COSC 2306
Data Programming

OOP and Class

Class and Object Attributes

```
class Person:
    population = 0
    def __init__(self, name):
        #Initializes the person.
        self.name = name
        print(f"Initializing {self.name}")
        # When this person is created, he/she adds to the population
        Person.population += 1
    def sayHi(self):
        #Greets the other person.
        print(f"Hi, my name is {self.name}")
```

```
p = Person('Tom')        >>>Initializing Tom
p.sayHi()                >>> Hi, my name is Tom.
```

How about print(p) and print(f"{p.sayHi()}")? Instance address and None

Class and Object Attributes

```
def howMany(self):
    #Prints the current population.
    # There will always be at least one person
    if Person.population == 1:
        print("I am the only person here.")
    else:
        print(f"We have {Person.population} persons here.")
```

```
tom = Person('Tom')
tom.sayHi()
tom.howMany()
```

```
Initializing Tom
Hi, my name is Tom.
I am the only person here.
```

```
kate = Person('Kate')
kate.sayHi()
kate.howMany()
```

```
Initializing Kate
Hi, my name is Kate.
We have 2 persons here.
```

```
tom.sayHi()
tom.howMany()
```

```
Hi, my name is Tom.
We have 2 persons here.
```

Operator overloading

Binary Operators:

OPERATOR	MAGIC METHOD
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)

source: https://www.geeksforgeeks.org/operator-overloading-in-python/

Operator overloading

```
class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, o):
        return self.a + o.a

ob1 = A(1)
ob2 = A(2)
ob3 = A("Geeks")
ob4 = A("For")

print(ob1 + ob2)
print(ob3 + ob4)

# Actual working when Binary Operator is used.
print(A.__add__(ob1, ob2))
print(A.__add__(ob3, ob4))

#And can also be Understand as :
print(ob1.__add__(ob2))
print(ob3.__add__(ob4))
```

```
3
GeeksFor

3
GeeksFor

3
GeeksFor
```

Operator overloading

Comparison Operators :

OPERATOR	MAGIC METHOD
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

source: https://www.geeksforgeeks.org/operator-overloading-in-python/

Operator overloading

```
class A:
    def __init__(self, a):
        self.a = a
    def __gt__(self, other):
        if(self.a>other.a):
            return True
        else:
            return False

ob1 = A(2)
ob2 = A(3)

if(ob1>ob2):
    print("ob1 is greater than ob2")
else:
    print("ob2 is greater than ob1")
```

ob2 is greater than ob1

Inheritance

- One of the major benefits of object oriented programming is **reuse** of code
- One of the ways this is achieved is through the inheritance mechanism
- Inheritance can be best imagined as implementing a *type and subtype* relationship between classes
- Consider this example:

Using Inheritance

```
class SchoolMember:
    # represents any school member
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print (f"Initialized SchoolMember: {self.name}")
    def tell(self):
        print (f"Name: {self.name} Age: {self.age}")
```

Using Inheritance

```
class Teacher(SchoolMember):
# represents a teacher
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print (f"Initialized Teacher: {self.name}")
    def tell(self):
        SchoolMember.tell(self)
        print (f"Salary: {self.salary}")
```

Using Inheritance

```
class Student(SchoolMember):
# represents a student
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print (f"Initialized Student: {self.name}")
    def tell(self):
        SchoolMember.tell(self)
        print (f"Marks: {self.marks}")
```

Using Inheritance

```
t = Teacher('Feng', 38, 100000)
s = Student('Tom', 21, 85)
members = [t, s]
for member in members:
    member.tell() # works for instances of Student & Teacher
```

```
Initialized SchoolMember: Feng
Initialized Teacher: Feng
Initialized SchoolMember: Tom
Initialized Student: Tom
Name: Feng Age: 38
Salary: 100000
Name: Tom Age: 21
Marks: 85
```

Multiple Inheritance

- Python supports a limited form of multiple inheritance as well

- A class definition with multiple base classes looks as follows:

```
class DerivedClassName(Base1, Base2, Base3):
<statement-1>
...
<statement-N>
```

- The only rule necessary to explain the semantics is the resolution rule used for class attribute references

Multiple Inheritance

- Resolution rule: depth-first, left-to-right
  - if an attribute is not found in DerivedClassName, it is searched in Base1
  - then (recursively) in the base classes of Base1
  - and only if it is not found there, it is searched in Base2
  - and so on

- A well-known problem: a class derived from two classes that happen to have a common base class
  - the instance will have a single copy of "instance variables" or data attributes used by the common base class

Attributes access

Python **does not** enforce access control to class resources

Convention:

```
class Point:
    def __init__(self):
        self.someAttribute       #public (any place)
        self._someOtherAttribute #protected (class derived from it)
        self.__secretAttribute   #private (within the class)
```

Container

Container: any object that holds an arbitrary number of other objects

- built-in containers: *tuple, list, set, dict*
- more container types are in the *collection* module

Iterators

Most container objects can be looped using a for statement:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
```

Iterators

```
for char in "123":
        print char
for line in open("myfile.txt"):
        print line
```

- This style of access is clear, concise, and convenient
- The use of iterators pervades and unifies Python
- Behind the scenes, the for statement calls iter() on the container object
- The function returns an iterator object that defines the method next() which accesses elements in the container one at a time
- When there are no more elements, next() raises a StopIteration exception which tells the for loop to terminate

Iterators

```
>>> s = 'abc'
>>> it = iter(s)
>>> print(it)
<iterator object at 0x00A1DB50>
>>> print(next(it))
'a'
>>> print(next(it))
'b'
```

Iterators

```
>>> print(next(it))
'c'
>>> print(next(it))
```

- Traceback (most recent call last):

```
File "<pyshell#6>", line 1, in -toplevel
  it.next()
StopIteration
```

Iterators

- Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to our classes

- Define a __iter__() method which returns an object with a next() method

- __iter__() must return the iterator object itself, other operations are optional (similar as __init__()

Iterators

```
class Reverse:
#Iterator for looping over a sequence backwards
def __init__(self, data):
    self.data = data
    self.index = len(data)
def __iter__(self):
    return self
def next(self):
    if self.index == 0:
            raise StopIteration
    self.index = self.index - 1
    return self.data[self.index]
```

Iterators

```
for char in Reverse('spam'):
        print char
```

```
m
a
p
s
```