O O P (Object Oriented Progamming)

https://www.linkedin.com/in/salimtieb/

salim@iesleopoldoqueipo.com
https://github.com/profesiglo21/

Announcements & Agenda

- UT6. Fundamentos de La Programación Orientada a Objetos (Moodle Access)
- Building a Calculator in Python using PyQt6 library, software architecture MVC and POO programming approach.
    - All resources will be available at Github
    - Surprise Kahoot! Test due to a non-determined date.
- Final Assignment is due Friday 27/5/2024

Object Oriented Programming

Motivation & Brief Overview

- good for another layer of abstraction/organization
  - i.e. the Dog class
- you've already been working with classes/objects!
  - most explicitly: lists in python
  - think: `list.append('banana')`
- you create new *types* of data to work with new Classes

we use classes as a “template” to create objects of that type.

Objects

- is an “**instance**” (dynamic snapshot) of that class
- create an object instance of a class by ~instantiating~ the class with this syntax:
  - `annie = Human(“Annie”)`
- uses the class “template”, and adds “personal”/unique information that belongs **only to that object**
  - aka: only belongs to that Instance™ of the class

```
Class Human:
    def __init__(self, name):
        self.name = name
```

Terminology: Attributes

- **instance attribute**: property of an object, specific only to that particular ***instance*** of the class
- **class attribute**: property of an object, but shared by ***all*** instances of the class
- **classes do not have access to *instance* attributes,** but instances have access to *class* attributes

Terminology: Methods

```
class Human:
  def talk(self):
    print(“Hello”)
annie = Human()
annie.talk()
Human.talk(annie)
```

- **methods**: functions that belong to a particular class
  - methods are *invoked* on particular objects
  - methods *must* take in an **object** as a parameter (typically: “`self`”), in order to know what it will be invoked on — this is the distinction between **methods vs. functions**!
- **bound method**: when you bind a “self” to the method
  - implicitly: `annie.talk()`
  - explicitly: `Human.talk(annie)`

Terminology: self

- self refers to the object itself

```
class Human:
    word = "Class"
    def __init__(self, word):
        self.word = word
        self.mood = “tired”
    def talk(self):
        print(word) # !!
        print(self.word)
annie = Human(“Instance”)
```

class <Human>

```
word: "Class"
__init__(self, word)
talk(self)
```

annie = Human(“instance”)

```
self.word: "Instance"
self.mood: "Tired"
```

Dot Notation & Lookup

```
class Human:
  name = “Human”
  def talk(self):
    print(“Hello”)
annie = Human()
annie.talk()
annie.name
Human.name
```

- **left side of dot**: the Class, or an Instance of the class (object)
- **right side of dot**: an attribute, or method
  - find the correct one through *lookup*
- attribute lookup is similar to what we saw before
  - look in object’s **“personal” / instance attributes** first
  - if you can’t find it, look at the **class attributes**
- methods are always defined *in the class*
  - look in the **class**

What's gonna happen?

goo.gl/paSxqF

- annie can't talk — why?
- when using the dot notation, we are *invoking* the method on the object
- we will **always** implicitly pass in the object as a parameter, when calling methods in this way
- but talk doesn't take in any arguments — therefore:
  - expected 0, but got 1

```
class Human:
    word = "Class"
    def __init__(self, word):
        self.word = word
    def talk():
        print(word)
annie = Human("Instance")
annie.talk()
```

do this on ur own time !!

What happens pt 2

goo.gl/P3HBHS

- Human.word => “Class”
- self.word => “Instance”
- word => undefined (local var)

```
class Human:
    word = "Class"
    def __init__(self, word):
        self.word = word
    def talk(self):
        print(Human.word)
        print(self.word)
        print(word)
annie = Human("Instance")
annie.talk()
```

do this on ur own time !!

What happens pt 3

goo.gl/wwg1mR

- first `.talk()` will print Class
- second `.talk()` will error, why?

- instances use **bound methods**,
  - **.talk** (aka lambda) expects one param, but gets none
  - it's not a bound method that implicitly passes in the object as 'self'

```
class Human:
    word = "Class"
    def __init__(self, word):
        self.word = word
    def talk(self):
        print("Class")
annie = Human("Instance")
annie.talk()
annie.talk = lambda self: print("Lambda")
annie.talk()
```

do this on ur own time !!

What happens pt 4

goo.gl/YBCNys

- wanted to make the distinction that having a lambda is totally ok in a class (a lil weird, but viable)
- because it's defined *inside* the class, this lambda function is a **method** that will implicitly or explicitly take in a "self"
- food for thought: does the param have to be named "self"?

  more food for thought: does your answer apply to just lambdas, or to any method?

```
class Human:
    word = "Class"
    def __init__(self, word):
        self.word = word
    def talk(self):
        print("Class")
    lamb = lambda self: print("lambda")
annie = Human("Instance")
annie.talk()
annie.lamb()
```

Inheritance (Herencia)

Subclasses

- subclasses **inherit** all its parent/base class' properties
  - you can access all the parent's attributes + methods
  - kinda like "parent frame"
  - you can also "override" Parent attributes/methods, by defining ones with the same name in Child class definition
- when do you subclass?
- for **is-a** relationships

why would homework need to sleep eat cry talk, like students?

```
class Human
class Student(Human) # Student is-a Human. this makes sense!
class Homework(Student) # Homeworks ≠ Students. nonsensical!
```

Dot Notation & Lookup

- **left side of dot**: the Class, or an Instance of the class (object)
- **right side of dot**: an attribute, or method
  - find the correct one through *lookup*

- attribute lookup is similar to what we saw before
  - look in object's **"personal" / instance attributes** first
  - if you can't find it, look at the **class attributes**
  - check the **parent class**, if not in the object's immediate class
- methods are always defined *in the class*
  - look in the **class**
  - if you can't find it, **look in parent class**

Tank you!! :)