LEARNING TO CODE

python™

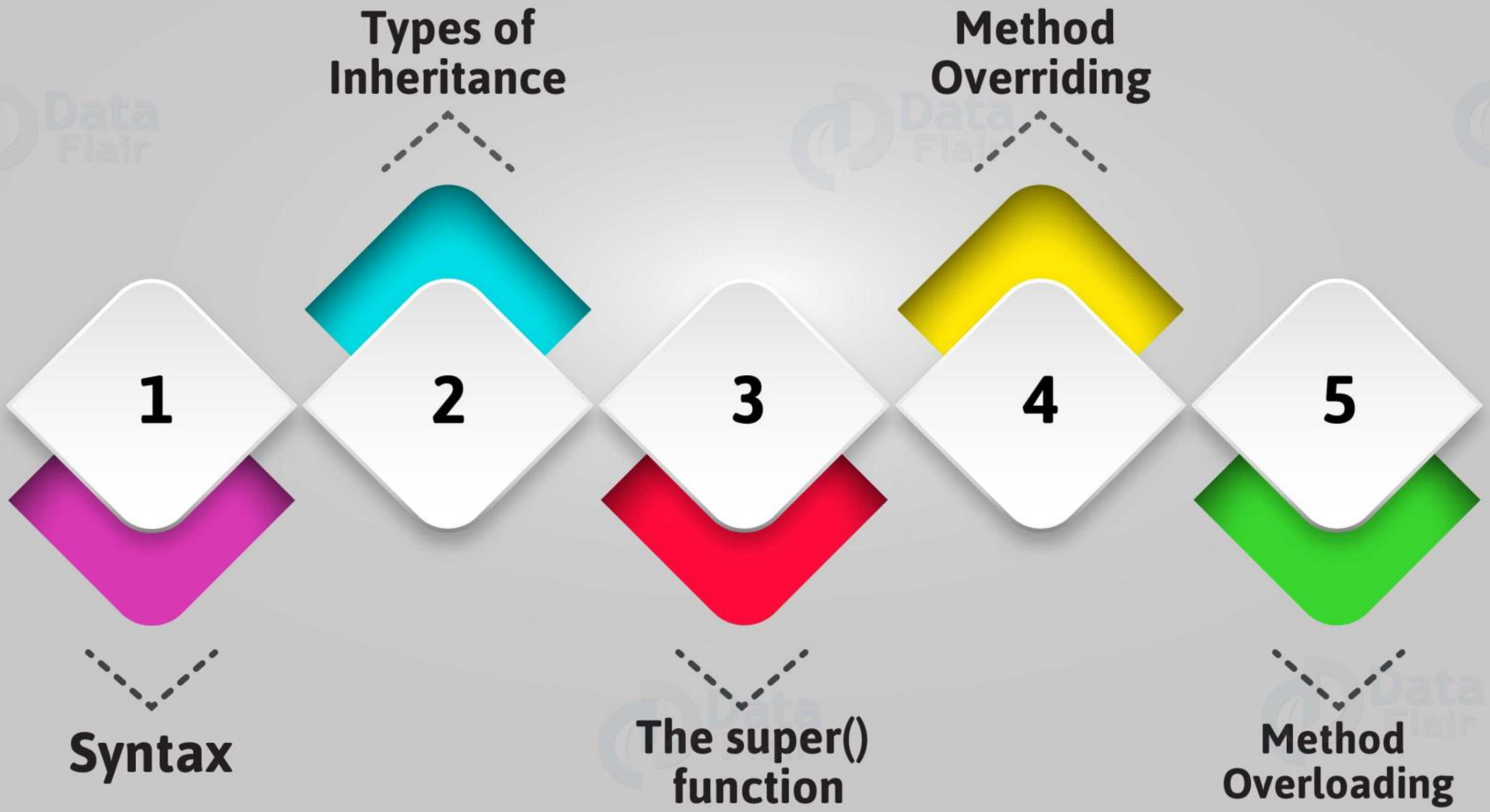
Coordinator: Konstantinos Emmanouilidis

IEEE SB OF THRACE

2017-2018

1. Inheritance
2. Operator Overloading

Python Inheritance



1 Syntax

2 Types of Inheritance

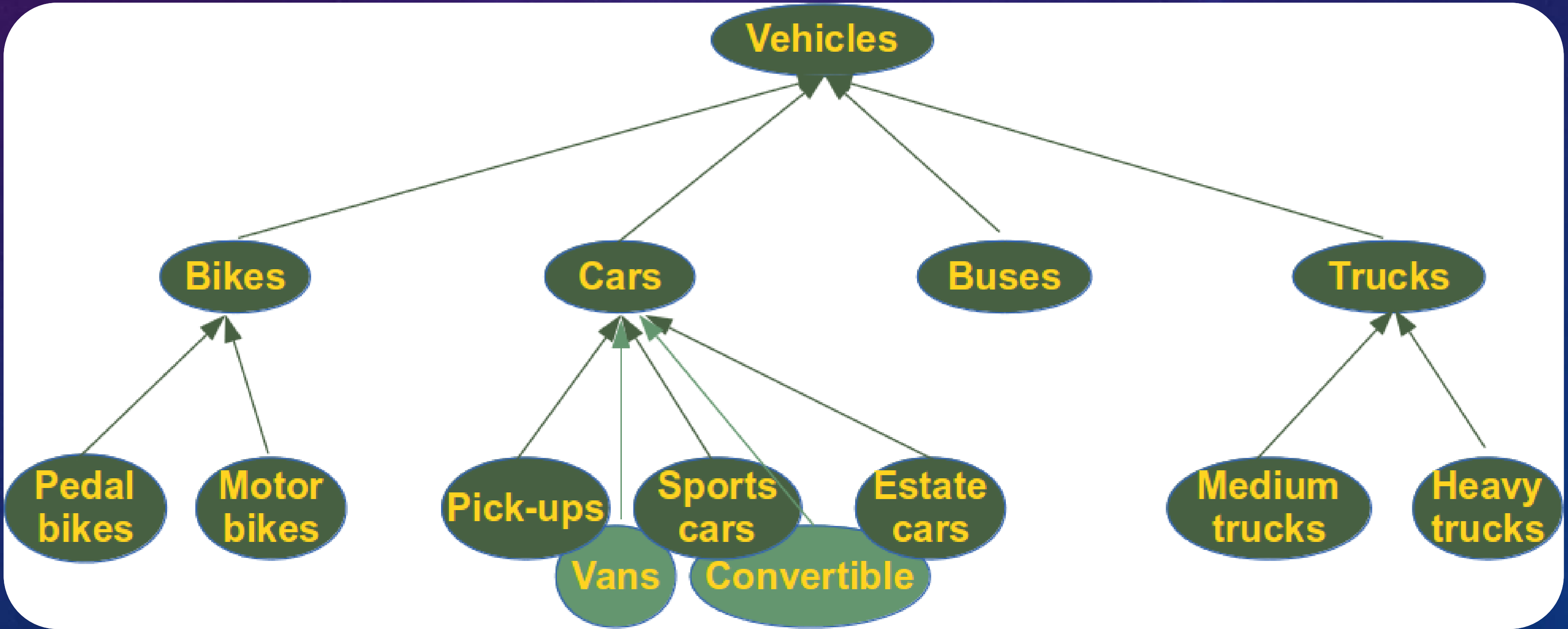
3 The super() function

4 Method Overriding

5 Method Overloading

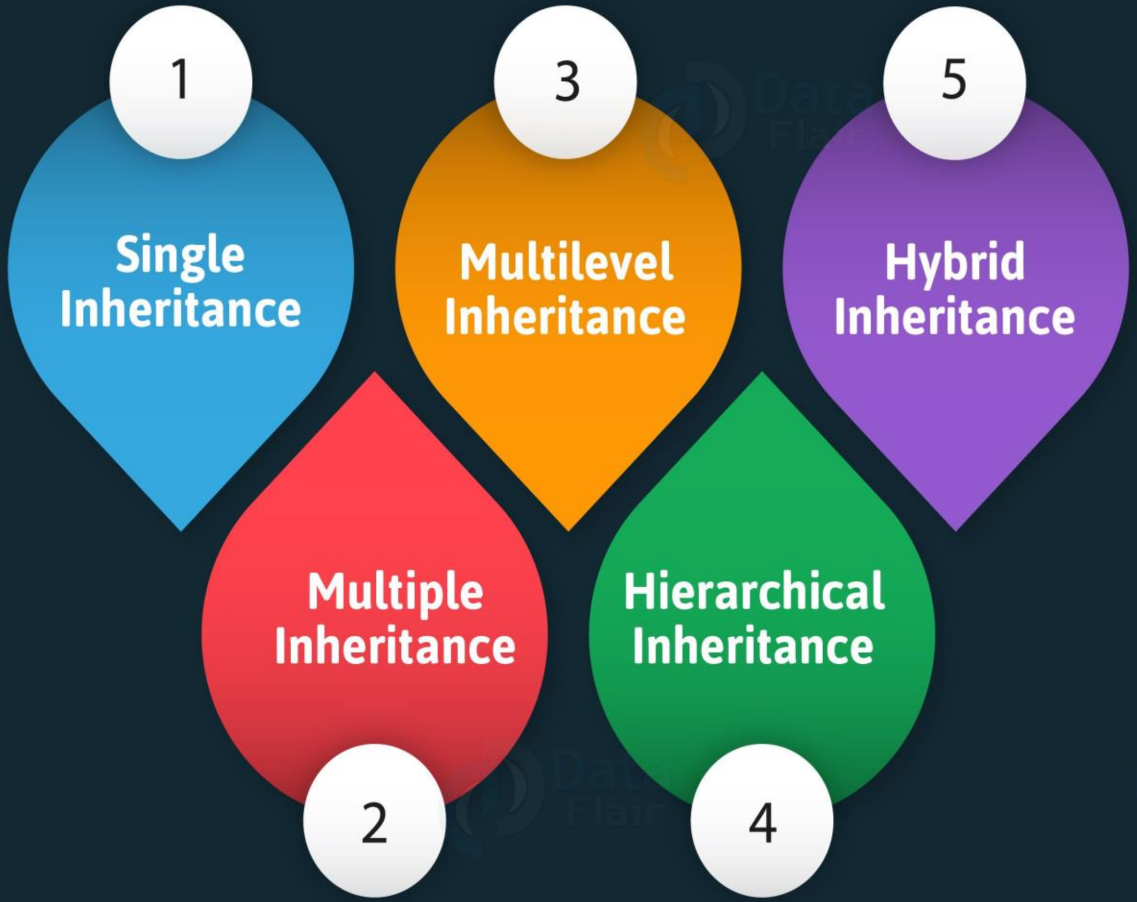
INHERITANCE

- A class uses code constructed within another classs
- Classes called **child classes** or **subclasses** inherit methods and variables from **parent classes** or **base classes**.



Python

Types of Inheritance



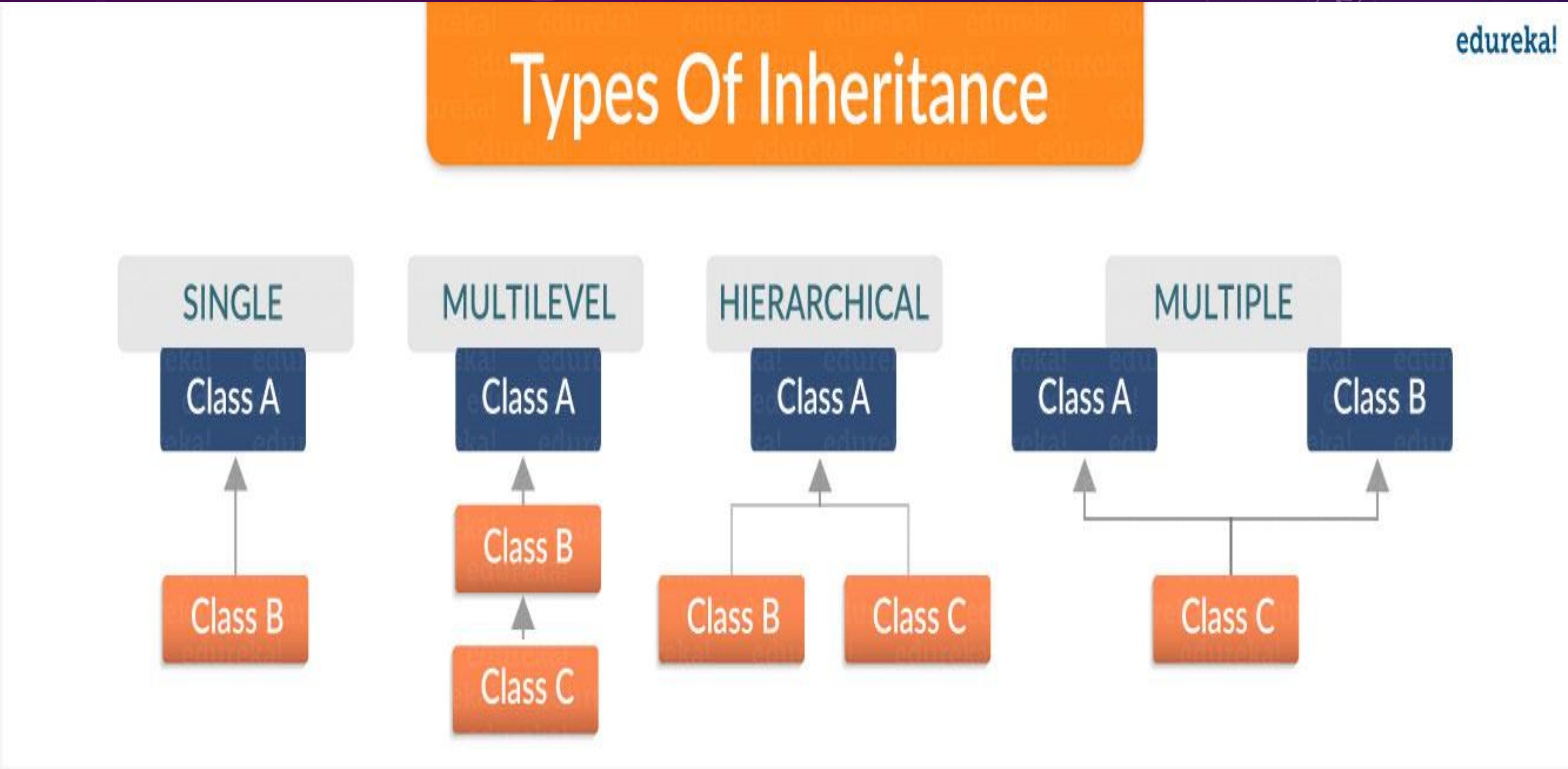
1 Single Inheritance

2 Multiple Inheritance

3 Multilevel Inheritance

4 Hierarchical Inheritance

5 Hybrid Inheritance



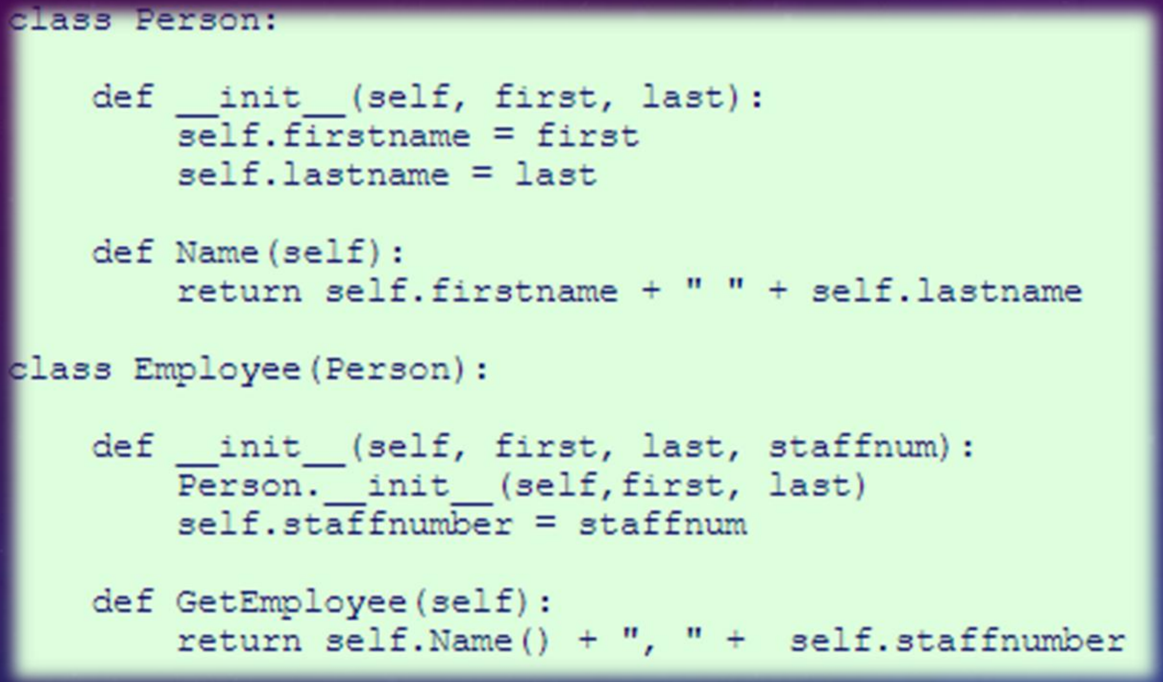
Types Of Inheritance

SYNTAX

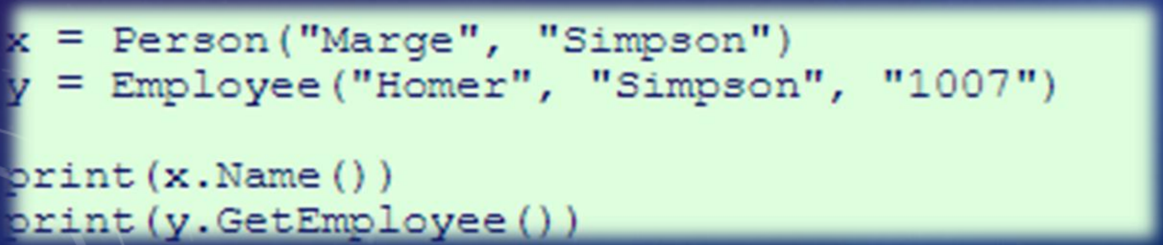
- **Derived class definition:**

```
class DerivedClassName(BaseClassName):
    commands
    .
    .
    .
```

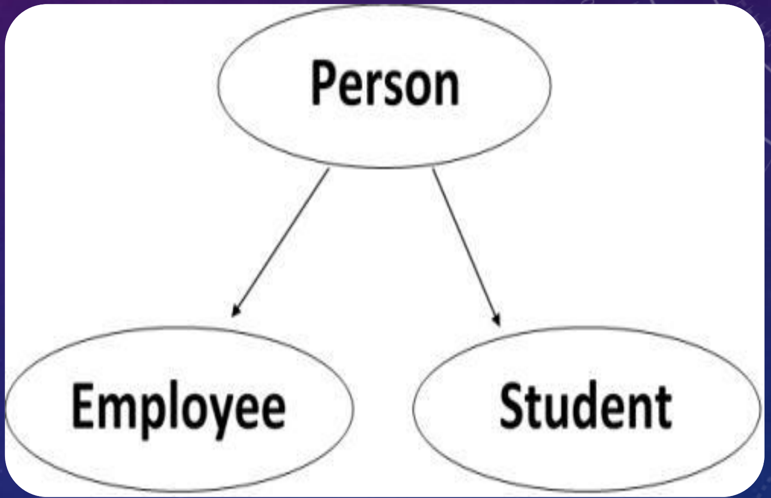
EXAMPLE

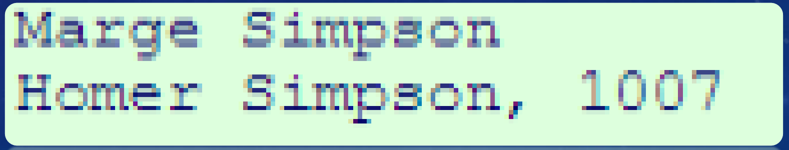


```
class Person:

    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):

    def __init__(self, first, last, staffnum):
        Person.__init__(self,first, last)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.Name() + ", " +  self.staffnumber
```



```
x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")

print(x.Name())
print(y.GetEmployee())
```



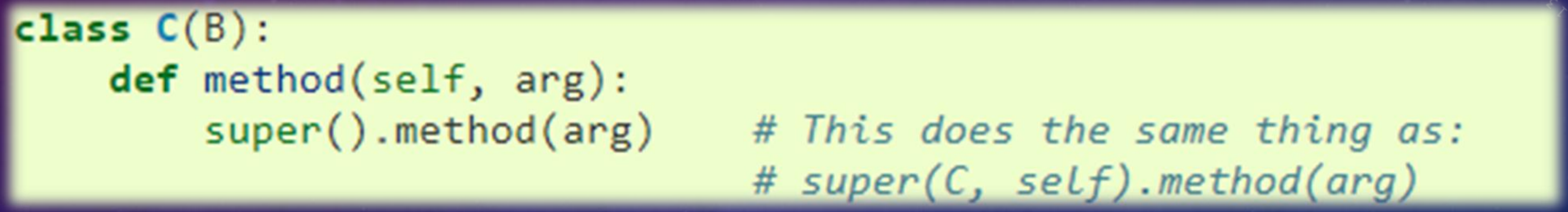


```
Marge Simpson
Homer Simpson, 1007
```

SUPER FUNCTION

- Use Case 1:
  - In a single inheritance in order to refer to the parent class or multiple classes without explicitly naming them.
  - It helps keep your code maintainable for the foreseeable future.
- Use Case 2:
  - Use in a dynamic execution environment for multiple or collaborative inheritance.
  - This use is considered exclusive to Python (not possible with languages that only support single inheritance).
  - Good design dictates that this method have the same calling signature in every case (the order of adapts to changes in the class hierarchy).

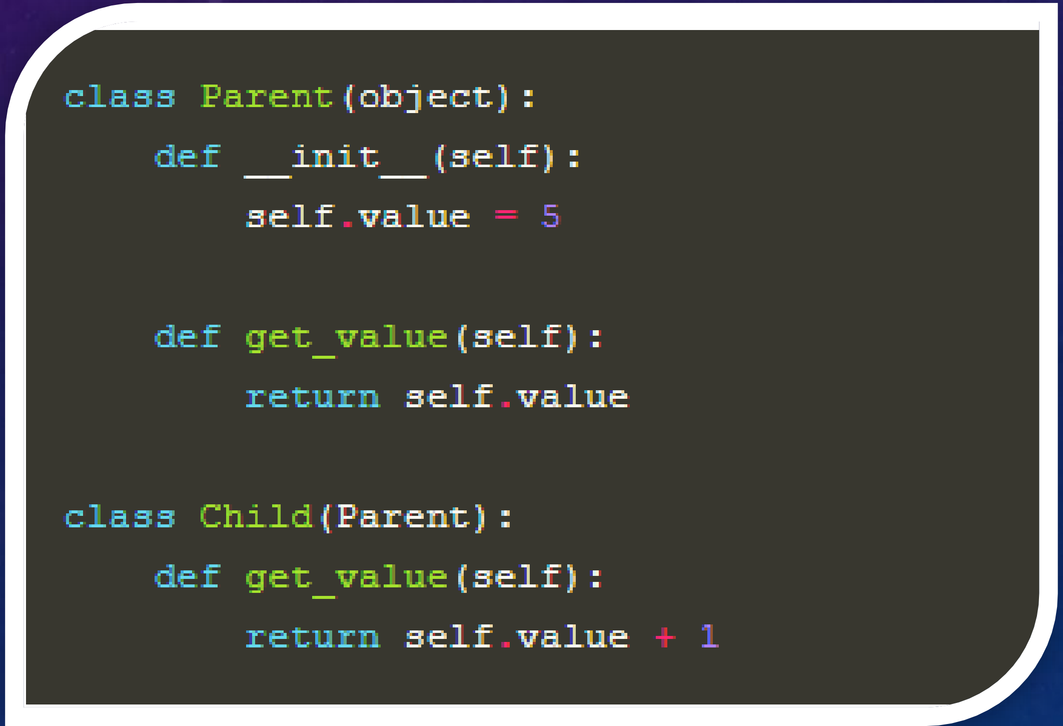
SYNTAX



```
class C(B):
    def method(self, arg):
        super().method(arg)    # This does the same thing as:
                               # super(C, self).method(arg)
```

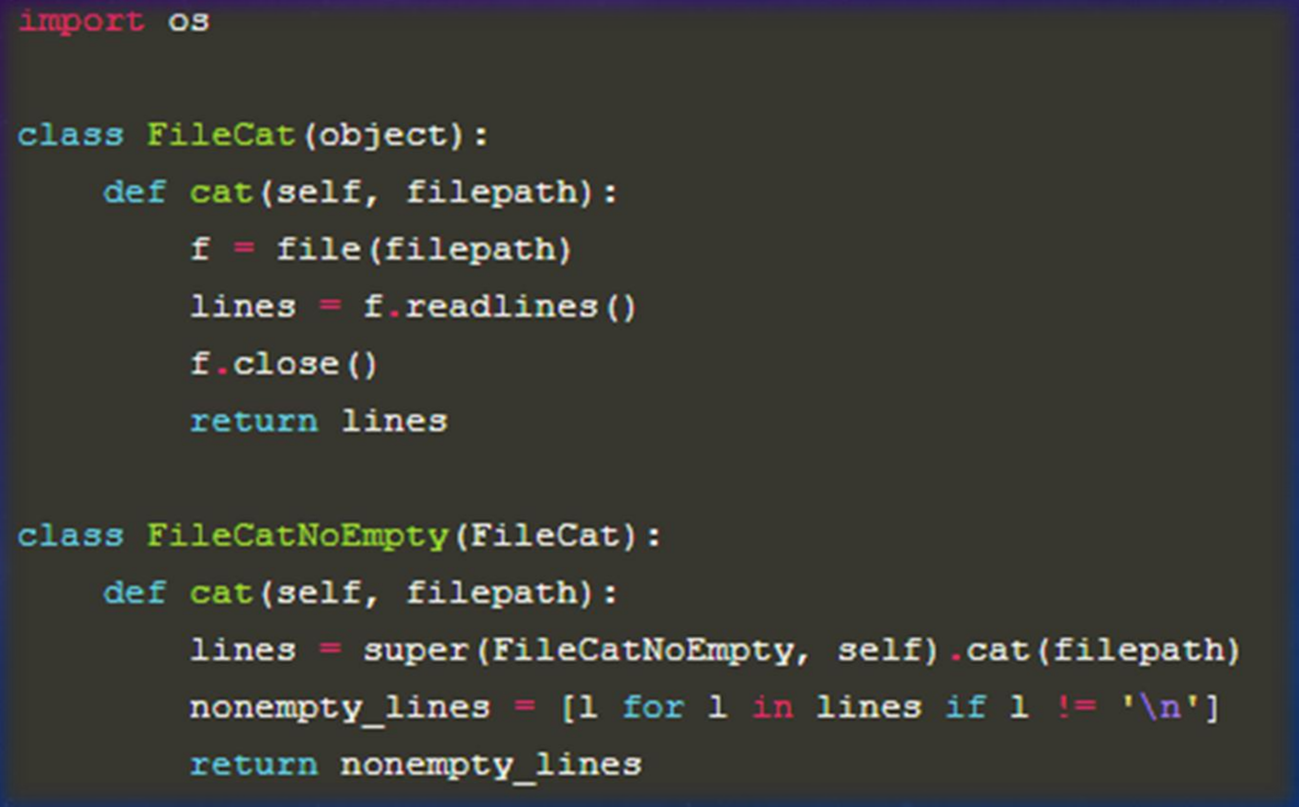
OVERRIDING

Overriding occurs simply defining in the child class
a method with the same name of a method in the parent class.



```
class Parent(object):
    def __init__(self):
        self.value = 5

    def get_value(self):
        return self.value

class Child(Parent):
    def get_value(self):
        return self.value + 1
```

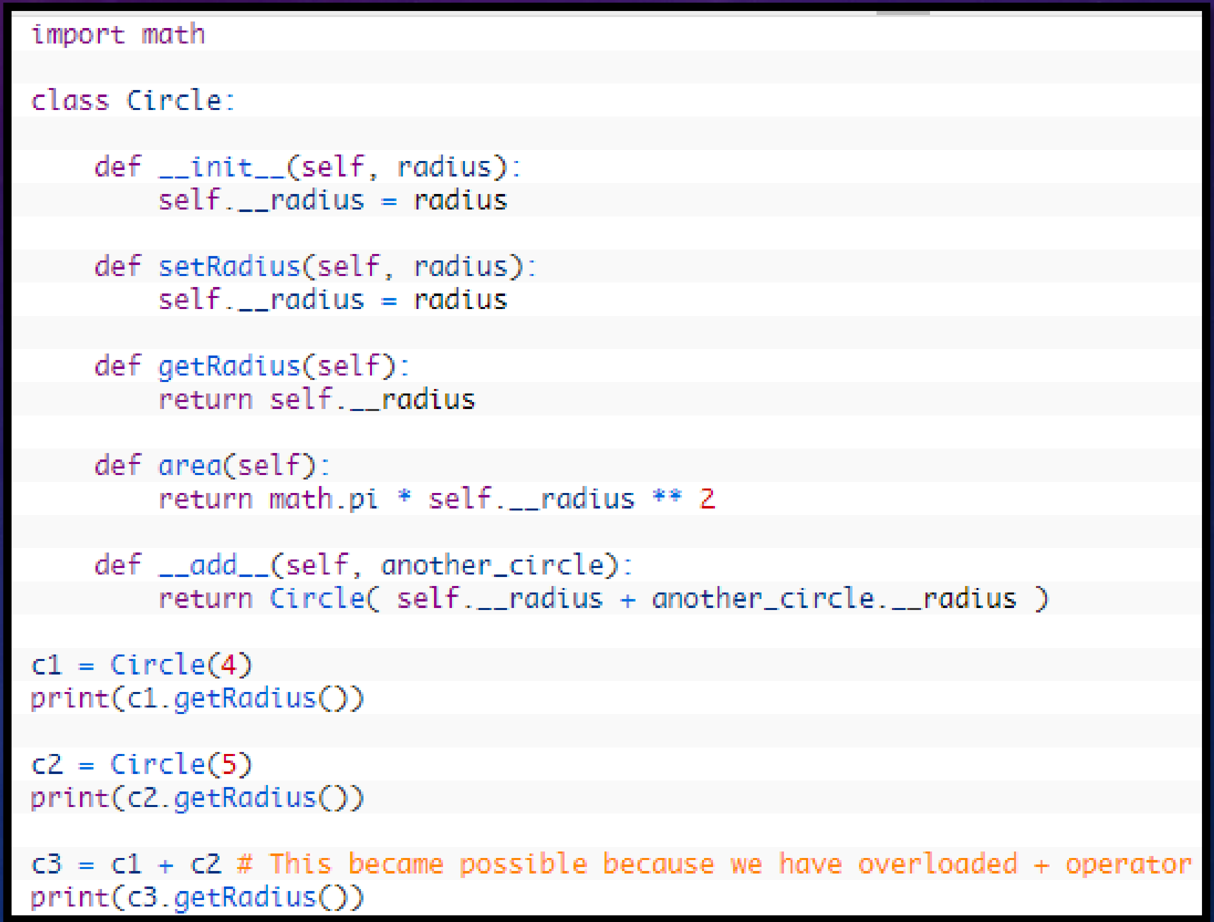
- Use super() for Python 3.x to call the original implementation of a method.
- This respects the resolution order in case of multiple inheritance and, for Python 3.x, protects from changes in the class hierarchy.



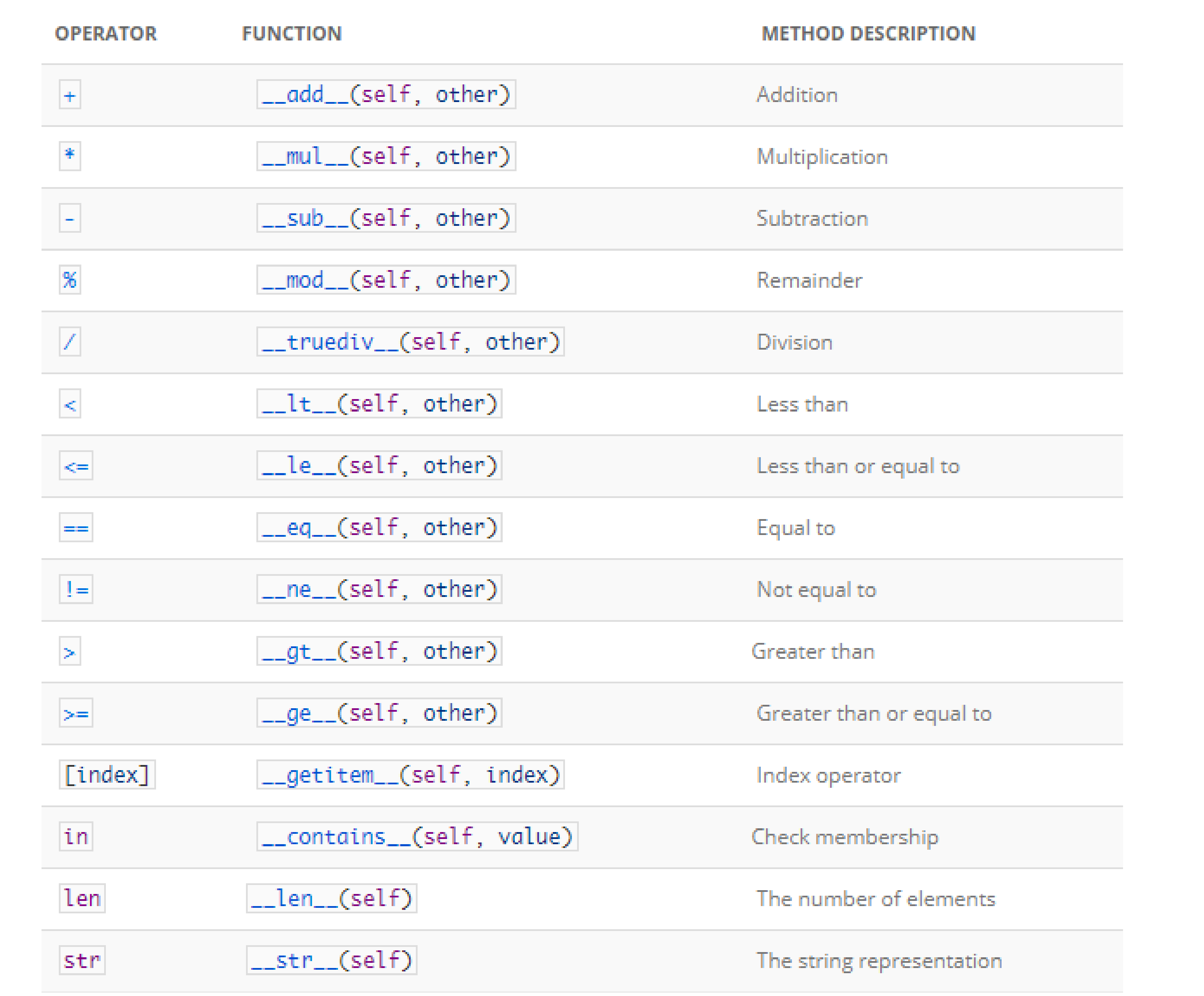
```python
import os

class FileCat(object):
    def cat(self, filepath):
        f = file(filepath)
        lines = f.readlines()
        f.close()
        return lines

class FileCatNoEmpty(FileCat):
    def cat(self, filepath):
        lines = super(FileCatNoEmpty, self).cat(filepath)
        nonempty_lines = [l for l in lines if l != '\n']
        return nonempty_lines
```

OPERATOR OVERLOADING

Defining methods for operators is known as operator overloading.

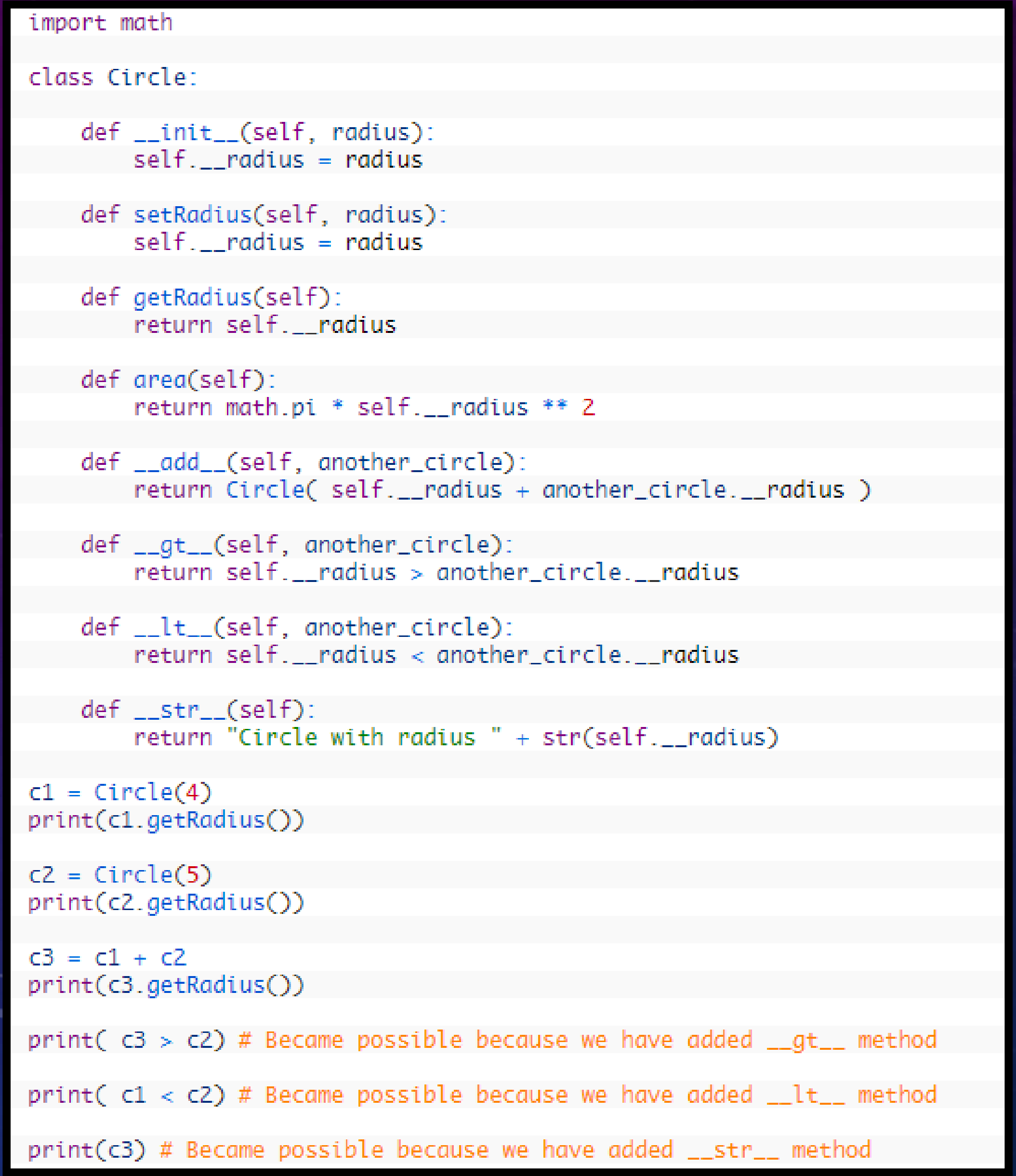


```
import math

class Circle:

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

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

    def getRadius(self):
        return self.__radius

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

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2 # This became possible because we have overloaded + operator
print(c3.getRadius())
```

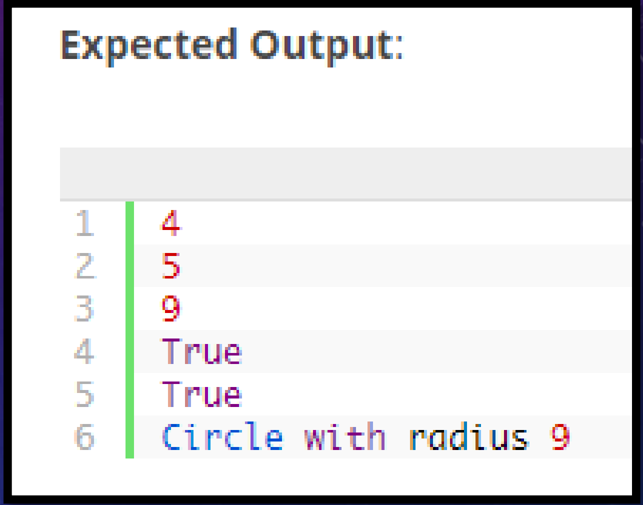


OPERATOR	FUNCTION	METHOD DESCRIPTION
`+`	`__add__(self, other)`	Addition
`*`	`__mul__(self, other)`	Multiplication
`-`	`__sub__(self, other)`	Subtraction
`%`	`__mod__(self, other)`	Remainder
`/`	`__truediv__(self, other)`	Division
`<`	`__lt__(self, other)`	Less than
`<=`	`__le__(self, other)`	Less than or equal to
`==`	`__eq__(self, other)`	Equal to
`!=`	`__ne__(self, other)`	Not equal to
`>`	`__gt__(self, other)`	Greater than
`>=`	`__ge__(self, other)`	Greater than or equal to
`[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

EXAMPLE



```python
import math

class Circle:

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

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

    def getRadius(self):
        return self.__radius

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

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

    def __gt__(self, another_circle):
        return self.__radius > another_circle.__radius

    def __lt__(self, another_circle):
        return self.__radius < another_circle.__radius

    def __str__(self):
        return "Circle with radius " + str(self.__radius)

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2
print(c3.getRadius())

print( c3 > c2) # Became possible because we have added __gt__ method

print( c1 < c2) # Became possible because we have added __lt__ method

print(c3) # Became possible because we have added __str__ method
```



Expected Output:

```
4
5
9
True
True
Circle with radius 9
```