Python Data Model

1. Object Attributes

By default, attributes (including functions) can be added dynamically to an object.

Exmaple: A function can be dynamically added to another function as an attribute.

In [1]:

```
def fun():
    pass

def yourfun():
    print("Your fun")

fun.yours = yourfun
fun.yours()
```

```
Your fun
```

Example: Class `Mine` is added with attribute `abc` and function `fun()` after class definition.

In [2]:

```
# Add a static function to class
def foo():
    print('foo()')

class Mine:
    pass

# Add an function to class
Mine.f = foo
Mine.f()

# Add an attribute to object
m = Mine()
m.v = 123
print(m.v)
```

```
foo()
123
```

2. Duck Typing

Python only check if an object fits for a purpose at the time of use.

- During execution, Python will simply accept any object which has a particular method.
- This is called **Duck Typing**.

```
"When I see a bird that walks like a duck and swim like a duck and quacks like a
duck, I call that bird a duck"
```

In [3]:

```
class Dog:
    def talk(self):
        print('woof')

class Tree:
    def talk(self):
        print('tree can talk?')

def make_sound(obj):
    obj.talk()

make_sound(Dog())
make_sound(Tree())
```

```
woof
tree can talk?
```

3. Introduction to Protocols

Recall that a list object supports following operations:

- slicing - return subset of list
- del() - delete an item
- print() - return string
- len() - return length

In [4]:

```
s = list(range(9))

print(s[1:5])
del s[5]
print(s)
len(s)
```

```
[1, 2, 3, 4]
[0, 1, 2, 3, 4, 6, 7, 8]
```

Out[4]:

```
8
```

Above methods are not built-in in the List object. In fact, these methods are available to many other object types, e.g. dictionary, set etc.

How does Python achieve it?

Python Protocols

Python Protocols are similar to **interfaces** in other programming languages.

- They pre-define a collection of methods an object must support to implement that protocol.
- These collection of methods are in the form of magic methods.

Python interpreter invokes these magic methods to perform basic object operations.

- Magic methods are dunder methods with leading and trailing double undscores. For example, the `__init__()` method.

Example:

Considering a class `Team` which represents a collection of members.

- Its initializer method `__init__()` initialize some members representing by string of "M0000X".

In [5]:

```
class Team:

    def __init__(self, members=None):
        if members:
            self._members = members
        else:
            self._members = ['M{:0>5}'.format(x) for x in range(5)]

t = Team()
print(t._members)
```

```
['M00000', 'M00001', 'M00002', 'M00003', 'M00004']
```

3.1 String Representation Protocol

The `str()` or `repr()` functions to get string representation of an object.

In [6]:

```
t = Team()
print(str(t))
print(repr(t))
```

```
<__main__.Team object at 0x000001E2A16D2358>
<__main__.Team object at 0x000001E2A16D2358>
```

But how can we get the string in the form of `['M00000', 'M00001', 'M00002', 'M00003', 'M00004']` ?

Let's extend `Team` class to a `Team1` class.

- Implements `__repr__()` & `__str__()` methods to return string representation of an object.

In [7]:

```python
class Team1(Team):

    def __str__(self):
        return str(self._members)

    def __repr__(self):
        return '{}({})'.format(self.__class__.__name__, self._members)

t = Team1()
print(str(t))
print(repr(t))
```

```
['M00000', 'M00001', 'M00002', 'M00003', 'M00004']
Team1(['M00000', 'M00001', 'M00002', 'M00003', 'M00004'])
```

3.2 Container Protocol

The `len()` function is used to check the size of a collection type, e.g. list and dictionary.

But it does not work on object of `Team1` . Following code will cause an Error.

In [8]:

```python
t = Team1()
# len(t)
```

How to make `Team1` object supports `len()` function to return member count?

- Implement a `__len__()` method which returns size of the member.

In [9]:

```python
class Team2(Team1):

    def __len__(self):
        return len(self._members)


t = Team2()
len(t)
```

Out[9]:

```
5
```

Recall that a list object supports following operations:

- slicing - return subset of list
- `del` - delete an item
- `for-loop` - Use as an iterable, e.g. in `for` loop
- `in` operator - Check wheather a member exists

In [10]:

```
s = list(range(9))

print(s[1:5])
del s[5]
for i in s:
    print(i, end=' ')
```

```
[1, 2, 3, 4]
0 1 2 3 4 6 7 8
```

To add above features to our class `Team1`, implement a `__getitem__()` method.

Note: Pyton also has `__contains__()` method and `__iter__()` & `__next__()` magic methods. But `__getitem__()` method is the fallback when those methods are not implemented.

In [11]:

```
class Team2(Team1):

    def __getitem__(self, position):
        return self._members[position]

    def __len__(self):
        return len(self._members)


t = Team2()

# # Indexing and Slicing
print(t[0], t[1:2])

# Check membership
print('M00001' in t)
print('M00009' in t)

# Iteration
for x in t:
    print(x, end=' ')
```

```
M00000 ['M00001']
True
False
M00000 M00001 M00002 M00003 M00004
```

3.3 Mutable Container

To support update item using indexing and modification of collection, we need to implmeent the `__setitem__()` & `__delitem__()` methods.

In [12]:

```
class Team3(Team2):

    def __setitem__(self, position, value):
        self._members[position] = value

    def __delitem__(self, position):
        del self._members[position]


t3 = Team3()

# Set an item
t3[0] = 'M00009'
print(t3[0])

# Delete an item
del t3[0]
print(t3[0])

# Shuffling and Sorting
import random
random.shuffle(t3)
print(t3)
print(sorted(t3))
```

```
M00009
M00001
['M00001', 'M00004', 'M00002', 'M00003']
['M00001', 'M00002', 'M00003', 'M00004']
```

3.4 Comparison Operators

Python provides following methods for implementing of comparison operations.

Operator	Method
<	`object.__lt__(self, other)`
<=	`object.__le__(self, other)`
==	`object.__eq__(self, other)`
!=	`object.__ne__(self, other)`
>=	`object.__ge__(self, other)`
>	`object.__gt__(self, other)`

Example:

We would like to compare two teams by its number of members, where the team with larger number of member is considered to be greater.

We only need to implement `__lt__()` , `__le__()` and `__eq__()` . Other methods are optional.

In [13]:

```python
class Team4(Team3):

    def __lt__(self, other):
        return len(self._members) < len(other._members)

    def __le__(self, other):
        return len(self._members) <= len(other._members)

    def __eq__(self, other):
        return len(self._members) == len(other._members)


t1 = Team4(['A', 'B', 'C', 'D'])
t2 = Team4(['C', 'D', 'E'])

print(t1 > t2)
print(t1 >= t2)
print(t1 != t2)
```

```
True
True
True
```

3.5 Arithmetic Operators

Python also provides a set of magic methods to override arithmetic operators.

Operator	Method
+	`object.__add__(self, other)`
-	`object.__sub__(self, other)`
+=	`object.__iadd__(self, other)`
-=	`object.__isub__(self, other)`

Example:

We would like to implement following features for `Team` class.

- C = A + B: Addition of team A and B creates a new team by combining members from both team.
- A += B: Add members in team B to A.

We will need to implement `__add__()` and `__iadd__()` .

In [14]:

```
class Team5(Team4):

    def __add__(self, other):
        return Team5(self._members + other._members)

    def __iadd__(self, other):
        self._members.extend(other._members)
        return self


ta = Team5(['A', 'B', 'C'])
tb = Team5(['D', 'E'])

tc = ta + tb
print(tc)

ta += tb
print(ta)
```

```
['A', 'B', 'C', 'D', 'E']
['A', 'B', 'C', 'D', 'E']
```

Reference

- Magic Methods and Operator Overloading (https://www.python-course.eu/python3_magic_methods.php)
- https://mypy.readthedocs.io/en/latest/protocols.html#predefined-protocols (https://mypy.readthedocs.io/en/latest/protocols.html#predefined-protocols)
- https://docs.python.org/3/library/collections.abc.html (https://docs.python.org/3/library/collections.abc.html)
- https://realpython.com/operator-function-overloading/ (https://realpython.com/operator-function-overloading/)