Objected Oriented Python Topics

- **Objects and Classes**
- **Inheritance**
- **Encapsulation**
- **Abstract Classes**
- **Interfaces**

Object Oriented Programming is a paradigm of programming used to represent real-world objects in programs.

Real-world objects have certain properties and behaviours. These are modelled with Classes and Objects in Python.

Properties or Data Fields capture the state of real-world objects and Methods capture the behavior of real-world objects

Objects and Classes

- ***Classes*** **and** ***Objects*** **are the two main aspects of Object Oriented Programming (OOP)**
- **A class creates a new** ***type*** **(remember data types?)**
  - **where objects are instances of the class type**

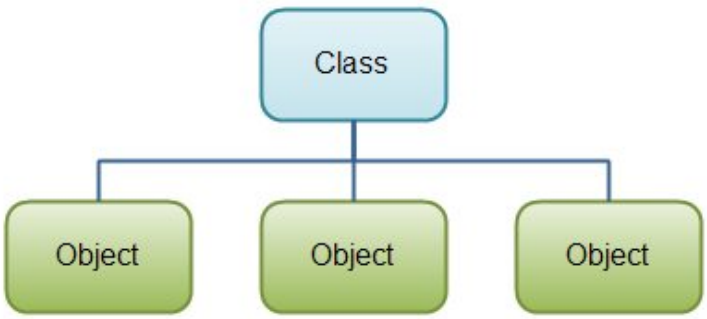


Fig 1 - Instantiating

In [ ]:

```
# create an empty class called `Person`
# (NOTE: class names always begin with a capital letter!)
class Person:
    pass # empty block

# Create object `p` from class `Person`
p = Person()

# Print `p`
print(p)
```

Attributes of a Class

- **(Data) Fields: Variables that belong to an object or class are referred to as fields**
  - **Objects can store data using ordinary variables that** ***belong*** **to the object**
- **Methods: Objects can also have functionality by using functions that belong to a class**
  - **Such functions are called methods of the class**
- **Collectively, the fields and methods can be referred to as the attributes of that class.**

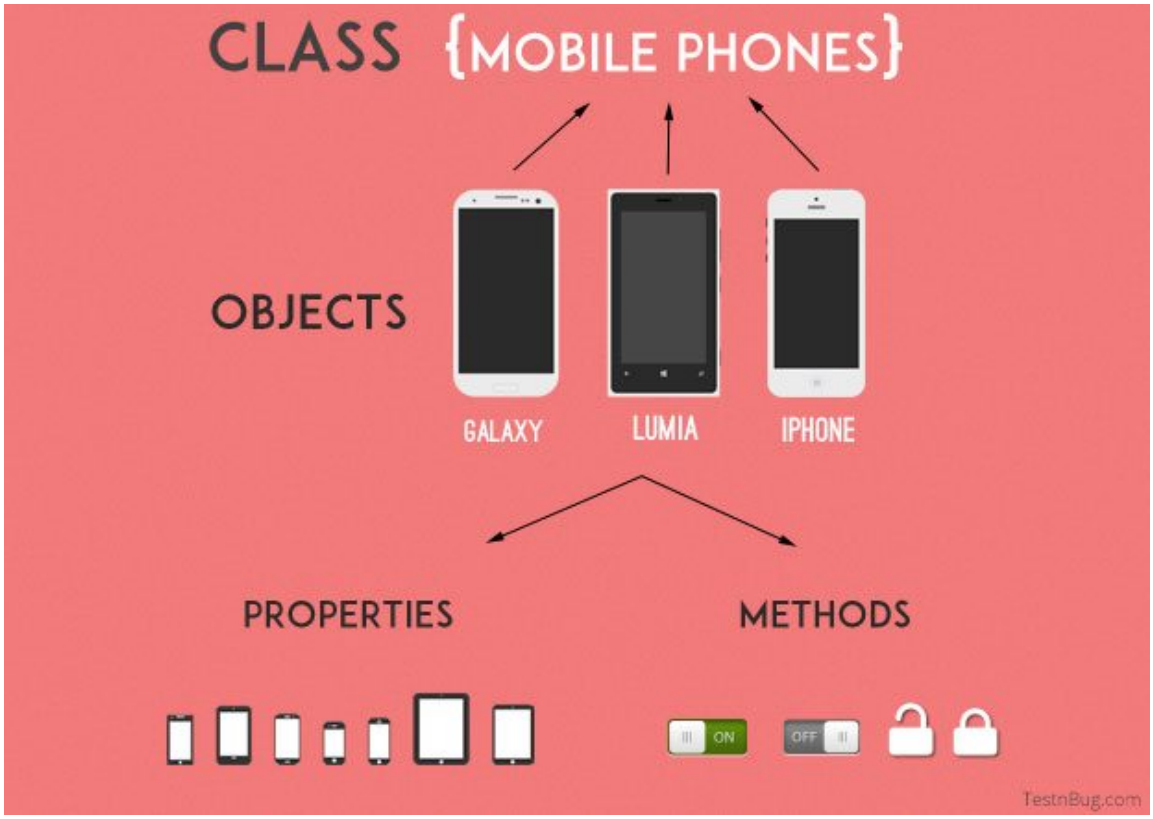


Fig 2a - Data Fields (Properties) and Methods

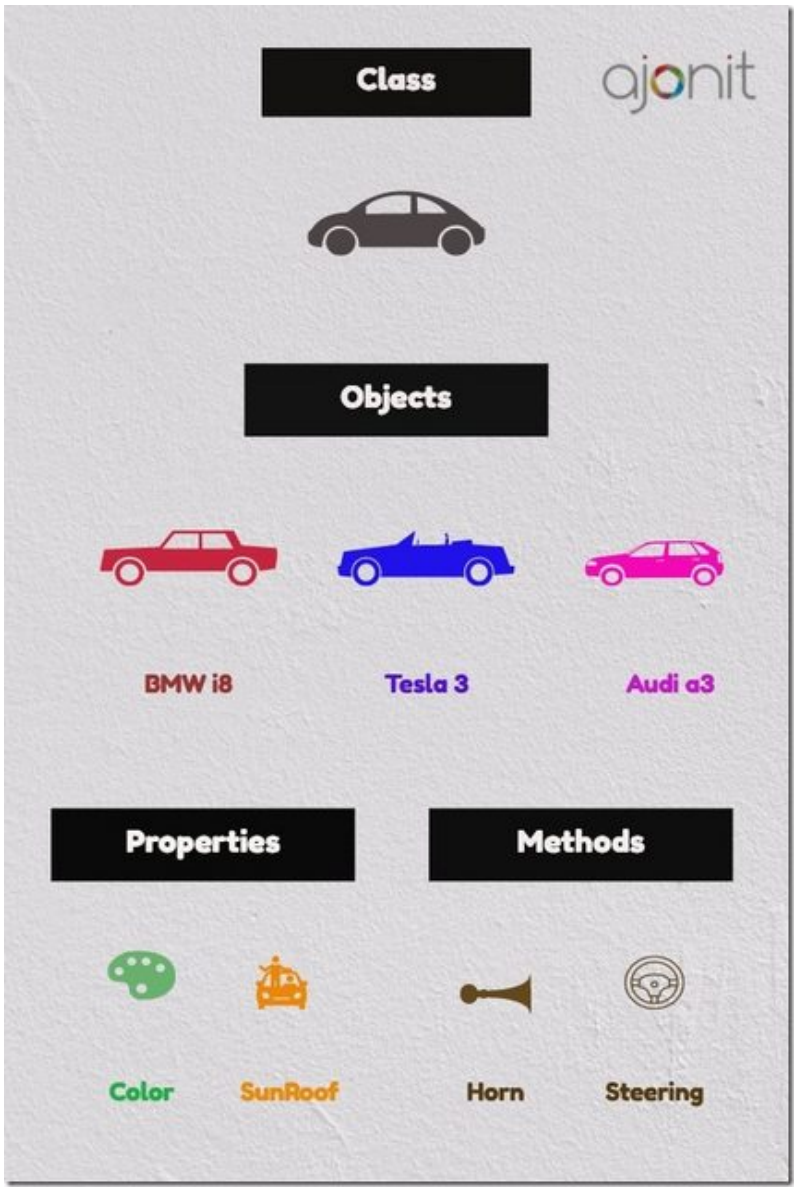


Fig 2b - Data Fields (Properties) and Methods

Methods

In [ ]:

```
# create a class called `Person`
class Person:
    # create a new built in method called say_hi
    def say_hi(self):
        print('Hello, how are you?')

# instantiate object `p` from `Person` class
p = Person()

# run the built-in method from object `p`
p.say_hi()
```

- **Notice that the `say_hi` method takes no parameters when calling it in the object**
  - **but still has the `self` in the class function definition**

`self` keyword

- **Class methods have only one specific difference from ordinary functions**
  - **they must have an extra first parameter that has to be added to the beginning of the parameter list**
  - **by convention, it is given the name `self`.**
- **do not give a value for this parameter when you call the method**
  - **Python will automatically provide it**
  - **this particular variable refers to the object itself**

Data Fields

- **we have already discussed the functionality part of classes and objects (i.e. *methods*),**
  - **now let us learn about the data part.**
- **the data part, i.e. *fields*, are nothing but ordinary variables that are bound to the *namespaces* of the classes and objects**
  - **recall scope of variables in functions, this works similar to that, but at `class` and `object` levels**

`__init__` method: the constructor function

- **There are many method names which have special significance in Python classes**
- **We will see the significance of the `__init__` method**
  - **The `__init__` method is run as soon as an object of a class is *instantiated* (i.e. created)**
  - **The method is useful to do any *initialization* you want to do with your object**
    - **i.e. passing initial values to your object variables**

In [ ]:

```
# create Class called `Person`
class Person:

    # create the __init__ constructor function
    def __init__(self,name):
        self.name = name # object variable

    # create method called say_hi
    # this will be a built-in function for this class `Person`
    def say_hi(self):
        print('Hello, my name is ', self.name)

# instatiate an object `p` from class `Person`
# remember to input the `name` as Class argument
p = Person('Jill')
```

```
# call built-in method
p.say_hi()
```

- **The variables under the `__init__` function have to be passed as arugments when creating the object from the class**
  - **When creating new instance `p`, of the class `Person`**
    - **we do so by using the class name, followed by the arguments in the parentheses**
- **We define the `__init__` method as taking a parameter name**
  - **We do not explicitly call the `init` method, this is the special significance of this method**
- **`self.name` means that there is something called "name" that is part of the object called "self"**
  - **the other `name` is a local variable**

Class Variable vs. Object Variable

- **there are two types of *data fields***
  - **class variables**
  - **object variables**
- **these are classified depending on whether the class or the object owns the variables respectively**

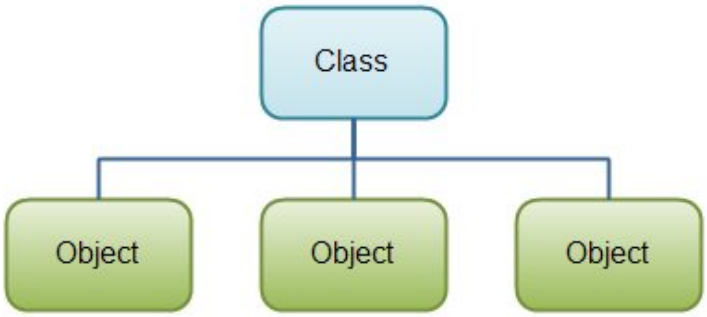


Fig 3 - Class vs. Object

class variable:

- **they can be accessed by all instances of that class**
- **here is only one copy of the class variable and when any one object makes a change to a class variable**
- **that change will be seen by all the other instances.**

object variable:

- **are owned by each individual object/instance of the class**
- **each object has its own copy of the field**
- **they are not shared and are not related in any way to the field by the same name in a different instance**

`Robot` Class Example

- **it is important to learn `classes` and `objects` with examples**
  - **so we will look at a `Robot` class example**
- **We define a `Robot` class with the following attributes:**
  - **Class Variables:**
    - **`population`: keeps count of the number of objects instantiated from the class**
  - **Class Function:**
    - **`how_many`: returns the current number of robots**
  - **Properties:**
    - **`name`: object varible that hold the name of the current robot**
  - **Methods:**
    - **`say_hi`: robot says hi**

- `die`: **robot gets terminated and updates the number of total robots**

ASIDE #1:

- ***Docstrings***:
  - **these are string literals that appear right after the definition of a function, method, class, or module**
  - **they appear when `help(function_name)` is run**

In [ ]:

```
# Docstrings Example

def is_even(num):
  # Docstring
  """
  Input: an integer
  Output: if input is even (True for even, False for not)
  """
  return num % 2 == 0

help(is_even)
```

ASIDE #2:

- `.format()`
  - **used to fill in the blanks of a string in the `print()` statement**

In [ ]:

```
# `.format()` example

pi_value = 3.14
print("The value of Pi is {}".format(pi_value))
```

Robot Class Setup

In [ ]:

```
# Define a class called Robot
class Robot:
  """Represents a robot, with a name.""" # Docstrings

  # Class Variable: population
  population = 0

  # Constructor Function: __init__()
  def __init__(self, name):
      """Initializes the data.""" # Docstrings
      self.name = name # Object Variable
      print("(Initializing {})".format(self.name))

      # When a new robot is created, the robot
      # adds to the population
      Robot.population += 1

  # Object Method: die()
  def die(self):
      """I am dying.""" # Docstrings
      print("{} is being destroyed!".format(self.name))

      Robot.population -= 1

      if Robot.population == 0:
          print("{} was the last one.".format(self.name))
      else:
```

```
            print("There are still {:d} robots working.".format(
                Robot.population))

    # Object Method: say_hi()
    def say_hi(self):
        """Greeting by the robot.
        Yeah, they can do that.""" # Docstrings
        print("Greetings, my masters call me {}.".format(self.name))

    # Class Method: how_many()
    @classmethod
    def how_many(cls):
        """Prints the current population.""" # Docstrings
        print("We have {:d} robots.".format(cls.population))
```

Explanation of Robot Class Setup

object variables

- the `name` variable belongs to the object (it is assigned using `self`) and hence is an *object variable*
- we refer to the *object variable* name using `self.name` notation in the methods of that object
- **NOTE:** an *object variable* with the same name as a *class variable* will hide the class variable!
- instead of `Robot.population`, we could have also used `self.__class__.population` because every object refers to its class via the `self.__class__` attribute

object methods

- in the `say_hi` built-in object method, the robot outputs a greeting
- in the `die` built-in object method, we simply decrease the `Robot.population` count by 1

docstrings

- in this program, we see the use of `docstrings` for classes as well as methods
  - we can access the class docstring using `Robot.__doc__` and the method docstring as `Robot.say_hi.__doc__`

constructor function

- observe that the `__init__` method is used to initialize the `Robot` instance with a name
  - in this method, we increase the population count by 1 since we have one more robot being added
  - also observe that the values of `self.name` is specific to each object which indicates the nature of object variables

class variable

- `population` belongs to the `Robot` class
  - hence is a *class variable*
- thus, we refer to the `population` *class variable* as `Robot.population` and not as `self.population`

class function

- the `how_many` is actually a method that belongs to the class and not to the object
- this means we can define it as either a `classmethod` or a `staticmethod` depending on whether we need to know which class we are part of
  - since we refer to a *class variable*, let's use `classmethod`
- we have marked the `how_many` method as a class method using a decorator
- *decorators* can be imagined to be a shortcut to calling a wrapper function (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the `@classmethod` decorator is the same as calling:

```
how_many = classmethod(how_many)
```

public vs. private attributes

- all class members are public

- ***One exception***: if you use data members with names using the double underscore prefix such as `__privatevar`, Python uses name-mangling to effectively make it a private variable.

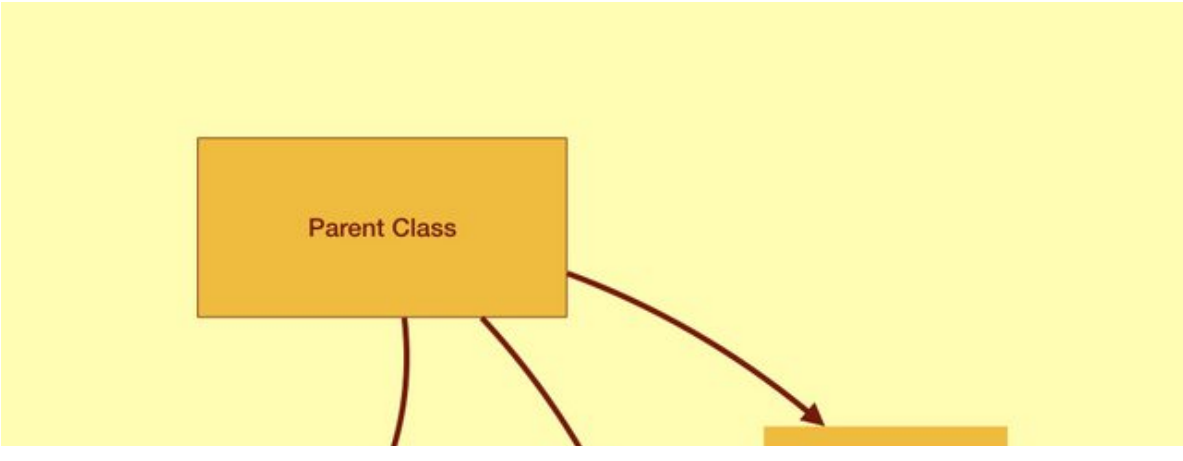
- the convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects
  - remember that this is only a convention and is not enforced by Python (except for the double underscore prefix)

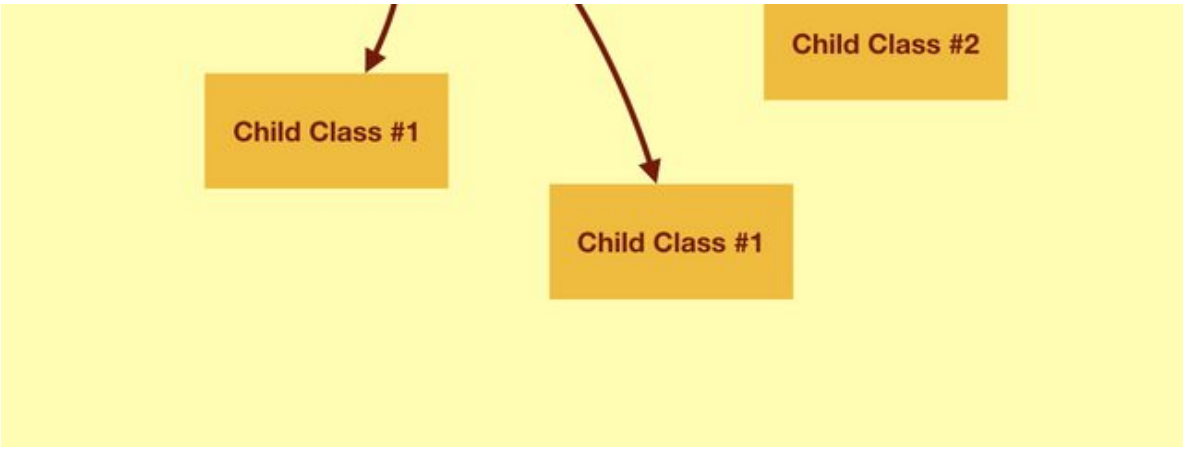
In [ ]:

```
# Using the above Robot Class setup

#-------------------------------------------------------
# Initialize Robot 1 called 'R2-D2' in `droid1`
droid1 = Robot("R2-D2")

# Call the Built-In Object Function for `droid1`
droid1.say_hi()

# Call the Class Function
Robot.how_many()

#-------------------------------------------------------
# Initialize Robot 2 called 'C-3PO'
droid2 = Robot("C-3PO")

# Call the Built-In Object Function for droid2
droid2.say_hi()

# Call the Class Function
Robot.how_many()

#-------------------------------------------------------
# Print a note about Robots working
print("\nRobots can do some work here.\n")

# Print a note about destroying Robots
print("Robots have finished their work. So let's destroy them.")

#-------------------------------------------------------
# Use Built-In Object Functions
droid1.die()
droid2.die()

# Call the Class Function
Robot.how_many()
```

Inheritance

- a major benefit of OOP is reuse of code
  - one way this is achieved is through *inheritance* mechanism
- inheritance can be best imagined as implementing a type (parent class) and subtype (child class) relationship between classes





School Members Example

Problem Statement:

- suppose you want to write a program for a college which has to keep track of
  - *teachers*
  - *students*
- they both have some common characteristics such as
  - name,
  - age and
  - address
- they also have specific characteristics such as
  - for *teachers*
    - salary
    - courses
    - leaves
  - for *students*
    - marks
    - fees

Solution Strategy:

independent classes

- you can create two independent classes for each type (*teachers* and *students*), but adding a new common characteristic would mean adding to both of these independent classes
  - this quickly becomes unwieldy

inherited classes

- a better way would be to create a common class called `SchoolMember` and then have the *teacher* and *student* classes **inherit** from this class
  - i.e. they will become sub-types (child classes) of this type (parent class) and then we can add specific characteristics to these sub-types
- there are many advantages to this approach

1. if we add/change any functionality in `SchoolMember`, this is automatically reflected in the subtypes as well
   - for example, you can add a new ID card field for both *teachers* and *students* by simply adding it to the `SchoolMember` class
   - however, changes in one subtype (child class) does not affect other subtype (child class)
2. another advantage is that you can refer to a *teacher* or *student* object as a `SchoolMember` object which could be useful in some situations such as counting of the number of school members
   - this is called polymorphism where a sub-type can be substituted in any situation where a parent type is expected,
   - i.e. the object can be treated as an instance of the parent class.
3. also observe that we reuse the code of the parent class and we do not need to repeat it in the child

c. also observe that we reuse the code of the parent class and we do not need to repeat it in the child classes as we would have had to in case we had used independent classes

- the `SchoolMember` class in this situation is known as the *base class*, *parent class* or the *superclass*.
- the *Teacher* and *Student* classes are called the *derived classes*, *child class* or *subclasses*

Implementing the Strategy:

Create a Parent Class

- parent class called `SchoolMember`

In [ ]:

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))

    def tell(self):
        '''Tell my details.'''
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")
```

Create a Child Class

- child class `TeacherClass` from Parent `SchoolMember`

In [ ]:

```
class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Initialized Teacher: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Salary: "{:d}"'.format(self.salary))
```

Create a Child Class

- child class `StudentClass` from Parent `SchoolMember`

In [ ]:

```
class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, grade):
        SchoolMember.__init__(self, name, age)
        self.grade = grade
        print('(Initialized Student: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Grade: "{:d}"'.format(self.grade))
```

Initialize a new Teacher Object and Student Object

In [ ]:

```
# initialize `t` - teacher object
```

```
t = Teacher('Mrs. Alyssa', 40, 30000)
# name: Mrs. Alyssa
# age: 40
# salary: 30000

# prints a blank line
print()

# initialize `s` - student object
s = Student('Daniel', 25, 75)
# name: Daniel
# age: 25
# grade: 75

# prints a blank line
print()

members = [t, s]
for member in members:
    # Works for both Teachers and Students
    member.tell()
```

Explanation of Inheritance

- to use inheritance, we specify the parent-class names in a tuple following the class name in the class definition
  - for example, `class Teacher(SchoolMember)`
- next, we observe that the `__init__` method of the parent-class is explicitly called using the `self` variable so that we can initialize the parent class part of an instance in the child-class
  - since we are defining a `__init__` method in *Teacher* and *Student* subclasses, Python does not automatically call the constructor of the child-class `SchoolMember`, you have to explicitly call it yourself
  - in contrast, if we had not defined an `__init__` method in the child-classes, Python will call the constructor of the parent-class automatically
- while we could treat instances of *Teacher* or *Student* as we would an instance of `SchoolMember` and access the tell method of `SchoolMember` by simply typing `Teacher.tell` or `Student.tell`, we instead define another `tell` method in each child-class (using the `tell` method of `SchoolMember` for part of it) to tailor it for that subclass
  - because we have done this, when we write `Teacher.tell`, Python uses the `tell` method for that child-class vs the parent-class
  - however, if we did not have a `tell` method in the child-class, Python would use the `tell` method in the parent-class
  - Python always starts looking for methods in the actual child-class type first
    - and if it doesn't find anything, it starts looking at the methods in the child-class's parent-classes, one-by-one in the order they are specified in the tuple in the class definition
- here we only have 1 base class, but you can have *multiple parent classes*
- a note on terminology - if more than one class is listed in the inheritance tuple, then it is called **multiple inheritance**
- the `end` parameter is used in the `print` function in the parent-class's `tell()` method to print a line and allow the next `print` to continue on the same line
  - this is a trick to make `print` not print a `\n` (newline) symbol at the end of the printing

Encapsulation

- an object's variables should not always be directly accessible
- to prevent accidental change, an object's variables can sometimes only be changed with an objects methods
  - those type of variables are private variables
- the methods can ensure the correct values are set
  - if an incorrect value is set, the method can return an error

- **Python does not have the *private* keyword, unlike some other object oriented languages, but encapsulation can be done**
  - **to achieve this, it relies on the convention**
  - **a class variable that should not directly be accessed should be prefixed with an underscore**

In [ ]:

```
## Encapsulation Example

# initialize a new class
class Robot:

  # constructor function
   def __init__(self):
      self.a = 123 # public object variable
      self._b = 123 # private object variable (by convention - not a hard variable)
      self.__c = 123 # Python enforced private variable

# instantiate a new object
new_robot = Robot()


print(new_robot.a) # publicly accessible
print(new_robot._b) # publicly accessible, but private by convention
print(new_robot.__c) # Python enforced private variable, access throws error
```

- **an attempt to access the Python enforced private variable yields an error as seen above**
- ***single underscore*:**
  - **Private variable, it should not be accessed directly. But nothing stops you from doing that (except convention).**
  - **More specifically, a "protected" variable, if drawing parallels to other programming langauages**
- ***double underscore*:**
  - **Private variable, harder to access but still possible.**
- **Both are still accessible: Python only has private variables by convention.**

Getters and Setters

- **private variables are intended to be changed using *getter* and *setter* methods**
  - **These provide indirect access to them**

In [ ]:

```
# initialize a new Robot class
class Robot:

  # constructor function
  def __init__(self):
    self.__version = 22

  # getter function
  def getVersion(self):
    print(self.__version)

  # setter function
  def setVersion(self, version):
    self.__version = version

# instantiate a new object called 'new_robot'
new_robot = Robot()

# use the getter function
new_robot.getVersion()
```

In [ ]:

```
# use the built in setter function
```

```
new_robot.setVersion(23)

# use the getter function again
new_robot.getVersion()
```

In [ ]:

```
# but try accessing the private variable
print(new_robot.__version) # throws error
```

Encapsulation Boundaries

- private properties do not extend to child classes, or the public object
- protected properties extend to child classes, but not the public object
- when overriding with functions in the child classes,
  - the access level can be same or weaker
  - not stronger
- once the object is created
  - only public attributes and methods can be accessed

Abstract Classes

- an abstract class can be considered as a blueprint for other classes
- it allows you to create a set of methods that *must* be created within any child classes built from the parent abstract class
- a class which contains one or more abstract methods is called an abstract class.
- an abstract method: *a method that has a declaration but does not have an implementation*
- while we are designing large functional units we use an abstract class.
  - when we want to provide a common interface for different implementations of a component, we use an abstract class

Why use Abstract Classes?

- by defining an abstract parent class, you can define a common Application Program Interface (API) for a set of subclasses (chidren classes)
- this capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible

Python and Abstract Classes

- by default, Python does not provide abstract classes
- Python comes with a module which provides the base for defining Abstract Base classes (ABC)
  - that module name is ABC
- ABC works by decorating methods of the parent class as abstract and then registering concrete classes as implementations of the abstract parent
- A method becomes abstract when decorated with the keyword `@abstractmethod`

Abstract Class Example

- define abstact class called `Polygon`
  - with an abstract method called noofsides
  - the abstract method forces this method to be defined again (overridden) in the children classes
- then, define following children classes from the parent `Polygon` class
  - `Triangle`
  - `Quadilateral`

    - `Pentagon`
    - `Hexagon`
- **within each child class, the abstract method `noofsides` is overrideen with a custom method, but with the same name**

In [ ]:

```
# imported abstract base class

from abc import ABC, abstractmethod

# Parent Abstract Method ---------------------------------------------------
class Polygon(ABC):

 # abstract method inside the parent Abstract Class
 def noofsides(self):
  pass

# Child Class #1 of Polygon Parent Abstract Class -------------------------
class Triangle(Polygon):

 # overriding abstract method
 def noofsides(self):
  print("I have 3 sides")

# Child Class #2 of Polygon Parent Abstract Class -------------------------
class Pentagon(Polygon):

 # overriding abstract method
 def noofsides(self):
  print("I have 5 sides")

# Child Class #3 of Polygon Parent Abstract Class -------------------------
class Hexagon(Polygon):

 # overriding abstract method
 def noofsides(self):
  print("I have 6 sides")

# Child Class #4 of Polygon Parent Abstract Class -------------------------
class Quadrilateral(Polygon):

 # overriding abstract method
 def noofsides(self):
  print("I have 4 sides")
```

- **Instantiate objects from the childen classes**

In [ ]:

```
# Create Objects from Children Classes

# Triangle Child Class Object
R = Triangle()
R.noofsides()

# Quadilateral Child Class Object
K = Quadrilateral()
K.noofsides()

# Pentagon Child Class Object
R = Pentagon()
R.noofsides()

# Hexagon Child Class Object
K = Hexagon()
K.noofsides()
```

Concrete Methods in Abstract Base Classes

- **concrete classes contain only concrete (normal) methods whereas abstract classes may contains both concrete methods and abstract methods**
- **concrete classes provide an implementation of abstract methods**
- **the abstract base class can also provide an implementation by invoking the methods via `super()` in the child classes**

In [ ]:

```
# Python program invoking a
# method using super()

from abc import ABC, abstractmethod

# create a parent Abstract Class
class Parent(ABC):
  # define concrete method in abstract class
 def rk(self):
  print("Abstract Base Class")

# create a child Concrete Class
class Child(Parent):
  # extend
 def rk(self):
  super().rk()
  print("subclass ")
```

- **call the child class to instantiate a new object**
- **in the above code, we can invoke the methods in abstract classes by using `super()`**

In [ ]:

```
# create new object from Concrete Class
new_object = Child()
new_object.rk()
```

Abstract Properties

- **abstract classes includes attributes in addition to methods**
- **you can require the attributes in concrete classes by defining them with `@abstractproperty`**

In [1]:

```
# Python program showing abstract properties

import abc
from abc import ABC, abstractmethod

# create a parent Abstract Class
class parent(ABC):
 @abc.abstractproperty # abstract property
 def geeks(self):
  return "parent class"

# create a child Concrete Class
class child(parent):
 @property # reference abstact property
 def geeks(self):
  return "child class"

try:
 new_object = parent()
 print(new_object.geeks)
except Exception as err:
```

```
 print (err)

new_object = child()
print(new_object.geeks)
```

```
Can't instantiate abstract class parent with abstract methods geeks
child class
```

- **in the above example, the Parent class cannot be instantiated because it has only an abstract version of the property getter method**

Abstract Class Instantiation

- **abstract classes are incomplete because they have methods which have no body**
- **if python allows creating an object for abstract classes, then using that object, if anyone calls the abstract method, there is no actual implementation to invoke**
- **so *we use an abstract class as a template* and according to the need we extend it and build on it before we can use it.**
- ***due to the fact that an abstract class is not a concrete class, it cannot be instantiated***
- **when we create an object from the abstract class, it raises an error**

In [ ]:

```
# Python program showing
# abstract class cannot
# be an instantiation

from abc import ABC,abstractmethod

class Animal(ABC):
 @abstractmethod
 def move(self):
  pass

class Human(Animal):
 def move(self):
  print("I can walk and run")

class Snake(Animal):
 def move(self):
  print("I can crawl")

class Dog(Animal):
 def move(self):
  print("I can bark")

class Lion(Animal):
 def move(self):
  print("I can roar")

cat = Animal()
```

Interface

- **it is like abstract classes but allows child classes to implement multiple classes**
- **cannot define variables (data members) in an interface or constructor functions**
  - **only abstract methods**
- **only public functions can be defined, no private and protected functions can be defined**
- **an interface is a set of publicly accessible methods on an object which can be used by other parts of the program to interact with that object**
  - **Interfaces set clear boundaries and help us organize our code better**

informal Interfaces - Dynamic Language and Duck Typing

Informal Interfaces - Dynamic Language and Duck Typing

- There's no explicit `interface` keyword in Python like Java/C++ because Python is a dynamically typed language
  - in the dynamic language world, things are more implicit
  - more focus on how an object behaves, rather than it's type/class
- if we have an object that can fly and quack like a duck, we consider it as a duck
  - this is called "Duck Typing"
- to be safe, we often handle the exceptions in a `try..except` block or use `hasattr` to check if an object has the specific method
- in the Python world, we often hear "file like object" or "an iterable"
  - if an object has a `read` method, it can be treated as a 'file like object'
  - if it has an `__iter__` magic method, it is an iterable
- so any object, regardless of it's class/type, can conform to a certain interface just by implementing the expected behavior (methods)
  - these informal interfaces are termed as *protocols*
  - since they are informal, they can not be formally enforced
  - they are mostly illustrated in the documentations or defined by convention

Informal Interface Example

In [ ]:

```
## Create class called Team
class Team:

  # define constructor
  def __init__(self, members):
      self.__members = members

  # define built-in length method
  def __len__(self):
      return len(self.__members)

  # define membership method
  def __contains__(self, member):
      return member in self.__members

## Instantiate object from Team Class
justice_league_fav = Team(["batman", "wonder woman", "flash"])

## Size protocol
print(len(justice_league_fav))

## Membership protocol
print("batman" in justice_league_fav)
print("superman" in justice_league_fav)
print("cyborg" not in justice_league_fav)
```

- in the above example, by implementing the `__len__` and `__contains__` method, we can now directly use the `len` function on a `Team` instance and check for membership using the `in` and `not in` operators
- so we can see that protocols are like informal interfaces

Formal Interfaces - Abstact Base Classes

- there are situations where informal interfaces or duck typing in general can cause confusion
- for example, a Bird and Aeroplane both can `fly()`
  - but they are not the same thing even if they implement the same interfaces / protocols.
- Abstract Base Classes or ABCs can help solve this issue
- the concept behind ABCs is simple:
  - we define base classes which are abstract in nature
  - we define certain methods on the base classes as abstract methods
  - any objects deriving from these base classes are forced to implement these methods

- any objects deriving from these base classes are forced to implement those methods.
- since we're using base classes, if we see an object has our class as a base class, *we can say that this object implements the interface*
- that is now we can use types to tell if an object implements a certain interface

Formal Interface - Example

- there's the `abc` module which has a metaclass named `ABCMeta`
  - ABCs (Abstract Base Classes) are created from this metaclass
  - we can either use it directly as the metaclass of our ABC i.e. something like this
    - `class Bird(metaclass=abc.ABCMeta):`
    - or we can subclass from the `abc.ABC` class which has the `abc.ABCMeta` as it's metaclass already
- then we have to use the `abc.abstractmethod` decorator to mark our methods abstract

In [2]:

```
import abc

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

- if any class derives from our base `Bird` class, it must implement the fly method too

In [ ]:

```
class Parrot(Bird):
    def fly(self):
        print("Flying")

p = Parrot()

# check if p is an instance of Bird Class
isinstance(p, Bird)
```

- since our parrot is recognized as an instance of Bird ABC, we can be sure from it's type that it definitely implements our desired interface

- let's define another `ABC` named `Aeroplane`:

In [ ]:

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

class Boeing(Aeroplane):
  def fly(self):
    print("Flying!")

b = Boeing()

isinstance(p, Aeroplane)
isinstance(b, Bird)
```

- even though both objects have the same method `fly`
  - we can differentiate easily which one implements the `Bird` interface and which implements the `Aeroplane` interface.

Interfaces and Multiple Base Classes

- **multiple parent abstract classes can be implemented in a child class**

In [6]:

```
## define `apple` and `banana` parent classes
class Apple(abc.ABC):
  @abc.abstractmethod
  def apple_one(self):
    pass

class Banana(abc.ABC):
  @abc.abstractmethod
  def banana_one(self):
    pass

## define child class from Apple and Banana
class Fruits(Apple, Banana):

  def apple_one(self):
    print("Child Class - Apple")

  def banana_one(self):
    print("Child Class - Banana")
```

- **instantiate object and use methods from both parent classes**

In [9]:

```
## instantiate objects

fruit_bowl = Fruits()
fruit_bowl.apple_one() # call function initialized in Apple Base Class
fruit_bowl.banana_one() # call function initialized in Banana Base Class
```

```
Child Class - Apple
Child Class - Banana
```

Reference

- Swaroop's Blog
- Encapsulation in Python
- Abstract Base Classes - Python Docs
- Abstract Classes
- Informal Interface
- Abstract Base Classes for Containers - Python Docs