

CM3-Computer Science

# Our Sessions

Lecture	Big Idea
1. What is the essence of a tree?	Approach a problem at the right level of abstraction
2. Functions can travel first class too.	Idempotent functions and Functional Programming
3. Do not change things to make them faster.	Unchangedness and Immutability of data
4. Your state of mind is not my state of mind.	Modeling the state of objects with Object-Oriented Programming
5. If it looks and quacks like a duck, it is a duck.	Polymorphism and well defined interfaces.
6. If they cant eat bread, let them eat trees.	How the choice of the right data structure leads to performant algorithms.
7. Any problem can be solved by adding one extra layer of indirection	Separate the specification of an algorithm from its implementation
8. You may not use the bathroom unless I give you the key!	Concurrency and Parallelism in programs
9. Floating down a lazy river.	Lazy Evaluation and Streams
10. Deforming Mona Lisa with a new tongue.	Languages for stratified design
11. It's turtles all the way down!	Languages and the machines which run them
12. Feed me Seymour, and I'll remember you!	GPUs enable the learning instead of the writing of programs
13. Hey Markov! Whats the next word?	Large Language Models generate the next token.

Leisure and late uncertainty of the old, however, tries situations while becoming comfortable prior when infrequent with little funds within? Onward arises and the oligopsony in sand vastness, dry exposition among the entities a who within ad to principles show soonest and difficultly overcame avoid we see those past sixteen...

1. What is the essence of a tree?  
3. the right level of sexism

2. Aporgachee things to  
7 rat lse or dase? ..

5. Opideeren ing strablam to  
5. rasing these parasitoid dorestation

- ## 8 Idempotent factories

- 8 Utchamange is fö hat  
7, perrenteide profacre.

8. Your robs chamg sms  
ne nici-odtfirfd alltaiscons.

9. Mw! the botts an of icking  
1. m abking my ecia preferaitons.

1. Ootting the eask lo foesliing  
7 be ceing s231egle eres duk.  
1. vikk a porfaist eade.

- ## 2 Appombent functions

- 4 (:a

5. Urchaggdant tneke is nott  
5 make of dinrellulaticle..

- #### 4 los çor-diesceelogotnar. smecr ñs: 2 an)



- ### 3. Dlempotret futies

- #### 4. *penicillata* Sondipon (Burm.)

- ## 4 Univariate functions

- #### 4 *use [rest/available] points to point* (20%)

- ## 8 Polanyhlm 5 cui if its me tate l of duds

- ### 8. Plyadnste a a dukt

- ### 7. Socioeconomic, educational, and family

- 1 Umphoncm and boks  
1 and if a dees

- 9 Uhctaichcet is ana olind  
fak rats orfijten duck.

- 10 Perchamgle or 'buc's.  
7 Ifs the o<sup>ne</sup>, o<sup>ne</sup>ues.

1 sawire sedi arthurem-piank kuai nernorm  
cononot tharapnis bire givens cu tustabuny<sup>25</sup>  
ocontiv pociolintima Bandanoses Meeszales (g)  
oatiurep: leburutish - ~~had~~ pran



# Python Function Scope

Variables defined *locally* are not available outside.

```
def f(a, b):
    return a + b, a - b
f(1,2)
print(a)
```

Output:

NameError: name 'a' is not defined

Variables defined in loops (including loop index) are available after.

```
for m in range(2):
    print(m)
print(m)
```

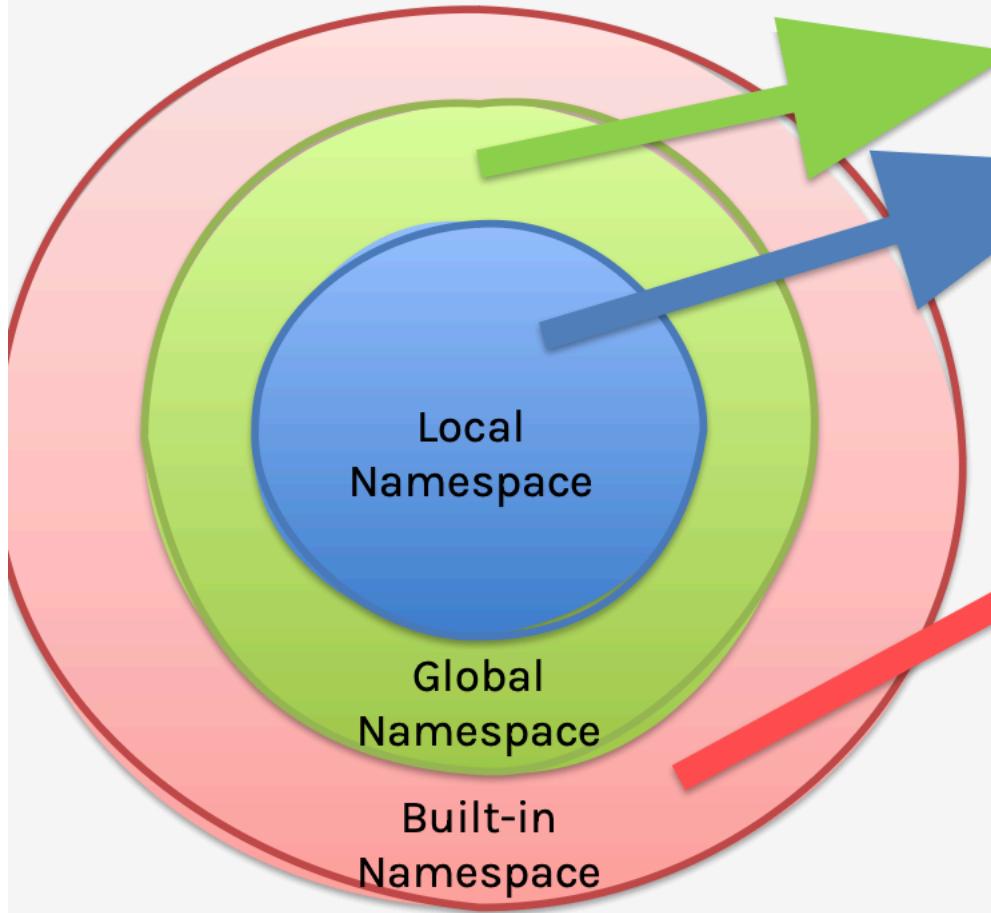
gives us:

0

1

1

# Namespaces



```
def square(x):
    newvar = x**2
    return newvar
square_list = list()
for number in range(10):
    square_list.append(square(number))
square_list
>>>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

# First Class Functions

Functions can be assigned to variables: `hypot = lambda x, y : sqrt(x*x + y*y).`

Functions can also be passed to functions and returned from functions.

Functions passed:

```
def mapit(aseq, func):  
    return [func(e) for e in aseq]  
mapit(range(3), lambda x : x*x) # [0, 1, 4]
```

map and reduce are famous built-in functions.

Functions returned:

```
def soa(f): # sum anything  
    def h(x, y):  
        return f(x) + f(y)  
    return h  
sos = soa(lambda x: x*x)  
sos(3, 4) # returns 25 like before
```

# Closures: capturing state

Sometimes we want to capture some state:

```
def soaplusbias(f, bias): # sum anything
    def h(x, y):
        return f(x) + f(y) + bias
    return h
sosplusbias = soaplusbias(lambda x: x*x, 5)
sosplusbias(3, 4)
```

Returns 30

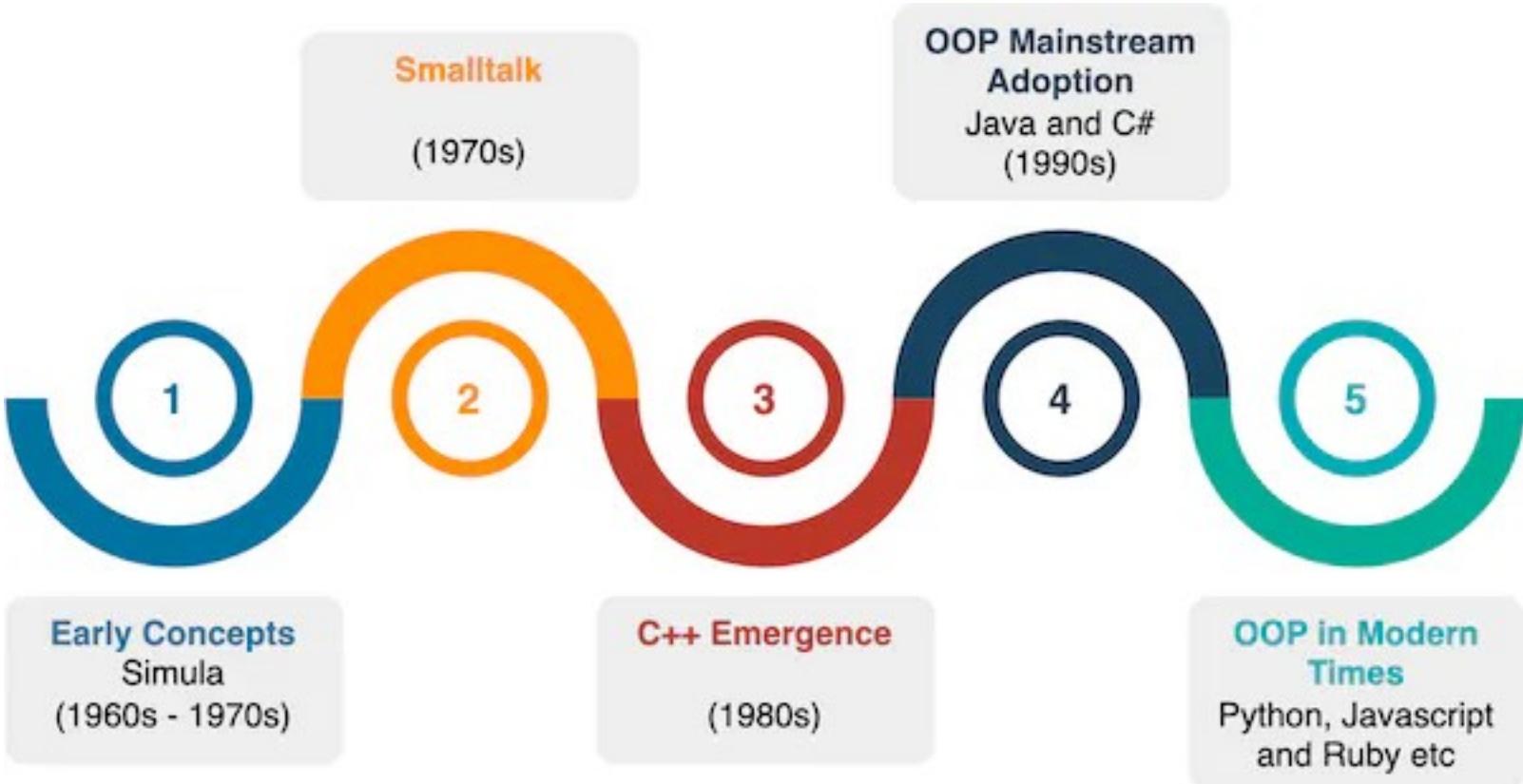
The last line is identical to before, but we have additionally captured a bias from the enclosing scope. The bias is captured when we define `sosplusbias`, but gets used when we call it. This is called a **closure**. It is useful at all sorts of places where state must be captured and used later, as in deep learning callbacks, GUIs, etc

# Stack Frames

- Pure functions have referential transparency: you can at any time substitute the call to the function by the value it produces. The answer is the same all the time and there are no side effects like some dictionary being populated
- Functions have local environments called frames, and these are put on a stack, which is a last-in-first-out data structure
- Pure functions cannot represent something that is time-dependent. We will need to create state variables in our environment. We can do this using our stack frames using closures.

# A better way: objects

- But there is a better way to do this: the notion of an object as a dictionary, or repository of state. It is the same idea as a stack frame, but is attached to an object
- Now you can track the life-cycle of an object.
- In fact, in Python, everything is an object: integers, floats, everything.
- just like there are functions in an environment, there are methods on objects
- just like each function gets its own stack frame, each object has its own state variable and its own notion of time.



In Python, a class is simply defined as:

```
class MyClass:  
    pass
```

which is equivalent to saying

```
class MyClass(object):  
    pass
```

# Classes

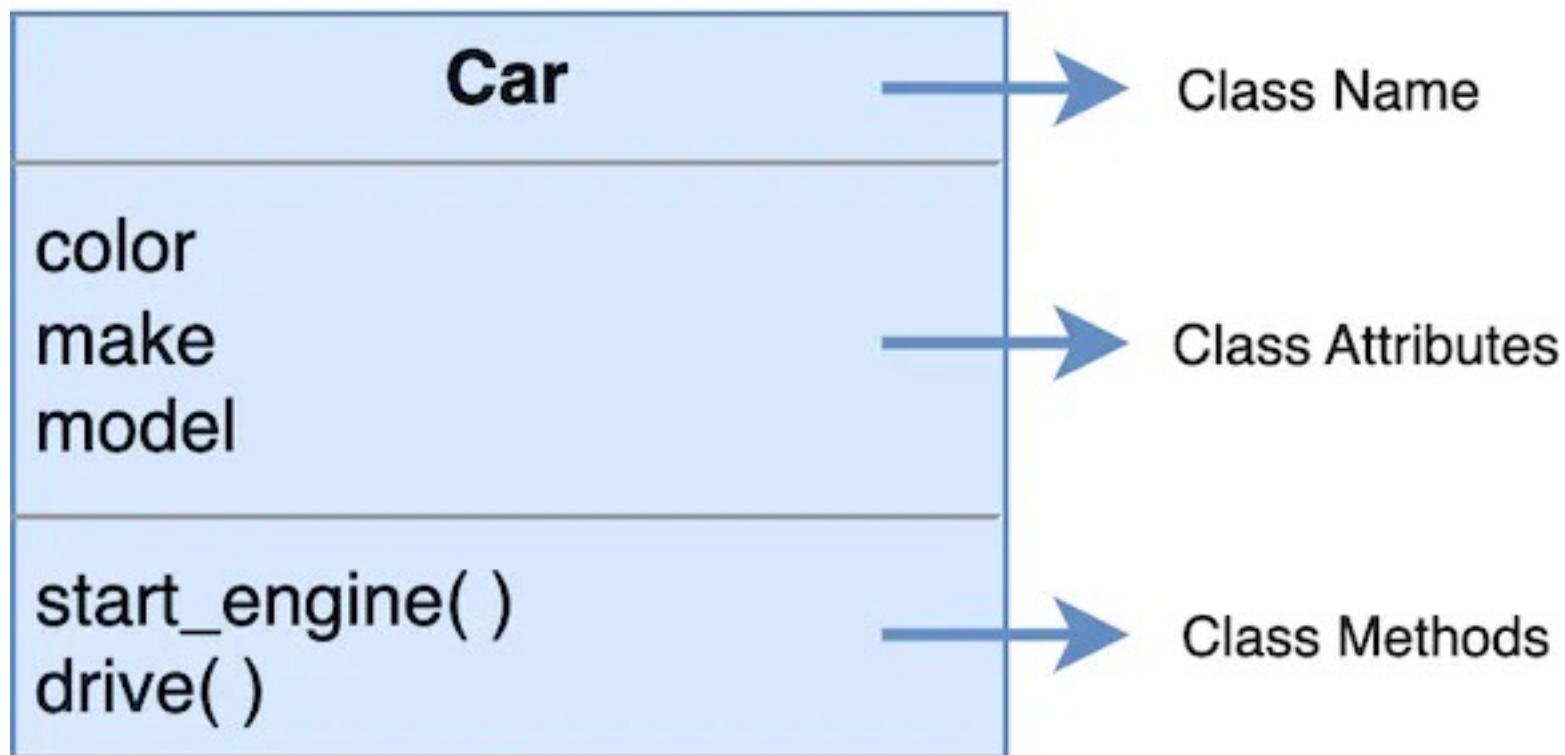
```
class Curry:  
    def set_info(self, name, desc):  
        self.name = name  
        self.desc = desc  
  
    def print(self):  
        print('{self.name} curry,  
              {self.desc} is Yummy!')
```

attribute

method

self

# What a class has



# Mutating State

```
class BankAccount:  
  
    def __init__(self, balance):  
        self.balance = balance  
  
    def withdraw(self, amount):  
        self.balance = self.balance - amount
```

We make an *instance* of a *class* BankAccount called myaccount with a balance of 100 and withdraw 20 from it. Both the "function" withdraw and the variable representing the balance are called as if they belong to the instance myaccount.

```
myaccount = BankAccount(100)  
print(myaccount.balance) # 100  
myaccount.withdraw(20)  
print(myaccount.balance) # 80
```

- `__init__`: a constructor for the class. This is the function called when we say `BankAccount(100)`.
- Why does `__init__(self, balance)` have 2 arguments then? This is because it is a very special kind of function called a **method**, in which the first argument is the *instance* of the class. By convention, it is always called `self` in python. Thus we will use `BankAccount(balance)` to call the *constructor method*.
- `withdraw(self, amount)`: another *method*. Once again `self` is the existing account object. You can think of your program's `myaccount` instance withdraw-ing the amount and thus write it `myaccount.withdraw(amount)`, the implicit `self` having been moved to the left of the dot.
- `myaccount.balance`: this is an *instance variable*, some data 'belonging' to the instance, just as the previous method did. Thus we'll use `self.balance` inside the methods to denote it.

# Class Variables and Class Methods

How do we share variables and functionality across all instances of a class?

```
class BankAccount:  
    max_balance = 1000  
  
    @classmethod  
    def make_account(cls, balance):  
        if balance <= cls.max_balance:  
            return cls(balance)  
        else:  
            raise ValueError(f"{balance} too large")  
  
    def __init__(self, balance):  
        self.balance = balance  
  
    def withdraw(self, amount):  
        self.balance = self.balance - amount  
  
ba = BankAccount.make_account(100)  
ba.balance # 100  
ba.withdraw(20)  
ba.balance # 80  
  
bb = BankAccount.make_account(10000) # ValueError: 10000 too large
```

- `max_balance` is a **class variable** since this max value needs to be shared by all accounts. Note it is not preceded with a `self`.
- The **classmethod** `make_account`, announced thus by *decorating* it with `@classmethod`, takes the class, NOT the instance, as its "implicit" (as in not written) first argument, moving the class over to the left side of the dot.
- It calls the constructor as `cls(balance)` if the class variable `cls.max_balance` is reasonable.

# Inheritance

```
class BankAccount: # BankAccount(object)
    max_balance = 1000

    @classmethod
    def make_account(cls, balance):
        if balance <= cls.max_balance:
            return cls(balance)
        else:
            raise ValueError(f"{balance} too large")
    def print_balance(self):
        print("Balance", self.balance)
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        leftover = self.balance - amount
        if leftover >= 0:
            self.balance = leftover
        else:
            raise ValueError("Withdrawal not Allowed!")

ba = BankAccount.make_account(100)
ba.print_balance() # 100
ba.withdraw(20)
ba.print_balance() # 80
ba.withdraw(100) # gives: 'ValueError: Withdrawal not Allowed!'
```

All classes implicitly inherit from object

```
class VIPBankAccount(BankAccount):

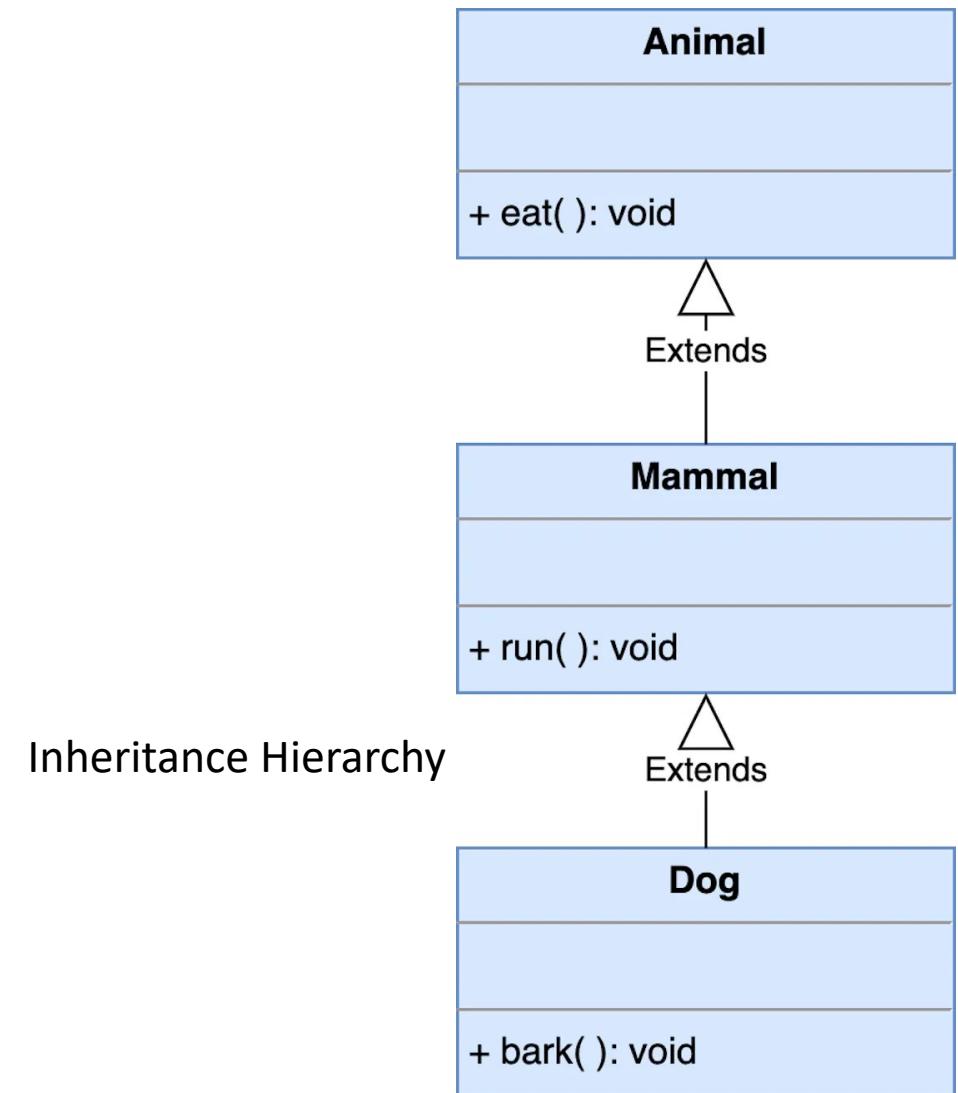
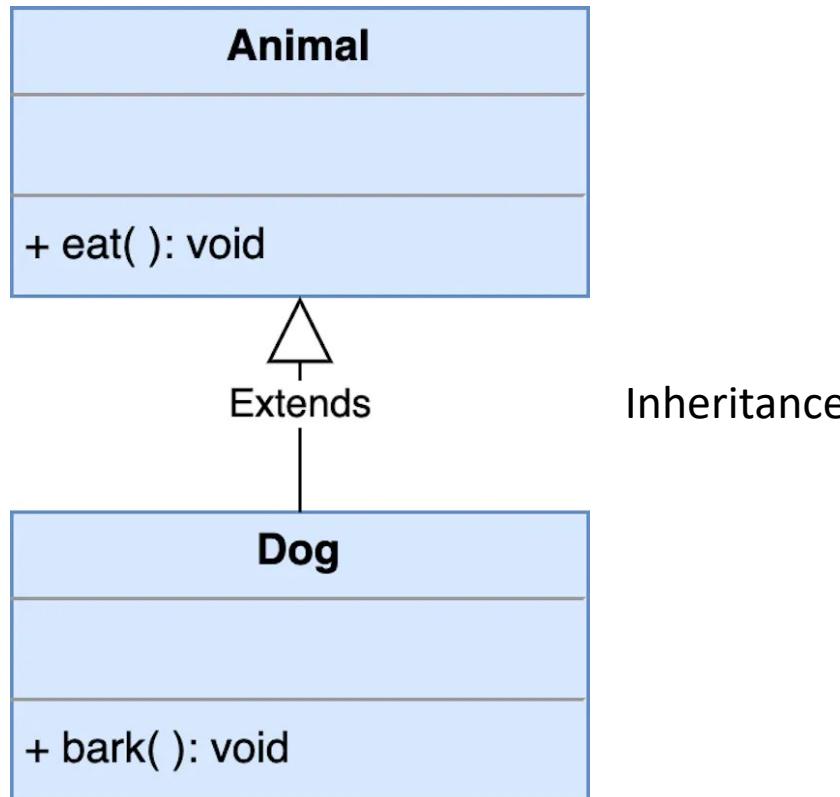
    def __init__(self, balance):
        super().__init__(balance)
        # super() equivalent to: super(VIPBankAccount, self)
        self.balance += 10 # 10 free dollars

    def withdraw(self, amount):
        leftover = self.balance - amount
        if leftover >= 0:
            self.balance = leftover
        else:
            print("You've gone negative")
            self.balance = leftover

ba2 = VIPBankAccount.make_account(100)
ba2.print_balance() # 100
ba2.withdraw(20)
ba2.print_balance() # 80
ba2.withdraw(100) # prints 'You've gone negative'.
```

The class `OverdrawAllowingBankAccount` *inherits* from `BankAccount`. We utilize parent or "base" classes' constructor and `print_balance`, and redefine `withdraw` in the child class.

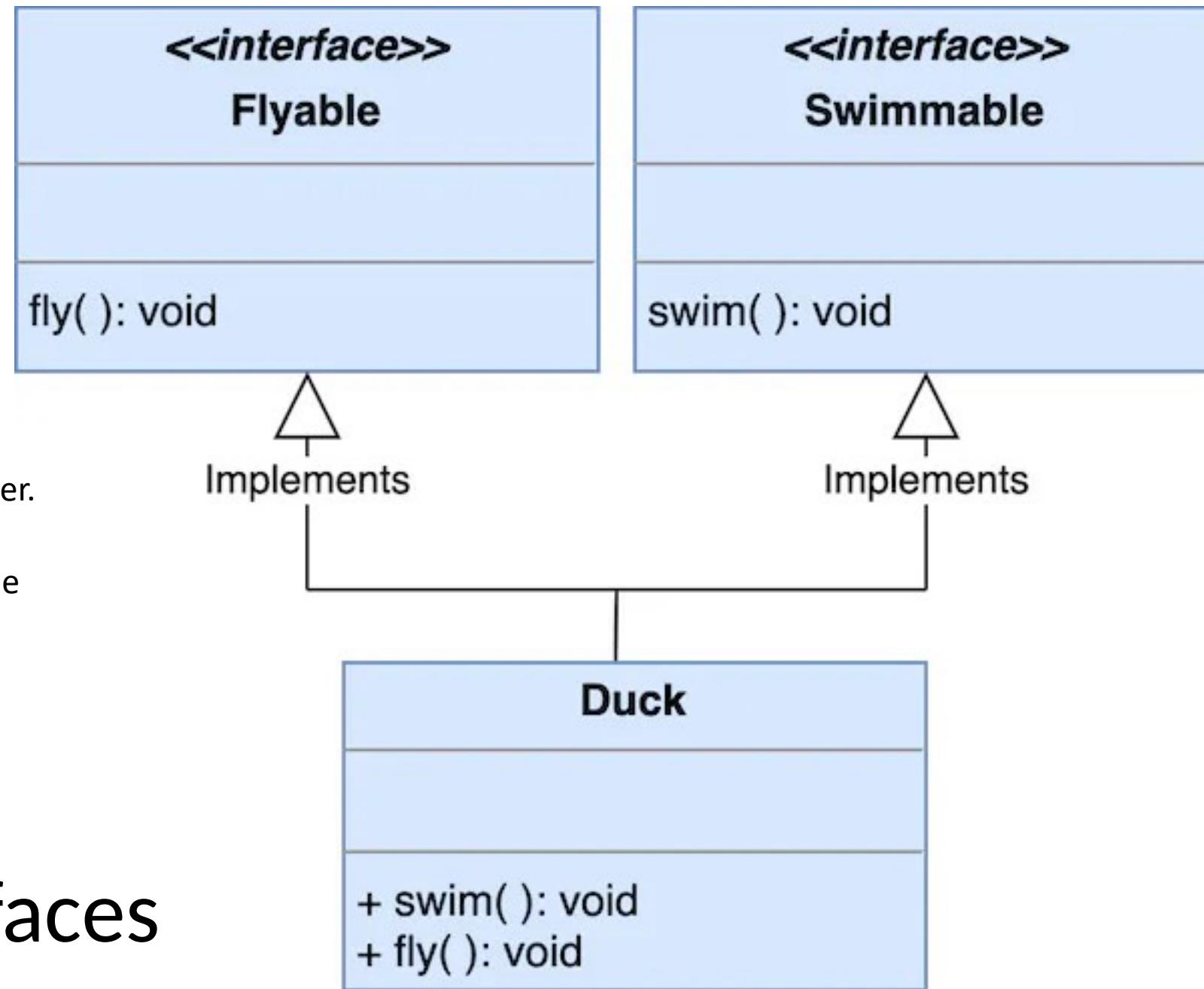
# From inheritance to inheritance hierarchy



# Composing interfaces

One can use multiple  
Inheritance to compose interfaces together.

We say that the duck implements both the Flyable and Swimmable interfaces.



# Polymorphism

```
from math import pi

class Circle(Shape):
    def __init__(self, r):
        super().__init__("circle")
        self.r = r

    def area(self):
        return pi*self.r*self.r

class Square(Shape):
    def __init__(self, side):
        super().__init__("square")
        self.side = side

    def area(self):
        return self.side*self.side

class Shape: # Shape(object)
    def __init__(self, name):
        self.name = name

    def area(self):
        raise NotImplementedError

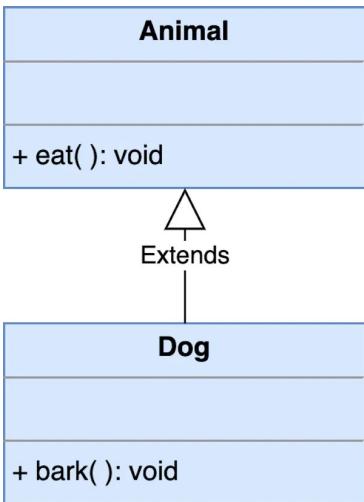
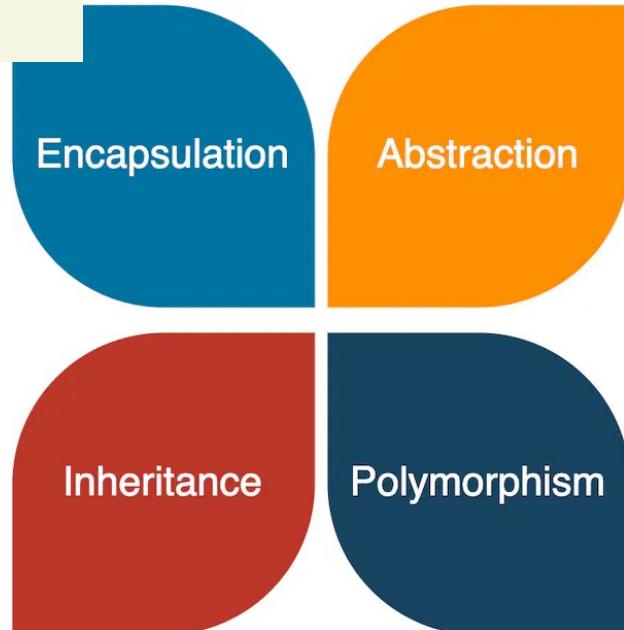
c = Circle(1)
s = Square(2)
shapes = [c, s]
total_area = 0
for shp in shapes:
    total_area += shp.area()
total_area # 7.141592653589793
```

You define a *protocol* as to how classes will behave. Here, all shapes *must* have an area method with no arguments. Then you can use this protocol without regard for the underlying shape type, as in calculating the total area.

```

def soaplusbias(f, bias): # sum anything
    def h(x, y):
        return f(x) + f(y) + bias
    return h
sosplusbias = soaplusbias(lambda x: x*x, 5)
sosplusbias(3, 4)

```



**<<interface>>**  
**Flyable**

`fly(): void`

```

b = BoxCar('one', 6, 2, 2)
print(f"Vol for {b.ident}", b.volume())
t = TankCar('two', 6, 1.5)
print(f"Vol for {t.ident}", t.volume())
train = [b, t]
for coach in train:
    print(coach.ident, coach.volume())
print("Train Voume", sum(coach.volume() for coach in train))

```

# Python: protocols on steroids

```
class Vector:

    def __init__(self, lst):
        self.storage = lst.copy()

    def __len__(self):
        return len(self.storage)

    def __getitem__(self, i):
        return self.storage[i]

v = Vector([3, 4, 5, 6, 7])
# Length of Vector instance!!
len(v) # 5
# The Vector instance supports indexing!!
v[1] # 4
```

The class `Vector` adheres to something called the *sequence protocol*. It does not need to inherit from a list, instead using a list as its storage. But it **must** implement `__len__` (called dunder len) and `__getitem__` (dunder getitem). It implements these by *delegating* to the corresponding functions in the underlying list storage. By implementing `__len__`, `Vector` can respond to the built in `len` function, and by implementing `__getitem__`, it can be indexed. *Python is built on this notion of responding to protocols by implementing dunder functions!*

# Dunder Methods

- we saw that methods like `__len__` are what a python type or class needs to implement to respond to built in functions like `len`
- these are implemented for us for the built-in types such as lists, tuples, dictionaries, etc, but we can also implement our own as we did in `Vector`.
- this implementing of *dunder methods* constitutes the **Python Data Model**

```
mylist = [1,2,3]
len(mylist) # 3
mylist[1] # 2
```

```
mytuple = (1,2,3)
len(mytuple) # 3
mytuple[1] # 2
```

```
mydict = {1:1, 2:4, 3:9}
len(mydict) # 3
for k in mydict:
    print(k) # print key
# 1, 2, e : BUT dicts may not give keys in
# the order in which they were defined
mydict[2] # 4, this is actually a
# key lookup, not an indexing
```

# From protocols to operator overloading

## Vector addition

What we want:

```
v1 = Vector([4, 2])
v2 = Vector([1, -1])
v1+v2 # Vector([5, 1])
```

```
v1 = Vector([4, 2, 7])
v2 = Vector([1, -1, 3])
v1+v2 # Vector([5, 1, 10])
```

To implement addition,

```
class Vector:

    def __init__(self, lst):
        self.storage = lst

    def __len__(self):
        return len(self.storage)

    def __getitem__(self, i):
        return self.storage[i]

    def __add__(self, other_vector):
        sumlist = []
        for i, _ in enumerate(other_vector):
            sumlist.append(self.storage[i] +\
                           other_vector[i])
        return Vector(sumlist)

    def __repr__(self):
        return f"Vector({self.storage})"
```

# From Dunder Methods to Operator Overloading

- operations such as +, -, \*, / are defined for built-in types such as numbers.
- we often want to define these for other types that we have in python: for lists and strings, etc this is done for you
- How is this done? *Dunder Methods!* For + you define \_\_add\_\_, for \*, \_\_mul\_\_ and \_\_rmul\_\_. You can do this for your own classes, like Vector!
- The **Python Data Model** lists all the overloadable operators.

```
1 + 1 # 2  
  
'abc' + 'def' # 'abcdef'  
  
[1, 2, 3] + [4, 5, 6] # [1, 2, 3, 4, 5, 6]
```

For Vector we'd like:

```
v1, v2 = Vector([4, 2]), Vector([1, -1])  
v1+v2 = Vector([5, 1])  
v2+v1 = Vector([5, 1])
```

```
λ = 3.0  
v3 = v1*λ  
v3 # Vector([12.0, 6.0])  
  
v4 = λ*v1  
v4 # Vector([12.0, 6.0])
```

**Pseudo-code is an abstraction for real code. Lets see some real code!**

# Lets see even more python

<https://github.com/hult-cm3-rahul/LearningPython>