Polymorphism

- Sometimes an object comes in many types or forms. If we have a button, there are many different draw outputs (round button, check button, square button, button with image) but they do share the same logic: onClick(). We access them using the same method . This idea is called Polymorphism.

- Polymorphism is based on the greek words Poly (many) and morphism (forms). We will create a structure that can take or use many forms of objects. ¶

Example 1 :

In [2]:

```
class Bear(object):
    def sound(self):
        print ("Groarrr")

class Dog(object):
    def sound(self):
        print ("Woof woof!")

def makeSound(animalType):
    animalType.sound()


bearObj = Bear()
dogObj = Dog()

makeSound(bearObj)
makeSound(dogObj)
```

```
Groarrr
Woof woof!
```

Example 2 :

In [3]:

```
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.")
```

In [4]:

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

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

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

In [5]:

```
# We have two objects, sammy of the Shark class, and casey of the Clownfish class.
# Our for loop iterates through these objects, calling the swim(), swim_backwards()
sammy = Shark()

casey = Clownfish()

for fish in (sammy, casey):
    fish.swim()
    fish.swim_backwards()
    fish.skeleton()
```

```
The shark is swimming.
The shark cannot swim backwards, but can sink backwards.
The shark's skeleton is made of cartilage.
The clownfish is swimming.
The clownfish can swim backwards.
The clownfish's skeleton is made of bone.
```

Python Operator Overloading

- Python operators work for built-in classes. But same operator behaves differently with different types.
  For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and

concatenate two strings.

- It is possible because + operator is overloaded by both int class and str class. The operators are actually methods defined in respective classes.
- This feature in Python, that allows same operator to have different meaning according to the context is called **operator overloading**.

For e.g. To use + operator with custom objects you need to define a method called __add__ .

In [37]:

```
class OverloadingTest:
    def __init__(self,x = 0):
        self.x = x
```

In [38]:

```
obj1 = OverloadingTest()
obj2 = OverloadingTest()
```

In [39]:

```
obj1 + obj2
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-39-77e173e2d662> in <module>()
----> 1 obj1 + obj2

TypeError: unsupported operand type(s) for +: 'OverloadingTest' and 'O
verloadingTest'
```

In the above example we add __**add**__ method which allows use to use + operator to add two objects. Inside the__**add**__ method we are creating a new object and returning it to the caller.

In [46]:

```
class OverloadingTest:
    def __init__(self,x = 0):
        self.x = x
    def __add__(self,other_object):
        return(self.x + other_object.x)

obj1 = OverloadingTest(8)
obj2 = OverloadingTest(9)

print(obj1 + obj2)
# This became possible because we have overloaded + operator by adding a method nam
```

```
17
```

Python has many other special methods like __add__ , see the list below:

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
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

In [ ]: