Programming technologies

Classes and Objects

Learning Goals/Objectives

Be able to read, comprehend, trace, adapt and create Python code that:

- Define a custom class
- Create of class objects
- Encapsulation realization
- Inheritance realization
- Polymorphism realization

Define custom class

General information

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

Classes provide a means of bundling data (state) and functionality (behavior) of entity together

Creating a new class creates a new type of object, allowing new instances of that type to be made

Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state

Class definitions, like function definitions (def statements) must be executed before they have any effect

Class Definition Syntax

To create a class, use the keyword class:

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

Minimum Class Definition

```
class MyClass:
    pass
```

Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

`MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

Creates a new instance of the class and assigns this object to the local variable `x`.

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        """some description"""
        return 'hello world'

x = MyClass()
```

Built In Class Attributes

Attribute	Description
__dict__	Dictionary containing the class's namespace
__doc__	The class’s documentation string, or None if unavailable
__name__	The class name
__module__	The name of the module the class was defined in, or None if unavailable
__bases__	A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list

```
print(MyClass.__dict__)
print(MyClass.__doc__)
print(MyClass.__name__)
…
```

Instance Objects

An object that is created using a class is said to be an instance of that class

Class instantiation uses function notation:

```
# Creates a new instance of the class and assigns this object to the local variable x
x = MyClass()
```

All instance objects contain a unique identity and type

```
print(id(MyClass))
x, y = MyClass(), MyClass()
print(id(x))
print(id(y))
print(type(MyClass))
print(type(x))
```

Constructors in Python

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the `__init__()` method is called the constructor and is always called when an object is created.
Syntax of constructor declaration :

```
def __init__(self):
    # body of the constructor
```

Types of constructors :
default constructor: The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.
parameterized constructor: constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Default constructor

```
class Person:

    # default constructor
    def __init__(self):
        self.name = "John"
        self.surname = "Doe"

    # a method for string presentation of instance
    def __str__(self):
        return self.name + " " + self.surname

# creating object of the class
somebody = Person()

# calling the instance method using the object obj
print(somebody)
```

Parameterized constructor

```
class Person:

    # parameterized constructor
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    # a method for string presentation of instance
    def __str__(self):
        return self.name + " " + self.surname

# creating object of the class
mike = Person("Mike", "Peerson")

# calling the instance method using the object obj
print(mike)
```

Destructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.

The `__del__()` method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration :

```
def __del__(self):
    # body of destructor
```

Note : A reference to objects is also deleted when the object goes out of reference or when the program ends.

Destructors in Python

```
class Employee:

    # default constructor
    def __init__(self):
        print('Employee created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Python’s special methods

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	__add__(self, other)	Addition
*	__mul__(self, other)	Multiply
-	__sub__(self, other)	Subtraction
==	__eq__(self, other)	Equal
!=	__ne__(self, other)	Not equal
[index]	__getitem__(self, index)	Index operator
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

OOP principles

- encapsulation
- inheritance
- polymorphism

OOP - Encapsulation

Encapsulation - Encapsulation is a mechanism of wrapping the data (data attributes) and methods acting on the data (method attributes) together as a single unit (class)

Encapsulation can be used to hide the class attributes (internal representation of an object is generally hidden from view outside of the object's definition)

By default, attributes in classes are public, which means that from anywhere in the program we can get an attribute of an object and change it. Concept of keeping implementation details hidden from the rest of the system is key to object oriented design

Public, protected, private interfaces

In Python there are no attribute access modifiers such as public, protected or private all attributes are public. But there is a convention to define protected and private interfaces

Name	Notation	Behavior
name	public	Accessible from anywhere. Python attributes and methods are public by default
_name	protected	Like a public member, but it shouldn't be directly access from outside
__name	private	*Accessible only in their own class

*Python allows to access or modify a variable/method that is considered private. To get access to private variable/method used: <instance>. _< className>__<name>

OOP - Encapsulation

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

    def display_info(self):
        print("Name:", self.name, "\tAge:", self.age)

tom = Person("Tom", 23)
tom.name = "Spider-Man"          # changes of field name
tom.age = -129                   # changes of field age
tom.display_info()               # Name: Spider-Man    Age: -129
```

OOP - Encapsulation

```
class Person:
    def __init__(self, name, age):
        self.__name = name    # set name
        self.__age = age      # set age
    def set_age(self, age):
        if age in range(1, 150):
            self.__age = age
        else:
            print("Invalid age")
    def get_age(self):
        return self.__age
    def get_name(self):
        return self.__name
    def display_info(self):
        print("Name:", self.__name, "\tAge:", self.__age)

tom = Person("Tom", 23)
tom.__age = 43              # Field age does not change
tom.display_info()          # Name: Tom  Age: 23
tom.set_age(-3486)          # INvalid age
tom.set_age(25)
tom.display_info()          # Name: Tom  Age: 25
```

Properties

A property is a special sort of class member, intermediate in functionality between a field (or data member) and a method.

Access to protected/private attributes is achieved through property attributes.

This method involves the use of annotations that are preceded by the **@** symbol.

To create a getter property, the **@property** annotation is placed over the property.

To create a setter property, the annotation **@getter_property_name.setter** is set above the property.

Example:

@property

@<name>.setter

@<name>.deleter

Properties

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

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 150):
            self.__age = age
        else:
            print("Invalid age")

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print("Name:", self.__name, "\tage:", self.__age)

tom = Person("Tom", 23)

tom.display_info()      # Name: Tom  age: 23
tom.age = -3486         # Invalid age
print(tom.age)          # 23
tom.age = 36
tom.display_info()      # Name: Tom  Age: 36
```

Inheritance

Inheritance is a form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities. The existing class is called the base class (superclass), and the new class is referred to as the derived class (subclass)

- "Inheritance is the ability to define a new class that is a modified version of an existing class." – Allen Downey, Think Python
- "A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a "kind of" hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of superclasses." Grady Booch, Object Oriented Design

```
class Subclass(Superclass):
    #definition of subclass
```

Types of Inheritance

Single inheritance

subclasses inherit the features of one superclass

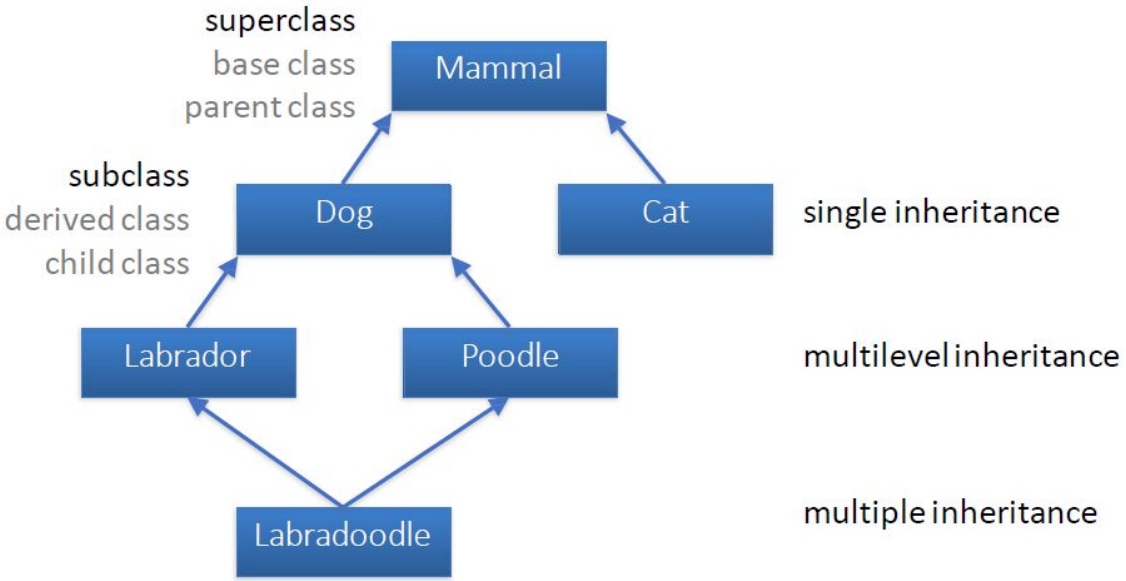
Multiple inheritance

one class can have more than one superclass and inherit features from all parent classes

Multilevel inheritance

a subclass is inherited from another subclass

Types of Inheritance



Single Inheritance

The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- The name BaseClassName must be defined in a scope containing the derived class definition
- Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class
- DerivedClassName() creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object

Inheritance

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

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Invalid age")

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print("Name:", self.__name, "\tAge:", self.__age)
```

Inheritance

```
class Employee(Person):

    def __init__(self, name, age, company):
        Person.__init__(self, name, age)
        self.__company = company

    @property
    def company(self):
        return self.__company

    @company.setter
    def company(self, company):
        self.__company = company

    def display_info(self):
        # print("Name:", self.__name, "\tAge:", self.__age, "\tWorks in ", company)
        # Not correct, self.__name, self.__age - private fields
        print("Name:", self.name, "\tAge:", self.age, "\tWorks in ", self.__company)


tom = Employee("Tom", 23, "Google")
tom.age = 33
tom.display_info()
```

Methods overriding

- Derived classes may override methods of their base classes
- Method overriding, in object oriented programming, is a language feature that allows a derived class to provide a specific implementation of a method that is already provided by one of its base classes or parent classes
- To override a method in the base class, derived class needs to define a method of same signature
- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`

Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3, ...):
    <statement-1>
    .
    .
    .
    <statement-N>
```

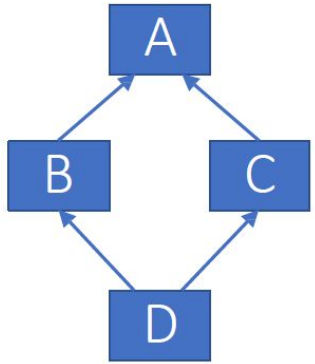
Method Resolution Order: depth first, left to right.

> Thus, if an attribute is not found in Subclass, it is searched in Superclass1, then recursively in the classes of Superclass1, and only if it is not found there, it is searched in Superclass2, and so on

Most of the time, single inheritance is good enough

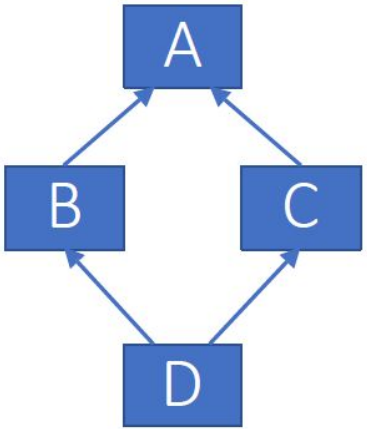
The diamond problem in multiple inheritance

- The "diamond problem" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?
- To solve the problem Python uses the list of classes to inherit from as an ordered list (list of classes is created using the C3 linearization (or Method Resolution Order (MRO)) algorithm).
- The name C3 refers to the three important properties of the resulting linearization:
  - a consistent extended precedence graph,
  - preservation of local precedence order,
  - fitting the monotonicity criterion.



More about MRO: https://www.geeksforgeeks.org/method-resolution-order-in-python-inheritance/

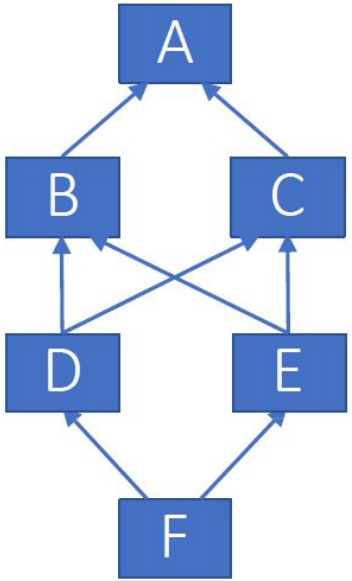
The diamond problem in multiple inheritance




```
class A:
    def who_am_i(self):
        print ('I am a A')

class B(A):
    def who_am_i(self):
        print ('I am a B')

class C(A):
    def who_am_i(self):
        print ('I am a C')

class D(B, C):
    pass
```

```
>> obj D()

>> print(D.__mro__)
<class '__main__.D'>,
<class '__main__.B'>,
<class '__main__.C'>,
<class '__main__.A'>,
<class 'object'>

>> obj.who_am_i()
I am a B
```

The diamond problem in multiple inheritance




```
class A:
    def who_am_i(self):
        print('I am a A')
class B(A):
    def who_am_i(self):
        print('I am a B')
class C(A):
    def who_am_i(self):
        print('I am a C')
class D(B, C):
    def who_am_i(self):
        print('I am a D')
class E(C, B):
    def who_am_i(self):
        print('I am a E')
class F(D, E):
    pass
```

```
Traceback (most recent call last):
  File "main.py", line 16, in <module>
    class F(D, E):
TypeError: Cannot create a consistent method
resolution
order (MRO) for bases B, C
```

super()

```
super([type[, object or type]])
```

return a proxy object that delegates method calls to a parent or sibling class of type (is useful for accessing inherited methods that have been overridden in a class)

A typical superclass call looks like this:

```
class B(A):
    def method(self, arg):
        super().method(arg)
```

super() is implemented as part of the binding process for explicit dotted attribute lookups such as super().getitem__(name)

In a class hierarchy with single inheritance, super can be used to refer to parent classes without

naming them explicitly, thus making the code more maintainable

In a class hierarchy with multiple inheritance, super makes it possible to implement

“diamond diagrams” where multiple base classes implement the same method

super()

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]
    def input_sides(self):
        self.sides = [float(input('Enter side ' + str(i + 1 ) + '
: ')) for i in range(self.n)]

class Triangle(Polygon):
    def __init__(self):
        super().__init__(3)
        # super(Triangle, self).__init__(3) is possible too
    def find_area(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s * (s - a) * (s - b) * (s - c)) ** 0.5
        return area

t = Triangle()
t.input_sides()
print('Area of triangle is %0.2f' % t.find_area())
```

super()

```
class A:
    def __init__(self):
        print('start A')
        self.a = 1
        print('end A')
class B(A):
    def __init__(self):
        print('start B')
        super().__init__()
        self.b = self.a + 1
        print('end B')
class C(A):
    def __init__(self):
        print('start C')
        super().__init__()
        self.c = self.a + 1
        print('end C')
class D(B, C):
    def __init__(self, value):
        print('start D')
        super().__init__()
        self.d = self.a + self.b + self.c
        print('end D')

obj=D(5)
print(obj.a, obj.b, obj.c, obj.d, sep=';')
```

Classmethod and Staticmethod

Static methods

a static method knows nothing about the class or instance

```
@staticmethod
def f(arg1, arg2, ...):
    suite
```

a static method does not receive the instance or class as implicit first argument

Class methods

a class method knows its class

```
@classmethod
def f(cls, arg1, arg2, ...):
    suite
```

a class method receives the class as implicit first argument (cls)

Classmethod and Staticmethod

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __str__(self):
        return 'Pizza(%s)' % str(self.ingredients)

    @classmethod
    def margherita(cls):
        return cls(['cheese', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['cheese', 'tomatoes', 'ham', 'mushrooms'])

    @staticmethod
    def circle_area(R):
        from math import pi
        return R ** 2 * pi

print(Pizza.margherita())
print(Pizza.prosciutto())
print('Pizza area is %.2f' % Pizza.circle_area(0.15))
```

Classmethod and Staticmethod

```
class Foo:
    message = "I'm Foo class"

    @classmethod
    def class_method(cls):
        print(cls.message)

    @staticmethod
    def static_method():
        print(Foo.message)

class Bar(Foo):
    message = "I'm Bar class"


Foo.class_method()
Foo.static_method()
Bar.class_method()
Bar.static_method()
```

Polymorphism

The word polymorphism means having many forms.

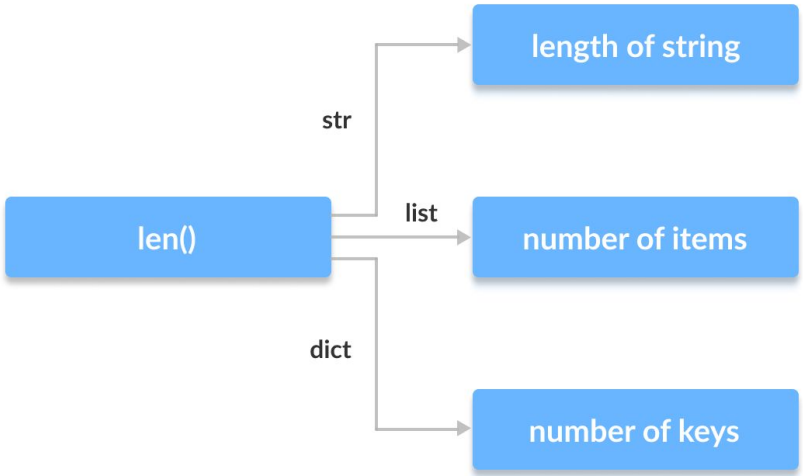
In programming, ***polymorphism*** means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Example

```
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
    return x + y + z

# Driver code
print(add(2, 3)) // 5
print(add(2, 3, 4)) // 9
print(add("str1", " ", "str2")) // "str1 str2"
```

Example

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```



Polymorphism in OOP

Polymorphism is a very important idea in object-oriented programming.

We can use the idea of polymorphism for class methods, since different classes in Python can have methods with the same name.

Later we can generalize calling these methods by ignoring the object we are working with.

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

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")


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

    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Bark")
```

```
cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

OUTPUT:

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Polymorphism in OOP

As in other programming languages, in Python, child classes can inherit methods and attributes from their parent class. We can override some methods and attributes specifically to match the child class, and this behavior is known as method overriding.

Polymorphism allows us to have access to these overridden methods and attributes that have the same name as in the parent class.

```
from math import pi


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

    def area(self):
        pass

    def fact(self):
        return "I am a two-dimensional shape."

    def __str__(self):
        return self.name


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

    def area(self):
        return self.length**2

    def fact(self):
        return "Squares have each angle equal to 90 degrees."
```

```
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return pi*self.radius**2


a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())
```

OUTPUT:

```
Circle
I am a two-dimensional shape.
Squares have each angle equal to 90 degrees.
153.93804002589985
```

Abstract Base Classes

- Abstract base class is a class that has no instances. An abstract class is written with the expectation that its concrete subclasses will add to its structure and behavior, typically by implementing its abstract operations Grady Booch, Object oriented analysis and design with applications
- Before a class derived from an abstract base class can be instantiated, all abstract methods of its parent classes must be implemented by some class in the derivation chain
- Abstract base classes complement duck typing by providing a way to define interfaces when other techniques like hasattr() would be clumsy or subtly wrong (for example with magic methods)

Abstract Base Classes

Abstract classes can be created in several ways:

- By use of the module abc which provides the infrastructure for defining abstract base classes in Python
- In many dynamically typed languages, any class that has a method but doesn't implement it (raise a not implemented error), can be considered as abstract
- By inheriting from an abstract base class and not overriding all missing features necessary to complete the class definition

Example
Employee, SalesPerson, Manager

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def setBonus(self, bonus):
        self.bonus = bonus

    def toPay(self):
        return self.salary + self.bonus


class SalesPerson(Employee):
    def __init__(self, name, salary, percent):
        super().__init__(name, salary)
        self.percent = percent

    def setBonus(self, bonus):
        if (self.percent > 200):
            self.bonus = bonus * 3
        elif(self.percent > 100):
            self.bonus = bonus * 2
        else:
            self.bonus = bonus
```

```
class Manager(Employee):
    def __init__(self, name, salary, clientAmount):
        super().__init__(name, salary)
        self.quantity = clientAmount

    def setBonus(self, bonus):
        if (self.quantity > 150):
            self.bonus = bonus + 1000
        elif (self.quantity > 100):
            self.bonus = bonus + 500
        else:
            self.bonus = bonus

person1 = Employee("John", 1000)
sales1 = SalesPerson("James", 800, 210)
manager1 = Manager("Paul", 1200, 170)

stuff = [person1, sales1, manager1]

for empl in stuff:
    empl.setBonus(200)

for empl in stuff:
    print(f"Name: {empl.name} Salary: {empl.salary} Bonus: {empl.bonus} Total: {empl.toPay()}")
```