Unit-9 Advanced OOP Concepts and Introduction to NumPy

Polymorphism

What is 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.

In [1]:
```python
#Example 1: Polymorphism in addition operator
#We know that the + operator is used extensively in Python programs.
#But, it does not have a single usage.
#For integer data types, + operator is used to perform arithmetic addition operation.
num1 = 1
num2 = 2
print(num1+num2)
```

```
3
```

In [2]:
```python
#Example 1: Polymorphism in addition operator
#We know that the + operator is used extensively in Python programs. B
#But, it does not have a single usage.
#Similarly, for string data types, + operator is used to perform concatenation.
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

```
Python Programming
```

Function Polymorphism in Python

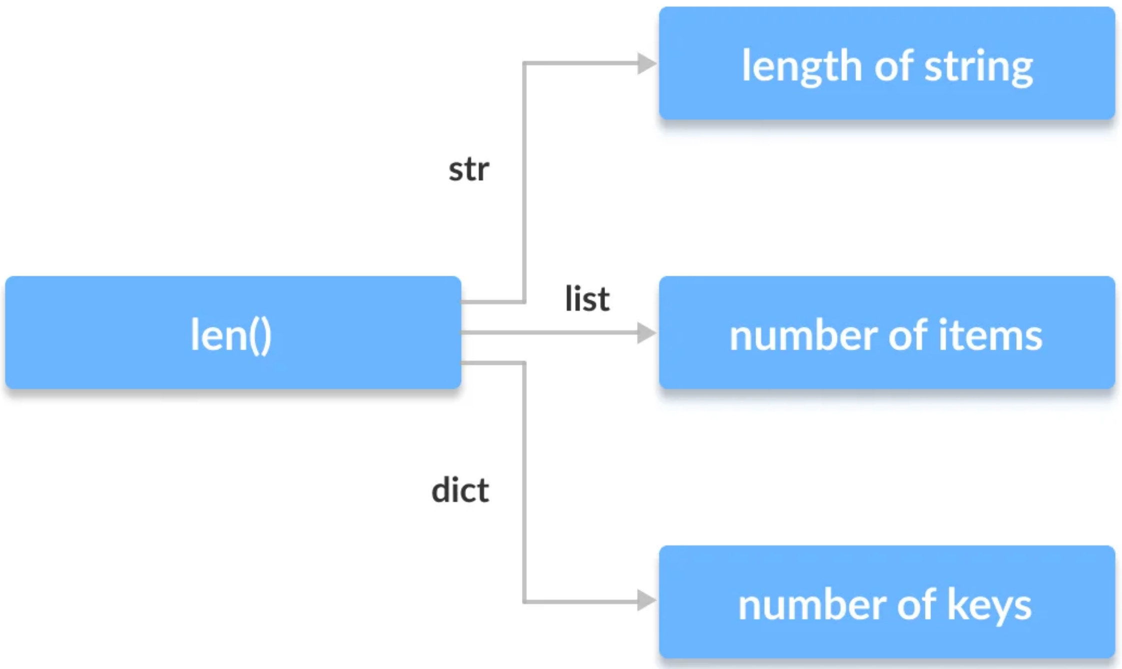
There are some functions in Python which are compatible to run with multiple data types.

One such function is the len() function. It can run with many data types in Python. Let's look at some example use cases of the function.

In [3]:
```python
#Example 2: Polymorphic len() function
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

```
9
3
2
```

Here, we can see that many 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

Polymorphism is a very important concept in Object-Oriented Programming. We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

In [4]:
```python
#Example 3: Polymorphism in Class Methods
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()
```

```
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
```

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.

In [8]:

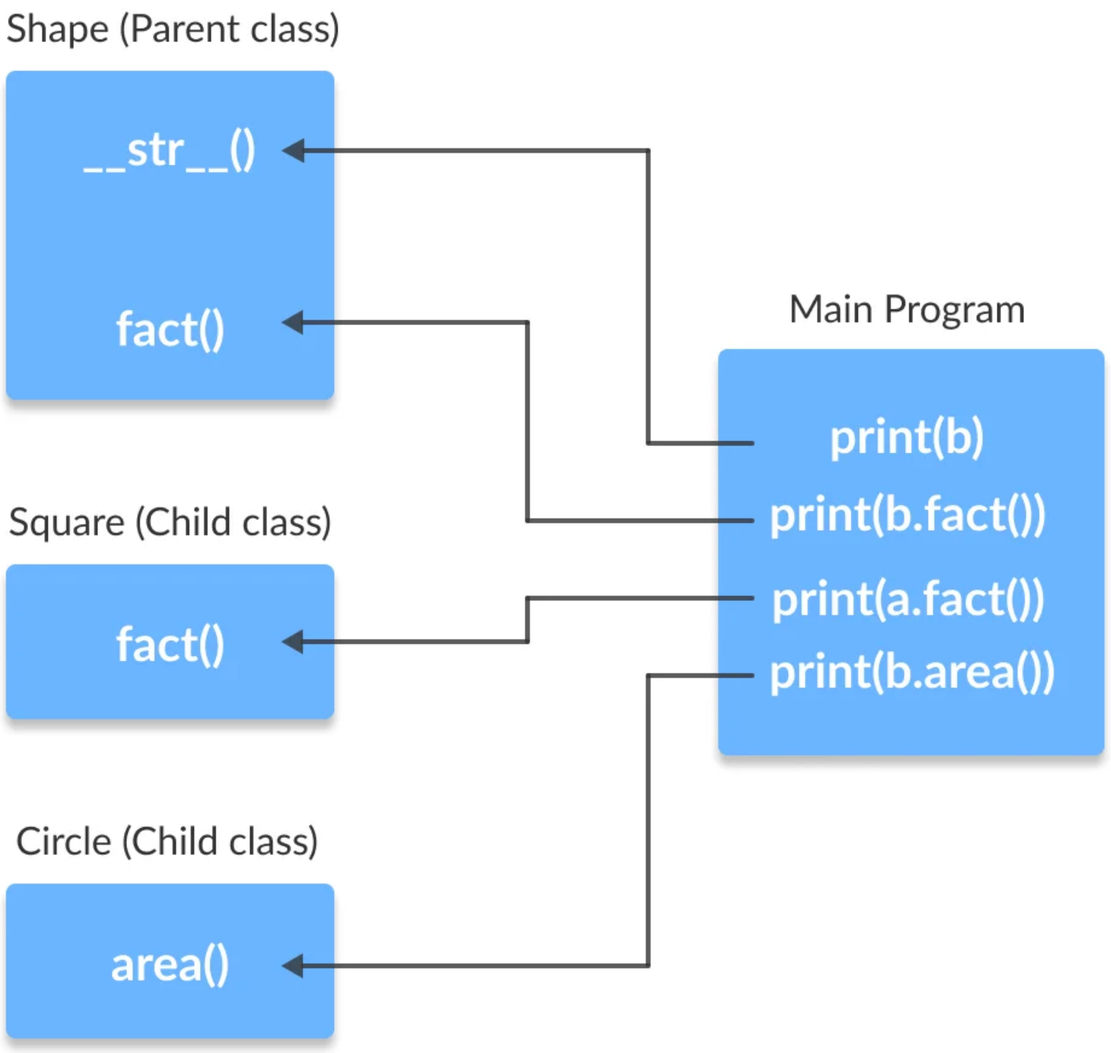
```
#Example 4: Method Overriding
from math import pi


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())
```

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

Here, we can see that the methods such as **str**(), which have not been overridden in the child classes, are used from the parent class.

Due to polymorphism, the Python interpreter automatically recognizes that the fact() method for object a(Square class) is overridden. So, it uses the one defined in the child class.

On the other hand, since the fact() method for object b isn't overridden, it is used from the Parent Shape class.



In [11]:
```python
#Example 5: Another example of method overriding
class P:
    def hello(self,name):
        print('Hello from parent class', name)

class C(P):
    def hello(self,name):
        print('Hello from child class', name)
```

```
c = C()
c.hello('abc')
```

```
Hello from child class abc
```

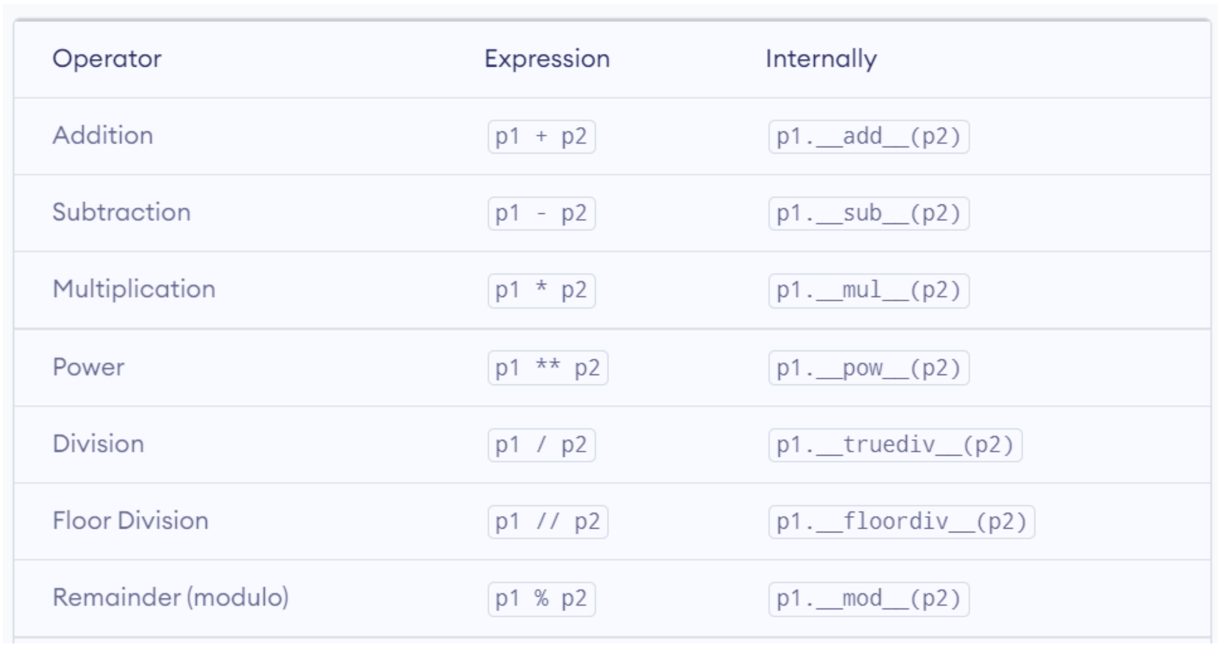
In [12]:
```
p = P()
p.hello('abc')
```

```
Hello from parent class abc
```

Method Overloading

Like other languages (for example, method overloading in C++) do, python does not support method overloading by default. But there are different ways to achieve method overloading in Python.

The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.



Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`

In [13]:
```
#Example 6: Example of method of overloading
# First product method.
# Takes two argument and print their
# product
def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
    p = a * b*c
    print(p)
# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

```
100
```

In [14]:
```python
# Example 6: Method of Overloading with an error
#First product method.
# Takes two argument and print their
# product
def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
    p = a * b*c
    print(p)
# Uncommenting the below line shows an error
product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [14], line 15
     13     print(p)
     14 # Uncommenting the below line shows an error
---> 15 product(4, 5)
     17 # This line will call the second product method
     18 product(4, 5, 5)

TypeError: product() missing 1 required positional argument: 'c'
```

In [15]:
```python
class Time:
    def __init__(self, h, m):
        self.h = h
        self.m = m
    def __add__(self, other):
        return Time(self.h + other.h, self.m + other.m)
    def display(self):
        print('Hours',self.h)
        print('Minutes',self.m)
```

In [17]:
```python
t1 = Time(2,30)
t2 = Time(3, 43)
```

In [18]:
```python
t3 = t1 + t2
```

In [19]:
```python
t1.display()
t2.display()
t3.display()
```

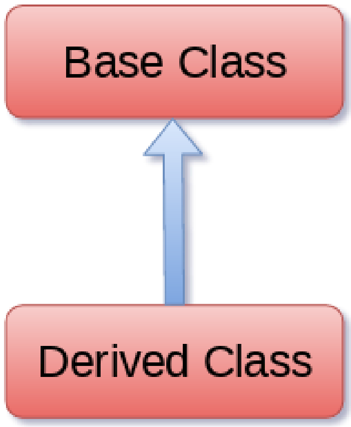
```
Hours 2
Minutes 30
Hours 3
Minutes 43
Hours 5
Minutes 73
```

Inheritance in Python:

Inheritance is the capability of one class to derive or inherit the properties from another class.

Benefits of inheritance are: -It represents real-world relationships well. -It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it. -It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Python Inheritance Syntax Class BaseClass: {Body} Class DerivedClass(BaseClass): {Body}



In [2]:
```
#Example of Inheritance
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(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
        print('The area of the triangle is %0.2f' %area)
```

In [3]:
```
t = Triangle()
```

In [4]:
```
t.inputSides()
```

```
Enter side 1 : 2
Enter side 2 : 4
Enter side 3 : 5
```

In [5]:
```
t.dispSides()
```

```
Side 1 is 2.0
Side 2 is 4.0
Side 3 is 5.0
```

In [6]:
```
t.findArea()
```

```
The area of the triangle is 3.80
```

Prepared by Abhi Shah

Different types of Inheritance:

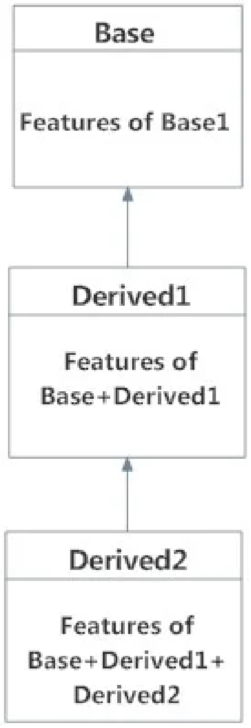
1. Single level inheritance: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.
2. Multi level inheritances: When we have a child and grandchild relationship. Unlike java, python shows multiple inheritances.

Multi level Inheritances-

Syntax of Multi level inheritance: class Base: pass

class Derived1(Base): pass

class Derived2(Derived1): pass


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

In [2]:
```
# A Python program to demonstrate multi level inheritance

# Base or Super class. Note object in bracket.
# (Generally, object is made ancestor of all classes)
# In Python 3.x "class Person" is
# equivalent to "class Person(object)"

class Base(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name


# Inherited or Sub class (Note Person in bracket)
```

```
class Child(Base):

    # Constructor
    def __init__(self, name, age):
        Base.__init__(self, name)
        self.age = age

    # To get name
    def getAge(self):
        return self.age
# Inherited or Sub class (Note Person in bracket)

class GrandChild(Child):

    # Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address

    # To get address
    def getAddress(self):
        return self.address


# Driver code
g = GrandChild("Ashley", 23, "Noida")
print(g.getName(), g.getAge(), g.getAddress())
```

```
Ashley 23 Noida
```

In [3]:

```
#Program for multiple Inheritance 2
class Father:
    def height(self):
        print("Height is 6.o foot")
class Mother:
    def color(self):
        print('Color is brown')
class Child(Father,Mother):
    pass
```

In [5]:

```
c=Child()
print('Child is inherited qualities')
```

```
Child is inherited qualities
```

In [6]:

```
c.height()
c.color()
```

```
Height is 6.o foot
Color is brown
```

Method of Resolution Order:

MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents. In case of multiple parents, the order is the same as tuples of base classes.

MRO of a class can be viewed as the **mro** attribute or the mro() method. The former returns a tuple while the latter returns a list.

In [7]:

```
class P1: # parent class 1
    def foo(self):
        print('called P1-foo()')
```

```
class P2: # parent class 2
    def foo(self):
        print('called P2-foo()')
    def bar(self):
        print('called P2-bar()')

class C1(P1, P2): # child 1 der. from P1, P2
    pass

class C2(P1, P2): # child 2 der. from P1, P2
    def bar(self):
        print('called C2-bar()')

class GC(C1, C2): # define grandchild class
      pass # derived from C1 and C2
```

In [8]:

```
gc = GC()
gc.foo()
gc.bar()
```

```
called P1-foo()
called C2-bar()
```

In [9]:

```
GC.__mro__
```

Out[9]:

```
(__main__.GC, __main__.C1, __main__.C2, __main__.P1, __main__.P2, object)
```

In [10]:

```
#Example of more complex method of resolution order
# Demonstration of MRO

class X:
    pass


class Y:
    pass


class Z:
    pass


class A(X, Y):
    pass


class B(Y, Z):
    pass


class M(B, A, Z):
    pass

# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
#  <class '__main__.A'>, <class '__main__.X'>,
#  <class '__main__.Y'>, <class '__main__.Z'>,
#  <class 'object'>]

print(M.mro())
```

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.X'>, <class
'__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```

Abstract Class

Here, we have an abstract class Vehicle. It is abstract because it is inheriting the abstract class abc. The class Vehicle have an abstract method called no_of_wheels, which do not have any definition, because abstract methods are not defined(or abstract methods remain empty, and they expects the classes inheriting the abstract classes to provide the implementation for the method). But, other classes which inherits the Vehicle class, like Bike, Tempo or Truck, defines the method no_of_wheels, and they provide their own implementation for the abstract method. Suppose, bike have 2 wheels, so it prints "Bike have 2 wheels" in the inherited abstract method no_of_wheels. And, similarly, Tempo and Truck classes also provide their own implementations. Some notable points on Abstract classes are:

Abstract classes cannot be instantiated. In simple words, we cannot create objects for the abstract classes. An Abstract class can contain the both types of methods -- normal and abstract method. In the abstract methods, we do not provide any definition or code. But in the normal methods, we provide the implementation of the code needed for the method.

In [11]:
```
#Example of Abstract Class
from abc import ABC

class Vehicle(ABC):  # inherits abstract class
    #abstract method
    def no_of_wheels(self):
        pass

class Bike(Vehicle):
    def no_of_wheels(self): # provide definition for abstract method
        print("Bikes have 2 wheels")

class Rickshaw(Vehicle):
    def no_of_wheels(self):  # provide definition for abstract method
        print("Rickshaws have 3 wheels")

class Truck(Vehicle):  # provide definition for abstract method
    def no_of_wheels(self):
        print("Trucks have 4 wheels")
```

In [12]:
```
bike = Bike()
bike.no_of_wheels()
rickshaw = Rickshaw()
rickshaw.no_of_wheels()
truck = Truck()
truck.no_of_wheels()
```

```
Bikes have 2 wheels
Rickshaws have 3 wheels
Trucks have 4 wheels
```

Why Use Abstract Base Classes? As we have discussed above, abstract classes are used to create a blueprint of our classes as they don't contain the method implementation. This is a very useful capability, especially in situations where child classes should provide their own separate implementation. Also, in complex projects involving large teams and a huge codebase, It is fairly difficult to remember all the class names.

Importance of Abstract Classes As we’ve discussed in the above section, we define a blueprint for our classes using an abstract class. The importance of using abstract classes in Python is that if our subclasses don’t follow that blueprint, Python will give an error. Thus we can make sure that our classes follow the structure and implement all the abstract methods defined in our abstract class.

Let’s take an example to understand this. Suppose in our example of the Circle class, we do not define the draw() method; instead, we define a draw_circle() method, which is doing the same thing as the draw().

In [6]:
```
from abc import ABC, abstractmethod

class Shape(ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    @abstractmethod
    def draw(self):
        pass

class Circle(Shape):
    def __init__(self):
        super().__init__("circle")

    def draw(self):
        print("Drawing a Circle")
class Triangle(Shape):

    def __init__(self):
        super().__init__("triangle")

    def draw(self):
        print("Drawing a Triangle")
```

In [7]:
```
#create a circle object
circle = Circle()
circle.draw()

#create a triangle object
triangle = Triangle()
triangle.draw()
```

```
Drawing a Circle
Drawing a Triangle
```

Abstract Properties

Abstract class in Python also provides the functionality of abstract properties in addition to abstract methods. Properties are Pythonic ways of using getters and setters. The abc module has a @abstractproperty decorator to use abstract properties. Just like abstract methods, we need to define abstract properties in implementation classes. Otherwise, Python will raise the error.

As we have been told, properties are used in Python for getters and setters. Abstract property is provided by the abc module to force the child class to provide getters and setters for a variable in Python.

In [8]:
```
from abc import ABC, abstractmethod, abstractproperty

class Shape(ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    @abstractproperty
    def name(self):
        pass

    @abstractmethod
    def draw(self):
        pass
```

```
In [9]:  class Circle(Shape):
             def __init__(self):
                 super().__init__("circle")

             @property
             def name(self):
                 return self.shape_name
             def draw(self):
                 print("Drawing a Circle")
```

```
In [10]: circle = Circle()
         print(f"The shape name is: {circle.name}")
```

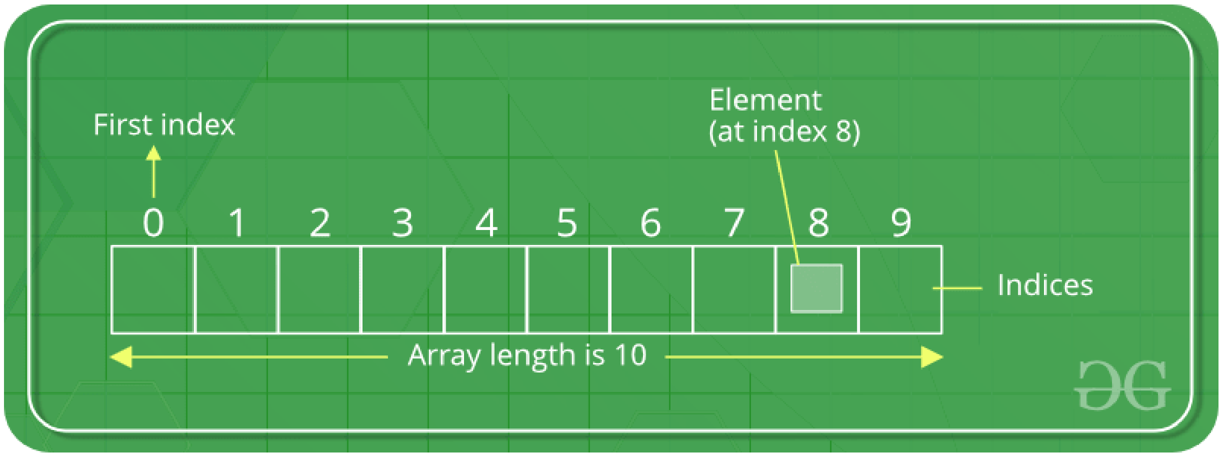
```
The shape name is: circle
```

Concrete Methods in Abstract Base Classes

The abstract classes may also contain concrete methods that have the implementation of the method and can be used by all the concrete classes. To define a concrete method in an abstract class, we simply define a method with implementation and don’t decorate it with the @abstractmethod decorator. If needed, we may also override this concrete method in the concrete class to provide any additional functionality as per user needs.

Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). For simplicity, we can think of an array a fleet of stairs where on each step is placed a value (let’s say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on. Array can be handled in Python by a module named array. They can be useful when we have to manipulate only a specific data type values. A user can treat lists as arrays. However, user cannot constraint the type of elements stored in a list. If you create arrays using the array module, all elements of the array must be of the same type.



Creating a Array

Array in Python can be created by importing array module. array(data_type, value_list) is used to create an array with data type and value list specified in its arguments.

Numpy

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

In [13]:
```
#Example of numpy
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```
```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

In [14]:
```
#Example Use a tuple to create a NumPy array:

import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```
```
[1 2 3 4 5]
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

In [15]:
```
#Example Create a 1-D array containing the values 1,2,3,4,5:

import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```
```
[1 2 3 4 5]
```

Example 2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

In [16]:
```
#Example Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
```

```
[[1 2 3]
 [4 5 6]]
```

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

In [17]:
```
#Example- Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2

import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
```

```
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
```

Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

In [18]:
```
#Example of Checking how many dimensions the arrays have:

import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

In [20]:
```
#Example Get the first element from the following array:

import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[1])
```

```
2
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the row represents the dimension and the index represents the column.

In [21]:
```
#Example Access the element on the first row, second column:

import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
```

```
2nd element on 1st row:  2
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

In [22]:
```
#Example Access the third element of the second array of the first array:

import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])
```

```
6
```

NumPy Array Slicing

Slicing arrays Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [start:end].

We can also define the step, like this: [start:end:step].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

In [23]:
```
#Example Slice elements from index 1 to index 5 from the following array:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```
```
[2 3 4 5]
```

In [24]:
```
#Example Slice elements from index 4 to the end of the array:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```
```
[5 6 7]
```

In [25]:
```
#Negative Slicing Use the minus operator to refer to an index from the end:
#Example Slice from the index 3 from the end to index 1 from the end:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```
```
[5 6]
```

In [26]:
```
#STEP
#Use the step value to determine the step of the slicing:

#Example
#Return every other element from index 1 to index 5:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```
```
[2 4]
```

Slicing 2D Arrays

In [27]:
```
#Example From the second element, slice elements from index 1 to index 4 (not included):

import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
```

```
print(arr[1, 1:4])
```

```
[7 8 9]
```

In [28]:

```
#Example From both elements, slice index 1 to index 4 (not included), this will return a 2-D arr

import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

In [29]:

```
#slicing rows
x = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9],],
[[11,12,13], [14,15,16], [17,18,19],],
[[21,22,23], [24,25,26], [27,28,29]]])
x[1]
```

Out[29]:

```
array([[11, 12, 13],
       [14, 15, 16],
       [17, 18, 19]])
```

Shape of an Array

The shape of an array is the number of elements in each dimension.

Get the Shape of an Array

NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

In [30]:

```
#Example
#Print the shape of a 2-D array:

import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```

```
(2, 4)
```

The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

What does the shape tuple represent? Integers at every index tells about the number of elements the corresponding dimension has.

In the example above at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

NumPy Array Reshaping

Reshaping arrays Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

In [31]:
```
#Example-Convert the following 1-D array with 12 elements into a 2-D array.

#The outermost dimension will have 4 arrays, each with 3 elements:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Reshape From 1-D to 3-D

In [32]:
```
#Example
#Convert the following 1-D array with 12 elements into a 3-D array.

#The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

```
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Can We Reshape Into any Shape? Yes, as long as the elements required for reshaping are equal in both shapes.

We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

In [33]:
```
#Example
#Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will r

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(3, 3)

print(newarr)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [33], line 8
      4 import numpy as np
      6 arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
----> 8 newarr = arr.reshape(3, 3)
     10 print(newarr)

ValueError: cannot reshape array of size 8 into shape (3,3)
```

Iterating Arrays

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

If we iterate on a 1-D array it will go through each element one by one.

In [34]:
```
#Example
#Iterate on the elements of the following 1-D array:

import numpy as np

arr = np.array([1, 2, 3])

for x in arr:
    print(x)
```

```
1
2
3
```

Iterating 2-D Arrays

In a 2-D array it will go through all the rows.

In [35]:
```
#Example
#Iterate on the elements of the following 2-D array:

import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    print(x)
```

```
[1 2 3]
[4 5 6]
```

Iterating 3-D Arrays In a 3-D array it will go through all the 2-D arrays.

In [36]:
```
#Example
#Iterate on the elements of the following 3-D array:

import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
    print(x)
```

```
[[1 2 3]
 [4 5 6]]
[[ 7  8  9]
 [10 11 12]]
```

Built in functions

Joining NumPy Arrays Joining means putting contents of two or more arrays in a single array.

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

In [37]:
```
#Example of Join two arrays

import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

```
[1 2 3 4 5 6]
```

In [38]:
```
#Example Join two 2-D arrays along rows (axis=1):

import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

```
[[1 2 5 6]
 [3 4 7 8]]
```

Splitting NumPy Arrays

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.

In [39]:
```
#Example Split the array in 3 parts:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 3)

print(newarr)
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

If the array has less elements than required, it will adjust from the end accordingly.

In [40]:
```python
#Example Split the array in 4 parts:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 4)

print(newarr)
```

```
[array([1, 2]), array([3, 4]), array([5]), array([6])]
```

Split Into Arrays The return value of the array_split() method is an array containing each of the split as an array.

If you split an array into 3 arrays, you can access them from the result just like any array element:

In [41]:
```python
#Example Access the splitted arrays:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 3)

print(newarr[0])
print(newarr[1])
print(newarr[2])
```

```
[1 2]
[3 4]
[5 6]
```

Splitting 2-D Arrays Use the same syntax when splitting 2-D arrays.

Use the array_split() method, pass in the array you want to split and the number of splits you want to do.

In [42]:
```python
#Example Split the 2-D array into three 2-D arrays.

import numpy as np

arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])

newarr = np.array_split(arr, 3)

print(newarr)
```

```
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

In [43]:
```python
#Example Split the 2-D array into three 2-D arrays.

import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3)

print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
```

In [44]:
```python
#Example Split the 2-D array into three 2-D arrays along rows.

import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3, axis=1)

print(newarr)
```

```
[array([[ 1],
       [ 4],
       [ 7],
       [10],
       [13],
       [16]]), array([[ 2],
       [ 5],
       [ 8],
       [11],
       [14],
       [17]]), array([[ 3],
       [ 6],
       [ 9],
       [12],
       [15],
       [18]])]
```

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the where() method.

In [45]:
```python
#Example Find the indexes where the value is 4:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 4, 4])

x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6], dtype=int64),)
```

In [46]:
```python
#Example Find the indexes where the values are even:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

x = np.where(arr%2 == 0)

print(x)
```

```
(array([1, 3, 5, 7], dtype=int64),)
```

Sorting Arrays

Sorting means putting elements in an ordered sequence.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

In [47]:
```
#Example Sort the array:

import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

```
[0 1 2 3]
```

You can also sort arrays of strings, or any other data type:

In [48]:
```
#Example Sort the array alphabetically:

import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

Sorting a 2-D Array If you use the sort() method on a 2-D array, both arrays will be sorted:

In [49]:
```
#Example Sort a 2-D array:

import numpy as np

arr = np.array([[3, 2, 4], [5, 0, 1]])

print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```