

# Learning Objectives - Getters and Setters

---

- Define the terms getter, setter, and data validation
- Differentiate between using the property function and the @property decorator for getters and setter
- Demonstrate validating data for type and value errors

# Getters and Setters with Methods

## Getters and Setters

Assume that programmers are respecting the convention that a single underscore makes an attribute private. Getters and setters are then used to access and manipulate these attributes. Getters (sometimes called accessors) are used to return a private attribute, and setters (sometimes called mutators) are used to update the value of a private attribute.

### ▼ Why are they called getters and setters?

The name getter comes from the fact that many programmers write a method that starts with `get_` followed by the attribute name. Similarly, the name setter comes from programmers write a method that starts with `set_` followed by the attribute name.

The `get_model` (a getter) acts as an intermediary between the user (outside the class) and the private attribute `_model` (inside the class). Similarly, the user invokes `set_model` (a setter) to update the “private” attribute `_model`.

```
class Phone:
    def __init__(self, model, storage, megapixels):
        self._model = model
        self._storage = storage
        self._megapixels = megapixels

    def get_model(self):
        return self._model

    def set_model(self, new_model):
        self._model = new_model

my_phone = Phone("iPhone", 256, 12)
print(my_phone.get_model())
my_phone.set_model("Galaxy S20")
print(my_phone.get_model())
```

challenge

## Try this variation:

- Write getters and setters for all of the instance variables in the Phone class.

### ▼ Solution

Getter methods should return the attribute, while setter methods should update the attribute with a new value.

```
class Phone:
    def __init__(self, model, storage, megapixels):
        self._model = model
        self._storage = storage
        self._megapixels = megapixels

    def get_model(self):
        return self._model

    def get_storage(self):
        return self._storage

    def get_megapixels(self):
        return self._megapixels

    def set_model(self, new_model):
        self._model = new_model

    def set_storage(self, new_storage):
        self._storage = new_storage

    def set_megapixels(self, new_megapixels):
        self._megapixels = new_megapixels
```

## Using Methods is not the “Python Way”

The Python community prides itself on simple, easy to read code. The code example below shows why using methods to implement getters and setters is not common.

### ▼ The Zen of Python

Python user Timothy Peters summarized the ethos of Python with what is

now called the [Zen of Python](#). This is worth a read as it gives you some general rules as to how your code should be written.

The class below has three private attributes. Getters and setters will be used to set the value of the third attribute to the sum of the first two.

```
class TestClass:
    def __init__(self, num1, num2):
        self._num1 = num1
        self._num2 = num2
        self._sum = 0

    def get_num1(self):
        return self._num1

    def set_num1(self, new_value):
        self._num1 = new_value

    def get_num2(self):
        return self._num2

    def set_num2(self, new_value):
        self._num2 = new_value

    def get_sum(self):
        return self._sum

    def set_sum(self, new_value):
        self._sum = new_value

obj = TestClass(5, 7)
print(obj.get_num1())
print(obj.get_num2())
obj.set_sum(obj.get_num1() + obj.get_num2())
print(obj.get_sum())
```

In particular, it is the following line that violates the Zen of Python.

```
obj.set_sum(obj.get_num1() + obj.get_num2())
```

You could argue that it is not simple nor beautiful. A more pythonic way of rewriting the offending line would be as this:

```
obj.sum(obj.num1 + obj.num2)
```

But this solution ignores private attributes, getters, and setters, right? Python has something called the property object which allows you to write code as if all the attributes are public. Behind the scenes, however, the leading underscore convention, getters, and setters are being used.

# Getters with the Property Decorator

---

## The Property Class

The property class in Python allows the programmer to use getters and setters in a pythonic way. There are two ways to implement the property class. One way is with the @property decorator, and the second way is with the property function. Both implementations do the same thing, and there is no difference in how the user would interact with the class. There is a slight difference in how the code is written. The @property decorator will be introduced first.

## Getters with the Property Decorator

The end goal is to be able to reference `.name` as if it were a public attribute. Start by setting up the `Person` class with the getter `name`. Test the code by invoking the getter.

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

    def name(self):
        return self._name

c = Person("Calvin")
print(c.name())
```

The program works as expected, but we want to get rid of the parentheses after `name`. Python uses the @property decorator so that `name` assumes getter behavior. Add the decorator and remove the inner parentheses from the print statement.

```

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

    @property
    def name(self):
        return self._name

c = Person("Calvin")
print(c.name)

```

To the end user, this looks and feels like the Person class has the public attribute name when in fact it is a getter for the `_name` attribute.

challenge

## Try this variation:

- Keep the `@property` decorator and change the print statement to

```
print(c.name())
```

### ▼ Why is there an error?

The `@property` decorator means `name` is treated as an instance variable and not a method. So `c.name` is the string "Calvin". Adding the parentheses to `c.name` means you are trying to "call" the string. You can call a method or a function, but you cannot call a string. That is why you see the error `TypeError: 'str' object is not callable`.

- Modify the Person class such that:
  - there is the instance variable `_age` and the getter `age` to access this variable.
  - Create the instance `c` with the attributes "Calvin" and 6.
  - Print these attributes with the getter methods.

### ▼ Solution

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

```
@property
def name(self):
    return self._name
```

```
@property
def age(self):
    return self._age
```

```
c = Person("Calvin", 6)
print(c.name)
print(c.age)
```



# Setters with the Property Decorator

---

## Setter

Setters can also take the `@property` decorator, but its implementation is different from the getter. The setter decorator starts with a `@` followed by the name of the getter method, and it ends with `.setter`. So if the getter is called `name`, the setter decorator is `@name.setter`. The setter also has the same name as the getter, but its parameters are `self` and the new value.

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

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, new_name):
        self._name = new_name

c = Person("Calvin")
print(c.name)
c.name = "Hobbes"
print(c.name)
```

To the end user, there is no indication that getters and setters are being used. It appears as though the programmer is directly manipulating the instance variable `name`. In reality, however, getters and setters are acting as an intermediary between the programmer and the private attribute `_name`.

challenge

## Try this variation:

- Comment out the @property decorator and run the code.

```
#@property  
def name(self):
```

### ▼ Why is there an error?

When you use the @property decorator, Python is actually making a property object. This object has a .setter method. If @property is commented out, there is no property object. So Python assumes that .setter is a method on the @name class, but that class does not exist. That is why you see the error `AttributeError: 'function' object has no attribute 'setter'`.

## Data Validation

The Person class below has the instance variables `_name` and `_age`. It also has getters and setters for both. Notice that the object `c` has its `_name` attribute changed to `False` and its `_age` set to `-17`.

```

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

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, new_name):
        self._name = new_name

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, new_age):
        self._age = new_age

c = Person("Calvin", "6")
print(c.name)
print(c.age)
c.age = -17
c.name = False
print(c.name)
print(c.age)

```

To Python, this is acceptable; no errors were made. To a human, however, a name should be expressed as a string and an age should be a positive number. Data validation is the process of asking if this data is appropriate for its intended use. Setters allow you to validate the data before changing the object.

Modify the setter for name so that it checks to see the data type of new\_name is not a string. If this is the case, raise an exception and display an error message to the user.

```

    @name.setter
    def name(self, new_name):
        if type(new_name) != str:
            raise TypeError("Names must be expressed as a string")
        self._name = new_name

```

## ▼ Raising Exceptions

The `raise` keyword stops the program and generates a custom error message. This is called raising an exception. `raise` is always followed by an exception (the error being committed). Python has lots of built-in exceptions, but the two of the most common are `TypeError` and `ValueError`. A `TypeError` occurs when a value is the wrong data type, and a `ValueError` occurs when the value is incorrect.

challenge

### Try this variation:

- Change the line of code that says `c.name = False` to `c.name = "False"`.
- Add data validation to the setter for `age`. Verify that the value of `new_age` is a positive number. **Hint**, see the “Raising Exceptions” drop down above.

#### ▼ Possible Solution

```
@age.setter
def age(self, new_age):
    if new_age < 0:
        raise ValueError("Age must be a positive number.")
    self._age = new_age
```

# Getters and Setters with the Property Function

---

## Getters with the Property Function

On the previous pages, we saw how to implement getters and setters with the `@property` decorator. The same functionality can be added by using the `property` method. Start with the `Person` class. Remove the `@property` decorator and change the name of the getter to `get_name`.

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

    def get_name(self):
        return self._name
```

Since `name` will be used as the getter and setter, create the variable `name` and set its value to `property(get_name)`. Instantiate an object with `"Calvin"` and invoke the getter.

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

    def get_name(self):
        return self._name

    name = property(get_name)

c = Person("Calvin")
print(c.name)
```

challenge

## Try this variation:

- Add the `_age` attribute to the `Person` class and create the getter `get_age`.

### ▼ Solution

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

    def get_name(self):
        return self._name

    def get_age(self):
        return self._age

    name = property(get_name)
    age = property(get_age)

c = Person("Calvin", 6)
print(c.name)
print(c.age)
```

## Setters with the Property Function

Creating a setter is done in a similar manner. Create `set_name` as a method to update the `_name` attribute. Then add `set_name` to the property function.

### ▼ Order is Important

When adding the getter and setter to the property function, it is the getter that always goes first. The setter is always second.

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

    def get_name(self):
        return self._name

    def set_name(self, new_name):
        self._name = new_name

    name = property(get_name, set_name)

c = Person("Calvin")
print(c.name)
c.name = "Hobbes"
print(c.name)
```

challenge

## Try this variation:

- Add the `_age` attribute to the `Person` class and create a getter and a setter. Change `_name` to "Hobbes" and `_age` to 8.

### ▼ Solution

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

    def get_name(self):
        return self._name

    def set_name(self, new_name):
        self._name = new_name

    def get_age(self):
        return self._age

    def set_age(self, new_age):
        self._age = new_age

    name = property(get_name, set_name)
    age = property(get_age, set_age)

c = Person("Calvin", 6)
print(c.name)
print(c.age)
c.name = "Hobbes"
c.age = 8
print(c.name)
print(c.age)
```



# **Getters and Setters Formative Assessment 1**

---

## **Getters and Setters Formative Assessment 2**

---