Python – Part4 – Sets, Dictionaries, Classes

For efficient data operations, Python supports List, Tuple, Set and Dictionary types. In this handout, you will study the Set, Dictionary, and some object-oriented concepts related to creating classes.

Set DataType:

The set datatype in Python is similar to the concept of a Set in Math, which is to contain some items. The important concept to note about a set is that it cannot contain duplicates. When we load a data into a set that has some duplicate values, it automatically removes the duplicates and only keeps one of the duplicate items. Another important point is that an item in a set is not stored at a particular position, so we cannot index a set variable by a position as we can in a list or a tuple. Set elements cannot be modified, but we can add, or remove items to a set.

Set operations in Python include: union, intersection, difference, and a few other capabilities that you will see in the following example.

Create a new Python project called **SetTest**. Then add a main function to the SetTest.py by typing main and clicking on the tab key twice. Then gradually type the following code in the SetTest.py (up to a print statement, and study the different operations with the set data type.

```
import sys

def main():
    # sets cannot have dulicate values, cannot change values directly
    # sets contain unordered data
    s1 = {"apples", "oranges", "bananas", "plum"}
    print(type(s1))
    print(s1)


    #----------set constructor-----------
    # we can load a tuple, or a list into a set
    s2 = set(("apples", "bananas", "mangoes","plum","mangoes"))
    # duplicates are automatically removed when loading data into a set


    # you cannot access items in a set by referring to an index or a key.
    #print(s2[2])

    # just like list or tuple, there is a len function available that will
    # tell us the number of items in a set
    print(len(s2))
    # we can also loop over a set variable and print the items in a set

    for x in s2:
        print(x)

    # you cannot change items in a set, but you can add items
    s2.add('kiwi')
    print(s2)
```

```
    # ----------add 2 sets using update method---------------
    s3 = ("strawberry", "blueberry")
    s2.update(s3) # update object can be a list or tuple
    print(s2)

    #------------------removing items---------------
    s2.remove("strawberry")
    # If the item to remove does not exist, remove() will raise an error,
    # you can use discard which does not raise an error
    s2.discard("blueberry")
    print(s2)

    #--------------clear method empties the set------------
    s2.clear
    print(s2)

    #------------union, intersection, difference of sets------------
    sa = {5, 20, 30, 40}
    sb = {7, 8, 20, 40}
    sc = sa.union(sb)  # 20 an 40 will appear one time
    print(sc)

    sd = sa.intersection(sb)
    print(sd)

    se = sa.difference(sb)
    print(se)

    #sets are immutable, but there is an indirect way to create the effect
    # of changing a set by loadig the set in a list, then modifying the list
    # and putting it back in a set
    listsc = list(sc)
    pos = listsc.index(20)
    listsc[pos] = listsc[pos] + 2
    sc = set(listsc)
    print(sc)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Dictionary DataType:

The dictionary in Python stores key:value pairs. For example, in representing the data for a student, the key can be 'fname', and its value 'Bill'. The different fields (or keys) and the associated values of a student can be represented in a dictionary as:

s1 = {**'fname'**:'Bill', **'lname'**:'Baker',**'id'**:1234,**'test1'**:85, **'test2'**:91}

The keys are shown in bold above. After each key, after the colon, the value of the key is indicated. The different key, value pairs are separated by a comma. As you can see, the dictionary makes the representation of data completely unambiguous and the data can be accessed simply by indexing over the key rather than its position. For example, to access the test2 score of the student in the above example, we will write it as:

t2 = s1['test2']

The keys in a dictionary are strings, where as the value of a key can be of any type, such as a string, number, object of a class etc.. Another important point to note that is that a dictionary internally uses a hash table to store the keys and the values. This makes the accessing of unordered information in a dictionary extremely efficient, especially when the data size is large. For example, in an online store, if we needed to lookup order information for a customer, the ordered can be used as the key, and the order detail would be the value. Even if we have millions of orders, we will simply lookup the order detail as (assuming all orders are stored in a dictionary called orders):

    orderinfo = orders['orderid']

The indexing (or search) using a key will result in an error if the key does not exist. To handle such situations, we use the get method on a dictionary object, e.g..

    t3 = s1.get('test3',-1)

The second parameter to the get method is the default value. If the key test3 does not exist, then t3 will be initialized to -1. If it does exits, then t3 will get the value of test3.

Create another Python project called **DictionaryTest**.
Add a class to the project by right clicking on the project name, and choosing "Add New Item". Then select the class option, with a name of Employee.py.

Type the following code in the Employee.py.

```
class Employee(object):
    def __init__(self,fnm,lnm,id,hrsw,pr):  # constructor - initialize data
        self.firstname = fnm
        self.lasttname = lnm
        self.id = id
        self.hours_worked = hrsw
        self.pay_rate = pr

    def compute_pay(self, overtime_rate):
        return self.hours_worked * self.pay_rate * overtime_rate
```

Add another class to the project called Student with the following code in Student.py.

```
class Student(object):
    def __init__(self, **kwargs): # ** makes the parameters a dictionary
        self.first_name = kwargs["first_name"]
        self.last_name = kwargs["last_name"]
        self.id = kwargs["id"]

    def tostring(self):
        return self.first_name + " " + self.last_name + " " + str(self.id)
```

Add another class to the project called Orders, with the following code in the Orders.py.

```
class Order(object):
    def __init__(self, orderid,productname,price,quantity):
        self.orderid = orderid
        self.product_name = productname
        self.price = price
        self.quantity = quantity
```

```
    def print_order_detail(self):
        info = (str(self.orderid) + " " + self.product_name + " " + str(self.price) +
                str(self.quantity))
        return info
```

Double click on the DictionaryTest.py in the solution explorer. After adding the main method in it, gradually type the following code (up to a print statement) to understand the concepts in the code.

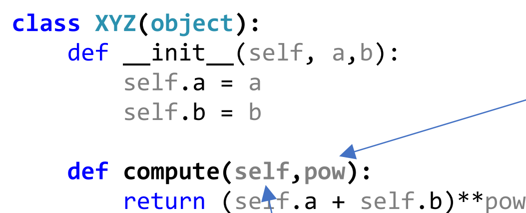
```
import sys
from Employee import Employee
from Student import Student
from Order import Order
def main():
    d1 = {
        "fname": "Bill",
        "lname": "Baker",
        "id": 1234 }
    print(type(d1))
    print(d1["lname"])
    d1["lname"] = "Bakerson"
    print(d1["lname"])

    # duplicate keys cannot be stored in a dictionary
    # if there is a duplicate key, the value replaces the original
    d1c = {'fname':'Bill','lname':'Baker','id':1234,'test':85,'test':91}
    print(d1c)

    # The values in dictionary items can be of any data type:
    e1 = Employee("Sally","Simpson",123,35,25.5)
    d1["bestemployee"] = e1
    d1["colors"] = ["red", "white", "blue"]
    print(d1["bestemployee"].firstname)

    print(d1.get("colors")) # get value of colors key

    allkeys = d1.keys()  # try values()
    print(allkeys)
    d2 = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
    }
    d2.update({"year": 2020})
    print(d2)

    #----------------to remove a key, use pop(key)------------
    d2.pop("year")
    print(d2)

    for x in d2:  # d2.keys(),  d2.values()
        print(d2[x])
```

```
for x, y in d1.items():  # for both key and value
    print(x, y)

d3 = d1.copy()   # can also use d3 = dict(d1)
print(d3)

#---------------clear() to clear the dictionary----
d2.clear()
print(d2)


# ---------looking up information in a dictionary
students = { "12345": "Bill", "12346":"Sally", "12347":"John"}

# find name of student whose id is 12347
name = students["12347"]
print(name)

#--------looking up information without using a dictionary
# looping over a list (inefficient approach)
s1 = Student(first_name="Bill",last_name="Baker",id=12345)
s2 = Student(first_name="John",last_name="Williams",id=12346)
s3 = Student(first_name="Cindy", last_name="Jacobs",id=12347)
slist = []
slist.append(s1)
slist.append(s2)
slist.append(s3)
# find name of student whose id is 12347
for i in range(0,len(slist)):
    if slist[i].id == 12347:
        print(slist[i].first_name)

# dictionary provides a very fast access to unordered data
# it eliminates the looping need as internally, it stores the
# the key value pairs as a hashtable
#--------------orders dictionary----------------
o1 = Order(1000,"laptop",895.50,2)
o2 = Order(1001,"cell phone",795.50,5)
o3 = Order(1002,"calculator",45.50,7)
orders = {}  # empty dictionary
orders[o1.orderid] = o1  # dict[key] = value
orders[o2.orderid] = o2
orders[o3.orderid] = o3

# find details of order whose orderid is 1001
ordertofind = input("please enter orderid:")
ord = orders.get(int(ordertofind))  # using get method to lookup a key's value
if ord is not None:
    print(ord.print_order_detail())
else:
    print('order does not exist')

#---------another way - use indexing by key on the dictionary
# to handle the error in case the key does not exist,
# enclose the indexing code in a try except block (you will learn
```

```
    # the details of the try except in a later handout)
    try:
        order = orders[int(ordertofind)]
        print(order.print_order_detail())
    except KeyError as ke:
        print('order does not exist')

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Python Class and Static Methods:

In Python, when we create a class, each method in it is usually declared with self as the first parameter. As explained in the lecture, the self keyword implicitly gets tied to the object through which the method is being called. For example, consider a class XYZ with two fields and a compute method as shown below. Create a Python project called "ClassStaticMethods", then add a class called XYZ.py to it.



```
class XYZ(object):
    def __init__(self, a,b):
        self.a = a
        self.b = b

    def compute(self,pow):
        return (self.a + self.b)**pow
```

After creating an object of the above class, we can trigger the compute method as:

```
    x1 = XYZ(5,7)
    res = x1.compute(3) # will compute (5+7)**3
    print(res)
```

Notice that in the call to **x1.compute(3)**, x1 is passed to the compute method as the self, so that it can access the fields a and b, before taking the power of their sum.

If the method in the class does not need to access the fields of a particular object, then we do not call the method with an object (e.g., x1.compute(3)), instead, we trigger the method by:

the name of the class.Name of the method

Such methods that are fully self contained, and only need the data passed in the parameters to do the computation (and do not need to access the data stored in an object) are referred to as static methods. Optionally, we can decorate a static method by putting @staticmethod in the previous line (just to be clear that, that is our intention and that we did not forget to put the self as the first parameter.

If we added a method called compute_avg, that can take three numbers as input and return the average to the XYZ class, then such a method will not need its first parameter to be self (and thus it is a static method). The modified XYZ class will appear as:

```
class XYZ(object):
    def __init__(self, a,b):
        self.a = a
        self.b = b
```

```
    def compute(self,pow):
        return (self.a + self.b)**pow

    @staticmethod  # this line is optional, and added just for clarity
    def compute_avg(a,b,c):
        return (a+b+c)/3
```

If we wanted to call the compute_avg method in the XYZ class, it will be called as:

```
res2 = XYZ.compute_avg(5, 7, 11)
print(res2)
```

In Python, a class cannot have more than one function with the same name. The capability to have more than one function with the same name is referred to as "method overloading" in the object-oriented terminology. Python does not allow method or constructor overloading. To overcome this limitation, Python allows tuples and dictionaries to be passed as parameters to a function. For example, if we wanted to have a function that can take the average of any number of parameters (not just 3 as in the above example), we will declare the function compute_avg with a parameter name starting with a *, indicating that the calling parameters will e converted to a tuple by this function.
Modify the compute_avg in the XYZ class to appear as:

```
    @staticmethod  # this line is optional, and added just for clarity
    def compute_avg(*data):  # * means, the parameter is a tuple
        return sum(data)/len(data)
```

Now you can call the compute_avg with any number of parameters as:

```
res2 = XYZ.compute_avg(5, 7, 11)
print(res2)

res3 = XYZ.compute_avg(6, 8, 3, 9, 7)
print(res3)
```

To allow for alternate construction (or initialization, since only one def __init__ is allowed), we can add a **"class method"** to the class, whose job is to internally trigger the def __init__. For example, in the XYZ class example, suppose we wanted to create an object where the data in the a and b fields will be the double of what we are passing. This can be accomplished by a class method as (shown in bold):

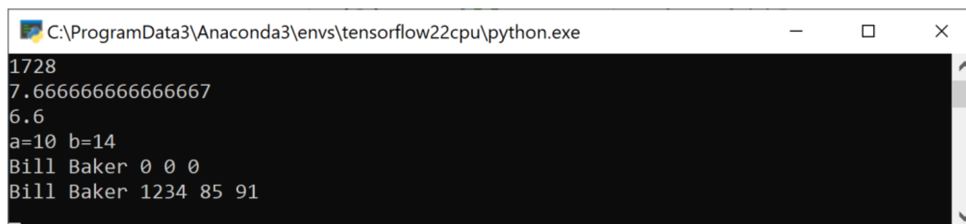
```
class XYZ(object):
    def __init__(self, a,b):
        self.a = a
        self.b = b

    @classmethod
    def create_object(cls,a,b): # cls is a keyword
        return cls(a*2, b*2) # triggers def __init__ implicitly to create the object

    def compute(self,pow):
        return (self.a + self.b)**pow

    @staticmethod  # this line is optional, and added just for clarity
```

```
    def compute_avg(*data):  # * means, the parameter is a tuple
        return sum(data)/len(data)

    def __str__(self):  # e.g., print(x1) will call __str__ implicitly
        return 'a=' + str(self.a) + ' b='+str(self.b)
```

The test code in the main method in the ClassStaticMethods.py appears as:

```
import sys
from XYZ import XYZ

def main():
    x1 = XYZ(5,7)
    res = x1.compute(3) # will compute (5+7)**3
    print(res)

    res2 = XYZ.compute_avg(5, 7, 11)
    print(res2)

    res3 = XYZ.compute_avg(6, 8, 3, 9, 7)
    print(res3)

    x2 = XYZ.create_object(5,7)
    print(x2)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

If you run the project, the output will appear as:



Functions with Variable Number of Parameters:

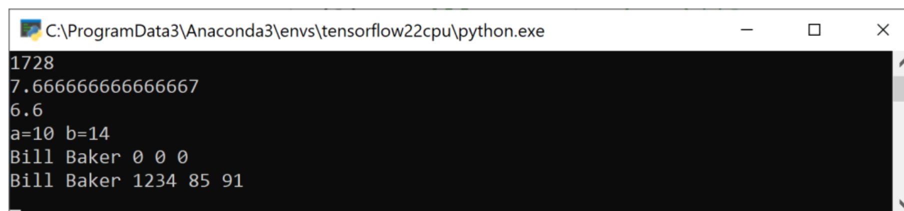
In Python, we can use three techniques to create flexible functions that can accept variable number of parameters. These include optional parameters, passing parameters as a tuple, and passing parameters as a dictionary. To show an example of this, add a class to the ClassStaticMethods project called Student, with the following code in it. Note that in the def __init__, the id, test1, test2 parameters have default values of 0 specified by assigning the parameter a value in the input parameter list. Such parameters are optional in the calling code. If no value for these parameters is passed, the default value is used, otherwise the calling value is used. The default (or optional) parameters have to appear after the normal parameters in the function declaration.

```
class Student(object):
    def __init__(self, fnm, lnm, id=0, test1=0,test2=0): #id,test1,test2 are optional
        self.first_name = fnm
```

```
        self.last_name = lnm
        self.id = id
        self.test1 = test1
        self.test2 = test2
```

Modify the ClassStaticMethods.py to appear as:

```
import sys
from Student import Student
from XYZ import XYZ

def main():
    x1 = XYZ(5,7)
    res = x1.compute(3) # will compute (5+7)**3
    print(res)

    res2 = XYZ.compute_avg(5, 7, 11)
    print(res2)

    res3 = XYZ.compute_avg(6, 8, 3, 9, 7)
    print(res3)

    x2 = XYZ.create_object(5,7)
    print(x2)

    s1 = Student("Bill","Baker") # id, test1, test2 are optional in the call
    print(s1.first_name + " " + s1.last_name + " " + str(s1.id) + " " +
          str(s1.test1) + " " + str(s1.test2))

    s1a = Student("Bill","Baker",1234,85,91) # data specified for optional parameters
    print(s1a.first_name + " " + s1a.last_name + " " + str(s1a.id) + " " +
          str(s1a.test1) + " " + str(s1a.test2))

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```



When we prefix a parameter with a **, in Python, it means that the function will receive the data as a dictionary. Modify the Student class to appear as shown below. You will see a special function __str__ below which returns a string representation of the fields of the class object. Methods that start with __ are usually triggered implicitly. If we create a Student object s1, then if we write print(s1), the __str__ function in the Student class will be called implicitly. The constructor in the student class has also been changed to accept variable number of parameters as a dictionary. The caller can create a student object with no parameters, or just the first_name, or with any of the remaining fields. The **kwargs parameter in the def __init__ indicates that the name value pairs passed to create a Student object will be

converted to a dictionary. The kwargs is not a keyword, and can be any name you pick. It so happens that kwargs is a popular name for a constructor parameter when it receives parameters as a dictionary. The student creation code now will typically appear as a name=value pair list e.g.,

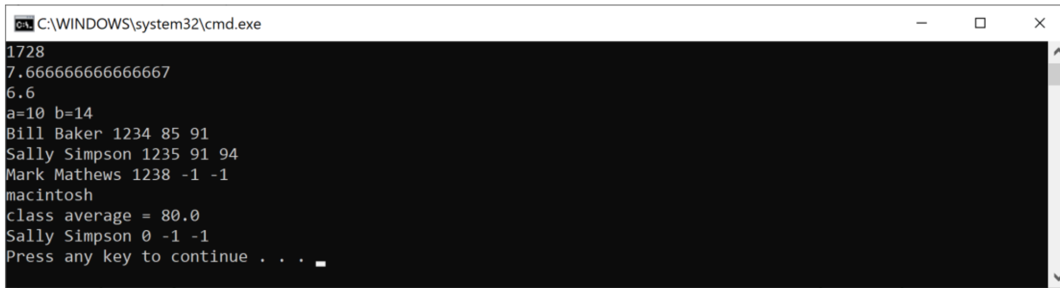
s1 = Student(first_name='Bill', last_name='Baker).

```
class Student(object):
    #def __init__(self, fnm, lnm, id=0, test1=0, test2=0): #id,test1,test2 are
optional
    #    self.first_name = fnm
    #    self.last_name = lnm
    #    self.id = id
    #    self.test1 = test1
    #    self.test2 = test2
    def __init__(self, **kwargs):  # ** means parameter is a dictionary
        self.first_name = kwargs.get("first_name","") #kwargs["first_name"]
        self.last_name = kwargs.get("last_name","") #kwargs["last_name"]
        self.id = kwargs.get("id",0) # kwargs["id"]
        self.test1 = kwargs.get("test1",-1) # kwargs["test1"]
        self.test2 = kwargs.get("test2", -1) # kwargs["test2"]

    # Python does not have function or constructor overloading
    #  class method allows us to create an object with different inputs
    # than the init. The first parameter is always the keyword cls
    # that allows us to indirectly call __init__
    @classmethod
    def create_by_full_or_lastname(cls, fullname):
        fname = ""
        lname = ""
        if fullname.find(" ") > 0:
            fname, lname = fullname.split(' ')
        else:
            lname = fullname
        student = cls(first_name=fname, last_name = lname) # cls implicitly calls __init__
        return student


    def __str__(self):  # implicitly called
        return self.first_name + " " + self.last_name + " " + str(self.id) + " " + \
                str(self.test1) + " " + str(self.test2)

    @staticmethod
    def all_students_test_average(*scores):  # scores is a tuple
        sum = 0
        for test in scores:
            sum = sum + test
        return sum/len(scores)
```

Modify the main in the ClassStaticMethods.py to appear as:

```
import sys
from Student import Student
from XYZ import XYZ
```

```python
def main():
    x1 = XYZ(5,7)
    res = x1.compute(3) # will compute (5+7)**3
    print(res)

    res2 = XYZ.compute_avg(5, 7, 11)
    print(res2)

    res3 = XYZ.compute_avg(6, 8, 3, 9, 7)
    print(res3)

    x2 = XYZ.create_object(5,7)
    print(x2)

    # since def init receives data as a dictionary, we have to pass name=value
    # for each field of student that we want to initialize explicitly
    # the order for the parameters does not matter now
    s1 = Student(first_name="Bill",last_name="Baker", id=1234, test1=85, test2=91)
    print(s1)
    #s1 = Student("Bill","Baker") # id, test1, test2 are optional in the call
    #print(s1.first_name + " " + s1.last_name + " " + str(s1.id) + " " +
    #      str(s1.test1) + " " + str(s1.test2))

    #s1a = Student("Bill","Baker",1234,85,91)  # data specified for optional parameters
    #print(s1a.first_name + " " + s1a.last_name + " " + str(s1a.id) + " " +
    #      str(s1a.test1) + " " + str(s1a.test2))

    s2 = Student(first_name="Sally", last_name="Simpson", id=1235, test1=91, test2=94)
    print(s2)

    s3 = Student(id=1238, first_name="Mark", last_name="Mathews")
    print(s3)

    dt = {"fruit":"apple", "price":75.50, "quantity":50, "category":"macintosh"}
    nm = dt.get("category","not known") # we can use [key], but get is a better choice
    # the second parameter in the get for a dictionary is the default value if key
    # does not exist
    print(nm)

    # call the static method all_students_test_average
    avg = Student.all_students_test_average(85,78,91,67,77,82)
    print('class average =',avg)

    # call the class method
    s4 = Student.create_by_full_or_lastname("Sally Simpson")
    print(s4)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```


C:\WINDOWS\system32\cmd.exe

```
1728
7.666666666666667
6.6
a=10 b=14
Bill Baker 1234 85 91
Sally Simpson 1235 91 94
Mark Mathews 1238 -1 -1
macintosh
class average = 80.0
Sally Simpson 0 -1 -1
Press any key to continue . . .
```