PASADENA CITY COLLEGE

CIS 112

Intro to Programming Using Python

Module 2 Part 3

Agenda for the Day!

- Still more OOP stuff!
  - extended function argument syntax and decorators;
  - static and class methods;
  - attribute encapsulation;

Yet more Advanced Class and OOP Topics

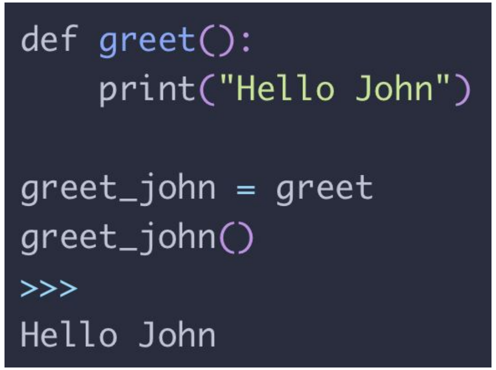
Decorators

Python Decorators

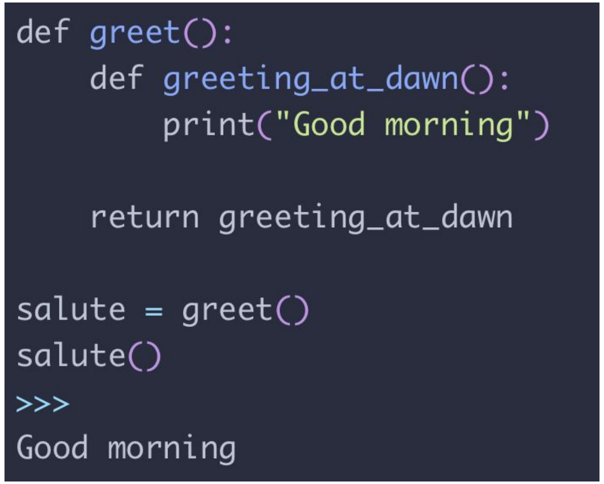
- With inheritance/polymorphism we introduced the ability to take class objects and, without modifying the base class, effectively extend their functionality by creating child classes.
  - Can we leverage the same conceptual approach to functional programming? YES! With **DECORATORS**
- **Decorators** are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of a function or class.
- Decorators allow us to **wrap another function** in order to **extend the behaviour of the wrapped function**, without permanently modifying it.
- Key Elements to note:
  - In Python, functions are **first class objects** which means that functions in Python can be used or passed as arguments.
    - Properties of first class functions:
      - 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**.
      - You can **store** them in **data structures** such as hash tables, lists, …

More Python Decorators!

- A **function decorator** in Python is just a function that takes in another function as an argument, extending the decorated function’s functionality without changing its structure.
- In other words, a decorator wraps another function, amplifies its behavior, and returns it.
- So let’s see how we go about creating that…
    - First, remember, as a function is an object, it means it can be assigned to another variable
    - A function can also be returned from another function



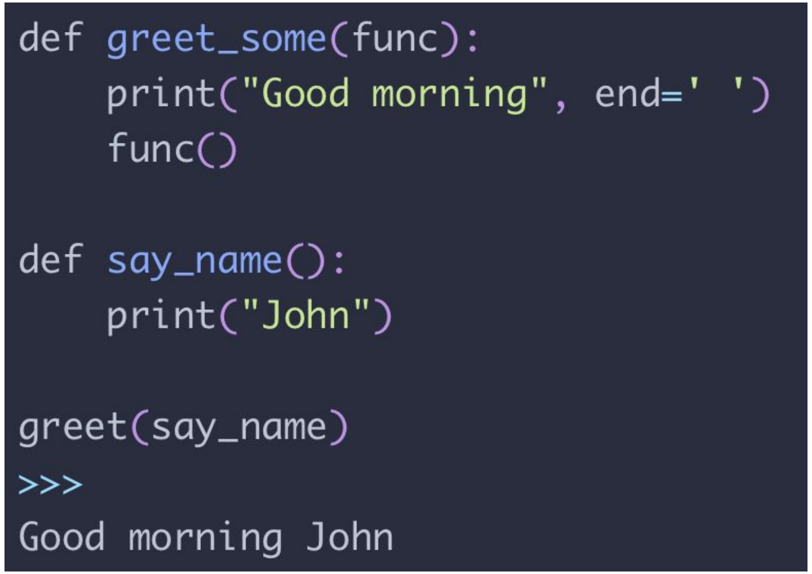
```
def greet():
    print("Hello John")

greet_john = greet
greet_john()
>>>
Hello John
```



```
def greet():
    def greeting_at_dawn():
        print("Good morning")

    return greeting_at_dawn

salute = greet()
salute()
>>>
Good morning
```

Still...more decorators?

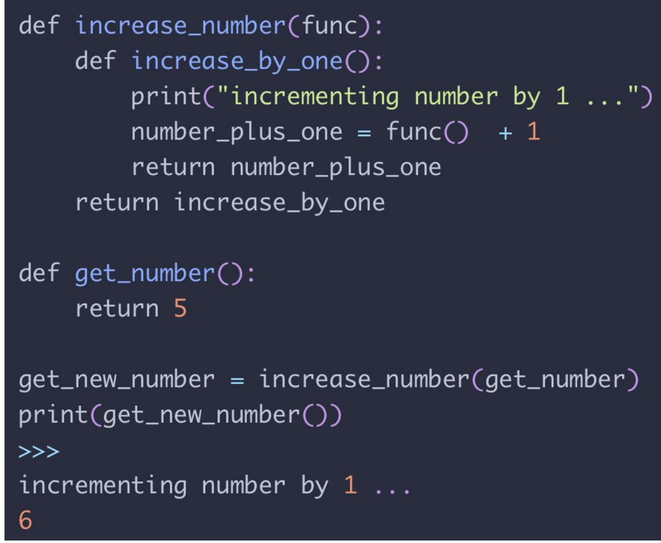
- A function can also be passed as an argument of another function
  - A function that receives a function argument is known as a **higher order function.**



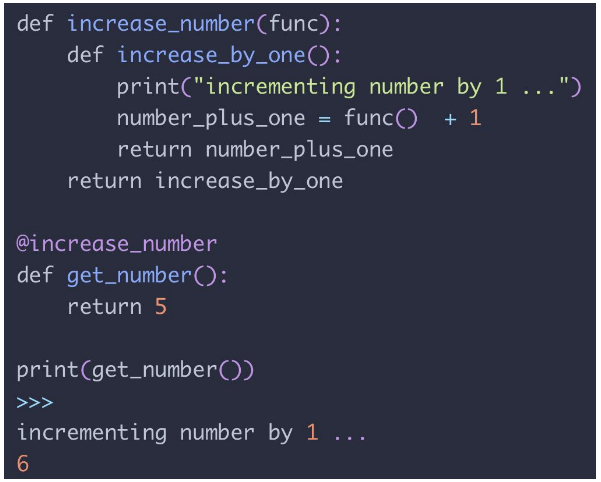
```
def greet_some(func):
    print("Good morning", end=' ')
    func()

def say_name():
    print("John")

greet(say_name)
>>>
Good morning John
```

Are we really still doing decorators?

- A simple decorator function **starts with a function definition**, the **decorator function**, and **then a nested function within the outer wrapper function**.
- Always keep these two main points in mind when defining decorators:
  - To implement decorators, define an outer function that takes a function argument.
  - Nest a wrapper function within the outer decorator function, which also wraps the decorated function.
- A simple decorator function is easily identified when it begins with the **@ prefix, coupled with the decorated function underneath.**



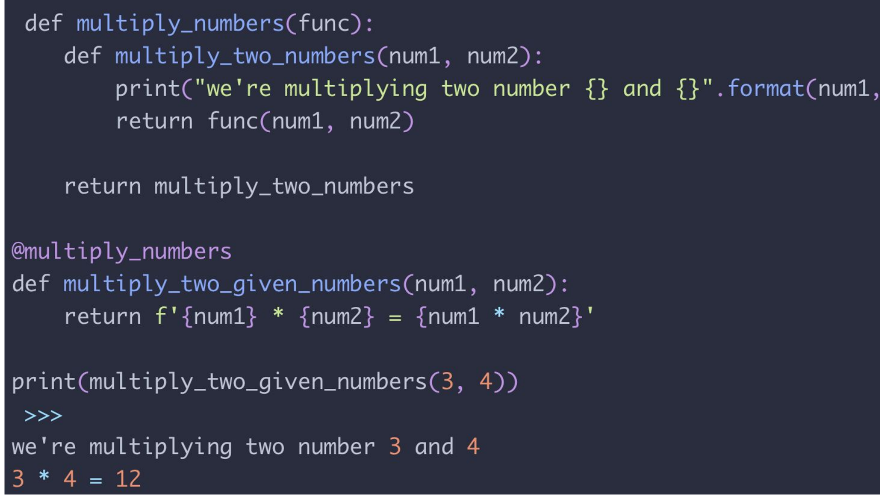
```
def increase_number(func):
    def increase_by_one():
        print("incrementing number by 1 ...")
        number_plus_one = func()  + 1
        return number_plus_one
    return increase_by_one

def get_number():
    return 5

get_new_number = increase_number(get_number)
print(get_new_number())
>>>
incrementing number by 1 ...
6
```



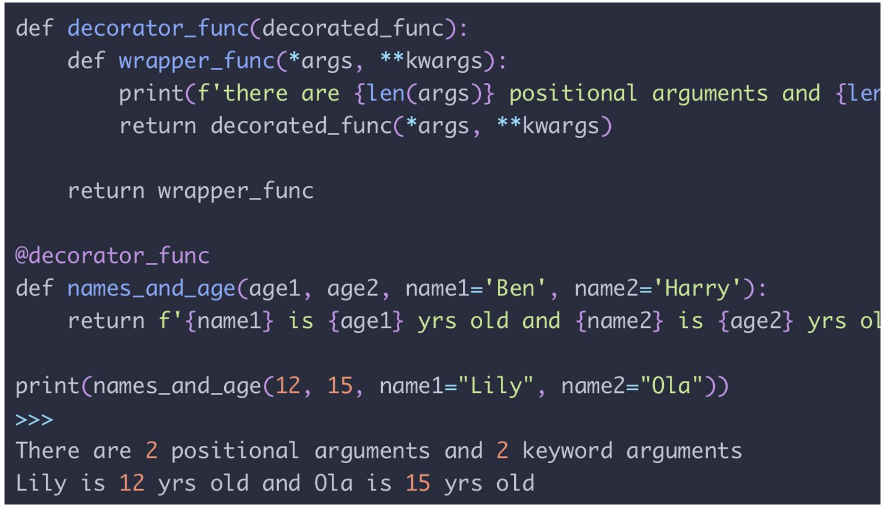
```
def increase_number(func):
    def increase_by_one():
        print("incrementing number by 1 ...")
        number_plus_one = func()  + 1
        return number_plus_one
    return increase_by_one

@increase_number
def get_number():
    return 5

print(get_number())
>>>
incrementing number by 1 ...
6
```

OMG make it stop!

- There are cases where you may need to **pass parameters to a decorator**. The way around this is to **pass parameters to the wrapper function**, which are then passed down to the decorated function.
- Having parameters passed to the inner function or nested function makes it even more powerful and robust, as it gives more flexibility for manipulating the decorated function.
- Any number of arguments (***args**) or keyword arguments (****kwargs**) can be passed unto the decorated function.
- ***args** allows the collection of all positional arguments, while the ****kwargs** is for all keyword arguments needed during the function call.



```
 def multiply_numbers(func):
    def multiply_two_numbers(num1, num2):
        print("we're multiplying two number {} and {}".format(num1,
        return func(num1, num2)

    return multiply_two_numbers

@multiply_numbers
def multiply_two_given_numbers(num1, num2):
    return f'{num1} * {num2} = {num1 * num2}'

print(multiply_two_given_numbers(3, 4))
 >>>
we're multiplying two number 3 and 4
3 * 4 = 12
```



```
def decorator_func(decorated_func):
    def wrapper_func(*args, **kwargs):
        print(f'there are {len(args)} positional arguments and {len
        return decorated_func(*args, **kwargs)

    return wrapper_func

@decorator_func
def names_and_age(age1, age2, name1='Ben', name2='Harry'):
    return f'{name1} is {age1} yrs old and {name2} is {age2} yrs ol

print(names_and_age(12, 15, name1="Lily", name2="Ola"))
>>>
There are 2 positional arguments and 2 keyword arguments
Lily is 12 yrs old and Ola is 15 yrs old
```

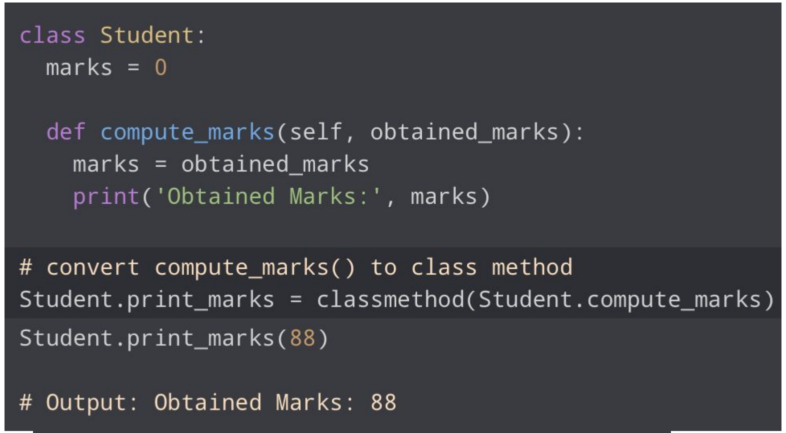
Static vs. Class Methods

Method Maddness!

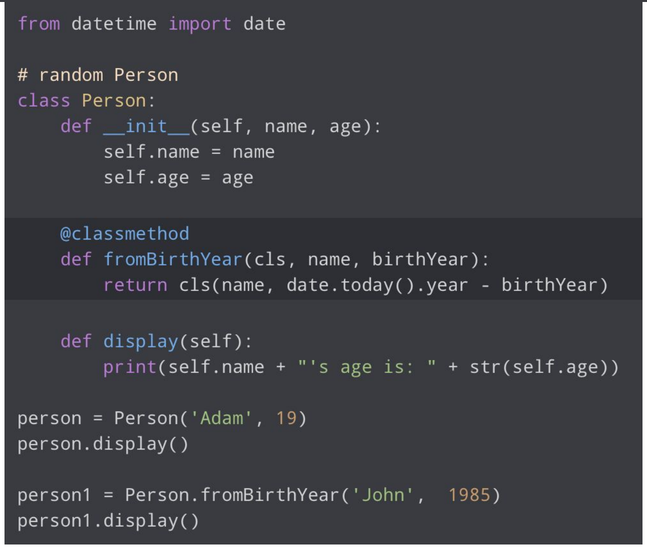
- We need to now explore how methods defined within a class are bound to that class.
  - We already introduced the idea that attributes could be class-level, or object-level, so we're going to extend this paradigm to methods as well.
  - Up till now all methods we've defined within a class are bound to specific objects. The object must first be initiated, and then the method can be invoked specifically to the scope of the object (i.e., it can modify object-level attributes)
- But what about class-level methods? Do we have the ability to define functions that act on the class-level?
- Turns out we do! And we'll explore two elements: **Class** and **Static** methods

Class Methods

- A class method is a method that is bound to a class rather than its specific objects. It doesn't require creation of a class instance, (much like static methods, but stay tuned to those!).
- The difference between a static method and a class method is:
  - Static method knows nothing about the class and just deals with the parameters
  - Class method works with the class since its parameter is always the class itself.
- When do you use the class method?
  - **Factory methods** factory methods are methods that return a class object (like constructor) for different use cases (think of it like a customized object generator.
  - **Class-Level modifiers** can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances.
- The class method can be called both by the class and its object.
- Syntax:
  - In this case the classmethod() function converts the method into a class-level method allowing for modification of class-level data
  - Newer Python versions, you can use the @classmethod decorator for classmethod definition.
    - ‘Cls’ has to then be included as a parameter as the class itself is passed in



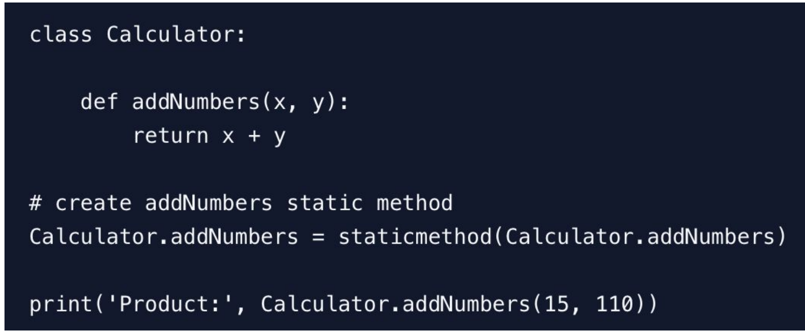
```
class Student:
  marks = 0

  def compute_marks(self, obtained_marks):
    marks = obtained_marks
    print('Obtained Marks:', marks)

# convert compute_marks() to class method
Student.print_marks = classmethod(Student.compute_marks)
Student.print_marks(88)

# Output: Obtained Marks: 88
```



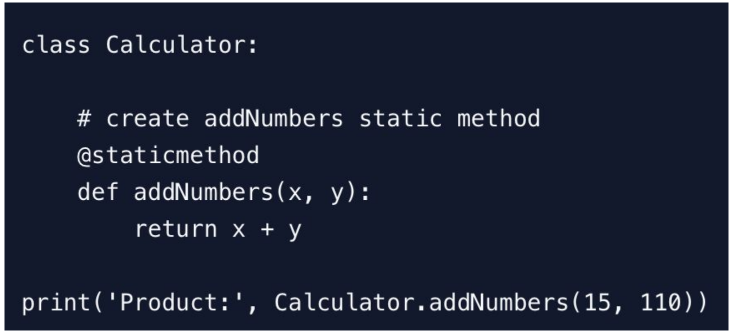
```
from datetime import date

# random Person
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def fromBirthYear(cls, name, birthYear):
        return cls(name, date.today().year - birthYear)

    def display(self):
        print(self.name + "'s age is: " + str(self.age))

person = Person('Adam', 19)
person.display()

person1 = Person.fromBirthYear('John',  1985)
person1.display()
```

Static Methods

- Static methods in Python are extremely similar to python class level methods, in that they are both defined in the scope of a class definition, are **bound to a class rather than the objects for that class,** and therefore do not need to be instantiated to be utilized.
- This means that **a static method can be called without an object for that class**. I.e., you can invoke a static method without first instantiating the class!
- This also means that **static methods cannot modify the state of an object** as they are not bound to it.
- They also **cannot modify the class-state** as they do not take in the class as an input, rather they can simply take in external inputs
- They are typically thought of as **Utility-type methods**, that take some external parameters and work upon those parameters.
  - We embed them within the class because it is useful to organize them that way even if they aren't directly operating on class attributes
- Syntax
  - Much like class methods, the **staticmethod()** call can pull a class method and generate a static method
  - The **@staticmethod** decorator can also be utilized as a decorator to generate static methods



```python
class Calculator:

    def addNumbers(x, y):
        return x + y

# create addNumbers static method
Calculator.addNumbers = staticmethod(Calculator.addNumbers)

print('Product:', Calculator.addNumbers(15, 110))
```



```python
class Calculator:

    # create addNumbers static method
    @staticmethod
    def addNumbers(x, y):
        return x + y

print('Product:', Calculator.addNumbers(15, 110))
```

Summing it all up!

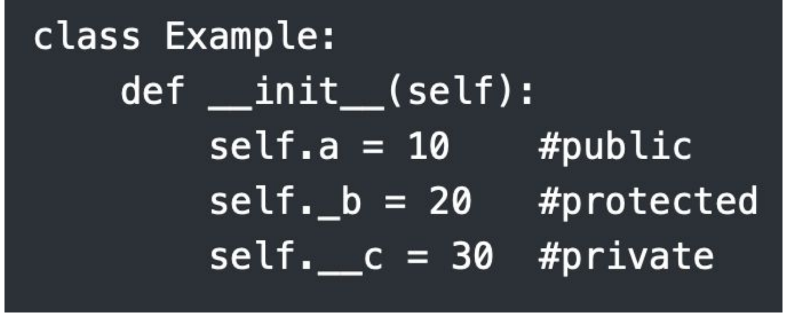
- Class Method vs Static Method
  - Both methods have a very clear use-case. When we need some functionality not w.r.t an Object but w.r.t the complete class, we make a method static. This is pretty much advantageous when we need to create Utility methods as they aren’t tied to an object lifecycle.
  - Also note that in a both methods, we don’t need the self to be passed as the first argument.
  - A **class method** takes **cls** as the first parameter while a **static method** needs **no specific parameters**.
  - A **class method can access or modify** the class state while a **static method can’t access or modify** it.
  - In general, **static methods know nothing** about the class state. They are **utility-type methods** that take some parameters and work upon those parameters.
  - On the other hand class methods must have class as a parameter.

Encapsulation

Encapsulation in Python

- **Encapsulation** is one of the fundamental concepts in object-oriented programming (OOP).
- It describes the idea of bundling data and the methods within one unit or **object**. This puts restrictions on accessing variables and methods directly and can prevent the **accidental modification of data**.
  - To prevent accidental change, an object’s variable can only be changed by an object’s method directly. Those types of variables are known as **private variables**.
- A class is an example of encapsulation as it encapsulates all the data and its member functions, variables, etc into a singular object.
  - The goal of information hiding is to ensure that an object’s state is always valid by controlling access to attributes that are hidden from the outside world.

Hidden you say...

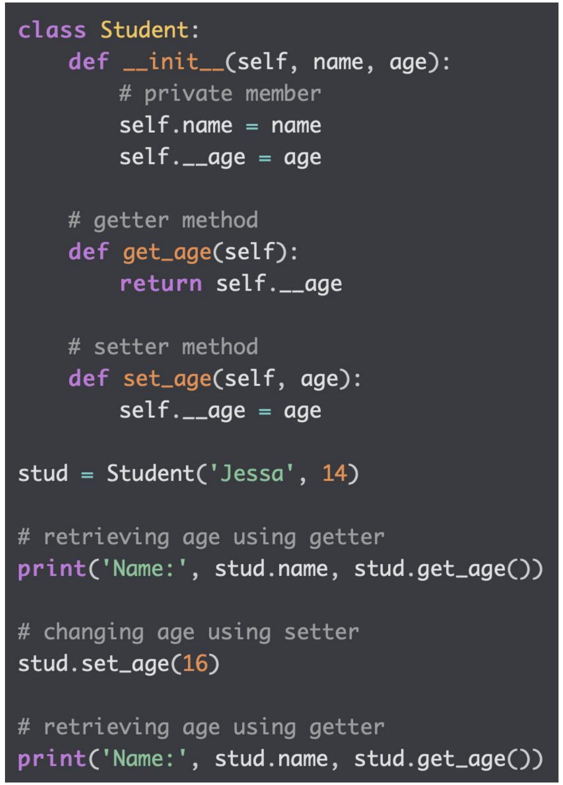


```
class Example:
    def __init__(self):
        self.a = 10    #public
        self._b = 20   #protected
        self.__c = 30  #private
```

- Before going any further with Encapsulation, we first need to talk a little about variable scope.
  - From a functional programming perspective we’re already familiar with variable scope and **Global** vs **Local** variables.
    - Global variables, defined outside the scope of a function, are accessible and potentially modifiable within the function
    - Local variables are defined within the scope of the function, and can only be modified within that function’s execution
      - Given that framework, where do OOP data elements, defined within a class fall?
        - Are those data attributes globally accessible, or restricted?
      - Turns out they can be either, and more!
- Data Access
  - Various object-oriented languages like C++, Java, and Python control access modifications used to restrict access to the variables and methods of the class. Most programming languages, Python included, maintain three forms of access modifiers to class data: **Public**, **Protected** and **Private**:
    - **Public** data are easily accessible, and modifiable from any part of the program.
      - All data attributes and method of a class are public by default.
    - **Private** attributes are accessible and modifiable **within the class only**. The private access modifier is the most secure access modifier.
      - Data members of a class are declared private by adding a double underscore ‘__’ symbol before the data member of that class.
    - **Protected** attributes are restricted outside of the class entirely, but accessible to other classes through inheritance. Protected attributes of a parent class are therefore only accessible to a child class derived from it.
      - Data members of a class are declared protected by adding a single underscore ‘_’ symbol before the data member of that class.

But if it's restricted, what good is it?

- So if I can't reference a private attribute what the heck am I supposed to do with it?
  - Turns out there is a way! I can sneak that information out by defining a class method (and remember, within the class I can access elements!) to output the attribute, and then call that method outside of the class.
    - And actually this isn't just a 'hack' this is kind of the point!
- The goal of restricting access to data within a program (i.e., performing proper encapsulation, is about control. We want to ensure that we're appropriately governing when and how attributes are accessed and modified. To govern these, we create custom class methods: **getters** and **setters**
  - **Getters** allow us to return a restricted attribute when we want to avoid direct access to private variables, or control the manner in which they're returned
  - **Setters** limit the ability to modify the attribute to only permissible options. This is done to add validation logic for setting a value
    - For instance in our bank ATM code, the balance attribute should be private, only allowing for withdrawal and deposits using our input validation methods
    - Similarly our account_holder attribute could also be private, and we could limit what is returned to simply the first few letters of the name, and restrict account name updates to specific admin accessible methods



```python
class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```