❖ __dir__():

Python's __dir__() magic method implements the functionality of the dir() built-in function. Semantically, dir() returns all (function, object, or variable) names in a given scope. However, the magic method __dir__() converts any return value to a sorted list.

[1]:
```
class Supermarket:
# Function __dir()___ which list all
# the base attributes to be used.
    def __dir__(self):
            return['customer_name', 'product',    'quantity', 'price', 'date']

# user-defined object of class supermarket
my_cart = Supermarket()

# listing out the dir() method
print(dir(my_cart))
```

```
['customer_name', 'date', 'price', 'product', 'quantity']
```

❖ __str__():

The Python **__str__** method returns a string representation of the object on which it is called. For example, if you call **print(x)** an object **x**, Python internally calls **x.__str__()** to determine the string representation of object x. This method is also used to implement the built-in **str()** function.

[6]:
```
class Complex:
    # Constructor
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    # For call to str(). Prints readable form
    def __str__(self):
        return '%s + i%s' % (self.real, self.imag)
# Driver program to test above
t = Complex(10, 20)
# Same as "print t"
print (str(t))
```

```
10 + i20
```

❖ __getattribute__():

Python's magic method **__getattribute__()** implements the built-in **getattr()** function that returns the value associated with a given attribute name. If the **__getattribute__()** error results in an **AttributeError** due to a non-existent attribute, Python will call the **__getattr__()** function for resolution.

```
class Person:
    def __getattribute__(self, attr_name):
        print('hello world')
    def __getattr__(self, attr_name):
        print('hello universe')

alice = Person()
getattr(alice, "age")
# hello world
```

```
hello world
```

```
class Person:
    def __getattribute__(self, attr_name):
        raise AttributeError
    def __getattr__(self, attr_name):
        print('hello universe')

alice = Person()
getattr(alice, 'age')
```

```
hello universe
```

❖ __format__():

The Python **__format__()** method implements the built-in **format()** function as well as the **string.format()** method. So, when you call **format(x, spec)** or **string.format(spec)**, Python attempts to call **x.__format__(spec)**. The return value is a string.

```
class Data:
    def __format__(self, spec):
        return 'hello ' + spec
x = Data()
print(format(x, 'world'))
```

```
hello world
```

❖ __init_subclass__():

The Python **class.__init_subclass__(cls)** method is called on a given class each time a subclass cls for that class is created.

```
class Parent:
    def __init_subclass__(cls):
        print('Subclass of Parent Created!')
class Child(Parent):
    pass
class Grandchild(Child):
    pass
```

```
Subclass of Parent Created!
Subclass of Parent Created!
```

❖ __hash__():

The Python **__hash__()** method implements the built-in **hash()** function. So, when you cal hash(x), Python attempts to call **x.__hash__()**. If the return value is not an integer or the **x.__hash__()** method is not defined, Python will raise a **TypeError**.

```
class Data:
    def __hash__(self):
        return 42
x = Data()
res = hash(x)
print(res)
```

```
42
```

```
class Data:
    def __hash__(self):
        return 'finxter'
x = Data()
res = hash(x)
print(res)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [14], line 5
      3         return 'finxter'
      4 x = Data()
----> 5 res = hash(x)
      6 print(res)

TypeError: __hash__ method should return an integer
```

❖ **__eq__():**

To customize the behavior of the equality operator **x == y**, override the **__eq__()** dunder method in your class definition. Python internally calls **x.__eq__(y)** to compare two objects using **x == y**. If the **__eq__()** method is not defined, Python will use the **is** operator per default that checks for two arbitrary objects whether they reside on the same memory address.

```
class Person:
    def __init__(self, age):
        self.age = age
    def __eq__(self, other):
        return self.age == other.age
alice = Person(18)
bob = Person(19)
carl = Person(18)
print(alice == bob)
print(alice == carl)
```

```
False
True
```

❖ **__new__():**

Python's **__new__(cls)** magic method creates a new instance of class **cls**. The remaining arguments are passed to the object constructor. The return value is the newly-created object—an instance of **cls**.

```
class My_Class(object):
    def __new__(cls):
        print("Python is great!")
        return super(My_Class, cls).__new__(cls)
My_Class()
```

```
Python is great!
<__main__.My_Class at 0x36e09f8>
```

❖ **__repr__():**

The Python **__repr__** method returns a string representation of the object on which it is called. It implements the built-in **repr()** function. If you call **print(x)** an object **x**, Python internally calls **x.__str__()** to determine the string representation of object **x**. If this isn't implemented, Python calls **x.__repr__()**.

```
class Car:
    def __init__(self, color, brand):
        self.color = color
        self.brand = brand
    def __repr__(self):
        return '123'

porsche = Car('black', 'porsche')
tesla = Car('silver', 'tesla')
print(str(porsche))
print(str(tesla))
```

```
123
123
```

❖ **__setattr__():**

Python's magic method **__setattr__()** implements the built-in **setattr()** function that takes an object and an attribute name as arguments and removes the attribute from the object.

```
class Person:
    def __setattr__(self, attr_name, attr_value):
        print(attr_name, attr_value)

alice = Person()
setattr(alice, 'age', 32)
```

```
age 32
```

❖ **__sizeof__():**

Python's **__new__(cls)** magic method creates a new instance of class **cls**. The remaining arguments are passed to the object constructor. The return value is the newly-created object—an instance of **cls**.

```
import sys
class Data:
    def __sizeof__(self):
        return 42


x = Data()
print(x.__sizeof__())

print(sys.getsizeof(x))
```

```
42
50
```

❖ **__init__():**

All classes have a function called __init__(), which is always executed when the class is being initiated. Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

❖ **__delattr__():**

Python's magic method `__delattr__()` implements the built-in `delattr()` function that takes an object and an attribute name as arguments and removes the attribute from the object.

```
class Car:
    def __init__(self):
        self.speed = 100
    def __delattr__(self, attr):
        self.speed = 42
# Create object
porsche = Car()
print(porsche.speed)
delattr(porsche, 'speed')
print(porsche.speed)
```

```
100
42
```

❖ **__ne__():**

To customize the behavior of the non-equality operator **x != y**, override the **__ne__()** dunder method in your class definition. Python internally calls **x.__ne__(y)** to compare two objects using **x != y**. If the **__ne__()** method is not defined, Python will use the **is not** operator per default that checks for two arbitrary objects whether they reside on a different memory address.

```
class Person:
    def __init__(self, age):
        self.age = age
    def __ne__(self, other):
        return self.age != other.age
alice = Person(18)
bob = Person(19)
carl = Person(18)
print(alice != bob)
print(alice != carl)
```

```
True
False
```

❖ **__contains__():**

The Python `__contains__()` magic method implements the membership operation, i.e., the `in` keyword. Semantically, the method returns `True` if the argument object exists in the sequence on which it is called, and `False` otherwise. For example, `3 in [1, 2, 3]` returns `True` as defined by the list method `[1, 2, 3].__contains__(3)`.

```
class MyClass:
    def __init__(self, my_collection):
        self.my_collection = my_collection
    def __contains__(self, item):
        return item in self.my_collection
my = MyClass('hello world')
print('hello' in my)
```

```
True
```

❖ **__it__():**

To customize the behavior of the less than operator `x < y`, override the `__lt__()` dunder method in your class definition. Python internally calls `x.__lt__(y)` to obtain a return value when comparing two objects using `x < y`. The return value can be any data type because any value can automatically converted to a Boolean by using the `bool()` built-in function. If the `__lt__()` method is not defined, Python will raise a `TypeError`.

```
class Person:
    def __init__(self, age):
        self.age = age
    def __lt__(self, other):
        return self.age < other.age
alice = Person(18)
bob = Person(17)
carl = Person(18)
print(alice < bob)
print(alice < carl)
print(bob < alice)
```

```
False
False
True
```

❖ **__le__():**

To customize the behavior of the less than or equal to operator **x <= y**, override the **__le__()** dunder method in your class definition. Python internally calls **x.__le__(y)** to obtain a return value when comparing two objects using **x <= y**. The return value can be any data type because any value can automatically converted to a Boolean by using the **bool()** built-in function. If the **__le__()** method is not defined, Python will raise a **TypeError**.

```
class Person:
    def __init__(self, age):
        self.age = age
    def __le__(self, other):
        return self.age <= other.age
alice = Person(18)
bob = Person(17)
carl = Person(18)
print(alice <= bob)
print(alice <= carl)
print(bob <= alice)
```

```
False
True
True
```

❖ **__gt__():**

To customize the behavior of the greather than operator **x > y**, override the **__gt__()** dunder method in your class definition. Python internally calls **x.__gt__(y)** to obtain a return value when comparing two objects using **x > y**. The return value can be any data type because any value can automatically converted to a Boolean by using the **bool()** built-in function. If the **__gt__()** method is not defined, Python will raise a **TypeError**.

```
class Person:
    def __init__(self, age):
        self.age = age
    def __gt__(self, other):
        return self.age > other.age
alice = Person(18)
bob = Person(17)
carl = Person(18)
print(alice > bob)
print(alice > carl)
print(bob > alice)
```

```
True
False
False
```

❖ **__doc__():**

Python objects have an attribute called `__doc__` that provides a documentation of the object. For example, you simply call `Dog.__doc__` on your class `Dog` to retrieve its documentation as a string.

```
class Dog:
    """Your best friend."""
    def do_nothing(self):
        pass
print(Dog.__doc__)
```

```
Your best friend.
```

❖ **__index__():**

Python's `__index__(self)` method is called on an object to get its associated integer value. The returned integer is used in slicing or as the basis for the conversion in the built-in functions `bin()`, `hex()`, and `oct()`. The `__index__()` method is also used as a fallback for `int()`, `float()`, and `complex()` functions when their corresponding magic methods are not defined.

```
class My_Integer:
    def __init__(self, i):
        self.i = i

    def __index__(self):
        return self.i
x = My_Integer(1)
y = My_Integer(8)
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(my_list[x])
print(my_list[y])
```

```
2
9
```

❖ `__reversed__()`:

Python's `__reversed__` magic method implements the `reversed()` built-in function that returns a reverse iterator over the values of the given sequence such as a list, a tuple, or a string.

```
class Person:
    def __init__(self, name):
        self.name = name
    def __reversed__(self):
        return self.name[::-1]
alice = Person('alice')
print(reversed(alice))
```

```
ecila
```

❖ `__call__()`:

The Python `__call__` method makes a class callable, so you can call objects of the class like a normal function. For example, if you define `__call__(self)` on an object `x` of class `X`, you can call it like so: `x()`. The return value of the called object is the return value of the `__call__()` method.

```
class Person:
    def __call__(self, other):
        return f'Hi {other}'
alice = Person()
print(alice('Bob'))
```

```
Hi Bob
```

How to import a class from another file in Python ?

In this article, we will see How to import a class from another file in Python.

Import in Python is analogous to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. The import statement is that the commonest way of invoking the import machinery, but it's not the sole way. The import statement consists of the import keyword alongside the name of the module.

Getting Started

Here we have created a class named GFG which has two methods: add() and sub (). Apart from that an explicit function is created named method() in the same python file. This file will act as a module for the main python file.

```
class GFG:

    # methods
    def add(self, a, b):
        return a + b
    def sub(self, a, b):
        return a - b

# explicit function
def method():
    print("GFG")
```

Let the name of the above python file be **module.py**.

Importing

It's now time to import the module and start trying out our new class and functions. Here, we will import a module named **module** and create the object of the class named GFG inside that module. Now, we can use its methods and variables.

```
import module

# Created a class object
object = module.GFG()

# Calling and printing class methods
print(object.add(15,5))
print(object.sub(15,5))

# Calling the function
module.method()
```

Importing the module as we mentioned earlier will automatically bring over every single class and performance within the module into the namespace. If you're only getting to use one function, you'll prevent the namespace from being cluttered by only importing that function as demonstrated in the program below:

```
# import module
from module import method

# call method from that module
method()
```