Klaudia Lubicka

OOP - Theory & Examples | Activities part #2

access modifiers
getters / setters

inheritance
method overriding
method overloading

abstract classes and interfaces

INHERITANCE

Inheritance is one of the four main principles of OOP. It allows the programmer to extend the functionality of a class by creating a *parent-child* relationship between classes. In such a relationship, the *child* (also called *sub* or *derived* class) inherits from the *parent* (also called *super* or *base* class). The reader should note that these terms may be used interchangeably based on the context of each discussion. Inheritance is extremely useful, as it facilitates *code reusability*, thus minimizing code and making it easier to maintain. An important concept relating to child classes is that they may have their own new attributes and methods, and can optionally *override* the functionality of the respective parent class.

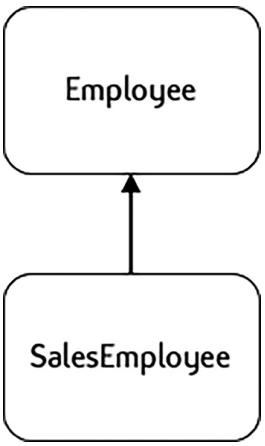
Inheritance:
Allows the extension of the functionality of a *parent/super/base* class, by creating a *child/sub/derived* class that inherits its attributes and behavior.

Inheritance in Python

The Python syntax for implementing the concept of inheritance is the following:

```
Class Parent:
Parent class definition
Class Child(Parent):
Child class definition
```

As a practical example of inheritance, the reader can consider two classes, a super class named `Employee` and a sub class named `SalesEmployee`. Instead of creating the general attributes of `SalesEmployee` (e.g., *first name*, *last name*, *salary*, *or allowances*) from scratch, they can be inherited from `Employee`. Accordingly, the sub class can also inherit the setters and getters, and generally all the *functionality* of the `Employee` class. Additional attributes that may be unique to `SalesEmployee` (e.g., *commission rate*) can be also added to the inherited ones, as required.



Parent-child relationship between classes.

The implementation of this particular example of super class `Employee` and sub class `SalesEmployee` is presented in the Python script examples below. In the first script, `Employee` class is defined with private attributes `__first`, `__last`, `__salary`, and `__allowances`, and class method `getTotalSalary()`. In the second, `SalesEmployee` class is created as an empty class, hence the use of the `pass` keyword. Private attributes and the method are inherited from the `Employee` class. Note that the name of super class `Employee` is passed to `SalesEmployee` as an argument:

```
# Define class 'Employee' and its private attributes and method
class Employee():

        def __init__(self, first, last, salary, allowances):
               self.__first = first
               self.__last = last
               self.__salary = salary
               self.__allowances = allowances

        def getTotalSalary(self):
               return self.__salary + self.__allowances

# Create object 'emp1' and print the total salary of the current employee
emp1 = Employee("George", "White", 16000, 5200)
print(emp1.getTotalSalary())
```

```
21200
```

```
# Define sub class 'SalesEmployee' based on super class 'Employee'
class salesEmployee(Employee):
        pass

""" Create a new object of the sub class that inherits
attributes and behavior from the super class """
semp1 = salesEmployee("Alex", "Flora", 12000, 4000)
print(semp1.getTotalSalary()) # Method of the superclass is invoked
```

```
16000
```

When the `semp1` object is instantiated, Python scans `SalesEmployee` for an initialization method (i.e., `__init__()`). If this is not found, it scans and executes the initialization method of the super class (i.e., `Employee`), with the parameters associated with the current object. Similarly, when `getTotalSalary()` is invoked for object `semp1`, the method is called from the super class, since it does not exist in the sub class. The same order of resolution is followed for all methods and attributes in the sub class.

Customizing the Sub Class

As mentioned, sub classes can be further customized by adding new attributes and methods. For instance, in the case of sub class `SalesEmployee` this can be done by adding attribute `commission_percent`. The reader should note that attempting to use the added attribute for an object that belongs to the `Employee` class will raise an error. This is because there is no such

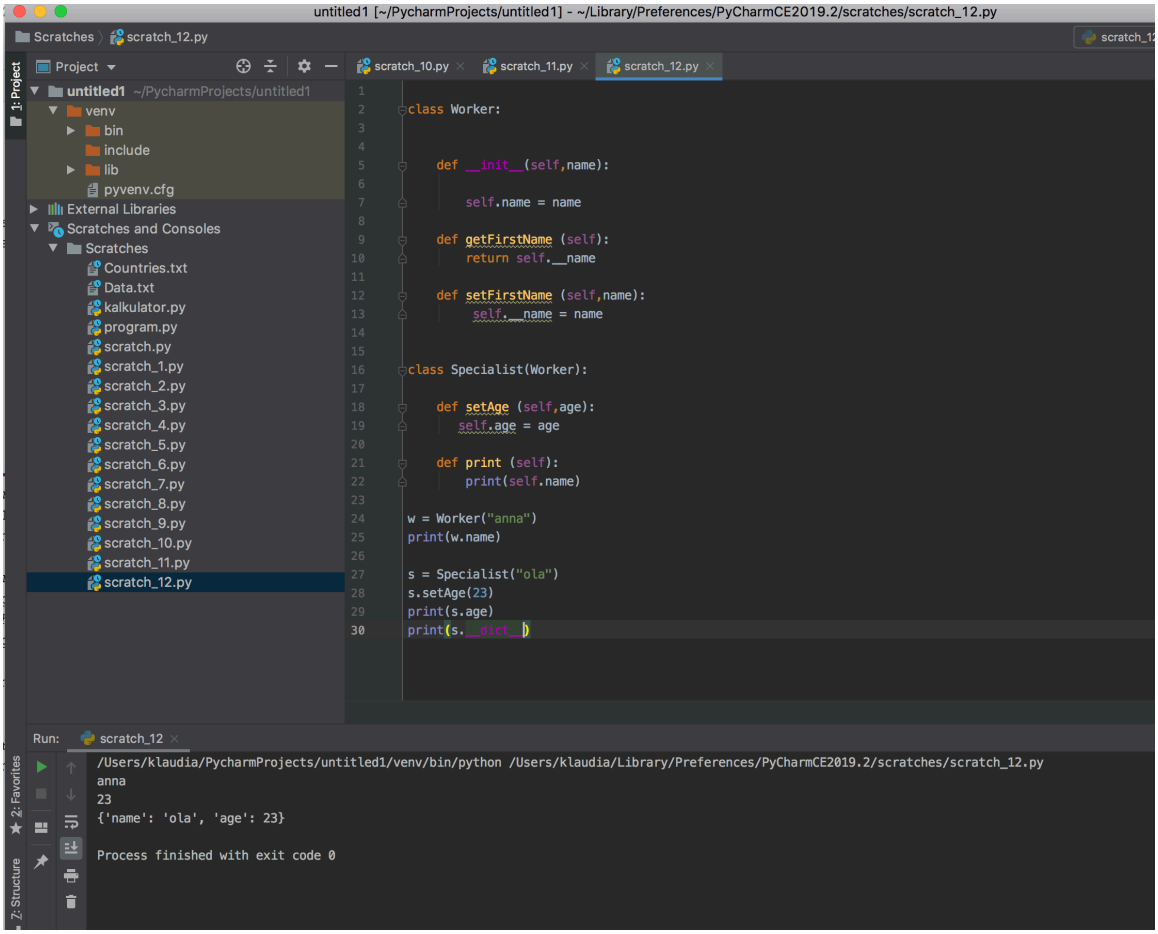
- **Customize Sub Classes:** Add attributes and/or methods to sub classes to extend their behavior beyond that of the super class. Using the added behavior on objects of the super class will raise an error. Attributes of the super class that will be used in the sub class need to be declared as *protected*.

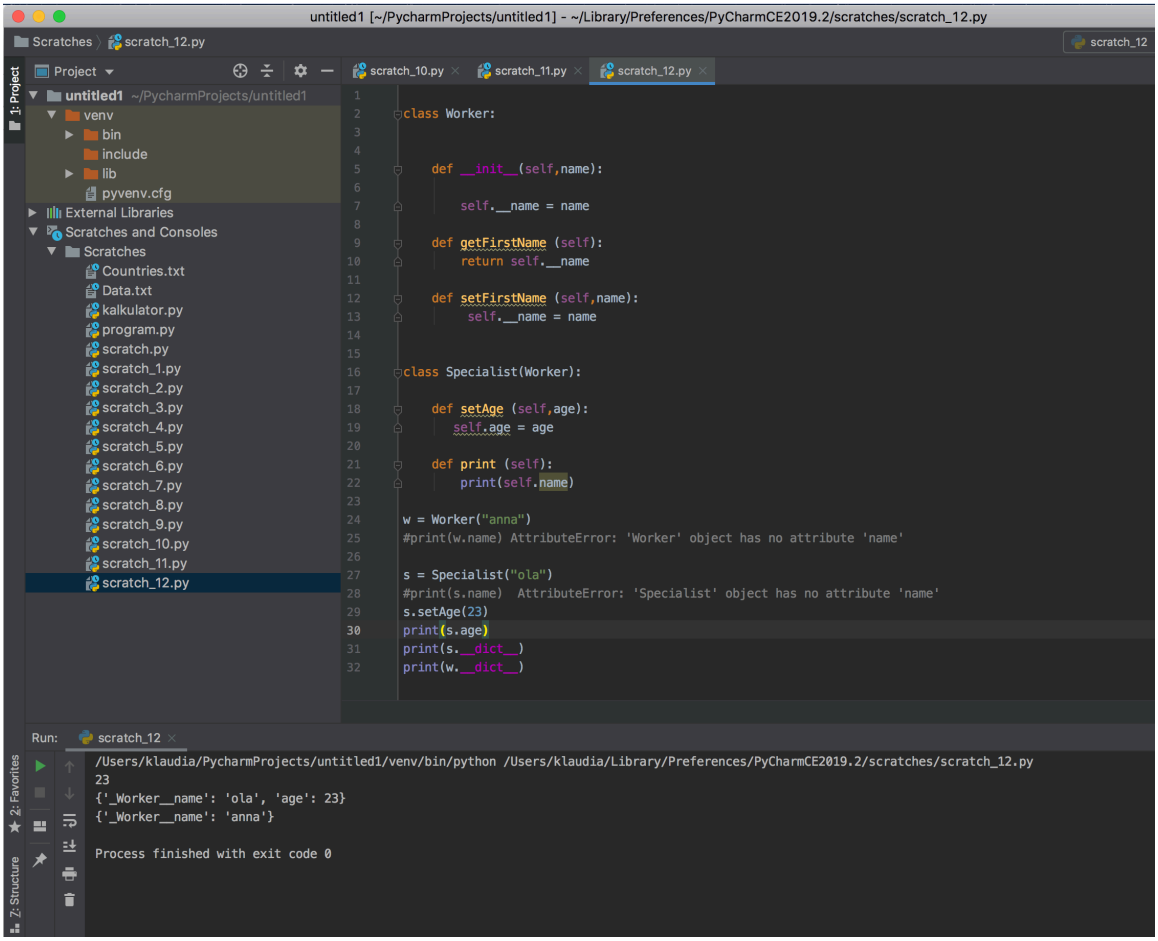
attribute or method in the super class. It is also worth noting that in order to be able to use super class attributes `salary` and `allowances`, they must be declared as *protected* instead of *private*. The following scripts demonstrate these concepts:

```
# Define class 'Employee'
class Employee():

      """ Define the constructor of the class with parameters.
      Define the attributes of the class """
      def __init__(self, first, last, salary, allowances):
            self.__first = first
            self.__last = last
protected   self._salary = salary
protected   self._allowances = allowances

      # Define a derived attribute
      def getTotalSalary(self):
            return self._salary + self._allowances

# Define the 'SalesEmployee' sub class
class salesEmployee(Employee):

      # Use the property decorator to define the getter method
      @property
      def commissionPercent(self):
            return self.__comm

      # Use the property decorator to define the setter method
      @commissionPercent.setter
      def commissionPercent(self, value):
            self.__comm = value

# Create and use object 'emp1' based on super class 'Employee'
emp1 = Employee("Maria", "Rena", 15000, 5000)
print(emp1.getTotalSalary())

# Create and use object 'semp1' based on sub class 'SalesEmployee'
semp1 = salesEmployee("Alex", "Flora", 16000, 6000)
# The attribute is set in the sub class
semp1.commissionPercent = 0.05

print(semp1.commissionPercent)

""" The next line generates an error since its
attribute only exists in the sub class """
print(emp1.commissionPercent)
```

Output 3.4.1.1:

```
20000
0.05

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-0e8e58d5eaf8> in <module>
     40 """ The next line generates an error since its
     41 attribute only exists in the sub class """
---> 42 print(empl.commissionPercent)
     43
     44 # Print the attributes of objects 'empl' and 'sempl'

AttributeError: 'Employee' object has no attribute 'commissionPercent'
```


untitled1 [~/PycharmProjects/untitled1] - ~/Library/Preferences/PyCharmCE2019.2/scratches/scratch_12.py
Scratches
scratch_12.py
Project
untitled1 ~/PycharmProjects/untitled1
venv
bin
include
lib
pyvenv.cfg
External Libraries
Scratches and Consoles
Scratches
Countries.txt
Data.txt
kalkulator.py
program.py
scratch.py
scratch_1.py
scratch_2.py
scratch_3.py
scratch_4.py
scratch_5.py
scratch_6.py
scratch_7.py
scratch_8.py
scratch_9.py
scratch_10.py
scratch_11.py
scratch_12.py
scratch_10.py
scratch_11.py
scratch_12.py
class Worker:
def __init__(self,name):
self.name = name
def getFirstName (self):
return self.__name
def setFirstName (self,name):
self.__name = name
class Specialist(Worker):
def setAge (self,age):
self.age = age
def print (self):
print(self.name)
w = Worker("anna")
print(w.name)
s = Specialist("ola")
s.setAge(23)
print(s.age)
print(s.__dict__)
Run:
scratch_12
/Users/klaudia/PycharmProjects/untitled1/venv/bin/python /Users/klaudia/Library/Preferences/PyCharmCE2019.2/scratches/scratch_12.py
anna
23
{'name': 'ola', 'age': 23}
Process finished with exit code 0
1: Project
2: Favorites
7: Structure


untitled1 [~/PycharmProjects/untitled1] - ~/Library/Preferences/PyCharmCE2019.2/scratches/scratch_12.py
Scratches
scratch_12.py
scratch_12
Project
untitled1 ~/PycharmProjects/untitled1
venv
bin
include
lib
pyvenv.cfg
External Libraries
Scratches and Consoles
Scratches
Countries.txt
Data.txt
kalkulator.py
program.py
scratch.py
scratch_1.py
scratch_2.py
scratch_3.py
scratch_4.py
scratch_5.py
scratch_6.py
scratch_7.py
scratch_8.py
scratch_9.py
scratch_10.py
scratch_11.py
scratch_12.py
scratch_10.py
scratch_11.py
scratch_12.py
class Worker:
def __init__(self,name):
self.__name = name
def getFirstName (self):
return self.__name
def setFirstName (self,name):
self.__name = name
class Specialist(Worker):
def setAge (self,age):
self.age = age
def print (self):
print(self.name)
w = Worker("anna")
#print(w.name) AttributeError: 'Worker' object has no attribute 'name'
s = Specialist("ola")
#print(s.name) AttributeError: 'Specialist' object has no attribute 'name'
s.setAge(23)
print(s.age)
print(s.__dict__)
print(w.__dict__)
Run:
scratch_12
/Users/klaudia/PycharmProjects/untitled1/venv/bin/python /Users/klaudia/Library/Preferences/PyCharmCE2019.2/scratches/scratch_12.py
23
{'_Worker__name': 'ola', 'age': 23}
{'_Worker__name': 'anna'}
Process finished with exit code 0
1: Project
2: Favorites
Z: Structure

Method Overriding

Method overriding is another important programming feature that is common in OOP languages. It allows a sub class to contain a method with a different implementation than the one inherited from the super class. In the context of the previous examples, the programmer may wish to compute the total salary of a sales employee by adding *commissions* to their salary and allowances. In this case, sub class method `getTotalSalary()` must be implemented differently to the original one inherited from `Employee`. As shown in the following example, super class method `getTotalSalary()` has to be called in the implementation of sub class method `getTotalSalary()`:

```
1   # Define class 'Employee'
2   class Employee():
3
4         # Define the constructor and the attributes of the super class
5         def __init__(self, first, last, salary, allowances):
6               self.__first = first
7               self.__last = last
8               self._salary = salary
9               self._allowances = allowances
10
11        # Define 'getTotalSalary'
12        def getTotalSalary(self):
13              return self._salary + self._allowances
14
15  # Define sub class 'salesEmployee'
16  class salesEmployee(Employee):
17
18        # Use the property decorator to define the getter method
19        @property
20        def commissionPercent(self):
21              return self.__comm
22
23        # Use the property decorator to define the setter method
24        @commissionPercent.setter
25        def commissionPercent(self, value):
26              self.__comm = value
27
```

```
        # Super class getter overrides the parent class method
        def getTotalSalary(self):
        return super().getTotalSalary() + (super().getTotalSalary()
*self.__comm)

# Create and use object 'emp1' based on super class 'Employee'
emp1 = Employee("Maria", "Rena", 15000, 5000)
print(emp1.getTotalSalary())

# Create and use object 'semp1' based on sub class 'salesEmployee'
semp1 = salesEmployee("Alex", "Flora", 16000, 6000)

# Set the attribute in the sub class
semp1.commissionPercent = 0.05

# Invoke the overridden getter method from the sub class
print(semp1.getTotalSalary())
```

```
20000
23100.0
```

3.4.2.1 Overriding the Constructor Method

The concept of method overriding is also used to create customized constructors in the sub class. In this case, the `super()` method is used to invoke the `__init__()` method of the super class, as shown in the following script:

Constructor Overriding: Call the `__init__()` method of the super class to access the constructor and add attributes to extend it.

```
# Define class 'Employee'
class Employee():

        # Define the constructor of the super class and its attributes
        def __init__(self, first, last, salary, allowances):
                self.__first = first
                self.__last = last
                self._salary = salary # Protected attribute
                self.__allowances = allowances

        # Define the getter of the class
        def getTotalSalary(self):
                return self._salary + self.__allowances

# Define sub class 'salesEmployee'
class salesEmployee(Employee):

        """ Define the constructor of the sub class adding the 'comm'
        attribute. Call the 'init' method of the super class """
        def __init__(self, first, last, salary, allowances, comm):
```

```
21              super().__init__(first, last, salary, allowances)
22              self.__comm = comm
23
24        # Access protected attribute '_salary' from the sub class
25        def getTotalSalary(self):
26              return super().getTotalSalary() + (self._salary *
27  self.__comm)
28
29  # Create and use object 'emp1' based on the super class
30  emp1 = Employee("Maria", "Rena", 15000, 5000)
31  print(emp1.getTotalSalary())
32
33  # Create and use object 'semp1' based on the sub class
34  semp1 = salesEmployee("Alex", "Flora", 16000, 6000, 0.05)
35  print(semp1.getTotalSalary()) # Method of the child class is invoked
```

```
20000
22800.0
```

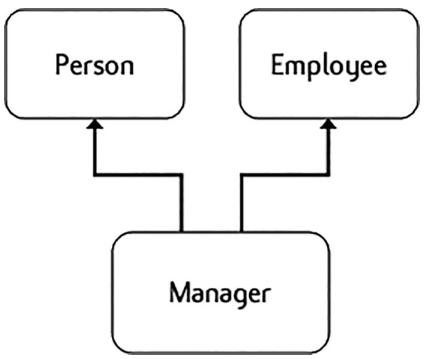
Multiple Inheritance

Sub classes can inherit attributes and methods from multiple super classes, a concept known as *multiple inheritance*. In Python, this can be implemented using the following syntax:

– **Multiple Inheritance:** The concept of having a sub class inheriting from more than one super classes.

```
class Parent1
      pass
class Parent2
      pass
class Child (Parent1, Parent2):
      pass
```

As an example of multiple inheritance, presents a structure consisting of two super classes (`Person` and `Employee`) and one sub class (`Manager`) that inherits from both super classes.



A representation of multiple inheritance between three classes.

The following Python scripts implement this structure. ' the constructor in the Manager class calls the respective constructors of both super classes during initialization. Methods getFullName and getContact are inherited from super class Person, while getAnnualSalary and getDepartment are inherited from Employee:

```
# Define the first super class ('Person')
class Person():

        # Define class constructor and attributes
        def __init__(self, firstName, lastName, contact):
                self.__firstName = firstName
                self.__lastName = lastName
                self.__contact = contact

        # Getter for the first & last name of the first super class
        def getFullName(self):
                return "Employee name is: " + self.__firstName +" " \
+ self.__lastName

        # Define the getter for the contact of the first parent
        def getContact(self):
                return "Contact number is: " + self.__contact
# Define the second Parent base class Employee
class Employee():
        # The constructor & the attributes of the second super class
        def __init__(self, salary, dept):
                self.__salary = salary
                self.__dept = dept

        # Define the getter for the salary of the second super class
        def getAnnualSalary(self):
                return "The annual salary is: " + str(self.__salary * 12)

        # The getter for the department of the 2nd super class
        def getDepartment(self):
                return "The employee belongs to the department: " +\
                        self.__dept
# Define subclass 'Manager' inheriting from both 'Person' and 'Employee'
class Manager(Person, Employee):
        def __init__(self, firstName, lastName, contact, salary, dept):
                Person.__init__(self, firstName, lastName, contact)
                Employee.__init__(self, salary, dept)
```

```
39
40  # Create and use a new instance of the 'Manager' class
41  mgr1 = Manager("Maria", "Rena", "0123456789", 14500, "Marketing")
42
43  # Call inherited behaviour from super class 'Person'
44  print(mgr1.getFullName())
45  print(mgr1.getContact())
46
47  # Call inherited behaviour from super class 'Employee'
48  print(mgr1.getAnnualSalary())
49  print(mgr1.getDepartment())
```

```
Employee name is: Maria Rena
Contact number is: 0123456789
The annual salary is: 174000
The employee belongs to the department: Marketing
```

POLYMORPHISM – METHOD OVERLOADING

Another powerful feature of OOP languages is the support of *method overloading*. This is a fundamental element of *polymorphism*, the option of defining and using two or more methods with the same name but different parameter lists or *signatures*. Overloading a method improves code readability and maintainability, as implementation is divided into multiple methods instead of being concentrated into a single, complex one.

– Polymorphism/ Method Overloading: The concept of using method overloading to implement two or more methods with the same name but different signatures.

While method overloading is a prominent feature in many OOP languages, such as Java and C++, it is not entirely supported in Python. Python is a *dynamically typed* language and *datatype binding* occurs at runtime. This is known as *late binding* and it differs from the *static binding* used in languages like Java and C++, in which overloaded methods are invoked at *compile time* based on the arguments they are supplied with. In Python, when multiple methods with the same name are defined, the last definition overrides all previous ones. As an example, consider method `calculateTotalSalary()` in the `Employee` class. The method computes the annual salary of the employee without the bonus. A second method that calculates the total salary plus the bonus can be implemented with the same name, thus, overloading `calculateTotalSalary()`. In this case, the first method will be ignored and any reference to it will raise an error, as shown in the following example:

```
1   # Define class 'Employee'
2   class Employee:
3
4           # Define method 'calculateTotalSalary'
5           def calculateTotalSalary(self):
6                   return(self.salary + self.allowances)
7           # Define a method overloading 'calculateTotalSalary'
8           def calculateTotalSalary (self, bonus):
9                   return(self.salary + self.allowances) + bonus
```

```
10
11  # Create and use the 'emp1' object
12  emp1 = Employee()
13  emp1.salary = 15000
14  emp1.allowances = 5000
15  print("Total salary is ", emp1.calculateTotalSalary(2000))
16
17  # Create and use the 'emp2' object
18  emp2 = Employee()
19  emp2.salary = 18000
20  emp2.allowances = 4000
21  # This method call will generate an error
22  print("Total salary is ", emp2.calculateTotalSalary())
```

```
Total salary is  22000
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-517b173547e9> in <module>
     22
     23 # This method call will generate an error
---> 24 print("Total salary is ", emp2.calculateTotalSalary())

TypeError: calculateTotalSalary() missing 1 required positional argument: 'bonus'
```

Method Overloading through Optional Parameters in Python

– **Method Overloading in Python:** In Python, use optional method parameters to emulate the method overloading feature available in other OOP languages.

Although Python does not directly support method overloading in the same form as other OOP languages, it offers an alternative approach to achieve the same functionality. Instead of resorting to the creation of multiple methods, it allows methods to take *optional parameters* with *default values*. When a method is invoked in the code, the programmer can choose whether to provide the parameter values or not. This, in turn, dictates which block of statements would be executed within the method. Commonly, the `None` value is used to assign a default null value to the attribute.

In the example below, constructor method `calculateTotalSalary()` is defined with optional parameter `bonus`. The implementation subsequently returns different values, depending on whether a new value has been assigned to the optional parameter. If this is not the case, the default `None` value is used.

```
1   class Employee:
2
3           def calculateTotalSalary(self, bonus = None):
4           # None statement supports both 'is' and '==' comparison operators
5                   if bonus is None:
6                           return(self.salary + self.allowances)
7                   else:
```

```
8                       return(self.salary + self.allowances) + bonus
9
10  emp1 = Employee()
11  emp1.salary = 15000
12  emp1.allowances = 5000
13  emp2 = Employee()
14  emp2.salary = 18000
15  emp2.allowances = 4000
16
17  print("Total salary is ", emp2.calculateTotalSalary(2000))
18  print("Total salary is ", emp1.calculateTotalSalary())
```

```
Total salary is 24000
Total salary is 20000
```

ABSTRACT CLASSES AND INTERFACES IN PYTHON

An *abstract* class is a class that cannot be instantiated. It serves as a blueprint or template for creating sub classes, but it cannot be used to create objects. An abstract class contains declarations of abstract methods. Declarations of this type include the names and parameter lists of the methods, but no implementation. The latter must be defined in the corresponding sub class.

Abstract Class: A class that cannot be instantiated, but serves as a template for sub classes. Abstract classes contain declarations of abstract methods (i.e., methods whose implementation must be defined in the sub classes or non-abstract methods).

In order to create abstract classes and methods, modules `ABC` and `abstractmethod` must be imported to the program. The syntax for doing so is the following:

```
from abc import ABC, abstractmethod
```

`ABC` stands for *Abstract Base Classes*. Newly created abstract classes inherit from `ABC` and must include at least one abstract method using the `@abstractmethod` built-in decorator,

with no implementation. The following script provides an example of an abstract class (i.e., `Employee`) with one abstract method (i.e., `getTotalSalary()`). Running this script raises an error, since abstract classes cannot instantiate objects:

```
# Import ABC
from abc import ABC, abstractmethod

# Define abstract class 'Employee'
class Employee(ABC):

        # Define abstract method 'getTotalSalary', which must be empty
        @abstractmethod
        def getTotalSalary(self):
              Pass

# Abstract classes cannot instantiate objects
emp1 = Employee()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-47be1b52dd97> in <module>
     11
     12 # Abstract classes cannot instantiate objects
---> 13 emp1 = Employee()

TypeError: can't instantiate abstract class Employee with abstract methods getTotalSalary
```

Once the abstract class is implemented, it can be used as a super class for deriving sub classes. Sub classes of this type must implement the abstract method of the abstract class as a minimum requirement. In this context, as shown in the first of the following scripts, sub class `FullTimeEmployee` will raise an error, since it does not implement the abstract method (i.e., `getTotalSalary()`) of its super abstract class (i.e., `Employee`). On the contrary, the second script presents the implementation of abstract method `getTotalSalary()` that resolves this issue:

```
# Import ABC
from abc import ABC, abstractmethod

# Define abstract class 'Employee'
class Employee(ABC):

        # Define abstract method 'getTotalSalary'
        @abstractmethod
        def getTotalSalary(self):
              Pass

# Define class 'fullTimeEmployee' based on the abstract class
class fullTimeEmployee(Employee):

        # Define the constructor of the sub class and its attributes
        def __init__(self, first, last, salary, allowances):
```

```
17                self.__first = first
18                self._last = last
19                self.__salary = salary
20                self.__allowances = allowances
21
22  # Error will be raised as the sub class does not implement
23  # the abstract method
24  ftl = fullTimeEmployee("Maria", "Rena", 15000, 6000)
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-7e5c51df1210> in <module>
     21
     22 # Error will be  raised as the sub class does not implement the abstract method
---> 23 ftl = fullTimeEmployee("Maria", "Rena", 15000, 6000)

TypeError: Can't instantiate abstract class fullTimeEmployee with abstract methods getTotalSalary
```

```
1   # Import ABC
2   from abc import ABC, abstractmethod
3
4   # Define abstract class 'Employee'
5   class Employee(ABC):
6
7         # Define abstract method 'getTotalSalary'
8         @abstractmethod
9         def getTotalSalary(self):
10                Pass
11
12  # Define class 'fullTimeEmployee' based on the abstract class
13  class fullTimeEmployee(Employee):
14
15        # Define the constructor of the sub class and its attributes
16        def __init__(self, first, last, salary, allowances):
17              self.__first = first
18              self._last = last
19              self.__salary = salary
20              self.__allowances = allowances
21
22        # Implement the abstract method of the abstract class
23        def getTotalSalary(self):
24              return self.__salary + self.__allowances
25
26  # Create and use a new 'fullTimeEmployee' object
27  ftl = fullTimeEmployee("Maria", "Rena", 15000, 6000)
28  print(ftl.getTotalSalary())
```

```
21000
```

Abstract classes may include both abstract and non-abstract methods with implementations. Sub classes that inherit from the abstract class also inherit the implemented methods. If required, the latter can be overridden, but in all cases, implementations must include the abstract method.

INTERFACES

In OOP, an *interface* refers to a class that serves as a template for the creation of other classes. Its main purpose is to improve the organization and efficiency of the code by providing blueprints for prospective classes. As such, interfaces describe the behavior of inherited classes, similarly to abstract classes. However, contrary to the latter, they cannot contain non-abstract methods. Python does not support the explicit creation of interfaces.

– **Interface:** A class that cannot be instantiated but serves as a template for sub classes. Unlike abstract classes, interfaces cannot have non-abstract methods.

MODULES AND PACKAGES IN PYTHON

Modules and *packages* refer to structures used for organizing code in Python. Modules are files containing Python code structures (e.g., classes, methods, attributes, or simple variables) signified by the *.py* file extension. Instead of rewriting particular blocks of code, modules can be imported into other Python files or applications, thus allowing for a *modular* programming approach based on *reusable* code.

Module: A module provides a way of organizing code in Python. Modules can host classes, methods, attributes, or even simple variables that can be imported and reused in other classes. Modules are commonly used with abstract classes.

Abstract classes and interfaces are two of the programming structures commonly stored in modules, from where they can be imported on demand. In the example provided in the following script, the entire definition of class `Employee` is stored in a module named *employee.py*:

```
1   # 'Employee' module saved in 'employee.py' file
2   class Employee:
3
4         # Define the constructor and private attributes of the class
5         def __init__(self, first, last, salary):
6               self.__firstName = first
7               self.__lastName = last
8               self.__salary = salary
9
10        # Define the getter for annual salary
11        def getAnnualSalary(self):
12              return self.__salary * 12
13
14        # Define the getter for fullName
15        def getFullName(self):
16              return self.__firstName + " " + self.__lastName
```

The import Statement

Python module files are imported using the `import` statement. The statement may include one or more modules. The syntax is the following:

```
import module1, [module2, module3…]
```

Once a module is imported, its classes and methods can be referenced using its name as a prefix (i.e., `module.classname`). The following example imports the `Employee` class from the associated *employee.py* module, and accesses its attributes and methods from the main body of the program:

The `import` Statement: Used to import either specific methods and attributes or entire classes stored in modules.

```
1   # Import the 'employee.py' file as a module
2   import employee
3
4   # Use the module to create and use a new object
5   emp1 = employee.Employee("Maria", "Rena", 15000)
6   print(emp1.getFullName())
7   print()
```

```
Maria Rena
```

The from…import Statement

A Python module may contain several classes, methods, attributes, or variables. The `from…import` statement allows the programmer to selectively import specific components from a module. The syntax is the following:

```
from module import name1, [name2, name3…]
```

Note that the names used in this example (e.g., name1, name2, name3) represent names of classes, methods, or attributes.

To import all objects from a module the following syntax can be used:

```
from module import *
```

The reader should note that if a specific class is imported *explicitly*, it can be referenced without a prefix, like in the next example:

```
1   # Import class 'Employee' from 'employee' module in 'employee.py'
2   from employee import Employee
3
4   # Use the imported class to create and use a new object
5   emp1 = Employee("Alex", "Flora", 18000)
6   print(emp1.getFullName())
7   print(emp1.getAnnualSalary())
```

```
Alex Flora
216000
```

PACKAGES

A *package* is a collection of modules grouped together in a common folder. The package folder is identified in a file with the designated name *__init__.py*, which indicates that the folder is a package. The *__init__.py* file can be empty, but it must be always present in the package folder. Once the package structure is created, Python modules can be added as required. The example illustrates the structure of a package named *hr*, containing the mandatory *__init__.py* file, and a module named *employee.py*.

Package: A mechanism used to store a number of different modules in the same folder for better code organization.

Modules contained in packages can be imported to an application using the package name as a prefix in the import statement, as shown in the following scripts:

```
1   # Import the employee module from the 'hr' package
2   import hr.employee
3
4   # Use 'Employee' class stored in the module to create & use an object
5   emp1 = hr.employee.Employee("Alex", "Flora", 16000)
6   print(emp1.getFullName())
7   print(emp1.getAnnualSalary())
```

```
Alex Flora
216000
```

```
1   # Import 'Employee' class in the employee module from 'hr' package
2   from hr.employee import Employee
3
4   # Use the 'Employee' class of the module to create and use an object
5   emp2 = Employee ("Alex", "Flora", 15000)
6   print(emp2.getFullName())
7   print(emp2.getAnnualSalary())
```

Package hr contains the __init__.py file and the employee.py module.

```
Alex Flora
180000
```

Using Modules to Store Abstract Classes

Modules may be also used to store abstract classes or interfaces. In the following example, abstract class `IEmployee` is stored in module *employee.py*, which is contained in the *hr* package named:

```
1   # Use 'abc' module to create an abstract class: store it as a module
2   # ('employee.py') in the hr package
3   from abc import ABC, abstractmethod
4
5   # Define abstract class 'IEmployee' and its behavior
6   class IEmployee(ABC):
7           @abstractmethod
8           def getTotalSalary(self):
9                   Pass
10          @abstractmethod
11          def getFullName(self):
12                  Pass
```

The following script demonstrates how the programmer can import the `IEmployee` class to the application, and use it to create a sub class (`FullTimeEmployee`):

```
1   # Import the 'IEmployee' class from the employee module ('hr' package)
2   from hr.employee import IEmployee
3
4   # Define a new sub class inheriting from the 'IEmployee' super class
5   class fullTimeEmployee(IEmployee):
6
7           # The constructor, attributes & behavior of the sub class
8           def __init__(self, first, last, salary, allowances):
9                   self.__first = first
10                  self.__last = last
11                  self.__salary = salary
12                  self.__allowances = allowances
13
14          def getTotalSalary(self):
15                  return self.__salary + self.__allowances
16
17          def getFullName(self):
18                  return self.__first + " " + self.__last
19
20  # Create and use a new object
21  ftl = fullTimeEmployee("Maria", "Rena", 15000, 6000)
22  print(ftl.getFullName())
23  print(ftl.getTotalSalary())
```

```
Maria Rena
21000
```

Relations Between Objects

In addition to *inheritance* and *implementation* that we've already seen, there are other types of relations between objects that we haven't talked about yet.

UML Association. Professor communicates with students.

Association is a type of relationship in which one object uses or interacts with another. In UML diagrams the association relationship is shown by a simple arrow drawn from an object and pointing to the object it uses. By the way, having a bi-directional association is a completely normal thing. In this case, the arrow has a point at each end.

In general, you use an association to represent something like a field in a class. The link is always there, in that you can always ask an order for its customer. It need not actually be a field, if you are modeling from a more interface perspective, it can just indicate the presence of a method that will return the order's customer.

UML Dependency. Professor depends on salary.

Dependency is a weaker variant of association that usually implies that there's no permanent link between objects. Dependency typically (but not always) implies that an object accepts another object as a method parameter, instantiates, or uses another object. Here's how you can spot a dependency between classes: a dependency exists between two classes if changes to the definition of one class result in modifications in another class.

UML Composition. University consists of departments.

Composition is a "whole-part" relationship between two objects, one of which is composed of one or more instances of the other. The distinction between this relation and others is that the component can only exist as a part of the container. In UML the composition relationship is shown by a line with a filled diamond at the container end and an arrow at the end pointing toward the component.

While we talk about relations between objects, keep in mind that UML represents relations between *classes*. It means that a university object might consist of multiple departments even though you see just one “block” for each entity in the diagram. UML notation can represent quantities on both sides of relationships, but it’s okay to omit them if the quantities are clear from the context.

UML Aggregation. Department contains professors.

Aggregation is a less strict variant of composition, where one object merely contains a reference to another. The container doesn’t control the life cycle of the component. The component can exist without the container and can be linked to several containers at the same time. In UML the aggregation relationship is drawn the same as for composition, but with an empty diamond at the arrow’s base.

02

Sherwood real estate requires an application to manage properties. There are two types of properties: apartments and houses. Each property may be available for rent or sale.

Both types of properties are described using a reference number, address, built-up area, number of bedrooms, number of bathrooms, number of parking slots, pool availability, and gym availability. A house requires extra attributes such as the number of floors, plot size and house type (villa or townhouse). An apartment requires additional attributes such as floor and number of balconies.

Each type of property (house or apartment) may be available for rent or sale.

A rental property should include attributes such as deposit amount, yearly rent, furnished (yes or no), and maids' room (yes or no). A property available for sale has attributes such as sale price and estimated annual service charge.

All properties include a fixed agent commission of 2%. Both types of sale properties have a fixed tax of 4%.

All properties require a method to display the details of the property.

All properties should include a method to compute the agent commission. For rental properties, agent commission is calculated by using the yearly rental amount, whereas for purchase properties it is calculated using the sale price.

Both types of purchase properties should include a method to compute the tax amount. Tax amount is computed based on the sale price.

Design and implement a Python application that creates the four types of properties (e.g., `RentalApartment`, `RentalHouse`, `SaleApartment`, `SaleHouse`) by using multiple inheritance and abstract classes. Implement class attributes and instance attributes using encapsulation. All numeric attributes, such as price, should be validated for inputs with a suitable minimum and maximum price.

Define the methods in the abstract class and implement it in the respective classes. Override the `print` method to display each property details.

Test your application by creating new properties of each type and calling the respective methods.