Object-Oriented Programming

Aidan Hunt, University of Washington Note: These notes cover both Lecture 13 and Lecture 14

Learning Objectives

After this lesson, students will be able to:

- Describe the key features of object oriented programming as a concept.
- Identify key components of objects in Python, such as data attributes and methods.
- Utilize Python syntax for defining and instantiating objects.
- Utilize Python syntax for calling object methods.

Check-in

- Homework 6 to be assigned this week
- Homework 5 grading in progress

Framing

So far, our programming has been *procedural*: a list of tasks for our computer to execute. Today, we'll explore an alternative framework for writing programs called object-oriented programming, which allows us to create "objects" that can accomplish tasks for us in a more flexible and modular way. Object-oriented programming has actually been with us this entire time --- it's how Python and many other programming languages are implemented!

What is object-oriented programming?

(See slides on course website)

Object-oriented programming (OOP) is a framework for:

- Designing your own data structures
- Grouping data and corresponding functions/processing methodology together.
- Intuitively representing physical objects in the programming space.

Motivation: Load Cell

- Processing data from load cell occurs in multiple steps. Components may change.
- Usual framework:
  - Define a function that performs this kind of calculation
  - Every time we want to process data, gather the individual pieces (tare, calibration, rotation matrix) and call this function.

- OOP framework: create a `LoadCell` **object** that represents the load cell in the programming space.
  - This object knows the tare, calibration, rotation of the physical load cell it represents
  - This object knows how to process data collected by the physical load cell it represents

Advantages of the Object-Oriented Approach

- Can give the `LoadCell` object additional functions (e.g., checking data validity, taking tares, estimating rotation matrix)
- More interactive
- Streamlines processing for multiple load cells

We've been using objects this whole time!

- NumPy Arrays and DataFrames are objects
- Lists and Strings are objects
- Everything in Python is an object!

A simple example of object-oriented programming

(See slides on course website)

An object encapsulates **state** and **behavior**. To unpack what this means, consider an object that represents something that many of us have interacted with: a Music Player (e.g., an iPod).

State

State of an object is data that describes the object.

Music Player state could be described by:

- The on/off state
- The volume
- A list of songs
- The color

Behavior

Behavior of an object are actions that the object can perform. Behaviors often, but not always, operate on the object's state.

For our Music Player, its behaviors could be:

- Turning the music player on or off.
- Increasing or decreasing the volume.
- Selecting a song from the list of songs.
- Adding a new song to the list of songs.
- Shuffling the song list.

Encapsulation

An object should **encapsulate** its state and behavior to guide the user experience. The user shouldn't need to know how the object works "under the hood" in order to use it.

- You don't know all the details about how your Music Player is designed or implemented, but you know how to use it!
- Think about the objects that we have used so far (e.g., NumPy arrays).
  - Don't know how they are implemented in Python
  - But we can easily use them via their associated methods

Implementing Objects

A `class` is a blueprint for how to make an object.

- This blueprint defines what it means to be a member of that class.
- When you create an instance of a class, you are creating an object from the blueprint. Objects are individuals, even though they share the same blueprint.

Inheritance

A class can build upon existing classes and add new state/behavior to the existing blueprint.

- This is called "inheritance"
- This is a way of factoring out redundancy between similar objects

Creating a MusicPlayer object in Python

Defining classes

Let's create a MusicPlayer class in Python! The key syntax is:

- Use the keyword `class`, then the class name.
- The keyword `pass` can be used as a placeholder for code-to-be (so we don't get errors)
- That's it! We've made a basic class

In [17]:
```
# Defining a class
class MusicPlayer:
    pass
```

In [18]:
```
# Create an instance of that class
myMusic = MusicPlayer()
```

In [19]:
```
# Check the type of the object that we created
type(myMusic)
```

Out[19]:
```
__main__.MusicPlayer
```

Adding object state

Okay, we've made a class, but it doesn't do anything. Let's define some state:

- Power (on/off)
- The volume

- A list of songs
- The color

Now, we can define an "initializer" or "constructor" that, when our object is created, initializes the state.

- This is just a special function.
- The indentation shows that this function belongs to the class
- The constructor takes a special argument, `self`, which refers to the *instance* of the class being created.
- The constructor returns a `MusicPlayer` object, but we don't need to explicitly return it.

In [20]:
```
# Defining a class
class MusicPlayer:

    # Define a constructor
    def __init__(self):
        self.power = False    # Power is well represented as a boolean
        self.volume = 0       # Volume is well represented as a numeric value
        self.color = 'black' # Color is well represented as a string
        self.songList = []    # Song list is well represented as a list of strings!
```

In [21]:
```
# Create an instance of that class
myMusic = MusicPlayer()

# Look at attributes (returns a dictionary)
vars(myMusic)
```

Out[21]:
```
{'power': False, 'volume': 0, 'color': 'black', 'songList': []}
```

You can **access** the attributes of your object using dot notation.

In [22]:
```
myMusic.power
```

Out[22]:
```
False
```

You can also **set** the attributes of your object using dot notation.

Like any other function, we can add additional inputs to our constructor function.

- For example, perhaps we want to set the color on initialization
- And perhaps we want to allow the user to provide a list of songs, but default to an empty list
- Note that `self.color = color` is assigning the value in the parameter `color` to the attribute `self.color`.

In [23]:
```
# Defining a class
class MusicPlayer:

    # Define a constructor
    def __init__(self, color, songList=[]):
        self.power = False    # Power is well represented as a boolean
        self.volume = 0       # Volume is well represented as a numeric value
        self.color = color # Color is well represented as a string
        self.songList = songList    # Song list is well represented as a list of strings!
```

In [24]:
```
myMusic = MusicPlayer('green', ['Let it be', 'Clint Eastwood', 'The Water'])
vars(myMusic)
```

Out[24]:
```
{'power': False,
 'volume': 0,
 'color': 'green',
 'songList': ['Let it be', 'Clint Eastwood', 'The Water']}
```

Adding Object Behavior

Okay, our object has behaviors, now let's make it do things for us. One example was turning the music player on-and-off.

- Define within the class block just like the `__init__` function
- Since this function is changing the object state, we don't have to return anything.
- Include docstrings as normal!

In [25]:
```
# Defining a class
class MusicPlayer:

    # Define a constructor
    def __init__(self, color, songList=[]):
        '''
        Constructs and returns a MusicPlayer object with the given and,
        if provided, the given song list. The MusicPlayer's power is
        initialized to False and the MusicPlayer's volume is initialized
        to 0.
        '''
        self.power = False    # Power is well represented as a boolean
        self.volume = 0       # Volume is well represented as a numeric value
        self.color = color # Color is well represented as a string
        self.songList = songList    # Song list is well represented as a list of strings!

    # Define a function for turning the MusicPlayer on and off
    def togglePower(self):
        '''
        Toggles the MusicPlayer power. If the MusicPlayer is off, turns
        it on. If on, turns it off. A message is printed displaying the
        new power state.
        '''
        # Invert power
        self.power = not self.power

        # Print message
        if self.power:
            print('Powering music player on.')
        else:
            print('Powering music player off.')
```

In [26]:
```
# Note that we must recreate the instance if we have changed the class (instance does no
myMusic.togglePower()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_2896\3312038632.py in <module>
      1 # Note that we must recreate the instance if we have changed the class (instance
 does not update automatically)
----> 2 myMusic.togglePower()

AttributeError: 'MusicPlayer' object has no attribute 'togglePower'
```

In [27]:
```
# Create object (power is off by default)
myMusic = MusicPlayer('green', ['Let it be', 'Clint Eastwood', 'The Water'])
```

```
# Toggle power
myMusic.togglePower()

# Toggle power
myMusic.togglePower()

# Toggle power
myMusic.togglePower()
```

```
Powering music player on.
Powering music player off.
Powering music player on.
```

Let's add some behavior for changing up the song data:

- **A function for changing the volume**
  - Takes in a number (numeric value expected)
  - If the number is between 0 and 100, set it as the new volume
  - Otherwise, print a message saying invalid volume, and don't change the volume
- **A function for adding a song to the songlist**
  - Takes an song input (string expected)
  - If its a string, adds it to the list
  - Adds it to the list
- **A function for playing a song**
  - Takes an integer representing the index of the song in the que to play
  - If the integer is a valid index for a song in the list
    - pop it from the songList
    - print the message `"Now playing <songName>"`
    - return the popped song
  - If the integer is an invalid index for a song in the list
    - print message `"Cannot play song"`
    - return None

Try these on your own, and then we'll do them together. Assume for now that the user will enter valid inputs.

In [28]:
```
# Defining a class
class MusicPlayer:

    # Define a constructor
    def __init__(self, color, songList=[]):
        '''
        Constructs and returns a MusicPlayer object with the given and,
        if provided, the given song list. The MusicPlayer's power is
        initialized to False and the MusicPlayer's volume is initialized
        to 0.
        '''
        self.power = False    # Power is well represented as a boolean
        self.volume = 0       # Volume is well represented as a numeric value
        self.color = color # Color is well represented as a string
        self.songList = songList    # Song list is well represented as a list of strings!

    # MusicPlayer related behaviors

    def togglePower(self):
        '''
        Toggles the MusicPlayer power. If the MusicPlayer is off, turns
        it on. If on, turns it off. A message is printed displaying the
```

```
        new power state.
        '''
        # Invert power
        self.power = not self.power

        # Print message
        if self.power:
            print('Powering music player on.')
        else:
            print('Powering music player off.')

    def changeVolume(self, newVolume):
        '''
        Given a new volume value between 0 and 100,
        sets the volume to be that value. If the volume
        value is invalid, the volume is not change and a
        message is printed.
        '''
        if 0 <= newVolume <= 100:
            self.volume = newVolume
        else:
            print('Invalid volume provided.')

    def addSong(self, song):
        '''
        Given a song name (as a string), adds the song
        to the song list if it a string. Otherwise, a
        message is printed saying the song cannot be added.
        '''
        if isinstance(song, str):
            self.songList.append(song)
        else:
            print(song, 'cannot be added to the queue.')

    def playSong(self, songNumber):
        '''
        Given the index of a song in the songList, removes
        the song from the list and "plays" it, and returns
        the song name as a string. If the songNumber
        provided is not a valid index, prints a message stating
        so.
        '''
        if 0 <= songNumber < len(self.songList):
            currSong = self.songList.pop(songNumber)
            print('Now playing:', currSong)
            return currSong
        else:
            print('Cannot play song.')
            return None
```

In [29]:
```
# Create object (power is off by default)
myMusic = MusicPlayer('green', ['Let it be', 'Clint Eastwood', 'The Water'])
```

In [30]:
```
# Change volume
myMusic.changeVolume(50)
vars(myMusic)
```

Out[30]:
```
{'power': False,
 'volume': 50,
 'color': 'green',
 'songList': ['Let it be', 'Clint Eastwood', 'The Water']}
```

In [31]:
```
# Try an invalid volume
myMusic.changeVolume(-90)
```

```
Invalid volume provided.
```

In [32]:
```
# Add a song
myMusic.addSong('Transatlantacism')
vars(myMusic)
```

Out[32]:
```
{'power': False,
 'volume': 50,
 'color': 'green',
 'songList': ['Let it be', 'Clint Eastwood', 'The Water', 'Transatlantacism']}
```

In [33]:
```
# Play a song
myMusic.playSong(1)
```

```
Now playing: Clint Eastwood
```

Out[33]:
```
'Clint Eastwood'
```

In [34]:
```
# Play an invalid song
myMusic.playSong(50)
```

```
Cannot play song.
```

Special behavior: Type conversion

What happens if we print our music player?

In [35]:
```
print(myMusic)
```

```
<__main__.MusicPlayer object at 0x0000022640422A60>
```

We just get the usual weird message saying where the MusicPlayer is stored in memory. But we can overwrite this with a special `__str__` function (see bottom-most function).

Note: Just like `__init__`, these special methods that start and end with two underscores that are meant to be called by Python, but never directly by the user.

In [36]:
```
# Defining a class
class MusicPlayer:

    # Define a constructor
    def __init__(self, color, songList=[]):
        '''
        Constructs and returns a MusicPlayer object with the given and,
        if provided, the given song list. The MusicPlayer's power is
        initialized to False and the MusicPlayer's volume is initialized
        to 0.
        '''
        self.power = False    # Power is well represented as a boolean
        self.volume = 0       # Volume is well represented as a numeric value
        self.color = color # Color is well represented as a string
        self.songList = songList    # Song list is well represented as a list of strings!

    # MusicPlayer related behaviors

    def togglePower(self):
        '''
        Toggles the MusicPlayer power. If the MusicPlayer is off, turns
        it on. If on, turns it off. A message is printed displaying the
        new power state.
        '''
        # Invert power
        self.power = not self.power
```

```
        # Print message
        if self.power:
            print('Powering music player on.')
        else:
            print('Powering music player off.')

    def changeVolume(self, newVolume):
        '''
        Given a new volume value between 0 and 100,
        sets the volume to be that value. If the volume
        value is invalid, the volume is not change and a
        message is printed.
        '''
        if 0 <= newVolume <= 100:
            self.volume = newVolume
        else:
            print('Invalid volume provided.')

    def addSong(self, song):
        '''
        Given a song name (as a string), adds the song
        to the song list if it a string. Otherwise, a
        message is printed saying the song cannot be added.
        '''
        if isinstance(song, str):
            self.songList.append(song)
        else:
            print(song, 'cannot be added to the queue.')

    def playSong(self, songNumber):
        '''
        Given the index of a song in the songList, removes
        the song from the list and "plays" it, and returns
        the song name as a string. If the songNumber
        provided is not a valid index, prints a message stating
        so.
        '''
        if 0 <= songNumber < len(self.songList):
            currSong = self.songList.pop(songNumber)
            print('Now playing:', currSong)
            return currSong
        else:
            print('Cannot play song.')
            return None

    # Methods for type conversion

    def __str__(self):
        '''
        Returns the songList of the MusicPlayer
        as its string representation.
        '''
        return str(self.songList)
```

In [37]:

```
# Create object (power is off by default)
myMusic = MusicPlayer('green', ['Let it be', 'Clint Eastwood', 'The Water'])
print(myMusic)
```

```
['Let it be', 'Clint Eastwood', 'The Water']
```

We can define this type of method for converting our music player to other types. For example, say that we want `int(myMusic)` and `float(myMusic)` to represent the MusicPlayer as the length of the song list. Then (see final methods):

In [38]:
```
# Defining a class
class MusicPlayer:

    # Define a constructor
    def __init__(self, color, songList=[]):
        '''
        Constructs and returns a MusicPlayer object with the given and,
        if provided, the given song list. The MusicPlayer's power is
        initialized to False and the MusicPlayer's volume is initialized
        to 0.
        '''
        self.power = False    # Power is well represented as a boolean
        self.volume = 0       # Volume is well represented as a numeric value
        self.color = color # Color is well represented as a string
        self.songList = songList    # Song list is well represented as a list of strings!

    # MusicPlayer related behaviors

    def togglePower(self):
        '''
        Toggles the MusicPlayer power. If the MusicPlayer is off, turns
        it on. If on, turns it off. A message is printed displaying the
        new power state.
        '''
        # Invert power
        self.power = not self.power

        # Print message
        if self.power:
            print('Powering music player on.')
        else:
            print('Powering music player off.')

    def changeVolume(self, newVolume):
        '''
        Given a new volume value between 0 and 100,
        sets the volume to be that value. If the volume
        value is invalid, the volume is not change and a
        message is printed.
        '''
        if 0 <= newVolume <= 100:
            self.volume = newVolume
        else:
            print('Invalid volume provided.')

    def addSong(self, song):
        '''
        Given a song name (as a string), adds the song
        to the song list if it a string. Otherwise, a
        message is printed saying the song cannot be added.
        '''
        if isinstance(song, str):
            self.songList.append(song)
        else:
            print(song, 'cannot be added to the queue.')

    def playSong(self, songNumber):
        '''
        Given the index of a song in the songList, removes
        the song from the list and "plays" it, and returns
        the song name as a string. If the songNumber
        provided is not a valid index, prints a message stating
        so.
        '''
        if 0 <= songNumber < len(self.songList):
            currSong = self.songList.pop(songNumber)
```

```
            print('Now playing:', currSong)
            return currSong
        else:
            print('Cannot play song.')
            return None

    # Methods for type conversion

    def __str__(self):
        '''
        Returns the songList of the MusicPlayer
        as its string representation.
        '''
        return str(self.songList)

    def __int__(self):
        '''
        Returns the length of the songList of the MusicPlayer
        as its int representation.
        '''
        return len(self.songList)

    def __float__(self):
        '''
        Returns the length of the songList of the MusicPlayer
        as its float representation.
        '''
        return float(self.__int__())
```

In [39]:
```
# Create object (power is off by default)
myMusic = MusicPlayer('green', ['Let it be', 'Clint Eastwood', 'The Water'])
```

In [40]:
```
int(myMusic)
```

Out[40]:
```
3
```

In [41]:
```
float(myMusic)
```

Out[41]:
```
3.0
```

Inheritance: Superclasses and subclasses

Now let's consider a `SmartPhone` object, which we want to share several aspects of the `MusicPlayer` object, but with some specialized behavior. The `SmartPhone` class can "inherit" from the MusicPlayer class.

In [42]:
```
# Define smartphone class
class SmartPhone(MusicPlayer):
    pass
```

In [43]:
```
# Create a smartphone using the same syntax as a MusicPlayer
# The SmartPhone looks exactly like a MusicPlayer!
sp = SmartPhone('black')
vars(sp)
```

Out[43]:
```
{'power': False, 'volume': 0, 'color': 'black', 'songList': []}
```

Based on the code above, a SmartPhone is the same as a MusicPlayer. But we can add new state and behavior entirely, or modify existing state/behavior. For example:

- Let's modify the constructor to add a new attribute, `.contacts`, a list of names

- Let's add a new function: given a name input, "call" the name if it is in the contacts list

In [44]:
```
# Define smartphone class
class SmartPhone(MusicPlayer):

    # Constructor: updated with new attribute
    def __init__(self, color, songList=[], contacts=[]):
        '''
        Constructs and returns a SmartPhone object with the given color, and,
        if provided, the given song list and contact list. The MusicPlayer's
        power is initialized to False and the MusicPlayer's volume
        is initialized to 0.
        '''
        # Add the new state
        self.contacts = contacts

        # Call the superclass constructor to do the rest
        super().__init__(color, songList)

    # Add a new function for calling a contact
    def call(self, name):
        '''
        Given the name of a contact to call, "calls" the contact
        if the name is in the contacts list. Otherwise, a message
        is printed saying that the contact was not found.
        '''
        if name in self.contacts:
            print('Calling', name)
        else:
            print(name, 'not found in contact list.')

    # Overwrite an old function: toggling the power
    def togglePower(self):
        '''
        Toggles the MusicPlayer power. If the MusicPlayer is off, turns
        it on. If on, turns it off. A message is printed displaying the
        new power state.
        '''
        # Invert power
        self.power = not self.power

        # Print message
        if self.power:
            print('Powering smart phone on.')
        else:
            print('Powering smart phone off.')
```

In [45]:
```
# Create a SmartPhone object
sp = SmartPhone('pink', songList=['Let it be', 'Clint Eastwood', 'The Water'],
                contacts=['Aidan', 'Stacey', 'Conor', 'Brittany'])
```

In [46]:
```
# Use overwritten superclass method, like toggling power
sp.togglePower()
sp.togglePower()
```

```
Powering smart phone on.
Powering smart phone off.
```

In [47]:
```
# Use unaltered superclass method, like playing a song
sp.playSong(1)
```

```
Now playing: Clint Eastwood
```

Out[47]:
```
'Clint Eastwood'
```

In [48]:
```
# Use new method defined by the smartphone subclass
sp.call('Aidan')
```

```
Calling Aidan
```

Type meaning for subclasses

What type is a `SmartPhone` object?

In [49]:
```
type(sp)
```

Out[49]:
```
__main__.SmartPhone
```

In [50]:
```
# Check to see if sp is a SmartPhone object
isinstance(sp, SmartPhone)
```

Out[50]:
```
True
```

In [51]:
```
# Check to see if sp is a MusicPlayer object
isinstance(sp, MusicPlayer)
```

Out[51]:
```
True
```

The `SmartPhone` object represented by `sp` is an instance of both the `SmartPhone` subclass and the `MusicPlayer` superclass! It is a member of both categories because of the superclass-subclass relationship between `MusicPlayer` and `SmartPhone`.

Privacy

How do we prevent someone from doing this and breaking our object (on accident or on purpose)?

In [52]:
```
# Changing the songList property to something that is not valid.
sp.songList = 12345
vars(sp)
```

Out[52]:
```
{'contacts': ['Aidan', 'Stacey', 'Conor', 'Brittany'],
 'power': False,
 'volume': 0,
 'color': 'pink',
 'songList': 12345}
```

In [53]:
```
# Adding a brand new attribute that doesn't mean anything
sp.newAttribute = 'Hello'
vars(sp)
```

Out[53]:
```
{'contacts': ['Aidan', 'Stacey', 'Conor', 'Brittany'],
 'power': False,
 'volume': 0,
 'color': 'pink',
 'songList': 12345,
 'newAttribute': 'Hello'}
```

There are ways to enforce certain conditions on object attributes, but they get very complicated very quickly.

- Python is a very "open" language, so enforcing types/shapes/formats for objects can be challenging.
- Culture is to document your functions it is clear what inputs/data types, etc are expected. Then it is the user's fault if something breaks.

- But if there are SAFETY concerns with your object breaking (e.g., motor control), then adding stricter privacy may be useful.

There are many ways to do it, and each have their own tradeoffs.

- Implementing "properties" (e.g., attributes with special associated getter/setter functions)
- `__slots__` properties for preventing new attributes
- I encourage you to look into this further if you are interested.

In [54]:
```
# Example of setter and getter methods for a color property (see last methods)

class MusicPlayer:

    # Define a constructor
    def __init__(self, color, songList=[]):
        '''
        Constructs and returns a MusicPlayer object with the given and,
        if provided, the given song list. The MusicPlayer's power is
        initialized to False and the MusicPlayer's volume is initialized
        to 0.
        '''
        self.power = False    # Power is well represented as a boolean
        self.volume = 0       # Volume is well represented as a numeric value
        self.color = color # Color is well represented as a string
        self.songList = songList    # Song list is well represented as a list of strings!

    # MusicPlayer related behaviors

    def togglePower(self):
        '''
        Toggles the MusicPlayer power. If the MusicPlayer is off, turns
        it on. If on, turns it off. A message is printed displaying the
        new power state.
        '''
        # Invert power
        self.power = not self.power

        # Print message
        if self.power:
            print('Powering music player on.')
        else:
            print('Powering music player off.')

    def changeVolume(self, newVolume):
        '''
        Given a new volume value between 0 and 100,
        sets the volume to be that value. If the volume
        value is invalid, the volume is not change and a
        message is printed.
        '''
        if 0 <= newVolume <= 100:
            self.volume = newVolume
        else:
            print('Invalid volume provided.')

    def addSong(self, song):
        '''
        Given a song name (as a string), adds the song
        to the song list if it a string. Otherwise, a
        message is printed saying the song cannot be added.
        '''
        if isinstance(song, str):
            self.songList.append(song)
```

```
        else:
            print(song, 'cannot be added to the queue.')

    def playSong(self, songNumber):
        '''
        Given the index of a song in the songList, removes
        the song from the list and "plays" it, and returns
        the song name as a string. If the songNumber
        provided is not a valid index, prints a message stating
        so.
        '''
        if 0 <= songNumber < len(self.songList):
            currSong = self.songList.pop(songNumber)
            print('Now playing:', currSong)
            return currSong
        else:
            print('Cannot play song.')
            return None

    # Type conversion methods

    def __str__(self):
        '''
        Returns the songList of the MusicPlayer
        as its string representation.
        '''
        return str(self.songList)

    def __int__(self):
        '''
        Returns the length of the songList of the MusicPlayer
        as its int representation.
        '''
        return len(self.songList)

    def __float__(self):
        '''
        Returns the length of the songList of the MusicPlayer
        as its float representation.
        '''
        return float(self.__int__())

    # Setting/getting methods for the color property

    @property
    def color(self):
        '''
        This function defines what is returned to the user when they use
        dot notation to get the color attribute of a MusicPlayer. Here,
        we are just returning value of the color attribute directly.
        Note that we are using a "private" attribute ._color, whereas
        .color is now a "property".
        '''
        return self._color

    @color.setter
    def color(self, newColor):
        '''
        This function defines what is returned to the user when they use
        dot notation to try to set the color attribute of a MusicPlayer.
        If the newColor provided is a string, we set the color attribute to
        be newColor. Otherwise, we raise a TypeError.
        '''
        if type(newColor) == str:
            self._color = newColor
        else:
```

```
            raise TypeError('Provided color must be of type str.')
```

In [55]:
```
myMusic = MusicPlayer('green')
```

In [56]:
```
# Get the color (like before)
myMusic.color
```

Out[56]:
```
'green'
```

In [57]:
```
# Change the color (like before)
myMusic.color = 'Pink'
myMusic.color
```

Out[57]:
```
'Pink'
```

In [58]:
```
# Try to change the color to something invalid
myMusic.color = 5555555
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_2896\1019815319.py in <module>
      1 # Try to change the color to something invalid
----> 2 myMusic.color = 5555555

~\AppData\Local\Temp\ipykernel_2896\1030640280.py in color(self, newColor)
    119             self._color = newColor
    120         else:
--> 121             raise TypeError('Provided color must be of type str.')
    122

TypeError: Provided color must be of type str.
```