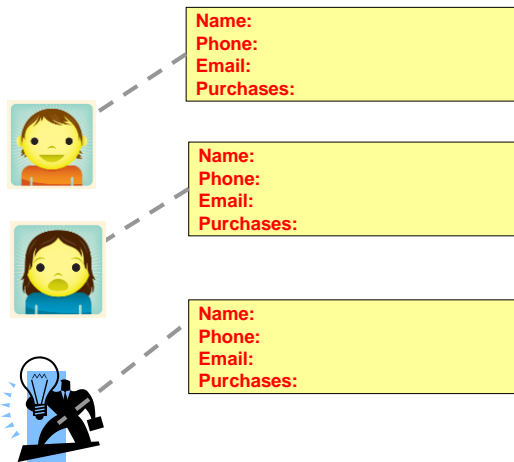
Copyright information unknown

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 Class¹

- Format:**

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

Note the convention: The first letter is capitalized.

- Example (attributes are explicitly named):**

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

Describes what information that would be tracked by a "Client" but doesn't yet create a client variable

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

```
client = ["xxxxxxxxxxxxxxxx",
          [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.

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()`

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

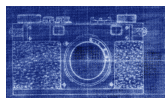


Image copyright unknown



Example:

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

Example:

```
firstClient = Client()
```

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

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)
```



```
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
```

```
start()
```

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:

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()
```

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

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

Format:

<Ref. name> = <Class name>()

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

Format:

<Ref. name>.<attribute name>

James Tam

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).

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
```

New terms:

- `__init__()`
- Constructor


Revisiting A Previous Example: `__init__()`

- Python:
 - `__init__()` is used to initialize the attributes
- Classes have a special function (actually '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()`

• Format:

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

This calls the
`init()`
constructor



• Example:

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

James Tam

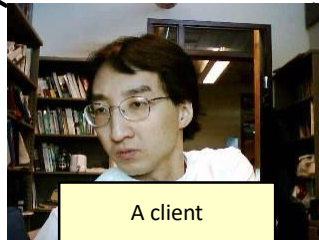
Classes Have **Attributes** But Also **Behaviors**

ATTRIBUTES

Name:
Phone:
Email:
Purchases:

BEHAVIORS

Open account
Buy investments
Sell investments
Close account



A client

Image of James courtesy of 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 composite¹.
 - E.g.,
 - `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

Defining Class Methods

Format:

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

Example:

```
class Person:
    def __init__(self):
        self.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)

Reminder: When the attributes are accessed INSIDEs 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:** 2personV1.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()
```

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))
    print("name of friend is %s" %(myFriend.name))
```

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

```
def start():
    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

Check global scope for variable declaration

Check local scope for variable declaration

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

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

Error: 'Name' is neither local nor global

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

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
```

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

Scope {

Problem

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

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")

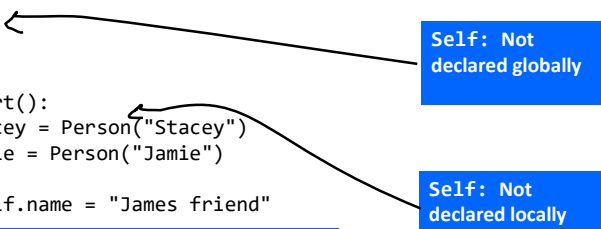
    #self.name = "Jamie's friend"
```

Recall: 'self' must be (and really can only be) used within a class definition.

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 = "James friend"
```

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

James Tam

Decomposing Large Programs: By File

- Because real life programs are large, they are not only divided into functions but also split into multiple files.
- Example: There's so many files for the game RPG Icewind Dale that they are distributed among several folders (multi-file install is not unique to this game).

Cache	2023-07-02 5:57 PM	File folder	
CD2	2023-07-02 6:00 PM	File folder	
Characters	2023-07-02 6:12 PM	File folder	
Data	2023-07-02 6:00 PM	File folder	
Music	2023-07-02 5:59 PM	File folder	
Override	2023-07-02 5:59 PM	File folder	
Portraits	2023-07-02 6:13 PM	File folder	
Scripts	2023-07-02 5:57 PM	File folder	
Sounds	2023-07-09 1:32 AM	File folder	
Temp	2023-07-02 6:08 PM	File folder	
Tempsave	2023-07-02 5:59 PM	File folder	
3dfr.dll	1998-06-02 6:32 AM	Application exten...	689 KB
CHITINKEY	2000-06-22 1:45 AM	KEY File	212 KB
Config.exe	2000-06-27 6:37 PM	Application	713 KB
Dialog.tlk	2000-06-22 11:39 PM	TLK File	2,838 KB
icewind.ini	2023-07-02 6:08 PM	Configuration sett...	1 KB
Icewind_Dale_I_Manual.pdf	2023-07-03 9:20 AM	Adobe Acrobat D...	1,532 KB
IDMain.exe	2000-08-14 1:33 PM	Application	6,140 KB
Keymap.ini	2000-06-22 9:42 PM	Configuration sett...	8 KB
Language.ini	2000-06-22 11:03 PM	Configuration sett...	12 KB
README_ENG.TXT	2000-08-09 1:53 PM	Text Document	44 KB
Uninst.iou	2023-07-02 6:03 PM	ISU File	617 KB

James Tam

Python File Decomposition: Modules

- Each module is a separate library of python features (functions, class definitions).
- Recall: the 'Random' module:

Name of file:
random.py

File contains 1 or
more functions

```

random - Notepad
File Edit Format View Help
def randrange(start, stop=None, step=1):
    """Choose a random item from range(start, stop[, step]).

    This fixes the problem with randint() which includes the
    endpoint; in Python this is usually not what you want.

    ..

    # This code is a bit messy to make it fast for the
    # common case while still doing adequate error checking.
    try:
        istart = _index(start)
    except TypeError:
        istart = int(start)
        if istart != start:
            _warn('randrange() will raise TypeError in the future',

```

James Tam

Review: Using The Code In A Module

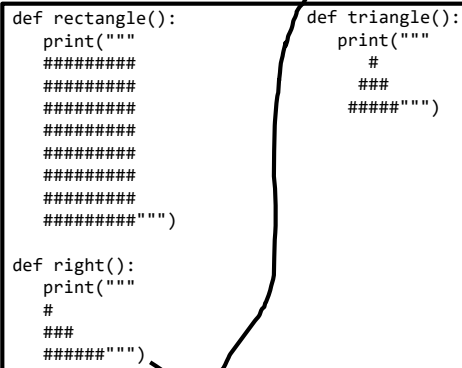
- Add the name 'Random' to your program
 - **Format:**
import <Module/filename>
 - **Example:**
import random
- Running a function/method from this module
 - **Format:**
<Module/filename>.<function/method call>
 - **Example:**
print(random.randrange(1,6))

James Tam

Defining Your Own Module

- Name of the folder containing the full online example:
1st_module_example
- To start the whole program run the module with the 'start' function (in this case it is Driver.py).

Filename: Draw.py



```
def rectangle():
    print("""
    #####
    #####
    #####
    #####
    #####
    #####
    #####""")

def right():
    print("""
    #
    ##
    #####""")

def triangle():
    print("""
    #
    ##
    #####""")
```

James Tam

Defining Your Own Module (2)

– Filename: **Driver.py**

```
import Draw

def start():
    Draw.rectangle()
    Draw.right()
    Draw.triangle()

start()
```

Adding the name
'Draw' to your
program

Running functions from
the 'Draw' module
(similar to running the
random methods this
requires <module
name>.<method name>

Filename: **Draw.py**

```
def rectangle():
    print("""
    #####
    #####
    #####
    #####
    #####
    #####""")

def triangle():
    print("""
    #
    ###
    #####""")

def right():
    print("""
    #
    #
    #####""")
```

Naming The **Starting Module**

- Recall: The function that starts a program (first one called) should have a good self-explanatory name e.g., "start()" or follow common convention e.g., "main()"
- Similarly the **file module that contains the 'start()' or 'main()' function** should be given an appropriate name e.g., "Driver.py", "Start.py", "Main.py" (it's the 'driver' of the program or the starting point)

Filename: **"Driver.py"**

```
def start():
    #Instructions

start()
```

James Tam

Importing Modules Containing Class Definitions

- A common convention is to have the module (file) name match the name of the class e.g. file/module 'Person.py' contains definition for "class Person"
- **Approach 1:** import just the name of the file containing the class definition.
 - Example: `import PersonFile` (similar this previous approach: `import Random`)
 - Advantage:
 - Allows access to **other 'names'** in the file e.g. other utility methods, constants etc.
 - Recall Random.py contains: `Random.randrange(0,6)`, `Random.randint(1,6)`
 - Disadvantage:
 - The **name of the file** must be included along with the name of the **function/method/attribute** e.g. `Random.randint(1,6)`

File: Random.py

```
def randint(start,end):
def randrange(start,end):
```

James Tam

Importing Modules Containing Class Definitions

- **Approach 2:** import just the name of the file containing the class definition.
 - Example: `from PersonFile import Person`
(or using this approach with an existing library `from Random import randint`)
 - Advantage:
 - Imports only the names needed (reduced conflicts between the names in the module being imported and the file where the import is included).
 - `from Random import randint`
 - `#only imports the randint name`

James Tam

Approach 1: An Example

- **Name of the folder complete online example:**

2nd_oo_module_example

- Only the **module/filename** is imported so **other names** (e.g. **class**, **method**, **function name**) must be prefaced by the module name ('context' needed).

Filename:

PersonFile.py

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

def fun():
    print("called fun")
```

Filename:

Driver.py

```
import PersonFile

def start():
    #Only filename imported
    aPerson = PersonFile.Person()
    aPerson.sayName()
    PersonFile.fun()

start()
```

James Tam

Approach 2: An Example

- **Name of the folder complete online example:**

3rd_oo_module_example

- More specific: only imports the **class name**, other names (e.g. function, global names) cannot be accessed.
 - The **class name** doesn't need to be prefaced by the **module name**.
 - References to the imported name are shorter.
 - Reduces the possibility of naming conflicts e.g. if there's already a function 'fun' in Driver.py (won't conflict with 'fun' in Person).

Filename:

PersonFile.py

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

def fun():
    print("called fun")
```

Filename:

Driver.py

```
from PersonFile import Person

def start():
    aPerson = Person()
    aPerson.sayName()

start()
```

h

Recall: Objected Approach **Ties Behaviors (Methods) To** Classes

- Capabilities are defined in class `Flyer` mean that all objects whose type is a `Flyer` have thee abilities of **abilities of a flyer**.

```
class Flyer():
    def fly(self):
        ...

class Flock:
    def __init__(self):
        self.aFlock = []
        for i in range(0,12,1):
            aFlyer = Flyer() #Each element can 'fly'
            self.aFlock.append(aFlyer)

    def takeFirstFlight(self):
        for i in range(0,12,1):
            self.aFlock[i].fly() #Flyer types 'flying'
```

James Tam

An Addition Object-Oriented Concept: **Inheritance**

- Capabilities are defined in a class (in this case it's the `Flyer()` which is a **parent class**) and all classes that inherit all the **abilities of a flyer** (in this cases the child class: `Airplane`).

```
class Flyer():
    def fly(self):
        ...
```

- Via inheritance:** class definitions be extended by specifying that '**child**' classes (derived from the **parent**) **inherit** (are able to access) the attributes and methods of the parent.

```
class Airplane(Flyer):
```

In python this allows an Airplane object to 'fly' by inheriting the abilities of the 'Flyer'.

Alternative example: Java

```
public class Airplane extends
Flyer
{
}
```

James Tam

Inheritance: A Complete Example

- Name of the folder complete online example:
4th_inheritance_example

```
#Flyer.py
class Flyer():
    def fly(self):
        print("flying")

#Airplane.py
# 'Flyer' not defined here
from Flyer import Flyer

class Airplane(Flyer):
    def refuel(self):
        print("Fueling up!")

#Person.py
class Person():
    def doPersonStuff(self):
        print
        ("Doing people things")

#Driver.py
from Flyer import Flyer
from Person import Person
from Airplane import Airplane
def start():
    aFlyer = Flyer()
    aFlyer.fly()
    aPlane = Airplane()
    aPlane.refuel()
    aPlane.fly()
    aPerson = Person()
    aPerson.doPersonStuff()
    #aPerson.fly() #Error
```

James Tam

Object-Oriented Design: Advantage Over Procedural Decomposition

- Procedural approach: functions can allow for nonsensical behaviors e.g. "flying pigs"
- E.g.

```
def fly():
    ...

pigs = list["pig1", "pig2"]
fly(pigs)
```

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
- What is a constructor (`__init__` in Python), when it is used and why is it used
- How to divide your program into different modules
- How inheritance can allow access to group of derived classes.

James Tam

Copyright Notification

- "Unless otherwise indicated, all images in this presentation are used with permission from Microsoft."

James Tam