Lecture 7 Object Oriented Programming Features (2021.10.21 & 11.2 & 11.4)

§1 Call-by-Reference / Call-by-Value

THE PROBLEM OF CALL-BY-REFERENCE / CALL-BY-VALUE

Circle.py

```
import math

class Circle:
    # Construct a circle object
    def __init__(self, radius = 1):
        self.radius = radius

    def getPerimeter(self):
        return 2 * self.radius * math.pi

    def getArea(self):
        return self.radius * self.radius * math.pi

    def setRadius(self, radius):
        self.radius = radius
```

Test.py

```
from Circle import Circle

def main():
    # Create a Circle object with radius 1
    myCircle = Circle()

    # Print areas for radius 1, 2, 3, 4, and 5
    n = 5
    printAreas(myCircle, n)

    # Display myCircle.radius and times
    print("\nRadius is", myCircle.radius)
    print("n is", n)

# Print a table of areas for radius
def printAreas(c, times):
    print("Radius \t\tArea")
    while times >= 1:
        print(c.radius, "\t\t", c.getArea())
        c.radius = c.radius + 1
        times = times - 1

main() # Call the main function
```

```
Radius                  Area
1                       3.141592653589793
2                       12.566370614359172
3                       29.274333882308138
4                       50.26548245743669
5                       79.53981633974483

Radius is 6
n is 5
```

香港中文大學(深圳)

```
def main():
    # Create a Circle object with radius 1
    myCircle = Circle()

    # Print areas for radius 1, 2, 3, 4, and 5
    n = 5
    printAreas(myCircle, n)

    # Display myCircle.radius and times
    print("\nRadius is", myCircle.radius)
    print("n is", n)
```

```
# Print a table of areas for radius
def printAreas(c, times):
    print("Radius \t\tArea")
    while times >= 1:
        print(c.radius, "\t\t", c.getArea())
        c.radius = c.radius + 1
        times = times - 1
```

1. Call-by-Value

1° Copies the value of an argument into the formal parameter of that function

2° Change made to the parameter of the main function do not affect the argument

2. Call-by-Reference

1° Copies the reference (address) of an argument into the formal parameter.

2° The reference is used to access the actual argument used in the function call

3° Changes made in the parameter alter the passing argument.

```
class B:
    def __init__(self):
        self.data = 0

def f(a, b, c):
    a = 1
    b.data = 1
    c = B()
    c.data = 1

a = 0
b = B()
c = B()
f(a,b,c)
print(a, b.data, c.data)
```

3. Discussion

1° Pass-by-value for immutable object

2° Pass-by-reference for mutable object

考虑步骤(个人理解):

① 将所有 def 部分中的变量名称换为 main 部分中未出现的名称

② 假设 main 部分中存在变量名 a，a 引用的 object 为 t (t可以是 string, int, list ……)，表示为 a → t

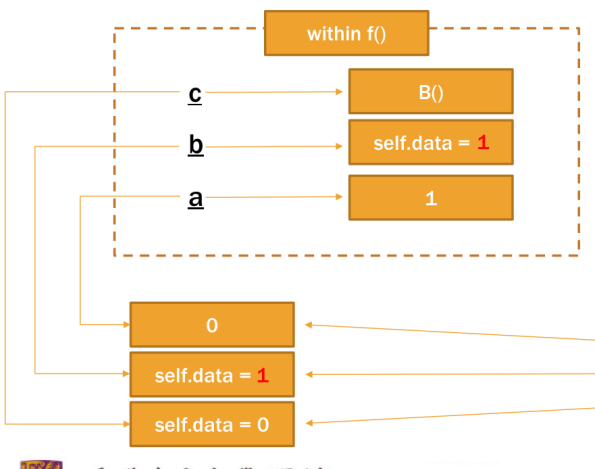
③ 假设存在 def(x) (x不与 main 部分中的任何变量名重合)，对局部变量 x 进行一系列操作

④ 当 main 部分出现 def(a) 时，python 创建一变量名 x，其引用的 object 与 a 相同，即 x → t，id(a) = id(x).

⑤ 接下来原本在 def 部分中对 x 进行的操作将对 t 进行

⑥ 一旦 x 引用的 object 更换，即 x = t' (t' 为另一个 object)，后续原本在 def 中对 x 的操作将对 t' 进行

⑦ 之后若 main 部分对 a 操作，则 a 引用的 object 为被操作过的 t (此时的 t 与先前的 t 为相同的 object，id 相同)




```
class B:
    def __init__(self):
        self.data = 0

def f(a, b, c):
    a = 1
    b.data = 1
    c = B()
    c.data = 1

a = 0
b = B()
c = B()
f(a,b,c)
print(a, b.data, c.data)
```

010

⑧ 对于 mutable object：内容相同，id可能不同

对于 immutable object：内容相同，id一定相同

- List, Dictionary, Tuple

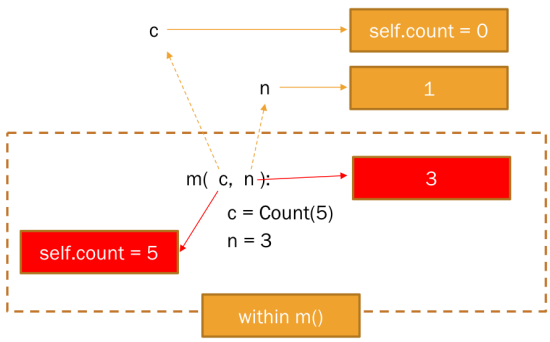
```
def f(myList, myDict,myTuple):
    myList[0]=0
    myList=[2,3,4]
    myDict["key1"] = "value2"
    myTuple = (2,3,4)

myList = [1,2,3]
myDict = {"key1":"value1"}
myTuple = (1,2,3)

f(myList, myDict,myTuple)

print(myList)
print(myDict)
print(myTuple)
```

```
[0, 2, 3]
{'key1': 'value2'}
(1, 2, 3)
```

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

EXAMPLE

```
class Count:
    def __init__(self, count = 0):
        self.count = count

def main():
    c = Count()
    n = 1
    m(c, n)

    print("count is", c.count)
    print("n is", n)

def m(c, n):
    c = Count(5)
    n = 3

main() # Call the main function
```



What would be the output of the above program?

```
count is 0
n is 1
```

§2 Private Data Fields

1. Issue of direct access to data fields

1° Data may be tampered with c.radius = -1 or tv1.channel = 125

2° Classes become difficult to maintain and vulnerable to bug.

- Suppose you want to modify the Circle class to ensure that the radius is non-negative after other programs have already used the class
- You have to change not only the Circle class but also the programs that use it, because the clients may have modified the radius directly

2. Private data field

1° Private data fields and methods can be accessed within a class, but their cannot be accessed outside the class.

2° In Python, name of private data field / method: two leading underscores.

3° If you want to call a private data, you can write another method in the class.

Test.py

```
import math

class Circle:
        def __init__(self, radius = 1):
                self.__radius=radius
        def getRadius(self):
                return self.__radius
        def getPerimeter(self):
                return 2*self.__radius*math.pi
        def getArea(self):
                return self.__radius**2*math.pi
        def setRadis(self, radius):
                self.__radius=radius

c = Circle(5)
print(c.getRadius(),"\t",c.getArea(),"\t",c.getPerimeter())
c.setRadis(10)
print(c.getRadius(),"\t",c.getArea(),"\t",c.getPerimeter())
```

PRACTICE

```
class A:
    def __init__(self, i):
        self.__i = i

def main():
    a = A(5)
    print(a.__i)

main() # Call the main function
```

- What is the problem with this program? a.__i is private

PRACTICE

```
def main():
    a = A()
    a.print()

class A:
    def __init__(self, newS = "Welcome"):
        self.__s = newS

    def print(self):
        print(self.__s)

main() # Call the main function
```

- Is the above code correct?
- If yes, what would be the output? Welcome

3. Naming style

1° Single pre underscore: _variable

① for internal use, but it doesn't stop you from accessing them.

② effects the names that are imported from the module (doesn't import the names which starts with a single pre underscore)

2° Single post underscore: variable_
① used to avoid conflicts with the Python Keywords.
3° Double pre underscore: __variable
① used for the name mangling
② private
③ to avoid the overriding of the variable in subclass
4° Double pre and post underscores: __variable__
① magic method
② it's better to stay away them.

§3 Abstraction

1. Abstraction

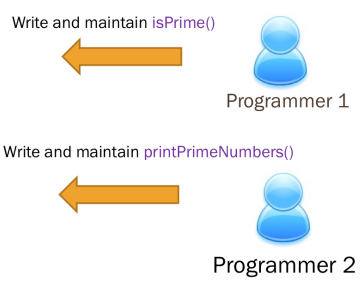
1° Separate the implementation of a part of code from the usage

2° Make your code easy to remain, debug and reuse

2. Function abstraction

Separating the implementation of a function from its usage.

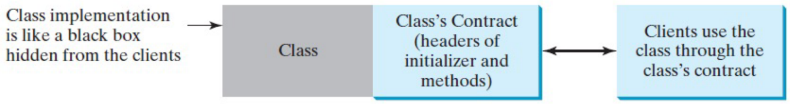
```
def isPrime(number):
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            return False
        divisor +=1
    return True

def printPrimeNumbers(numberOfPrimes=50):
    NUMBER_OF_PRIMES_PER_LINE = 10
    count = 0
    number = 2
    while count<numberOfPrimes:
        if isPrime(number):
            count+=1
            print(number, end=" ")
            if count % NUMBER_OF_PRIMES_PER_LINE == 0:
                print()
        number+=1

print("The first 50 prime numbers are:")
printPrimeNumbers();
```



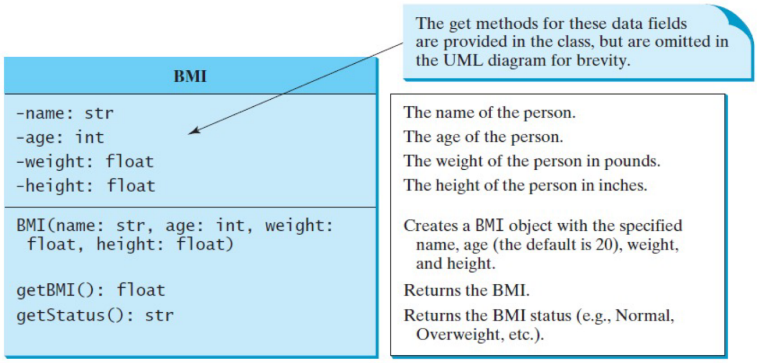
3. Class abstraction

1° Combines data and methods into a single object and hides the data fields and method implementation from the user.

- The user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user.



EXAMPLE – BMI CALCULATION CONTRACT



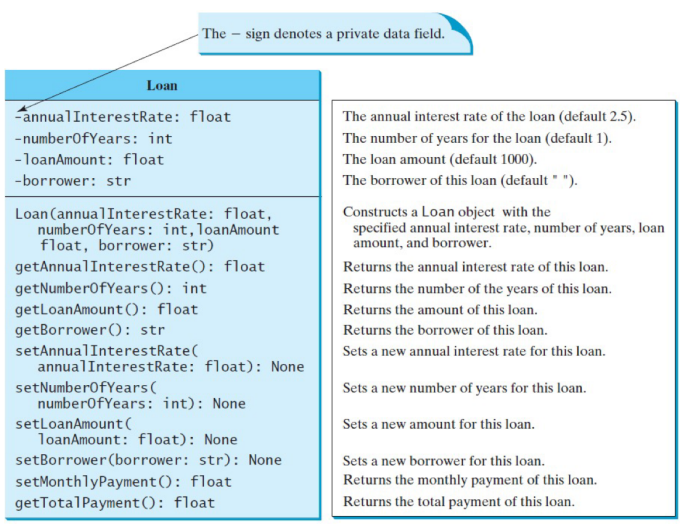
- We can use the BMI class if you have its contract
- You don't need to know the details about how it is implemented!!

```
from BMI import BMI

def main():
    bmi1 = BMI("John Doe", 18, 145, 70)
    print("The BMI for", bmi1.getName(), "is",
        bmi1.getBMI(), bmi1.getStatus())

    bmi2 = BMI("Peter King", 50, 215, 70)
    print("The BMI for", bmi2.getName(), "is",
        bmi2.getBMI(), bmi2.getStatus())

main() # Call the main function
```

THE BMI CLASS

```
class BMI:
    def __init__(self, name, age, weight, height):
        self.__name = name
        self.__age = age
        self.__weight = weight
        self.__height = height

    def getBMI(self):
        KILOGRAMS_PER_POUND = 0.45359237
        METERS_PER_INCH = 0.0254
        bmi = self.__weight * KILOGRAMS_PER_POUND /  \
            ((self.__height * METERS_PER_INCH) * \
            (self.__height * METERS_PER_INCH))
        return round(bmi * 100) / 100

    def getStatus(self):
        bmi = self.getBMI()
        if bmi < 18.5:
            return "Underweight"
        elif bmi < 25:
            return "Normal"
        elif bmi < 30:
            return "Overweight"
        else:
            return "Obese"

    def getName(self):
        return self.__name

    def getAge(self):
        return self.__age

    def getWeight(self):
        return self.__weight

    def getHeight(self):
        return self.__height
```


EXAMPLE - LOAN



```
from Loan import Loan

def main():
    # Enter yearly interest rate
    annualInterestRate = eval(input
        ("Enter yearly interest rate, for example, 7.25: "))

    # Enter number of years
    numberOfYears = eval(input(
        "Enter number of years as an integer: "))

    # Enter loan amount
    loanAmount = eval(input(
        "Enter loan amount, for example, 120000.95: "))

    # Enter a borrower
    borrower = input("Enter a borrower's name: ")

    # Create a Loan object
    loan = Loan(annualInterestRate, numberOfYears,
        loanAmount, borrower)

    # Display loan date, monthly payment, and total payment
    print("The loan is for", loan.getBorrower())
    print("The monthly payment is",
        format(loan.getMonthlyPayment(), ".2f"))
    print("The total payment is",
        format(loan.getTotalPayment(), ".2f"))

main() # Call the main function
```

```
Enter yearly interest rate, for example, 7.25: 2.5 Enter
Enter number of years as an integer: 5 Enter
Enter loan amount, for example, 120000.95: 1000 Enter
Enter a borrower's name: John Jones Enter
The loan is for John Jones
The monthly payment is 17.75
The total payment is 1064.84
```

EXAMPLE OF LOAN CLASS

```
class Loan :
    def __init__(self, annualInterestRate = 2.5,
        numberOfYears = 1, loanAmount = 1000, borrower = " "):
        self.__annualInterestRate = annualInterestRate
        self.__numberOfYears = numberOfYears
        self.__loanAmount = loanAmount
        self.__borrower = borrower

    def getAnnualInterestRate(self):
        return self.__annualInterestRate

    def getNumberOfYears(self):
        return self.__numberOfYears

    def getLoanAmount(self):
        return self.__loanAmount

    def getBorrower(self):
        return self.__borrower

    def setAnnualInterestRate(self, annualInterestRate):
        self.__annualInterestRate = annualInterestRate

    def setNumberOfYears(self, numberOfYears):
        self.__numberOfYears = numberOfYears

    def setLoanAmount(self, loanAmount):
        self.__loanAmount = loanAmount

    def setBorrower(self, borrower):
        self.__borrower = borrower

    def getMonthlyPayment(self):
        monthlyInterestRate = self.__annualInterestRate / 1200
        monthlyPayment = \
          self.__loanAmount * monthlyInterestRate / (1 - (1 /
          (1 + monthlyInterestRate) ** (self.__numberOfYears * 12)))
        return monthlyPayment

    def getTotalPayment(self):
        totalPayment = self.getMonthlyPayment() * \
            self.__numberOfYears * 12
        return totalPayment
```

PRACTICE

(*Algebra: quadratic equations*) Design a class named `QuadraticEquation` for a quadratic equation $ax^2 + bx + c = 0$. The class contains:

- The private data fields `a`, `b`, and `c` that represent three coefficients.
- A constructor for the arguments for `a`, `b`, and `c`.
- Three `get` methods for `a`, `b`, and `c`.
- A method named `getDiscriminant()` that returns the discriminant, which is $b^2 - 4ac$.
- The methods named `getRoot1()` and `getRoot2()` for returning the two roots of the equation using these formulas:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

```
import math

class QuadraticEquation:
    def __init__(self, a, b, c):
        self.__a = a
        self.__b = b
        self.__c = c

    def getA(self):
        return self.__a

    def getB(self):
        return self.__b

    def getC(self):
        return self.__c

    def getDiscriminant(self):
        return self.__b * self.__b - 4 * self.__a * self.__c

    def getRoot1(self):
        if self.getDiscriminant() < 0:
            return 0
        else:
            return (-self.__b + self.getDiscriminant()) / (2 * self.__a)

    def getRoot2(self):
        if self.getDiscriminant() < 0:
            return 0
        else:
            return (-self.__b - self.getDiscriminant()) / (2 * self.__a)
```

```python
def main():
    a, b, c = eval(input("Enter a, b, c: "))
    equation = QuadraticEquation(a, b, c)
    discriminant = equation.getDiscriminant()

    if discriminant < 0:
        print("The equation has no roots")
    elif discriminant == 0:
        print("The root is", equation.getRoot1())
    else: # (discriminant >= 0)
        print("The roots are", equation.getRoot1(), "and", equation.getRoot2())

main()
```

§4 Inheritance

1. Superclass and subclass

1° Superclass

A general class generalizes common properties and behaviours of different classes.

2° Subclass

The specialized class inherits the properties and methods from the general class.

2. Inheritance

EXAMPLE OF INHERITANCE

- A subclass inherits accessible data fields and methods from its superclass, but it can also have other data fields and methods

Test.py

```
class A:
    def __init__(self,value):
        self.__v = value
    def try (self,x):
        print(self.__v+x)


class B(A):
    def __init__(self, value, newvalue):
        super().__init__(value)
        self.__nv = newvalue
    def getNV(self):
        return self.__nv

b = B(2,3)
b.try (3)
print(b.getNV())
```

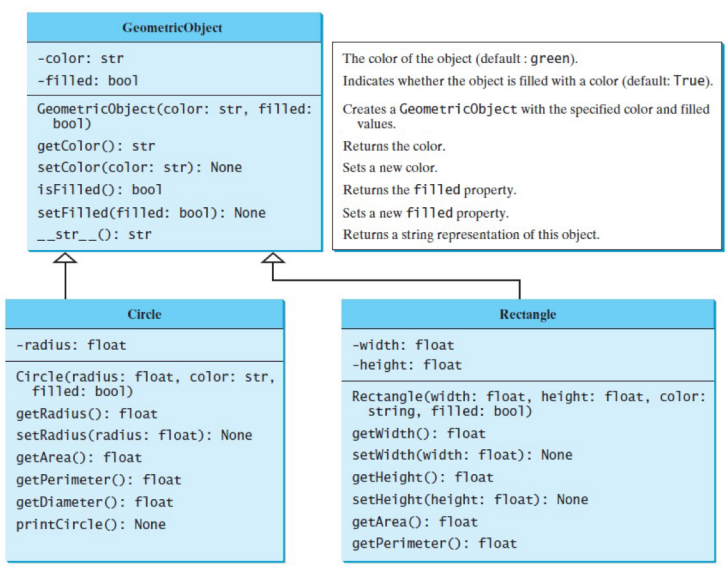
or super(B, self).__init__(value)

or A.__init__(self, value)

Result:

```
5
3
```

GEOMETRIC OBJECT AND TWO OF ITS SUBCLASSES



THE CODE FOR GEOMETRICOBJECT

```
class GeometricObject:                                          GeometricObject class
    def __init__(self, color = "green", filled = True):        initializer
        self.__color = color                                    data fields
        self.__filled = filled

    def getColor(self):                                         getColor
        return self.__color

    def setColor(self, color):                                  setColor
        self.__color = color

    def isFilled(self):                                         isFilled
        return self.__filled

    def setFilled(self, filled):
        self.__filled = filled

    def __str__(self):
        return "color: " + self.__color + \
            " and filled: " + str(self.__filled)
```

THE CODE FOR CIRCLE CLASS

- A subclass inherits accessible data fields and methods from its superclass, but it can also have other data fields and methods

```
from GeometricObject import GeometricObject
import math # math.pi is used in the class

class Circle(GeometricObject):
    def __init__(self, radius):
        super().__init__()
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def setRadius(self, radius):
        self.__radius = radius

    def getArea(self):
        return self.__radius * self.__radius * math.pi

    def getDiameter(self):
        return 2 * self.__radius

    def getPerimeter(self):
        return 2 * self.__radius * math.pi

    def printCircle(self):
        print(self.__str__() + " radius: " + str(self.__radius))
```

Circle class inherits the methods getColor, setColor, isFilled, setFilled, and __str__

The printCircle method invokes the __str__() method defined to obtain properties defined in the superclass

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

THE CODE FOR RECTANGLE CLASS

```
from GeometricObject import GeometricObject

class Rectangle(GeometricObject):                               extend superclass
    def __init__(self, width = 1, height = 1):                 initializer
        super().__init__()                                      superclass initializer
        self.__width = width
        self.__height = height

    def getWidth(self):                                         methods
        return self.__width

    def setWidth(self, width):
        self.__width = width

    def getHeight(self):
        return self.__height

    def setHeight(self, height):
        self.__height = self.__height

    def getArea(self):
        return self.__width * self.__height

    def getPerimeter(self):
        return 2 * (self.__width + self.__height)
```

THE CODE FOR TESTING CIRCLE AND RECTANGLE

```
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    circle = Circle(1.5)
    print("A circle", circle)
    print("The radius is", circle.getRadius())
    print("The area is", circle.getArea())
    print("The diameter is", circle.getDiameter())

    rectangle = Rectangle(2, 4)
    print("\nA rectangle", rectangle)
    print("The area is", rectangle.getArea())
    print("The perimeter is", rectangle.getPerimeter())

main() # Call the main function
```

```
A circle color: green and filled: True
The radius is 1.5
The area is 7.06858347058
The diameter is 3.0

A rectangle color: green and filled: True
The area is 8
The perimeter is 12
```

1° A subclass contains more information and methods than

its superclass.

2° A subclass and its superclass must have the is-a relationship.
But not all is-a relationship should be modelled using inheritance.

3. Overriding methods

1° Sometimes it's necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

EXAMPLE

- The __str__() method in the GeometricObject class returns the string describing a geometric object. This method can be overridden to return the string describing a circle

```
class Circle(GeometricObject):
    # Other methods are omitted

    # Override the __str__ method defined in GeometricObject
    def __str__(self):
        return super().__str__() + " radius: " + str(radius)
```

__str__ in superclass

- The __str__() method is defined in the GeometricObject class and modified in the Circle class. Both methods can be used in the Circle class. To invoke the __str__ method defined in the GeometricObject class from the Circle class, use super().__str__()

PRACTICE

What would be the output of the following program?

```
class A:
    def __init__(self, i = 0):
        self.i = i

    def m1(self):
        self.i += 2

class B(A):
    def __init__(self, j = 0):
        super().__init__(3)
        self.j = j

    def m1(self):
        self.i += 1

def main():
    b = B()
    b.m1()
    print(b.i)
    print(b.j)

main() # Call the main function
```

Result:

```
4
0
```

NAME MANGLING WITH SUPERCLASS

```
class Test:
    def __init__(self):
        self.foo = 1
        self._bar = 2
        self.__baz = 3

class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden'
    def printbaz(self):
        print(self.foo)
        print(self._bar)
        print(self.__baz)
```

```
e = ExtendedTest()
e.printbaz()
print(dir(e))
print(e._Test__baz)
print(e._ExtendedTest__baz)
```

```
overridden
overridden
overridden
['_ExtendedTest__baz', '_Test__baz', '__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_bar', 'foo', 'printbaz']
3
overridden
```

CAN I GO WITHOUT __INIT__() ?

```
class Student:
    def __init__(self):
        self.name = "Student"

    def printStudent(self):
        print(self.name)

class GraduateStudent(Student):

    def printStudent(self):
        print("Postgraduate: "+self.name)


a = Student()
b = GraduateStudent()
a.printStudent()
b.printStudent()
```

If you don't have to pass arguments for initializer function, maybe you can go without it!

In that case, you did not override it!

By default, the super().__init__() was called in default __init__() function

```
Student
Postgraduate: Student
```

2) Private

```
class Student:
    def __init__(self):
        self.__name = "Student"

    def printStudent(self):
        print(self.__name)

class GraduateStudent(Student):

    def printStudent(self):
        print("Postgraduate: "+self.__name)


a = Student()
b = GraduateStudent()
a.printStudent()
b.printStudent()
```

```
Student
Traceback (most recent call last):
  File "test70.py", line 16, in <module>
    b.printStudent()
  File "test70.py", line 10, in printStudent
    print("Postgraduate: "+self.__name)
AttributeError: 'GraduateStudent' object has no
attribute '_GraduateStudent__name'
```