In the Python programming language, there is no such thing as primitive data types as in C, instead, data stored inside objects

Type and identity of objects

Here is a definition of identity and type of objects from the Python documentation:

*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*](https://docs.python.org/3/reference/expressions.html#is)*’ operator compares the identity of two objects; the*[*id()*](https://docs.python.org/3/library/functions.html#id)*function returns an integer representing its identity.*

What we might think of as a variable, instead think of it as a **name**. When we assign a value to a variable:

<name> = <object>

We are actually **binding** a **name** to an **object.**One implication of this is that multiple names can be bound to a single object(in the example, there is only one “**HBTN”**string object, but two **names***s1 and s2*):

>>> s1 = "HBTN"  
>>> s2 = "HBTN"  
>>> id(s1)  
140121621438224  
>>> id(s1)  
140121621438224

Integers in this regard in Cpython are implemented differently:

>>> a = -10  
>>> b = -10  
>>> c = -10  
>>> id(a), id(b), id(c)  
(26866128, 26866392, 26866200)

As you can witness, although integers are immutable, the address of each variable is different while having the same value. Look at another example:

>>> a = 32  
>>> b = 32  
>>> c = 32  
>>> id(a), id(b), id(c)  
(26586736, 26586736, 26586736)

In this case, the addresses are the same. The reason behind this mystery is when you lunch the Python interpreter, it creates integer objects from -5 to 256 This is code extract from Cpython source code:

#define NSMALLPOSINTS 257  
#define NSMALLNEGINTS 5

So this means that if the integer is in this range the new integer object will not be created because it has already been created by the interpreter.

An **assignment statement** creates a **symbolic name** that you can use to reference an object. The statement x = 'foo' creates a symbolic name x that refers to the string object 'foo'.

A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a [dictionary](https://realpython.com/python-dicts) in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

 In a Python program, there are four types of namespaces:

1. Built-In
2. Global
3. Enclosing
4. Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they’re no longer needed. Typically, many namespaces will exist at any given time.

The **built-in namespace** contains the names of all of Python’s built-in objects. These are available at all times when Python is running. You can list the objects in the built-in.

dir(\_\_builtins\_\_)

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

The **global namespace** contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

### The Local and Enclosing Namespaces

As you learned in the previous tutorial on [functions](https://realpython.com/defining-your-own-python-function), the interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

Functions don’t exist independently from one another only at the level of the main program.

Each of these namespaces remains in existence until its respective function terminates. Python might not immediately reclaim the memory allocated for those namespaces when their functions terminate, but all references to the objects they contain cease to be valid.

The answer lies in the concept of **scope**. The [scope](https://realpython.com/python-scope-legb-rule/) of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.

To return to the above question, if your code refers to the name x, then Python searches for x in the following namespaces in the order shown:

1. **Local**: If you refer to x inside a function, then the interpreter first searches for it in the innermost scope that’s local to that function.
2. **Enclosing**: If x isn’t in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function’s scope.
3. **Global**: If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
4. **Built-in**: If it can’t find x anywhere else, then the interpreter tries the built-in scope.

This is the **LEGB rule** as it’s commonly called in Python literature (although the term doesn’t actually appear in the [Python documentation](https://docs.python.org/3)). The interpreter searches for a name from the inside out, looking in the **l**ocal, **e**nclosing, **g**lobal, and finally the **b**uilt-in scope:

If the interpreter doesn’t find the name in any of these locations, then Python raises a [NameError exception](https://docs.python.org/3/library/exceptions.html" \l "NameError).

Python provides built-in functions called globals() and locals() that allow you to access global and local namespace dictionaries.

### The globals() function

The built-in function globals() returns a reference to the current global namespace dictionary. You can use it to access the objects in the global namespace.

>>> x = 'foo'

>>> globals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None,

'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None,

'\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>,

'x': 'foo'}

### The locals() function

Python also provides a corresponding built-in function called locals(). It’s similar to globals() but accesses objects in the local namespace instead:

>>>

>>> def f(x, y):

... s = 'foo'

... print(locals())

...

>>> f(10, 0.5)

{'s': 'foo', 'y': 0.5, 'x': 10}

When called within f(), locals() returns a dictionary representing the function’s local namespace. Notice that, in addition to the locally defined variable s, the local namespace includes the function parameters x and y since these are local to f() as well.

If you call locals() outside a function in the main program, then it behaves the same as globals().

locals(), on the other hand, returns a dictionary that is a current copy of the local namespace, not a reference to it. Further additions to the local namespace won’t affect a previous return value from locals() until you call it again. Also, you can’t modify objects in the actual local namespace using the return value from locals()

A function can modify an object of mutable type that’s outside its local scope if it modifies the object in place:

### The global Declaration

What if you really do need to modify a value in the global scope from within f()? This is possible in Python using the global declaration:

>>>

>>> x = 20

>>> def f():

... global x

... x = 40

... print(x)

...

>>> f()

40

>>> x

40

The global x statement indicates that while f() executes, references to the name x will refer to the x that is in the global namespace. That means the assignment x = 40 doesn’t create a new reference. It assigns a new value to x in the global scope instead

NON LOCAL SCOPE

To modify x in the enclosing scope from inside g(), you need the analogous keyword [nonlocal](https://realpython.com/python-keywords/#variable-handling-keywords-del-global-nonlocal). Names specified after the nonlocal keyword refer to variables in the nearest enclosing scope:

>>>

1>>> def f():

2... x = 20

3...

4... def g():

5... nonlocal x

6... x = 40

7...

8... g()

9... print(x)

10...

11

12>>> f()

1340

## Python Operator Overloading

[Python operators](https://www.programiz.com/python-programming/operators) work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

## The Internals of Operations Like len() and []

Every class in Python defines its own behavior for built-in functions and methods. When you pass an instance of some class to a built-in function or use an operator on the instance, it is actually equivalent to calling a special method with relevant arguments.

If there is a built-in function, func(), and the corresponding special method for the function is \_\_func\_\_(), Python interprets a call to the function as obj.\_\_func\_\_(), where obj is the object. In the case of operators, if you have an operator opr and the corresponding special method for it is \_\_opr\_\_(), Python interprets something like obj1 <opr> obj2 as obj1.\_\_opr\_\_(obj2).

So, when you’re 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.

Therefore, when you define these special methods in your own class, you override the behavior of the function or operator associated with them because, behind the scenes, Python is calling your method

>>> a = 'Real Python'

>>> b = ['Real', 'Python']

>>> len(a)

11

>>> a.\_\_len\_\_()

11

>>> b[0]

'Real'

>>> b.\_\_getitem\_\_(0)

'Real'

>>> dir(a)

['\_\_add\_\_',

'\_\_class\_\_',

'\_\_contains\_\_',

'\_\_delattr\_\_',

'\_\_dir\_\_',

...,

'\_\_iter\_\_',

'\_\_le\_\_',

'\_\_len\_\_',

'\_\_lt\_\_',

...,

'swapcase',

'title',

'translate',

'upper',

'zfill'

But, when overloading len(), you should keep in mind that Python requires the function to return an integer. If your method were to return anything other than an integer, you would get a TypeError. This, most probably, is to keep it consistent with the fact that len() is generally used to obtain the length of a sequence, which can only be an integer.

You can dictate the behavior of the abs() built-in for instances of your class by defining the \_\_abs\_\_() special method in the class. There are no restrictions on the return value of abs(), and you get a TypeError when the special method is absent in your class definition.

### #Printing Your Objects Prettily Using str()

The str() built-in is used to cast an instance of a class to a str object, or more appropriately, to obtain a user-friendly string representation of the object which can be read by a normal user rather than the programmer. You can define the string format your object should be displayed in when passed to str() by defining the \_\_str\_\_() method in your class. Moreover, \_\_str\_\_() is the method that is used by Python when you call [print()](https://realpython.com/python-print/) on your object.

### #Representing Your Objects Using repr()

The repr() built-in is used to obtain the parsable string representation of an object. If an object is parsable, that means that Python should be able to recreate the object from the representation when repr is used in conjunction with functions like [eval()](https://realpython.com/python-eval-function/). To define the behavior of repr(), you can use the \_\_repr\_\_() special method.

### #Making Your Objects Truthy or Falsey Using 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\_\_() (\_\_nonzero\_\_() in Python 2.x) special method.

The behavior defined here will determine the truth value of an instance in all contexts that require obtaining a truth value such as in if statements.

### Making Your Objects Capable of Being Added Using +

The special method corresponding to the + operator is the \_\_add\_\_() method. Adding a custom definition of \_\_add\_\_() changes the behavior of the operator. It is recommended that \_\_add\_\_() returns a new instance of the class instead of modifying the calling instance itself. You’ll see this behavior quite commonly in Python:

>>>

>>> a = 'Real'

>>> a + 'Python' # Gives new str instance

'RealPython'

>>> a # Values unchanged

'Real'

>>> a = a + 'Python' # Creates new instance and assigns a to it

>>> a

'RealPython'

You can see above that using the + operator on a str object actually returns a new str instance, keeping the value of the calling instance (a) unmodified. To change it, we need to explicitly assign the new instance to a.

Let’s implement the ability to append new items to our cart in the Order class using the operator. We’ll follow the recommended practice and make the operator return a new Order instance that has our required changes instead of making the changes directly to our instance:

>>>

>>> 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(['banana', 'apple'], 'Real Python')

>>> (order + 'orange').cart # New Order instance

['banana', 'apple', 'orange']

>>> order.cart # Original instance unchanged

['banana', 'apple']

>>> order = order + 'mango' # Changing the original instance

>>> order.cart

['banana', 'apple', 'mango']

Similarly, you have the \_\_sub\_\_(), \_\_mul\_\_(), and other special methods which define the behavior of -, \*, and so on. These methods should return a new instance of the class as well.

### Shortcuts: the += Operator

The += operator stands as a shortcut to the expression obj1 = obj1 + obj2. The special method corresponding to it is \_\_iadd\_\_(). The \_\_iadd\_\_() method should make changes directly to the self argument and return the result, which may or may not be self. This behavior is quite different from \_\_add\_\_() since the latter creates a new object and returns that, as you saw above.

Roughly, any += use on two objects is equivalent to this:

>>> result = obj1 + obj2

>>> obj1 = result

Here, result is the value returned by \_\_iadd\_\_(). The second assignment is taken care of automatically by Python, meaning that you do not need to explicitly assign obj1 to the result as in the case of obj1 = obj1 + obj2.

Let’s make this possible for the Order class so that new items can be appended to the cart using +=:

>>>

>>> class Order:

... def \_\_init\_\_(self, cart, customer):

... self.cart = list(cart)

... self.customer = customer

...

... def \_\_iadd\_\_(self, other):

... self.cart.append(other)

... return self

...

>>> order = Order(['banana', 'apple'], 'Real Python')

>>> order += 'mango'

>>> order.cart

['banana', 'apple', 'mango']

### 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. You can change its behavior using the \_\_getitem\_\_() special method.

Let’s configure our Order class so that we can directly use the object and obtain an item from the cart:

>>>

>>> class Order:

... def \_\_init\_\_(self, cart, customer):

... self.cart = list(cart)

... self.customer = customer

...

... def \_\_getitem\_\_(self, key):

... return self.cart[key]

...

>>> order = Order(['banana', 'apple'], 'Real Python')

>>> order[0]

'banana'

>>> order[-1]

'apple'

You’ll notice that above, the name of the argument to \_\_getitem\_\_() is not index but key. This is because the argument can be of mainly three forms: **an integer value**, in which case it is either an index or a dictionary key, **a string value**, in which case it is a dictionary key, and [**a slice object**](https://docs.python.org/3/library/functions.html#slice), in which case it will slice the sequence used by the class. While there are other possibilities, these are the ones most commonly encountered.

Since our internal [data structure](https://realpython.com/python-data-structures/) is a list, we can use the [] operator to slice the list, as in this case, the key argument will be a slice object. This is one of the biggest advantages of having a \_\_getitem\_\_() definition in your class. As long as you’re using data structures that support slicing (lists, tuples, strings, and so on), you can configure your objects to directly slice the structure:

>>>

>>> order[1:]

['apple']

>>> order[::-1]

['apple', 'banana']

### Reverse Operators: Making Your Classes Mathematically Correct

While defining the \_\_add\_\_(), \_\_sub\_\_(), \_\_mul\_\_(), and similar special methods allows you to use the operators when your class instance is the left-hand side operand, the operator will not work if the class instance is the right-hand side operand:

>>>

>>> class Mock:

... def \_\_init\_\_(self, num):

... self.num = num

... def \_\_add\_\_(self, other):

... return Mock(self.num + other)

...

>>> mock = Mock(5)

>>> mock = mock + 6

>>> mock.num

11

>>> mock = 6 + Mock(5)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'Mock'

If your class represents a mathematical entity like a vector, a coordinate, or a [complex number](https://realpython.com/python-numbers/#complex-numbers), applying the operators should work in both the cases since it is a valid mathematical operation.

A Complete Example:

from math import hypot, atan, sin, cos

class CustomComplex:

def \_\_init\_\_(self, real, imag):

self.real = real

self.imag = imag

def conjugate(self):

return self.\_\_class\_\_(self.real, -self.imag)

def argz(self):

return atan(self.imag / self.real)

def \_\_abs\_\_(self):

return hypot(self.real, self.imag)

def \_\_repr\_\_(self):

return f"{self.\_\_class\_\_.\_\_name\_\_}({self.real}, {self.imag})"

def \_\_str\_\_(self):

return f"({self.real}{self.imag:+}j)"

def \_\_add\_\_(self, other):

if isinstance(other, float) or isinstance(other, int):

real\_part = self.real + other

imag\_part = self.imag

if isinstance(other, CustomComplex):

real\_part = self.real + other.real

imag\_part = self.imag + other.imag

return self.\_\_class\_\_(real\_part, imag\_part)

def \_\_sub\_\_(self, other):

if isinstance(other, float) or isinstance(other, int):

real\_part = self.real - other

imag\_part = self.imag

if isinstance(other, CustomComplex):

real\_part = self.real - other.real

imag\_part = self.imag - other.imag

return self.\_\_class\_\_(real\_part, imag\_part)

def \_\_mul\_\_(self, other):

if isinstance(other, int) or isinstance(other, float):

real\_part = self.real \* other

imag\_part = self.imag \* other

if isinstance(other, CustomComplex):

real\_part = (self.real \* other.real) - (self.imag \* other.imag)

imag\_part = (self.real \* other.imag) + (self.imag \* other.real)

return self.\_\_class\_\_(real\_part, imag\_part)

def \_\_radd\_\_(self, other):

  return self.\_\_add\_\_(other)

def \_\_rmul\_\_(self, other):

  return self.\_\_mul\_\_(other)

def \_\_rsub\_\_(self, other):

# x - y != y - x

if isinstance(other, float) or isinstance(other, int):

real\_part = other - self.real

imag\_part = -self.imag

return self.\_\_class\_\_(real\_part, imag\_part)

def \_\_eq\_\_(self, other):

# Note: generally, floats should not be compared directly

# due to floating-point precision

return (self.real == other.real) and (self.imag == other.imag)

def \_\_ne\_\_(self, other):

return (self.real != other.real) or (self.imag != other.imag)

def \_\_eq\_\_(self, other):

# Note: generally, floats should not be compared directly

# due to floating-point precision

return (self.real == other.real) and (self.imag == other.imag)

def \_\_ne\_\_(self, other):

return (self.real != other.real) or (self.imag != other.imag)

<https://docs.python.org/3/reference/datamodel.html>

del x doesn’t directly call x.\_\_del\_\_() — the former decrements the reference count for x by one, and the latter is only called when x’s reference count reaches zero.

object.**\_\_lt\_\_**(*self*, *other*)

object.**\_\_le\_\_**(*self*, *other*)

object.**\_\_eq\_\_**(*self*, *other*)

object.**\_\_ne\_\_**(*self*, *other*)

object.**\_\_gt\_\_**(*self*, *other*)

object.**\_\_ge\_\_**(*self*, *other*)

These are the so-called “rich comparison” methods. The correspondence between operator symbols and method names is as follows: x<y calls x.\_\_lt\_\_(y), x<=y calls x.\_\_le\_\_(y), x==y calls x.\_\_eq\_\_(y), x!=y calls x.\_\_ne\_\_(y), x>y calls x.\_\_gt\_\_(y), and x>=y calls x.\_\_ge\_\_(y).

A rich comparison method may return the singleton NotImplemented if it does not implement the operation for a given pair of arguments. By convention, False and True are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an if statement), Python will call [bool()](https://docs.python.org/3/library/functions.html" \l "bool" \o "bool) on the value to determine if the result is true or false.