

# Foundations of Programming (Python)

by Dirk Biesinger

## Module 08 of 10

### Introduction

Object Oriented Programming is a different way of thinking about programming. The roots of the methodology go back to MIT in the 1950s and 60s and concentrated on artificial intelligence projects. That it is used as a general paradigm goes back to the 1990s, when Delphi was an OO approach of the Borland team (creators and maintainers then of the Pascal and turbo pascal languages). Beginnings of C++ also stems out of this time as well as Java (to name a selection). Today most programming languages are or have elements of OOP (even COBOL).

The basic and “new” concept is, that everything is an object. An object has attributes and methods. We have so far worked with these all the time. Python is a pure Object Oriented Language. In this chapter you’ll take your first steps into understanding OOP and learn about:

- Creating Classes to define Objects
- Write methods and create attributes for objects
- Instantiate Objects (from classes)
- Restrict Access to an object’s attributes

We will be using spyder as our IDE.

### Index

Module 08 of 10.....	1
Introduction .....	1
Index.....	1
Classes.....	2
Fields .....	3
LAB 08-A: Working with Classes: .....	3
Constructors.....	3
LAB 08-B: Working with Constructors: .....	4
Destructors.....	4
The Keyword self.....	5
Attributes .....	5
Lab 08-C: Working with Attributes .....	6
Properties.....	6
LAB 08-D: Working with properties .....	10
Methods.....	10
The __str__() method: .....	11
LAB 08-E: Working with Methods: .....	12

Static Methods .....	13
Private Methods:.....	14
Decorators.....	16
Type hints.....	16
Doc Strings .....	16
Using classes .....	17
Summary .....	17

OOP has a reputation of being difficult to understand and overly complicated. But you’ve already made use of these principles all the time! You just don’t know it yet. So, let’s see how we construct an object and add “live” to it. You’ll soon recognize the patterns!

## Classes

Classes are the blueprint for an object. A class packages the data and functionality of an object. The object itself that you’ll actually use is an instantiation of this class. Like when you’re creating a string or tuple or list. Each instance (the individual objects) has the same functionality as the class, but is “customized” by the attributes used during creation (or initialization).

Don’t worry if it is not clear (yet), it’ll make sense in a moment. For this we’ll go back to our CD Inventory. We have used all the building blocks of this blueprint. To differentiate when they are used in a class, we call functions methods and variables and constants are fields.

Each object of a type (each instance of the class) uses the same blueprint, but generates it’s own copy in memory. This way, changes to one object do not affect the other objects of the same type.

Let’s develop a class that holds CD information.

The principle construct of a class has the following structure (in pseudo-code). Keep in mind that not all parts are required for each class.

**Note:** I will be omitting docstrings to keep the listings short enough to embed in text

```

1  #-----#
2  # Title: Can_00.py
3  # Desc: CanOnAString - Demonstrator class
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class MyClassName(InheritFromBaseClass):
9
10     ...# -- Fields -- #
11     ...# -- Constructor -- #
12     ...# -- Attributes -- #
13     ...# -- Properties -- #
14     ...# -- Methods -- #
15     ...pass
16
17
```

Figure 1 - structure of a class (pseudocode)

Line 8: We’re creating a class named MyClassName. This class inherits from the Class in parenthesis. If no parent class is given, Python implicitly defaults to `Object`.

## Fields

Fields are the data stores of a class. Fields get created same way as variables so far. This listing shows a minimal class with a few fields. We create two object instances in lines 18 and 21 and populate the fields in the following line. Notice that we need to refer the instance aka `objectName.fieldName` for this. We also can read the values in the fields with the same syntax.

```
1  #-----#
2  # Title: Can_01.py
3  # Desc: CanOnAString - Demonstrator class 1
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     ...#---Fields---#
11     ...message = ''
12
13     ...#---Constructor---#
14     ...#---Attributes---#
15     ...#---Properties---#
16     ...#---Methods---#
17
18     objCan1 = CanOnAString()
19     objCan1.message = 'There is no spoon'
20
21     objCan2 = CanOnAString()
22     objCan2.message = 'Live long and prosper'
23
24     print('1st: {}'.format(objCan1.message))
25     print('2nd: {}'.format(objCan2.message))
26
```

Listing 1 - classes 1

```
In [1]: runfile('C:/_FDProgramming/Mod08/Can_01.py', wdir='C:/_FDProgramming/Mod08')
1st: There is no spoon
2nd: Live long and prosper
```

Figure 2 - classes 1

## LAB 08-A: Working with Classes:

In this Lab, we'll work with classes.

The task / data we use: Create a class to hold information about music tracks / songs on a CD / music album. (position, title, length)

- Create a class file, save it as Lab08\_A.py in the Mod\_08 folder.
- Add code to create a class TrackInfo.
- Add code to create fields for position, title, length with data types int, string, string, respectively.
- Test the class and write down how the code works.

## Constructors

Constructors are a dedicated (special) method that is invoked when creating an object. Constructors are a convenient way to ensure proper datatypes in the fields. They also allow for a pre-population (defaults) of values to the fields. Same principle as using default values in functions.

Python's constructor method is the dunder init method: `__init__()`. To instantiate an object you simply call the Class's name as if it were a function:

```
objCan1 = CanOnAString('There is no spoon')
```

Python implicitly calls the `__init__()` method and passes any arguments provided when creating an object to the `__init__()` method.

```
1  #-----#
2  # Title: Can_02.py
3  # Desc: CanOnAString - Demonstrator class 2
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     ...#---Fields---#
11     ...message=''
12
13     ...#---Constructor---#
14     def __init__(self, msg):
15         ...#---Attributes---#
16         ...self.message=msg
17     ...#---Properties---#
18     ...#---Methods---#
19
20     objCan1 = CanOnAString('There is no spoon')
21
22     objCan2 = CanOnAString('Live long and prosper')
23
24     print('1st: {}'.format(objCan1.message))
25     print('2nd: {}'.format(objCan2.message))
26
```

Listing 2 - classes 2

```
In [2]: runfile('C:/_FDProgramming/Mod08/Can_02.py', wdir='C:/_FDProgramming/Mod08')
1st: There is no spoon
2nd: Live long and prosper
```

Figure 3 - classes 2

Constructors are a specialized method. They are run once during creation of the object. They are limited to this one purpose and hence are implicitly called when creating an object with a function call like syntax.

### LAB 08-B: Working with Constructors:

In this Lab, we'll add a constructor to the class we created in Lab 08-A.

- Make a copy of Lab08\_A.py and save it as Lab08\_B.py
- Add code to create a constructor.
- Add code to populate the class fields.
- Test the class and write down how the code works.

### Destructors

As the name implies, these methods are used when an object gets destroyed or de-allocated. Depending on the language you're using, these need to be more or less sophisticated. In most cases, they are responsible of freeing up the memory used by the object, cleaning up references and similar tasks. In python, usually you don't need to use

destructors, as the cleanup is done by the runtime. However, there are advanced constructs and contexts in which it is necessary to do some housekeeping. These are too advanced for this course. I mentioned them so you heard about them. In general terms their code looks like:

```
def __del__(self):  
    """automatically called on destruction of object"""  
    # TODO add housekeeping functionality here
```

Listing 3 - destructor

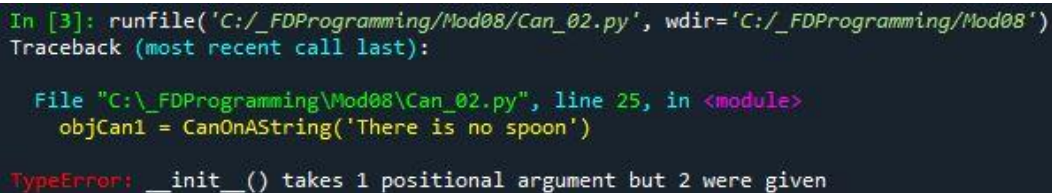
## The Keyword self

First of all, `self` is not an *official* keyword. But its use and convention in the community is so universal it might as well be.

So, let's unravel the mystery around it: It is the first parameter in every method. Every method that is called by an object automatically receives a reference to that object. This way the class knows on which object to use the methods. Think back when we were working with lists. We could hand around the reference to the list WITHOUT copying the data all the time. It's the same thing here: By calling a method on an object, this method gets a reference to the object calling the method and therefor can access the attributes, methods and handle what we want it to do. And that is all there is to the mystery of self.

**Note:** Although it is possible to name this first parameter something other than self, you shouldn't. Everybody will expect to see self.

It is not automatically added if you forgot to type it. In case of listing 2, leaving out self and creating a new object with the parameters we expect for it, we would get an error:



```
In [3]: runfile('C:/_FDProgramming/Mod08/Can_02.py', wdir='C:/_FDProgramming/Mod08')  
Traceback (most recent call last):  
  
  File "C:/_FDProgramming/Mod08/Can_02.py", line 25, in <module>  
    objCan1 = CanOnAString('There is no spoon')  
  
TypeError: __init__() takes 1 positional argument but 2 were given
```

Figure 4 - three parameters set by programmer and self not included in constructor

## Attributes

In Python, Attributes are internal fields or variables that hold data. Below code demonstrates implicitly created fields.

```

1  #-----#
2  # Title: Can_03.py
3  # Desc: CanOnAString - Demonstrator class
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     ...# --- Fields --- #
11     ...fldMessage = ''
12
13     ...# --- Constructor --- #
14     ...def __init__(self, msg):
15         ...# --- Attributes --- #
16         ...self.message = msg
17
18     ...# --- Properties --- #
19     ...# --- Methods --- #
20
21     objCan1 = CanOnAString('There is no spoon')
22
23
24     print('1st: {}'.format(objCan1.message))
25     print('2nd: {}'.format(objCan1.fldMessage))
26

```

Listing 4 - classes 3

```

In [4]: runfile('C:/_FDProgramming/Mod08/Can_03.py', wdir='C:/_FDProgramming/Mod08')
1st: There is no spoon
2nd:

```

Figure 5 - classes 3

**Note:** This feature is not typically seen in most other languages. It is due to the implicit and automatic nature of Python.

One issue with attributes is that they are ‘just’ variables. **You have no control** over what goes into them or how they change during runtime from external to your class **unless** you **write specific code to validate values** before they are assigned. In order to facilitate this, we can use special methods called Properties.

### Lab 08-C: Working with Attributes

In this Lab, we’ll add attributes to the class we created in Lab 08-B.

- Make a copy of Lab08\_B.py and save it as Lab08\_C.py
- Add code to create attributes.
- Add code to populate the class attributes.
- Test the class and write down how the code works.

### Properties

One common concept of controlling validity of values assigned to attributes in your class is to make the attributes private and enforce the interaction with them thru methods that have control mechanism build in. These special methods are called properties. Typically, you’ll create two for each attribute: One to set it and one to access it. Most often they are called “getter” or “accessor” for reading (or getting) the attribute and “setter” or “mutator” for writing (or setting) of the attribute. In most programming languages you can use a private keyword to make attributes (also applicable to functions) private to only be accessible from within the class.



In Python, you make an attribute private by pre-pending a double underscore to its name.

Getter methods let you add code to format the fields or attributes data. Even when no formatting is necessary, a getter property is required to allow access to the private field or attribute.

Getters are created like any other method with the addition of a decorator to identify it as a property getter. (more on decorators later in this module)

Setter methods let you add validation as well as error handling statements to evaluate parameters passed into the property. If the values are valid, then it is assigned to a field or attribute.

Setters are created like any other method with the addition of a decorator to identify it as a property setter.

When defining getters and setters in Python, make sure to first define the getter property followed by the setter property.

```
1  #-----#
2  # Title: Can_04.py
3  # Desc: CanOnAString -- Demonstrator class 4
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     #---Fields---#
11     #---Constructor---#
12     def __init__(self, msg):
13         #---Attributes---#
14         self.__message = msg
15         #---Properties---#
16
17         @property
18         def message(self):
19             return self.__message.title()
20
21         @message.setter
22         def message(self, value):
23             if str(value).isnumeric():
24                 raise Exception('The Message can\'t be cryptic')
25             else:
26                 self.__message = value
27         #---Methods---#
28
29     objCan1 = CanOnAString('There is no spoon')
30
31     print('1st: {}'.format(objCan1.message))
32     print('Now let\'s test direct access!')
33     print('2nd: {}'.format(objCan1.__message))
34
```

Listing 5 - classes 4

```
In [6]: runfile('C:/_FDProgramming/Mod08/Can_04.py', wdir='C:/_FDProgramming/Mod08')
1st: There Is No Spoon
Now let's test direct access!
Traceback (most recent call last):

  File "C:\_FDProgramming\Mod08\Can_04.py", line 33, in <module>
    print('2nd: {}'.format(objCan1.__message))
AttributeError: 'CanOnAString' object has no attribute '__message'
```

Figure 6 - classes 4

**Note:** Python uses an inconsistent nomenclature for the decorators for setters and getters.

In OOP in general, it is considered best practice to only work with the data in a class through a method or property. This creates a layer of abstraction. Abstraction means that the programmer using your class does not need to be concerned about the details and inner workings. Simply using your class and relying that your class gets the job done is another way of looking at this. And in order to achieve this, setters and getters create this abstraction for accessing the class attributes and fields.

**Note:** In Python by design, everything is open and accessible. It is more by convention that dunder and single underscore names inside a class are not being accessed directly. (technically it's possible). This means the “private” of a dunder attribute, field or method is private because others respect the privacy!

Putting it all together and some tests:



```

1  #-----#
2  # Title: Can_05.py
3  # Desc: CanOnAString - Demonstrator class 5
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     #---Fields---#
11     #---Constructor---#
12     def __init__(self, msg):
13         #---Attributes---#
14         self.__message = msg
15         #---Properties---#
16
17         @property
18         def message(self):
19             return self.__message.title()
20
21         @message.setter
22         def message(self, value):
23             if str(value).isnumeric():
24                 raise Exception('The Message can\'t be cryptic')
25             else:
26                 self.__message = value
27         #---Methods---#
28
29     objCan1 = CanOnAString('There is no spoon')
30     print('original message: {}'.format(objCan1.message))
31     objCan1.message = 'Live Long and prosper'
32     print('updated message: {}'.format(objCan1.message))
33
34     print('Now Let\'s check the Exception:')
35     try:
36         objCan1.message = 123456
37     except Exception as e:
38         print(e)
39
40     print('Now Let\'s test direct access!')
41     try:
42         print(objCan1.__message)
43     except Exception as e:
44         print(e, e.__doc__, sep='\n\n')
45
46     print('Now Let\'s try assigning to the hidden attribute directly:')
47     objCan1.__message = 'Sometimes your whole life boils down to one insane move'
48     print('this should not be done, but it appears Python ignores it anyhow (without throwing an Exception):')
49     print('last message: {}'.format(objCan1.message))
50     try:
51         print(objCan1.__message)
52     except Exception as e:
53         print(e, e.__doc__, sep='\n\n')
54

```

Listing 6 - classes 5

```

In [8]: runfile('C:/_FDProgramming/Mod08/Can_05.py', wdir='C:/_FDProgramming/Mod08')
original message: There Is No Spoon
updated message: Live Long And Prosper
Now let's check the Exception:
The Message can't be cryptic
Now let's test direct access!
'CanOnAString' object has no attribute '__message'

Attribute not found.
Now let's try assigning to the hidden attribute directly:
this should not be done, but it appears Python ignores it anyhow (without throwing an Exception):
last message: Live Long And Prosper
Sometimes your whole life boils down to one insane move

```

Figure 7 - classes 5

Lines 8 to 27 define the class. Lines 29 and on test / use the class.

Line 29 creates an object with the assigned message.

Line 30 getter returns the message with title formatting (every first letter in a word is capitalized)

Line 31 setter updates the message.

Line 32 getter returns the updated message

Line 36 setter creates an Exception with custom error message

Line 42 trying to read the attribute directly yields an Exception as the Attribute is not found.

Line 47 assigning a value directly to an attribute does not yield an error message.

Line 49 shows that the update was not written to the attribute message.

Line 51 however, now yields the updated message.

What happened on the part line 46 and following is due to the “violation” of the convention to not access hidden or private attributes or methods in Python. The results we see are because of how Python organizes its objects. Line 47 does not access the hidden attribute, but in fact creates an additional component in the object that has nothing to do with the hidden attribute. That is why there is no error message and also why we can access it afterwards in line 51.

## LAB 08-D: Working with properties

In this Lab, we'll add properties to the class we created in Lab 08-C.

- Make a copy of Lab08\_C.py and save it as Lab08\_D.py
- Add code to create properties (setters and getters) for all three attributes.
- Add code to verify the validity of the values.
- Test the class and write down how the code works.

## Methods

Methods are like functions in a script: They allow you organize your statements into blocks that can be invoked by calling the method's name. The one difference is that a method call also submits a reference to the object it's invoked on, so the first attribute supplied to a method is the “self” reference.

```

1  #-----#
2  # Title: Can_06.py
3  # Desc: CanOnAString -- Demonstrator class 6
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     #---Fields---#
11     #---Constructor---#
12     def __init__(self, msg):
13         #---Attributes---#
14         self.__message = msg
15
16     #---Properties---#
17     @property
18     def message(self):
19         return self.__message.title()
20
21     @message.setter
22     def message(self, value):
23         if str(value).isnumeric():
24             raise Exception('The Message can\'t be cryptic')
25         else:
26             self.__message = value
27
28     #---Methods---#
29     def noAnswer(self):
30         return 'I know Kung Fu'
31
32     objCan1 = CanOnAString('There is no spoon')
33     print(objCan1.noAnswer())
34

```

Listing 7 - classes 6

```

In [10]: runfile('C:/_FDProgramming/Mod08/Can_06.py', wdir='C:/_FDProgramming/Mod08')
I know Kung Fu

```

Figure 8 - classes 6

The `__str__()` method:

In most languages, most classes include a method that returns some or all of the objects data as string. The build in method in Python is called `__str__()`. In many other languages it's called something like ToString().

In Python, if you do not define your own `__str__()` method, your class will implicitly inherit the method from the class object. This python object method returns the name of the class and an address identifier. You might have seen this already in this course. However, you can override this method with your own method to return something more useful, like the class's attributes or our answering machine response.

```

1  #-----#
2  # Title: Can_07.py
3  # Desc: CanOnAString -- Demonstrator class
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class DemoCan():
9      ...pass
10
11 class CanOnAString():
12
13     ...# --- Fields --- #
14     ...# --- Constructor --- #
15     def __init__(self, msg):
16         ...# --- Attributes --- #
17         ...self.__message = msg
18
19     ...# --- Properties --- #
20     ...@property
21     def message(self):
22         ...return self.__message.title()
23
24     ...@message.setter
25     def message(self, value):
26         ...if str(value).isnumeric():
27             ...raise Exception('The Message can\'t be cryptic')
28         ...else:
29             ...self.__message = value
30
31     ...# --- Methods --- #
32     def noAnswer(self):
33         ...return 'I know Kung Fu'
34
35     def __str__(self):
36         ...return self.noAnswer()
37
38 objCan1 = CanOnAString('There is no spoon')
39 objCan2 = DemoCan()
40
41 print(objCan2)
42 s = str(objCan1)
43 print(s)
44 print(objCan1)
45 print(objCan1.__str__())
46

```

Listing 8 - classes 7

```

In [18]: runfile('C:/_FDProgramming/Mod08/Can_07.py', wdir='C:/_FDProgramming/Mod08')
<__main__.DemoCan object at 0x000001A6EFC5D08>
I know Kung Fu
I know Kung Fu
I know Kung Fu

```

Figure 9 - classes 7

**Note:** We will explore inheritance more in module 09.

### LAB 08-E: Working with Methods:

In this Lab, we'll add methods to the class we created in Lab 08-D.



- Make a copy of Lab08\_D.py and save it as Lab08\_E.py
- Add code to create a `__str__` method to return a formatted content of the attributes.
- Add code to verify the proper functioning of the method.
- Test the class and write down how the code works.

## Static Methods

There are instances where you want methods to be called on class level and not on instance level. One example could be to keep track how many cans we have on our string:

```

1  #-----#
2  # Title: Can_08.py
3  # Desc: CanOnAString - Demonstrator class 8
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     #---Fields---#
11     numCans = 0
12     #---Constructor---#
13     def __init__(self, msg):
14         #---Attributes---#
15         self.__message = msg
16         CanOnAString.numCans += 1
17
18     #---Properties---#
19     @property
20     def message(self):
21         return self.__message.title()
22
23     @message.setter
24     def message(self, value):
25         if str(value).isnumeric():
26             raise Exception('The Message can\'t be cryptic')
27         else:
28             self.__message = value
29
30     #---Methods---#
31     @staticmethod
32     def connected():
33         return '\nThere are {} cans connected to the string.'.format(CanOnAString.numCans)
34
35     def noAnswer(self):
36         return 'I know Kung Fu'
37
38     def __str__(self):
39         return self.noAnswer()
40
41     print('Connecting some cans')
42     objCan1 = CanOnAString('There is no spoon')
43     objCan2 = CanOnAString('Live Long and prosper')
44     objCan3 = CanOnAString('Sometimes your whole life boils down to one insane move')
45
46     print(CanOnAString.connected())
47
48     print('Accessing the class field thru an object: {}'.format(objCan1.numCans))
49

```

Listing 9 - classes 8

```
In [21]: runfile('C:/_FDProgramming/Mod08/Can_08.py', wdir='C:/_FDProgramming/Mod08')
Connecting some cans

There are 3 cans connected to the string.
Accessing the class field thru an object: 3
```

Figure 10 - classes 8

It is easier and more practical to use class fields and static methods to keep track of information that concerns the class. Otherwise, on every change to this information, a synchronization over all the object instances of this class would need to happen.

To indicate a static method, the decorator `@staticmethod` is used. Also, as these methods don't run on the object level, no reference to the calling object is passed in. (No 'self' as first attribute).

Classes can have both: instance methods and static methods. Generally, when a class focuses on processing data, static methods will be used. If a class focuses on storing data, and hence instantiation makes sense, instance methods will be used. This is not a universal rule, but more a general guideline. In the end, what is needed / makes sense will be programmed.

#### Private Methods:

As with attributes, it's possible to have methods for internal processing only. These private methods are indicated by two leading underscores. These can be useful to create a layer of abstraction or for maintaining fields.

In older languages, it was critical to keep book on objects, mostly to dis-allocate memory when an object or the application was closed. (Otherwise these objects would stay active in memory and block this capacity. The only way to clear these remnants from memory was (and is) to reboot the computer.

Python self-cleans objects that are no longer referenced to avoid this. Let's develop our counter example anyhow, as it is a useful concept for certain problems (and might be necessary in other languages).

**Note:** We will not utilize this feature in our Labs or assignments.

Lines 31-33 define a public static method.

Lines 35-37 define a private static method.

As our field `numCans` is now a private field, we can only access it via a public method to read it out, no longer direct.



```

1  #-----#
2  # Title: Can_09.py
3  # Desc: CanOnAString -- Demonstrator class
4  # Change Log: (Who, When, What)
5  # DBiesinger, 2030-Jan-01, Created File
6  #-----#
7
8  class CanOnAString():
9
10     ...# --- Fields ---#
11     ...__numCans = 0
12     ...# --- Constructor ---#
13     def __init__(self, msg):
14         ...# --- Attributes ---#
15         ...self.__message = msg
16         ...CanOnAString.__incrementCount()
17
18     ...# --- Properties ---#
19     ...@property
20     def message(self):
21         ...return self.__message.title()
22
23     ...@message.setter
24     def message(self, value):
25         ...if str(value).isnumeric():
26             ...raise Exception('The Message can\'t be cryptic')
27         ...else:
28             ...self.__message = value
29
30     ...# --- Methods ---#
31     ...@staticmethod
32     def connected():
33         ...return '\nThere are {} cans connected to the string.'.format(CanOnAString.__numCans)
34
35     ...@staticmethod
36     def __incrementCount():
37         ...CanOnAString.__numCans += 1
38
39     def noAnswer(self):
40         ...return 'I know Kung Fu'
41
42     def __str__(self):
43         ...return self.noAnswer()
44
45     print('Connecting some cans')
46     objCan1 = CanOnAString('There is no spoon')
47     objCan2 = CanOnAString('Live long and prosper')
48     objCan3 = CanOnAString('Sometimes your whole life boils down to one insane move')
49
50     print(CanOnAString.connected())
51

```

Listing 10

```

In [23]: runfile('C:/_FDProgramming/Mod08/Can_09.py', wdir='C:/_FDProgramming/Mod08')
Connecting some cans

There are 3 cans connected to the string.

```

Figure 11 - classes 9

## Decorators

We have seen multiple instances now where decorators are changing the behavior or scope of functions. In simple terms, decorators wrap some functionality around a function (or method). It is possible to write your own wrappers (or decorators). But these are more advanced concepts than what we'll cover in this course.

## Type hints

Python version 3.5 and up supports type hints. These show what type of data is expected in attributes as well as what type is returned.

**Note:** In many languages these type definitions are necessary and enforced. In Python, these are not enforced and have a more “informal” character. It helps to inform other developers what is expected.

Type hints and return types are announced using the following syntax:

```
def echo(self, value: str) -> str:
    """echoes back the message and makes it loud"""
    return value.upper()
```

Listing 11 - type hints

Examining this function in spyder help (CTRL + I) yields:

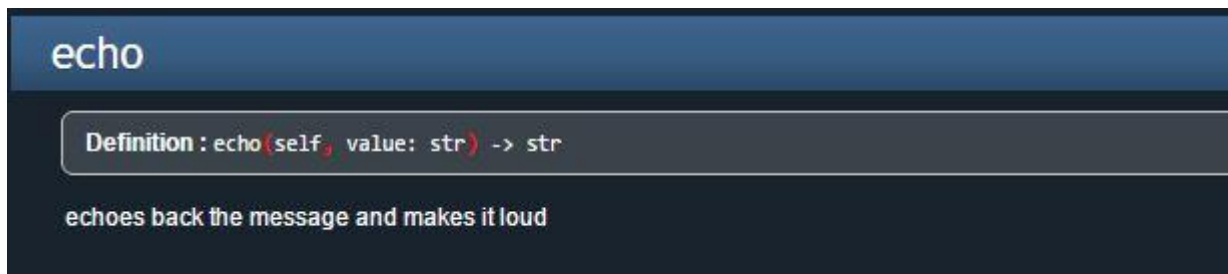


Figure 12 - type hints

## Doc Strings

Same as with functions, we should include docstrings in our classes, properties and methods.

```
class CanOnAString():
    """Cans On A String Demo Class
    ...
    Attributes:
        msg: a string containing the last message send from this can
    ...
    """
```

Listing 12 - doc string class

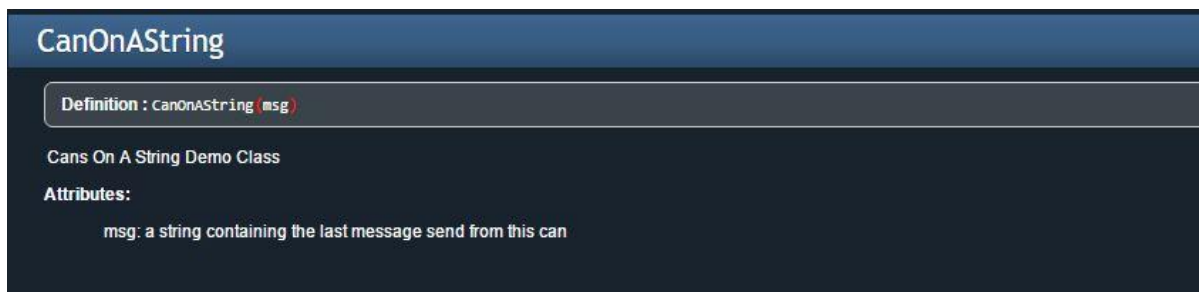


Figure 13 - doc string class

## Using classes

Processing classes are typically used thru their methods. Data classes are mostly used implicitly thru object instances.

You have used many in this manner: Lists, tuples, dictionaries, strings to name some.

You can use your own classes / objects in the same manner as rows of data. But, as you define these rows, you are now in control directly how the data behaves, and what types of data can be set into which attribute! You can of course store your data rows based on your custom class in a list and iterate over it.

## Summary

In this Module, we started exploring Object Oriented Programming. We created classes to define objects, wrote methods and create attributes for objects, instantiated objects (from classes) and restricted access to an object's attributes.

We used spyder as our IDE.

At this point you should be able to answer the following questions from memory. For the ones you can't, please review the subject. It might help to imagine being asked these questions by a co-worker or in an interview setting.

- What is the difference between a class and the objects made from a class?
- What are the components that make up the standard pattern of a class?
- What is the purpose of a class constructor?
- When do you use the keyword "self?"
- When do you use the keyword "@staticmethod?"
- How are fields and attributes and property functions related?
- What is the difference between a property and a method?
- Why do you include a docstring in a class?

When you can answer all of these from memory, it's time to complete the assignment and move to the next module.