Chapter 28
Monkey Patching and Attribute Lookup

28.1 Introduction

Monkey Patching is a term you might well come across when looking into the Python further or when searching the web for Python related concepts. It relates to the ability in Python to extend the functionality associated with a class/type at runtime.

Although not directly related to Monkey Patching; how Python looks up attributes and how this process can be managed is a useful aspect to understand. In particular, how to handle unknown attributes can be very useful in managing situations in which Monkey Patching might be used to solve an initial attribute incompatibility.

This chapter explores both Monkey Patching and Python Attribute lookup.

28.2 What Is Monkey Patching?

Monkey Patching is the idea that it is possible to add behaviour to an existing object, at runtime, to meet some requirement that originally the type did not meet. This can happen for example as there is no fixed requirement for a class to implement all of a protocol; in many cases a class may only implement as much of a protocol as is required to meet the current needs; if at a later stage, other elements of a protocol are required then they can be added.

Of course, if this is likely to be a common occurrence then the features can be added to the class for use by everyone; but if not, then those features can be added dynamically at runtime to an object itself. This avoids the public interface of the type becoming cluttered with rarely used features/functionality.

© Springer Nature Switzerland AG 2019
J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_28

28.2.1 How Does Monkey Patching Work?

Python is a dynamic language which allows the definition of a type to change at runtime. As methods on objects are in essence just another attribute of a class, albeit one that can be executed, it is possible to add new functionality to a class by defining new attributes that will hold references to the new behaviour.

28.2.2 Monkey Patching Example

The following class, `Bag`, implements an initialisation method `__init__()`, `__str__()` and the `__getitem__()` method used to support indexed access to a container (or collection) type.

```
class Bag():
    def __init__(self):
        self.data = ['a', 'b', 'c']

    def __getitem__(self, pos):
        return self.data[pos]

    def __str__(self):
        return 'Bag(' + str(self.data) + ')'

b = Bag()
print(b)
```

This creates a `Bag` object and prints out the contents of the `Bag`:

```
Bag(['a', 'b', 'c'])
```

However, if we try to run

```
print(len(b))
```

We will get a runtime error:

```
Traceback (most recent call last):
  File "Bag.py", line 12, in <module>
    print(len(b))
TypeError: object of type 'Bag' has no len()
```

This is because the `len()` function expects the object passed to it will implement the `__len__()` method that is be used to obtain its length. In this case the class `Bag` does not implement this method.

However, we can define a stand-alone function that behaves in the way we would need the `Bag` to calculate its length, for example:

```
def get_length(self):
    return len(self.data)
```

This function takes a single parameter which we have called `self`. It then uses this parameter to reference an attribute called `data` which itself uses `len()` to return the length of the associated data items.

At present this function has no relationship to the `Bag` class other than the fact that it assumes that whatever is passed into it will have an attribute called `data`—which the `Bag` class does.

In fact, the `get_length()` function is compatible with any class that has an attribute `data` that can be used to determine its length.

We can now associate it with the `Bag` class; this can be done by assigning the function reference (in practice the function name) to an appropriate attribute on the `Bag` class. Since the `len()` function expects a class to implement the `__len__()` method we can assign the `get_length()` function to the `__len__()` attribute. This effectively adds a *new* method to the `Bag` class with the signature `__len__(self)`:

```
# Monkey patching
Bag.__len__ = get_length
```

Now when we invoke

```
print(len(b))
```

We get the value `3` being printed out.

We have now *Monkey Patched* the class `Bag` so that the missing method becomes available.

28.2.3 The Self Parameter

One of the reasons that Monkey Patching works is because all methods receive the special first parameter (called `self` by convention) representing the object itself. This means that any function that treats the first parameter as being a reference to an object can potentially be used to define a method on a class.

If a function does not assume that the first parameter is a reference to an object (the one holding the method) then it cannot be used to add new functionality to a class.

28.2.4 Adding New Data to a Class

Monkey patching is not just limited to functionality; it is also possible to add new data attributes to a class. For example, if we wanted each Bag to have a *name* then we could add a new attribute to the class to hold its name:

```
Bag.name = 'My Bag'
print(b.name)
```

Which prints out the string 'My Bag' which now acts as a default value of the name of any Bag. Once the attribute is added we can then change the name of this particular instance of a Bag:

For example, if we extend the above example:

```
Bag.name = 'My Bag'
print(b.name)

b.name = 'Johns Bag'
print(b.name)

b2 = Bag()
print(b2.name)
```

We can now generate:

```
My Bag
Johns Bag
My Bag
```

28.3 Attribute Lookup

As shown above, Python is very dynamic and it is easy to add attributes and methods to a class, but how does this work. It is worth considering how Python manages attribute and method lookup for an object.

Python classes can have both *class* and *instance* oriented attributes, for example the following class Student has a *class* attribute count (that is associated with the class itself) and an *instance* or *object* attribute name. Thus each instance of the class Student will have its own name attribute.

```
class Student:
    count = 0

    def __init__(self, name):
        self.name = name
        Student.count += 1
```

Whenever an instance of the class `Student` is created the `Student.count` attribute will be incremented b y `1`.

To manage these attributes Python maintains internal dictionaries; one for *class* attributes and one for *object* attributes. These dictionaries are called `__dict__` and can be accessed either from the class `<class>.__dict__` (for class attributes) or from an instance of the class `<instance.>__dict__` (for object attributes). For example:

```
student = Student('John')

# Class attribute dictionary
print('Student.__dict__:', Student.__dict__)
# Instance / Object dictionary
print('student.__dict__:', student.__dict__)
```

Which produces the output shown below (note that the class dictionary holds more information than just the class attribute count):

```
Student.__dict__: {'__module__': '__main__', 'count': 1,
                   '__init__': <function Student.__init__ at
                   0x10d515158>, '__dict__': <attribute
                   '__dict__' of 'Student' objects>,
                   '__weakref__': <attribute '__weakref__' of
                   'Student' objects>, '__doc__': None}
student.__dict__: {'name': 'John'}
```

To look up an attribute, Python does the following for class attributes:

1. Search the class Dictionary for an attribute
2. If the attribute is *not* found in step 1 then search the parent class(es) dictionaries

For object attributes, Python first searches the instance dictionary and repeats the above steps, it thus performs these steps:

1. Search the object/instance dictionary
2. If the attribute was *not* found in step 1, then search the class Dictionary for an attribute
3. If the attribute is *not* found in step 2, then search the parent class(es) dictionaries

Thus given the following statements, different steps are taken each time:

```
student = Student('John')

print('Student.count:', Student.count)  # class lookup
print('student.name:', student.name)  # instance lookup
print('student.count:', student.count)  # lookup finds class
attribute
```

The output as expected is that either attempt to access the class attribute `count` will result in the value 1 where as the object attribute `name` returns 'John'.

```
Student.count: 1
student.name: John
student.count: 1
```

As the dictionaries used to hold the class and object attributes are just that dictionaries, they provide another way to access the attributes of a class such as Student. That is you can write code that will access an attribute value using the appropriate `__dict__` rather than the more usual *dot* notation, for example the following are equivalent:

```
# class lookup
print('Student.count:', Student.count)
print("Student.__dict__['count']:", Student.__dict__['count'])

# Instance / Object Lookup
print('student.name:', student.name)
print("student.__dict__['name']:", student.__dict__['name'])
```

In both cases the end result is the same, either the class attribute `count` is accessed or the object/instance attribute `name` is accessed:

```
Student.count: 1
Student.__dict__['count']: 1
student.name: John
student.__dict__['name']: John
```

However, accessing attributes via the `__dict__` does not trigger a search process; it is instead a direct lookup in the associated dictionary container. Thus if you try to access a class variable via the objects `__dict__` then you will get an error. This is illustrated below where we attempt to access the `count` class variable via the *student* object:

```
# Attempt to look up class variable via object
print('student.name:', student.name)
print("student.__dict__['count']:", student.__dict__['count'])
```

This will generate a `KeyError` indicating that the object `__dict__` does not hold a key called 'count':

```
Traceback (most recent call last):
  File "Student.py", line 60, in <module>
    print("student.__dict__['count']:",
student.__dict__['count'])
KeyError: 'count'
```

28.4 Handling Unknown Attribute Access

Monkey patching is of course very flexible and very useful when you know what you need to provide; however what happens when an attribute reference (or method invocation) occurs when it is not expected? By default an error is generated such as the AttributeError below:

```
student = Student('John')

res1 = student.dummy_attribute
print('p.dummy_attribute:', res1)
```

This generates an AttributeError for the dummy_attribute

```
Traceback (most recent call last):
  File "Student.py", line 51, in <module>
    res1 = student.dummy_attribute
AttributeError: 'Student' object has no attribute
'dummy_attribute'
```

You can of course catch the `AttributeError` if you want; but the means wrapping your code in a `try` block. An alternative approach is to define a method called `__getattr__()`; this method will be called when an attribute is not found in the objects (and classes) `__dict__` dictionary. This method can then perform whatever action is appropriate such as logging a message or providing a default value etc.

For example, if we modify the definition of the `Student` class to include a `__getattr__()` method such that a default value is returned:

```
class Student:
    count = 0

    def __init__(self, name):
        self.name = name
        Student.count += 1

    # Method called if attribute is unknown

    def __getattr__(self, attribute):
        print('__getattr__: ', attribute)
        return 'default'
```

Now when we try and access the `dummy_attribute` on a student object we will get the string 'default' returned:

```
student = Student('John')

res1 = student.dummy_attribute
print('p.dummy_attribute:', res1)
```

Now generates:

```
__getattr__:  dummy_attribute
p.dummy_attribute: default
```

Note that the `__getattr__()` method is only called for unknown attributes as Python first looks in `__dict__` and thus if the attribute is found, no call is made to the `__getattr__()` method.

Also note that if an attribute is accessed directly from the `__dict__` (for example `student.__dict__['name']`) then the `__getattr__()` method is never invoked.

28.5 Handling Unknown Method Invocations

The `__getattr__()` method is also invoked if an unknown method is called. For example, if we call the method `dummy_method()` on a `Student` object then an error is raised (in fact this is again an `AttributeError`). However, if we define a `__getattr__()` method we can return a reference to a method to use as a default. For example, if we modify the `__getattr__()` method to return a method reference (i.e. the name of a method in the `Student` class):

```
class Student:
    count = 0

    def __init__(self, name):
        self.name = name
        Student.count += 1

    # Method called if attribute is unknown
    def __getattr__(self, attribute):
        print('__getattr__: ', attribute)
        return self.my_default

    def my_default(self):
        return 'default'
```

Now when a undefined method is invoked (such as `dummy-method()`) `__getattr__()` will be called. This method will return a reference to the method `my_default()`. This will be run and the value returned as a side effect of the method invocation as indicated by the round baskets (the '`()`') after the call to the original method:

```
student = Student('John')

res2 = student.dummy_method()
print('student.dummy_method():', res2)
```

Which produces the following output: rather than an error message

```
__getattr__:  dummy_method
student.dummy_method(): default
```

28.6 Intercepting Attribute Lookup

It is also possible to always intercept attribute lookups using the *dot* nation (e.g. `student.name`) by implementing the `__getattribute__()` method. This method will always be called instead of looking the attribute up in the objects dictionary. The `__getattr__()` method will only be called if an `AttributeError` is raised or the `__getattribute__()` method explicitly calls the `__getattr__()` method.

The `__getattribute__()` method should therefore return an attribute value (which may be a default value) or raise an `AttributeError` if appropriate.

It is important to avoid implementing code that will recursively call itself (for example calling `self.name` within `__getattribute__()` will result in a recursive call back to `__getattribute__()`!). To avoid this the implementation of the method should either access the `__dict__` directory or call the base classes `__getattribute__()` method.

An example is given below of a simple `__getattribute__()` method that logs the call to a method and then passes the invocation onto the base class implementation:

```
class Student:
    count = 0

    def __init__(self, name):
        self.name = name
        Student.count += 1

    # Method called if attribute is unknown
    def __getattr__(self, attribute):
        print('__getattr__: ', attribute)
        return 'default'

    # Method will always be called when an attribute
    # is accessed, will only called __getattr__ if it
    # does so explicitly or if an AttributeError is raised
    def __getattribute__(self, name):
        print('__getattribute__()', name)
        return object.__getattribute__(self, name)

    def my_default(self):
        return 'default'
```

We can use this version of the class with the following code snippet

```
student = Student('Katie')

print('student.name:', student.name)  # instance lookup

res1 = student.dummy_attribute # invoke missing attribute

print('student.dummy_attribute:', res1)
```

The output from this is now:

```
__getattribute__() name
student.name: Katie
__getattribute__() dummy_attribute
__getattr__:  dummy_attribute
student.dummy_attribute: default
```

As you can see from this the `__getattribute__()` method is called for both `student.name` and `student.dummy_attribute`. However the `__getattr__()` method is only called when the `dummy_attribute` is accessed.

Note that `__getattribute__()` is only invoked for attribute access and not for method invocation (unlike `__getattr__()`).

28.7 Intercepting Setting an Attribute

It is also possible to intercept object/instance attribute assignment when the *dot* notation (e.g. `student.name = 'Bob'`) is being used. This can be done by implementing the `__setattr__()` method. This method is invoked instead of the assignment.

The `__setattr()` method can perform any action required including storing the value supplied. However, to do this it should either insert the value directly into the objects dictionary (e.g. `student.__dict__['name'] = 'Bob'`) or preferably call the base class `__setattr__()` method, for example `object.__setattr__(self, name, value)` as shown below for the `Student` class:

```
class Student:
    count = 0

    def __init__(self, name):
        self.name = name
        Student.count += 1

    # Method will always be called when an attribute is set
    def __setattr__(self, name, value):
        print('__setattr__:', name, value)
        object.__setattr__(self, name, value)
```

If we now define the follow program that uses the `Student` class:

```
student = Student('John')

student.name = 'Bob'
print('student.name:', student.name)   # instance lookup
```

Running this could we generate the following output:

```
__setattr__: name John
__setattr__: name Bob
student.name: Bob
```

There are a few things to note about this output:

- Firstly the assignment of the Student's `name` within the `__init__()` method also invokes the `__setattr__()` method.
- Secondly, the assignment to the class variable `count` does not invoke the `__setattr__()` method.
- Thirdly the `student.name` assignment does of course invoke the `__setattr__()` method.

28.8 Online Resources

For further information on Monkey Patching the following resources may be of interest:

- https://en.wikipedia.org/wiki/Monkey_patch Wikipedia page on Monkey Patching.
- http://net-informations.com/python/iq/patching.htm A discussion on whether Monkey Patching should be considered good or bad practice.

28.9 Exercises

This exercises focusses on attribute look up.

You should add a method to the `Account` class that can be used to handle how accounts should behave when an attempt is made to access an undefined attribute.

In this case you should log the attempt to access the attribute (which means print out a warning message) then return a default value of -1.

For example, if you had the following line to your application:

```
print('acc1.branch:', acc1.branch)
```

Then this should invoke the `__getattr__()` method for the undefined attribute `branch`. Print a warning message and then return the value `-1` which will be printed by the above statement.

The output of this statement should be something like:

```
__getattr__: unknown attribute accessed -  branch
acc1.branch: -1
```