Decorators in Python

Prelude

Python is different from other languages in that, in Python, *everything* is an Object in Python. What does that mean? I'll give you a couple of examples that will help you see how to think about this. This describes how Python handles things behind the scenes, out of sight of the programmer, but can help guide how you think about your coding.

Example 1: Integers
Let's say you have this code:

```
x = 3
y = 5
print(x + y)
```

When the Python interpreter reads this, it sees that the variable `x` is set equal to an integer value, so it decides that `x` is of type Integer. What does that mean? It actually means that `x` is an instance of the `class Integer`*, with all its associated methods and attributes. This includes the method `__add__()`, which tells Python what to do if presented with two Integer variables/values and is asked to add them together. Java or C or just about every other language that came before Python would think of the integer variable and the addition operation totally separately: It assigns a small chunk of memory to store the value of `x`. It would then define how to add integers as part of defining the addition operator. So, it would think of the Integer value as a fundamentally different type of thing than the operator +, where Python thinks of them as two different parts of the same object.

Aside: The fact that the same operator + works with floats, integers, strings, etc. is called *polymorphism*: It's an operator that decides what to do with its inputs based on the data type of the inputs, and behaves differently with different data types as inputs.

*-Actually, the class is called `Integral`. I'm not really sure why they went with the adjective form and not the noun form, but they did.

Example 2: Custom classes
Let's say you write:

```
class Book():
    def __init__(self, author, title):
        self.author = author
        self.title = title
    def __str__(self):
```

```
        return f"{self.title}, by {self.author}"

myBook = Book("Me", "How Awesome Am I?")
yourBook = Book("You", "You're Not As Awesome As I Am.")
print(myBook)
```

What the Python Interpreter does when it sees `print(myBook)` is it looks to see if there is a `__str__()` method inside the class Book. It sees it, and then it uses the string that is constructed there to decide what to print out when asked to print the contents of the variable myBook. So, you get:

```
How Awesome Am I?, by Me
```

What happens if you don't provide a `__str__()` method? You get something like: `<__main__.Book object at 0x10b08ebf0>`. Where did it get that?

That is the `__str__()` method in `class Object()` in the built-in Python. Because *everything* is an `Object` in Python, everything inherits these standard methods and falls back to using them if a new method of the same name hasn't been created by the developer in the new class. The built-in class `Object()` doesn't have, for example, an `__add__()` method, because Python doesn't know how to add Objects together – how to add depends on what the `Object` is. So, if you ask Python to calculate `myBook + yourBook`, then it will throw an error and tell you it doesn't know what to do with that. More specifically, you get something like: `TypeError: unsupported operand type(s) for +: 'Book' and 'Book'`-- Python is recognizing that the type `Book` does not have an `__add__()` operation, and there isn't a built-in `__add__()` method for `Object()` the way there is for `__str__()`. But, you could define an `__add__()` method, and then Python would use it, whether or not it makes any shred of sense to add books to each other. In a very real way, Python handles your custom `Book` class the same way that it handles the built-in class `Integral`.

Example 3: Modules/Libraries
When you put, at the top of your Python program:

```
import time
```

and then you want to use, say, the `sleep()` function from the time module, how do you call it?

```
time.sleep(5)
```

If you have, say, a `Book` class, and you've written a method `getPage()`, how do you call it?

```
myBook.getPage(5)
```

Those look the same. Why is that? Hmm…..

Everything is an `Object` in Python. Even modules. So, we usually think of `sleep` as a function in the `time` module, but it is also correct to think of `sleep` as a method of the `time` Object. Sometimes it is useful to think of it one way, sometimes the other.

Functions as Objects

So, are functions Objects? *Everything* is an Object in Python, so yes. In fact, every function is both an Object, and also a method of an Object (a method of its module, just like `sleep` inside of `time`). What Object (i.e. what module) is, say, `print()` inside of? All of the built-in functions of Python live inside a module called `__builtins__`, and the Python interpreter always runs:

```
from __builtins__ import *
```

before reading your program, and automatically loads all the built-in functions into the global namespace/scope.

How is it helpful to think of functions as Objects?

In order to get a simple enough example to make this quick, it will be a little silly, but bear with me on this. Let's say we have a program that is keeping track of scores of sports teams, but we aren't going to say what sport they play.

```
# score is a list of two integers, the two teams' scores.
# teamScored is either 0 or 1, to say which team scored.
def addTD(score, teamScored):
     # Assuming they are playing American football, so worth 6
points.
     score[teamScored] += 6
     return score
def addBasket(score, teamScored):
     # Assuming Basketball, basket is worth 2 points.
     score[teamScored] += 2
     return score
def score(score, teamScored, whichScore):
     # whichScore tells which function to use
     score = whichScore(score, teamScored)
     print(f"SCORE!!! New Score: {score[0]} to {score[1]}")
     return score
```

```
# Football game
ramsPatriots = [0, 0]
# Basketball game
warriorsKnicks = [0, 0]
# Rams score a touchdown
score(ramsPatriots, 0, addTD)
# Knicks score a basket
score(warriorsKnicks, 1, addBasket)
```

If you run this, you get output:

```
SCORE!!! New Score: 6 to 0
SCORE!!! New Score: 0 to 2
```

In the last function, `score`, we have the parameter `whichScore`. What data type should it receive?

It should receive a function. Notice that we just want to send the *name* of the function. Normally, we don't write the name of the function unless we want to use that function right now, so we always use parentheses after the name of the function. Here, we are just passing the name of the function, so we leave the parentheses off it because we don't want the function to run right now, we just want to let the `score` function know which other function to use.

So, we can write a function that takes a different function as an input. 🤔 How could that help us?

I thought we were going to talk about Decorators

Ah, yes, now we are ready to talk about decorators. Since this is Software Development not HGTV, we are talking about an extra line of code in Python, not someone who is going to come into your house and make everything look beautiful.

A decorator is a shortcut that modifies the way a function or method works. They are most often used on methods inside of classes, so I'll just assume that we are always talking about methods.

What's really happening is that the method is getting passed into another function as an input, and that other function is modifying the way your method behaves. It's just that Python abbreviates things so it's quicker to write.

The way you use a decorator is to add a line immediately before the `def myMethod(self):` line. Decorators always start with the `@` symbol. There are a bunch of pre-built ones in Python, I'll show you a couple.

First Example: `@classmethod`

If you write a class and then put a method inside that class, what is the first argument that **every** method takes? You always start with `self`, right? The anarchist in the back of the room says: "Rules are made to be broken!"

Here's a good reason to break that rule (no anarchy involved). Let's add a new method to the `Book` class:

```
    @classmethod
    def bookFromString(cls, bookStr):
        # Assumes input matches output from Book.__str__()
        bookSplit = bookStr.split(', by ')
        return Book(bookSplit[1], bookSplit[0])

myBook = Book.bookFromString("How Awesome Am I?, by Me")
```

This is one of the most common decorators, and one of the most common ways to use it. Let's break it down:

- If you are using the `@classmethod` decorator, the first argument for the method is not `self`, it is `cls`, which says it is taking the entire class as an argument.
- Normally, a method is a method of *an instance of the class*. So, like before, we could have `myBook.getPage(5)`. There, `getPage()` is called from the instance `myBook`.
- A `@classmethod` is exactly what it says: It's a method of the class itself. So, when we call it, we call it directly from the class: `Book.bookFromString()`
- The most common usage of the `@classmethod` decorator is to create an alternate constructor, as it is used here: you construct an instance of the `Book` class using a different collection of inputs. So, here, instead of creating a `Book` by giving the author and the title as two separate parameters, we create a `Book` by giving a string that has both the title and author inside it, in a certain format. The class method just returns a new instance of the `Book` class.

Class Attributes

This is a bit of a tangent, but it seems ridiculous not to mention it here: Classes are made up of methods and attributes and we just built a class *method*, so is there also a way to build a class *attribute*? Yes, sure, you just declare it between the `class` declaration and the `def __init__()`, like this:

```
class Book():
```

```
    library = "Manchester Library at The Bishop's School"
    def __init__(self, author, title):
```

Then, later, you can access this value using:

```
print(Book.library)
```

In addition, every instance of the class inherits this attribute automatically, and its default value is the one given there, like this:

```
print(myBook.library)
```

You can reset the value for any individual instance if you want, without changing the value for the class or for any other instance:

```
myBook.library = "Dr. Jaiclin's Personal Book Collection"
```

If you are inside the class, you access instance attributes using `self.attribute`. To access a class attribute inside the class, use `__class__.attribute`.

So, you can use class attributes if you want every item to have the same default value for an attribute, unless it is later overridden, or if you want the class itself to have an attribute, such as a library for books.

Class Attributes that are Lists

There's one trick to this: If the class attribute is a list, then each instance of the class inherits a *pointer* to that list (just like when you pass a list as an argument to a function), so if *any* instance changes that list, it is changed for *every* instance. .

This can be helpful, for example: Once upon a time, there was a little robot who had a few motors on it. I wanted to write a fairy tale about it, but instead I programmed its motors.

So, I wrote a `Motor` class that represented the controls for a motor on a robot. When the robot started up, the program would create one instance of this class for each motor that was connected, and the methods of that instance controlled that motor. So, to set the speed of the motor driving the front left wheel to 100%, you could have:

```
motorFrontLeft.setSpeed(100)
```

Then, there was a class attribute that I called `instances` that was a list of all instances of the motor class, one for each motor on the robot. Each time a new motor was initialized, the motor

was added to that list as part of the `__init__()` using `Motor.instances.append(self)`. Since the class attribute is a list, each time a motor adds itself to the list, the list is changed for the class, and for every instance of the class. That made it easy to do something like:

```
Motor.setSpeedAll(100)
```

which would just be a shortcut for:

```
@classmethod
def setSpeedAll(cls, speed):
for motor in Motor.instances:
     motor.setSpeed(speed)
```

This made for really readable, easy-to-write code using this class, you can see exactly what is going on without comments.

Notice that this is a different way to use the `@classmethod` decorator, where I'm not making an alternative constructor for the `Motor` class, I'm setting up a method that is called directly from the `Motor` class and is applied to the class as a whole instead of to one instance of the class, and this is an example of why you'd want to do that.

Other decorators: Getters and Setters

Getters and setters are methods used to get the value of (getter) or set the value of (setter) an attribute. Some languages enforce this idea by making the attributes inaccessible outside of the class itself, so that the only way to get or set the value of an attribute is through a getter or setter method. Python does not do this, but there is still a way to use getters and setters in Python, but only when it's necessary.

Continuing with the `Book` class from above, if we want to get the value of the author or title attribute, we just refer to it the same way we refer to a variable, except with a dot in it:

```
print(myBook.author)
print(myBook.title)
```

If we want to set that value instead, same thing, we just treat it like a variable that has a dot in the name:

```
myBook.author = "I'm the author"
myBook.title = "I'm so vain, I probably think this book is about me."
```

Getter Application #1: Read-Only Attributes

Python uses the name `@property` for an attribute with a Getter method, for example:

```
class Book():
    def __init__(self, author, title):
        self._author = author
        self.title = title
    def __str__(self):
        return f"{self.title}, by {self.author}"
    @property
    def author(self):
        return self._author
```

This seems pretty silly, because all I've done is complicate things: I replaced `self.author` with something called `self._author`, and then created a method that returns that value. (See below for more about the name `_author`). But, the job of a decorator is to change the behavior of the function, so we'd better think about that.

What this usage of this decorator does is it makes the `author` attribute *read-only*. If I set up the Book class this way, I can follow up with:

```
myBook = Book("Me", "How Awesome Am I?")
print(myBook.author)
```

Notice that the presence of the Getter method doesn't change how you retrieve the value, you still just refer to it like any other attribute. This is a little odd, because `author` is a method which would normally mean it should be called like a function. But, again, the job of the decorator is to modify the behavior of the method, and it makes the method not callable (so trying to call it like this: `myBook.author()` will crash the program), and makes `author` behave like an attribute. This is great because then the developer doesn't even have to know how the attribute is programmed, it is used the same way as any other attribute.

As I said, the attribute is read-only, so if you use:

```
myBook.author = "Dr. J"
```

This code will crash, and throw this exception: `AttributeError: can't set attribute`

In general, Python is really flexible, it wants to let the developer mess with things. Sometimes, you don't want to be that flexible. This decorator is a way you can enforce a protection of your data so that it can't be changed.

Why would you want that?

- If your class represents a customer to your business, and you have a whole bunch of other records (maybe stuff you sold to this customer) that identify this customer based on a Customer Number, you really don't want to change that Customer Number or else the rest of your data will become useless. So, set that value when you first create the customer and make sure you leave it alone by making it read-only using `@property`.
- If your class represents a sensor on a robot, when you create the instance, the processor connects to the sensor using its address. Once the sensor is checked and you know it is working, you want to leave that address information alone so that every time you go to get sensor values, that check should succeed. So, make the sensor address read-only using `@property`.

So, why did I use `_author` in that example? I used it intentionally for this helper attribute: this is a programming convention. When I use the word convention in this context, I mean that it is an agreed-upon way of doing things – there's no enforcement of this convention by the Python interpreter, it's just how things are done. Sort of like why we always put the *x*-axis horizontally and the *y*-axis vertically in math class: because that's how everyone else does it; math works fine if you don't do that, but everyone who reads your work will be confused. Some languages have ways of protecting an attribute, but Python doesn't do this directly. Putting the _ character in front of the attribute name doesn't actually change how Python behaves with it, but it tells a developer familiar with the typical conventions what you intended.

Getter Application #2: Validation in the Setter

Let's add another attribute for the price of the book. This is a little bit of a diversion, but is worth thinking about if you haven't before: Since I don't want to break all the other code I've written, I'll add it as an optional keyword argument (known as a `kwarg` in Python documentation):

```
class Book():
    def __init__(self, author, title, price = 0):
        self.author = author
        self.title = title
        self.price = price
```

If you're not familiar with kwargs, this means that, if a third parameter is there, it gets assigned to the variable `price`, but if not, then it uses the value 0 for `price` instead. You can also supply the keyword `price` in the call instead, so all of these work and produce the same result:

```
myBook = Book("Me", "I'm so vain, I think you'll spend $100 to read
about me")
myBook.price = 100
```

```
myBook = Book("Me", "I'm so vain, I think you'll spend $100 to read
about me", 100)

myBook = Book("Me", "I'm so vain, I think you'll spend $100 to read
about me", price = 100)
```

Often, the last of these is only useful if you have multiple keyword arguments, and you want to specify which one the value gets applied to.

Ok, so now we have a `price` attribute in the `Book` class. But, we want to make sure that that price is not negative. How to do this? Set up a getter and setter method, and write validation code in the setter method:

```
class Book():
    def __init__(self, author, title, price = 0):
        self.author = author
        self.title = title
        self.price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, newPrice):
        if newPrice >= 0:
            self._price = newPrice
        else:
            raise ValueError("Book price can't be negative.")
```

What this does is set up the `price` attribute so that it behaves from the outside like any other attribute, but on the inside, it's a little more complicated because it has a validation function that it passes through on its way to having its value set. It'll crash out if you try to give it a negative value.

So, the sample code from above will still work fine:

```
myBook = Book("Me", "I'm so vain, I think you'll spend $100 to read
about me")
myBook.price = 100
```

will produce an instance of the Book class with a price set at 100.

There's a detail in here: Notice I'm using the `_price` construction again, but I did *not* use it in the `__init__()` constructor method. Why? Because I went through the work to set up

validation code for that attribute, and I want Python to use it every time. So, when I use `self.price = price`, Python interprets the `self.price` as *a call to the setter method*, and forces the input into the constructor to go through the validation you set up in the setter method. Then, `self._price` stores the validated value.

Getter Application #3: Formatted Output

For one quick last example, I'll make the getter method in the price attribute slightly more complicated by enforcing that it will output the price using standard dollars and cents format:

```
@property
def price(self):
    return f"${self._price:.2f}"
```

This means that you will always know that you will get the price as a string, with a $ as the first character, and two decimal places to the right of the decimal point every time, regardless of the price and whether it was input as an integer or a float.

Summary Thoughts about Getters and Setters

- Python is built to make getters and setters unnecessary for the most typical case: when each attribute is readable and writable freely, you can just use the typical variable-style syntax.
- Getters and setters allow the developer to get more complex behavior built in to their classes without needing to change away from the variable-style syntax: you can have formatted output in your getters, or validation in your setters without needing to use function-style syntax.
- Together, if you use them properly, these allow downstream developers to to rely on variable-style syntax for all attribute values, which simplifies their usage of your classes.
- If you are working with developers with a lot of history in other languages, they may need to get used to this – this is opposite of what a lot of languages do (certainly Java), where you are forced into function-style syntax when interacting with attributes.

Other decorators

I could go on (and on and on … as I'm sure you know), but I won't. This gives you a framework to make sense of what a decorator is and how it is used in Python. Even if you don't know what the specific decorator does, you have the basic idea: the behavior of whatever comes on the line after the decorator is being modified somehow by the decorator. The decorator is a shortcut way of passing that method/function/object into some other function/method/object that modifies its behavior.