TEMASEK
JUNIOR COLLEGE

Temasek Junior College
H2 Computing
Object-Oriented Programming in Python

Object-Oriented Programming - in Python

Everything is a class in Python.
Guido van Rossum has designed the language according to the principle "first-class everything". He wrote: "One of my goals for Python was to make it so that all objects were 'first class.' By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth."

This means that "everything" is treated the same way, everything is a class: functions and methods are values just like lists, integers or floats. Each of these are instances of their corresponding classes.

1 Python OOPs

Python is an object-oriented programming language. It allows us to develop applications using Object Oriented approach. In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below:

- Class
- Object
- Method
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

2 Class (ADT/UDT/Virtual object)

- A class is the definition of all the **attributes** and **methods** which are the common aspects of all objects created from it.
- A class is a blueprint for the object. Classes are data structures that the user defines.

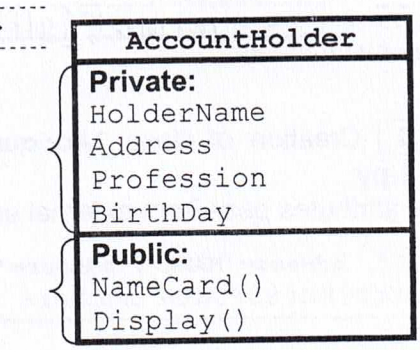
Example 1 A "real-life" class "`AccountHolder`".

Class diagram:

Class names →

Private attributes /Data members

Public methods/ Member functions



2.1 Python Class

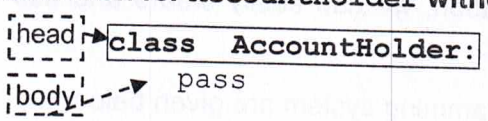
2.2 Define a class in Python

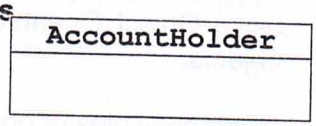
In Python, a class is defined by using a keyword **`class`**.

- Syntax of a class definition:

```
class ClassName :
    <statement-1>
    ::::
    ::::
    <statement-N>
```

- A class creates a new local namespace to define its all attributes. These attributes may be data or functions.
- The fundamental syntactical structure of a class in Python:
  A **class** consists of two parts: the **header** and the **body**.
  - The header usually consists of just one line of code. It begins with the keyword "class" followed by a blank and an arbitrary name for the class. The class name is "**`AccountHolder`**" in this case.
  - The body of a class consists of an indented block of statements.
- **Class definition for `AccountHolder` without data members**





Task 1 Creation of class **'AccountHolder'** with default values in data members **>> Task_1.py**

```
class AccountHolder:
    ## Constructor -- assign default values to data members
    def __init__(self):
        ### Instance attributes/Data members with default values
        self.HolderName = "NAME"
        self.Address    = "ADDRESS"
        self.Profession = "PROFESSION"
        self.BirthDay   = "01/01/1900"
        print(f"+++ Account Holder ('{self.HolderName}') has been created")

    ## Destructor
    def __del__(self):
        print(f"--- Account Holder ('{self.HolderName}') has been destroyed")
        return True
```

Class diagram

Task 1_1 Creation of class **'AccountHolder'** with default values in arguments **>> Task_1_1.py**

Values of attributes pass from external environment via parameters

```
def __init__(self, hdName='NAME', hdAddr='ADDRESS', hdProf='PROFESSION', hdBday='01/01/1900'):
## Instance attributes/Data members
```

2.3 Methods

Method is a function that is associated with an object. In Python, method is not unique to class instances. Any object type (class) can have methods.

- Methods are essentially functions
- Parameters in method: the method is called pass by **object reference**. This is basically an object-oriented way of passing parameters

2.3.1 Defining a method directly inside (indented) of a class definition.

- A method is "just" a function which is defined **inside** of a class.

2.3.2 Method differs from a function only in two aspects:

- It belongs to a class, and it is defined **within** a class
- The first parameter in the definition of a method has to be a **reference** to the **instance**, which called the method. This parameter is usually called "**`self`**".
- "**`self`**" is not a Python keyword.

Notes:

✓ **__double_leading_and_trailing_underscore__**
e.g. `__init__`, `__del__`. "defined" objects or attributes that live in user-controlled namespaces. Never invent such names; only use them as documented.

✓ The connection between the methods with the object is indicated by a "dot" ("•") written between them. E.g. `ah1•getBirthDay()`, `ah1.setBirthDay(newBirthDay)`.

2.3.3 Constructor method `__init__`

- To define the attributes of an instance right after its creation, `__init__()` is a method which is immediately and automatically called after an instance has been created. This name is fixed and it is not possible to choose another name.
- The `__init__()` method is used to initialize an instance. The `__init__()` method can be anywhere in a class definition, but it is usually the first method of a class, i.e. it follows right after the class header

2.3.4 Destructor method `__del__`

- It is called when the instance is about to be destroyed and if there is no other reference to this instance.
- If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.
- The usage of the `__del__()` method is very problematic. Use carefully

Note:

✓ There is no explicit constructor or destructor method in Python.

2.4 Instance/Object attributes

- For other object-oriented language, the terms attributes and properties are usually used synonymously.
- In Python: properties and attributes are essentially different things. So far **`AccountHolder`** have `HolderName, Address, Profession and BirthDay` as instance attributes.
- Attributes are created inside of a class definition.
- **Instance attribute** are defined in the class body using **`self`** keyword usually in the **__init__()** method, e.g. **`self`**•`HolderName`
- Instance attributes are not shared by objects. Every object has its own copy of the instance attribute

3 Objects

- When we define a class only the description or a blueprint of the object is created. There is no memory allocation until we create its **object**. The **object** or **instance** contains real data or information.
- Instantiation is nothing but creating a new object/instance of a class. Let's create the object of the above class we defined

3.1 Python Object

```
class AccountHolder:
  def __init__(self, . . .):
  :::
def main():
    ah1 = AccountHolder()
```

Instantiation: `ah1` is the new object or instance of class `AccountHolder`

Note: **`self`** corresponds to the **`AccountHolder`** object **`ah1`**.

3.2 Some methods for object

`__doc__` – The instances (objects) possess dictionaries `__dict__`, which use to store attributes and corresponding values.

`vars()` – This function displays the attribute of an instance in the form of an dictionary.

`dir()` – This function displays more attributes than vars function, as it is not limited to instance. It displays the class attributes as well. It also displays the attributes of its ancestor classes.

3.3 Objects instantiation

Task 2 Creation of objects for class 'AccountHolder' **>> Task_2.py**

```
# Task_2.py
class AccountHolder:
   ## Constructor -- assign default values to data members
   ## Destructor
            :::::
def main():
   ah1 = AccountHolder()    # ***** Object instantiation *****
   print(ah1)
   print(f"\n 01] ah1 ID={id(ah1)}")
```

```
   print(f" 07] ah1 dictionary elements:
     {ah1Fields['HolderName']}, {ah1Fields['Address']},\
     {ah1Fields['Profession']}, {ah1Fields['BirthDay']}")
   print(f" 08] ah1Fields.values() - {ah1Fields.values()}" )
     ah1List = list(ah1Fields)  ##Convert dict to list
```

```
   ah2= AccountHolder()
   ah3 = ah2
   print(f"\n 10] ah1 ID={id(ah1)},  ah2 ID={id(ah2)},\
         ah3 ID={id(ah3)}")
   print(f"\n 11] ah1 == ah2 : {ah1 == ah2}")
   print(f"\n 12] ah3 == ah2 : {ah3 == ah2}")
## Driver
main()
```

3.4 print() the object using the __str__ method

- **__str__** method represents the class objects as a string – it can be used for classes.
- The __str__ method should be defined in a way that is easy to read and outputs all the members of the class. This method is also used as a debugging tool when the members of a class need to be checked.
- The __str__ method is called when the following functions are invoked on the object and return a string:
  - print()
  - str()

Note:

✓ The connection between the instance attributes with the object is indicated by a "dot" ("•") written between them, e.g. **self•**HolderName.

Task 3 Printing objects for class 'AccountHolder' **>> Task_3.py**

```
# Task_3.py
class AccountHolder:
    ## Constructor -- assign default values to data members
    def __init__(self):
            :::
    def __str__(self):
        
    def __del__ (self):
            :::
def main():
    ah1 = AccountHolder()
    print(ah1)
    print(ah1.__str__())
##Driver
main()
```

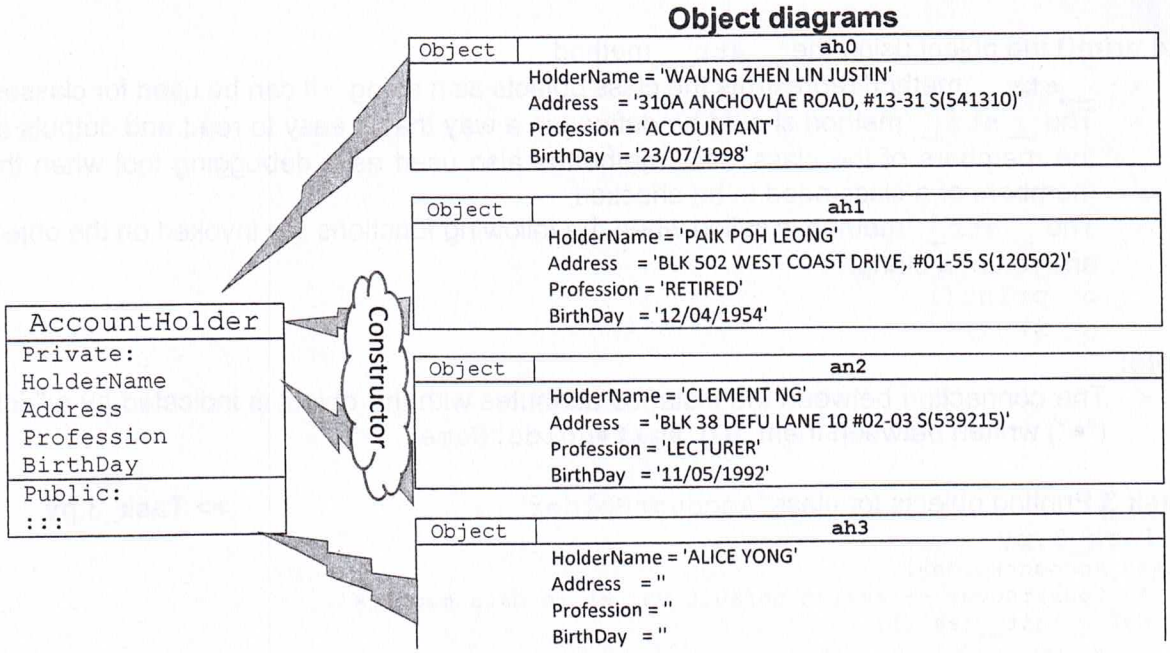
apply in print()

Both calling __str__()

3.5 Assign values to data members by parameters in constructor

Task 4 Values of attributes pass from external environment via parameters **>> Task_4.py**

```
class Employee:
    ## Constructor -- assign default values to data members
    def __init__(self, hdName='NAME', hdAddr='ADDRESS', hdProf='PROFESSION', hdBday='01/01/1900'):
        ### Instance attributes/Data members
        self.HolderName = hdName
        self.Address    = hdAddr
        self.Profession = hdProf
        self.BirthDay   = hdBday
    def __str__(self):
        :::
    def __del__ (self):
        :::
def main():
    ah1 = AccountHolder()    # ***** Object instantiation *****
    print(f"01] object ah1: {ah1}")
    # **** Values of attributes pass from external environment
    ah2 = 
    ah3 = 
    ah4 = 
    ah5 = 
    print(f"02] object ah2: {ah2}")
    print(f"03] object ah3: {ah3}")
    print(f"04] object ah4: {ah4}")
    print(f"05] object ah5: {ah5}")
##Driver
main()
```



3.6 Class member visibility I : Public members

All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

3.6.1 Class member visibility: Public attributes

- **Public** Attributes can and should be freely used.

Task 5 Public attributes **>> Task_5.py**

```
## Task_5.py
## ***** Module: class member visibility: Public attributes
class AccountHolder:
   ## Constructor
   def __init__(self, hdName='NAME', hdAddr='ADDRESS', hdProf='PROFESSION', hdBday='01/01/1900'):
    ### Instance attributes/Data members
       self.HolderName = hdName
       self.Address    = hdAddr
       self.Profession = hdProf
       self.BirthDay   = hdBday
    def __str__(self):
          :::
   def __del__ (self):
          :::
def main():
    ah1 = AccountHolder()    # ***** Object instantiation *****
    print(f"01] object ah1: {ah1}")
    # **** Values of attributes pass from external environment
    ah2 = AccountHolder('WAUNG ... , 'ACCOUNTANT', '23/07/1998')
    print(f"02] object ah2: {ah2}")
    print(f"01] object ah1: {ah1}")
# ***** Driver *****
main()
```

Class diagram

access in main()

3.6.2 Public methods

Task 6 Implement a public `NameCard()` method **>> Task _6.py.**

```
## Task_6.py
## ***** Module: class member visibility: Public method
class AccountHolder:
    ## Constructor
    def __init__(self, hdName='NAME', hdAddr='ADDRESS', hdProf='PROFESSION', hdBday='01/01/1900'):
    :::
   def NameCard(self):
    def __str__(self):
        :::
    def __del__ (self):
        :::
def main():
    ah1 = AccountHolder()    # ***** Object instantiation *****
    print(f"01] object ah1: {ah1}")
    # **** Values of attributes pass from external environment
    ah2 = AccountHolder('WAUNG ... , 'ACCOUNTANT', '23/07/1998')
    print(f"02] object ah2: {ah2}")
    ah1.HolderName = 'AISAH'
    ah1.Address = 'Blk 35 Mandalay Road #13-37 S(308215)'
    ah1.Profession = 'BANKER'
    ah1.BirthDay = '03/08/1980'
    print(f"01] object ah1: {ah1}")
    print(
    print(
# ***** Driver *****
main()
```

access in main()

Class diagram

AccountHolder
Public: HolderName Address Profession BirthDay
Public: **NameCard()** ~~Display()~~

3.7 Array of objects

Task 7 Creation of array of objects of class AccountHolder **>> Task_7.py**

```
## Task_7.py
class AccountHolder:
   ## Constructor
      :::
   ## Destructor
      :::
def main():
   newAH = []          # Get array ready
   LENGTH = 10         # Length of array
   ## Creation of array of objects
##   newAH = [AccountHolder() for i in range (LENGTH)]
## Or
   for i in range(LENGTH):
      
##Display all objects
   for i in range (len(newAH)):
      print(f"\t{i+1:2} { newAH[i]} {id(newAH[i])}")
   newAH[3].HolderName = " AISAH"
   print(f" newAH[3].HolderName: { newAH[3].HolderName}")

   newAH.append(AccountHolder())

   for i in range (len(newAH)):
      print(f"\t{i+1:2} { newAH[i]} {id(newAH[i])}")
   for i in range (len(newAH)):
      print(f"\t{i+1:2}\n { newAH[i].NameCard()}")
## Driver
main()     29/07/2022
```

Task 8 Read data from text file and assign into an array of objects **>> Task_8.py**

```
class AccountHolder:
      :::
def main():
   newAH = []    # Get array ready
   sourceFileName     = "accHolders.DAT"
   accHolderFile      = open(sourceFileName, "r")
   for record in accHolderFile:
      
##Display all objects
   for i in range (len(newAH)):
      print(f"\t{i+1:2} { newAH[i]} {id(newAH[i])}")
   for i in range (len(newAH)):
      print(f"\t{i+1:2}\n { newAH[i].NameCard()}")
# ***** Driver *****
main()
```

Task 8_1 Convert text file data into csv file **>> Task_8_1.py**

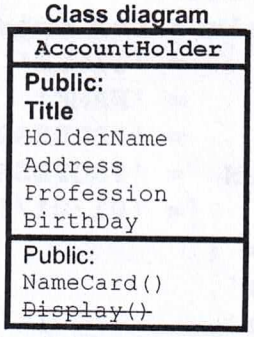
Task 8_2 Read data from csv file and assign into an array of objects **>> Task_2_1.py**

Task 9 Processing with text file. Make changes in Task_8 by adding an attribute 'Title' into class AccountHolder and place it before attribute HolderName. Make the necessary changes in argument (hdTitl) and methods in the class definition. Default value of 'Title' is 'TITLE', other values use the dictionary given. **>>**

Task_9.py

```
# Dictionary for Title
TitleTypes = {
    'T1': 'Mr',
    'T2': 'Ms',
    'T3': 'Mdm',
    'T4': 'Mrs',
    'T5': 'Ms',
    'T6': 'Doctor'
    'TD': 'TITLE'
}
```

Class diagram



Task 9_1 Processing with csv file. **>> Task_9_1.py**

Task 10 With Task_9, implement a public **Display()** method for displaying all attributes with the following format. **>> Task _10.py.**

Account holder record

Title Date of birth

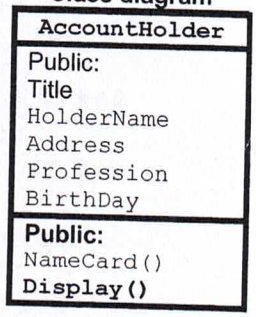
Name

Profession

Address

```
def Display(self): # Use of Dictionary for EmployeeType
  print(f{"Account holder record ":24}
        f"{'-' * 70\n}"}
        f"{'-' * 70}\n}")
  return ''
```

Class diagram



Task 10_1 With Task_9_1, implement a public method **Display()**. **>> Task_10_1.py**

4 Constants used in Python

- Add constants into Task_10.py

Task 11 Using constant names **>> Task _11.py.**

```
## Task_11.py
##Constant section
##Default values for objects for class AccountHolder
HD_TITLE       = 'TITLE'
HD_NAME        = 'NAME'
HD_ADDRESS     = 'ADDRESS'
HD_PROFESSION  = 'PROFESSION'
HD_BIRTHDAY    = '01/01/1900'
TitleTypes = {
    'T1': 'Mr',
    'T2': 'Ms',
    'T3': 'Mdm',
    'T4': 'Mrs',
    'T5': 'Ms',
    'T6': 'Doctor'
    'TD': 'TITLE'
}
class AccountHolder:
    ## Constructor -- assign default values to data members
    def __init__(self, hdTilt=HD_TITLE, hdName=HD_NAME,\
        hdAddr= HD_ADDRESS, hdProf= HD_PROFESSION, hdBday= HD_BIRTHDAY):
### Public Instance attributes/Data members
      :::
def main():
    ah1 = AccountHolder()
    print(ah1)
      :::
## Driver
main()
```

Task 12 Import constant from module with text file processing **>> Task _12.py**

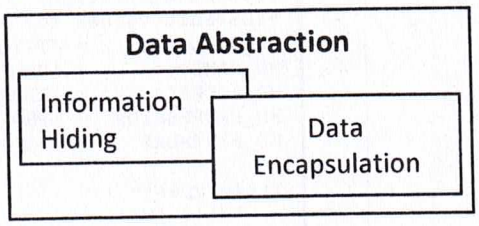
```
## aC.py
##Constant section
##Default values for objects for class AccountHolder
HD_TITLE       = 'TITLE'
HD_NAME        = 'NAME'
HD_ADDRESS     = 'ADDRESS'
HD_PROFESSION  = 'PROFESSION'
HD_BIRTHDAY    = '01/01/1900'

TitleTypes = {
   'T1': 'Mr',
   'T2': 'Ms',
   'T3': 'Mdm',
   'T4': 'Mrs',
   'T5': 'Ms',
   'T6': 'Doctor',
   'TD': 'TITLE'
}
```

```
# Task_12.py
## Using constant names from a module
### Constant section
import aC          #File aC.py keeping constants
class AccountHolder:
   ## Constructor
   def __init__(self, hdTilt=aC.HD_TITLE, hdName=aC.HD_NAME, \
                hdAddr=aC.HD_ADDRESS, hdProf=aC.HD_PROFESSION,\
                hdBday=aC.HD_BIRTHDAY):
  ### Public Instance attributes/Data members
      :::
   def Display(self): # Use of Dictionary for TitleTypes
      print(f"{'Account holder record ':24}\n"
            f"{'-' * 70}\n"
            f"{'Title: ' + aC.TitleTypes[self.Title]:34}"
            f"{'Date of birth: ' + self.BirthDay:>34}\n"
            f"{'Name: ' + self.HolderName:30}\n"
            f"{'Profession: ' + self.Profession:22}\n"
            f"{'Address: ' + self.Address:18}\n"
            f"{'-' * 70}")
      return ''
def main():
   e1 = AccountHolder()
      :::

##Driver
main()
```

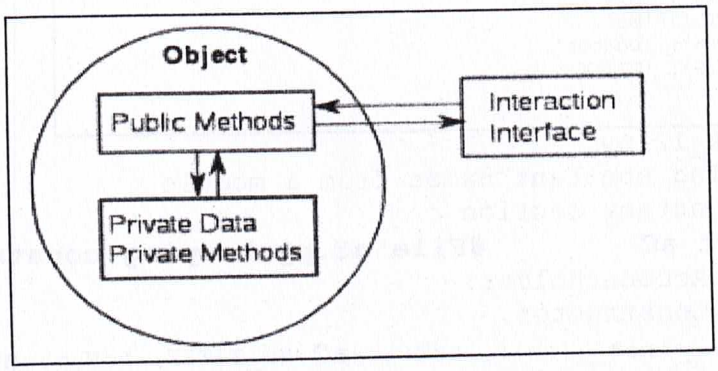
Task 12_1 Import constant from module with csv file processing **>> Task _12_1.py**

5 Data Abstraction, Data Encapsulation, and Information Hiding

Data Abstraction = Data Encapsulation + Data Hiding



- **Encapsulation** It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.
- **Information (Data) hiding** is the principle that some internal information or data is "hidden", so that it can't be accidentally changed.



5.1 Private attribute of objects/instances

- **Private** attributes should only be used by the owner, i.e. inside of the class definition itself.
- To restrict the access to class private attributes:
  Prefix an attribute name with **two** leading underscores "__".
  - The attribute is now inaccessible and invisible from outside.
  - It's neither possible to read nor write to those attributes except inside of the class definition itself.

Notes:

✓ **double underscore prefix**

e.g. `__EmployeeID`, `__Name`, `__TelephoneNumber` and `__EmployeeType` causes the Python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses. This is also called name mangling—the interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later.

Task_13 Demonstration of the behaviour of private attribute types. **>> Task_13.py**

```
## Task_13.py
## Demo of Private attributes
## Using constant from a module
####Constant section
import aC          #Module aC.py keeping constants
class AccountHolder:
   ## Constructor
   def __init__(self, hdTilt=aC.HD_TITLE, hdName=aC.HD_NAME, \
                hdAddr=aC.HD_ADDRESS, hdProf=aC.HD_PROFESSION,\
                hdBday=aC.HD_BIRTHDAY):
      ### Instance attributes/Data members
      self.__Title     = hdTilt  ##Private
      self.HolderName = hdName  ##Public
      self.Address     = hdAddr  ##Public
      self.Profession = hdProf  ##Public
      self.BirthDay    = hdBday  ##Public

   def NameCard(self):
        return f"{'='*50}\n"\
               f"\t{self.__Title}. {self.HolderName}\n"\
               f"\t{self.Address}\n\t{self.Profession}\n"\
               f"{'='*50}"
   def __str__(self):
        return f"Title: {self.__Title}\t" \
               f"Name: {self.HolderName}\t" \
               f"|Address:{self.Address} \t|Profession: {self.Profession}\t "\
               f"|BirthDay: {self.BirthDay} "    ## Destructor
   def __del__(self):
      return True

   def Display(self): # Use of Dictionary for TitleTypes
      print(f"{'Account holder record ':24}\n"
            f"{'-' * 70}\n"
            f"{'Title: ' + aC.TitleTypes[self.__Title]:34}"
            f"{'Date of birth: ' + self.BirthDay:>34}\n"
            f"{'Name: ' + self.HolderName:30}\n"
            f"{'Profession: ' + self.Profession:22}\n"
            f"{'Address: ' + self.Address:18}\n"
            f"{'-' * 70}")
      return ''
```

AccountHolder
Private: Title HolderName Address Profession BirthDay
Private: Constructor() Destructor() **Public:** NameCard() **Display()**

Title of **private** attributes with **double underscore prefix**

```
def main():
   newAH = [] # Get array ready
   sourceFileName = "dataSource/accHolders.csv"
   ahCsvFile = open(sourceFileName, 'r', newline='', encoding='UTF8')
   ahRecords = csv.reader(ahCsvFile)
   next(ahRecords) #skipping header
   # Iterate over each row in the csv using reader object
   for ahrow in ahRecords:
      newAH.append(AccountHolder(ahrow[0], ahrow[1], ahrow[2], ahrow[3], ahrow[4]))
   ahCsvFile.close()
   print('1]', newAH[2])
   newAH[2].HolderName = 'Chang Pin Thong'
   print('2]', newAH[2])
   print('3]', newAH[2].__Title)
   newAH[2].__Title = 't4'
   print('4]', newAH[2])

# ***** Driver *****
main()
```

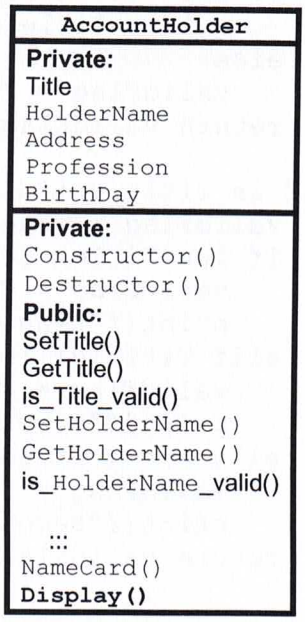
- Different situation in the design of object-oriented programming languages:
  - The first decision to take is how to protect the data which should be **private**.
  - The second decision is what to do if **accessing** or **changing private data** occurs. Of course, the private data may be protected in a way that it can't be accessed under no circumstances. This is hardly possible in practice.
- Encapsulation is often accomplished by providing two kinds of methods for attributes:
  - The methods for retrieving or accessing the values of attributes are called getter (accessor) methods. Getter methods do not change the values of attributes; they just return the values.
  - The methods used for changing the values of attributes are called setter (mutator) methods.
- **Public instead of Private Attributes**
  Summary of the usage of private and public attributes, getters and setters and **properties**:
  Assume that a new class with an instance or class attribute "`Title`", will be needed for the design of the class. Observe the following issues:
  - Will the value of "`Title`" be needed by the possible users of our class?
  - If not, can or should make it a private attribute.
  - If it has to be accessed, make it accessible as a **public attribute**
  - Define it as a **private attribute** with the corresponding **property**, if and only if we have to do some **checks** or **transformation** of the data. (For class `Employee`, where the attribute "`Title`" has to be **fixed at 2 characters, the first character is an upper case letter 'T' and the remaining is digit**, which is ensured by the **property** "`SetTitle()`")

```
## Task_14.py (Continue)
    def NameCard(self):
        :::
    def __str__(self):
        :::
    ## Destructor
        :::
    def Display(self):
        :::
def main():
    newAH = []    # Get array ready
    sourceFileName = "dataSource/accHolders.csv"
    ahCsvFile = open(sourceFileName, 'r', newline='', encoding='UTF8')
    ahRecords = csv.reader(ahCsvFile)
    next(ahRecords) #skipping header
    # Iterate over each row in the csv using reader object
    for ahrow in ahRecords:
        newAH.append(AccountHolder(ahrow[0], ahrow[1], ahrow[2], ahrow[3], ahrow[4]))
    ahCsvFile.close()
    print('1]', newAH[2])
    newAH[2].setTitle('T4')
    print('2]', newAH[2].getTitle())
    newAH[2].setTitle('s')
    print('3]', newAH[2].getTitle())
    newAH[2].setTitle('TT2')
    print('4]', newAH[2].getTitle())
    newAH[2].setTitle('Tb')
    print('5]', newAH[2].getTitle())
    newAH[2].setTitle('t9')
    print('6]', newAH[2].getTitle())
# ***** Driver *****
main()
```

Task_15 Complete all Getters and Setters for private attributes of `AccountHolder`

>> Task_15.py

Given:
HolderName: Length between 3 to 66 characters
Address: Length between 20 to 66 characters
Profession: Length between 10 to 40 characters
BirthDay: Format 'DD/MM/YYYY'
- DD: 01 TO 31
- MM: 01 TO 12
- YYYY: 1900 TO current year
- Check for leap year: Feb, 29 days



Task_14 Demonstration of Getter and Setter for private attribute types. **>> Task_14.py**

```
# Task_14.py
# Demo of Getter and Setter methods for private attribute
# and validation check for Title
## Using constant from a module
### Constant section
import ec         #Module ec.py keeping constants
### Validation check requirements for Title
TITLE_LENGTH    = 2
TITLE_1ST_CHAR = 'T'
TITLE_2ND_RANGE = '0123456789'

class AccountHolder:
   ## Constructor -- assign default values to data members
   def __init__(self, hdTilt=aC.HD_TITLE, \
                      hdName=aC.HD_NAME, \
                      hdAddr=aC.HD_ADDRESS, \
                      hdProf=aC.HD_PROFESSION,\
                      hdBday=aC.HD_BIRTHDAY):
      ### Instance attributes/Data members
      self.__Title = aC.HD_TITLE      #Private, set default value
      if hdTilt != aC.HD_TITLE:
         self.setTitle(hdTilt)

      self.HolderName = hdName         ##Public
      self.Address    = hdAddr         ##Public
      self.Profession = hdProf         ##Public
      self.BirthDay   = hdBday         ##Public

   #---- Accessor (Getter) methodS
   def getTitle(self):
      return self.__Title
   #---- Mutator (Setter) methods
   def setTitle(self, newTitle):
      validFlag = True
      if self.is_Title_valid(newTitle)== True:
         self.__Title = newTitle
      else:
         validFlag = False
      return validFlag

   def is_Title_valid(self, Title): #Validation checks for __Title
      validFlag = True
      if len(Title) != TITLE_LENGTH:              ##Length check
         validFlag = False
         print(f"Length of Title must be 2!")
      elif Title[0] != TITLE_1ST_CHAR:            ##Format check
         validFlag = False
         print(f"First character must be T!")
      elif not ( Title[1] in TITLE_2ND_RANGE):    ##Format check
         validFlag = False
         print(f"Second character must be one of 0 to 9!")
      return validFlag
```

AccountHolder
Private: Title **Public:** HolderName Address Profession BirthDay
Private: Constructor() Destructor() **Public:** SetTitle() GetTitle() is_Title_valid() ::: NameCard() **Display()**

5.3 Creating module AccountHolderClass.py for class `AccountHolder` and import the class from this file.

Task_16 Save class `AccountHolder` to AccountHolder.py **>> Task_16.py**

```
# Task_16.py
import csv
from AccountHolderClass import AccountHolder
def main():
    newAH = []    # Get array ready
    sourceFileName = "dataSource/accHolders.csv"
    ahCsvFile = open(sourceFileName, 'r', newline='', encoding='UTF8')
    ahRecords = csv.reader(ahCsvFile)
    next(ahRecords) #skipping header
    # Iterate over each row in the csv using reader object
    for ahrow in ahRecords:
        newAH.append(AccountHolder(ahrow[0], ahrow[1], ahrow[2], ahrow[3], ahrow[4]))
    ahCsvFile.close()
    ##Display all objects
    for i in range (len(newAH)):
       print(f"{i+1:2}")
       print(f"{ newAH[i].Display()}")
    for i in range (len(newAH)):
       print(f"{i+1:2}")
       print(f"{ newAH[i].NameCard()}")
##***** Driver *****
main()
```

Task_17 File status checking **>> Task_17.py**

```
from AccountHolderClass import AccountHolder
def FileChecking(sourceFileName):          ##This is not a method
    fileStatus = FILE_NOTFOUND
    if os.path.exists(sourceFileName):
         FileForCheck = open(sourceFileName,"r")
         print(f"\n\t *** File {sourceFileName} was found ****)
         if os.path.getsize(sourceFileName) == 0:
              print(f"\n\t *** File {sourceFileName} is empty  ***")
              fileStatus = FILE_EMPTY
          else:# File is not empty, can read or append records or overwite file
             fileStatus = FILE_EXIST
          FileForCheck.close
    else:
         print(f"\n\t *** File {sourceFileName} not found ***")
         fileStatus = FILE_NOTFOUND
    return fileStatus
def main():
   newAH = []      # Get array ready
   sourceFileName = "dataSource/accHolders.csv"
   fileStatus = FileChecking(sourceFileName)
   if (fileStatus == FILE_EXIST):
      ahCsvFile = open(sourceFileName, 'r', newline='', encoding='UTF8')
      :::
   elif fileStatus == FILE_NOTFOUND or fileStatus == FILE_EMPTY:
      print("File is not avaliable.")
##***** Driver *****
main()
```

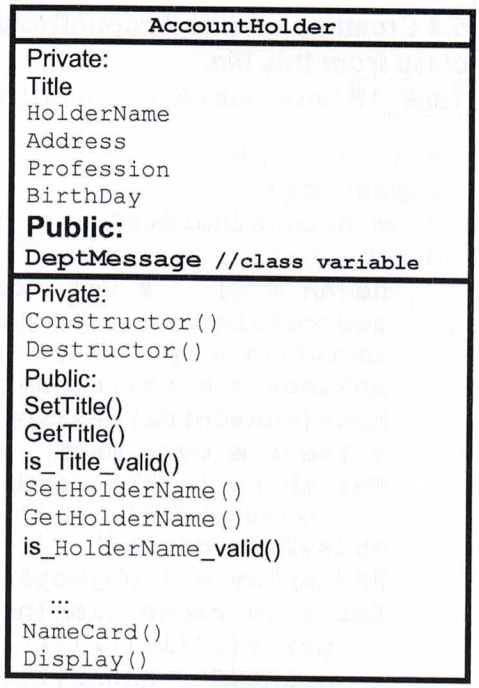
Task_17_1 Creating File status checking module [**fileOperations.py**]. **>> Task_17_1.py**

6 Class Attributes/variables

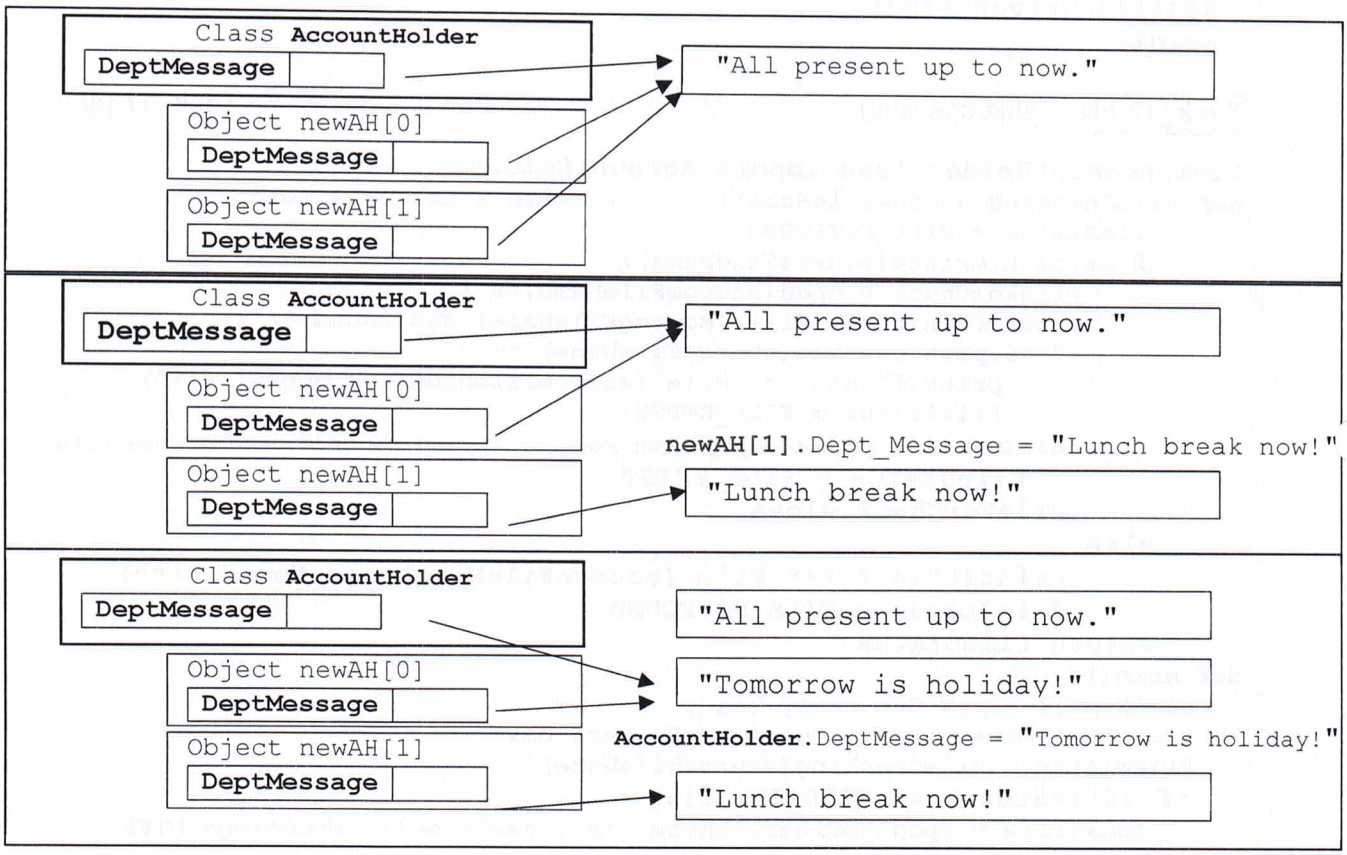
- Instance attributes are owned by the specific instances of a class.
- For two different instances the instance attributes are usually different.

6.1 Defining attributes at the class level

- Class attributes are attributes which are owned by the class itself.
- They will be shared by all the instances of the class.
- They have the same value for every instance.
- To change a class attribute, do it with the notation **ClassName•AttributeName**. Otherwise, will create a new instance variable.
- Defining class attributes outside of all the methods, usually placed at the top, right below the class header.
- Don't change the value in the instance level if not necessary.



In the following Python program that the class attribute "**Dept_Message**" is the same for all instances, e.g. "ah1" and "ah2" etc. Besides this, we see that we can access a class attribute via an instance or via the class name:



Task_18 Demonstration of the class attributes **>>Task_18.py**

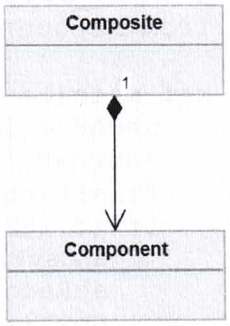
```
# AccountHolderClass.py
class AccountHolder:
    #Class attribute: Dept_Message
    DeptMessage = "All present up to now."

 # Task_18.py
 from AccountHolderClass import AccountHolder

 def main():
     newAH = []    # Get array ready
     sourceFileName = "dataSource/accHolders.csv"
     fileStatus = FileChecking(sourceFileName)
     if (fileStatus == FILE_EXIST):
         ahCsvFile = open(sourceFileName, 'r', newline='', encoding='UTF8')
         ahRecords = csv.reader(ahCsvFile)
         next(ahRecords) #skipping header
         # Iterate over each row in the csv using reader object
         for ahrow in ahRecords:
             newAH.append(AccountHolder(ahrow[0], ahrow[1], ahrow[2], ahrow[3], ahrow[4]))
         ahCsvFile.close()
         print(f"00]{AccountHolder.DeptMessage}")
         ##Display some objects
         for i in range (3):
             print(f"{i+1:2}")
             print(f"{ newAH[i].Display()} {newAH[i].DeptMessage}")
         for i in range (4,10):
             print(f"{i+1:2}")
             print(f"{ newAH[i].Display()} {AccountHolder.DeptMessage}")
         newAH[1].DeptMessage = "Lunch break now!"
         for i in range (3):
             print(f"{i+1:2}")
             print(f"{ newAH[i].Display()} {newAH[i].DeptMessage}")
         AccountHolder.Dept_Message = "Tomorrow is holiday!"
         for i in range (3):
             print(f"{i+1:2}")
             print(f"{ newAH[i].Display()} {newAH[i].DeptMessage}")

     elif fileStatus == FILE_NOTFOUND or fileStatus == FILE_EMPTY:
         print("File is not avaliable.")
##***** Driver *****
main()
```

7 Composition (Has-A relation)

Composition is a concept that models a **has a** relationship. It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component. This relationship means that a Composite **has a** Component.

- **UML represents composition**

Composition is represented through a line with a diamond at the composite class pointing to the component class. The composite side can express the cardinality of the relationship. The cardinality indicates the number or valid range of Component instances the Composite class will contain.
In the diagram , the 1 represents that the Composite class contains one object of type Component. Cardinality can be expressed in the following ways:

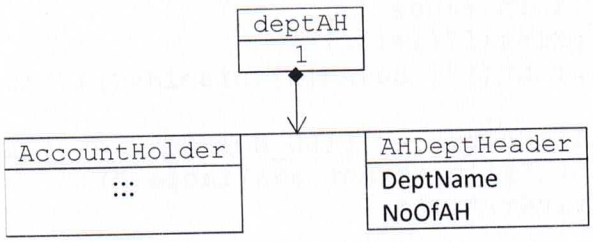


- **A number** indicates the number of Component instances that are contained in the Composite.
- **The * symbol** indicates that the Composite class can contain a variable number of Component instances.
- **A range 1..4** indicates that the Composite class can contain a range of Component instances. The range is indicated with the minimum and maximum number of instances, or minimum and many instances like in **1..***.

Note: Classes that contain objects of other classes are usually referred to as composites, where classes that are used to create more complex types are referred to as components.

Composition enables you to reuse code by adding objects to other objects, as opposed to inheriting the interface and implementation of other classes.

Task 19 Demonstration of composition. Object of Class `deptAH object` contains `AHDeptHeader and AccountHolder` objects. **>> Task_19.py**



Task 19_1 Demonstration of composition. Further improvement. **>> Task_19_1.py**

Tutorial 1 OOP in Python – Task 20, 20_1, 20_2

Task 20 Class `PassengerVehicle`. **>> Task_20.py**

Task 20_1 BubbleSortIndexing. **>> Task_20_1_BubbleSortIndexing.py**

Task 20_2 BubbleSortObjects. **>> Task_20_2_BubbleSortObjects**

8 Inheritance (Is-A Relationship)

Inheritance is a feature of object-oriented programming. It specifies that one object acquires all the properties and behaviours of parent object. By using inheritance, you can define a new class with a little or no changes to the existing class. The new class is known as derived class or child class and from which it inherits the properties is called base class or parent class. It provides re-usability of the code.

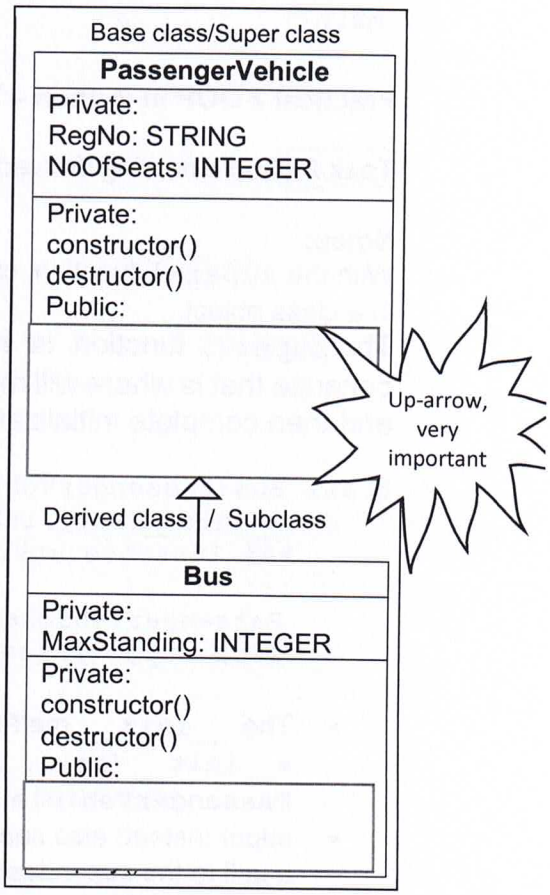
8.1 Inheritance in Python

- Python supports inheritance, it even supports multiple inheritance. Classes can inherit from other classes.
- A class can inherit attributes and behaviour methods from another class, called the superclass.
- A class which inherits from a superclass is called a subclass, also called heir class or child class. Superclasses are sometimes called ancestors as well.
- There exists a hierarchy relationship between classes.
- **Abstract class**: a base class that is never used to create objects directly

The syntax for a subclass definition looks like this:

```
class BaseClass:
      Body of base class
class DerivedClass(BaseClass):
 Body of derived class
```

Example 2 [Refer to Practical 1 OOP in Python for class `PassengerVehicle`]
A transport company has a number of vehicles which can carry passengers. Each vehicle is classified either as a bus or as a coach. All vehicles have a registration number and have a certain number of seats for the passengers. A bus can have a maximum number of standing passengers.



Class diagram for Is-A relationship between vehicles (class `PassengerVehicle`) and buses (class `Bus`).

Inheritance, also called generalization, allows us to capture a hierarchal relationship between classes and objects. For instance, `PassengerVehicle` is a generalization of `Bus`. Inheritance is very useful from a code reuse perspective.

For example, to represent information about `Bus` we recognize that they have a lot in common so generalise/categorise them both as `PassengerVehicle`.

Inheritance produces a hierarchical structure. In this scenario, `PassengerVehicle` could be described as a **base class**, **super class** or **parent class**, as it is the main class from which other class `Bus` are created.

Classes that inherit from others are called **subclasses**, **derived classes** or **child classes**. This example has been simplified to include just one type of `PassengerVehicle`, but the same principle can now be used to define further subclasses.

8.2 Derived class/Subclass

Implementing **base class/superclass `PassengerVehicle`**

Refer to **Task 20** Class **`PassengerVehicle`**.

Task 21 Implementing derived class/subclass **`Bus`** (BusClass.py) **>> Task_21.py**
The transport company has a bus with registration number `'SG3072K'`. The bus has seats for 61 passengers and is allowed to carry 10 standing passengers.

Write program code to:
(i) create an instance of an object with identifier `Bus1` that has the properties of the Bus.
(ii) demonstrate the successful creation of the object by displaying its property values.

```
## Task_21.py
from PassengerVehicleClass import PassengerVehicle
from BusClass import Bus
def main():
##Object Bus1, initialised with base and derived classes method __init__
   Bus1 = Bus('SG3072K', 61, 10)
##callng base and derived classes method __str__
   print(f"01]\tObject Bus1: {Bus1}")
##callng base and derived classes method display
   Bus1.display()
## calling inherited method
   print(f"03]\tReg. no.: {Bus1.getRegNo()}")
:::
#Driver
main()
```

Practical 2 OOP in Python – Task 22, Task 22_1

Task 22 Implementing derived class/subclass `Coach` (CoachClass.py) **>> Task_22.py**

Notes:
With the `super()` function, can gain access to inherited methods that have been overwritten in a class object.
The `super()` function is most commonly used within the `__init__()` method because that is where will most likely need to add some uniqueness to the child class and then complete initialization from the parent.

```
class Bus(PassengerVehicle): #Inherited from PassengerVehicle
   def__init__(self,inRegNo,inNoOfSeats,inMaxStanding):
      ### Instance attributes/Data members
      :::
       PassengerVehicle.__init__(self, inRegNo, inNoOfSeats)
       ##or super(PassengerVehic,self).__init__(inRegNo, inNoOfSeats)
```

- The **`__init__`** method of **`Bus`** class explicitly invokes the **`PassengerVehicle •__init__(self, inRegNo, inNoOfSeats, inMaxStanding)`** method of the **`PassengerVehicle`** class.
- super instead also can be used. **`super().__init__()`** is automatically replaced by a call to the superclasses method, in this case **`PassengerVehicle •__init__`**.

8.3 Overriding and Overloading

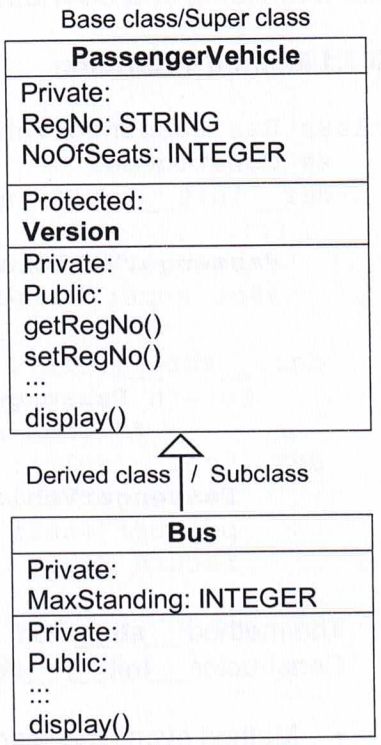
8.3.1 Method overriding

```
class Bus(PassengerVehicle): #Inherited from PassengerVehicle
   ## Constructor
   def__init__(self,inRegNo,inNoOfSeats,inMaxStanding):
      :::
      PassengerVehicle.__init__(self, inRegNo, inNoOfSeats)
      ##or super(PassengerVehic,self).__init__(inRegNo, inNoOfSeats)

    def __str__(self):
        return PassengerVehicle.__str__(self) + \
               f" |Max no. standing :{self.__MaxStanding}"
    def __del__(self):
         PassengerVehicle.__del__(self)
         print(f"{self.getRegNo()}: Destroying in Bus class.")
         return True
```

The method **__str__** from PassengerVehicle is overridden in Bus.
Constructor **__init__**, **__del__** are overridden also.

- Method overriding is an object-oriented programming feature that allows a subclass to provide a different implementation of a method that is already defined by its superclass or by one of its superclasses.
- The implementation in the subclass overrides the implementation of the superclass by providing a method with the same name, same parameters or signature, and same return type as the method of the parent class.
- Overriding means if there are two same methods present in the superclass and the subclass, the contents of the subclass method are executed.
- Overwriting is not a different concept but usually a term wrongly used for overriding!

9 Class member (Instance attributes) visibility – Summary of Public, Protected, and Private Attributes



Attributes classification in object-oriented programming:

- **Public** attributes can and should be **freely** used.
- **Private** attributes should only be used by the owner, i.e. inside of the class definition itself.
- **Protected (restricted)** attributes may be used, but at your own risk. Essentially, this means that they should only be used under certain conditions.
- Python uses a special **naming scheme** for attributes to control the accessibility of the attributes.
- There are two ways to restrict the access to class attributes:
  - First, **prefix** an attribute name with **two** leading underscores "__".
    - The attribute is now **inaccessible and invisible from outside**.
    - It's **neither possible to read nor write** to those attributes **except inside** of the class definition itself.
    - `__RegNo`, `__NoOfSeats` etc are attribute names of objects `pv1` and `pv2`, which **cannot** be freely used inside or outside of a class definition. This corresponds to private attributes of class `PassengerVehicle`.
  - Second, **prefix** an attribute name with **a** leading underscore "_".
    - This marks the attribute as **protected,** e.g., **_Version**.
    - It tells users of the class not to use this attribute unless, somebody writes a **subclass**.
    - should not access such attributes from outside the class.

To summarize the attribute types:

Naming	Type	Meaning
name	**Public**	These attributes can **be freely used inside or outside** of a class definition.
_name	**Protected**	Protected attributes should not be used outside of the class definition, **unless inside of a subclass definition**.
__name	**Private**	This kind of attribute is **inaccessible and invisible**. It's neither possible to read nor write to those attributes, **except inside** of the class definition itself.

10 Polymorphism

- Polymorphism is made by two words "poly" and "morphs". Poly means many and Morphs means form, shape. It defines that one task can be performed in different ways.
- The general description of polymorphism is that it allows different objects to respond to the same message in different ways, the response specific to the type of object.
- This is achieved by various types of 'overloading'-allowing a symbol (an operator like '+', or a function name like show,) to have more than one meaning depending on the context in which it is used.

10.1 Polymorphism in Python

Python can have polymorphic functions or methods. Polymorphic functions or methods can be applied to arguments of different types, and can behave differently depending on the type of the arguments to which are applied. Also define the same function name with a varying number of parameter.

Polymorphic method `display()` in `Bus` and in `Coach` subclasses.

Practical 3 OOP in Python – Task 23

TEMASEK JUNIOR COLLEGE

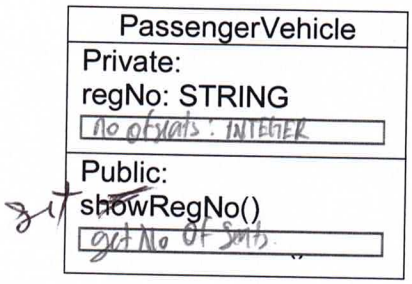
Temasek Junior College
H2 Computing
Object-Oriented Programming in Python

Practical

Tutorial 1 OOP in Python

1 A transport company has a number of vehicles which can carry passengers. Each vehicle is classified either as a bus or as a coach. All vehicles have a registration number and have a certain number of seats for the passengers.

(a) Complete the class diagram with appropriate methods.



(b) Write program code in Python for the PassengerVehicle class and store in passengerVehicle.py.

In the `main()` function test the class:

```
def main():
      newVehicle = PassengerVehicle('SC3418Z', 5)
      #Display regNo

          #Display other attributes
main()
```

(c) PassengerVehicleSize.txt stored a set of passenger vehicles data.
Read these data into an array of objects of class PassengerVehicle and validate the following:
(i) Registration number is begun with two letters in upper case and start with 'S', followed by 1 to 4 digits, and ended with a check digit which is a letter in upper case.
(ii) The number of seats in between 60 to 120.

(d) For those vehicles with error(s) in data display the appropriate error message.