CSE 445
Lecture 2

Python

Hello Python

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.
    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000
    Returns: string
    '''
    if size < 0:
        raise ValueError('number must be non-negative')
    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('number too large')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

Enough to Understand the Code

- Indentation matters to meaning the code
  - Block structure indicated by indentation
- The first assignment to a variable creates it
  - Dynamic typing: no declarations, names don’t have types, objects do
- Assignment uses `=` and comparison uses `==`
- For numbers `+ - * / %` are as expected.
  - Use of `+` for string concatenation.
  - Use of `%` for string formatting (like printf in C)
- Logical operators are words (`and`, `or`, `not`) *not* symbols
- The basic printing command is `print`

Basic Datatypes

- Integers (default for numbers)

```
z = 5 / 2  # Answer 2, integer division
```

- Floats/Doubles

```
x = 3.456
```

- Strings
  - Can use ”…" or ’…’ to specify, "foo" == 'foo’
  - Unmatched can occur within the string
    `“John’s”` or `‘John said “foo!”.’`
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:
    `“““a‘b“c”””`

Whitespace

- Whitespace is meaningful in Python, especially indentation and placement of newlines
- Use a newline to end a line of code

  Use \ when must go to next line prematurely
- No braces { } to mark blocks of code, use *consistent* indentation instead
  - First line with *less* indentation is outside of the block
  - First line with *more* indentation starts a nested block
- Colons start of a new block in many constructs, e.g. function definitions, then clauses

Comments

- Start comments with #, rest of line is ignored
- Can include a “documentation string” as the first line of a new function or class you define
- Development environments, debugger, and other tools use it: it’s good style to include one

```
def fact(n):
    “““fact(n) assumes n is a positive integer and returns
  factorial of n.”””
  assert(n>0)
  return 1 if n==1 else n*fact(n-1)
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

    `bob   Bob   _bob   _2_bob_   bob_2   BoB`

- There are some reserved words:

```
and, assert, break, class, continue, def, del, elif,
else, except, exec, finally, for, from, global, if,
import, in, is, lambda, not, or, pass, print, raise,
return, try, while
```

Naming conventions

The Python community has these recommended naming conventions

- joined_lower for functions, methods and, attributes
- joined_lower or ALL_CAPS for constants
- StudlyCaps for classes
- camelCase only to conform to pre-existing conventions
- Attributes: interface, _internal, __private

Sequence Types

- Sequences are *containers* that hold objects
  - List
  - Tuple
  - String
- Finite, ordered, indexed by integers
- Tuple: `(1, “a”, [100], “foo”)`
  - An *immutable* ordered sequence of items
  - Items can be of mixed types, including collection types
- Strings: `“foo bar”`
  - An *immutable* ordered sequence of chars
  - Conceptually very much like a tuple
- List: `[“one”, “two”, 3]`
  - A *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
  - Tuples and strings are *immutable*
  - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
  - most examples will just show the operation performed on one

Sequence Types 1

- Define tuples using parentheses and commas

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Define lists are using square brackets and commas

```
>>> li = [“abc”, 34, 4.34, 23]
```

- Define strings using quotes (“, ‘, or “””).

```
>>> st = “Hello World”
>>> st = ‘Hello World’
>>> st = “””This is a multi-line
string that uses triple quotes.”””
```

Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket “array” notation
- *Note that all are 0 based…*

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> tu[1]      # Second item in the tuple.
 ‘abc’

>>> li = [“abc”, 34, 4.34, 23]
>>> li[1]      # Second item in the list.
 34

>>> st = “Hello World”
>>> st[1]      # Second character in string.
 ‘e’
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0

```
>>> t[1]
'abc'
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Returns copy of container with subset of original members. Start copying at first index, and stop copying *before* the second index

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit first index to make a copy starting from the beginning of container

```
>>> t[:2]
(23, 'abc')
```

Omit second index to make a copy starting at 1st index and going to end of the container

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

- [:] makes a *copy* of an entire sequence

```
>>> t[:]
(23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to same ref,
            # changing one affects both
>>> l2 = l1[:] # Independent copies, two refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*

+ Operator is Concatenation

- The + operator produces a *new* tuple, list, or string whose value is the *concatenation* of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> “Hello” + “ “ + “World”
 ‘Hello World’
```

Mutability:
Tuples vs. Lists

Lists are mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
 ['abc', 45, 4.34, 23]
```

- We can change lists *in place.*
- Name `li` still points to the same memory reference when we're done.

Tuples are immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

- You can't change a tuple.
- You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

- *The immutability of tuples means they are faster than lists*

Tuple details

- The comma is the tuple creation operator, not parentheses

```
>>> 1,
(1,)
```

- Python shows parentheses for clarity (best practice)

```
>>> (1,)
(1,)
```

- Don't forget the comma!

```
>>> (1)
1
```

- Trailing comma only required for singletons others
- Empty tuples have a special syntactic form

```
>>> ()
()
>>> tuple()
()
```

Tuples vs. Lists

- Lists slower but more powerful than tuples
  - Lists can be modified and they have many handy operations and methods
- Tuples are immutable & have fewer features
  - Sometimes an immutable collection is required (e.g., as a hash key)
  - Tuples used for multiple return values and parallel assignments

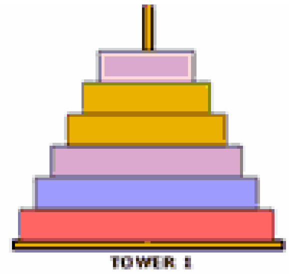
```
x,y,z = 100,200,300
old,new = new,old
```

- Convert tuples and lists using list() and tuple():

```
mylst = list(mytup); mytup = tuple(mylst)
```

Using Lists as Stacks

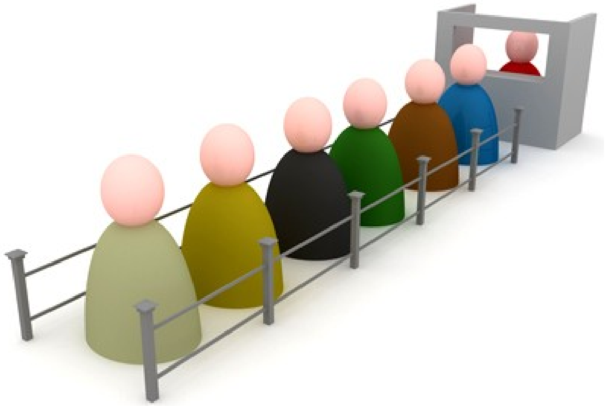
- The last element added is the first element retrieved
- To add an item to the stack, append() must be used
  - stack = [3, 4, 5]
  - stack.append(6)
  - Stack is now [3, 4, 5, 6]



- To retrieve an item from the top of the stack, pop must be used
  - Stack.pop()
  - 6 is output
  - Stack is now [3, 4, 5] again

Using Lists as Queues

- First element added is the first element retrieved
- To do this collections.deque must be implemented



```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

List Programming Tools

- Filter(function, sequence)
  - Returns a sequence consisting of the items from the sequence for which function(item) is true
  - Computes primes up to 25

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

Map Function

- Map(function, sequence)
  - Calls function(item) for each of the sequence's items

  - Compute...

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Reduce Function

- Reduce(function, sequence)
  - Returns a single value constructed by calling the binary function (function)
  - Comput

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

The del statement

- A specific index or range can be deleted

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

Looping Techniques

- Iteritems():
  - for retrieving key and values through a dictionary

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Set

- Create a set

```
a_set = {'a',}

type(a_set)
set

a_set = set()

type(a_set)
set
```

```
a_set = {'a','b','c','d','a','b','c'}

a_set
{'a', 'b', 'c', 'd'}
```

- Insert into set & check membership

```
a_set
{'a', 'b', 'c', 'd'}

a_set.add('c')

a_set
{'a', 'b', 'c', 'd'}
```

```
'a' in a_set
True

'e' in a_set
False
```

Set

- Some set operations

```
len(a_set)
4

b_set = a_set.copy()

c_set = {'d','b','a','c'}

a_set == b_set
True
```

- Some more operation

```
a_set.intersection(b_set)
{'a', 'b', 'c', 'd'}

a_set.union({'e','f'})
{'a', 'b', 'c', 'd', 'e', 'f'}
```

Subset operations

```
{1,2,3} < {1,2,4,5}
False

{1,2,3} < {1,2,4,5,3}
True

{1,2,3} <= {1,2,4,5,3}
True

{1,2,3}.issubset({1,2,4,5,3})
True
```

Set

- Set iterating & remove

```
In [255]: for a in a_set:
     ...:     print(a)
     ...:
b
c
a
d

In [256]: a_set.remove('a')

In [257]: a_set
Out[257]: {'b', 'c', 'd'}
```

Dictionary

- Creating a dictionary

```
a_map = {}

a_map = dict()

type(a_map)
dict
```

- Inserting elements of a dictionary

```
a_map['key1'] = 'value1'

a_map[2] = 'value2'

a_map[3.9] = 'value3'

a_map[True] = 'value4'

a_map
{'key1': 'value1', 2: 'value2', 3.9: 'value3', True: 'value4'}
```

Dictionary

- Accessing elements

```
In [197]: a_map[2]
Out[197]: 'value2'

In [198]: a_map[True]
Out[198]: 'value4'

In [199]: a_map[False]
Traceback (most recent call last):

  File "<ipython-input-199-ceb6c97d2420>", line 1, in <module>
    a_map[False]

KeyError: False
```

- Size of a dictionary

```
len(a_map)
4
```

Dictionary

- Iterating over dictionary

```
In [209]: for key,value in a_map.items():
     ...:     print(f'{key:<4}   {value}')
     ...:
key1   value1
2      value2
3.9    value3
1      value4
```

- Iterating over key/values

```
In [210]: for key in a_map.keys():
     ...:     print(key)
     ...:
key1
2
3.9
True
```

Dictionary

- Copy & delete

```
b_map = a_map.copy()

b_map
{'key1': 'value1', 3.9: 'value3', True: 'value4'}

a_map
{'key1': 'value1', 3.9: 'value3', True: 'value4'}

del a_map['key1']

a_map
{3.9: 'value3', True: 'value4'}

b_map
{'key1': 'value1', 3.9: 'value3', True: 'value4'}
```

Python OOP

Define a class

- How to define classes ?

```
class Person:
    species = 'human'
    def __init__(self,name,address):
        self.name = name
        self.address = address
```

- Start with the keyword class then name of the class
- Variables declared inside the class are class variable
- The __init__ method is the constructor of the class
- The self is the this pointer for the class

Using a class

- Instantiate a class using the name of the class

```
p = Person('John','123 main street')
p.name       #prints John
p.address   #print 123 main street
```

- Then access the methods/variables using instance name dot (.) the methods/variables names

Just one constructor?

- Not really
- Yes, we can define only one __init__ funciton but we can overload it
  - What happens if we define multiple __init__ function ?
  - Try it on your computer
- How do we overload ?
  - Use default arguments as None
  - Inside __init__ we can initialize using if…else

```
class Person:
    species = 'human'
    def __init__(self,name=None,address=None):
        self.name = name
        self.address = address
```

Instance attributes vs Class attributes

- What will be the output of the following code

```
class Person:
    species = 'human'
    age = 20
    def __init__(self,name=None,address=None):
        self.name = name
        self.address = address
    def setAge(self,a):
        self.age = a

p1 = Person()
p2 = Person()

p1.setAge(10)
p2.setAge(23)

print(p1.age)
print(p2.age)
```

Instance attributes vs Class attributes

- Class variables in python are defined just after the class definition and outside of any methods

```
class SomeClass:
    variable_1 = “ This is a class variable”
    variable_2 = 100   #this is also a class variable
```

- Unlike class variables, instance variables should be defined within methods

```
class SomeClass:
    variable_1 = “ This is a class variable”
    variable_2 = 100    #this is also a class variable.

    def __init__(self, param1, param2):
        self.instance_var1 = param1  #instance_var1 is a instance variable
        self.instance_var2 = param2   #instance_var2 is a instance variable
```

Instance, Class, and Static methods

- Just as there are instance and class variables, there are instance, class, and static methods
- These are intended to set or get status of the relevant class or instance
- So the purpose of the class methods is to set or get the details (status) of the class
- Purpose of instance methods is to set or get details about instances (objects)
- Static methods are different – used for grouping methods

Instance, Class, and Static Methods

- Let’s begin by writing a class that contains simple examples for all three method types

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Instance method

- The first method on `MyClass`, called `method`, is a regular *instance method*
- That's the basic, no-frills method type we use most of the time
- The method takes one parameter, `self`, which points to an instance of `MyClass` when the method is called (but of course instance methods can accept more than just one parameter)
- Through the `self` parameter, instance methods can freely access attributes and other methods on the same object
  - This gives them a lot of power when it comes to modifying an object's state.
- Instance methods can also access the class itself through the `self.__class__` attribute
  - This means instance methods can also modify class state

Class melthod

- Instead of accepting a `self` parameter, class methods take a `cls` parameter that points to the class—and not the object instance—when the method is called
- Because the class method only has access to this `cls` argument, it can’t modify object instance state
- That would require access to `self`
- However, class methods can still modify class state that applies across all instances of the class.

Static method

- The third method, `MyClass.staticmethod` was marked with a `@staticmethod` decorator to flag it as a *static method*
- This type of method takes neither a `self` nor a `cls` parameter (but of course it's free to accept an arbitrary number of other parameters)
- Therefore a static method can neither modify object state nor class state
- Static methods are restricted in what data they can access - and they're primarily a way to namespace your methods

Static method vs Class method

- The difference between a static method and a class method is:
  - Static method knows nothing about the class and just deals with the parameters.
  - Class method works with the class since its parameter is always the class itself.
- Static methods have very limited use case, because like class methods or any other methods within a class, they cannot access properties of the class itself
- However, when we need a utility function that doesn't access any properties of a class but makes sense that it belongs to the class, we use static functions/methods

Inheritance in python

- Child classes override *or* extend the functionality (e.g., attributes and behaviors) of parent classes
- In other words, child classes inherit all of the parent’s attributes and behaviors but can also specify different behavior to follow
- The most basic type of class is an `object`, which generally all other classes inherit as their parent
- When define a new class, Python 3 implicitly uses `object` as the parent class

Parent/Child concept example

- Let’s say we have a general Bank_account parent class that has Personal_account and Business_account child classes
- Many of the methods between personal and business accounts will be similar, such as methods to withdraw and deposit money, so those can belong to the parent class of Bank_account
- The Business_account subclass would have methods specific to it, including perhaps a way to collect business records and forms, as well as an employee_identification_number variable
- Similarly a Rectangle/Square/Triangle is a special case of a Polygon
  - All of them have area/sides
  - But different formula to calculate area, etc.

Inheritance example: Example parent class

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in 
range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

The child class

- We define Triangle which is a special Polygon

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3) # calling super __init__
        # OR super.__init__(3)
    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

- Triangle inherits the Polygon methods and define a new one
- The inherited methods need not be redefined

Method Overriding in Python

- In the above example, notice that `__init__()` method was defined in both classes, `Triangle` as well `Polygon`
  - When this happens, the method in the derived class overrides that in the base class
  - This is to say, `__init__()` in `Triangle` gets preference over the same in `Polygon`
- Generally when overriding a base method, we tend to extend the definition rather than simply replace it
- The same is being done by calling the method in base class from the one in derived class (calling `Polygon.__init__()` from `__init__()` in `Triangle`)
- A better option would be to use the built-in function `super()`
  So, `super().__init__(3)` is equivalent to `Polygon.__init__(self,3)` and is preferred
- Two built-in functions `isinstance()` and `issubclass()` are used to check inheritances
- Function `isinstance()` returns `True` if the object is an instance of the class or other classes derived from it

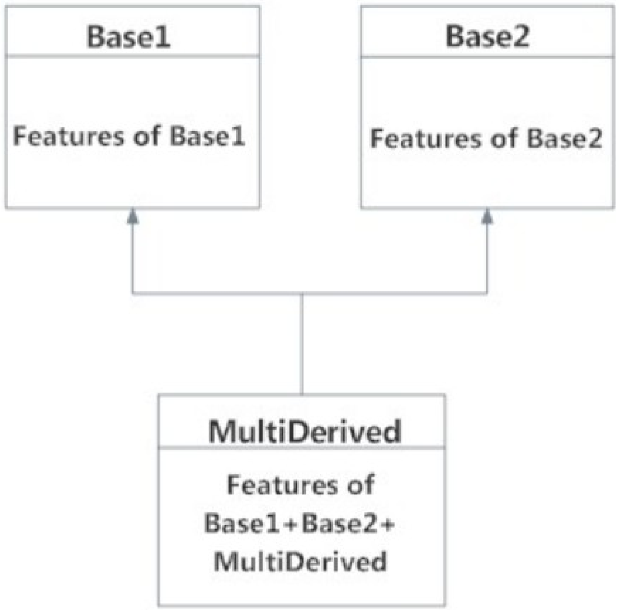
Multiple Inheritance

- In multiple inheritance, the features of all the base classes are inherited into the derived class
- The syntax for multiple inheritance is similar to single inheritance

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, 
Base2):
    pass
```



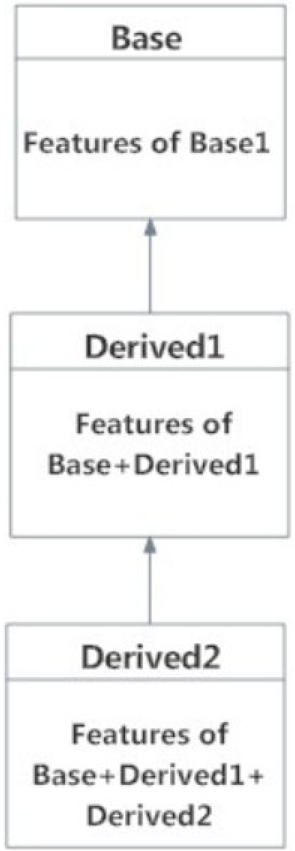
Multilevel Inheritance

- On the other hand, we can also inherit form a derived class
- This is called multilevel inheritance
  - It can be of any depth in Python
- In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```



Operator overloading

- Python operators work for built-in classes
- But same operator behaves differently with different types
  - For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings
- This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading

Operator overloading

- Try this code

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> p1 + p2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for
+: 'Point' and 'Point'
```

Operator overloading: Use special function

```python
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
```

Operator overloading: More spefical functions

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)