Jawaharlal Nehru Technological University Hyderabad

S C D E

Kukatpally, Hyderabad - 500 085, Telangana, India

Object Oriented Programming using Python part 3

Session 14 , 1 June 23

Dr N V Ganapathi Raju

Professor and HOD of IT

Gokaraju Rangaraju Institute of Eng and Tech

Introduction to Polymorphism

- Polymorphism is one of the important feature of Object-Oriented Programming (OOP).
- Polymorphism means that you can have multiple classes where each class implements the same variables or methods in different ways.
- It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.
    - A real-world example of polymorphism is suppose when if you are in classroom that time you behave like a student, when you are in market at that time you behave like a customer, when you are at your home at that time you behave like a son or daughter, such that same person is presented as having different behaviors.

Polymorphism in addition operator

- The + operator is used extensively in Python programs, have many usages.
- For integer data types, + operator is used to perform arithmetic addition operation.

```
num1, num2 = 10,20

print(num1+num2) # 30
```

- Similarly, for string data types, + operator is used to perform concatenation.

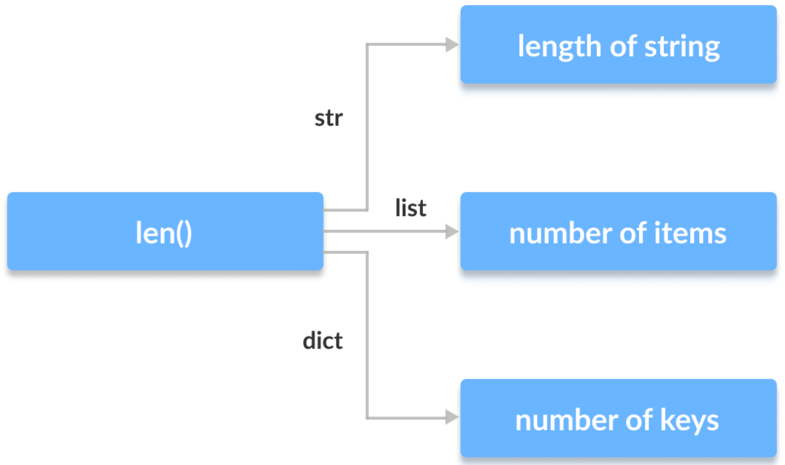
```
str1 = "Ten“

str2 = "Twenty“

print(str1+" "+str2) # Ten Twenty
```

- A single operator + has been used to carry out different operations for distinct data types.

Function Polymorphism

- **There are some functions in Python which are compatible to run with multiple data types.**
- **One such function is the len() function. It can run with many data types in Python.**

```
print(len("Conduira Online"))   # 15

print(len(["Python", "Java", "C"]))  # 3

print(len({"Organization": "Conduira", "Address": "Hyderabad"})) # 2
```



Note:

Data types such as string, list, tuple, set, and dictionary can work with the len() function. However, we can see that it returns specific information about specific data types.

Class Polymorphism in Python

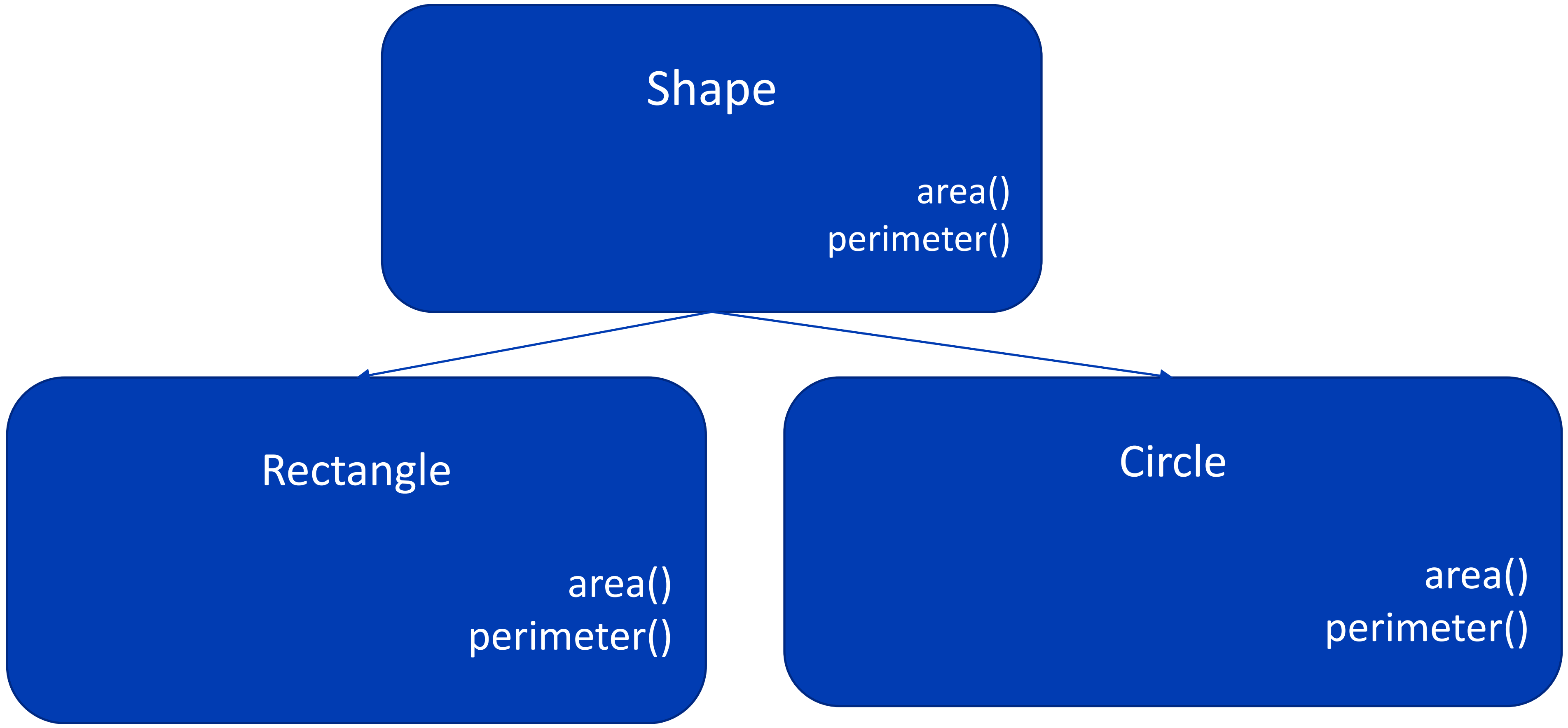
- We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

```
class Shark():
    def swim(self):
        print("The shark is swimming.")
    def swim_backwards(self):
        print("The shark cannot swim backwards but can sink backwards.")
    def skeleton(self):
        print("The shark's skeleton is made of cartilage.")

class Clownfish():
    def swim(self):
        print("The clownfish is swimming.")
    def swim_backwards(self):
        print("The clownfish can swim backwards.")
    def skeleton(self):
        print("The clownfish's skeleton is made of bone.")
```

```
sammy = Shark()
sammy.skeleton()

casey = Clownfish()
casey.skeleton()
```

```
The shark's skeleton is made of cartilage.

The clownfish's skeleton is made of bone.
```

Demonstrate Polymorphism



Demonstrate Polymorphism

```
import math
class Shape:
    def area(self):
        pass
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        print(f"Area of Rectangle is {self.width * self.height}")
    def perimeter(self):
        print(f"Perimeter of Rectangle is {2 * (self.width + self.height)}")
```

Demonstrate Polymorphism

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        print(f"Area of Circle is {math.pi * self.radius ** 2}")

    def perimeter(self):
        print(f"Perimeter of Circle is {2 * math.pi * self.radius}")
def shape_type(shape_obj):
    shape_obj.area()
    shape_obj.perimeter()

rectangle_obj = Rectangle(10, 20)
circle_obj = Circle(10)

for each_obj in [rectangle_obj, circle_obj]:
    shape_type(each_obj)
```

```
Area of Rectangle is 200
Perimeter of Rectangle is 60
Area of Circle is 314.1592653589793
Perimeter of Circle is 62.83185307179586
```

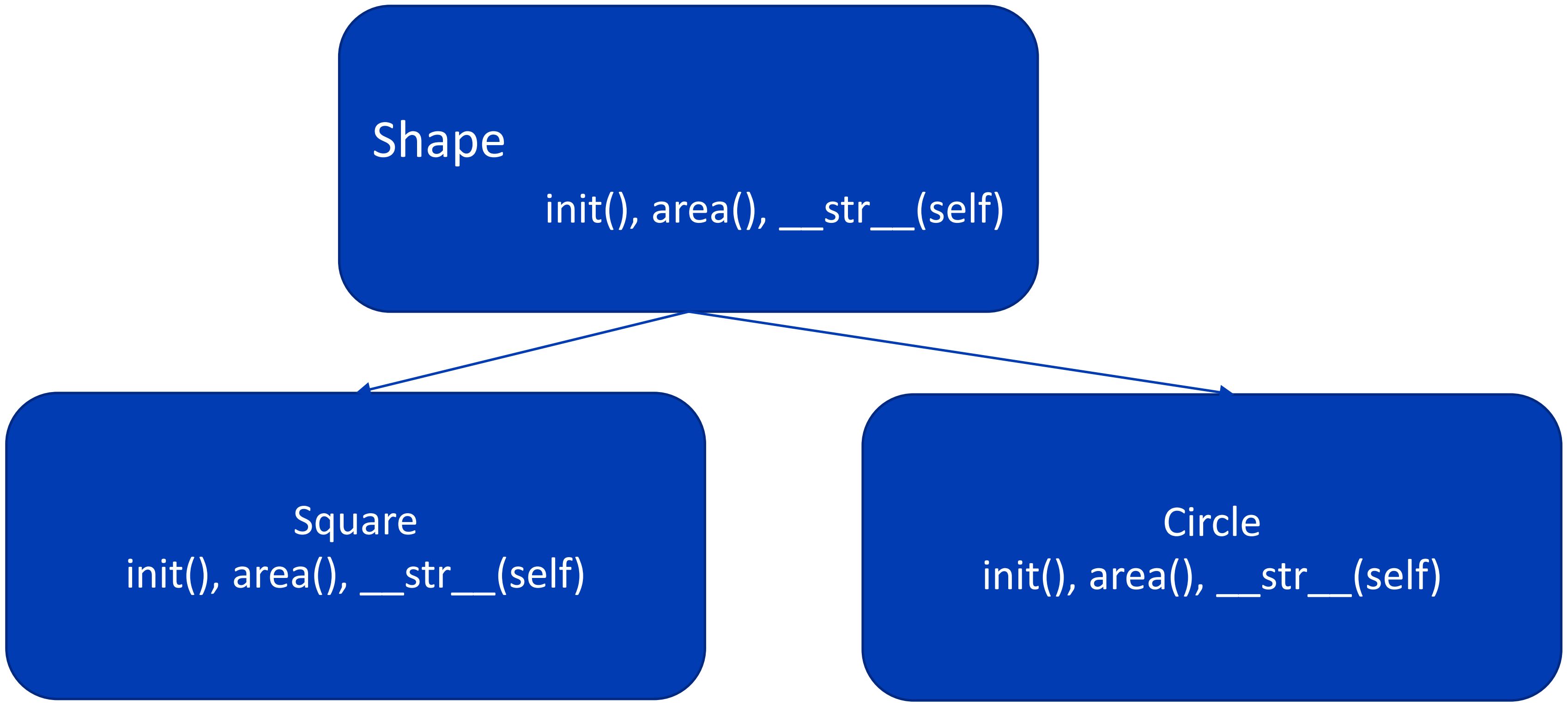
Explanation

- Shape is the base class while Rectangle and Circle – are the derived classes.
- All these classes have common methods area() and perimeter() added to them but their implementation is different as found in each class.
- Derived classes Rectangle and Circle have their own data attributes.
- Instance variables rectangle_obj and circle_obj are created for Rectangle and Circle classes respectively.
- The clearest way to express polymorphism is through the function shape_type() – , which takes any object and invokes the methods area() and perimeter() respectively.

Method Overriding

- Like in other programming languages, the child classes in Python also inherit methods and attributes from the parent class.
- We can redefine certain methods and attributes specifically to fit the child class, which is known as **Method Overriding**.
- Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.

Method Overriding


Shape
init(), area(), __str__(self)
Square
init(), area(), __str__(self)
Circle
init(), area(), __str__(self)

Method Overriding

```
from math import pi

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

    def area(self):
        pass

    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
```

Method Overriding

```
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(a)
print(a.area())
print(b)
print(b.area())
```

```
Square
16
Circle
153.93804002589985
```

The methods __str__(), which have not been overridden in the child classes, are used from the parent class.

Python Data Model

Note:

- If we use + or * operator on a str object in Python, we must have noticed its different behavior when compared to int or float objects.

```
print(1 + 2)                              3

# Concatenates the two strings
print(‘Hai ' + 'Conduira')                Hai Conduira

# Gives the product
print(3 * 2)                              6

# Repeats the string
print('Conduira! ' * 3)                   Conduira! Conduira! Conduira!
```

- We have observed that the same built-in operator or function shows different behavior for objects of different classes.
- This is called operator overloading or function overloading respectively.

Data Model

- Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.
- Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory.
- The 'is' operator compares the identity of two objects; the id() function returns an integer representing its identity.
- Data model as a description of Python as a framework. It formalizes the interfaces of the building blocks of the language itself, such as sequences, iterators, functions, classes, context managers, and so on.
- **While coding with any framework, you spend a lot of time implementing methods that are called by the framework. The same happens when you leverage the Python data model. The Python interpreter invokes special methods to perform basic object operations, often triggered by special syntax.**

Python Data Model

Case Example:

- Suppose we have a class representing an online order having a cart (a list) and a customer (a str or instance of another class which represents a customer).
  - In such a case, it is quite natural to want to obtain the length of the cart list. Someone new to Python might decide to implement a method called get_cart_len() in their class to do this. But we can configure the built-in len() in such a way that it returns the length of the cart list when given our object.
  - In another case, we might want to append something to the cart. Again, someone new to Python would think of implementing a method called append_to_cart() that takes an item and appends it to the cart list. But you can configure the + operator in such a way that it appends a new item to the cart.

Python Special (Magic) Methods

- Special methods have a naming convention, where the name starts with **two underscores, followed by an identifier and ends with another pair of underscores**.
- Essentially, each built-in function or operator has a special method corresponding to it.
- For example, there's **__len__(),** corresponding to len(), and **__add__(),** corresponding to the + operator.
- By default, most of the built-ins and operators will not work with objects of your classes. We must add the corresponding special methods in our class definition to make your object compatible with built-ins and operators.
- The behavior of the function or operator associated with it changes according to that defined in the method. This is exactly what the **Data Model** helps you accomplish.

Internals of Operations Like len() and []

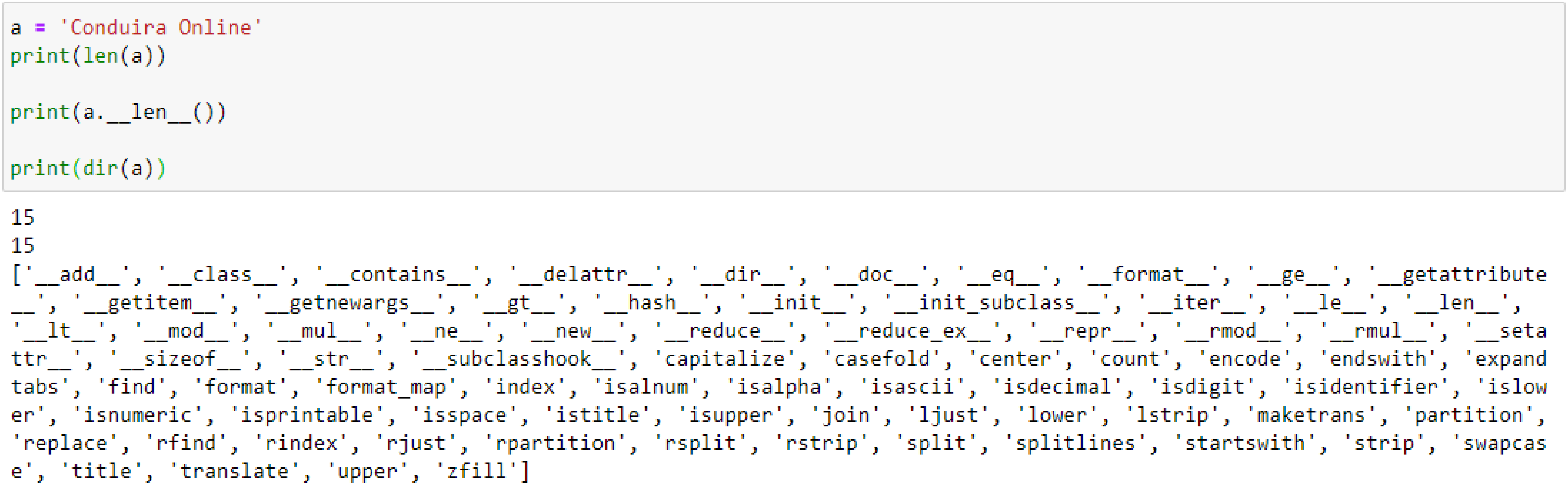
- When we calling len() on an object, Python handles the call as obj.__len__().
- When you use the [] operator on an iterable to obtain the value at an index, Python handles it as itr.__getitem__(index), where itr is the iterable object and index is the index you want to obtain.

```
a = 'Conduira Online'
b = ['Conduira', 'Online']
print(len(a))                       15

print(a.__len__())                  15

print(b[0])                         Conduira

print(b.__getitem__(0))             Conduira
```

- When you use the function or its corresponding special method, we get the same result.

Special methods of string object

- when we obtain the list of attributes and methods of a str object using dir(), we'll see these special methods in the list in addition to the usual methods available on str objects.



```python
a = 'Conduira Online'
print(len(a))

print(a.__len__())

print(dir(a))
```

```
15
15
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute
__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__seta
ttr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expand
tabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islow
er', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcas
e', 'title', 'translate', 'upper', 'zfill']
```

Python special methods

The special method names allow your objects to implement, support, and interact with basic language constructs such as:

- Iteration
- Collections
- Attribute access
- Operator overloading
- Function and method invocation
- Object creation and destruction
- String representation and formatting

Operator Overloading

- Operator overloading in Python is the ability of a single operator to perform more than one operation based on the class (type) of operands.

- For example, the + operator can be used to add two numbers, concatenate two strings or merge two lists.
  - This is possible because the + operator is overloaded with int and str classes.

- Similarly, we can define additional methods for these operators to extend their functionality to various new classes and this process is called Operator overloading.

Giving a Length to Your Objects Using len()

- To change the behavior of len(), we need to define the __len__() special method in ourclass.
- Whenever we pass an object of your class to len(), our custom definition of __len__() will be used to obtain the result.

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer

    def __len__(self):
        return len(self.cart)

order = Order(['C', 'Java', 'Python'], 'Conduira Online')
print(len(order))
print(order.customer)
```

```
3
Conduira Online
```

Indexing and Slicing Your Objects Using []

- The [] operator is called the indexing operator and is used in various contexts in Python such as getting the value at an index in sequences, getting the value associated with a key in dictionaries, or obtaining a part of a sequence through slicing.
- We can change its behavior using the __getitem__() special method.

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer

    def __getitem__(self, key):
        return self.cart[key]
```

```
order = Order(['C', 'Python', 'Java'], 'Conduira Online')

print(order[0]) # C

print(order[-1]) # Java
```

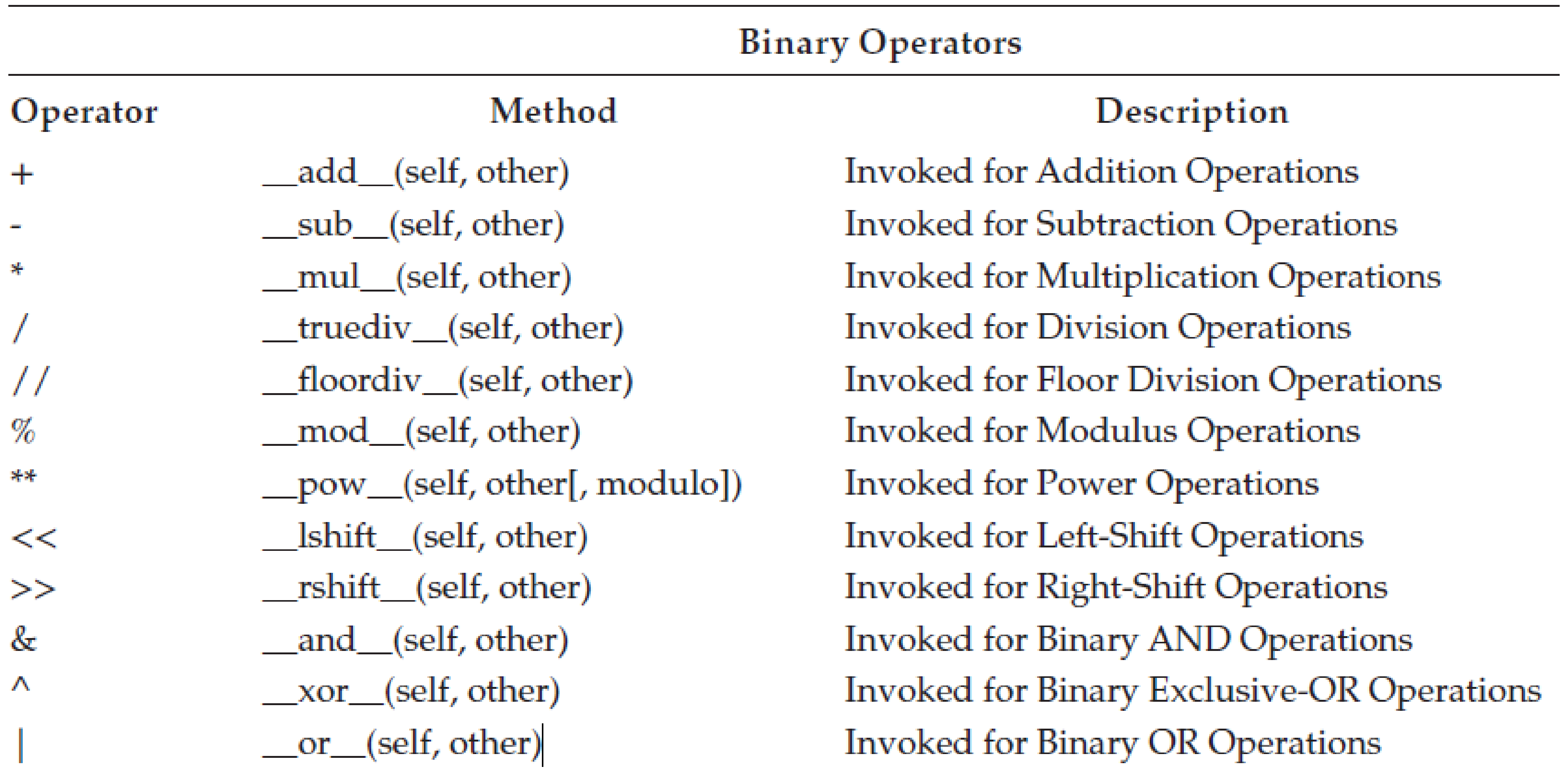
String Magic Methods

String Magic Methods	Description
__str__(self)	To get called by built-int str() method to return a string representation of a type.
__hash__(self)	To get called by built-int hash() method to return an integer.
__nonzero__(self)	To get called by built-int bool() method to return True or False.
__dir__(self)	To get called by built-int dir() method to return a list of attributes of a class.
__sizeof__(self)	To get called by built-int sys.getsizeof() method to return the size of an object.

Attribute Magic Methods

Attribute Magic Methods	Description
__getattr__(self, name)	Is called when the accessing attribute of a class.
__setattr__(self, name, value)	Is called when assigning a value to the attribute of a class.
__delattr__(self, name)	Is called when deleting an attribute of a class.

Operator Overloading and Magic Methods



Binary Operators

Operator	Method	Description
+	__add__(self, other)	Invoked for Addition Operations
-	__sub__(self, other)	Invoked for Subtraction Operations
*	__mul__(self, other)	Invoked for Multiplication Operations
/	__truediv__(self, other)	Invoked for Division Operations
//	__floordiv__(self, other)	Invoked for Floor Division Operations
%	__mod__(self, other)	Invoked for Modulus Operations
**	__pow__(self, other[, modulo])	Invoked for Power Operations
<<	__lshift__(self, other)	Invoked for Left-Shift Operations
>>	__rshift__(self, other)	Invoked for Right-Shift Operations
&	__and__(self, other)	Invoked for Binary AND Operations
^	__xor__(self, other)	Invoked for Binary Exclusive-OR Operations
|	__or__(self, other)	Invoked for Binary OR Operations

Dynamic typed language

- x = 10

- x = “conduira online”

- x = 10.50

Duck Typing:

- if it walks like a duck and it quacks like a duck, then it must be a duck”

Duck Typing

- if it walks like a duck and it quacks like a duck, then it must be a duck”

- Duck typing is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines.
- Using duck typing you do not check types at all. Instead you check for the presence of a given method or attribute.
- **Duck Typing** is a type system used in dynamic languages. For example, Python, Perl, Ruby, PHP, Javascript, etc. where the type or the class of an object is less important than the method it defines.
- Using Duck Typing, we do not check types at all. Instead, we check for the presence of a given method or attribute.

Duck Typing example

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(obj):
    obj.quack()
    obj.feathers()
```

```
donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

```
Quaaaaaack!
The duck has white and gray feathers.
The person imitates a duck.
The person takes a feather from the ground and shows it.
```

Duck Typing example

```
class Anaconda:
    def execute(self):
        print("Compiling");
        print("Running");

class Laptop:
    def code(self,ide):
        ide.execute()

ide = Anaconda()

lap1= Laptop()
lap1.code(ide)
```

```
Compiling
Running
```

Overloading + operator

- The special method corresponding to the + operator is the __add__() method.
- Adding a custom definition of __add__() changes the behavior of the operator.
- Example: implement the ability to append new items to shooping cart in the Order class using the operator.

```
class Order:
  def __init__(self, cart, customer):
    self.cart = list(cart)
    self.customer = customer

  def __add__(self, other):
    new_cart = self.cart.copy()
    new_cart.append(other)
    return Order(new_cart, self.customer)
```

```
order = Order(['C', 'Java'], 'Conduira Online')

print((order + 'Python').cart)  # New Order instance

print(order.cart)  # Original instance unchanged

order = order + 'Python'  # Changing the original instance
print(order.cart)
```

```
['C', 'Java', 'Python']
['C', 'Java']
['C', 'Java', 'Python']
```

Overloading bool()

- The bool() built-in can be used to obtain the truth value of an object.
- To define its behavior, you can use the __bool__() special method.
- The behavior defined will determine the truth value of an instance in all contexts that require obtaining a truth value such as in if statements.
- For the Order class, an instance can be truthy if the length of the cart list is non-zero. This can be used to check whether an order should be processed or not.

Overloading bool()

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer

    def __bool__(self):
        return len(self.cart) > 0

order1 = Order(['Java', 'CPP', 'Python'], 'Conduira Online')
order2 = Order([], 'Conduira Offline')
```

```
print(bool(order1))

print(bool(order2))

for order in [order1, order2]:
    if order:
        print(f"{order.customer}'s order is processing...")
    else:
        print(f"Empty order for customer {order.customer}")
```

```
True
False
Conduira Online's order is processing...
Empty order for customer Conduira Offline
```

Case 1: Write a class called Course that has the following

- A field name that is the name of the course, a field capacity that is the maximum number of students allowed in the course, and a list called student_IDs representing the students in the course by their ID numbers (stored as strings).
- A constructor that takes the name of the course and capacity and sets those fields accordingly. The constructor should also initialize student_IDs list to an empty list, but it should not take a list as a parameter. It should only have the course name and capacity as parameters.
- A method called is_full() that takes no arguments and returns True or False based on whether or not the course is full (i.e. if the number of students in the course is equal to or above the capacity).
- A method called add_student() that takes a student ID number and adds the student to the course by putting their ID number into the list. If the student is already in the course, they must not be added to the list, and if the course is full, the student must not be added to the course.
- Test the class by creating a Course object, adding several students to the class, and calling the is_full()
- method. Print out the value of the student_IDs field to make sure everything comes out as expected.

Case 2: RestaurantCheck

Write a class called `RestaurantCheck`. It should have the following:

- Fields called `check_number`, `sales_tax_percent`, `subtotal`, `table_number`, and `server_name` representing an identification for the check, the bill without tax added, the sales tax percentage, the table number, and the name of the server.
- A constructor that sets the values of all four fields
- A method called `calculate_total` that takes no arguments (besides `self`) and returns the total bill including sales tax.
- A method called `print_check` that writes to a file called `check###.txt`, where `###` is the check number and writes information about the check to that file, formatted like below:

```
Check Number: 443
Sales tax: 6.0%
Subtotal: $23.14
Total: $24.53
Table Number: 17
Server: Sonic the Hedgehog
```

Test the class by creating a `RestaurantCheck` object and calling the `print_check()` method.