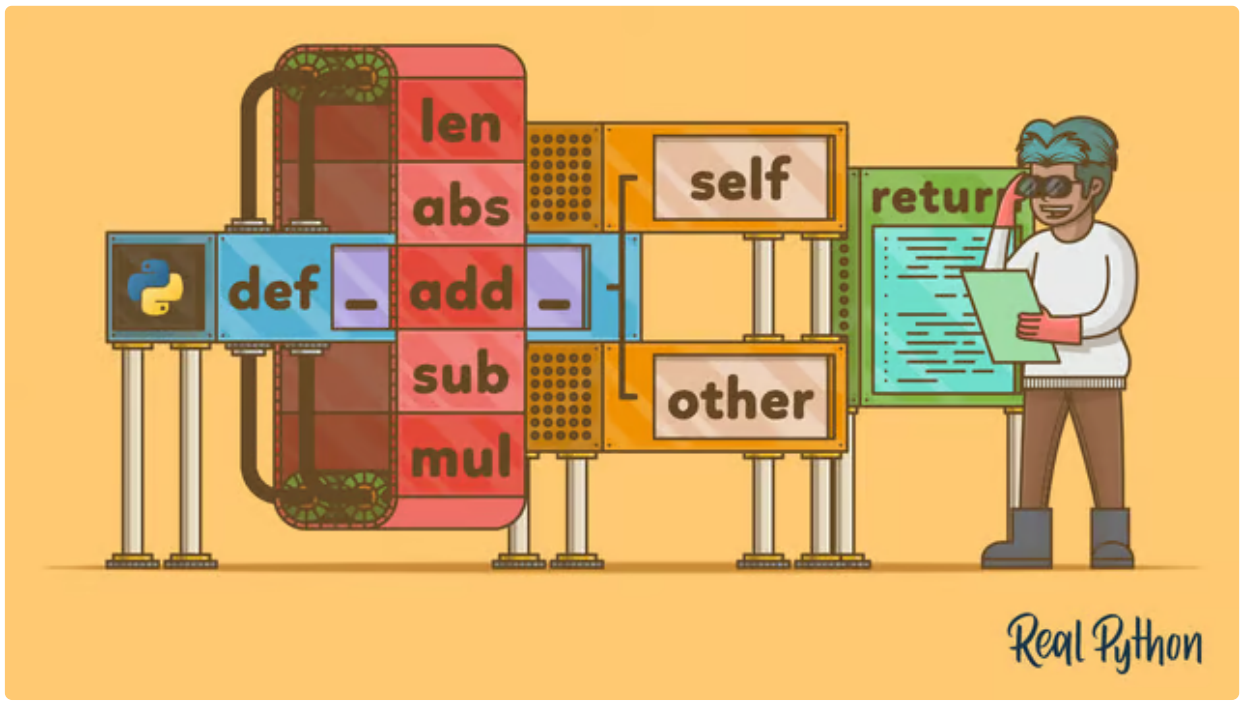

Operator and Function Overloading in Custom Python Classes

by Malay Agarwal 16 Comments

intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

If you've used the `+` or `*` operator on a `str` object in Python, you must have noticed its different behavior when compared to `int` or `float` objects:

Table of Contents

- The Python Data Model
- The Internals of Operations Like len() and []
- Overloading Built-in Functions
- Overloading Built-in Operators
- A Complete Example
- Recap and Resources

Python >>>

```
>>> # Adds the two numbers
>>> 1 + 2
3

>>> # Concatenates the two strings
>>> 'Real' + 'Python'
'RealPython'


>>> # Gives the product
>>> 3 * 2
6

>>> # Repeats the string
>>> 'Python' * 3
'PythonPythonPython'
```

You might have wondered how the same built-in operator or function shows different behavior for objects of different classes. This is called operator overloading or function overloading respectively. This article will help you understand this mechanism, so that you can do the same in your own Python classes and make your objects more Pythonic.

You'll learn the following:

- The API that handles operators and built-ins in Python
- The "secret" behind `len()` and other built-ins
- How to make your classes capable of using operators
- How to make your classes compatible with Python's built-in functions

As a bonus, you'll also see an example class, objects of which will be compatible with many of these operators and functions. Let's get started!

The Python Data Model

Say you have a class representing an online order having a cart (a `list`) and a customer (a `str` or instance of another class which represents a customer).

Note: If you need a refresher on OOP in Python, check out this tutorial on Real Python: Object-Oriented Programming (OOP) in Python 3

In such a case, it is quite natural to want to obtain the length of the cart list. Someone new to Python might decide to implement a method called `get_cart_len()` in their class to do this. But you can configure the built-in `len()` in such a way that it returns the length of the cart list when given our object.

In another case, we might want to append something to the cart. Again, someone new to Python would think of implementing a method called `append_to_cart()` that takes an item and appends it to the cart list. But you can configure the `+` operator in such a way that it appends a new item to the cart.

Python does all this using special methods. These special methods have a naming convention, where the name starts with two underscores, followed by an identifier and ends with another pair of underscores.

Essentially, each built-in function or operator has a special method corresponding to it. For example, there's `__len__()`, corresponding to `len()`, and `__add__()`, corresponding to the `+` operator.

By default, most of the built-ins and operators will not work with objects of your classes. You must add the corresponding special methods in your class definition to make your object compatible with built-ins and operators.

When you do this, the behavior of the function or operator associated with it changes according to that defined in the method.

This is exactly what the Data Model (Section 3 of the Python documentation) helps you accomplish. It lists all the special methods available and provides you with the means of overloading built-in functions and operators so that you can use them on your own objects.

Let's see what this means.

> **Fun fact:** Due to the naming convention used for these methods, they are also called *dunder methods* which is a shorthand for ***d**ouble **under**score methods*. Sometimes they're also referred to as *special methods* or *magic methods*. We prefer *dunder methods* though!

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. Let's get a better understanding of this:

Python >>>

```
>>> a = 'Real Python'
>>> b = ['Real', 'Python']
>>> len(a)
11
>>> a.__len__()
11
>>> b[0]
'Real'
>>> b.__getitem__(0)
'Real'
```

As you can see, when you use the function or its corresponding special method, you get the same result. In fact, when you obtain the list of attributes and methods of a `str` object using `dir()`, you'll see these special methods in the list in addition to the usual methods available on `str` objects:

Python >>>

```
>>> dir(a)
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 ...,
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 ...,
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

If the behavior of a built-in function or operator is not defined in the class by the special method, then you will get a `TypeError`.

So, how can you use special methods in *your* classes?

Overloading Built-in Functions

Many of the special methods defined in the Data Model can be used to change the behavior of functions such as `len`, `abs`, `hash`, `divmod`, and so on. To do this, you only need to define the corresponding special method in your class. Let's look at a few examples:

Giving a Length to Your Objects Using `len()`

To change the behavior of `len()`, you need to define the `__len__()` special method in your class. Whenever you pass an object of your class to `len()`, your custom definition of `__len__()` will be used to obtain the result. Let's implement `len()` for the order class we talked about in the beginning:

Python >>>

```
>>> class Order:
...     def __init__(self, cart, customer):
...         self.cart = list(cart)
...         self.customer = customer
...
...     def __len__(self):
...         return len(self.cart)
...
>>> order = Order(['banana', 'apple', 'mango'], 'Real Python')
>>> len(order)
3
```

As you can see, you can now use `len()` to directly obtain the length of the cart. Moreover, it makes more intuitive sense to say "length of order" rather than calling something like `order.get_cart_len()`. Your call is both Pythonic and more intuitive. When you don't have the `__len__()` method defined but still call `len()` on your object, you get a `TypeError`:

Python >>>

```
>>> class Order:
...     def __init__(self, cart, customer):
...         self.cart = list(cart)
...         self.customer = customer
...
>>> order = Order(['banana', 'apple', 'mango'], 'Real Python')
>>> len(order)  # Calling len when no __len__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'Order' has no len()
```

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:

Python >>>

```
>>> class Order:
...     def __init__(self, cart, customer):
...         self.cart = list(cart)
...         self.customer = customer
...
...     def __len__(self):
...         return float(len(self.cart))  # Return type changed to float
...
>>> order = Order(['banana', 'apple', 'mango'], 'Real Python')
>>> len(order)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

Making Your Objects Work With `abs()`

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.

In a class representing a vector in a two-dimensional space, `abs()` can be used to get the length of the vector. Let's see it in action:

Python >>>

```
>>> class Vector:
...     def __init__(self, x_comp, y_comp):
...         self.x_comp = x_comp
...         self.y_comp = y_comp
...
...     def __abs__(self):
...         return (self.x_comp ** 2 + self.y_comp ** 2) ** 0.5
...
>>> vector = Vector(3, 4)
>>> abs(vector)
5.0
```

It makes more intuitive sense to say “absolute value of vector” rather than calling something like `vector.get_mag()`.

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()` on your object.

Let’s implement this in the `Vector` class to format `Vector` objects as `xi+yj`. A negative y-component will be handled using the format mini-language:

Python >>>

```
>>> class Vector:
...     def __init__(self, x_comp, y_comp):
...         self.x_comp = x_comp
...         self.y_comp = y_comp
...
...     def __str__(self):
...         # By default, sign of +ve number is not displayed
...         # Using `+`, sign is always displayed
...         return f'{self.x_comp}i{self.y_comp:+}j'
...
>>> vector = Vector(3, 4)
>>> str(vector)
'3i+4j'
>>> print(vector)
3i+4j
```

It is necessary that `__str__()` returns a `str` object, and we get a `TypeError` if the return type is non-string.

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()`. To define the behavior of `repr()`, you can use the `__repr__()` special method.

This is also the method Python uses to display the object in a REPL session. If the `__repr__()` method is not defined, you will get something like `<__main__.Vector object at 0x...>` trying to look at the object in the REPL session. Let's see it in action in the `Vector` class:

Python >>>

```
>>> class Vector:
...     def __init__(self, x_comp, y_comp):
...         self.x_comp = x_comp
...         self.y_comp = y_comp
...
...     def __repr__(self):
...         return f'Vector({self.x_comp}, {self.y_comp})'
...

>>> vector = Vector(3, 4)
>>> repr(vector)
'Vector(3, 4)'

>>> b = eval(repr(vector))
>>> type(b), b.x_comp, b.y_comp
(__main__.Vector, 3, 4)

>>> vector  # Looking at object; __repr__ used
'Vector(3, 4)'
```

Note: In cases where the `__str__()` method is not defined, Python uses the `__repr__()` method to print the object, as well as to represent the object when `str()` is called on it. If both the methods are missing, it defaults to `<__main__.Vector ...>`. But `__repr__()` is the only method that is used to display the object in an interactive session. Absence of it in the class yields `<__main__.Vector ...>`.

Also, while this distinction between `__str__()` and `__repr__()` is the recommended behavior, many of the popular libraries ignore this distinction and use the two methods interchangeably.

Here's a recommended article on `__repr__()` and `__str__()` by our very own Dan Bader: Python String Conversion 101: Why Every Class Needs a "repr".

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.

As an example, for the `Order` class that was defined above, an instance can be considered to be truthy if the length of the cart list is non-zero. This can be used to check whether an order should be processed or not:

Python >>>

```
>>> class Order:
...     def __init__(self, cart, customer):
...         self.cart = list(cart)
...         self.customer = customer
...
...     def __bool__(self):
...         return len(self.cart) > 0
...
>>> order1 = Order(['banana', 'apple', 'mango'], 'Real Python')
>>> order2 = Order([], 'Python')

>>> bool(order1)
True
>>> bool(order2)
False

>>> for order in [order1, order2]:
...     if order:
...         print(f"{order.customer}'s order is processing...")
...     else:
...         print(f"Empty order for customer {order.customer}")
Real Python's order is processing...
Empty order for customer Python
```

> **Note:** When the `__bool__()` special method is not implemented in a class, the value returned by `__len__()` is used as the truth value, where a non-zero value indicates `True` and a zero value indicates `False`. In case both the methods are not implemented, all instances of the class are considered to be `True`.

There are many more special methods that overload built-in functions. You can find them in the documentation. Having discussed some of them, let's move to operators.

Overloading Built-in Operators

Changing the behavior of operators is just as simple as changing the behavior of functions. You define their corresponding special methods in your class, and the operators work according to the behavior defined in these methods.

These are different from the above special methods in the sense that they need to accept another argument in the definition other than `self`, generally referred to by the name `other`. Let's look at a few examples.

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:

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:

Python >>>

```
>>> 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:

Python

```
>>> 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 `+=`:

Python >>>

```
>>> 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']
```

As can be seen, any change is made directly to `self` and it is then returned. What happens when you return some random value, like a string or an integer?

Python >>>

```
>>> class Order:
...     def __init__(self, cart, customer):
...         self.cart = list(cart)
...         self.customer = customer
...
...     def __iadd__(self, other):
...         self.cart.append(other)
...         return 'Hey, I am string!'
...
>>> order = Order(['banana', 'apple'], 'Real Python')
>>> order += 'mango'
>>> order
'Hey, I am string!'
```

Even though the relevant item was appended to the cart, the value of `order` changed to what was returned by `__iadd__()`. Python implicitly handled the assignment for you. This can lead to surprising behavior if you forget to return something in your implementation:

Python >>>

```
>>> class Order:
...     def __init__(self, cart, customer):
...         self.cart = list(cart)
...         self.customer = customer
...
...     def __iadd__(self, other):
...         self.cart.append(other)
...
>>> order = Order(['banana', 'apple'], 'Real Python')
>>> order += 'mango'
>>> order  # No output
>>> type(order)
NoneType
```

Since all Python functions (or methods) return `None` implicitly, `order` is reassigned to `None` and the REPL session doesn't show any output when `order` is inspected. Looking at the type of `order`, you see that it is now `NoneType`. Therefore, always make sure that you're returning something in your implementation of `__iadd__()` and that it is the result of the operation and not anything else.

Similar to `__iadd__()`, you have `__isub__()`, `__imul__()`, `__idiv__()` and other special methods which define the behavior of `-=`, `*=`, `/=`, and others alike.

Note: When `__iadd__()` or its friends are missing from your class definition but you still use their operators on your objects, Python uses `__add__()` and its friends to get the result of the operation and assigns that to the calling instance. Generally speaking, it is safe to not implement `__iadd__()` and its friends in your classes as long as `__add__()` and its friends work properly (return something which is the result of the operation).

The Python documentation has a good explanation of these methods. Also, take a look at this example which shows the caveats involved with `+=` and the others when working with immutable types.

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:

Python >>>

```
>>> 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**, 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 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:

Python >>>

```
>>> order[1:]
['apple']
>>> order[::-1]
['apple', 'banana']
```

Note: There is a similar `__setitem__()` special method that is used to define the behavior of `obj[x] = y`. This method takes two arguments in addition to `self`, generally called `key` and `value`, and can be used to change the value at `key` to `value`.

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:

Python >>>

```
>>> 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, applying the operators should work in both the cases since it is a valid mathematical operation.

Moreover, if the operators work only when the instance is the left operand, we are violating the fundamental principle of commutativity in many cases. Therefore, to help you make your classes mathematically correct, Python provides you with **reverse special methods** such as `__radd__()`, `__rsub__()`, `__rmul__()`, and so on.

These handle calls such as `x + obj`, `x - obj`, and `x * obj`, where `x` is not an instance of the concerned class. Just like `__add__()` and the others, these reverse special methods should return a new instance of class with the changes of the operation rather than modifying the calling instance itself.

Let's configure `__radd__()` in the `Order` class in such a way that it will append something at the front of the cart. This can be used in cases where the cart is organized in terms of the priority of the orders:

Python >>>

```
>>> 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)
...
...     def __radd__(self, other):
...         new_cart = self.cart.copy()
...         new_cart.insert(0, other)
...         return Order(new_cart, self.customer)
...
>>> order = Order(['banana', 'apple'], 'Real Python')

>>> order = order + 'orange'
>>> order.cart
['banana', 'apple', 'orange']

>>> order = 'mango' + order
>>> order.cart
['mango', 'banana', 'apple', 'orange']
```

A Complete Example

To drive all these points home, it's better to look at an example class which implements these operators together.

Let's reinvent the wheel and implement our own class to represent complex numbers, `CustomComplex`. Objects of our class will support a variety of built-in functions and operators, making them behave very similar to the built-in complex numbers class:

Python

```
from math import hypot, atan, sin, cos

class CustomComplex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
```

The constructor handles only one kind of call, `CustomComplex(a, b)`. It takes positional arguments, representing the real and imaginary parts of the complex number.

Let's define two methods inside the class, `conjugate()` and `argz()`, which will give us the complex conjugate and the argument of a complex number respectively:

Python

```
def conjugate(self):
    return self.__class__(self.real, -self.imag)

def argz(self):
    return atan(self.imag / self.real)
```

Note: `__class__` is not a special method but a class attribute which is present by default. It has a reference to the class. By using it here, we are obtaining that and then calling the constructor in the usual manner. In other words, this is

equivalent to `CustomComplex(real, imag)`. This is done here to avoid refactoring the code if the name of the class changes someday.

Next, we configure `abs()` to return the modulus of a complex number:

Python

```
def __abs__(self):
    return hypot(self.real, self.imag)
```

We will follow the recommended distinction between `__repr__()` and `__str__()` and use the first for the parsable string representation and the second for a "pretty" representation.

The `__repr__()` method will simply return `CustomComplex(a, b)` in a string so that we can call `eval()` to recreate the object, while the `__str__()` method will return the complex number in brackets, as `(a+bj)`:

Python

```
def __repr__(self):
    return f"{self.__class__.__name__}({self.real}, {self.imag})"

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

Mathematically, it is possible to add any two complex numbers or add a real number to a complex number. Let's configure the `+` operator in such a way that it works for both cases.

The method will check the type of the right-hand side operator. In case it is an `int` or a `float`, it will increment only the real part (since any real number, `a`, is equivalent to `a+0j`), while in the case of another complex number, it will change both the parts:

Python

```
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)
```

Similarly, we define the behavior for `-` and `*`:

Python

```python
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)
```

Since both addition and multiplication are commutative, we can define their reverse operators by calling `__add__()` and `__mul__()` in `__radd__()` and `__rmul__()` respectively. On the other hand, the behavior of `__rsub__()` needs to be defined since subtraction is not commutative:

Python

```python
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)
```

Note: You might have noticed that we didn't add a construct to handle a `CustomComplex` instance here. This is because, in such a case, both the operands are instances of our class, and `__rsub__()` won't be responsible for handling the operation. Instead, `__sub__()` will be called. This is a subtle but important detail.

Now, we take care of the two operators, `==` and `!=`. The special methods used for them are `__eq__()` and `__ne__()`, respectively. Two complex numbers are said to be equal if their corresponding real and imaginary parts are both equal. They are said to be unequal when either one of these are unequal:

Python

```python
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)
```

> **Note:** The Floating-Point Guide is an article that talks about comparing floats and floating-point precision. It highlights the caveats involved in comparing floats directly, which is something we're doing here.

It is also possible to raise a complex number to any power using a simple formula. We configure the behavior for both the built-in `pow()` and the `**` operator using the `__pow__()` special method:

Python

```
def __pow__(self, other):
    r_raised = abs(self) ** other
    argz_multiplied = self.argz() * other

    real_part = round(r_raised * cos(argz_multiplied))
    imag_part = round(r_raised * sin(argz_multiplied))

    return self.__class__(real_part, imag_part)
```

> **Note:** Take a close look at the definition of the method. We are calling `abs()` to obtain the modulus of the complex number. So, once you've defined the special method for a particular function or operator in your class, it can be used in other methods of the same class.

Let's create two instances of this class, one having a positive imaginary part and one having a negative imaginary part:

Python >>>

```
>>> a = CustomComplex(1, 2)
>>> b = CustomComplex(3, -4)
```

String representations:

Python >>>

```
>>> a
CustomComplex(1, 2)
>>> b
CustomComplex(3, -4)
>>> print(a)
(1+2j)
>>> print(b)
(3-4j)
```

Recreating the object using `eval()` with `repr()`:

Python >>>

```
>>> b_copy = eval(repr(b))
>>> type(b_copy), b_copy.real, b_copy.imag
(__main__.CustomComplex, 3, -4)
```

Addition, subtraction, and multiplication:

Python >>>

```
>>> a + b
CustomComplex(4, -2)
>>> a - b
CustomComplex(-2, 6)
>>> a + 5
CustomComplex(6, 2)
>>> 3 - a
CustomComplex(2, -2)
>>> a * 6
CustomComplex(6, 12)
>>> a * (-6)
CustomComplex(-6, -12)
```

Equality and inequality checks:

Python >>>

```
>>> a == CustomComplex(1, 2)
True
>>> a ==  b
False
>>> a != b
True
>>> a != CustomComplex(1, 2)
False
```

Finally, raising a complex number to some power:

Python >>>

```
>>> a ** 2
CustomComplex(-3, 4)
>>> b ** 5
CustomComplex(-237, 3116)
```

As you can see, objects of our custom class behave and look like those of a built-in class and are very Pythonic. The full example code for this class is embedded below.

Solution: "A Complete Example" Show/Hide

Recap and Resources

In this tutorial, you learned about the Python Data Model and how the Data Model can be used to build Pythonic classes. You learned about changing the behavior of built-in functions such as `len()`, `abs()`, `str()`, `bool()`, and so on. You also learned about changing the behavior of built-in operators like `+`, `-`, `*`, `**`, and so forth.

After reading this, you can confidently create classes that make use of the best idiomatic features of Python and make your objects Pythonic!