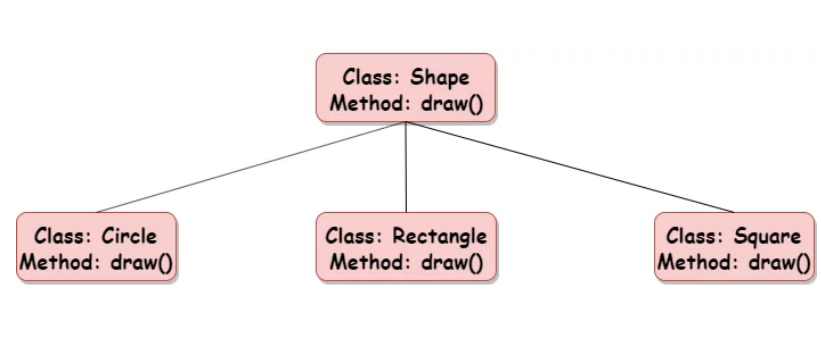
Polymorphism In Python

What Is Meant By Polymorphism ?

- let's break down the word polymorphism to poly and morphism
- In plain English, poly means many and morph means form. So polymorphism means many forms.

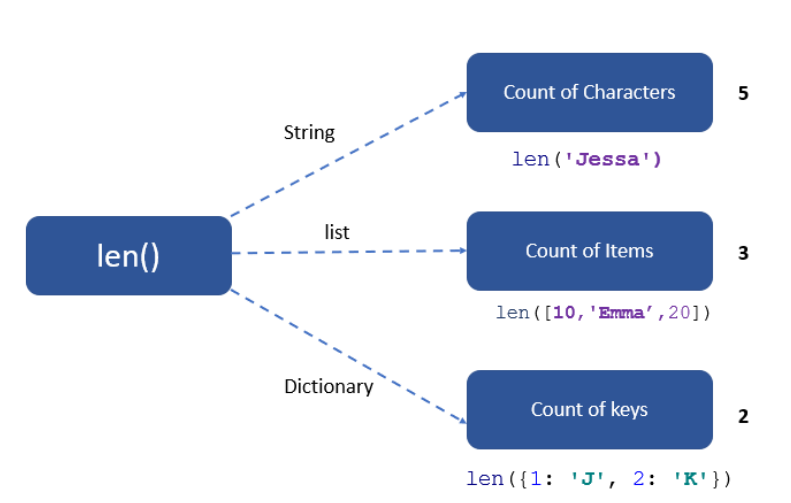
- Polymorphism refers to a function having the same name but being used in different ways and different scenarios.
- It allows us to define methods in the child class with the same name as their parent class



Benefits of Polymorphism :

- It helps in code reusability.
- Saves a lot of time for the programmers.
- A single variable can be used to store multiple data types.
- Easy to debug the codes.

Polymorphism in Built-in function len( ):



Polymorphic len() function

In [1]:
```python
#Polymorphism in Built-in function len( )
students = ['Emma', 'Jessa', 'Kelly']
school = 'ABC School'
details = {"name":"Emma","class":10,"school":"ABC School"}
# calculate count
print(len(students))
print(len(school))
print(len(details))
```

```
3
10
3
```

Polymorphism with user define function :

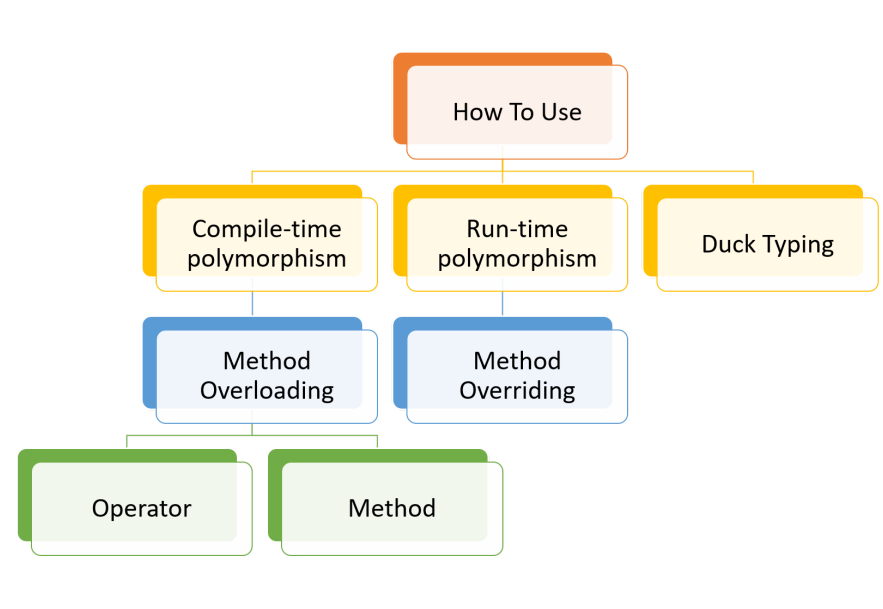
In [5]:
```python
#add function - we can create user-defined methods with the same name and different attributes list

def add(a,b,c=0):
    return a+b+c

print(add(2,3))
print(add(2,3,5))
```

```
5
10
```

Types of Polymorphism :



Duck Typing In Python :

- Duck typing is one of the concepts in polymorphism.
- It is derived from the following quote : If it looks like a duck and quacks like a duck, it's a duck
- It means something that is similar in its behavior to anything then it will consider a thing of that category

to which it is similar to

- The key takeaway is that when using duck typing it does not matter which class object we are passing,

rather what matters is the object should have the related method in it.

- We don't check types at all in this functionality, we check for the methods and their definition instead

In [3]:
```python
#Polymorphism With Class Method
#Different classes can hold different class method of same name
class Mercury:
  def weight(self, w):
    print(w * 0.38)

class Mars:
  def weight(self, w):
    print(w * 0.38)

class Earth:
  def weight(self, w):
    print(w)
#we can see we have called the same method name in every class
#but their functionality is different, and that is what we focus on.
for obj in Mercury(), Earth(), Mars():
  obj.weight(50)
```

```
19.0
50
19.0
```

In [9]:
```python
##Polymorphism With Class Method
class Rabbit():
    def age(self):
        print("This function determines the age of Rabbit.")
    def color(self):
        print("This function determines the color of Rabbit.")

class Horse():
    def age(self):
        print("This function determines the age of Horse.")
    def color(self):
        print("This function determines the color of Horse.")

obj1=Rabbit()
obj2=Horse()

#For serving the purpose of Polymorphism,
#a loop can be created that iterates through the tuple of objects
for method in (obj1,obj2):
    # One can then call a method without looking at the type of class to which the objects belong
    print("Calling Method :",method)
    method.age()
    method.color()
    print()
```

```
Calling Method : <__main__.Rabbit object at 0x000002A43BDD9520>
This function determines the age of Rabbit.
This function determines the color of Rabbit.

Calling Method : <__main__.Horse object at 0x000002A43BDD9C10>
This function determines the age of Horse.
This function determines the color of Horse.
```

Run-Time Polymorphism :

- Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism
- The runtime polymorphism can be achieved by method overriding
- It is also called dynamic or late binding

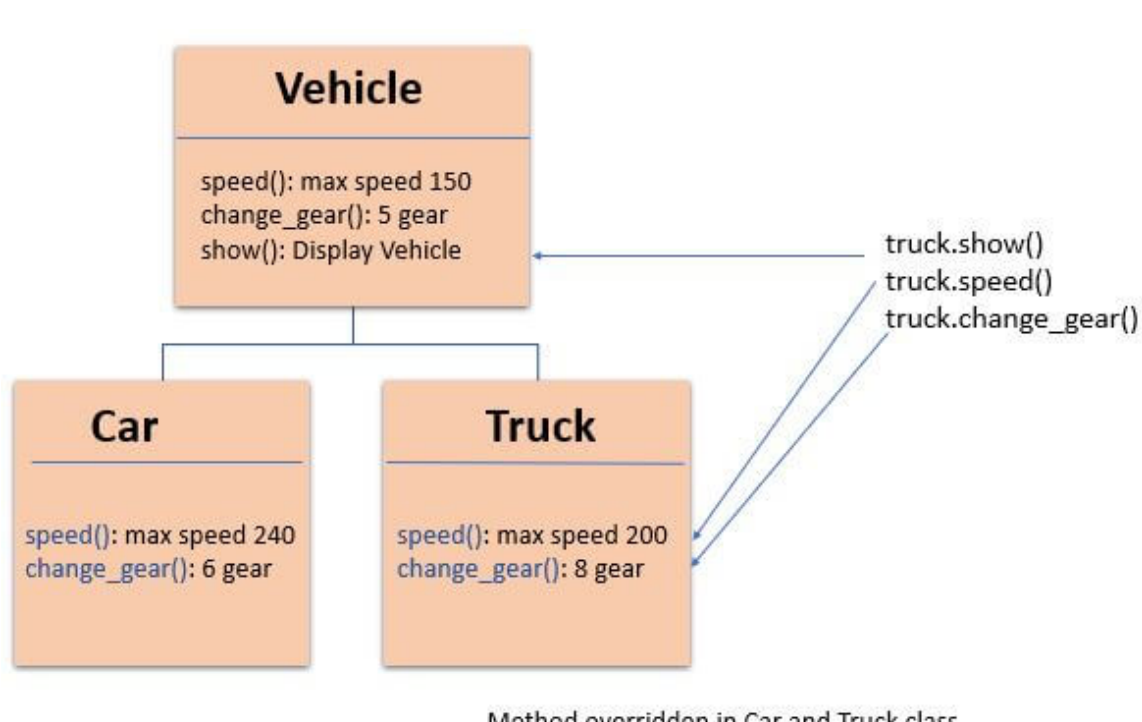
Polymorphism With Inheritance - Method Over Riding :

- Method Overriding,supports runtime polymorphism
- we can reimplement a method and change its functionality in child class according to our needs
- The process of re-implementing the inherited method in the child class is known as Method Overriding.
- We can inherit methods in the child class from the parent class that have the same name,

maybe the same parameters but differ in functionality.

- It is very beneficial in case our parent class method's functionality is not suitable in child class
- In method overriding we have two methods, one in parent class which is referred to as overridden method
- And one is the inherited method in child class which is referred to as an overriding method,

Both methods have the same name and same signature.



Method overridden in Car and Truck class

In [5]:
```python
#Parent Class
class Vehicle:

    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price

    def show(self):
        print('Details:', self.name, self.color, self.price)
    #overridden method
    def max_speed(self):
        print('Vehicle max speed is 150')
    #overridden method
    def change_gear(self):
        print('Vehicle change 6 gear')


# inherit from vehicle class
class Car(Vehicle):
    #overriding method
    def max_speed(self):
        print('Car max speed is 240')
    #overriding method
    def change_gear(self):
        print('Car change 7 gear')


# Car Object
car = Car('Car x1', 'Red', 20000)
# calls method from a Vehicle class
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
print()
# calls method from a Vehicle class
vehicle.show()
vehicle.max_speed()
vehicle.change_gear()
```

```
Details: Car x1 Red 20000
Car max speed is 240
Car change 7 gear

Details: Truck x1 white 75000
Vehicle max speed is 150
Vehicle change 6 gear
```

TO DO : Create a class truck that inherits from vechicle and over ride the methods max_speed and change_gear

Compile Time Polymorphism :

- Whenever an object is bound with its functionality at the compile time , is known as the

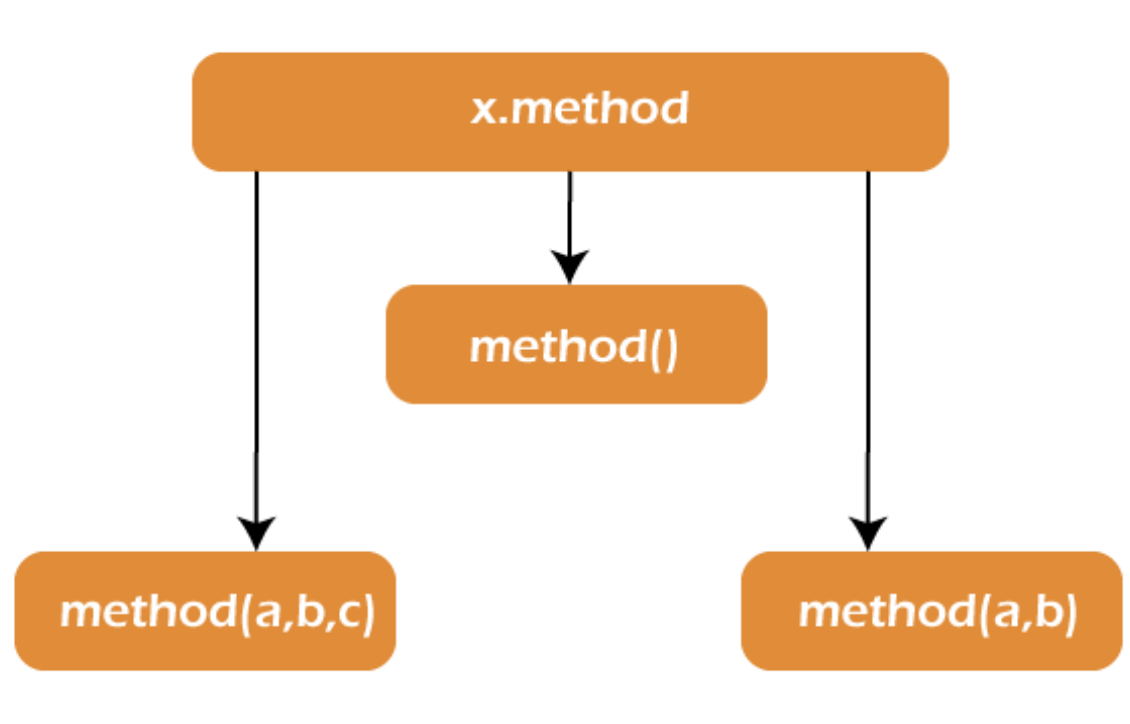
compile-time polymorphism.

- Compile-time polymorphism is achieved through method overloading
- It is also called static or early binding.

Method Overloading :

- Method overloading supports compile-time polymorphism.
- The simple meaning of method overloading is a class containing multiple methods with the same name

but having different parameters.



- Python does not support method overloading.
- Python considers only the latest defined method even if you overload the method.
- Python will raise a TypeError if you overload the method.

In [9]:
```python
#Method OverLoading
class A:
    def add(self,a,b):
        s = a+b
        print(s)
    def add(self,a,b,c):
        s = a+b+c
        print(s)

ob = A()
# the below line shows an error
#ob.add(10,20)

# This line will call the second add method
ob.add(3,7,5)
```

```
15
```

Method Over Loading In Python

- To overcome the above problem, we can use different ways to achieve the method overloading
- In Python, to overload the class method, we need to write the method's logic so that different code

executes inside the function depending on the parameter passes.

- For example, the built-in function range() takes three parameters and produce different result

depending upon the number of parameters passed to it.

In [10]:
```python
#built-in function range() - start - stop - step
for i in range(5): print(i, end=', ')
print()
for i in range(5, 10): print(i, end=', ')
print()
for i in range(2, 12, 2): print(i, end=', ')
```

```
0, 1, 2, 3, 4,
5, 6, 7, 8, 9,
2, 4, 6, 8, 10,
```

In [20]:
```python
#Method Loading in Python
class Book_Food:
  def book(self, lunch=None, dinner=None):
    if lunch and dinner:
      print("Lunch & Dinner Booked")
    elif lunch:
      print("Lunch Booked")
    elif dinner:
      print("Dinner Booked")
    else:
      print("No Food Item Booked")

obj = Book_Food()
print("With No Parameters")
obj.book()
print("****************")
print("With Single Parameter")
obj.book(lunch=1)
print("****************")
print("With Two Parameters")
obj.book(lunch=1,dinner=1)
#Note : This is not the ideal way of achieving method overloading
```

```
With No Parameters
No Food Item Booked
****************
With Single Parameter
Lunch Booked
****************
With Two Parameters
Lunch & Dinner Booked
```

Operator Overloading In python

- Another way of implementing polymorphism in python is through the use of operator overloading
- Operator Overloading is the process of using the same operator,in multiple forms depending

on the operands used.

- Lets say plus operator performs addition on two integers and at the same time it performs

concatenation on two strings and extension on lists

In [21]:
```python
#Polymorphism with + operator

#For integer types "+" operator is used to add the two integers
number = 10 + 20
print("+ with Integers :",number)
print()
#For strings types "+" operator is used to concate the two strings
str1 = "python "
str2 = " programming"
str3 = str1 + str2
print("+ with strings :",str3)

print()
#For lists types "+" operator is used to extend the elements in lists
list1 = [2,3,4,5,6]
list2 = [7,8,9,10,12]
list3 = list1 + list2
print("+ with lits :",list3)
```

```
+ with Integers : 30

+ with strings : python  programming

+ with lits : [2, 3, 4, 5, 6, 7, 8, 9, 10, 12]
```

How This Is Possible ?

- Always remember whatever happens in python is solely based on objects
- Obviously, the class must have some methods to compute its operands.
- So when you use a+b, python internally calls the magic method - int.__add__(a,b)method

In [11]:
```python
#Polymorphism with operators
a = 1
b = 2
# a and b are the two attributes of class int
print("+ with Integers :",int.__add__(a,b))
print()
# d and e are the two attributes of class str
d = 'Programming'
e = ' Language'
print("+ with strings :",str.__add__(d,e))
print()
#magic method mul
c ='Python '
print("Magic Method Mul :",str.__mul__(c,3))
```

```
+ with Integers : 3

+ with strings : Programming Language

Magic Method Mul : Python Python Python
```

So the moment you we use a

- + operator it calls an obj.__add__( )method.
- - operator calls an obj.__sub__( ) method.
- * operator calls an obj.__mul__( ) method.

Operator Over Loading to a user-defined class

Overloading + operator for custom objects

- Suppose we have two objects, and we want to add these two objects with a binary + operator
- However, it will throw an error if we perform addition because the compiler doesn't add two objects

In [34]:
```python
class Book:
    def __init__(self, pages):
        self.pages = pages

# creating two objects
b1 = Book(400)
b2 = Book(300)

# add two objects
print(b1 + b2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-b341e452818c> in <module>
      8
      9 # add two objects
---> 10 print(b1 + b2)

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

How Can We Achive ?

- Internally + operator is implemented by using __add__() method
- We have to override this method in our class if you want to add two custom objects.

In [35]:
```python
class Book:
    def __init__(self, pages):
        self.pages = pages

    # Overloading + operator with magic method
    def __add__(self, other):
        return self.pages + other.pages

b1 = Book(400)
b2 = Book(300)
print("Total number of pages: ", b1 + b2)
```

```
Total number of pages:  700
```

Overloading the * Operator

- The * operator is used to perform the multiplication
- Internally * operator is implemented by using the __mul__() method.

In [16]:
```python
#Let's see how to overload it to calculate the salary of an employee for a specific period
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __mul__(self, timesheet):
        print('Worked for', timesheet.days, 'days')
        # calculate salary
        return self.salary * timesheet.days


class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days


emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)
```

```
Worked for 50 days
salary is:  40000
```

Python - Magic or Dunder Methods

- Magic methods in Python are the special methods that start and end with the double underscores.
- They are also called Dunder methods ,Dunder here means "Double Under (Underscores)
- Magic methods are not meant to be called directly, but internally, through some other methods or actions.
- Built-in classes in Python define many magic methods.
- Use the dir() function to see the number of magic methods inherited by a class.
- Magic methods are most frequently used to define overloaded behaviours of predefined operators in Python
- In order to make the overloaded behaviour available in your own custom class,

the corresponding magic method should be overridden

In [38]:
```python
#Use the dir() function to see the number of magic methods inherited by a class int.
print(dir(int))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

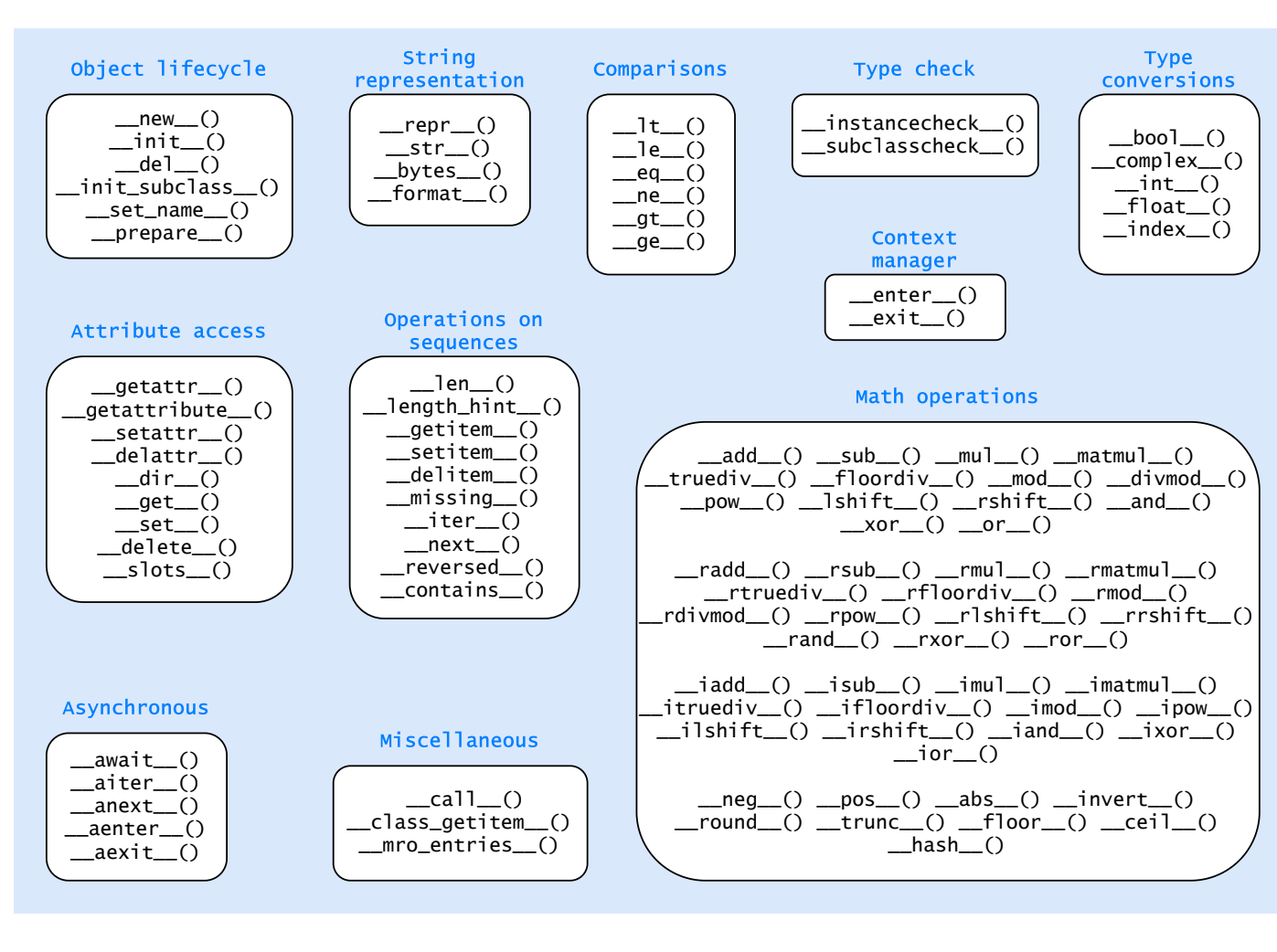
In [39]:
```python
# __ge__() method class to overload the >= operator.
class distance:
    def __init__(self, x=None,y=None):
        self.ft=x
        self.inch=y
    def __ge__(self, x):
        val1=self.ft*12+self.inch
        val2=x.ft*12+x.inch
        if val1>=val2:
            return True
        else:
            return False

d1=distance(2,1)
d2=distance(4,10)
#using >= to compare two objects
d1>=d2
```

Out[39]:
```
False
```

Important Magic Methods

- The following image list important magic methods in Python 3.



Polymorphism FAQ's :

- What is Polymorphism ?
- What are the types of Polymorphism ?
- What is method overloading ?
- What is operator overloading ?
- What is method overriding ?
- How is Inheritance useful to achieve Polymorphism ?
- What are the advantages of Polymorphism?
- What are the differences between Polymorphism and Inheritance ?
- Is it possible to implement method over loading in python ?
- What are magic methods in python ?