Chapter 10.
Classes and OOP

Starting out with Python

Classes and Object-Oriented Programming

Kyu Lee. Ph. D.

Computer Science

Navigator

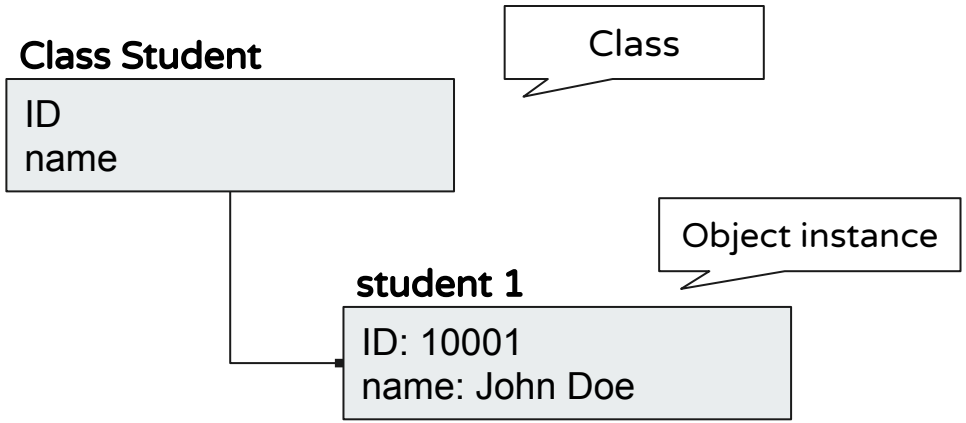
- Class Definition
- Non-Public Attributes
- Getter and Setter
- Decorator
- @property Decorator
- UML
- Class Attribute
- Class Customization

Class Definition

- Methods
  - Constructor
  - Accessor and Mutator (getter and setter)
- __str__

Classes

- Class and Object
  - Class
    - User-defined Data Structures which holds data and functions
  - An **object** is an **instance** of a **Class**
    - Class: Template
    - Object: Actual Values



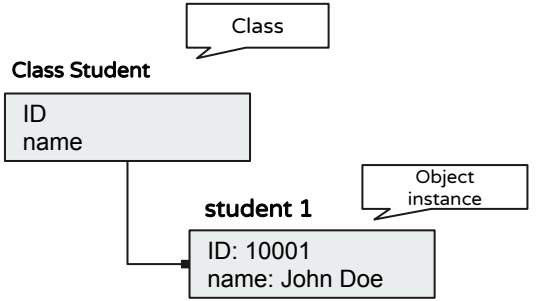
Classes

- Class and Object
  - Example

```
class Student:
        sid = 10
        sname = 'John'


s = Student ()

print (s.sid)
print (s.sname)

sid = 20
sname = 'James'
```

Class

Object instance



Constructor

- Constructor
  - A ***method*** is a function defined within a class.
  - The ***__init__*** method, commonly known as a ***constructor***, is responsible for setting up the initial state of the new instance

Object itself

```
class Student:
        def __init__(self):
                self.sid = 0
                self.sname = ' '

s = Student ( )
sid = 10
sname = 'John'

print (s.sid)
print (s.sname)
```

```
class Student:
        def __init__(self, sid, name):
                self.sid = sid
                self.sname = name

s = Student (1001, 'James')

print (s.sid)
print (s.sname)
```

Quick Check: Terms

- Make the lines between two items in the left and right sections

`class`	A name following a '.' symbol
`self`	A method parameter that refers to the class instance
`__init__`	An instantiation of a class
`instance`	A constructor methods that initializes a class instance
`attribute`	A group of related variables and functions

Quick Check:

- Classes

PARTICIPATION ACTIVITY 9.2.3: Classes.

1) A class can be used to group related variables together.
   - ○ True
   - ○ False

2) The __init__ method is called automatically.
   - ○ True
   - ○ False

3) Following the statement `t = Time()`, t references an instance of the Time class.
   - ○ True
   - ○ False

Lab 1: Class Design

- Implement the Class Rectangle
  - Attributes
    - height
    - width
  - Constructor
    - with height and width
    - e.g, Rectangle(10, 20)
- Make an object instance
  - Print the object attributes

```
class Rectangle:
    def __init__(self, h, w):
        self.height = h
        self.width = w
```

Methods

- Method
  - A function defined within a class

```
class Rectangle:

        def __init__(self, width, height):
                self.width = width
                self.height = height

        def getArea(self):
                return self.width * self.height

r1 = Rectangle(10,20)
print (r1.getArea())
```

Class Method

Call Method

Non-Public attribute

Data Abstraction

- Abstraction
  - Encapsulation or Information hiding
- ***Abstraction*** occurs when a user interacts with an object at a high level, allowing lower-level internal details to remain hidden

Don't do this

Data Abstraction: Hiding Attribute

- Non_Public Attribute
  - The attribute of the class object which cannot be accessed from the outside of the class

Data Abstraction: Hiding Attribute

- Non_Public Attribute
  - The attribute of the class object which cannot be accessed from the outside of the class
- A variable prefixed Single Leading Underscore _

It's a convention

  - should be treated as a non-public part of the API (whether it is a function, a method or a data member)

```
class Person:
    def   init  (self):
        self.name = 'Sarah'
        self._age = 26
```

Name mangling

- A variable prefixed Double Leading Underscore __
  - Python mangles these names and it is used to avoid name clashes with names defined by subclasses

```
class Person:
    def   init  (self):
        self.name = 'Sarah'
        self._age = 26
        self.__id = 30
```

Data Abstraction: Hiding Attribute

- A variable prefixed Single Leading Underscore _
  - should be treated as a non-public part of the API (whether it is a function, a method or a data member)
  - It is just a convention
    - It still can be accessed outside of the class

```
class Person:
   def   init  (self):
       self.name = 'Sarah'
       self._age = 26
```

```
>>> p = Person()
>>> p.name
Sarah
>>> p._age
26
```

Can be accessed outside of class

Data Abstraction: Hiding Attribute

- A variable prefixed Double Leading Underscore __

```
class Rectangle:

        def __init__(self, width, height):
                self.__width = width
                self.__height = height

        def getArea(self):
                return self.__width * self.__height

r1 = Rectangle(10,20)

print (r1.getArea())

# r1.__height
```

Error. No attribute

```
r1._Rectangle__height    # can be accessed with the class name
```

getter and setter
: accessor and mutator

getter and setter

- You should provide getter and setter methods, also known as accessors and mutators, respectively.
  - These methods offer a way to change the internal implementation of your attributes without changing your public API

```
class Point:
        def __init__(self,x,y):
                self.__x = x
                self.__y = y

        def get_x(self):
                return self.__x
        def set_x(self, x):
                self.__x = x
        def get_y(self):
                return self.__y
        def set_y(self, y):
                self.__y = y

p1 = Point(10, 20)
print(p1.get_x(), p1.get_y())
```

getter

setter

since p1.__x # Cannot be accessed

Property

- Properties represent an intermediate functionality
  - between a plain **attribute** (or field) and a **method**.
  - In other words, they allow you to create methods that behave like attributes.
    - For example, you can turn both `.x` and `.y` into properties.

- Python's `property()` is
  - the Pythonic way to avoid formal getter and setter methods in your code.
  - This function allows you
    - to turn class attributes into properties or managed attributes.

Pythonic Way: Property Class

- Property Class

  returns a property object.

- The property() class has the following syntax:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

  The property() has the following parameters:

  - fget is a function to get the value of the attribute, or the getter method.
  - fset is a function to set the value of the attribute, or the setter method.
  - fdel is a function to delete the attribute.
  - doc is a docstring i.e., a comment.

```
x = property(get_x, set_x)
y = property(get_y, set_y)
```

Property

- Single Leading Underscore Attributes

```
class Point:
        def __init__(self,x,y):
                self.x = x
                self.y = y

        def get_x(self):
                print('through get_x()')
                return self._x
        def set_x(self, x):
             if x < 0:
              self._x = 0
             else:
              self._x = x
        def get_y(self):
                print('through get_y()')
                return self._y
        def set_y(self, y):
                self._y = y

        x = property(get_x, set_x)
        y = property(get_y, set_y)
```

Property

```
>>> p1 = Point(50, 50)
>>> print (p1.get_x(), p1.get_y())
through get_x()
through get_y()
50 50
>>> p1.x = -10      # x = 0
>>> print (p1._x, p1.y) # _x without getter
through get_y()
0 50
>>> print (p1.x)
0
```

Since In set_x( ), if x < 0, x = 0

Call getter of x and y

```
# can be accessed _x directly
without setter.
p1._x = -20
print (p1._x)
```

What's Difference?

- Without Property?

```
class Point:
       def __init__(self,x,y):
              self._x = x
              self._y = y

       def get_x(self):
              print('through get_x()')
              return self._x
       def set_x(self, x):
            if x < 0:
             self._x = 0
            else:
             self._x = x
       def get_y(self):
              print('through get_y()')
              return self._y
       def set_y(self, y):
              self._y = y
```

```
>>> p1 = Point(50, 50)
>>> print (p1.get_x(), p1.get_y())
through get_x()
through get_y()
50 50
>>> p1._x = -10    # x = -10
>>> print (p1._x, p1._y)
-10 50
>>> p1.x = 100
>>> print (p1._x)
-10      # not 100
         # x is the distinct attribute
```

Call get_x() even there is no property()

No calling set_x( ) since there is no property( )

Direct access to Attribute

Property

- Double Leading Underscore

```
class Point:
       def __init__(self,x,y):
              self.x = x
              self.y = y

       def get_x(self):
              print('through get_x()')
              return self.__x
       def set_x(self, x):
            if x < 0:
             self.__x = 0
            else:
             self.__x = x
       def get_y(self):
              print('through get_y()')
              return self.__y
       def set_y(self, y):
              self.__y = y

       x = property(get_x, set_x)
       y = property(get_y, set_y)
```

```
>>> p1 = Point(50, 50)
>>> print (p1.get_x(), p1.get_y())
through get_x()
through get_y()
50 50
>>> p1.x = -10      # x = 0 set_x() is called
>>> print (p1.__x, p1.y) # Error
>>> print (p1._Point_x) # can be accessed
0
```

Lab 2: Class Rectangle with Property

- Implement the Class Rectangle (See the Page 21 and implement the same program)
  - Attributes
    - _height
    - _width
  - Constructor
    - with height and width
    - e.g, Rectangle(10, 20)
  - getter and setter:
    - get_width(), set_width(), get_height(), set_height()
  - Property( )
    - for width, width = property(get_width, set_width)
    - for height, height = property(get_hight, set_height)

```
class Rectangle:
    def __init__(self, h, w):
        self._height = h
        self._width = w
```

- Make an object instance
  - Change the attribute values through the set_width( ) or directly with the attributes name
  - Explain the difference between getting value through `height` and `get_height()`

Decorator

Reminder of Function Concepts

- Remind the function concepts
  - A function is an instance of the Object type.
  - You can store the function in a variable.
  - You can pass the function as a parameter to another function.
  - You can return the function from a function.

Reminder of Function Concepts

- Store the function in a variable

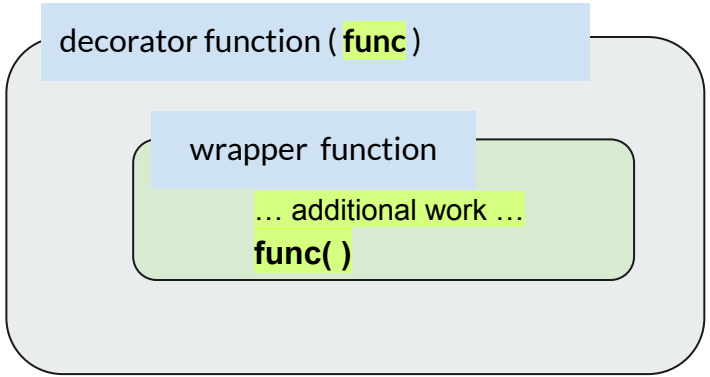
```
def addTwo(x, y):
        return x+y

sumTwo = addTwo
print (sumTwo(10, 20))
```

- Pass a function as a parameter / Return the function

```
def flex_adder(x):
        def inner_adder(y):
                return x+y
        return inner_adder

myadder10 = flex_adder(10)

print (myadder10(20))   # 30
```

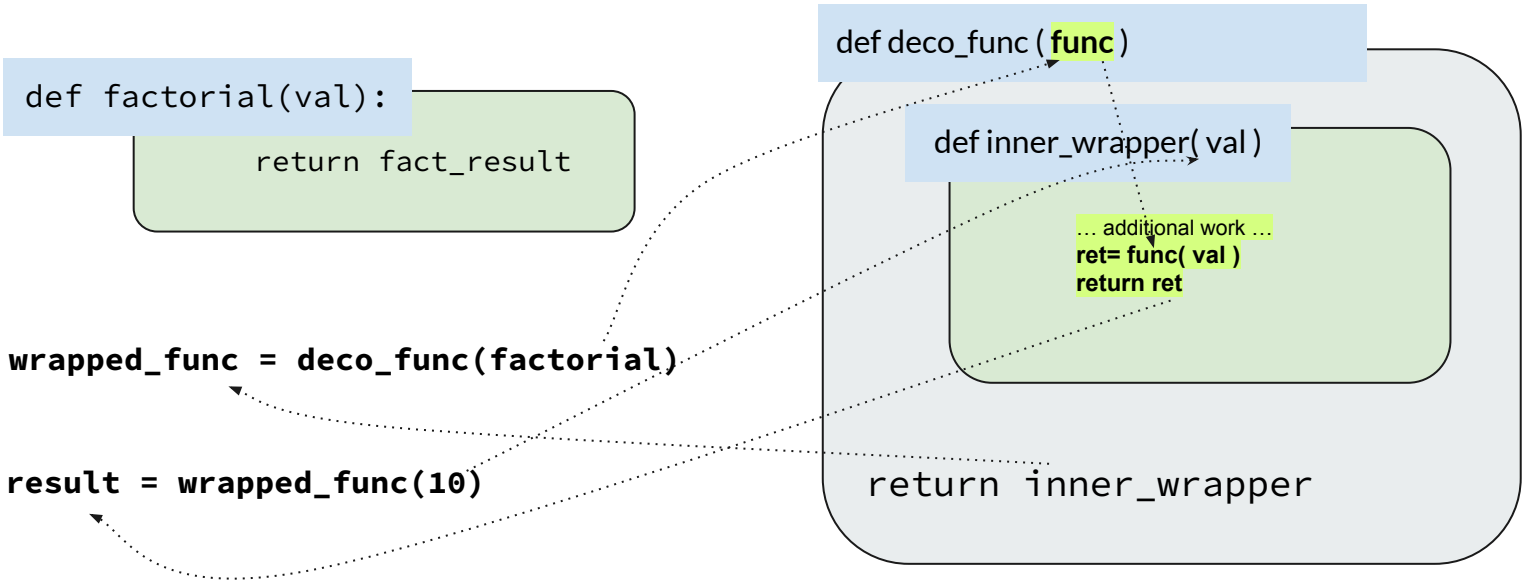
Decorator

- The purpose of Decorator
  - Can modify the behaviour of the function or class
- In Decorator,
  - the functions are taken as the **arguments** into another function
  - and then, **called** inside the **wrapper** function



Decorator

- Decorator Example
  - We have a function that return the factorial number
  - In Decorator, we will add the ‘time measurement’ to the factorial function



Decorator

- Decorator Example
  - We have a function that return the factorial number
  - In Decorator, we will add the ‘time measurement’ to the factorial function

```
import time

def factorial(num):
        fact = 1
        for i in range(1,num):
                fact *= i
        return fact


def deco_fact(func):
        #def inner_wrapper(*args, **kwargs):
        def inner_wrapper(val):
                begin = time.time()
                # ret = func(*args, **kwargs)
                ret = func(val)
                end = time.time()
                print ('Elapsed time',end-begin)
                return ret
        return inner_wrapper
```

```
>>> factorial(10)
362880

>>> wrapped_func = deco_fact(factorial)

>>> wrapped_func(10)
Elapsed time 1.9073486328125e-06
362880
```

@decorator

- Decorator Example

```
import time
def deco_fact(func):
        # def inner_wrapper(*args, **kwargs):
        def inner_wrapper(val):
                begin = time.time()
                # ret = func(*args, **kwargs)
                ret = func(val)
                print(f'Factorial of {val} is {ret}')
                end = time.time()
                print ('Elapsed time',end-begin)
                return ret
        return inner_wrapper

@deco_fact
def factorial(num):
        fact = 1
        for i in range(1,num):
                fact *= i
        return fact
```

```
>>> factorial(10)

Elapsed time 1.9073486328125e-06
362880
```

`@deco_fact`
is the same as
`factorial = deco_fact(factorial)`

Now, when we call factorial( ),
it will call inner_wrapper( )

Lab 3: Decorator

- Make the same program as below
- Run and test with calling
  - fatorial(10)

```
import time
def deco_fact(func):
        # def inner_wrapper(*args, **kwargs):
        def inner_wrapper(val):
                begin = time.time()
                # ret = func(*args, **kwargs)
                ret = func(val)
                print(f'Factorial of {val} is {ret}')
                end = time.time()
                print ('Elapsed time',end-begin)
                return ret
        return inner_wrapper

@deco_fact
def factorial(num):
        fact = 1
        for i in range(1,num):
                fact *= i
        return fact
def main():
   factorial(10)
```

- **Answer the following questions**
  - What makes the function factorial( ) can print “Elapsed time” ?
  - Explain the detail process how to print “elapsed time” when you call just “factorial(10)”

@property Decorator

@property decorator

- @property decorator is
  - a built-in decorator in Python which is helpful in defining the properties effortlessly
  - without manually calling the inbuilt function property().

```
class Point:
        def __init__(self,x,y):
                self.x = x
                self.y = y

        def get_x(self):
                print('through get_x()')
                return self._x
        def set_x(self, x):
                if x < 0:
              self._x = 0
             else:
              self._x = x
        def get_y(self):
                print('through get_y()')
                return self._y
        def set_y(self, y):
                self._y = y

        x = property(get_x, set_x)
        y = property(get_y, set_y)
```

@property

@x.setter

@property

@y.setter

Instead of this property( )

@property decorator

```
class Point:
        def __init__(self,x,y):
                self._x = x
                self._y = y

        @property
        def x(self):
                print('through getter')
                return self._x

        @x.setter
        def x(self, x):
                if x < 0:
                        self._x = 0
                else:
                        self._x = x
        @property
        def y(self):
                print('through getter')
                return self._y
        @y.setter
        def y(self, y):
                self._y = y

        # x = property(get_x, set_x)
        # y = property(get_y, set_y)
```

@property

@x.setter

Instead of this property( )

```
>>> p1 = Point(50, 50)
>>> p1.x = -10      # x = 0 setter is called
>>> print (p1.x, p1.y)
through getter()
through getter()
0 50
>>> p1._x = -10 # can be accessed
                  # without setter
>>> print (p1.x, p1.y)
through getter()
through getter()
-10 50
>>> print (p1._x, p1._y) #without getter
-10 50
```

Lab 4: Class Rectangle with @property decorator

- Design the Class Rectangle

"Design a class called `Rectangle` with the following attributes:

- `_height`: representing the height of the rectangle
- `_width`: representing the width of the rectangle

Implement the class constructor to initialize the `Rectangle` object with the given height and width. For example, `Rectangle(10, 20)` should create a `Rectangle` object with a height of 10 and a width of 20.

Use the `@property` decorator to define getters and setters for both the width and height attributes. Ensure that the getters and setters are implemented using the `@property` statements.

Your task is to write the class `Rectangle` and include the constructor, as well as the appropriate getters and setters using the `@property` decorator."

Lab 4: Class Rectangle with @property decorator

- Design the Class Rectangle
  - Attributes
    - _height
    - _width
  - Constructor
    - with height and width
    - e.g, Rectangle(10, 20)
  - @property
    - for width and height

```
class Rectangle:
    def __init__(self, h, w):
        self._height = h
        self._width = w
```

@property

@width.setter

@property

@height.setter

- Make an object instance
  - Change the attribute values through the setter( ) or directly with the attributes name
  - Print the object attributes

UML;
Unified Modeling Language

UML Diagram

UML diagram

- When designing a class, it is often helpful to draw a UML diagram
- provides a set of standard diagrams for graphically depicting object-oriented systems.

Figure 10-9 General layout of a UML diagram for a class

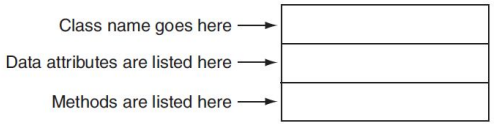


Figure 10-11 UML diagram for the CellPhone class

CellPhone
__manufact __model __retail_price
__init__(manufact, model, price) set_manufact(manufact) set_model(model) set_retail_price(price) get_manufact() get_model() get_retail_price()

Class Attributes

Class Attributes

- A **class attribute** is a variable that belongs to a certain class, and not a particular object.
  - Every instance of this class shares the *same* variable.
  - These attributes are usually defined outside the `__init__` constructor.
- An **instance/object attribute** is a variable that belongs to one (*and only one*) object.
  - Every instance of a class points to its own attributes variables.
  - These attributes are defined within the `__init__` constructor.
  -

Class Attributes Example

```
class Student:

        numofStudent = 0

        def __init__(self, id, name):
                self.id = id
                self.name = name
                Student.numofStudent = Student.numofStudent + 1

        @property
        def id(self):
                return self._id
        @id.setter
        def id(self, id):
                self._id = id
        @property
        def name(self):
                return self._name
        @name.setter
        def name(self, name):
                self._name = name
```

```
>>> s1 = Student(1001, 'John')
>>> s2 = Student(1002, 'Kay')


>>> print (Student.numofStudent)
2
>>> print (s2.numofStudent)
2
```

Practice the Class Implementation

Lab 5: Class Implementation

- Read all slides from page 43 to 48
  - Implement the **class Student**
  - Make the functions
    - **makeStudent(student_dict_list)**
    - **deleteOneStudent(slist, did)**
- Test your program
  - clone this repository https://github.com/LPC-CSDept/CS7L1005
  - complete main.py and then
    - python main.py
  - if there is no error
    - and then ‘pytest -rP’ and see the test result

Lab 5: [1] Class `Student` Implementation

"Implement a class called `Student` with the following properties:

- `_id`: representing the ID of the student
- `_name`: representing the name of the student

Define a class attribute called `numofStudents` to keep track of the total number of student objects created.
In the constructor of the `Student` class, whenever a new object is created, the `numofStudents` class attribute should be increased by 1.
Use the `@property` decorator to declare getters and setters for both the `_id` and `_name` properties.

- Implement the **`class Student`**
  - **Properties**
    - **`_id`**
    - **`_name`**
- **Requirements**
  - **Class Attribute should be defined `'numofStudents'`**
    - In constructor, whenever the object is created, 'numofStudents' should be increased by 1
  - **`_id and _name` should be declared with `@property`**
  - **add the __str__(self) method to the class Student**

```
def __str__(self):
  return (f'Student id: {self.id:>10} \t name:{self.name:>10}')
```

Lab 5: [2] `makeUpStudent( )`

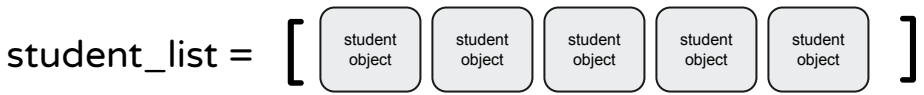
- `makeUpStudent(student_dict_list)`

  write the `makeUpStudent` function, which takes the `student_dict_list` as a parameter and returns a list of `Student` objects created from the dictionaries in the `student_dict_list`.

  Each dictionary in the `student_dict_list` represents student information, with keys `'id'` and `'name'` corresponding to the student's ID and name, respectively. For example:

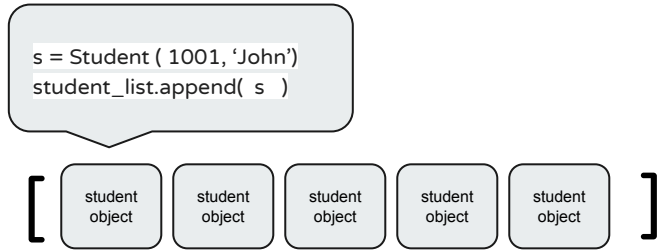
```
student_dict_list = [   {'id': 1001, 'name': 'John'},
                        {'id': 1002, 'name': 'James'},
                        {'id': 1003, 'name': 'Mark'},
                        {'id': 1004, 'name': 'Matthew'},
                        {'id': 1005, 'name': 'Arnold'}]
```

- Increase the class attribute '`numofStudent`' when the object has been added to the list
- Return value



Lab 5: [2] `makeUpStudent( )`

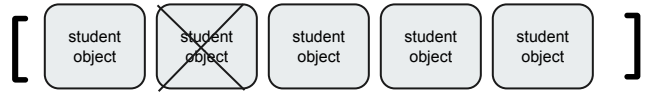
- `makeUpStudent(student_dict_list)`
  - [{'id':1001, 'name':'John'}, {'id':1002, 'name':'James'},{'id':1003, 'name':'Mark'}, {'id':1004, 'name':'Matthew'}, {'id':1005, 'name':'Arnold'}]
    - create an object and append it to the list
    - increase the **`numofStudent`** by 1
    - Repeat with all dictionary items
  - Return **`student_list`**



- After Call **`makeUpStudent( )`**,
  - check out the class attribute **`numofStudent`** value
  - it must be 5

Lab 5: [3] `deleteOneStudent()`

- `deleteOneStudent(s_list, did)`
  - delete one student object who has the '**did**' value from the list
  - When you delete the object
    - Use the 'del' statement
    - to delete it from the memory
    - and then remove from the list
  - decrease by 1 **`numofStudent`**

student_list = 


```
if did == student_list[i].did
      del student_list[i]
      student_list.remove(did)
numofStudent -= 1
```

Class Customization

- Operator Overloading

__str__( ) method

- To customize a class
  - you can implements instance methods with “**special method names** ” that the Python interpreter supports
  - Example,
    - To change how a class instance object is printed,
    - **__str__( )** method can be defined.

```
class Person:
   def   init  (self):
        self.name = 'Sarah'
        self. age = 26
   def __str__(self):
         return f’Name: {self.name:>10}, Age: {self.age:<10}’
```

```
>>> p = Person()
>>> print (p)
Name:      Sarah, Age:         26
```

Operator Overloading

- Class customization can redefine the functionality of built-in operator like
  - <, >=, +, - and *
  - `object.__lt__(self, other)`
  - `object.__le__(self, other)`
  - `object.__eq__(self, other)`
  - `object.__ne__(self, other)`
  - `object.__gt__(self, other)`
  - `object.__ge__(self, other)`
- These are the so-called “rich comparison” methods. The correspondence between operator symbols and method names is as follows:
  - `x<y` calls `x.__lt__(y)`,
  - `x<=y` calls `x.__le__(y)`,
  - `x==y` calls `x.__eq__(y)`,
  - `x!=y` calls `x.__ne__(y)`,
  - `x>y` calls `x.__gt__(y)`,
  - and `x>=y` calls `x.__ge__(y)`

Lab 6: Operator Overloading

Create a class called **`Rectangle`** with the following specifications:

- Object attributes: **`_width`** and **`_height`** representing the width and height of the rectangle, respectively.
- Use the **`@property`** decorator to define getters and setters for both `_width` and `_height`.
- Implement the **`__init__`**`(self, width, height)` method to initialize the `Rectangle` object with the given width and height.
- Implement the **`__str__`** method to provide a user-friendly output format when printing the width and height of the rectangle.
- Implement the **`__gt__`**`(self, other)` and **`__lt__`**`(self, other)` methods to compare rectangles based on their areas.

- Make the class Rectangle
  - Object Attributes:
    - **`_width`**
    - **`_height`**
    - **`@property and @width.setter`**
    - **`@property and @height.setter`**
      - **`Using the decorator @property for _width and _height`**
  - __init__(self, width, height)
  - __str__
    - Display the width and height in a user-friendly output format.
  - __gt__ , __lt__
    - compare the area
- Test your program
  - pytest -rP

Operator Overloading

- Methods for emulating numeric types

Method	Description
__add__(self, other)	Add (+)
__sub__(self, other)	Subtract (-)
__mul__(self, other)	Multiply (*)
__truediv__(self, other)	Divide (/)
__floordiv__(self, other)	Floored division (//)
__mod__(self, other)	Modulus (%)
__pow__(self, other)	Exponentiation (**)
__and__(self, other)	"and" logical operator
__or__(self, other)	"or" logical operator
__abs__(self)	Absolute value (abs())
__int__(self)	Convert to integer (int())
__float__(self)	Convert to floating point (float())

Lab 7: Operator Overloading

Create a class called **`Student`** with the following specifications:

- Class Attribute: **`numofStudent`**, which will be automatically incremented by 1 in the `__init__` function.
- Object Attributes: **`name`** representing the student's name, and **`scores`** representing a list of integer values representing the student's scores.
- Use the **`@property`** decorator to define getters and setters for both `name` and `scores`.
- Implement the **`__init__`**`(self, name, scores)` method to initialize the `Student` object with the given name and scores.
  - name: the string value for name
  - scores: list of the scores, e.g, [100, 90, 100]
- Implement the **`__str__`** method to provide a user-friendly output format when printing the student's name and scores.
- Implement the **`__gt__`**`(self, other)` and **`__lt__`**`(self, other)` methods to compare students based on the summation of their scores.
- Implement the **`__sub__`**`(self, other)` method to get the difference between the average scores of two students.

Lab 7: Operator Overloading

- Make the class Student
  - Class Attribute : **`numofStudent`**
    - It will be maintained to increase by 1 automatically in the __init__ function
  - Object Attributes:
    - **`name`**
    - **`scores:`** `list of integer values`
    - **`@property / @name.setter`**
    - **`@property /@scores.setter`**
  - `__init__(self, name, scores)`
    - `name: student's name, scores: list of scores`
  - `__str__`
    - Print name and scores with user-friendly output format
  - `__gt__ , __lt__`
    - compare the summation of scores
  - `__sub__`
    - Get the difference between two student's average of scores

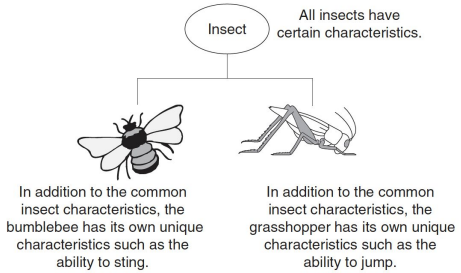
Lab 7: Operator Overloading

- Run Example of class Student
  - `s1 = Student('John', [80, 80,80])`
  - `s2 = Student('James', [100, 100, 100])`
  - `diff = s2 - s1`
  - `print (diff)    # diff should be 20. Difference between average of scores.`
  - `s1 > s2    # it should be False. Comparison of average of scores`
  - `print (numofStudent)  # it should be 2.`
- Test your program here
  - https://github.com/LPC-CSDept/CS7L1007

Inheritance

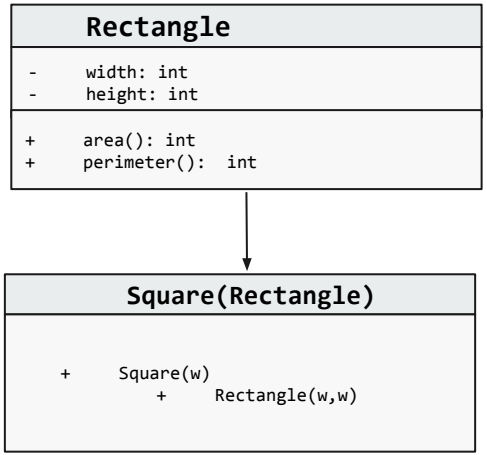
Inheritance

- Inheritance allows a new class to extend an **existing** class.
  - The new class **inherits** the members of the class it extends.
- Inheritance
  - the “Is a” Relationship
- Superclasses are also called base classes,
- and subclasses are also called derived classes.



Inheritance

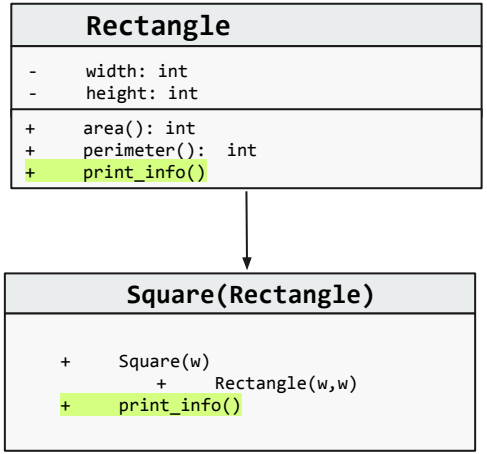
- Inheritance Example




```
class Rectangle:
        def __init__(self, w, h):
                self.width = w
                self.height = h
        @property
        def width(self):
                return self._width
        @width.setter
        def width(self, w):
                self._width = w
        @property
        def height(self):
                return self._height
        @height.setter
        def height(self, h):
                self._height = h
        def area(self):
                return self._height * self._width
        def perimeter(self):
                return self._height * 2 + self._width * 2

class Square(Rectangle):
        def __init__(self, w):
                Rectangle.__init__(self, w, w)

c = Square(10)
print (c.area())
print (c.perimeter())
```

Overriding Class Methods

- A derived class may define a method having the **same name** as a method in the base class.
  - Such a member function ***overrides*** the method of the base class.




```
class Rectangle:
        def __init__(self, w, h):
                self.width = w
                self.height = h

    ...

        def print_info(self):
                print (f'Rectangle Information. Width {self._width:<10}, Height: {self._height:<10}')


class Square(Rectangle):
        def __init__(self, w):
                Rectangle.__init__(self, w, w)
        def print_info(self):
                print (f'Rectangle Information. Width {self._width:<10}')

s = Square(10)
s.print_info()
```

Lab 8: Inheritance

Create a class called **`Person`** with the following specifications:

- Object attributes: **`name`** representing the person's name, **`addr`** representing the person's address, and **`tel`** representing the person's phone number.
- Implement the **`__init__`**`(self, name, addr, tel)` method to initialize the `Person` object with the given name, address, and phone number.
- Use the **`@property`** decorator to define getters and setters for all attributes `name, addr, and tel`

Next, create a class called **`Student`** that is derived from the `Person` class, with the following additional specifications:

- Class attribute: **`numofStudent`**, which will be automatically incremented by 1 in the `__init__` method.
- Object attributes: **`sid`** representing the student's ID, and **`scores`** representing a list of integer values representing the student's scores. Use the **`@property`** decorator to define getters and setters for all attributes
- Implement the **`__init__`**`(self, sid, scores)` method to initialize the `Student` object with the given student ID and scores. Additionally, make sure to call the `__init__` method of the `Person` class to initialize the inherited attributes.
- Implement the **`__str__`** method to provide a user-friendly output format when printing the student's object information, including the name, address, phone number, student ID, and scores.
- Implement the **`__gt__`**`(self, other)` and **`__lt__`**`(self, other)` methods to compare students based on the summation of their scores.
- Implement the **`__sub__`**`(self, other)` method to get the difference between the average scores of two students.

Lab 8: Inheritance

- Make the class Person
  - Object Attributes
    - **name: Person's name**
    - **addr: address**
    - **tel : phone number**
  - **def __init__(self, name, addr, tel)**
- Make the class Student derived from Person
  - Class Attribute : **numofStudent**
    - It will be maintained to increase by 1 automatically in the __init__ function
  - Object Attributes:
    - **sid: student's id**
    - **scores:** `list of integer values`
  - `__init__(self, sid, scores)`
    - `name: student's sid, scores: list of scores`
  - __str__
    - Print all student's object information with user-friendly output format
  - __gt__ , __lt__
    - compare the summation of scores
  - __sub__
    - Get the difference between two student's average of scores