CISS433: Python Programming
Lecture 19: More Operators

Yihsiang Liow

Agenda

- More on operators
- First introduction to iterators and exceptions
  - We will cover exceptions in greater detail later – it's an extremely important concept
  - We will probably not cover iterators in great detail. You'll find more on that in CISS350.

Operator Overloading

- Recall that in a class C we can define method **`__add__(self,x)`** and when you execute

  ```
  z = x + y
  ```

  where **`x,y`** are C-objects, then **`x+y`** is translated to **`C.__add__(x,y)`**
- **`__add__`** defines a new meaning for **`+`**
- We say that we are ***overloading*** **`+`**

Methods for Operator Overloading

__add__	x+y	C.__add__(x,y)
__radd__	a+x	C.__radd__(x,a)
__iadd__	x+=y	C.__iadd__(x,y)

[or __add__ if __iadd__ not defined]

and other arithmetic operators

__or__	x | y	C.__or__(x,y)
__call__	x(arg)	C.__call__(x,arg)

[Arbitrary number of arguments]

__len__	len(x)	C.__len__(x)

Methods for Overloading Operators

__getitem__	z = x[i]	z = C.__getitem__(x,i)
__setitem__	x[i] = z	C.__setitem__(x,i,z)
__cmp__	x==y, x<=y	C.__cmp__(x,y)
__lt__	x<y	C.__lt__(x,y) [or __cmp__ if __lt__ not defined]
__eq__	x==y	C.__eq__(x,y) [or __cmp__ if __lt__ not defined]
__getattr__	z = x.attr	C.__getattr__(x,attr)
__setattr__	x.attr = z	C.__setattr__(x,attr,z)

__getitem__ and for-loop

```
class C:

    def __init__(self):
        self.__attr = [1,2,3]
        pass

    def __getitem__(self,i):
        print("    __getitem__: i =", i)
        return self.__attr[i]

c = C()
for x in c:
    print(x)
```

- Can you tell me what's happening here?
- The `in` operator and list comprehension uses `__getitem__` too

__iter__ and for-loop

```
class C:

    def __init__(self):
        self.__index = None
        self.__list = [5,4,3,2,1]

    def __iter__(self):
        self.__index = -1
        return self

    def __next__(self):
        if self.__index >= len(self.__list)-1:
            raise StopIteration
        self.__index += 1
        return self.__list[self.__index]


c = C()
for x in c: print(x)
```

This is an exception

If **__iter__** not found, use **__getitem__**

__call__

- The __call__ method makes your objects look like functions.

```
class factorial:

    def __init__(self):
        pass

    def __call__(self,n):
        p = 1
        for i in range(1,n+1): p *= i
        return p

f = factorial()
print f(5)
```

__call__

- Rewrite the factorial function so that it ***keeps*** it computation of factorials So when f(5) is called again, it returns the factorial of 5 that was previously saved.

`__del__`

- In some OO languages, besides the constructor, there is a destructor
- The aim of the destructor is to perform some clean up operation before the object is destroyed
- For Python, memory is reclaimed automatically so usually the destructor is not defined
- If you want to perform some operation before the object is destroyed, then define the `__del__` method