# Classes And Objects

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

## Composites

- What you have seen
  - Lists
  - Strings
  - Tuples
- 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

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.

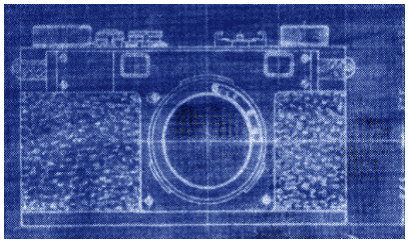
```
client = [“xxxxxxxxxxxxxxx",
          “0000000000",
          “xxxxxxxxx",
          0]
```

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

- 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.

## **New Term**: Class

- 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.



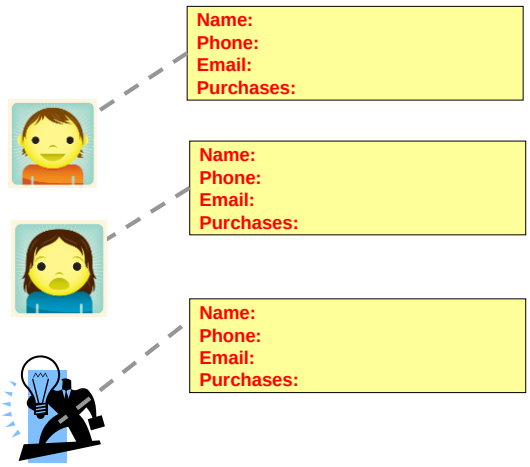
Copyright information unknown

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
```

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 = ["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

**New terms**:
- Instance
- 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.

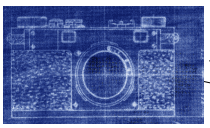


Image copyright unknown

```
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()
```

# The Client List Example Implemented Using Classes And Objects

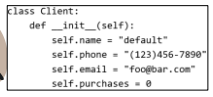
- **Name of the online example**: `1client.py`

```
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)

start()
```




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

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

# Important Details

- Accessing attributes **inside** the methods of the class.
  - Preface the attribute with 'self'

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

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)
  - Need to create a **reference** to the object first

```
firstClient = Client()
```

<Ref name> = <Class name>()

  - Then access the object through that reference

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

<Ref name>.<attribute name>

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

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__()`

- `__init__()` is used to initializing 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 composite[1].
  - E.g.,

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

List reference

Method operating on the list

- Unlike these pre-created functions, the ones that you define with your classes can be customized to do anything that a regular function can.
- Functions that are associated with classes 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 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**: 2personV1.py

```
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 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**:

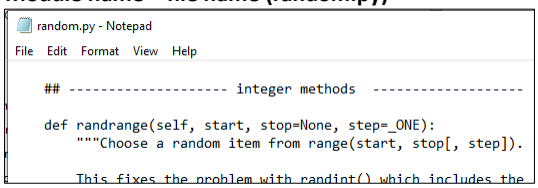
```
class Bar:
    def __init__(self):
         self.x = 0

    def fun1(self):
         print(self.x)  #Accessing attribute 'x'

    def fun2(self):
        self.fun1()     #Calling method 'fun1'
```

James Tam

## Subdividing A Program Into Parts

- In addition to splitting a program by functions a large program may be split into different files.
- **New definition: Module** = a file that contains a part of a program.

**Module name = file name (random.py)**




- Because the code resides in another file the name of the module must be imported:

```
import random
print(random.randrange(0,6))
```

James Tam

## Modules: Dividing Up A Large Program

**Import syntax:**

- # Import some functions
- From <file/mod name> import <fun1>, <fun2>...

- # Import all functions
- From <file/mod name> import *

- # Import only module/file (not functions)
- import <file/mod name>

Mod = short for module

## Different Types Of Imports: Pre-Created Module

- **Name of the folder containing the different versions of the program**: 3imports_of_random

```
import random

#Author:  James Tam
#Version 3A: June 10, 2024
# *Only import the module/filename

#Must specify the module name and the method name
print(random.randrange(0,100)+1) #Displays 1-100
print(random.randrange(1,4)+1) #Displays 1-4
```

```
from random import *

#Author:  James Tam
#Version 3B: June 10, 2024
# *Import all identifiers (e.g. function, constant names)

#The names 'randrange' and 'randint' are 'known' via the import so only the
#the method name must be specified.
print(randrange(0,100)+1) #Displays 1-100
print(randrange(1,4)+1) #Displays 1-4
```

```
from random import getrandbits #Separate multiple identi

#Author:  James Tam
#Version 3C: June 10, 2024
# *Import only the identifier(s) specified

#The names 'randrange' and 'randint' are 'known' via the
#the method name must be specified.
print(getrandbits(1)) #From python.org "Returns a non-ne

""" Won't work: methods and module haven't been imported
print(randrange(1,4)+1) #Displays 1-4
print(randrange(0,100)+1) #Displays 1-100
print(random.randrange(0,100)+1) #Displays 1-100
"""
```

# Different Types Of Imports: Code You Write

- **Name of the folder containing the different versions of the program**: `4multiple_modules_procedural`

- **Module_A**
  - Def fun1
  - Def fun2

- **Module_B**
  - Def fun3
  - Def fun4
  - Def fun5

- **Module_C**
  - Def fun1
  - Def fun6

**Module_D**
```
Imports Module_A
Def fun1()
Calls Module_A.fun1()a
```

**Driver**
```
Import Module_A
From Module_B import *
From Module_C import fun1, fun6
Import Module_D
```

This example services to illustrate:
1) How to decompose a procedural program into modules
2) Why you should only import needed identifiers (minimize the use of from <module> import *


- Module_A
  - Def fun1
  - Def fun2
- Module_B
  - Def fun3
  - Def fun4
  - Def fun5
- Module_C
  - Def fun1
  - Def fun6


# Different Imports: Function Calls

**Driver**
```
import Module_A
from Module_B import *
from Module_C import fun1, fun6
import Module_D
    Module_A.fun1()
    Module_A.fun2()
    fun3()  #Module_B
    fun4()  #Module_B
    fun5()  #Module_B
    fun1()  #Module_C
    fun6()  #Module_C
    Module_D.fun1()
```

**Module_D**
```
import Module_A

def fun1():
    print("Module_D.fun1()")
    Module_A.fun1()
```

```
Module_A.fun1()
Module_A.fun2()
Module_B.fun3()
Module_B.fun4()
Module_B.fun5()
Module_C.fun1()
Module_C.fun6()
Module_D.fun1()
Module_A.fun1()
```

James Tam

## 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")
```

- To use the code of class Person from another file module you must include an import:

```
from <filename> import <class name>
from PersonFile/PersonModule import Person
```

James Tam

## Modules And Classes: Complete Example

- **Name of the folder containing all the modules for this example**: `5multiple_modules_OO_simple`
  - To start the whole program run the module with the '`start`' function.

```
<< File = Driver.py >>
from Greetings import *

def start():
   aGreeting = Greetings()
   aGreeting.sayGreeting()

start()
```

When importing modules containing class definitions the syntax is (star '*' imports everything):

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

James Tam

## 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?")
```

James Tam

## 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`" (it's the 'driver' of the program or the starting point)

**Filename: "Driver.py"**

```
def start():
    #Instructions

start()
```

James Tam

## Complete Example: **Accessing Attributes** And **Methods**: Person Module

- **Name of the folder containing all the modules for this example**: 6multiple_modules_OO_intermediate
  - To start the whole program run the module with the 'start' function.

```
<< File: Person.py >>
class Person:
    def __init__(self,newName,newAge):
        self.name = newName
        self.age = newAge
```

This is a good summary of:
- Defining a class with attributes and methods (including __init__())
- Creating an object
- Accessing attributes inside of a class
- Accessing methods inside of a class
- Accessing attributes & methods outside of class
- Splitting a program into modules

## Complete Example: Accessing **Attributes** And **Methods**: Person Module (2)

```
    def haveBirthday(self):
        print("Happy Birthday!")
        self.mature()

    def mature(self):
        self.age = self.age + 1
```

James Tam

## Complete Example: Accessing **Attributes** And **Methods**: The "Driver" Module

```
<< File: Driver.py >>
from Person import Person

def start():
    aPerson = Person("Cartman",8)
    print("%s is %d." %(aPerson.name,aPerson.age))
    aPerson.haveBirthday()
    print("%s is %d." %(aPerson.name,aPerson.age))

start()
```

```
def __init__(self,newName,newAge):
    self.name = newName
    self.age = newAge
```

```
Cartman is 8.
Happy Birthday!
Cartman is 9.
```

```
def haveBirthday(self)
    print("Happy Birthday!")
    self.mature()
```

```
def mature(self):
    self.age = self.age + 1
```



## 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)
```

## Recall: Objected Approach Ties Behaviors (Functions/Methods) To Classes

- Definition of a class (in this example it's the parent whose methods are available to classes that are derived from this class)

```
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'

**Alternative example: Java**

```
public class Airplane extends
Flyer
{

}
```

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