Welcome!

COMP 3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 2: INHERITANCE AND UML DIAGRAMS

Agenda

1. Variables and Memory
2. Inheritance
3. Interface
4. Duck Typing
5. Protocols
6. Formal protocols and ABCs
7. UML

Review: Static vs Class methods

Static methods do NOT use cls, Class methods DO use cls

If we try to create a class method withOUT cls and run the program, there will be an error

TypeError: set_default_lives() takes 1 positional argument but 2 were given

MISSING CLS

```
@classmethod
def set_default_lives( num_lives):
     #code
```

Review: Static vs Class methods

Static methods do NOT use cls, Class methods DO use cls

If we try to create a static method with cls, and run the program, there will be an error. DON'T include cls for static methods!

TypeError: get_list_of_celebrity_cats() missing 1 required positional argument: 'cls'

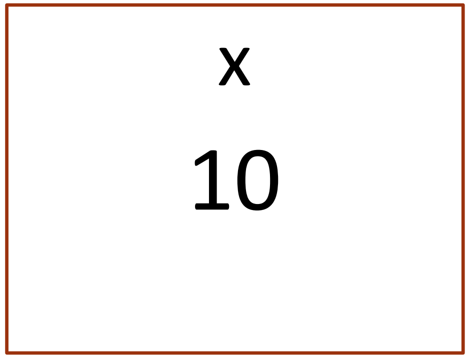
```
@staticmethod
def get_list_of_celebrity_cats(cls):
    return ["Tom", "Garfield", "Snagglepuss", "Cheshire Cat", "Cat in the Hat"]
```

cat.py

VARIABLES AND MEMORY

Java memory

int x = 10;



Can think of it like we created a box named x

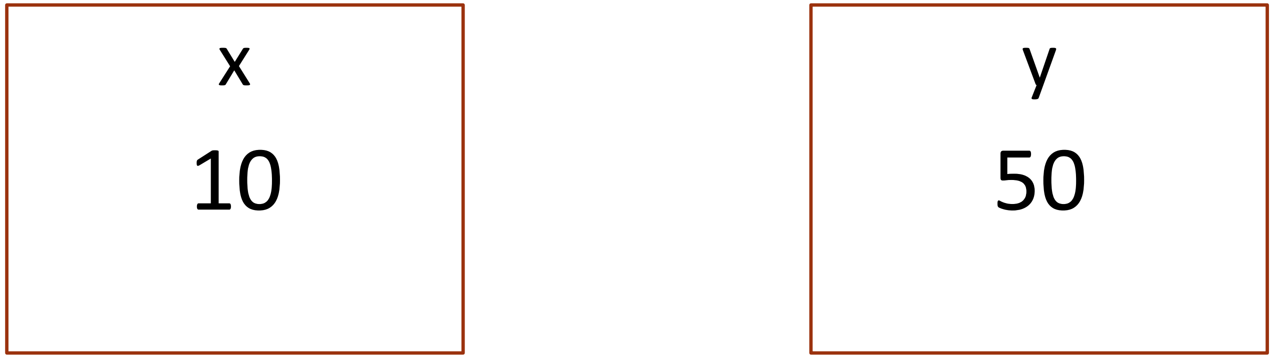
This box takes up a certain size in memory (maybe 4 bytes)

In this box we put a value in it (10)

Java memory

int x = 10;

int y = 50;



When creating a new variable y, a new box is allocated and the value 50 is placed inside that box

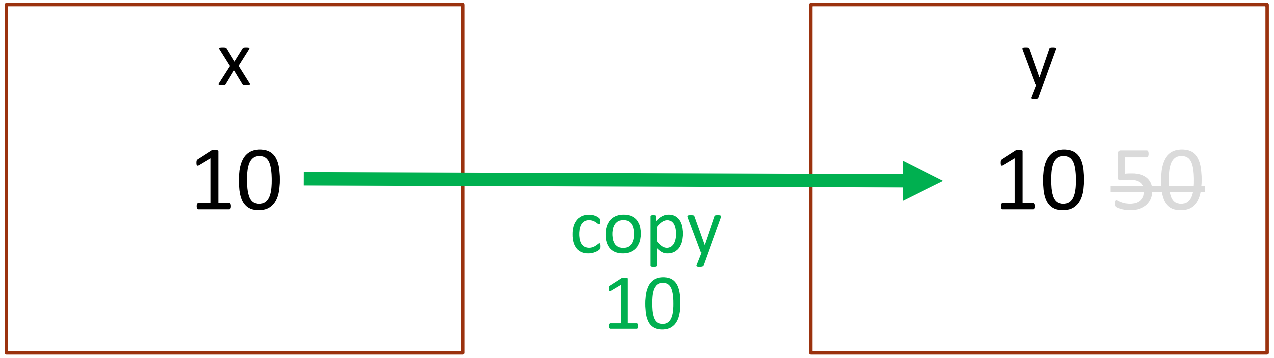
x and y are completely different entities

Java memory

```
int x = 10;

int y = 50;

y = x;
```



Here y = x means we copy the value in x, and place it in y

The copied value (10) replaces the previous value (50)

Java memory

```
int x = 10;

int y = 50;

y = x;

y = 99;
```

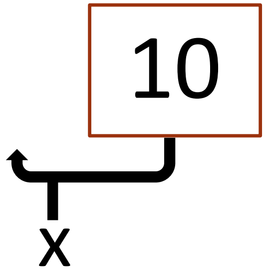
x	y
10	99 ~~10~~

When we change y to 99, it replaces the previous value

This does not affect x in any way

Python memory

x = 10;



Variables are more like tags instead of boxes

Memory is allocated for the value 10

Then we tag it with the variable x

Imagine a hook on a wall. When we create the value 10, we nail a hook on the wall with the value 10

Any variable that wants to access that value puts a tag on the hook

Python memory

x = 10;

y = 50;

When creating a new variable y, memory is allocated for the value 50, and we tag it with y

Python memory

x = 10;

y = 50;

y = x;

Here y = x means we want to move the y tag over to where x is. NO COPY

Both x and y are tagging the same value

This is why printing the memory of x and y shows the same memory address print(id(x)) == print(id(y))

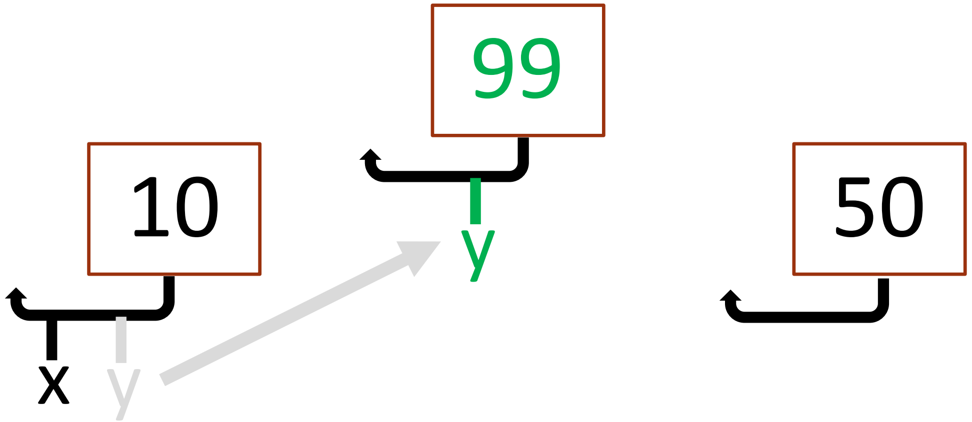
Python memory

x = 10;

y = 50;

y = x;

y = 99;



When we change y to 99 a new value is created in memory and the y tag moves there

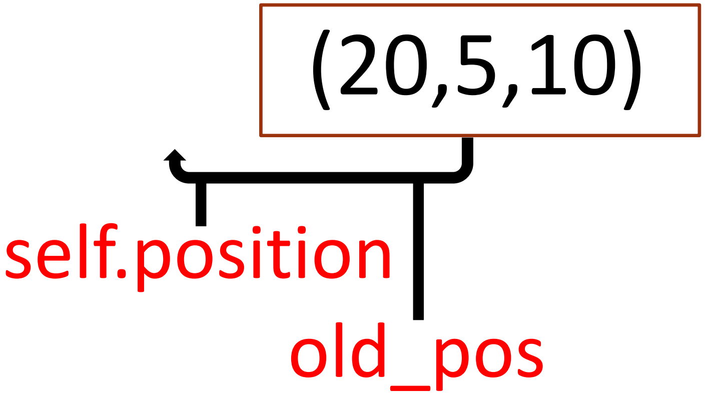
Notice how the value 50 is still in memory. This happens until the garbage collector realizes there are no variables associated with it. Then the memory is removed

Python memory - problem

```python
def move(self):
    old_pos = self.position

    self.position.x += self.velocity.x
    self.position.y += self.velocity.y
    self.position.z += self.velocity.z

    print(f"Old Pos: {old_pos} -> New
    Pos: {self._position}")
```



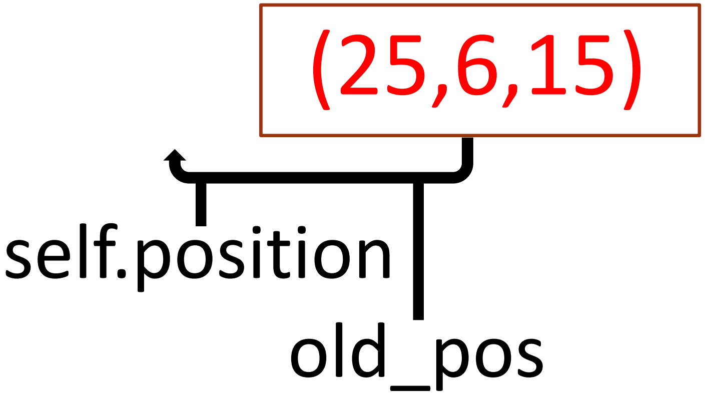
old_pos tagging same object as self.position. Any changes made to self.position also reflected in old_pos

Python memory - problem

```
def move(self):
    old_pos = self.position

    self.position.x += self.velocity.x
    self.position.y += self.velocity.y
    self.position.z += self.velocity.z

    print(f"Old Pos: {old_pos} -> New
    Pos: {self._position}")
```



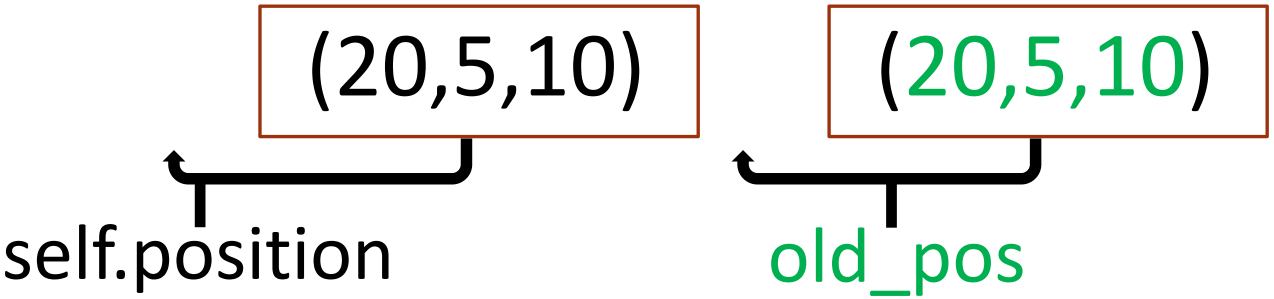
After executing the addition code, old_pos has the same value as self.position

Python memory - solution

```python
def move(self):
    old_pos = Vector(self.position.x,
                     self.position.y,
                     self.position.z)

    self.position.x += self.velocity.x
    self.position.y += self.velocity.y
    self.position.z += self.velocity.z

    print(f"Old Pos: {old_pos} -> New
    Pos: {self._position}")
```



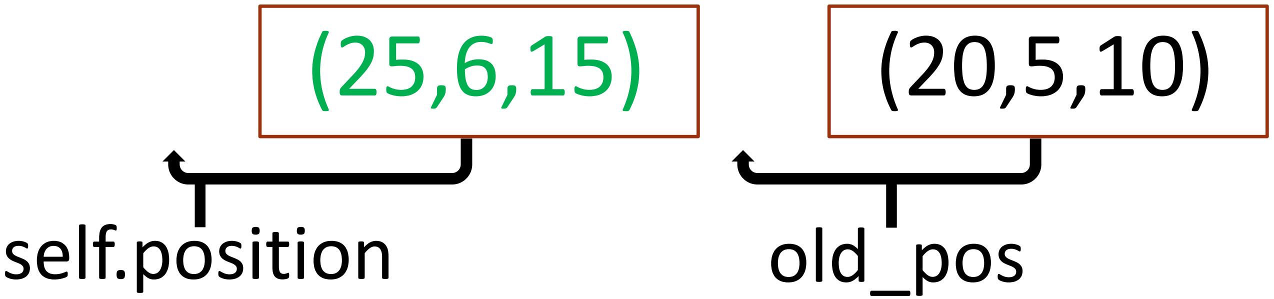
old_pos tagged to new Vector object created from self.position

Python memory - solution

```
def move(self):
    old_pos = Vector(self.position.x,
                     self.position.y,
                     self.position.z)

    self.position.x += self.velocity.x
    self.position.y += self.velocity.y
    self.position.z += self.velocity.z

    print(f"Old Pos: {old_pos} -> New
    Pos: {self._position}")
```



After executing addition code, old_pos remains the same because it’s a different object. self.position changes values due to the addition

INHERITANCE

Inheritance in Python

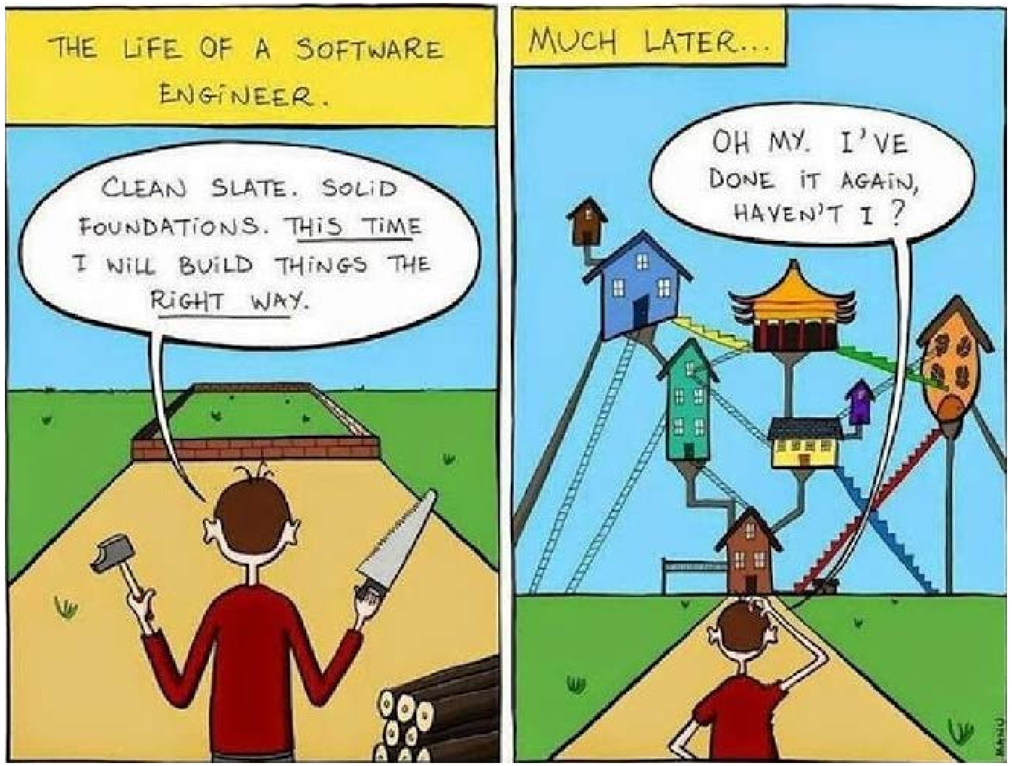
So easy

Almost the same approach as Java:

- Push common attributes as far up the inheritance hierarchy as possible
- Represent an is-a relationship
- Subclass (child class) is a specialized version of the superclass (parent class)

With one big difference:

- Multiple inheritance (oh the horror!)



Basic inheritance in Python

We specify the superclass as a parameter to the class definition:

```
class Subclass(Superclass):
    # class definition here
```

Basic inheritance in Python

Classes in Python implicitly extend the object class (just like Java)

```
class MySubClass:
    pass

#The above and below class declarations are equivalent

class MySubClass(object):
    pass
```

What can we extend

Almost anything!

Even the basic data types

1. string
2. int
3. float
4. List
5. All of them are classes

Overriding and super( )

We can extend behaviour by adding new methods

We can also override existing methods like __init__( )

We can access the original using the super( ) function

super( ) returns the object as an instance of the parent class

This lets us call the parent method directly

We can call super( ) from inside any method

You should be thinking of order of initialization at this point…

Order of initialization

Remember in COMP 2522 we were very interested in the order of initialization

Suppose we have a simple inheritance hierarchy

In which order are the __init__ methods called? What's the output?

```
class Animal:
    def __init__(self):
        print('Animal')

class Dog(Animal):
    def __init__(self):
        print(‘Dog')

a = Dog()
```

```
Output: Dog
```

Order of initialization

In which order are static variables initialized? What’s the output?

```
class Animal:
    print('A')
    animal_static_var = 5
    print('B')
    def __init__(self):
        print('Animal')

class Dog(Animal):
    print('C')
    dog_static_var = 10
    print('D')
    def __init__(self):
        print('Dog')

a = Dog()
```

Output:

A

B

C

D

Dog

Order of initialization

What about calls to super( )? When do they take place? What's the output?

```
class Animal:
    def __init__(self):
        print('Animal')

class Dog(Animal):
    def __init__(self):
        super().__init__()
        print('Dog')

a = Dog()
```

Output:

Animal

Dog

Order of initialization

What about calls to super( )? When do they take place? What’s the output?

```
class Animal:
    def __init__(self):
        print('Animal')

class Dog(Animal):
    def __init__(self):
        print('Dog')
        super().__init__()

a = Dog()
```

Output:

Dog

Animal

Order of initialization

Is a call to super( ) implicit? Or do we have to make it **explicitly** in Python?

Must have explicit super()

Let’s see this in action.

Enemy

enemy_inheritance.py

INTERFACES

The Java interface

Recall the Java interface is a set of methods and constants

It defines how to interact with a “kind” of thing

The concept of an interface exists in Python, too

There are two kinds:

1. Informal interfaces
   a) Duck typing
   b) Protocols
2. Formal interfaces
   a) Abstract base classes

DUCK TYPING

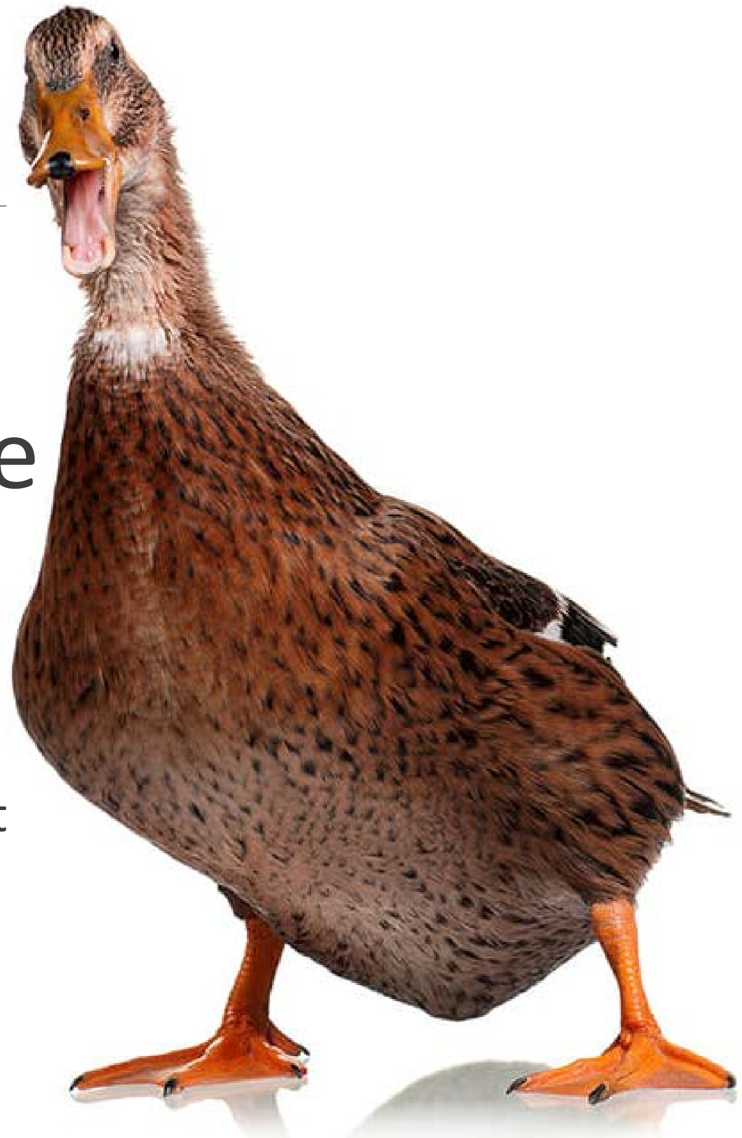
Duck typing

We say that Python uses **DUCK TYPING:**

"If a bird walks, swims, and quacks like a duck, then call it a duck”

For example, if an object can be concatenated, indexed, and converted to ASCII, that is, if it does everything a string can do, then let’s treat the object like a string.

Duck typing is when an object's suitability for typing takes place at runtime



Python uses dynamic typing

A Python programmer can use any type of object as an argument to a function

Consider a function add(x, y) that adds the two parameters

```
def add(a, b):
    return a + b
```

We can invoke this function using ints or using strings!

```
print(add(1, 1))
print(add('nanoo', 'nanoo'))
```

More dynamic typing

Python uses **dynamic typing** to determine the type of objects during runtime

For example, the consecutive statements **num = 5** and **num = '7'** first assign num to an integer type, and then a string type

The type of num can change, depending on the value it references

<u>The interpreter is responsible for checking that all operations are valid as the program executes</u>

If the function call add(5, '100') is evaluated, the interpreter generates an error when we try to add the string to an integer.

Static typing

In contrast to dynamic typing, many other languages like C, C++, and Java (of course) use **static typing**

Static typing requires the programmer to define the type of every variable and every function parameter in a program's source code

For example `int answerToEverything = 42`; to declare an int variable

When the source code is compiled, the compiler attempts to detect non type-safe operations, and halts the compilation process if such an operation is found.

Which is better?

Dynamic typing typically gives the programmer more flexibility than static typing

Dynamic typing can potentially introduce more bugs, because there is no compilation process that checks types

We say that Python is strongly typed

The interpreter keeps track of object types

We are not allowed to do anything that is incompatible with the type of data we are working with

We cannot perform operations inappropriate for the type of the object

Q: How can we prove this?

A: Try to add a number to a string!

(We can do this in weakly typed languages like JavaScript, but not Python)

What does this mean?

Suppose you have a function that accepts an object

It expects the object to have a method called list

We don’t have to specify what type the object is

The code will compile and execute as long as the object, whatever it is, has a method called list

This is great!

This is runtime polymorphism! This is fabulous advanced runtime polymorphism!

“We don’t care what kind of object you are, as long as you have a list( )!”

polymorphism.py, polymorphism_2.py

PROTOCOLS

The Python protocol

The lack of control doesn't mean there is no organization

Protocols are collections of methods that Python developers and libraries expect certain "kinds" of things to have

This is analogous to interfaces in the Java Collections Framework

There are some very helpful protocols in Python

In order to implement a protocol, a type must support certain operations

The Sized protocol

Implemented by having a method called __len__(self)

Invoked when an instance is passed to the built-in function len( )

This is how an object reports the correct value to the len( ) function

We may assume Python's built-in len( ) function works something like this:

```
def len(obj):
    return obj.__len__( )
```

__len__(self) returns the length of the object, an integer >= 0

Implemented by str, list, range, tuple, bytes, set, dict

The Container protocol

Provides the method __contains__(self, item)

When the membership operator **in** or **not in** is invoked on an object, this is the method invoked

This is how a class implements the membership test operator

__contains__(self, item) must return true if the item is **in** self, else false

Implemented by str, list, range, tuple, bytes, set,dict

The Sequence protocol

Permits the following:

1. Retrieving an item by index, i.e., item = seq[index]
2. Finding an item by value, i.e., index = seq.index(item)
3. Counting items, i.e., num = seq.count(item)
4. Producing a reverse iterator, i.e., r = reversed(seq)

Must implement

1. count(item)
2. index(item)
3. __len__( )
4. __getitem__(self, key)
5. __contains__(self, item)
6. __reversed__(self)

Implemented by str, list, range, tuple, bytes

And many more!

For example:

- An Iterator must provide __next__(self) and __iter__(self)
- A Collection must provide __len__(self), __iter__(self), and __contains__(self, item)

Check out https://docs.python.org/3/library/typing.html#classes-functions-and-decorators

Aside: so many dunders (__)

There are many methods in Python that begin and end with dunder

Sometimes we call these magic methods

These are usually the methods we need to implement in order to conform to protocols

We can also overload operators with these (more about this mind-blowing exercise later in the term)

Structural v nominal typing

Let’s think of objects

All objects have types. int, string, float, my_object, etc

When comparing two objects, how do we determine if they’re the same type?

- Are two types the same when they have the same name?
- Are two types the same when their internal structure is the same?

Nominal typing means the **name** is how we determine if they’re the same (Java, C#, C++ etc)

- “**MyClass** x and **MyClass** y are the same type because they use the name **MyClass**”

Structural typing means the **internal structure** is how we determine if they’re the same

- “**MyClass** x and **MyClass** y are the same type because they both have the same instance variables, methods etc”

Structural v nominal typing

Check out this pseudocode:

Class **2DPoint**: { int x, int y }

Class **XYObject** { int x, int y }

With Nominal Typing (Java), 2DPoint and XYObject are two different types because their **names are different**.

Although structurally they're identical, they both have x and y instance variables

Structural v nominal typing

Check out this pseudocode:

Class **2DPoint**: { int x, int y }

Class **XYObject** { int x, int y }

```
void sum (2DPoint point)
  return point.x + point.y
```

With Nominal typing (Java), we can’t pass an object of **XYObject** type into the sum function because the parameter type names don’t match

Even though structurally XYObject has the x and y variables to work with sum function

Structural v nominal typing

Check out this pseudocode:

Class **2DPoint**: { int x, int y }

Class **XYObject** { int x, int y }

```
void sum (point)
  return point.x + point.y
```

Compare that to structural typing (Python). **2DPoint** and **XYObject** are structurally equivalent

As a result, the sum function can work with both 2DPoint and XYObject types

Structural v nominal subtyping

In Java we used nominal subtyping

Nominal subtyping is based on the class hierarchy, i.e., if B extends A, then B is-a A

- We're explicitly saying the name B extends the name A

In Python duck typing goes hand in hand with structural typing, i.e., class B is a structural subtype of A if:

- B has the same attributes and methods as Q (with compatible types)
- B can be used instead of A without throwing any errors

Structural typing is flexible and adaptable

But there are situations where information interfaces or duck typing can cause confusing

FORMAL PROTOCOLS AND ABCS

There are some formal protocols

Python provides abstract base classes (ABCs)

The ABC complements duck-typing

ABCs define a set of methods and properties that a class must implement in order to be considered a duck-type instance of that class

Python provides built-in ABCs for data structures, numbers, streams, etc.

We can also create our own ABCs with the abc module

https://docs.python.org/3/library/abc.html#module-abc

Example

```
import abc

class Bird(abc.ABC):

    @abc.abstractmethod
    def fly(self):
        pass
```

Abstract methods tagged and mandatory

We decorate abstract methods with @abc.abstractmethod

If an abstract method is not implemented by a subclass, Python will generate a TypeError: Can't instantiate abstract class X with abstract methods Y

An implementing class is an instance of the ABC

```
class Parrot(Bird):
    def fly(self):
        print("Flying")
p = Parrot()
>>> isinstance(p, Bird)
True
```

Virtual subclasses

We can also register a class as a virtual subclass

The class will be treated as a subclass of the ABC

```
@Bird.register
class Robin:
    pass
r = Robin()
>>> issubclass(Robin, Bird)
True
>>> isinstance(r, Bird)
True
```

This is not commonly used. Can be used to make third-party package a subclass of our own abstract classes

- abstract_class.register(third_party_class)

You can read the rationale for its including in Python here: https://www.python.org/dev/peps/pep-3119/#overloading-isinstance-and-issubclass

UML REVIEW

We will use UML this term

Modelling language consisting of set of diagrams

Used to help developers to specify, visualize, construct, and document software system

Let’s review class diagrams

Class diagram

Purpose is to describe the static structure of the system

Shows the classes

- Attributes
- Operations

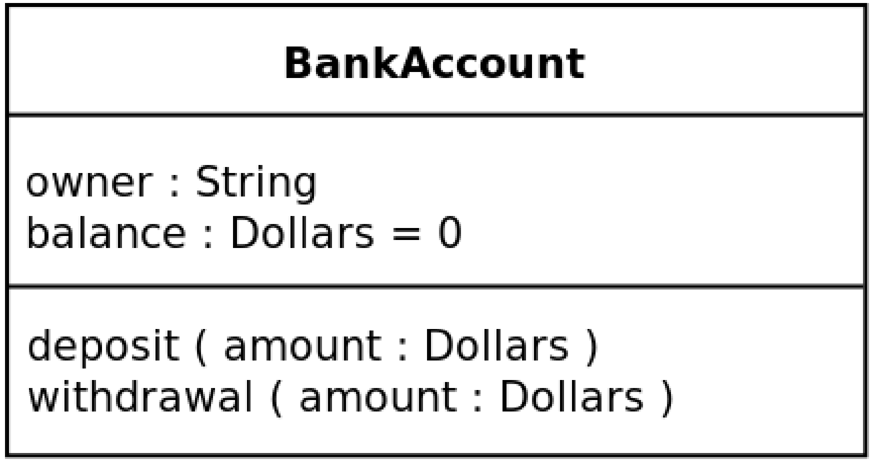
Relationships among objects

Language agnostic

Class diagram

Rectangle indicate a class

- Top section is the name of the class
- Middle section are the attributes and types
- Bottom section are the methods



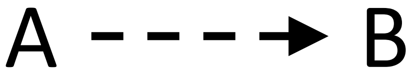
Class diagram - Relationships

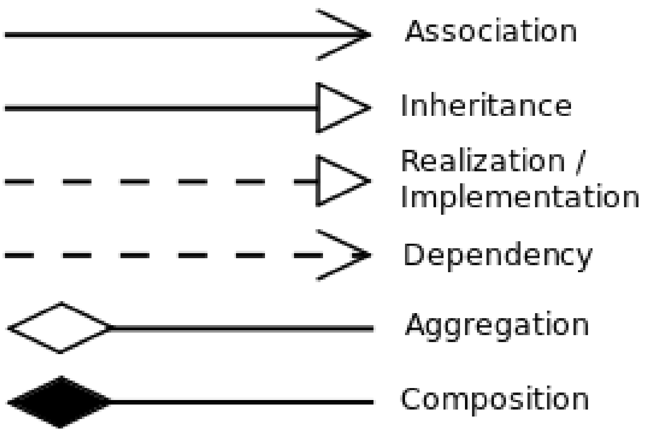
Arrows indicating how two classes are related

Association – A contains instance variables B

Inheritance – A is a subclass of B

Realization/Implementation – A is implementing interface B





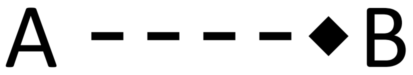
Class diagram - Relationships

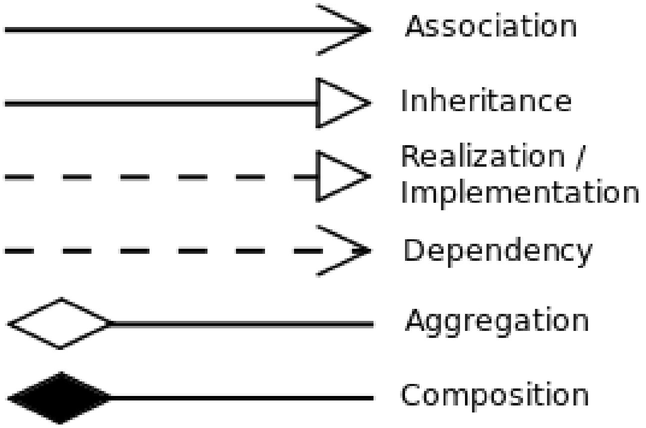
Arrows indicating how two classes are related

Dependency – A contains methods that use parameter B

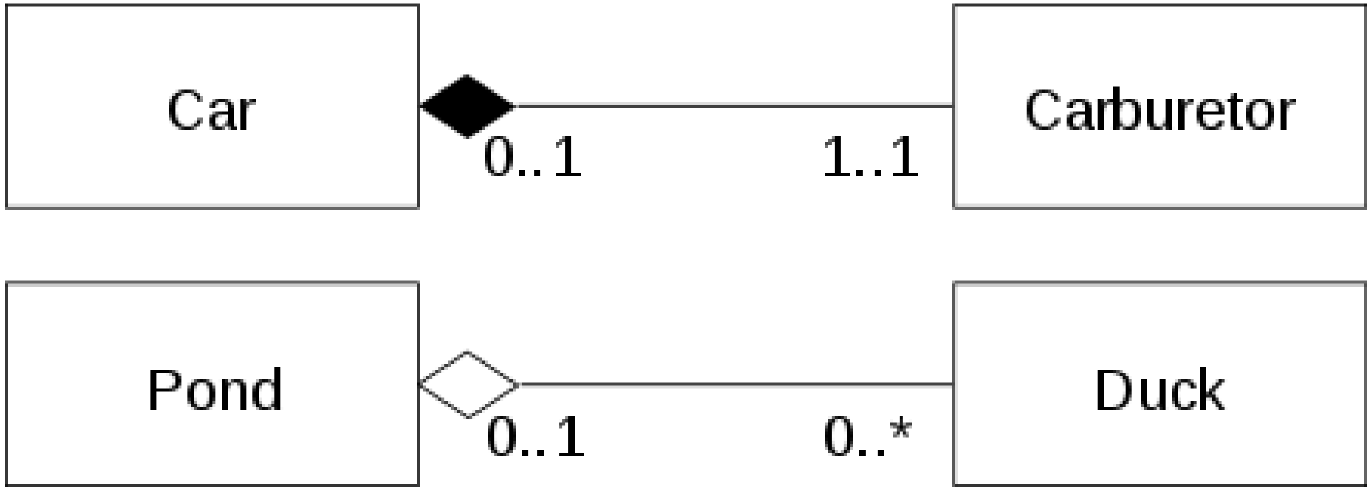
Aggregation – B is composed of A. If B is destroyed A still exists

Composition – B is composed of A. If B is destroyed, A also destroyed





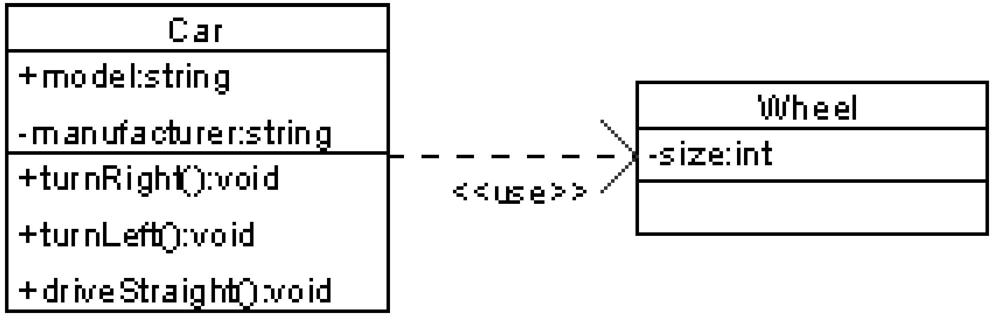
Class diagram



Car is composed of carburetors. Car has exactly 1 carburetor. Carburetor can belong to 0 to 1 cars

Pond is a aggregated with ducks. Pond has 0 to many Ducks. Ducks have 0 to 1 ponds

Class diagram

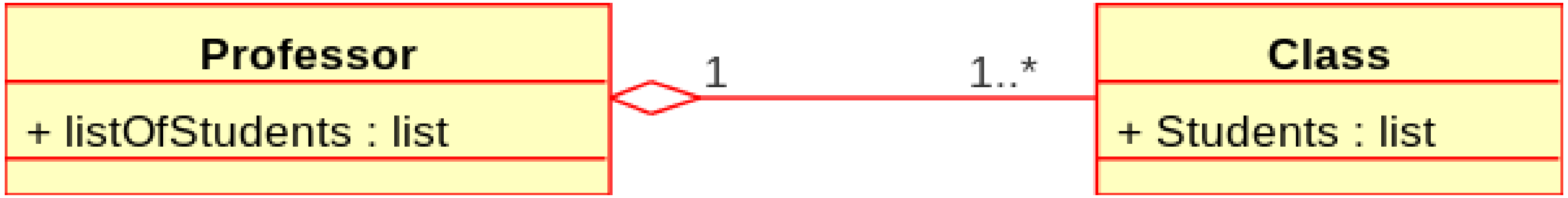


Car

- 2 strings named model, manufacturer
- Methods to turnRight, turnLeft, driveStraight

Car has a relationship with wheel. Some method uses wheel as a local variable to parameter

Class diagram



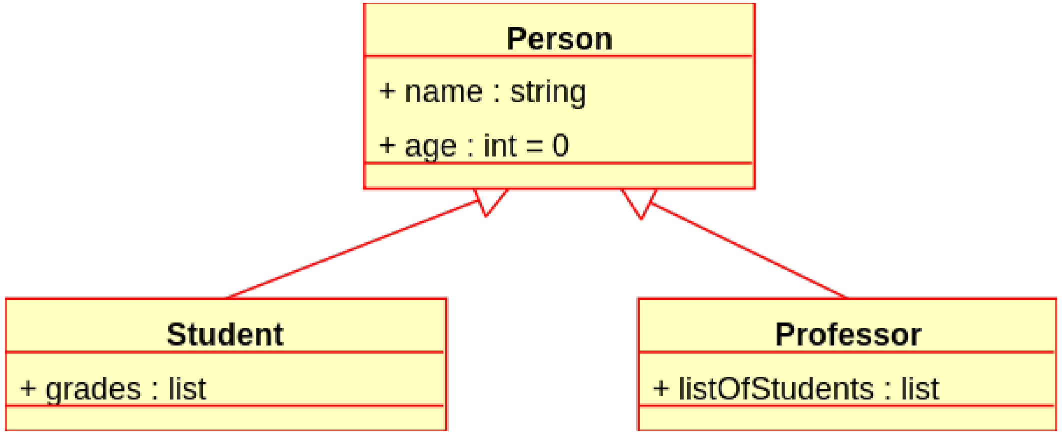
Professor is an aggregate of classes. Professor has 1 to many classes

- Has a list named listOfStudents

Each class has exactly 1 Professor

- Has a list named Students

Class diagram



Person class has a string name and int age as attributes

Student is a subclass of Person

- Has a list named grades

Professor is a subclass of Person

- Has a list named listOfStudents

That's about it for Week 2!

Next Week:

SOLID Design Principles

Dependencies & Coupling