Chapter 07 - Python Object-Oriented Shortcuts

April 4, 2021

0.1 Overview

- there are aspects of python that are more strucutral/functional programming
- **len()**:
  - we use `len()` function over `__len___` because the underscore represents that we should not call it directly
  - main reason is efficency
  - calling `__len__` calls the `__getattribute__` method, etc..
- **reversed()**:
  - we can customize `__reversed__` if we want some funky behavior
- **enumerate()**:
  - returns a sequence of tuples
- **all/any**:
  - accept an iteratable object and return `True` if all or any of the items evaluate to true
- **eval/exec/compile**:
  - execute string code as code inside the interputer; not safe
- **hasattr/getattr/setattr/delattr**:
  - allow attributes on an object to be manipulated by their string names
- **zip**:
  - takes two or more sequences and returns a new sequence of tuple

0.2 File I/O

- the `open()` built-in function is used to open a file and return a file object
- with open() you have `.write()` methods
- `wb` is for writing bytes to a file and `rb` allows us to read them
- once open we can use attributes such as `read`, `readline`, `readlines` methods to get the contents of the file
- use a `for` loop for reading because it can read line by line instead of the entire god damn file
- finally we need to use the `close` method

0.2.1 Placing it in context

- if we use `dir` on file-like objects, we see that they have two methods named `__enter__` and `__exit__`
- these methods turnd the file object into what is known as a `context manager`
- basically, if we use a special syntax called the `with` statement, these methods will be called before and after nested code is executed

```
with open('filename') as file:
    for line in file:
        print(line, end='')
```

- `with` statement is used in many placed where start-up or clean-up code needs to be executed
- with statement can apply to anything that has `__enter__` or `-__exit__`

[3]:
```
import random, string

class StringJoiner(list):
    def __enter__(self):
        return self

    def __exit__(self, type, value, tb):
        self.result = "".join(self)


with StringJoiner() as joiner:
    for i in range(15):
        joiner.append(random.choice(string.ascii_letters))

print(joiner.result)
```

```
DebMJTXCDMXHtCG
```

0.3 An alternative to method overloading

- python id duck typed and we only need one method that accepts any type of object
- in c++ maybe you need different objects because they have different paramaters
- `method overloading` is useful when we want a method with the same name to accept different numbers or sets of arguments

0.3.1 Defualt Arguments

- we can just do `default_args(x, y, z=None, a=False)`
- in the function, we could skip having to provide default value for `z` if we just initalize `a=True` in the function call
- you can place a `*` before the keyword-only arguments
-

[4]:
```
number = 5
def funky_function(number=number):
    print(number)

number = 6
funky_function(8)
funky_function()
print(number)
```

```
8
5
6
```

[8]:
```python
# Dont do this with lists, sets, dicts
# just make the defualt value none
def hello(b=[]):
    b.append('a')
    print(b)

hello()
hello()
```

```
['a']
['a', 'a']
```

0.3.2 Variable argument lists

- in python we can write methods that accept an arbitrary number of positional or keyword arguments without explicitly naming them
- we can also pass lists/dictionaries in some functions

[9]:
```python
def get_paged(*links):
    for link in links:
        print(link)
```

- we can also accept arbitrary keyword arguments
- these arrive in the function as a dictionary
- they are specified with two asterisks ** for example **kwargs
- dictionary is basically a json and we accept json like **kwargs

[13]:
```python
class Options:
    default_options = {
        'port': 21,
        'host': 'localhost',
        'username': None,
        'password': None,
        'debug': False
    }

    # we are making a copy of the dictionary
    # then we try to update it using supplied values
    def __init__(self, **kwargs):
        self.options = dict(Options.default_options)
        self.options.update(kwargs)

    def __getitem__(self, key):
        return self.options[key]
```

```
options = Options(username='dusty', password='drowssap', debug=True)
options['debug']
options['port']
options['username']
```

[13]: 'dusty'

- **kwargs is dangerous when because someone can pass wrong stuff into it but useful when you dont really know what being passed in

0.3.3 Unpacking Arguments

- given a list or dictionary of values, we can pass those values into a function as if they were normal positional or keyword arguments

[15]:
```
def show_args(args1, args2, args3="Three"):
    print(args1, args2, args3)

some_args = range(3)
more_args = {
    "args1": "ONE",
    "args2": "TWO"
}

print("Unpacking a sequence:", end=" ")

show_args(*some_args)
print("Unpacking a dict:",  end=" ")
show_args(**more_args)
```

```
Unpacking a sequence: 0 1 2
Unpacking a dict: ONE TWO Three
```

```
def __init__(self, **kwargs):
    self.options = {**Options.default_options, **kwargs}
```

0.4 Functions are Objects too

- in OOP languages, you're expected to create an object to sort of wrap the single method invloved
- there are times where we would like to pass around a small object that is simply called to perform an action
- in python, we don't need to wrap such methods in an object because funtions already are objects
- we can set attributes on function and we can pass them around to be called at a later time

[18]:
```
def my_function():
    print("The Function was Called")
```

```
my_function.description = "A silly function"

def second_function():
    print("The second was called")

second_function.description = "A sillier function."


def another_function(function):
    print("the description:", end=" ")
    print(function.description)
    print("The name:", end=" ")
    print(function.__name__)
    print("The class:", end=" ")
    print(function.__class__)
    print("Now I'll call the function passed in")
    function()

another_function(my_function)
print("")
another_function(second_function)
```

```
the description: A silly function
The name: my_function
The class: <class 'function'>
Now I'll call the function passed in
The Function was Called

the description: A sillier function.
The name: second_function
The class: <class 'function'>
Now I'll call the function passed in
The second was called
```

- the code shows above the function really is an object because we were able to assing `.description` attribute to it and called it successfully
- the fact that functions are top-level object is most often used to pass them around to be executed a later date
- if we have a callback function, the function is passed around like any other object and the timer never knows or cares what the orginal name of the function is or where it was defined
- when you call the `callback` function with the paramater, it is called
- methods are functions bound to an object

0.4.1 Using Functions as Attributes

- since functions are objects, they can be set as callback attributes on other objects
- it is possible to add or change function to an instantiated object

[19]:
```
class A:
    def print(self):
        print("my class is A")

def fake_print():
    print("my class is not A")

a = A()
a.print()
a.print = fake_print
a.print()
```

```
my class is A
my class is not A
```

- replacing a method like that is bad but it has its use in `monkey patching` in an automated test
- Monkey-patching can also be used to fix bugs or add features in third-party code

0.5 Callable Objects

- we can create object that can be called as if they were functions
- any object can be made callable by simply giving it a `__call__` method that accepts the required argument