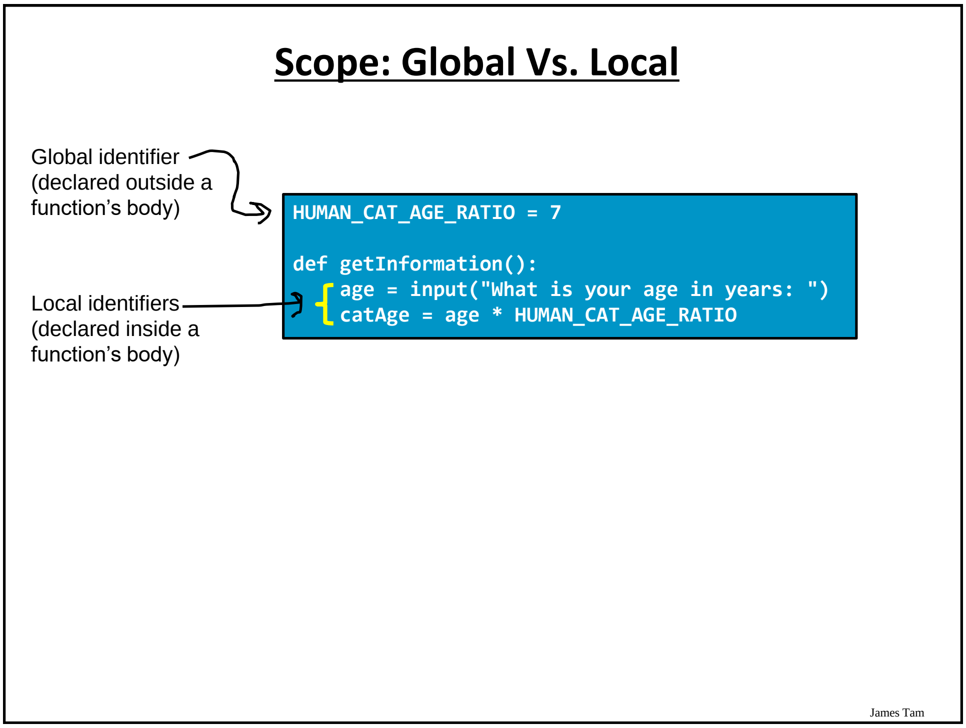
Review & Extra Details

Problem decomposition through functions: scope
Object-Orientated concepts

James Tam

Decomposition: Scope

James Tam



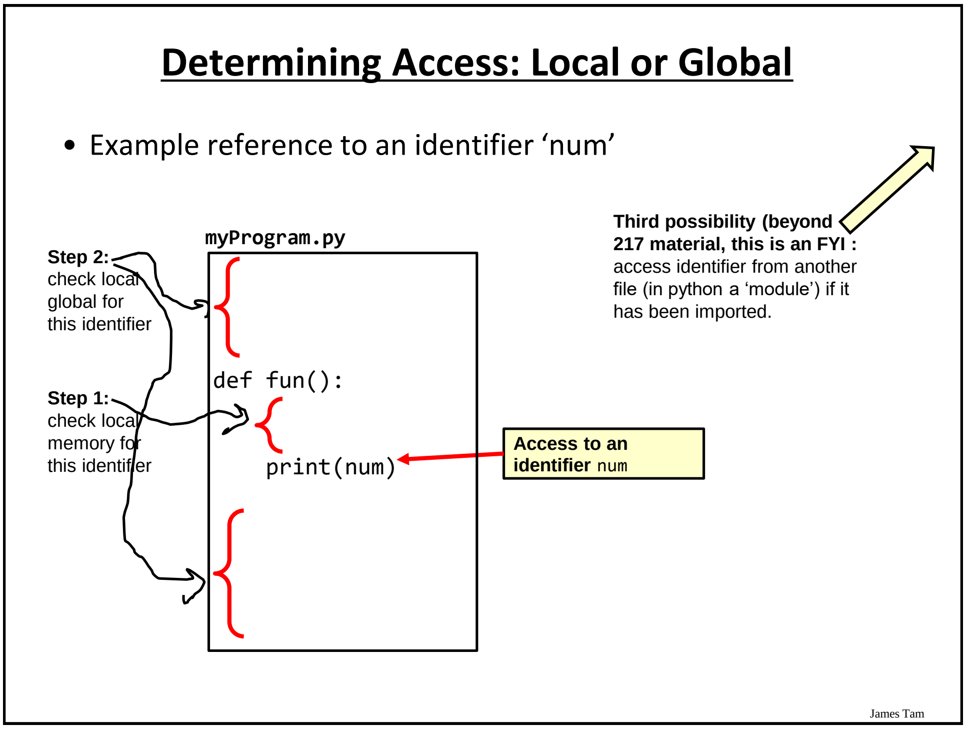
Scope: Global Vs. Local

Global identifier (declared outside a function's body)

Local identifiers (declared inside a function's body)

```
HUMAN_CAT_AGE_RATIO = 7

def getInformation():
    age = input("What is your age in years: ")
    catAge = age * HUMAN_CAT_AGE_RATIO
```

James Tam



Determining Access: Local or Global

- Example reference to an identifier 'num'

Step 2: check local global for this identifier

Step 1: check local memory for this identifier

```
myProgram.py

def fun():

    print(num)
```

Access to an identifier `num`

Third possibility (beyond 217 material, this is an FYI : access identifier from another file (in python a 'module') if it has been imported.

James Tam

Example: Local Access, Global Access

```
x = False
def fun(y):
    z = 3.0
    print(y,z)
    while(x):
        pass #FYI: Bad style: loop control not updated


fun(1)
```

Example: Shadowing

- Shadowing is an extension of the rule: first look local and then look global.
- It occurs when a **local identifier** matches the name of global identifier.
  - The local identifier shadows (or hides) the global identifier.
- Example:

```
age = 37
def fun():
    age = 73
    print(age)  #73 is the output
```

Object-Oriented (O-O) Principles

James Tam

Types

- They are real life categories (or concepts) of physical or even abstract entities
  - Cars
  - Cats
  - People
  - Pens
  - Dogs
  - Zombies
  - Super heroes
  - Web server
  - Etc.

Python implementation
(it defines a new type of variable)

```
class Person:
    def __init__(self):
        #etc.
```

James Tam

Instances

- Instances are actual examples of these categorical types (in this case examples of actual or fictional people).
  - Otto von Bismarck
  - Hohiro Kurita
  - Chun Li
  - Khan Noonien Singh
  - Tony Montana

Python implementation
(it defines a new type of variable)

```
khan =
  Person(200,"Khan!!!")
scarface =
  Person(42,"Tony Montana")
```

Attributes

- Information that is associated with the type e.g. a person has these attributes:
  - Height
  - Weight
  - Blood type
  - Age
  - Etc.
  - All examples or instances of this type will have these attributes.
- The specific value of these attributes can vary from instance to instance e.g. a ‘runner’ has a time for the runs but the actual times will depend upon the particular runner.

Python implementation

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

Methods

- Actions or possible behaviors of that type e.g. a person can carry out these actions:
  - Eat
  - Sleep
  - Breath
  - Procreate
  - Etc.
  - All examples or instances of the type can carry out these actions.
- The particular action (if any) undertaken by an instance will vary from instance to instance e.g. a live person is constantly breathing but the same cannot be said about procreating.

Python implementation

```
class Person:

    def __init__
      (self,age,name):
        self.age = age
        self.name = name

    def __str__(self):
        aStr = self.name + \
          " " + \
          str(self.__age)
        return(aStr)
```

Information Hiding/Encapsulation (Def. #2)

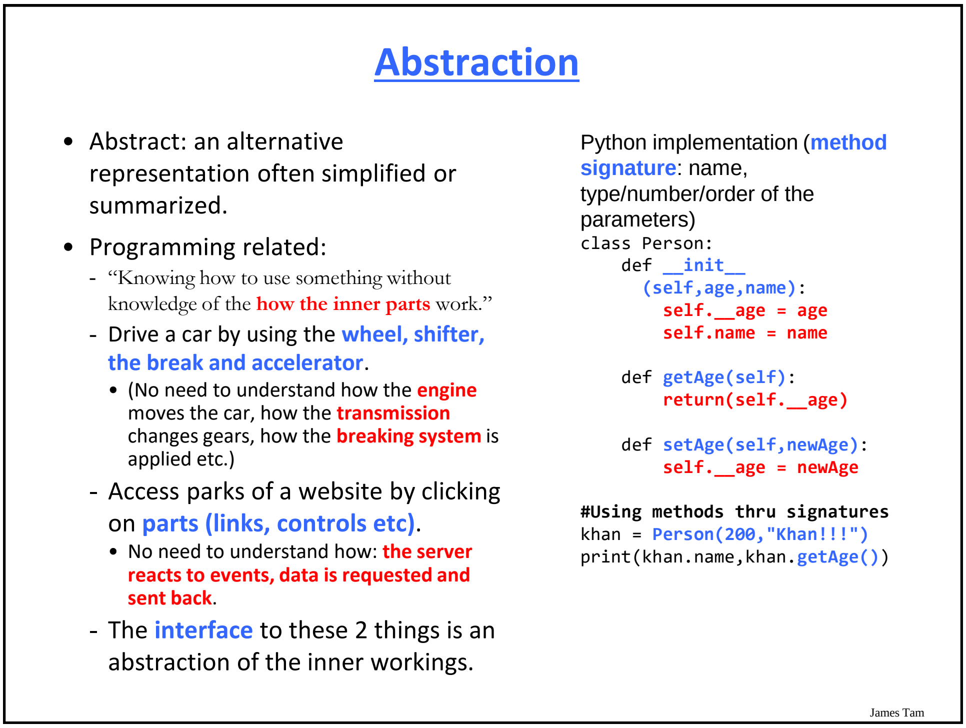
- The inner workings are protected from outside access.
  - One cannot directly access the mechanical parts of the car while driving.
    - One cannot directly access the **transmission** to change gears.
    - One cannot directly squeeze the **brake pads** to slow the car.
    - Etc.
  - When using a browser to access files on a website one does not have direct access to the storage device's file system as a computer user.
    - One cannot view all **folders/directories**.
    - In folders/directories that are visible on **cannot delete files**.

Python implementation (**access level of attributes**

```
class
Person:
    def __init__
      (self,age,name):
        self.__age = age
        self.name = name

    def getAge(self):
        return(self.__age)

    def setAge(self,newAge):
        self.__age = newAge

#Won't work outside of class
#print(khan.__age,khan.name)
```

```
self.__age = age
```

Underscores: signifies 'private' access restricted to within the class

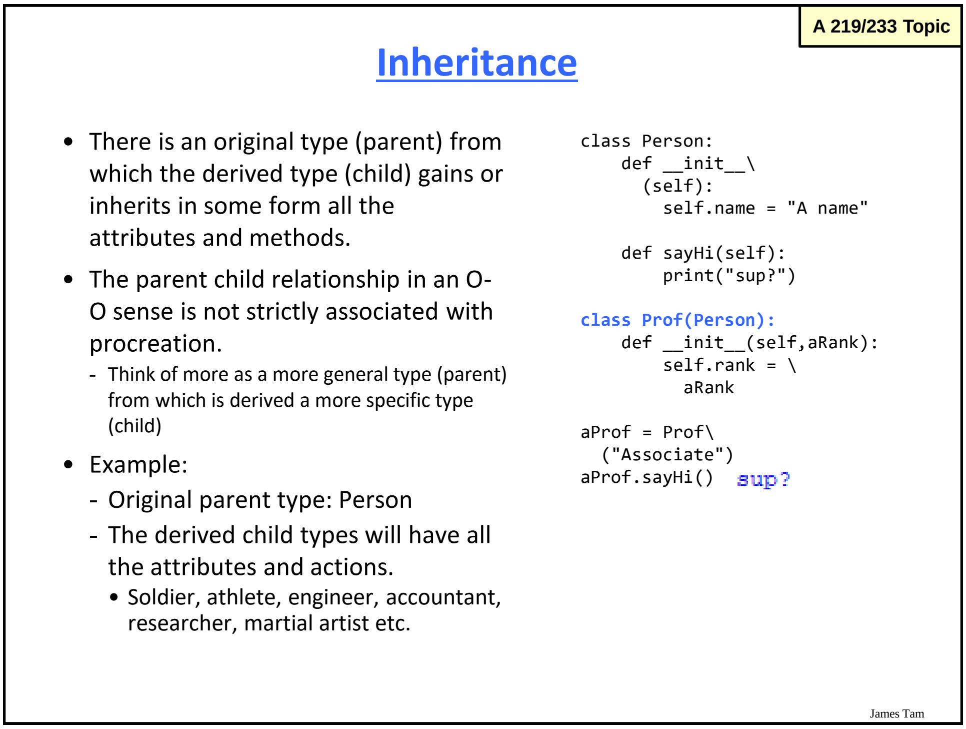


Abstraction

- Abstract: an alternative representation often simplified or summarized.
- Programming related:
  - "Knowing how to use something without knowledge of the **how the inner parts** work."
  - Drive a car by using the **wheel, shifter, the break and accelerator**.
    - (No need to understand how the **engine** moves the car, how the **transmission** changes gears, how the **breaking system** is applied etc.)
  - Access parks of a website by clicking on **parts (links, controls etc)**.
    - No need to understand how: **the server reacts to events, data is requested and sent back**.
  - The interface to these 2 things is an abstraction of the inner workings.

Python implementation (**method signature**: name, type/number/order of the parameters)

```
class Person:
    def __init__
      (self,age,name):
        self.__age = age
        self.name = name

    def getAge(self):
        return(self.__age)

    def setAge(self,newAge):
        self.__age = newAge

#Using methods thru signatures
khan = Person(200,"Khan!!!")
print(khan.name,khan.getAge())
```



A 219/233 Topic

Inheritance

- There is an original type (parent) from which the derived type (child) gains or inherits in some form all the attributes and methods.
- The parent child relationship in an O-O sense is not strictly associated with procreation.
  - Think of more as a more general type (parent) from which is derived a more specific type (child)
- Example:
  - Original parent type: Person
  - The derived child types will have all the attributes and actions.
    - Soldier, athlete, engineer, accountant, researcher, martial artist etc.

```
class Person:
    def __init__\
      (self):
        self.name = "A name"

    def sayHi(self):
        print("sup?")

class Prof(Person):
    def __init__(self,aRank):
        self.rank = \
          aRank

aProf = Prof\
  ("Associate")
aProf.sayHi()   sup?
```

A 219/233 Topic

Inheritance

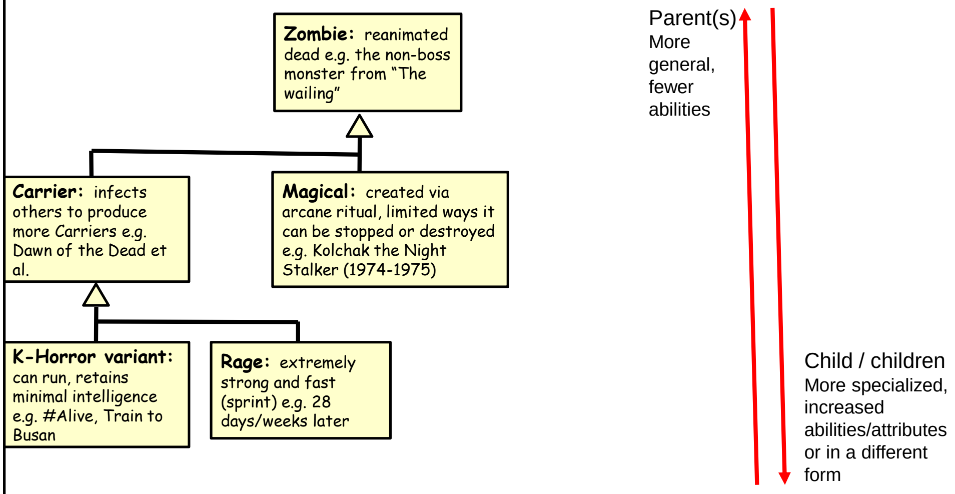
- There is an original type (parent) from which the derived type (child) gains or inherits in some form all the attributes and methods.
- The parent child relationship in an O-O sense is not strictly associated with procreation.
  - Think of more as a more general type (parent) from which is derived a more specific type (child)
- Example:
  - Original parent type: Person
  - The derived child types will have all the attributes and actions.
    - Soldier, athlete, engineer, accountant, researcher, martial artist etc.

```
class Person:
    def __init__\
      (self):
        self.name = "A name"

    def sayHi(self):
        print("sup?")

class Prof(Person):
    def __init__(self,aRank):
        self.rank = \
          aRank

aProf = Prof\
  ("Associate")
aProf.sayHi()   sup?
```

Prof is-a Person so it can call this method

James Tam

Example Inheritance Hierarchy



Zombie: reanimated dead e.g. the non-boss monster from "The wailing"

Carrier: infects others to produce more Carriers e.g. Dawn of the Dead et al.

Magical: created via arcane ritual, limited ways it can be stopped or destroyed e.g. Kolchak the Night Stalker (1974-1975)

K-Horror variant: can run, retains minimal intelligence e.g. #Alive, Train to Busan

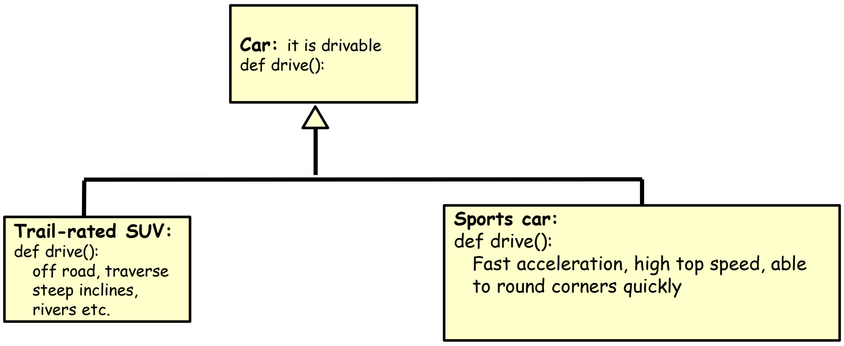
Rage: extremely strong and fast (sprint) e.g. 28 days/weeks later

Parent(s)
More general, fewer abilities

Child / children
More specialized, increased abilities/attributes or in a different form

James Tam

Another Hierarchy: Illustrating How Abilities Differ (Parent-Child)



Car: it is drivable
def drive():

Trail-rated SUV:
def drive():
off road, traverse steep inclines, rivers etc.

Sports car:
def drive():
Fast acceleration, high top speed, able to round corners quickly

Copyright Notification

- Unless otherwise indicated, all images in this presentation were provided courtesy of James Tam.