Welcome!

COMP 3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 2: REVISITING OBJECT ORIENTED PROGRAMMING

Graded Labs Start This Week!

- Labs are due Friday 11:59pm of the week that they’re assigned. I will not look at any commits that have been made past midnight.
- No extensions
- Labs are assessed and count for a total of 10% of your final grade!
- Absences must be justified with a doctor’s note, otherwise you will earn zero

Agenda

1. Datetime
2. Essential Object Oriented Programming.
   (All about Classes & Objects)
3. The core tenets of Object Oriented Programming
   1. Abstraction and Modularization
   2. Polymorphism
   3. Encapsulation
   4. Inheritance

DATETIME

An interesting idea

Running programs while you’re at your computer is fine

But it’s also useful for programs to execute without our direct supervisions

We can use our computer’s clock to run code:

- At some specified time and date
- At regular intervals

We can also write programs that launch other programs on a schedule by using the subprocess and threading modules…

Let’s start with the time module

Your system’s clock is set to a specific date, time, and time zone

The **time** module gives us access to the current time

We will look at two functions:

1. time.time( )
2. time.sleep( )

Everything is predicated on the Unix epoch

https://docs.python.org/3.7/library/time.html

Unix epoch

Time reference used in programming

The unix epoch is the "*point where time starts*"

Everything begins on January 01 1970 at 00:00:00, Coordinated Universal Time (UTC)

We count the number of seconds since this time and call it **seconds since the epoch**

time.time( )

```
import time
time.time( )
```

It returns the number of seconds that have elapsed “since time began”

It’s a float (very precise!)

We can use this to profile* code!

* Profile: measure how long a piece of code takes to run

Example

```
import time


product = 1
start_time = time.time()
for i in range(1, 100000):
    product = product * i
end_time = time.time()
print('The result is %s digits long.' % (len(str(product))))
print('Took %s seconds to calculate.' % (end_time - start_time))
```

time.sleep(seconds)

We can pause our code for a specified number of seconds (can be a float for precision)

The time.sleep(seconds) function will block until at least the specified number of seconds has elapsed

* Block: will not return and release your program to execute other code

Example

```
import time
for i in range(3):
    print(‘Tick’)
    time.sleep(1)
    print(‘Tock’)
    time.sleep(1)
```

The datetime module

The time module is good for getting a Unix epoch timestamp to work with

What if we want a date in a more convenient format?

Use the ***datetime module***

Lets us manipulate dates and times

The datetime module contains a new datatype called datetime

a datetime *represents a specific moment in time*

datetime object

The datetime data type in the datetime module can be used to create datetime objects

A datetime object uses integers to store:

1. Year
2. Month
3. Day
4. Hour
5. Minute
6. Second

Converting time types

We can convert a Unix epoch timestamp to a datetime object using the datetime module’s fromtimestamp( ) function

The date and time of the datetime will be converted to the local time zone

```
print(datetime.datetime.fromtimestamp(1000000)) #1970-
01-12 05:46:40

# need to import time

print(datetime.datetime.fromtimestamp(time.time()))
#2020-01-12 16:25:04.843986 Whatever the time is at the
moment you ran the code
```

Comparing times

datetime objects can be compared using the comparison operators

The later datetime object is the “greater” value

```
halloween = datetime.datetime(2018, 10, 31)
nyd = datetime.datetime(2019, 1, 1)
oct_31 = datetime.datetime(2018, 10, 31)
print(halloween == oct_31) True
print(halloween > nyd) #False
print (nyd != oct_31) #True
```

The timedelta datatype

datetime represents a point in time

timedelta represents a duration of time

We can create a timedelta that stores:

1. Weeks
2. Days
3. Hours
4. Minutes
5. Seconds
6. Milliseconds
7. Microseconds

timedelta example

```
halloween = datetime.datetime(2018, 10, 31)
nyd = datetime.datetime(2019, 1, 1)
delta = nyd - halloween
print(delta) #62 days, 0:00:00


delta2 = datetime.timedelta(days = 5, minutes = 11, seconds = 7)
print(delta2) #5 days, 0:11:07
```

https://docs.python.org/3.7/library/datetime.html#module-datetime

Hacking time

We can pause a program until a specific time or date:

```
import datetime

import time

final_exam = datetime.datetime(2020, 04, 10, 6)

while datetime.datetime.now() < final_exam:

    time.sleep(1)
```

https://docs.python.org/3.7/library/datetime.html#module-datetime

CLASSES AND OBJECTS

What?

What is Object Oriented Programming?

What are Classes?

What are Objects?

Object Oriented Programming

A programming paradigm that architects code as a system of interacting objects.

Lab 0 was procedural programming.

In 2522 you learned that we write classes that represent real-life things:

- **Define attributes** to store the state
- **Define behaviours** (write methods) that act on the state
- **Instantiate** the class and use it.

Why use classes and OOP?

Great modularization

- **Easy** to understand
- **Easy** to collaborate

Great for modeling real world problems

Encapsulates data and the functions that work on it

Adaptable and flexible use through **design idioms and patterns**

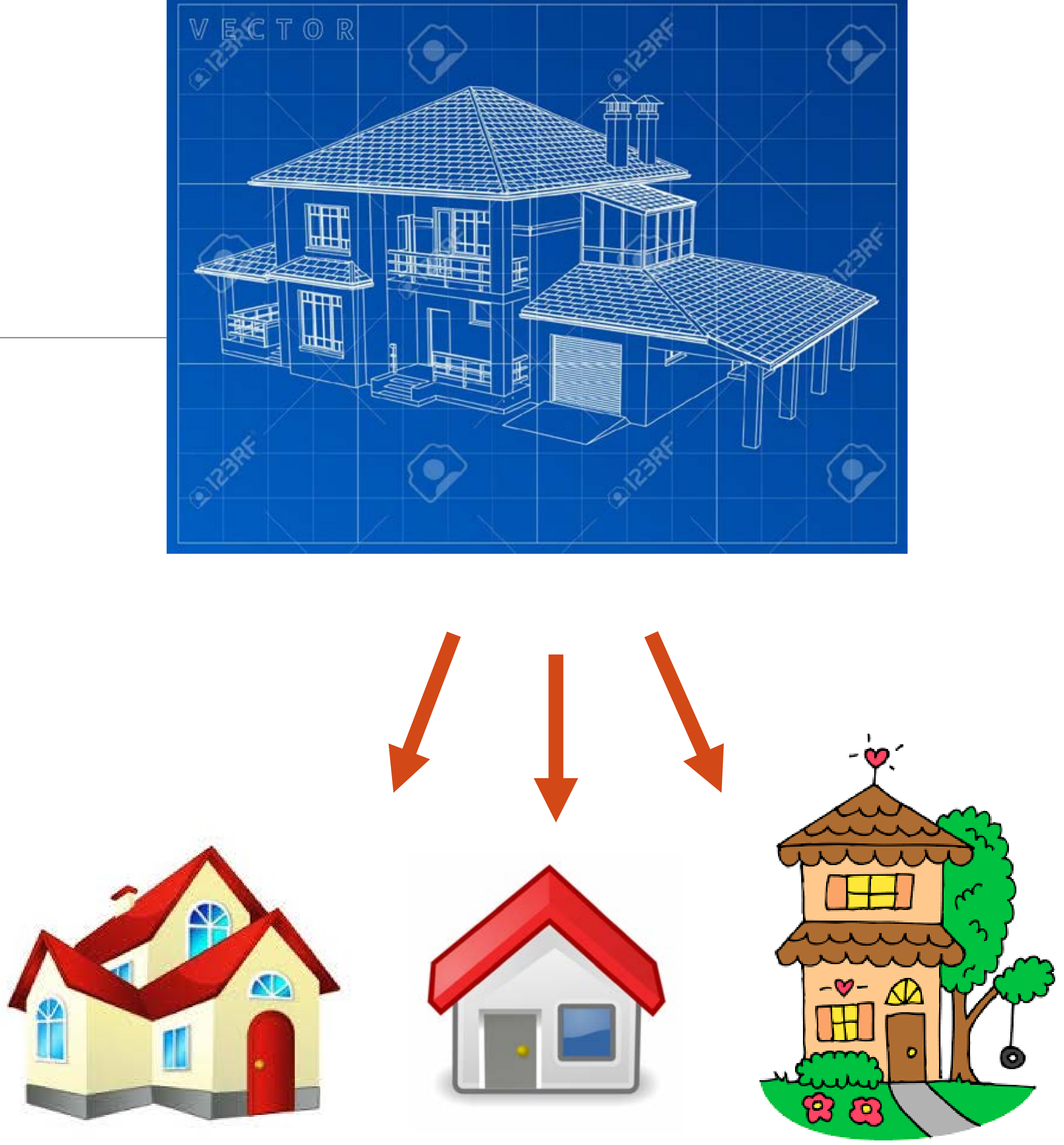
Compatible with modern software development approaches like Agile

Classes

A class is a blueprint, defining a Type. A class serves to create independent, distinct instances called objects in memory

A class defines all the **attributes** and **methods** that an instance of that class **(called an object)** will be composed of.

An object has a **state** and **behavior.** It is an encapsulation of data and methods that act on that data (and other data that it may receive from the system).



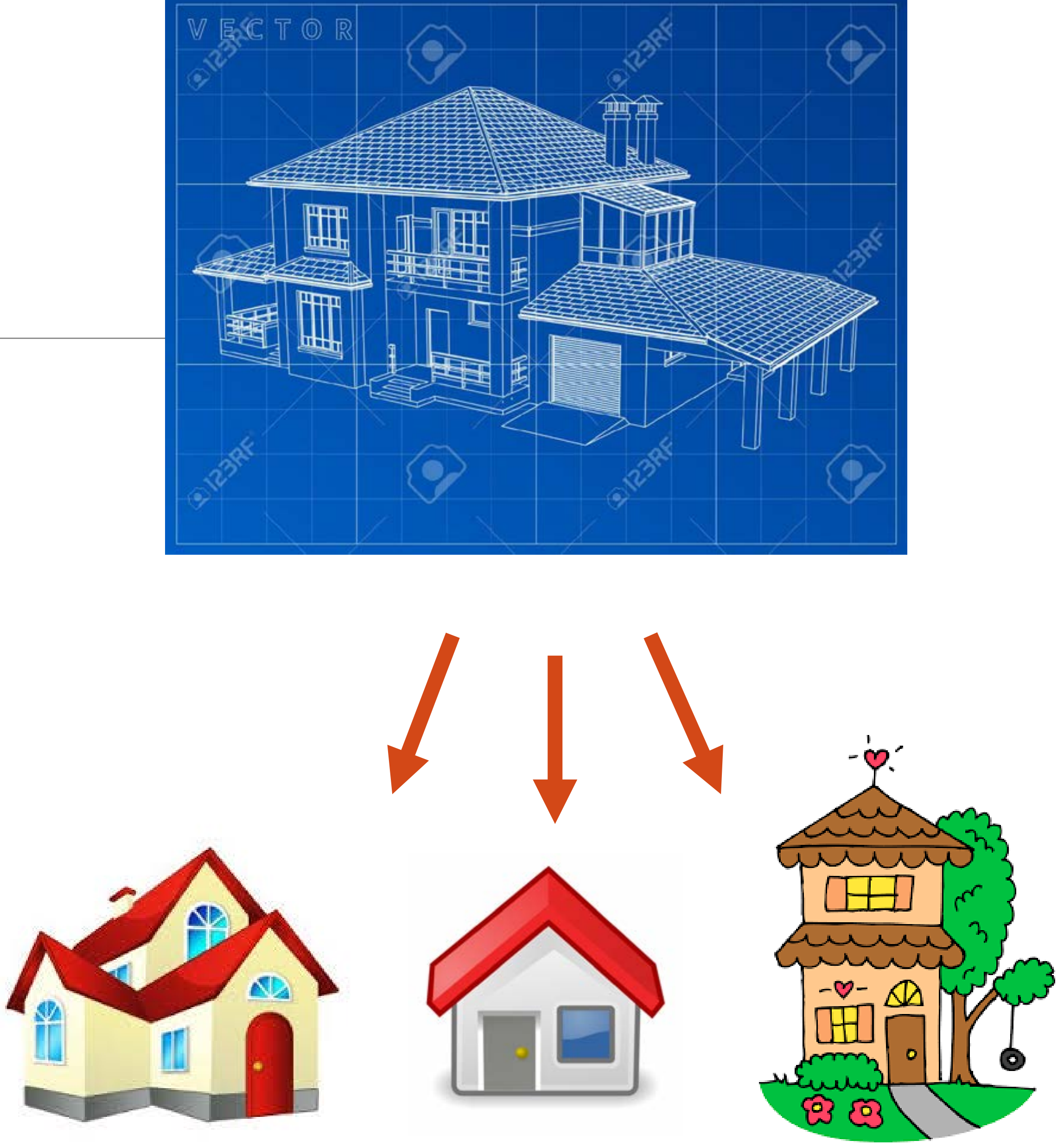
An Object Has A State

An object is an instance of a class with its own attributes (data).

The **state** of each object is the values of these variables at any given time

These variables are an object’s private property and last its lifetime

Mutating the state means we are changing the values in the instance variables



Let’s Create A Class

Methods vs Functions

Functionally speaking, they are the same.

Methods act on an object. Changing it’s state and working with its attributes

- First parameter is *‘self’ .* This is the object it is defined on and is passed automatically
- Accessed through an object

Functions are not object dependent

- Can be called without the assistance of objects
- Doesn’t accept self as a parameter

But they both ***do something*** and should be ***atomic***.

Special method #1: __init__(self)

__init__(self) method

a) Special built-in function called the ***initializer***

b) First method inside the class definition

c) Executed once when a new object is instantiated

d) Note the 2 leading underscores and the 2 trailing underscores *

e) Accepts one or more parameters:

   a) Self – every method call associated with a class automatically passes self, which is a reference to the instance itself.

   b) Data for the object's state

f) Note that the variables defined in the __init__ method are prefaced with self

Instance Variables?

In Python we often call them ***data attributes***!

We don’t define them at the top of the class, that’s a Java approach!

Python:

- Declare our variable and assign it a value inside the __init__(self) method
- Any variable prefaced with self in the __init__(self) method is available to every method in the class
- If it is not prefaced with self, it is just a local variable.

Out first class!

```
class Cat: #notice the class keyword
    def __init__(self, name, age, favorite_food):
        self._age = age
        self._name = name
        self._food_prefs = favorite_food


tobias = Cat("Tobias", 3, ["Sushi", "Raw fish"]) #instantiate cat
```

Notice the **self** parameter. All class methods need a self parameter, but it’s not explicitly passed as a parameter when instantiating/calling methods

Notice we didn’t define _age, _name, _food_prefs class variables above the init

Declaring them inside the __init__ is our way of declaring the class variables

What about visibility?

There are no access or visibility modifiers in Python

Everything in scope is always visible and transparent

- Private
- Package default
- Protected
- Everything is public always and forever.

And that works?

Yes!

Access modifiers are not Pythonic. Python provides unenforced guidelines and best practices and trusts you not to misuse this.

Python developers ***trust*** other programmers to stick to the rules and use good OOP approaches.

There Is A Standard!

Developers still need to communicate that it is not safe to access certain attributes in a class/object.

Single Underscore

```
class Cat:
    def __init__(self, age):
        self._age = age
```

A single underscore communicates to other developers that they should ideally not be touching this variable and modifying it is not recommended.

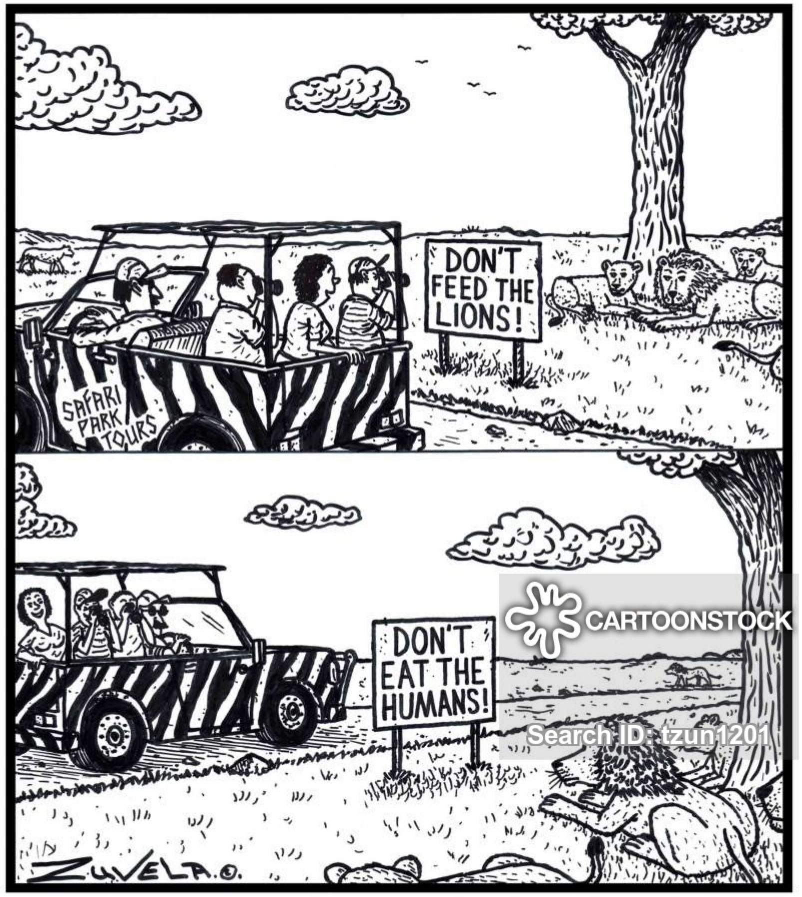
Double Underscores and Name Mangling

A variable name prefaced with double underscores (Dunder) is a way of saying "STAY AWAY".

Python makes it difficult to access this variable through **Name Mangling** , that means you can only access it by prefacing it with the class names and some more underscores.

Since it's a bother to access the variables developers are discouraged from detecting them and going near them.

```
class Cat:
    def __init__(self, age):
        self.__age = age
…
my_cat = Cat(5)
my_cat.age = 5 # ERROR NO NO NO
my_cat._Cat__age = 5 # Okay, but not good form
```



Special method #2: __repr__(self)

__repr__(self)

Is the same as Java's toString

Should contain unambiguous information about the current state of the object

Is implicitly called when we pass an object to the print( ) function

This is really for developers' eyes

Think of the information that should be in a log file

"name: Tobias age: 3 food_prefs: ['Sushi', 'Raw fish'], num_lives: 9"

* Remember to say dunder for double under

Special method #3: __str__(self)

__self__(self)

Is also the same as Java’s toString

Should be the same as the repr, except perhaps a little prettier (not a single line)

Not required

If not provided, __repr__(self) will always be used

“My name is Tobias and I'm 3 years old.

love eating ['Sushi', 'Raw fish']

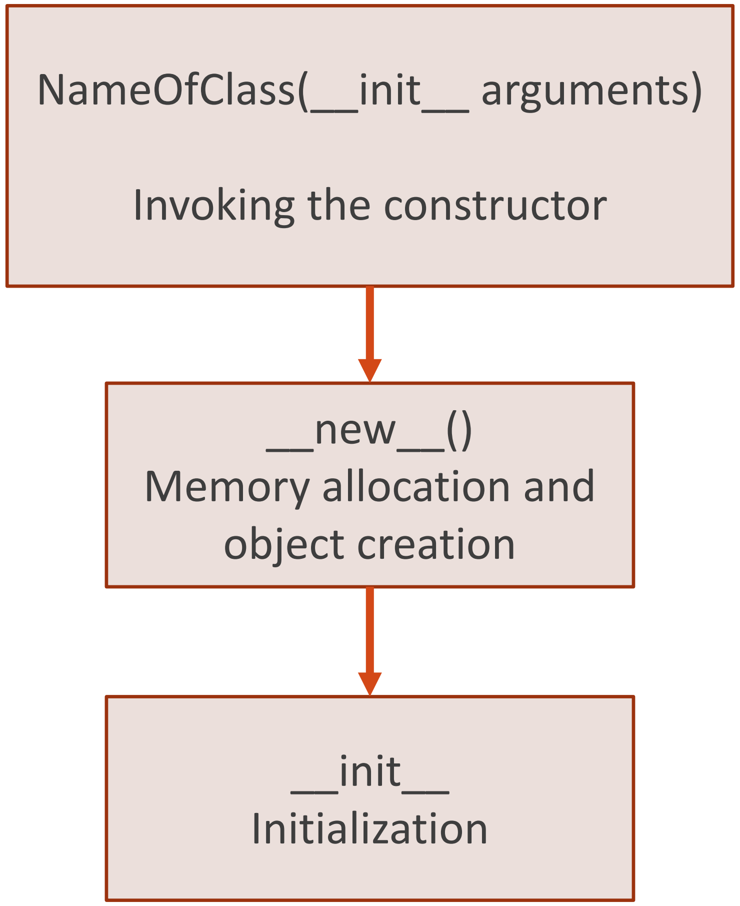
I have 9 lives remaining”

* Remember to say dunder for double under

Instantiation

Suppose we have a class called Die:

- Instantiate a Die by typing Die()
- This calls a special method called __new__(we never tamper with __new__ and we never call it directly)
- __new__ allocates memory and invokes the object initializer
- Die.__init__( ) is the initializer


NameOfClass(__init__ arguments)
Invoking the constructor
__new__()
Memory allocation and object creation
__init__
Initialization

Accessors & Mutators

Accessors: Methods that return the value of an attribute
Eg: get_player_health()

Mutators: Methods that set the value of a data attribute
Eg: set_player_strength(9000)

Self parameter is the:

- First parameter for all mutators
- Only parameter for all accessors

Require docstring comments

Must not generate side effects

- Accessors must not modify or print or send messages
- Mutators may only modify a single value, and only if it remains logical

Properties vs Accessors / Mutators

A property is a way to encapsulate the getter (Accessor) and setter (mutator) of an attribute.

Anyone interfacing the code with your library can still access the data attribute through a variable name.

The variable name automatically invokes the Accessor and Mutator.

Eg:

```
 # set inside class definition
temperature = property(get_temperature, set_temperature)

 ..

 ..

 my_engine.temperature = 70  # this would invoke set_temperature(70)
```

property_example.py

Static (Class) Variables

Java instances share static variables

Are there statics in Python?

Yes!

<u>Class variables</u> are ***<u>shared by all instances of a class</u>***

A class variable:

1. is declared inside the class but not inside a method
2. is accessible directly (remember no access variables!)
3. can be accessed using ClassName.class_variable

Class Methods

Class Methods are like Class Variables.

- They belong to a class rather than the object.
- They take *cls* as their first parameter instead of self
- *cls* stands for the class itself.
- *Can modify and access class variables but not instance variables*

While not quite like static functions, class methods can be accessed through the class name

```
MyClassName.ClassMethod(arguments)
```

Class Methods

For example, say we had a temperature class that stored temperature as farenheit. We could write a class method that allows us to create objects using celcius

```
class Temperature:
    def __init__(self, tempFarenheit):
        self.temperature = tempFarenheit

    @classmethod
    def from_celcius(cls, tempCelcius):
        tempFarenheit = (tempCelcius * 9/5) + 32
        return cls(tempFarenheit)

temp = Temperature.from_celcius(50)
```

Static Methods

The @staticmethod function decorator allows static functions to be defined.

- Static functions belong to and can be accessed by the class and not the object.
- Static functions do not require *cls* as its first parameter
- In fact, static functions can’t access the class variables/object at all. They behave as independent functions within the scope of a class.

Eg:

```
@staticmethod
def convert_to_fahrenheit(tempCelcius):
    celcius = (tempCelcius * 9/5) + 32
    return celcius
```

```
Temperature.convert_to_fahrenheit(100)
```

Static Methods vs Class Methods

Hold on, Static Methods and Class methods seem pretty similar, what’s the difference?

Similarities:

- Both are class level methods that are called on the class, NOT the instance
- No need to instantiate an instance of the class before calling the method

```
class Temperature:
       def __init__(self, tempFarenheit):
           self.temperature = tempFarenheit

       @classmethod
       def from_celcius(cls, tempCelcius):
           # class method code

       @staticmethod
       def convert_to_fahrenheit(tempCelcius):
           # static method code
```

```
# code is called on the class directly, not an instance
Temperature.convert_to_fahrenheit(100) #call static method
temp = Temperature.from_celcius(50) #call class method
```

Static Methods vs Class Methods

Hold on, Static Methods and Class methods seem pretty similar, what's the difference?

Differences:

- Static Methods – Purpose is to perform operation WITHOUT access to static class variables.
- use @staticmethod decorator. Do NOT use **cls** as first parameter.

```
@staticmethod
def convert_to_fahrenheit(tempCelcius):
    celcius = (tempCelcius * 9/5) + 32
    return celcius
```

- Class Methods – Purpose is to perform operation WITH access to static class
- Use @classmethod decorator. Use **cls** as first parameter. **cls** parameter automatically passed in

```
@classmethod
def from_celcius(cls, tempCelcius):
    tempFarenheit = (tempCelcius * 9/5) + 32
    return cls(tempFarenheit)
```

All the concepts together!

Let’s check out cat.py

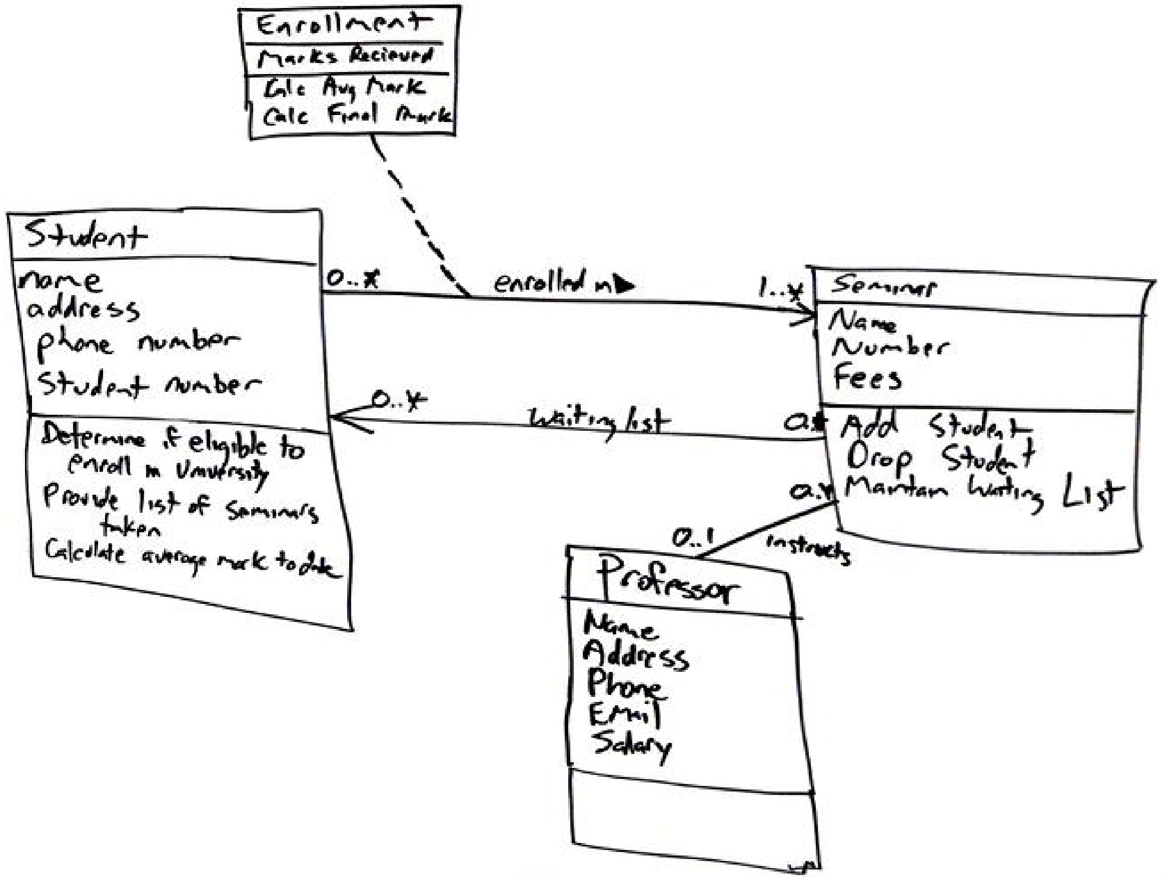
- Class declaration
- Instance/class variables
- Initializer (self)
- Instantiation
- Methods
- Static, class methods (cls)
- _ (‘private’ variables)
- __ (dunder)

Designing classes (and functions, too!)

Design before you code!

Every class needs to be well-defined:

1. Represents a single clear concept
2. Maintain information by storing data in instance variables
3. Perform actions by executing code in their methods and modifying the state of the program
4. Don't duplicate data
5. Don't store more than we need
6. Minimize "moving parts"



Designing classes (and functions, too!)

Be thoughtful about identifiers:

1. Think of names as documentation
2. **Invest time** in finding the most appropriate names
3. Use similar names for similar things, and different names for different things
4. Make names pronounceable
5. The source code should tell its own story
6. Public names should be understandable without knowing the internals
7. Don’t joke with names

OOP = Collaboration and composition

Remember that objects are (usually) team players

When some other object already has the data and logic for performing a task, we delegate the work to it

We say that the objects collaborate

An object will often store its collaborators as its data attributes, because it refers to them frequently throughout its lifetime

An object is **composed** of collaborators that it trusts with some of its specific responsibilities

OOP PRINCIPLES

What Makes OOP

Object-oriented programming (OOP) is a programming paradigm that **uses abstraction to create models** based on the real world

OOP uses several techniques from previously established paradigms, including:

1. Modularity and abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism.

Encapsulation (and information hiding)

Encapsulation:

- Putting data with the functions that manage it
- Wrapping them up in an impenetrable barrier of privacy (A class or a module)
- Making state change is reliable and predictable

Information hiding:

- Making a distinction between what is private (how a class' behaviours are implemented) and what is public (what you need to know in order to use the class correctly)
- Making the inner working inaccessible
- Exposing only the things that absolutely MUST be exposed

An encapsulated object can be thought of as a black box

The inner working are hidden from the client object

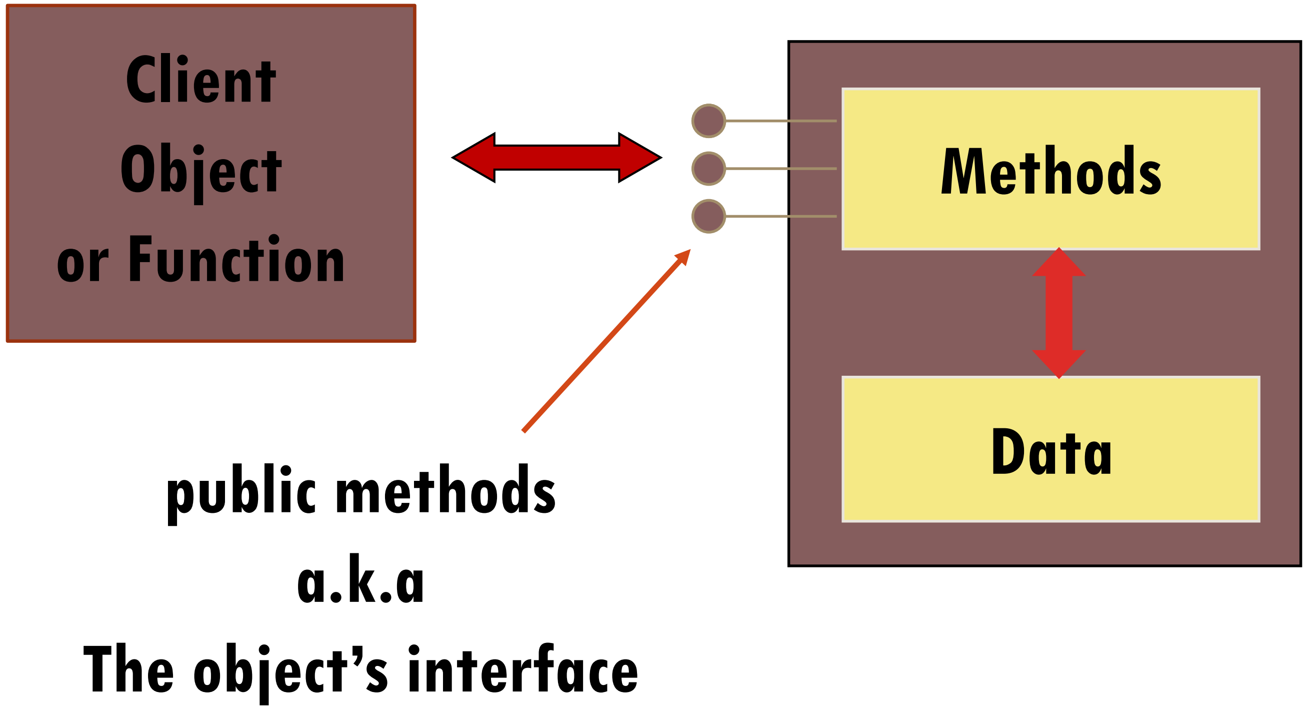
The client object invokes the methods of the object, which manage the instance data

Another related term is **Information Hiding**

It’s All About The Interface

The methods and data of one object visible to other objects and parts of the system is known as the ***public interface of that object***

Good OOP Design is centered around creating meaningful and maintainable interfaces.



Public vs private

We like to **expose a public interface**

The interface is how we interact with an object

But some things should be private!

We say the **implementation is private**

An object's data is conceptually private

How can we enforce privacy in Python?

1. Do not generate accessors and mutators for *every* instance variables
2. Only fields that are part of an object's public description should have accessor methods, and possibly mutator methods

Encapsulation in Python

We can "enforce" privacy with some *conventions.* Recall the _ and __ conventions!

In Python, we can **preface private function and instance variable identifiers with two underline characters**, i.e., __count

A **private method** is only intended to be used by other methods in a class

It is not intended to be visible or accessible outside the class

A **private variable** is only intended to be used inside the class, or the class methods

It is not intended to be visible or modifiable from outside the class

Encapsulation in Python

```
class Visibility:
    def public_func(self):
        print('public')

    def __private_func(self):
        print("private")

    def other_func(self):
        self.public_func()
        self.__private_func()

v = Visibility()
v.public_func() # public
v.other_func() # public private
v.__private_func() # ERROR 'Visibility' object has no attribute '__private'
```

Abstraction is another hot OOP buzzword

How do we use abstraction?

1. Divide a problem into sub-problems
2. Divide a sub-problem into sub-sub-problems
3. Keep doing this until the individual problems are small enough to solve
4. **Solve the small problems**
5. Ignore the details of the solutions and treat each one as a single encapsulated building block (abstract the details away)

Modularization

Abstraction uses modularization.

Modularization is the process of dividing something into well-defined parts .

Encapsulation is about Classes and Objects. **Modularization is about the system and the bigger picture**.

Each of these modules:

1. Is built separately
2. Encapsulates data with the methods that act on it. This could be functions and multiple classes.
3. Can be examined separately
4. Interacts in well-defined ways.

This is what building systems is all about! How do we decide when to split or maintain an ***'entity'***?

Polymorphism

Polymorphism is the ability of our system/object/functions/code to function differently and be context-aware

<u>Context-Aware:</u> When given the same interface, or command, your code behaves differently based on what kind of data and environment it's working on.

Eg:

5 + 4 = 9

"5" + "4" = "54"

The + operator exhibits polymorphism.

Duck Typing is one way to exhibit polymorphism.

Inheritance and polymorphism

The kernel of OOP

The **most important** part of OOP

Polymorphism **gives us absolute control over every source code dependency** in the system

It creates a plugin architecture in which modules that contain high level policies are separate from modules which contain low-level details

This lets us use abstraction and modularization to make our code units independently flexible, maintainable, and growable!

We'll see all this in action in the coming weeks. If this doesn't make sense, don't worry!

We'll come back to this and have a discussion around it towards the end of the semester

On Friday

- Ranges & Slices
- Scope (Python Fundamentals End)

- Inheritance
- Abstract Base Classes
- UML
- Lots of fun stuff!