Extra Topics From CPSC 231: O-O & Recursion

- Section I: Defining new types of variables that can have custom attributes and capabilities
- Section II: You will learn the definition of recursion as well as seeing how simple recursive programs work

James Tam

Section I: Introduction To Object-Oriented Programming

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.

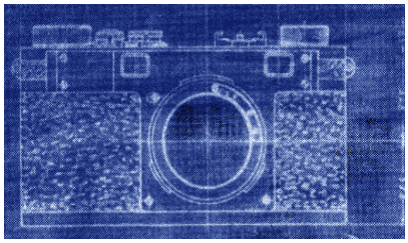
```
client = [“xxxxxxxxxxxxxxx",
                “0000000000",
                “xxxxxxxxx",
                0]
```

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.



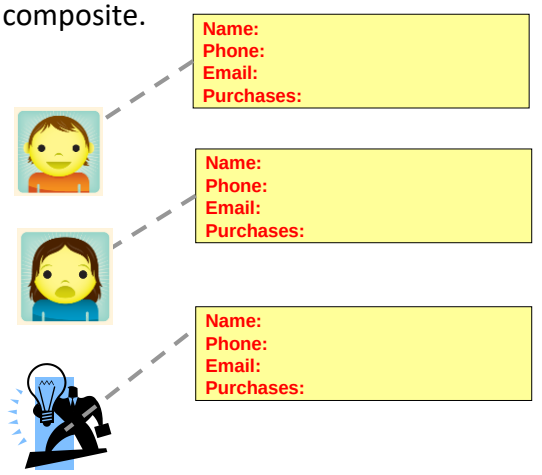
Copyright information unknown

James Tam

New term: Attribute

Classes Define A Composite Type

- The class definition specifies the type of information (called "**attributes**") tracked by each instance (example) of a composite.



Each of these fields could be analogous to a list element but they are accessed via the name

James Tam

Declaring A New Variable Type: 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 clearer):**

```
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 (index # mapped to a attribute is not self evident)**

```
client = ["xxxxxxxxxxxxxx",
          "0000000000",
          "xxxxxxxxx",
           0]
```

1 Although capitalization of the class name isn't the Python standard it is the standard with many other programming languages: Java, C++

New terms:
- Instance
- Object

Creating A Variable Of The New Type: 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 (~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.

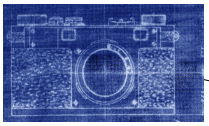


Image copyright unknown

```
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()
```

Accessing And Changing The Attributes - Outside Class Methods E.g. Inside `Start()`

- **Format:**

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

- **Example:**

```
def start():
    aClient.name = "James"
    print(aClient.name)
```

The Client List Example Implemented Using Classes And Objects

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

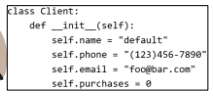
```
class Client:
    def __init__(self):
        self.name = "default"
        self.phone = "(123)456-7890"
        self.email = "foo@bar.com"
        self.purchases = 0
```

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

```
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>`

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
```

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

ATTRIBUTES
Name:
Phone:
Email:
Purchases:

BEHAVIORS
Open account
Buy investments
Sell investments
Close account

Image of James curtesy 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 associate with classes can be customized to do anything that a regular function can (because you implement them yourself).
- 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

New term: class method

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

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

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

Simple Python Example Implementing Inheritance

- **Name of the online example:** `3inheritance`
  - Derived child class access parent's attributes/methods

```
class Parent():
    def __init__(self):
        self.a = 1
        self.b = 2
    def display(self):
        print(self.a,self.b)

class Child(Parent): #Can access Parent's attributes/methods
    def __init__(self):
        super().__init__()
        super().display()
        self.c = "Attribute is unique to child"
    def displayUnique(self):
        print(self.c)
```

James Tam

Simple Python Example Implementing Inheritance (2)

```
def start():
    print("Parent")
    aParent = Parent()
    aParent.display()
    print(aParent.a,aParent.b)

    print("\nChild")
    aChild = Child()
    aChild.display()
    print(aChild.a,aChild.b,aChild.c)

    #Error: parent has no such attribute print(aParent.c)
    #Error: parent has no such method aParent.displayUnique()

start()
```

```
class Parent():
    def __init__(self):
        self.a = 1
        self.b = 2
    def display(self):
        print(self.a,self.b)
```

```
class Child(Parent): #Can access Parent's attri
    def __init__(self):
        super().__init__()
        super().display()
        self.c = "Attribute is unique to child"
    def displayUnique(self):
        print(self.c)
```

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
- How inheritance can allow access to group of derived classes.
  - The attributes and methods defined in the parent class can be accessed in the child class/classes.

James Tam

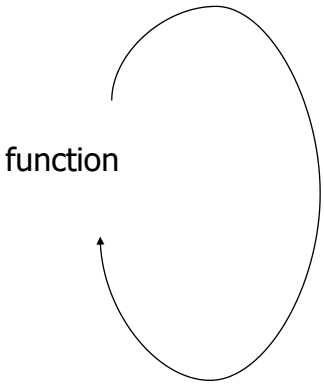
Section II: Introduction To Recursion

Basic Definition Of Recursion

- *"A programming technique whereby a function calls itself either directly or indirectly."*

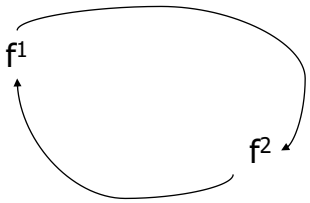
James Tam

Direct Call

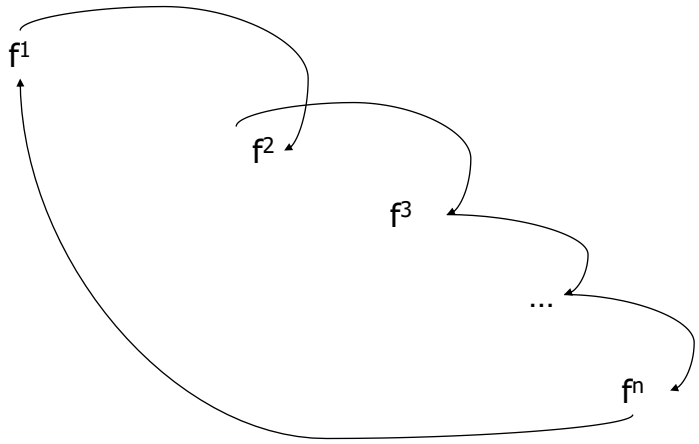



```
def fun ():
    …
    fun ()
    …
```

Indirect Call



Indirect Call



Indirect Call (2)

Name of the online example: `1simpleRecursive.py`

```
def fun1():
   fun2()

def fun2():
   fun1()

fun1()
```

Requirements For *Sensible* Recursion

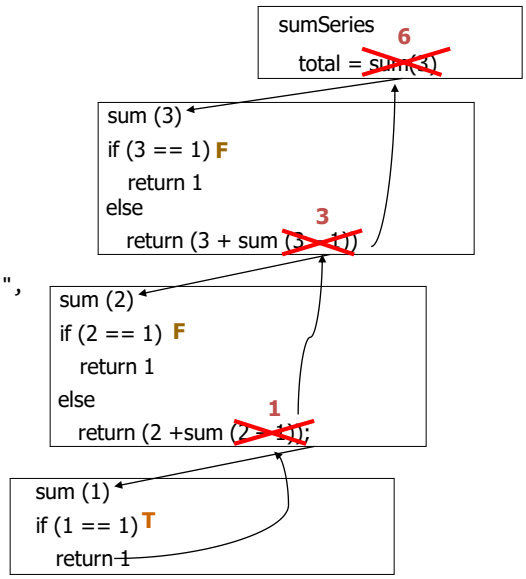
1) Base case
2) Progress is made (towards the base case)

Example Program: `2sumSeries.py`

```
def sum(no):
   if (no == 1):
      return 1
   else:
      return (no + sum(no-1) )


def start():
   last = input ("Enter the last
                  number: ")
   last = (int)last
   total = sum(last)
   print ("The sum of the series
           from 1 to", last, "is",
           total)

start()
```



When To Use Recursion

- When a problem can be divided into steps.
- The result of one step can be used in a previous step.
- There is a scenario when you can stop sub-dividing the problem into steps (step = recursive call) and return to a previous step.
  - Algorithm goes back to previous step with a partial solution to the problem (back tracking)
- All of the results together solve the problem.

When To Consider Alternatives To Recursion

- When a loop will solve the problem just as well
- Types of recursion (for both types a `return` statement is excepted)
  - **Tail recursion**
    - The last statement in the function is another recursive call to that function This form of recursion can easily be replaced with a loop.
  - **Non-tail recursion**
    - The last statement in the recursive function is not a recursive call.
    - This form of recursion is very difficult (read: impossible) to replace with a loop.

Example: Tail Recursion

- Tail recursion: A recursive call is the last statement in the recursive function.
- Name of the online example: `3tail.py`

```
def tail(no):
   if (no <= 3):
      print (no)
      tail(no+1)
   return()

tail(1)
```

Example: Non-Tail Recursion

- Non-Tail recursion: A statement which is not a recursive call to the function comprises the last statement in the recursive function.
- **Name of the online example**: `4nonTail.py`

```
def nonTail(no):
   if (no < 3):
      nonTail(no+1)
   print(no)
   return()

nonTail(1)
```

Error Handling Example Using Recursion

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

– Iterative/looping solution (month must be between 1 – 12)

```
month = -1
while ((month < 1) or (month > 12)):
        month = int(input("Enter birth month (1-12): "))
```

James Tam

Error Handling Example Using Recursion (2)

– **Name of the online example**:
`6errorHandling_Recursive.py`

– Recursive solution (day must be between 1 – 31)

```
def promptDay():
    day = int(input("Enter day of birth (1-31): "))
    if ((day < 1) or (day > 31)):
        day = promptDay()
    return(day)

day = promptDay()
print(day)
```

James Tam

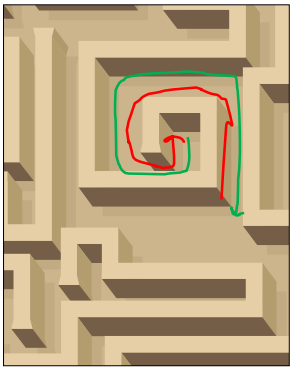
When To Use Iteration Or Recursion

- Rule of thumb for using iteration: if you can implement a solution using a loop then you should do so.
- When to employ a recursive solution: a loop cannot be employed.
  - "Back tracking" is needed.
  - Back tracking: When the repetition (whether via the iterations of a loop or a function calling itself over and over) ends the actual work of solving the problem occurs.
  - Examples: Traversing a maze, traversing a file system (folders/directories containing other folders).

James Tam

Applying Recursion: Traversing A Maze

- Picked the wrong direction in the maze?
- After repeatedly traversing the maze (going up, left, right, down) and you hit a dead end!



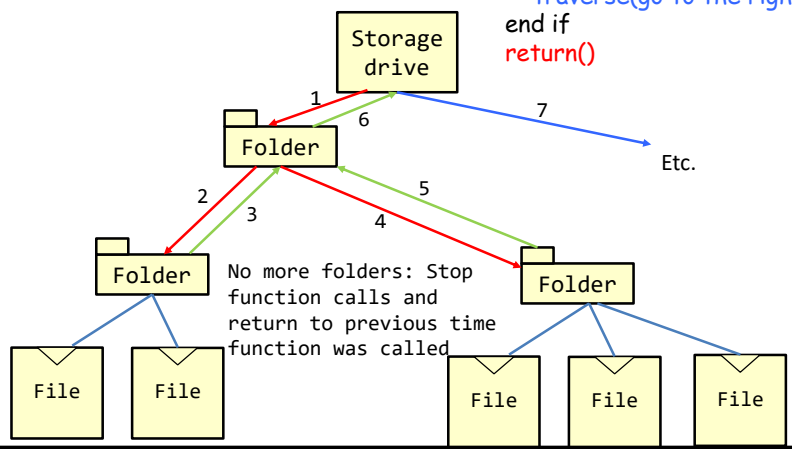
- You must "back track" (retrace your steps)

James Tam

Applying Recursion: Traversing A Directory/Folder Structure (Chart: James Tam)

Pseudo code

```
traverse(folder reference)
   If (reference leads a folder)
      traverse(go to left folder)
      traverse(go to the right folder)
   end if
   return()
```



Copyright Notification

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

James Tam