Abstract Data Structures and Objects Part 2

Week 3: Computer Programming for Data Scientists (7CCSMCMP)

10 October 2016

Classes

Classes are the cornerstone of object-orient programming languages

As Data Scientists, being able to *define* and *use* classes and object-oriented programming, will help you:

- Dive into and understand the packages and libraries that you use
- Help you create your own re-usable code and libraries
- Useful in organizing your code when you start creating your own Data Models

Classes are composed of

- **data elements**
  - *variables* or *attributes*
- **code elements**
  - *methods* or *functions* that perform actions on the data elements
    - like variables, functions have a type, a name, and a value
  - *constructors*

Classes are *hierarchical* - i..e, they are *extended* from other classes

Top of the class hierarchy is a *built-in class*, i.e., part of the programming language

for example, Java has `Object`, Python has the `object`

Classes - defining and instantiating objects

Classes are “blue prints” for creating *instances* of object

Example: A House

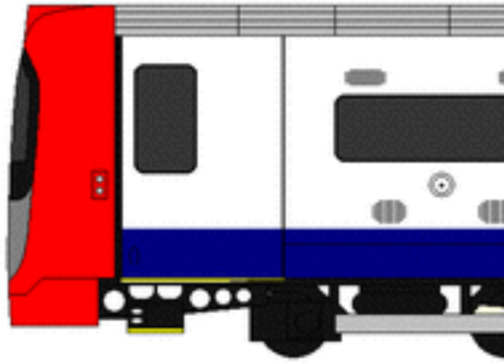
- *class* = architect’s blueprints
- *instance* = a house built following that blueprint

To instantiate = to build the house

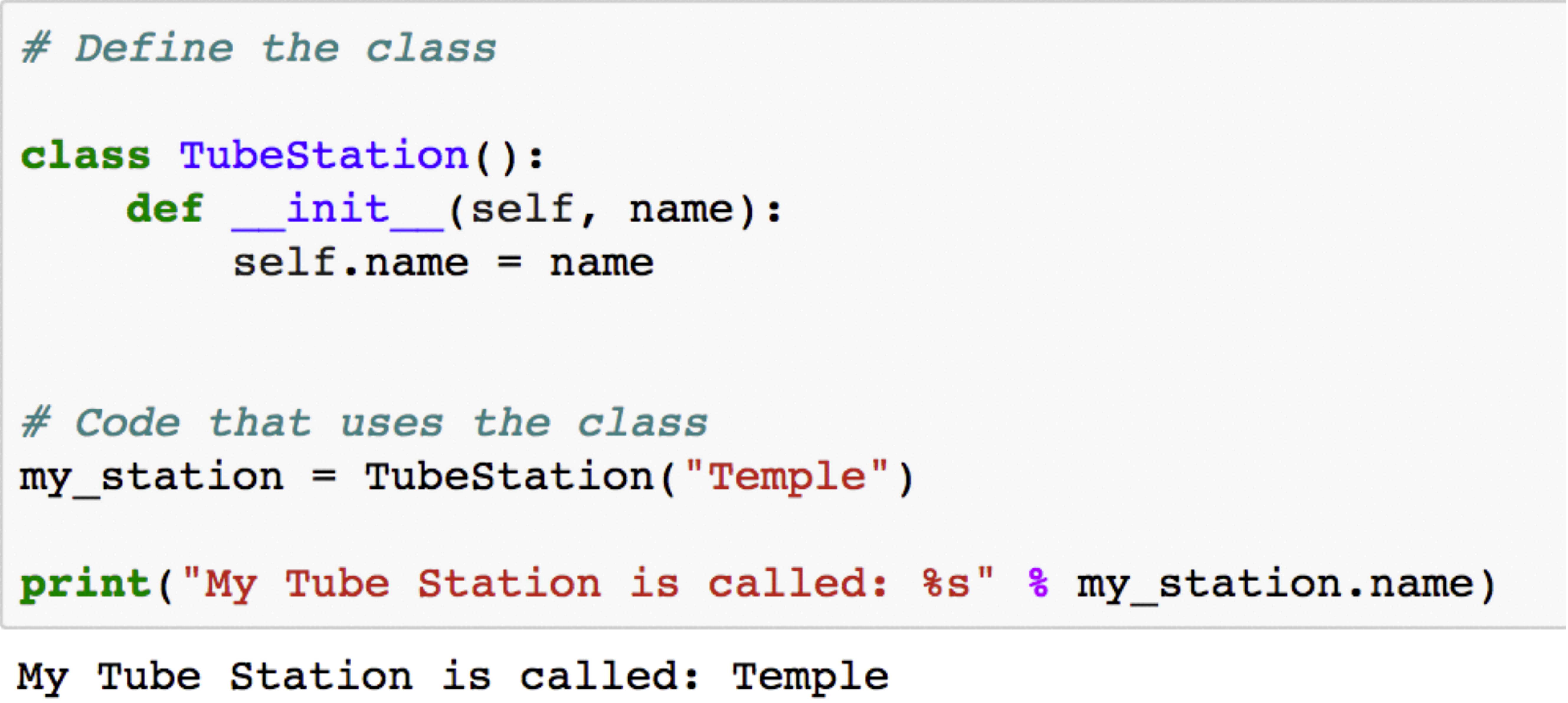
You can build **MANY** houses using the same blueprint, so you can instantiate **MANY** objects using the same class.

In order to use a class, you *instantiate* it by creating an *object* of that type.
This is like declaring a variable.

Use a special method that is defined in the class, called the *constructor* method.

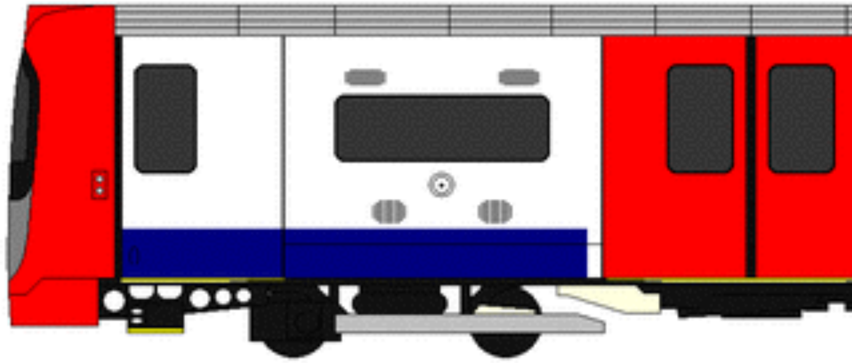


Classes - a simple example

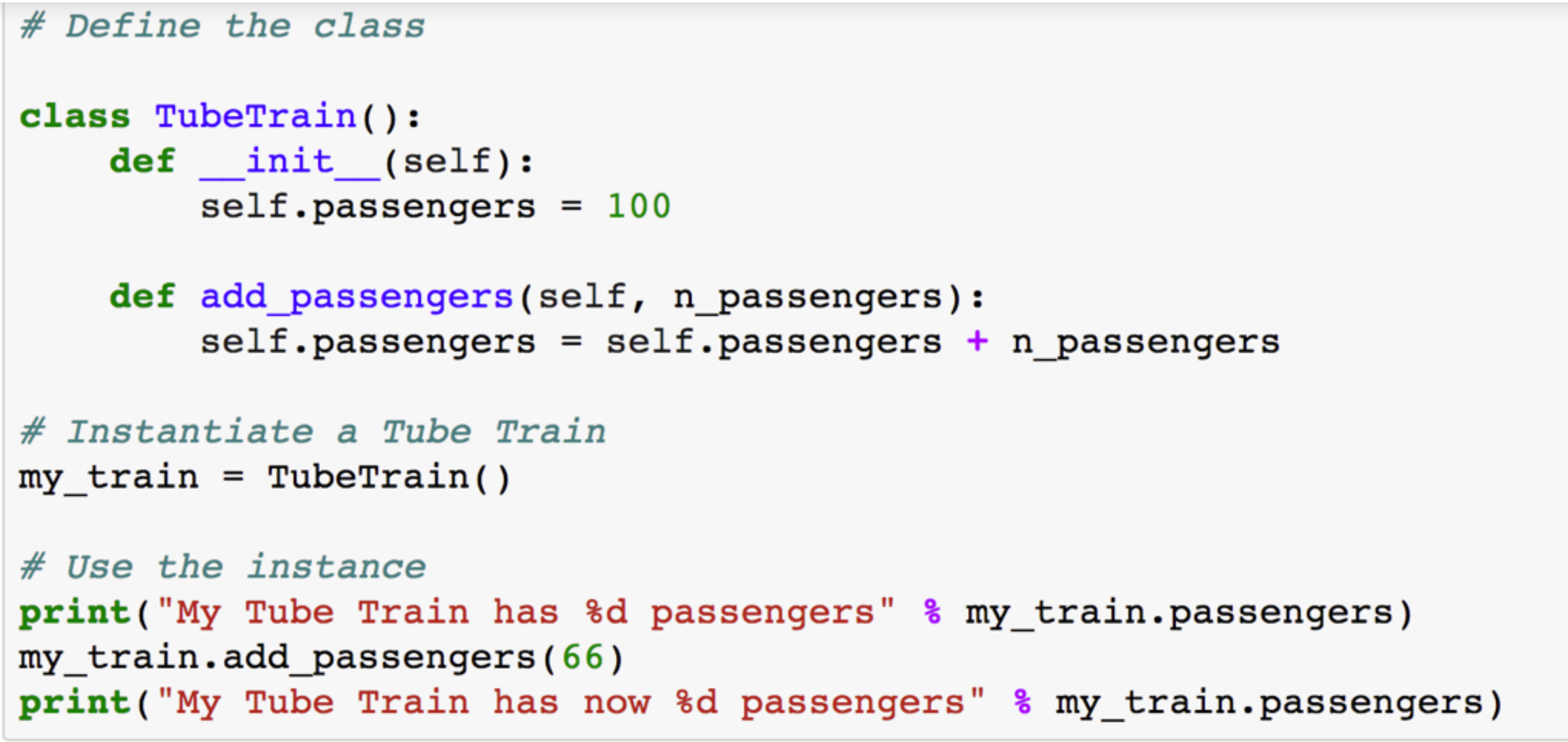


```python
# Define the class

class TubeStation():
    def __init__(self, name):
        self.name = name


# Code that uses the class
my_station = TubeStation("Temple")

print("My Tube Station is called: %s" % my_station.name)
```

```
My Tube Station is called: Temple
```

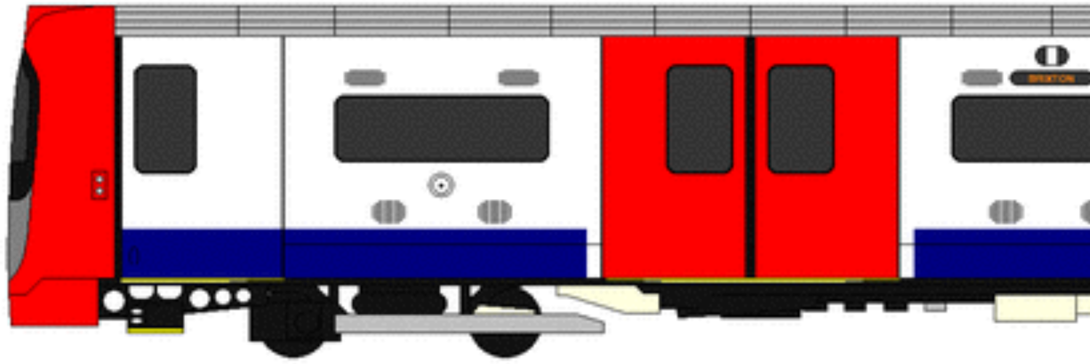


Classes - another example



```python
# Define the class

class TubeTrain():
    def __init__(self):
        self.passengers = 100

    def add_passengers(self, n_passengers):
        self.passengers = self.passengers + n_passengers

# Instantiate a Tube Train
my_train = TubeTrain()

# Use the instance
print("My Tube Train has %d passengers" % my_train.passengers)
my_train.add_passengers(66)
print("My Tube Train has now %d passengers" % my_train.passengers)
```

```
My Tube Train has 100 passengers
My Tube Train has now 166 passengers
```



Classes - dissecting the examples

`TubeStation` is the name of the class in the first example.

`TubeTrain` is the name of the class in the second example.

It extends the built-in default class type, which is `object`

- equivalent to writing:

```
class TubeStation(object):
```

or

```
class TubeTrain(object):
```

When new objects are instantiated, the special *constructor* method is called:

```
__init__
```

All methods requires at least one argument, `self`, which is a reference to the object itself:

```
def __init__(self):
```

Elements within the class are invoked using the object name, followed by the element's name, separated by a dot (.):

```
my_station.name
my_train.add_passengers(100)
```

Classes - the `__init__` constructor

The `__init__` constructor is a special method that is invoked automatically when an object is *instantiated*.

`__init__` does not explicitly return a value

(i.e. you would not call return when defining it)

`__init__` can have arguments other than `self`, like other class methods

Example:

```
    my_station = TubeStation(“Temple”)
```

or

```
    my_train = TubeTrain()
```

Classes - encapsulation

Encapsulation is another principle of object-oriented programming.

The general ideas are that:

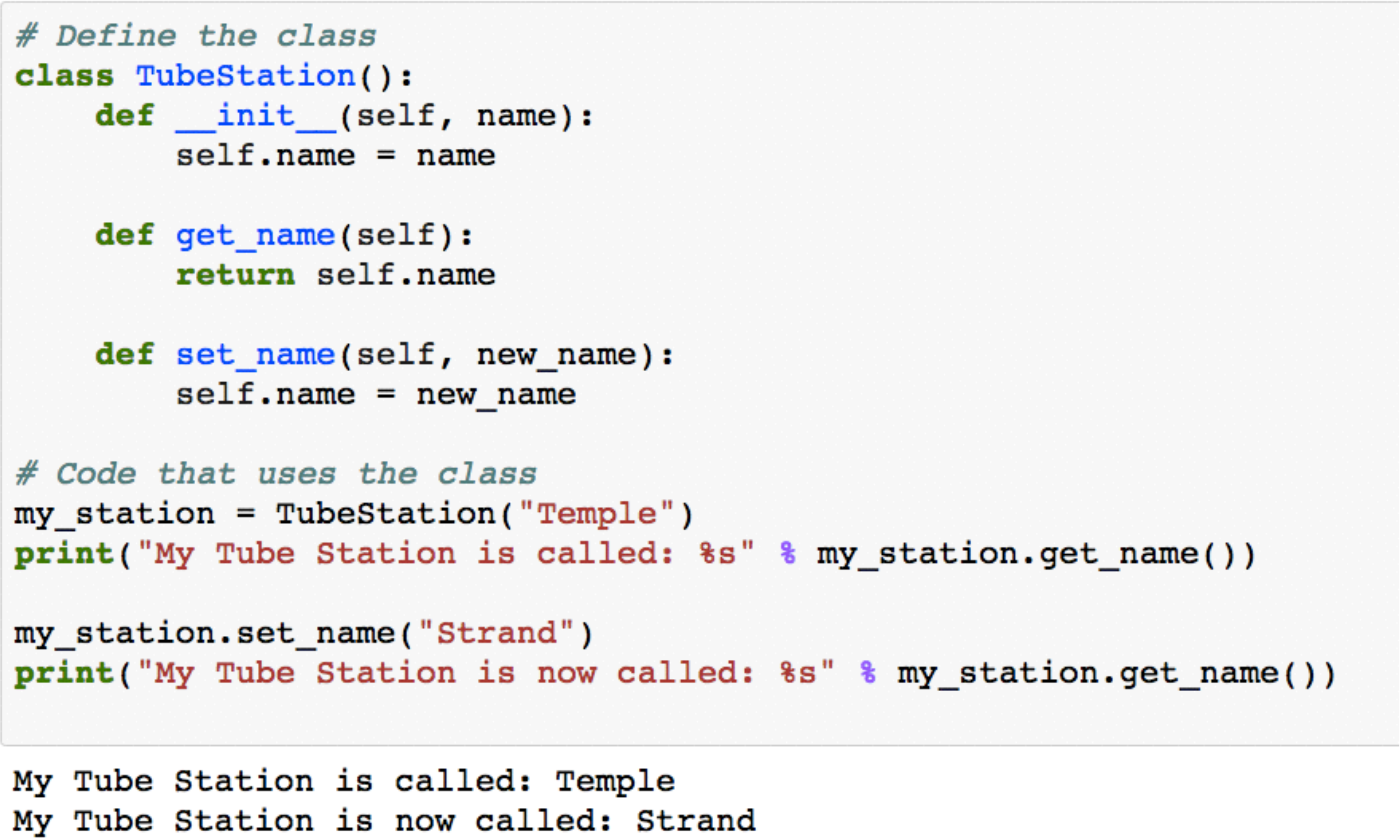
- objects should be self-contained and self-governing
- only methods that are part of an object should be able to change the object's data elements

Languages like C++ and Java have *private* attributes -

these attributes are *only* accessible by a classes methods and are not accessible from outside the class.

Classes - encapsulation

Some languages use *setter* and *getter* methods to “encapsulate” the object’s data

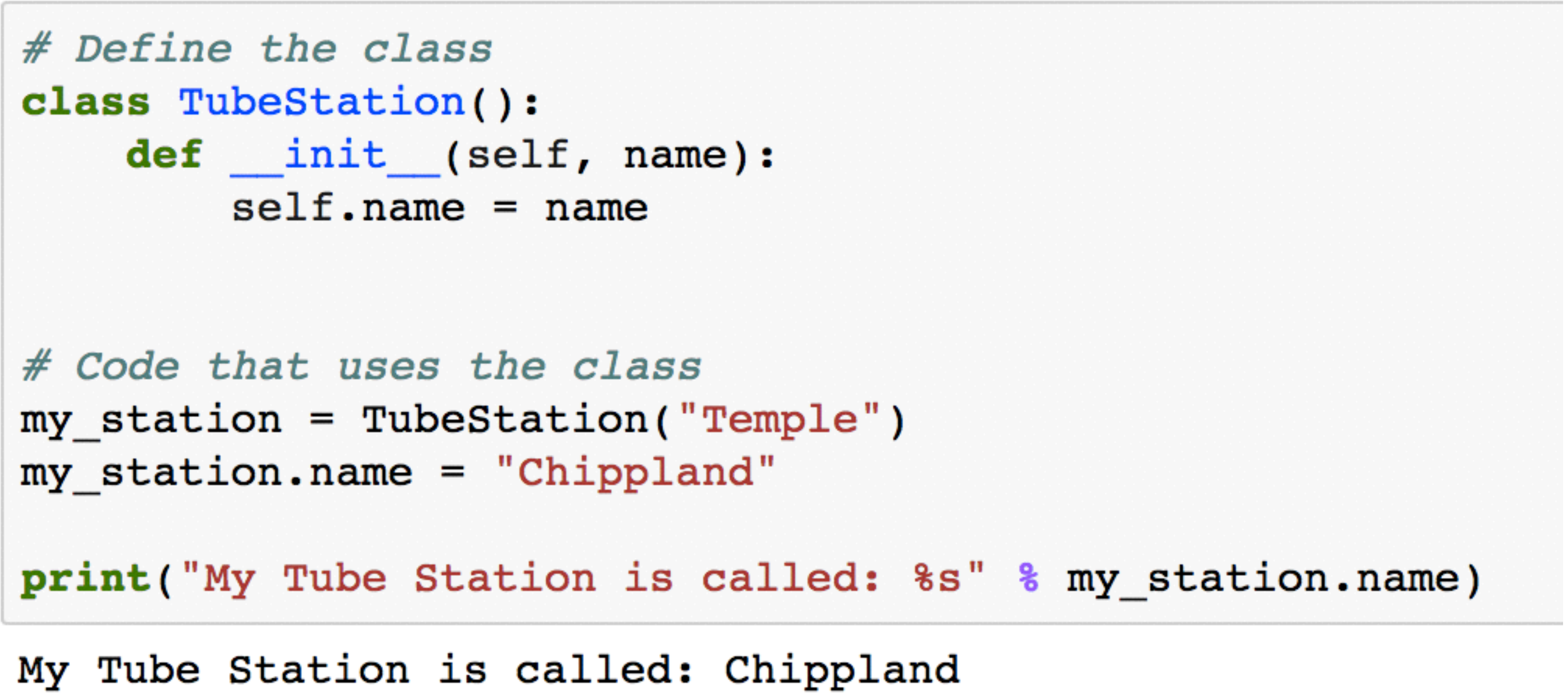


```
# Define the class
class TubeStation():
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

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

# Code that uses the class
my_station = TubeStation("Temple")
print("My Tube Station is called: %s" % my_station.get_name())

my_station.set_name("Strand")
print("My Tube Station is now called: %s" % my_station.get_name())
```

```
My Tube Station is called: Temple
My Tube Station is now called: Strand
```

Classes - encapsulation

In Python all attributes and methods are *public*, and are accessible from anywhere!

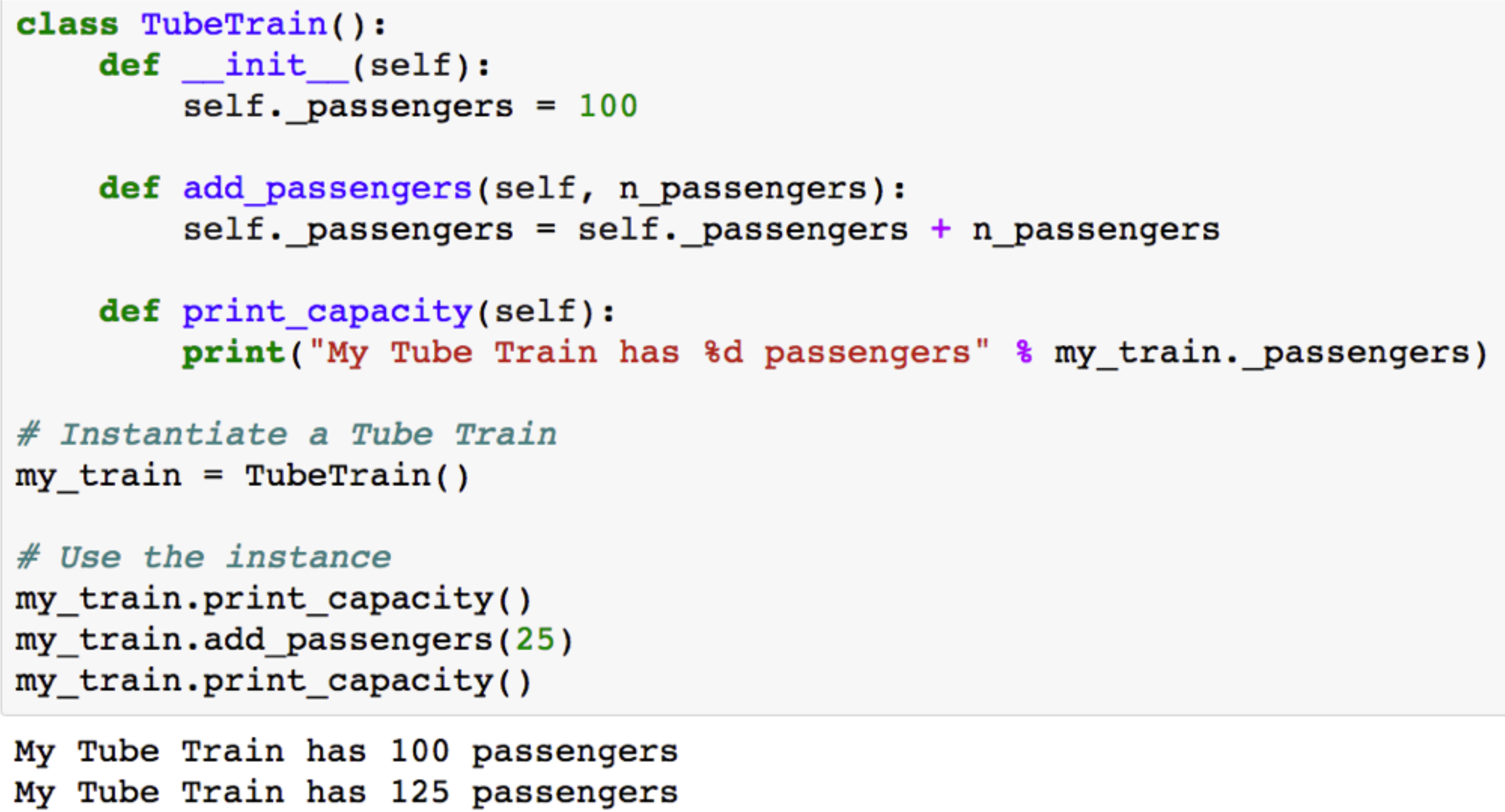


```python
# Define the class
class TubeStation():
    def __init__(self, name):
        self.name = name


# Code that uses the class
my_station = TubeStation("Temple")
my_station.name = "Chippland"

print("My Tube Station is called: %s" % my_station.name)
```

```
My Tube Station is called: Chippland
```

Classes - encapsulation

In Python, it is convention to add a single underscore ‘_’ in front of attributes and methods one would want to be considered *private*.



```
class TubeTrain():
    def __init__(self):
        self._passengers = 100

    def add_passengers(self, n_passengers):
        self._passengers = self._passengers + n_passengers

    def print_capacity(self):
        print("My Tube Train has %d passengers" % my_train._passengers)

# Instantiate a Tube Train
my_train = TubeTrain()

# Use the instance
my_train.print_capacity()
my_train.add_passengers(25)
my_train.print_capacity()
```

```
My Tube Train has 100 passengers
My Tube Train has 125 passengers
```

Classes - Inheritance

Inheritance is the means by which classes are created out of other classes.

Key concept in object-oriented programming.

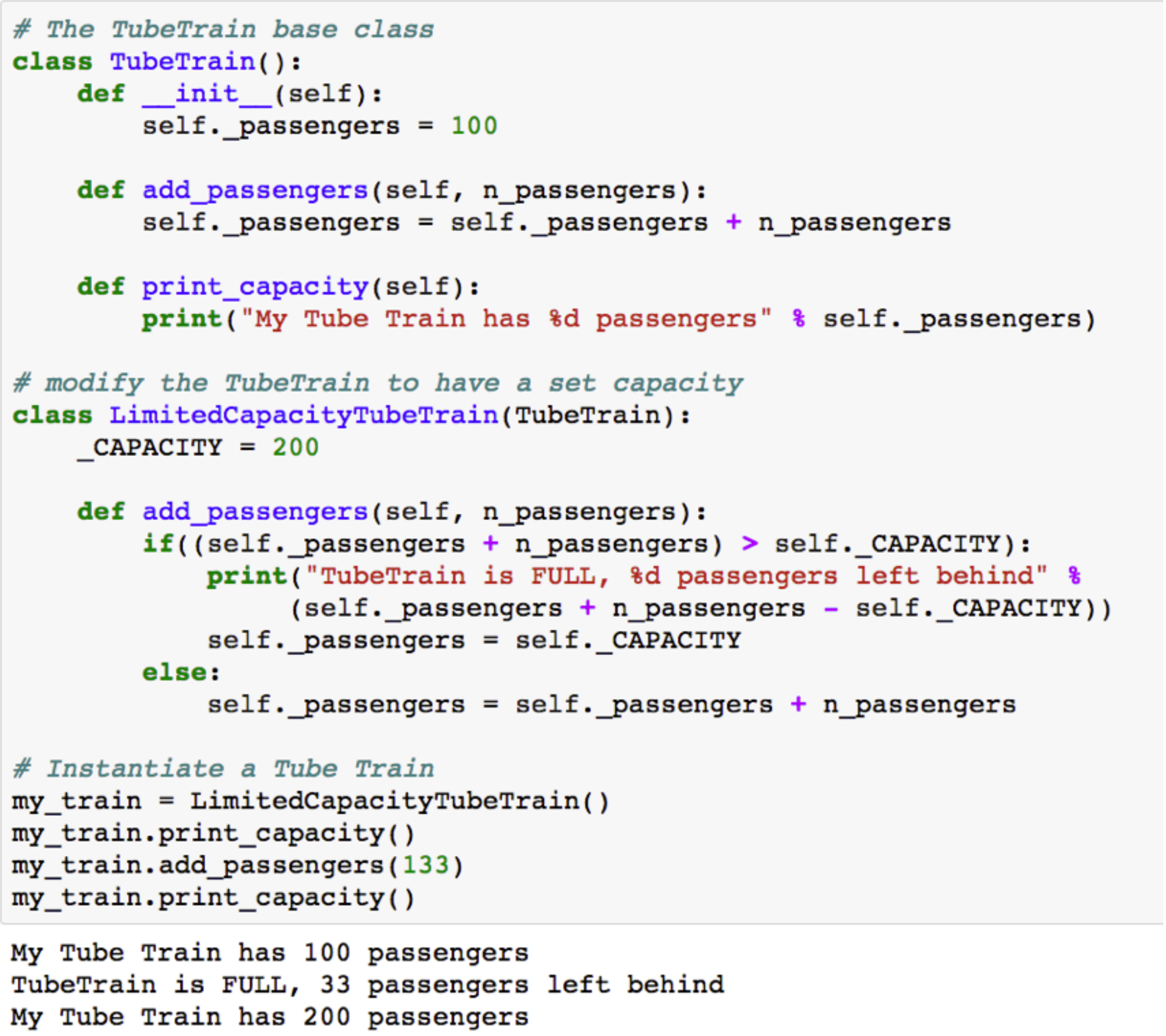
Idea is to *re-use* code from one class to create another, or *extend* another.

You can modify a class's *attributes* and/or *methods*

You might want to *change* the attributes and still be able to use the *same* methods.
/or/
You might want the change how the methods behave on the same attributes.

parent class

child class



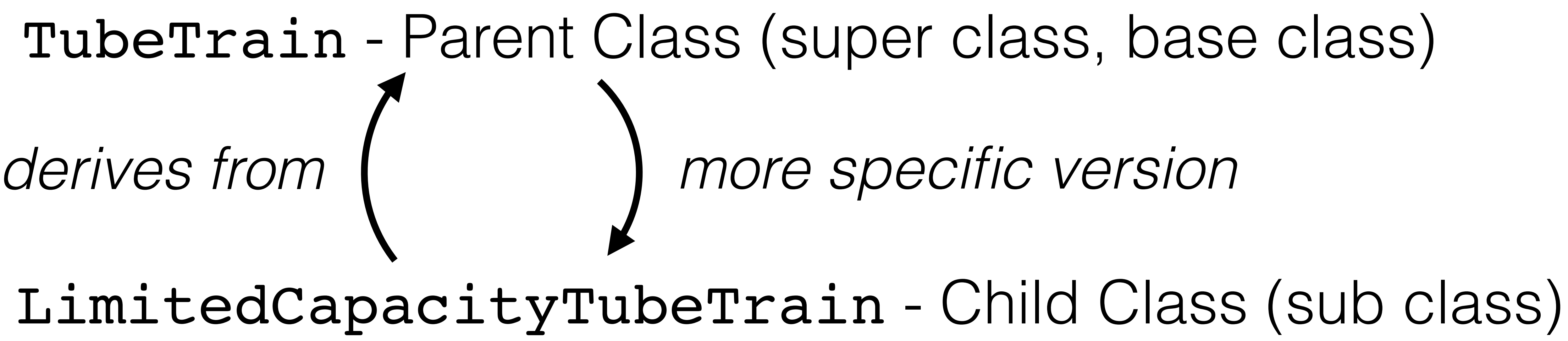
```python
# The TubeTrain base class
class TubeTrain():
    def __init__(self):
        self._passengers = 100

    def add_passengers(self, n_passengers):
        self._passengers = self._passengers + n_passengers

    def print_capacity(self):
        print("My Tube Train has %d passengers" % self._passengers)

# modify the TubeTrain to have a set capacity
class LimitedCapacityTubeTrain(TubeTrain):
    _CAPACITY = 200

    def add_passengers(self, n_passengers):
        if((self._passengers + n_passengers) > self._CAPACITY):
            print("TubeTrain is FULL, %d passengers left behind" %
                 (self._passengers + n_passengers - self._CAPACITY))
            self._passengers = self._CAPACITY
        else:
            self._passengers = self._passengers + n_passengers

# Instantiate a Tube Train
my_train = LimitedCapacityTubeTrain()
my_train.print_capacity()
my_train.add_passengers(133)
my_train.print_capacity()
```

```
My Tube Train has 100 passengers
TubeTrain is FULL, 33 passengers left behind
My Tube Train has 200 passengers
```

Classes - Inheritance Tree and Overriding Methods

TubeTrain is the *root* of the inheritance tree

Classes *derived* from other classes are called *children* or *subclasses* of the class they are derived from, which is also referred to as their *parent* class.



`TubeTrain` - Parent Class (super class, base class)

derives from

more specific version

`LimitedCapacityTubeTrain` - Child Class (sub class)

This is known as the ***is-a*** relationship between a *subclass* and its *parent* class.

When you *extend* or *derive a class*, you override methods defined in the parent class by defining them again in the child and giving the child version different behavior (i.e. `add_passengers`).

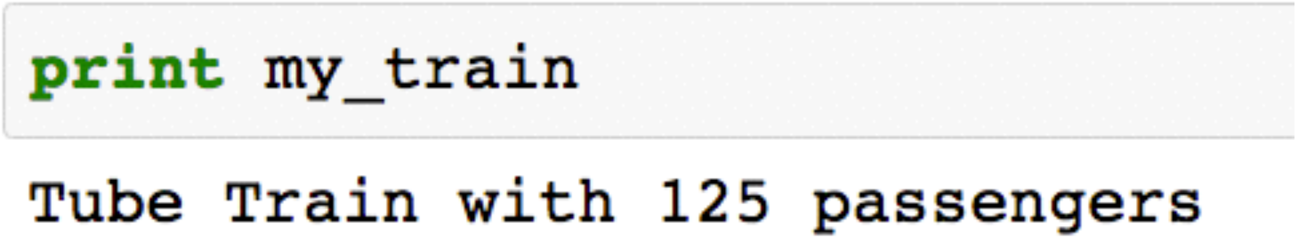
** The version of any method that is invoked is the definition closest to that object's class UP the inheritance tree.*

Classes - Overriding Python’s `object` methods

Can modify how your class will behave when being called by Python’s built-in methods.

Convention for Python’s special methods is that method’s name is surrounded by *double underscores* (i.e. `__<method>__`).

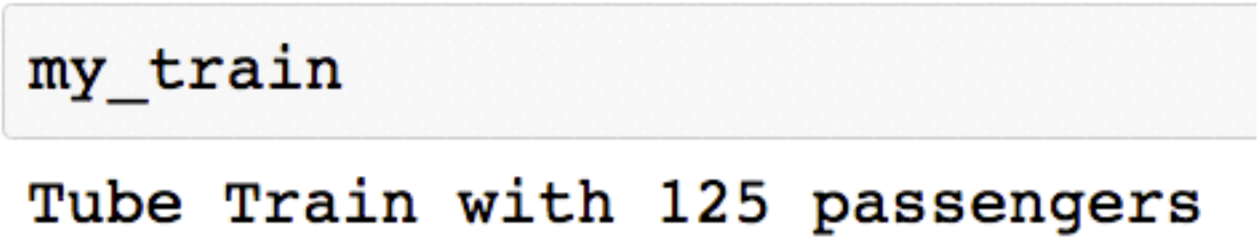
`__str__(self)` - override to change what is printed when covered to a string with `str(self)`or when your object is passed into the `print()` function



```
print my_train
```

```
Tube Train with 125 passengers
```

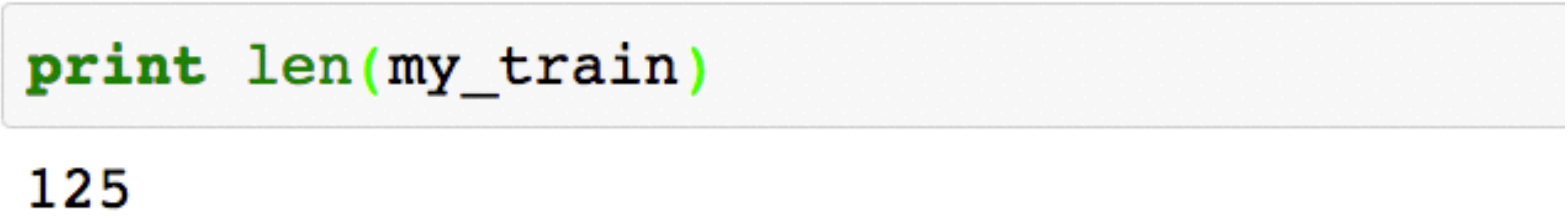
`__repr__(self)` - override it’s representation with `repr(self)` (i.e. when evaluated in Python interpreter)



```
my_train
```

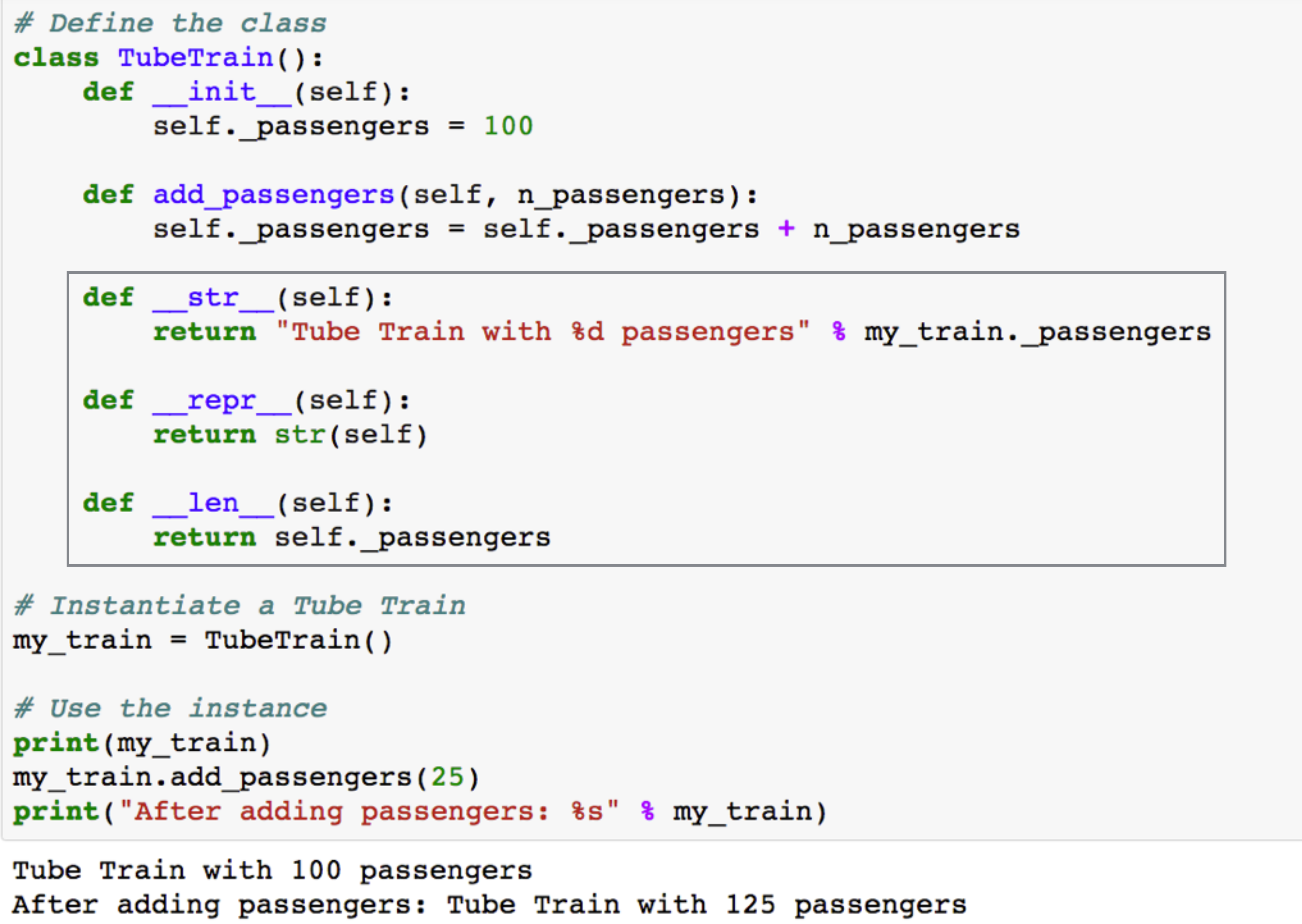
```
Tube Train with 125 passengers
```

`__len__(self)` - override what is returned when object is called with `len(self)`



```
print len(my_train)
```

```
125
```



```python
# Define the class
class TubeTrain():
    def __init__(self):
        self._passengers = 100

    def add_passengers(self, n_passengers):
        self._passengers = self._passengers + n_passengers

    def __str__(self):
        return "Tube Train with %d passengers" % my_train._passengers

    def __repr__(self):
        return str(self)

    def __len__(self):
        return self._passengers

# Instantiate a Tube Train
my_train = TubeTrain()

# Use the instance
print(my_train)
my_train.add_passengers(25)
print("After adding passengers: %s" % my_train)
```

```
Tube Train with 100 passengers
After adding passengers: Tube Train with 125 passengers
```

Classes - Other special methods - operator overloading

Define how objects can be compared with *comparison* operators

Operator Method to Override	**Operator Use**
`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`
`__lt__(self, other)`	`self < other`
`__gt__(self, other)`	`self > other`
`__le__(self, other)`	`self <= other`
`__ge__(self, other)`	`self >= other`

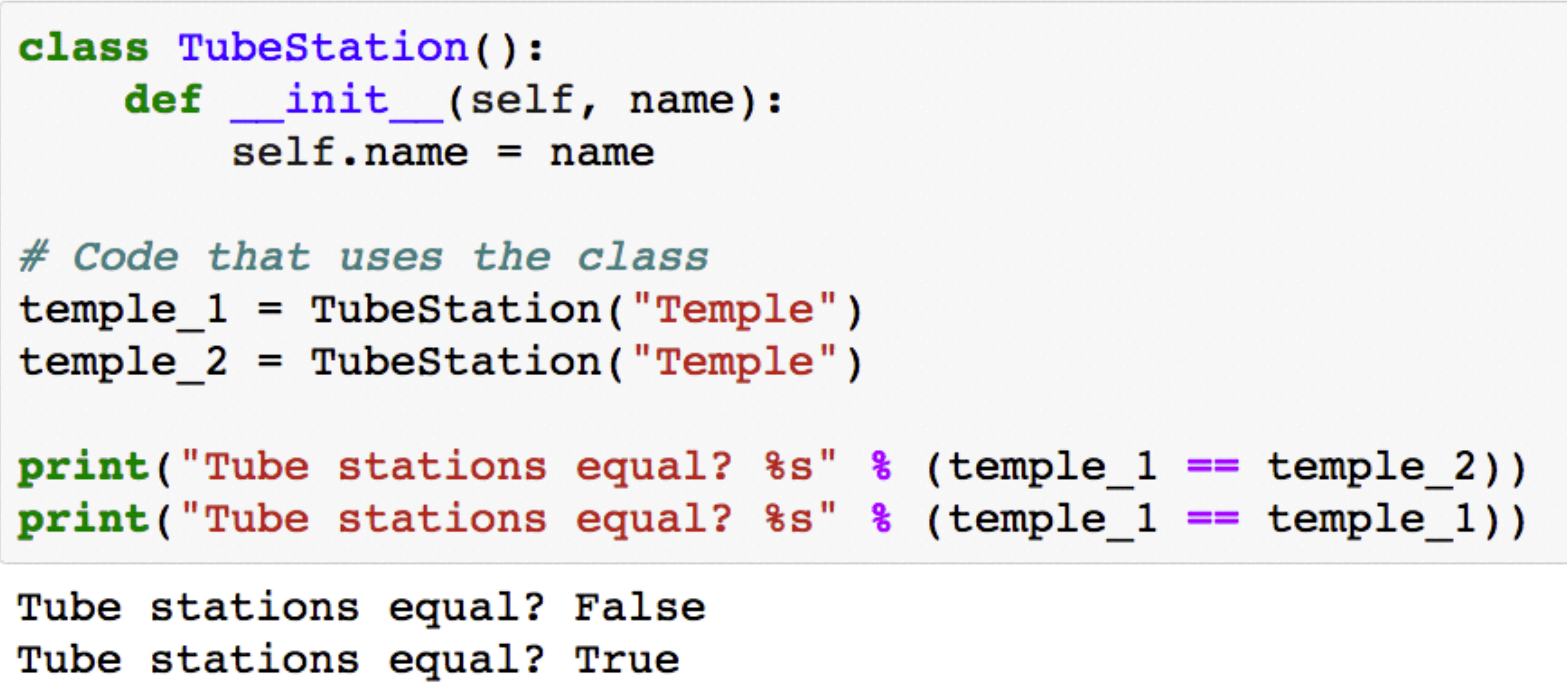
Define how objects respond when used with *mathematical* operators

Operator Method to Override	**Operator Use**
`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__mul__(self, other)`	`self * other`
`__truediv__(self, other)`	`self / other`
`__mod__(self, other)`	`self % other`
`__pow__(self, other)`	`self ** other`

Classes - Comparing objects with special method __eq__

Comparing objects are tricky!

Using a == comparison by default will only tell you if you are comparing the same *reference* to an object.

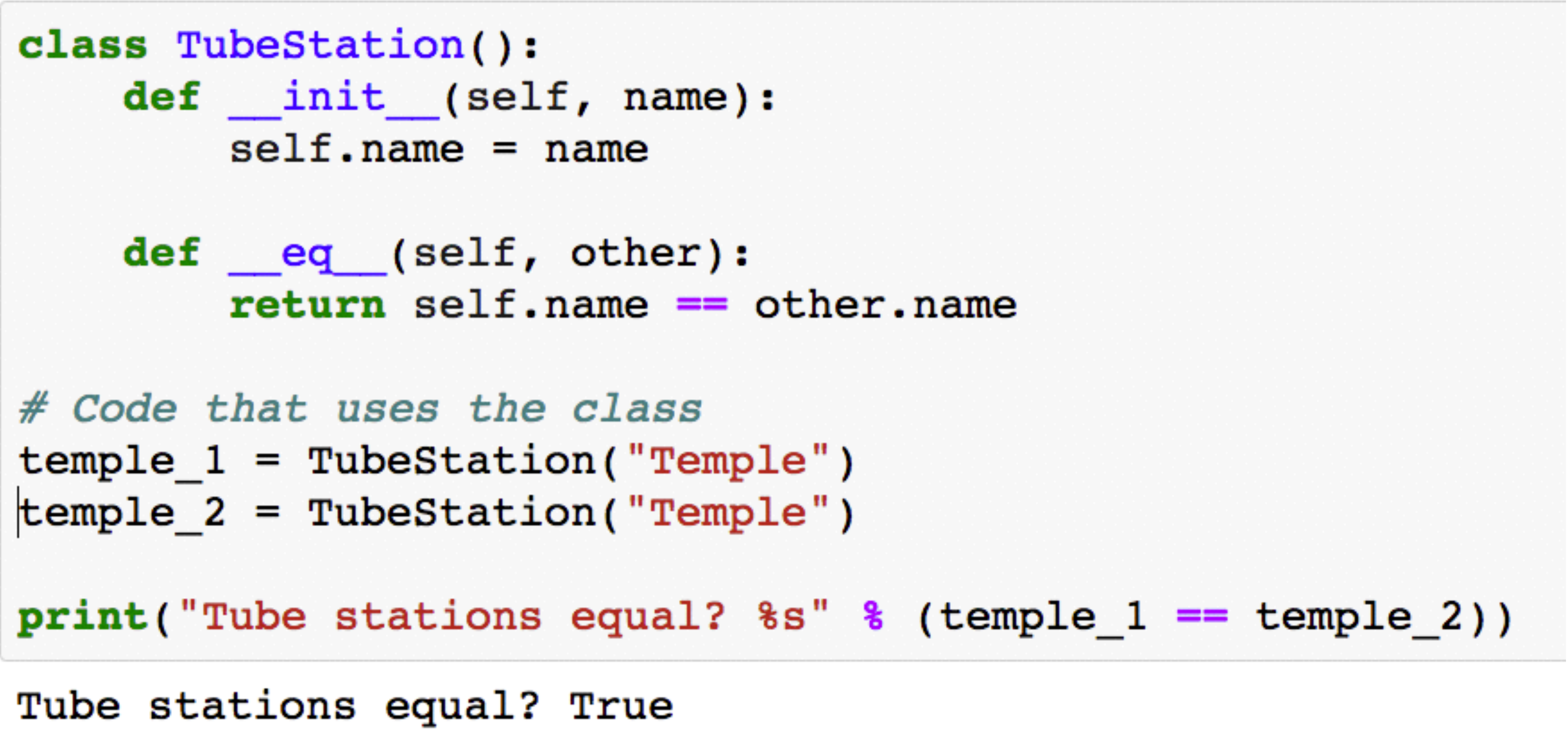


```
class TubeStation():
    def __init__(self, name):
        self.name = name

# Code that uses the class
temple_1 = TubeStation("Temple")
temple_2 = TubeStation("Temple")

print("Tube stations equal? %s" % (temple_1 == temple_2))
print("Tube stations equal? %s" % (temple_1 == temple_1))
```

```
Tube stations equal? False
Tube stations equal? True
```

Classes - Comparing objects with special method __eq__

Overriding the `__eq__` method allows you to define how two different objects can be considered equal.

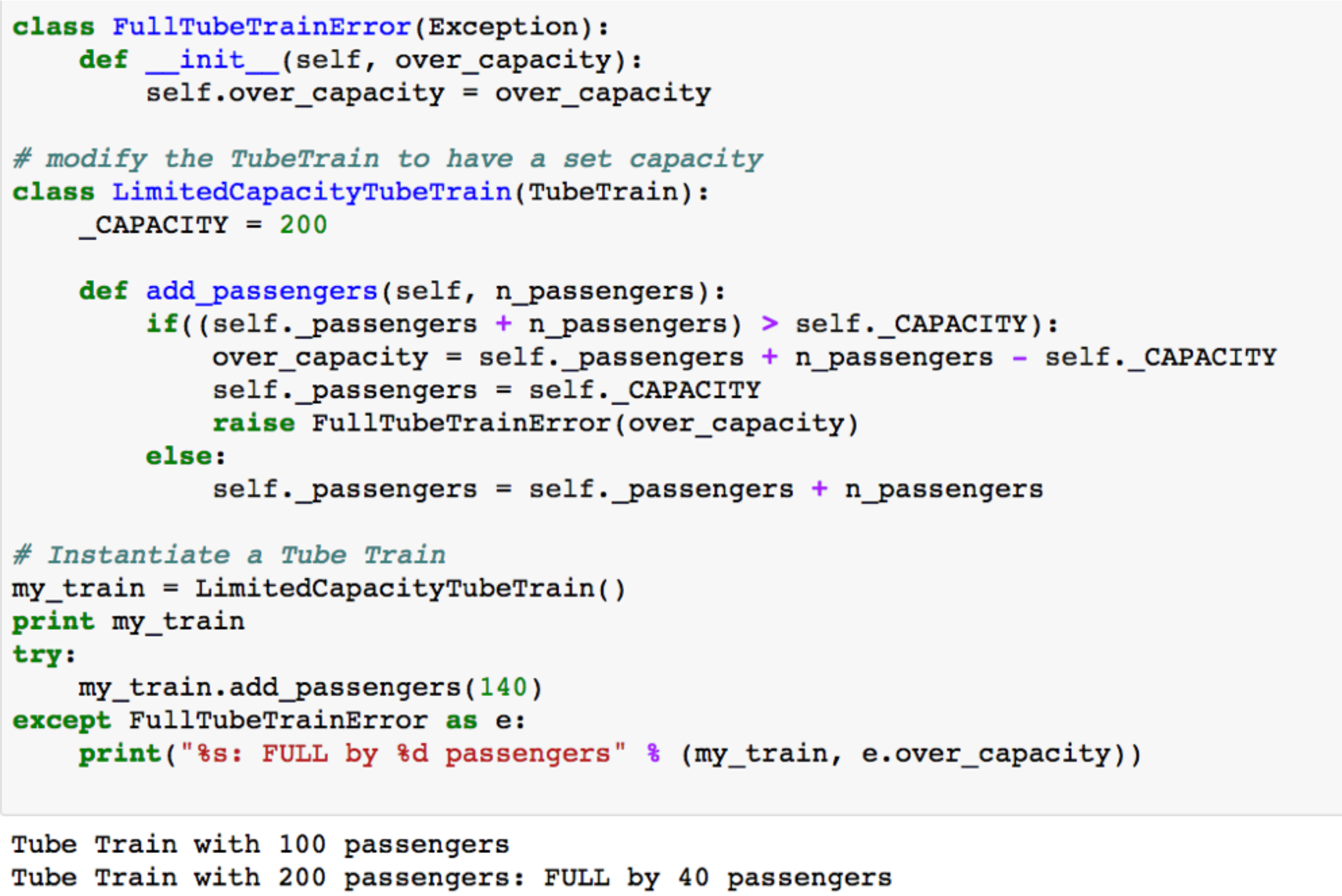


```
class TubeStation():
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

# Code that uses the class
temple_1 = TubeStation("Temple")
temple_2 = TubeStation("Temple")

print("Tube stations equal? %s" % (temple_1 == temple_2))
```

```
Tube stations equal? True
```

Exceptions - raising your own Exceptions

You can create your own Python Exceptions by extending the `Exception` class.



```
class FullTubeTrainError(Exception):
    def __init__(self, over_capacity):
        self.over_capacity = over_capacity

# modify the TubeTrain to have a set capacity
class LimitedCapacityTubeTrain(TubeTrain):
    _CAPACITY = 200

    def add_passengers(self, n_passengers):
        if((self._passengers + n_passengers) > self._CAPACITY):
            over_capacity = self._passengers + n_passengers - self._CAPACITY
            self._passengers = self._CAPACITY
            raise FullTubeTrainError(over_capacity)
        else:
            self._passengers = self._passengers + n_passengers

# Instantiate a Tube Train
my_train = LimitedCapacityTubeTrain()
print my_train
try:
    my_train.add_passengers(140)
except FullTubeTrainError as e:
    print("%s: FULL by %d passengers" % (my_train, e.over_capacity))
```

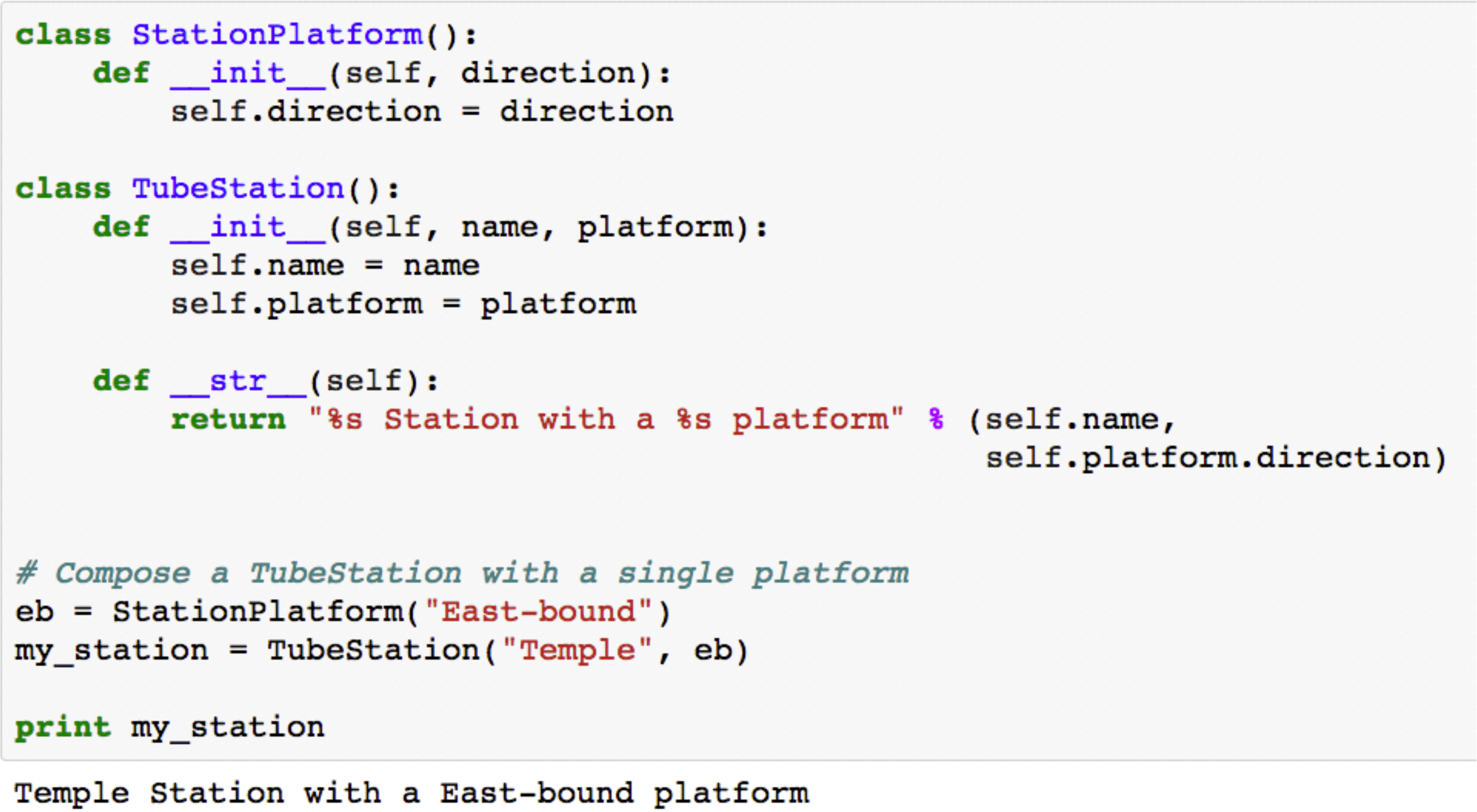
```
Tube Train with 100 passengers
Tube Train with 200 passengers: FULL by 40 passengers
```

Composition

Inheritance implements the *is-a* principle.

Composition implements the *has-a* principle.

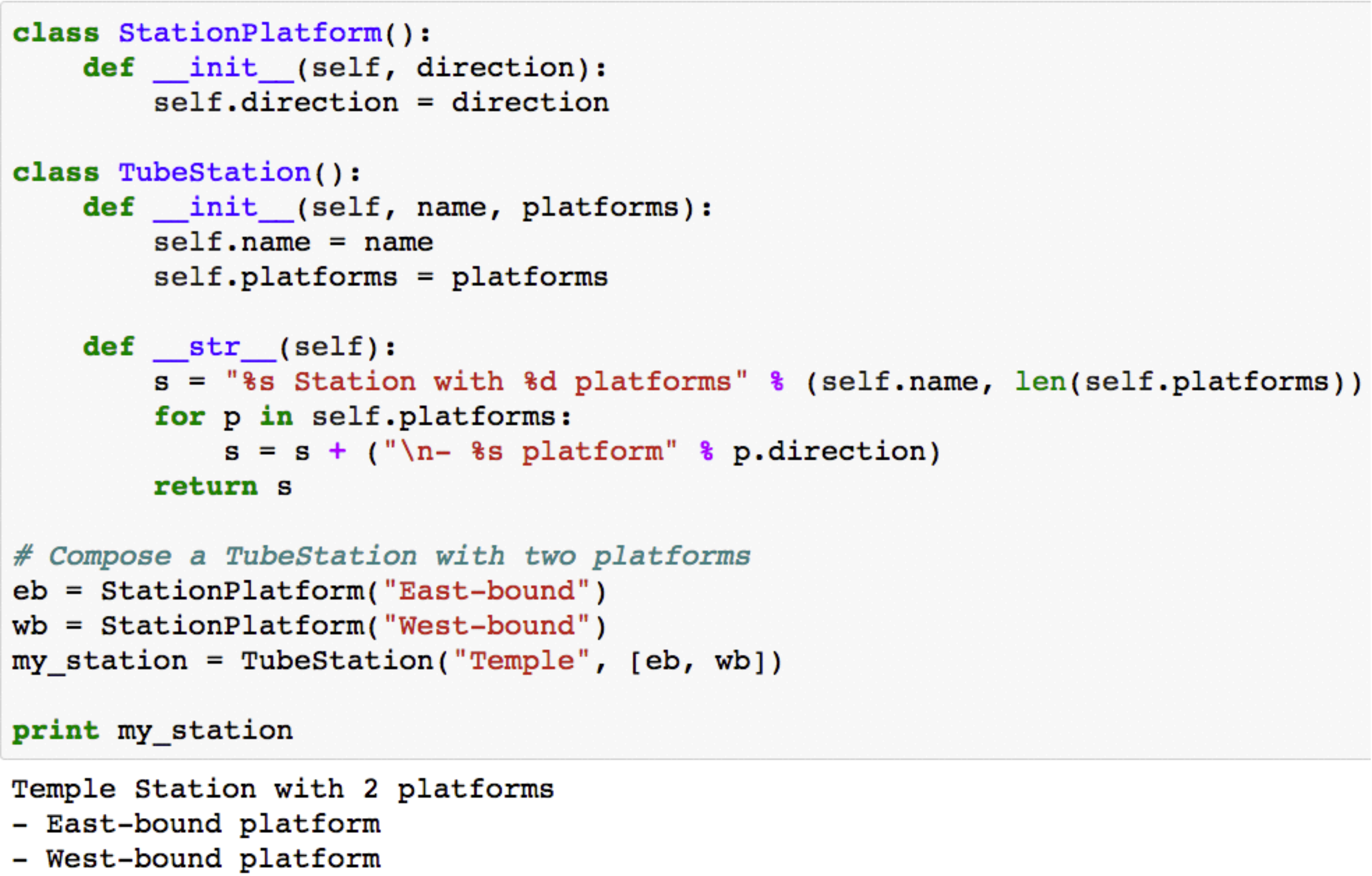
This refers to situations where objects have *attributes* that are other objects.



```
class StationPlatform():
    def __init__(self, direction):
        self.direction = direction

class TubeStation():
    def __init__(self, name, platform):
        self.name = name
        self.platform = platform

    def __str__(self):
        return "%s Station with a %s platform" % (self.name,
                                                  self.platform.direction)


# Compose a TubeStation with a single platform
eb = StationPlatform("East-bound")
my_station = TubeStation("Temple", eb)

print my_station
```

```
Temple Station with a East-bound platform
```

Composition - *more advanced example*



```python
class StationPlatform():
    def __init__(self, direction):
        self.direction = direction

class TubeStation():
    def __init__(self, name, platforms):
        self.name = name
        self.platforms = platforms

    def __str__(self):
        s = "%s Station with %d platforms" % (self.name, len(self.platforms))
        for p in self.platforms:
            s = s + ("\n- %s platform" % p.direction)
        return s

# Compose a TubeStation with two platforms
eb = StationPlatform("East-bound")
wb = StationPlatform("West-bound")
my_station = TubeStation("Temple", [eb, wb])

print my_station
```

```
Temple Station with 2 platforms
- East-bound platform
- West-bound platform
```

Pickling - Python object serialization

"*Pickling*" is the process whereby a Python object hierarchy is converted into a byte stream, and "*unpickling*" is the inverse operation, whereby a byte stream is converted back into an object hierarchy.

Pickling (and unpickling) is alternatively known as
"serialization", "marshalling," or "flattening".

But, **Pickling is Python-only compatible** - not useful as an ***interchange*** format!

<u>What can be pickled:</u>

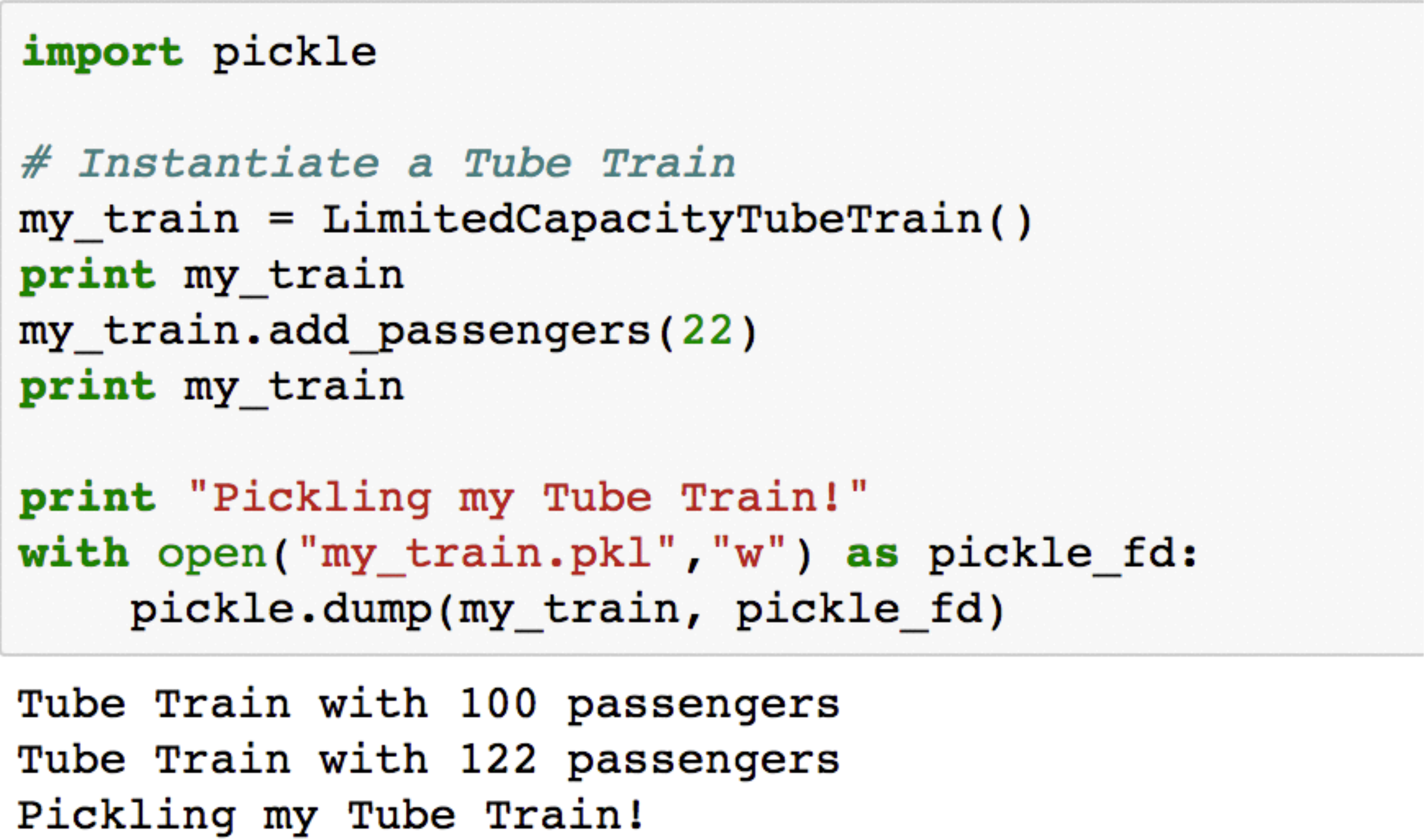
- `None`, `True`, and `False`
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module

Docs: https://docs.python.org/2/library/pickle.html

Pickling - Saving a Pickle file (`.pkl`)

Using Python **`pickle`** to *serialize* Python object structure.

```
pickle.dump(<objects>, <file descriptor)
```



```
import pickle

# Instantiate a Tube Train
my_train = LimitedCapacityTubeTrain()
print my_train
my_train.add_passengers(22)
print my_train

print "Pickling my Tube Train!"
with open("my_train.pkl","w") as pickle_fd:
    pickle.dump(my_train, pickle_fd)
```

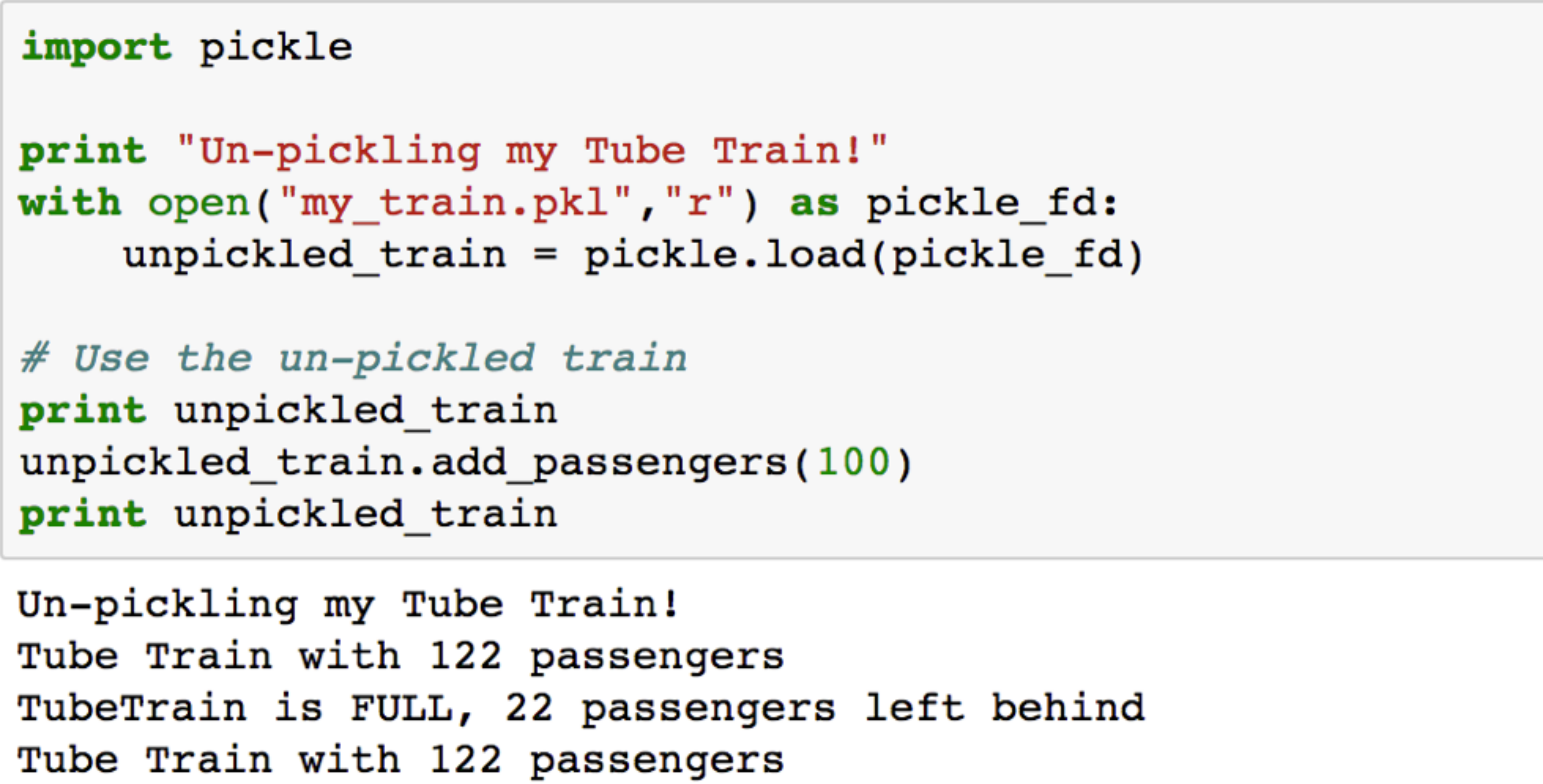
```
Tube Train with 100 passengers
Tube Train with 122 passengers
Pickling my Tube Train!
```

Pickling - Retrieving a Pickle file (`.pkl`)

Using Python **`pickle`** to *un-serialize* Python object structure.

```
<unpicked object> = pickle.load(<file descriptor>)
```

Class *Definition* is **NOT** pickled, need to include Class Definition in any new script.



```
import pickle

print "Un-pickling my Tube Train!"
with open("my_train.pkl","r") as pickle_fd:
    unpickled_train = pickle.load(pickle_fd)

# Use the un-pickled train
print unpickled_train
unpickled_train.add_passengers(100)
print unpickled_train
```

```
Un-pickling my Tube Train!
Tube Train with 122 passengers
TubeTrain is FULL, 22 passengers left behind
Tube Train with 122 passengers
```

Comparison of Data File-Handling Libraries

	`csv`	`xml`	`json`	`pickle`
type of data	tabular	hierarchical	hierarchical	object hierarchical
interchange compatibility	fair, depends on *dialect*	fair, depends	good	poor, only with your own python code
commonly used for	distribution of data/statistics	structured data, document file formats, HTML	web-based interchange, data API's	saving and retrieving work-in-progress
ease of import/ export python	good	ok, parse ElementTree	great, parse direct into primitives	super, parse direct into objects
performance	ok	can be slow to parse	fast	super fast

Summary

Classes - objected-oriented Python with class keyword

- Defining Classes and using the constructor `__init__`
- Inheritance - *is-a* relationship of objects
- Method Overrides
- Overriding Python's built-in object methods (i.e. `__str__`)
- Composition - *has-a* relationship of objects

Pickles - *Python-only* binary format to export objects