OBJECT ORIENTED PROGRAMMING

Brief Recap

- Classes, Objects and Methods
- Polymorphism
- Inheritance

Menu for Today!

- More on Classes
  - Class Variable
  - Private Variable
  - Functions: getattr, setattr
  - Destructors
- Python Iterables
  - Tuples
  - Dictionaries

MORE ON CLASSES

Class Variables

- Suppose we want to know how many students there are
  - *Create a "class variable"* `count` *inside the class* `Student`
  - This variable is shared by all instances of the class
  - Increment this variable every time we create a new instance of Student

Class Variables

```
class Student(Person):
    count = 0
    def __init__(self, first, last, rollNo):
        Person.__init(first, last)
        self.rollNo = rollNo
        Student.count += 1
student1 = Student(‘Rishi’, ‘Dutt’, ‘IMT2021001’)
student2 = Student(‘Keshav’, ‘Chandak’, ‘IMT2021003’)
print(f’Number of students : {Student.count}’)
```

Private Variables

```
class a:
    def __init__(self,a,b):
        self.x = a
        self.__y = b
    def getY(self):
        return(self.__y)
m = a(1,2)
m.x
m.__y
m.getY()
```

Functions to set and get Attributes

- The following two statements are equivalent

```
getattr(m,’x’)
m.x
getattr(m,’__y’) #type object 'a' has no attribute '__y’
```

- The following two statements are equivalent

```
setattr(m,’x’,4)
m.x = 4
```

Private Variables and setattr: Homework

- Try the following

```
m = a(1,2)
getattr(m,’__y’)
m.getY()
setattr(m,’__y’,3)
getattr(m,’__y’)
m.getY()
```

Destroying Objects

- Garbage Collection: delete unneeded objects automatically to free the memory space.
- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero.
  - *When an object's reference count reaches zero, Python collects it automatically.*
- You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space
  - *A destructor* `__del__()` *that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance*

Invoking a Destructor

```
class Student(Person):
    count = 0
    def __init__(self, first, last, rollNo):
        Person.__init(first, last)
        self.rollNo = rollNo
        Student.count += 1
    def __del__(self):
        print(f’Congrats {self.first}! You graduated!’)
        Student.count -= 1
```

Invoking a Destructor

```
student1 = Student(‘Rishi’, ‘Dutt’, ‘IMT2021001’)
student2 = Student(‘Keshav’, ‘Chandak’, ‘IMT2021003’)
print(f’Number of students : {Student.count}’)
del(student1)
del(student2)
```

Reference Count

Number of aliases that point to an object

- An object's reference count increases when
  - *It is assigned a new name*
  - *It is placed in a container (list, tuple, or dictionary).*
- The object's reference count decreases when
  - *It is deleted with del*
  - *Its reference is reassigned*
  - *Its reference goes out of scope.*

PYTHON ITERABLES

Python Iterables

- An `iterable` data type is any data type that provides the following capabilities:
  - *Get the “first” element*
  - *Get the “next” element*
  - *Inform us when there are no more elements to process*
- `List` is an `iterable`
- Other examples of iterable are `tuple`, `set` and `dictionary`

Unpacking Iterables

- Unpacking means assigning values of individual elements of an iterable to an variable
- Slicing is a form of unpacking

```
x = [1 , 2, 3]
y = x[0]
z = x[1:3]
```

- Iteration using for is also a form of unpacking

```
for i in x:
    print(i)
```

This syntax works on **all** iterable objects

Tuples

- Tuples are exactly like lists, except
  - *They are immutable*
  - *Syntactically, use* `( . . )` *for tuples instead of* [...] *for lists*

```
a = (1, 3, 5)
a[1] = 7 : error!!!
```

- Tuples can be indexed and sliced in the same way as lists
  - `a[1] = 3`
  - `a[1:3] = (3,5)`
- Tuples have their own methods : some methods are similar to list methods; others are different
  - `a.index(3) = 1`
  - `a.count(5) = 1`
  - `a.append(7) : error!!!`
- Iteration is very similar to lists

```
for i in a:
    print(i)
```

Associating Two Lists

- Consider the following code:

```
states = ['Karnataka', 'Maharashtra', ‘Gujarat', 'Telangana']
capitals = [‘Bengaluru', 'Mumbai', ‘Gandhinagar',
'Hyderabad']
state = 'Karnataka’
```

- How do we find the capital of `state`

```
def findCapital(state):
    stateIdx = states.index(state)
    return capitals[stateIdx]
```

- Assumes a certain relationship between the sequence of `states and capitals`
  - *Look-up is broken into a two step process*

Dictionaries

- We can use instead a powerful Python data structure called Dictionary

```
stateCapitals = {'KA': 'BLR', 'MH': 'BOM', 'AP': 'HYD’,
‘MYSORE’: ‘MYSORE’ }
```

```
stateCapitals['KA']  → 'BLR'
```

- Dictionaries are like lists, except that instead of indexing over integers, dictionaries are indexed over arbitrary values. The index is called a key

Dictionaries

- Python dictionary is a collection
  - *unordered*
  - *mutable*
  - *indexed.*
- Each item of a dictionary has a `key : value` pair
  - *A* `key: value` *pair is like a map*
  - *Keys must be unique and must be of immutable type(string, number or tuple*
  - *Values can repeat*
- Syntax uses curly brackets separated by commas.

Dictionaries and Lists: Similarities and Difference

- Elements of Dictionaries are accessed by key values, elements of Lists by indices
  - `statesCapital[1]`: *error*
  - `stateCapitals[‘Karnataka’]: ‘Bengaluru’`
- Iteration over lists returns elements, iteration over dictionaries returns key values. Try

```
for i in StatesCapital:
    print(i)
```

Dictionaries are Mutable

- We can add a key : value pair to a dictionary

```
stateCapitals['TS'] = 'HYD’
```

- We can change the value associated with a key

```
stateCapitals['AP'] = ‘Amaravati’
```

- We can delete the value associated with a key

```
del stateCapitals[‘MYSORE’]
```

Iterating on Dictionaries

- Recall that iteration over dictionaries returns keys.

```
for i in StatesCapital:
    print(i)
```

- What if we want to iterate over `key: value` pairs?

```
for i,j in StatesCapital.items():
    print(f’The capital of {i} is {j}’)
```

- `StatesCapital.items()` returns an object of class `dict_items`
  - Think of this class as a list of `(key, value)` tuples