LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 8: MUTABLE OBJECTS AND MAGIC METHODS

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

8. MUTABLE OBJECTS AND MAGIC METHODS

BY FRED ANNEXSTEIN

CHAPTER 9 OBJECTIVES

By the end of this module, you will be able to...

1. Understand programming with objects with mutable data.
2. Understand operator overloading and pythons magic methods, including __repr__, __add__, and __call__ methods.
3. Apply python programming to the shallow and deep modification of mutable objects.

LINKED LIST CLASS

Recall that a linked list, introduced in chapter 6, is composed of a first element and the rest of the list. The rest of a linked list is itself a linked list — a recursive definition. The empty list is a special case of a linked list that has no first element or rest. A linked list is a sequence: it has a finite length and supports element selection by index.

We can now implement a class with the same structure and behavior. In this OOP version, we will define its behavior using special method names that allow our class to work with the built-in `len` function and element selection

operator (square brackets or `operator.getitem`) in Python. These built-in functions invoke special method names of a class: length is computed by `__len__` and element selection is computed by `__getitem__`. The empty linked list is represented by an empty tuple, which has length 0. Here is a class definition to be used to create linked list objects.

```
class Link:
   empty = ()

   def __init__(self, first, rest=empty):
      self.first = first
      self.rest = rest

   def __getitem__(self, i):
      if i == 0:
          return self.first
      else:
          return self.rest[i-1]

   def __len__(self):
      return 1 + len(self.rest)

>>> s = Link(30, Link(40, Link(50)))
>>> len(s)
3
>>> s[1]
40
```

The definitions of __len__ and __getitem__ are recursive. The built-in Python function len invokes a method called __len__ when applied to a user-defined object argument. Likewise, the element selection operator

invokes a method called __getitem__. Thus, bodies of these two methods will call themselves indirectly. For __len__, the base case is reached when self.rest evaluates to the empty tuple, Link.empty, which has a length of 0.

The built-in isinstance function returns whether the first argument has a type that is or inherits from the second argument. isinstance(rest, Link) is true if rest is a Link instance or an instance of some sub-class of Link.

```
>>> isinstance(s,Link)
True
```

The Link class has a closure property. Just as an element of a list can itself be a list, a Link can contain a Link as its first element.

```
>>> s_1 = Link(s, Link(60))
>>> s_1
Link(Link(30, Link(40, Link(50))), Link(60))
```

The s_1 linked list has only two elements, but its first element is a linked list with three elements.

```
>>> len(s_1)
2
>>> len(s_1[0])
3
>>> s_1[0][2]
50
```

REPRESENTING CLASSES WITH STRINGS

Our implementation is basically complete, but an instance of the Link class is currently difficult to inspect. To help with debugging, we can also define a function to convert a Link to a string expression.

In Python, there are a few methods that you can implement in your class definition to customize how built-in functions that return representations of your class behave.

There are two string method traditionally used. One is called “str” and another called “repr”. The difference is that “repr” generally returns a string that can be processed by the interpreter to re-produce an exact copy of the object.

```
def __str__(self):
   if self.rest == Link.empty:
     str = ''
   else:
     str=" --> " + Link.__str__(self.rest)

   return '{0}{1}'.format(self.first, str)

def __repr__(self):
    if self.rest == Link.empty:
      repstr = ''
    else:
      repstr=", " + Link.__repr__(self.rest)

    return ‘Link({0}{1})'.
               format(self.first, repstr)
```

```
>>> s = Link(30, Link(40, Link(50)))

>>> str(s)
‘30 --> 40 --> 50'

>>> repr(s)
'Link(30, Link(40, Link(50)))’

>>> s
Link(30, Link(40, Link(50)))
```

So we see that when the interpreter reads an object, it calls the repr function on that object and returns the repr string without quotes. The major difference between using str() and repr() is found in the intended audience. repr() is intended to produce output that is mostly machine-readable (in many cases, it could be valid Python code), whereas str() is intended to be human-readable.

MAGIC METHODS

Magic methods are an important aspect of object-oriented programming in Python. Magic methods are special methods that you can define to add substantially to the usability of your classes.

These methods are sometimes called ‘dunder’ methods, since they are always surrounded by double underscores (e.g., __init__ , __len__, or __add__).

A basic magic method is the init method, `__init__`. This is the method used to define the initialization behavior of an object. The `__init__` method is not the first to get called when a new object is defined. First a method called `__new__`, which actually creates the instance, and then passes any arguments at creation on to the initializer. At the other end of the object's lifespan, there is a deletion method `__del__`.

Most collection classes will define a magic method for length, such as we have seen. We can use the length function to implement a comparison operation, in fact we are overloading the operator >.

```
def __gt__(self, other):
    return len(self) > len(other)


>>> s = Link(3, Link(4, Link(5)))
>>> t= Link(100, Link(200))
>>> s > t
True
>>> t > s
False
```

The call magic method enables an object to become function. Here we show a call magic method that enables linked list object may to call itself and apply a function to each of its elements. An alternative to this would be to

define an auxiliary function that defined a map function applied to a linked list object.

```
def __call__(self,f):
    if self == Link.empty:
      return self

    if self.rest == Link.empty:
      self.first = f(self.first)
      return self

    self.first = f(self.first)
    self.rest.__call__(f)
    return self

>>> s = Link(30, Link(40, Link(50)))
>>> s(lambda x:x*x)
Link(900, Link(1600, Link(2500)))
```

OPERATOR OVERLOADING

Usability is enhanced if user can apply commonly used arithmetic and comparison based operators. Python has many magic methods designed to support intuitive comparisons between objects using symbolic operators. They also provide a way to override the default Python behavior for comparisons of objects. Here is an example of overloading the

+ operator, so that two linked lists can be added together.

```
def __add__(self, other):
    if self is Link.empty:
       return other
    else:
       return Link(self.first,
          Link.__add__(self.rest, other))


>>> s = Link(3, Link(4, Link(5)))
>>> t = Link(100, Link(200))

>>> s + t
Link(3, Link(4, Link(5, Link(100,
Link(200)))))

>>> t + s
Link(100, Link(200, Link(3, Link(4,
Link(5)))))
```

Here's the list of those common comparison methods and what they do:

```
__eq__(self, other)
Defines behavior for the equality
operator, ==.

__ne__(self, other)
Defines behavior for the inequality
operator, !=.

__lt__(self, other)
Defines behavior for the less-than
operator, <.

__gt__(self, other)
```

```
Defines behavior for the greater-than
operator, >.

__le__(self, other)
Defines behavior for the less-than-or-equal-
to operator, <=.

__ge__(self, other)
Defines behavior for the greater-than-or-
equal-to operator, >=.
```

Here's the list of those common binary arithmetic operators that can implement magic methods:

```
__add__(self, other)
Implements addition.

__sub__(self, other)
Implements subtraction.

__mul__(self, other)
Implements multiplication.

__floordiv__(self, other)
Implements integer division using
the // operator.

__div__(self, other)
Implements division using the / operator.
```

Nearly all common arithmetic binary and unary operators can be converted to a magic method for a class. For more details see https://rszalski.github.io/magicmethods/#comparisons

LINK LIST CLASS WITH MAGIC METHODS

Here is a summary of all the code that we developed for this class.

```
class Link:
   empty = ()

   def __init__(self, first, rest=empty):
      self.first = first
      self.rest = rest

   def __getitem__(self, i):
      if i == 0:
          return self.first
      else:
          return self.rest[i-1
                           ]
   def __len__(self):
      return 1 + len(self.rest)

   def __gt__(self, other):
       return len(self)> len(other)

   def __str__(self):
       if self.rest == Link.empty:
         str = ''
       else:
         str=" --> " + Link.__str__(self.rest)
       return '{0}{1}'.format(self.first, str)

   def __repr__(self):
      if self.rest == Link.empty:
        repstr = ''
      else:
        repstr=", " + Link.__repr__(self.rest)
      return 'Link({0}{1})'.format(self.first,
                                   repstr)

   def __add__(self, other):
      if self is Link.empty:
        return other
      else:
        return Link(self.first, Link.__add__(
                                self.rest, other))
```

A TREE CLASS WITH REPR

Trees can also be represented by instances of user-defined classes, rather than nested instances of built-in sequence types. Recall that a tree is a data structure that has as an data attribute and an attribute that is a sequence of branches that are also trees.

Previously, we defined trees in a functional style. Here is an alternative class definition, which we will use to illustrate new features that are magic methods.

We define the trees so that they have internal values at the roots of each subtree. The internal value is called a label in the tree. The Tree class below represents such trees, in which each tree has a sequence of branches that are also trees. A repr function is defined to return a string that could be evaluated to a Tree.

```
class Tree:
   def __init__(self, label, branches=()):
       self.label = label
       for branch in branches:
           self.branches = branches

   def is_leaf(self):
            return not self.branches

   def __repr__(self):
       if self.branches:
           return 'Tree({0},
               {1})'.format(self.label,
               repr(self.branches))
       else:
```

```
            return
                'Tree({0})'.format(repr(self.
                label))
```

The Tree class can represent, for instance, the values computed in an expression tree for the recursive implementation of fib, the function for computing Fibonacci numbers. The function fib_tree(n) below returns a Tree that has the nth Fibonacci number as its label and a trace of all previously computed Fibonacci numbers within its branches.

```
def fib_tree(n):
        if n == 1:
            return Tree(0)
        elif n == 2:
            return Tree(1)
        else:
            left = fib_tree(n-2)
            right = fib_tree(n-1)
            return Tree(left.label +
right.label, (left, right))

>>> fib_tree(5)
Tree(3, (Tree(1, (Tree(0), Tree(1))),
Tree(2, (Tree(1), Tree(1, (Tree(0),
Tree(1)))))))
```