ICS Midterm Review Session

By Yifan Billy

Agenda

- Algorithms
  - Big-O Notation
  - Recursion
  - Sorting
  - Searching
  - Dynamic Programming
- OOP
  - A “class”
  - Operator Overloading
  - Inheritance
  - “_” and “__”
  - Decorators and Iterators
  - Matplotlib and Copy

Algor

h a task

Big-O Notati

- O(1): op
  - As
  - Li
  - Fo
- O(n): lo
  - Fo
  - W
- O(logn)
  - W
- Nested
- Parallel

```
def f(x):
    """Assume x is an int > 0"""
    ans = 0

    for i in range(1000):
        ans += 1
    print 'Number of additions so far', ans

    for i in range(x):
        ans += 1
    print 'Number of additions so far', ans

    for i in range(x):
        for j in range(x):
            ans += 1
            ans += 1
    print 'Number of additions so far', ans
    return ans
```

```
/2

in range(m):
= 1

n):
(1, n):
ange(1, n):
i+j+k)
```

Recursion: a function that calls itself

- Usual construction of a recursion function:
  - Base case
  - Loop structure

- Similar to a for-loop

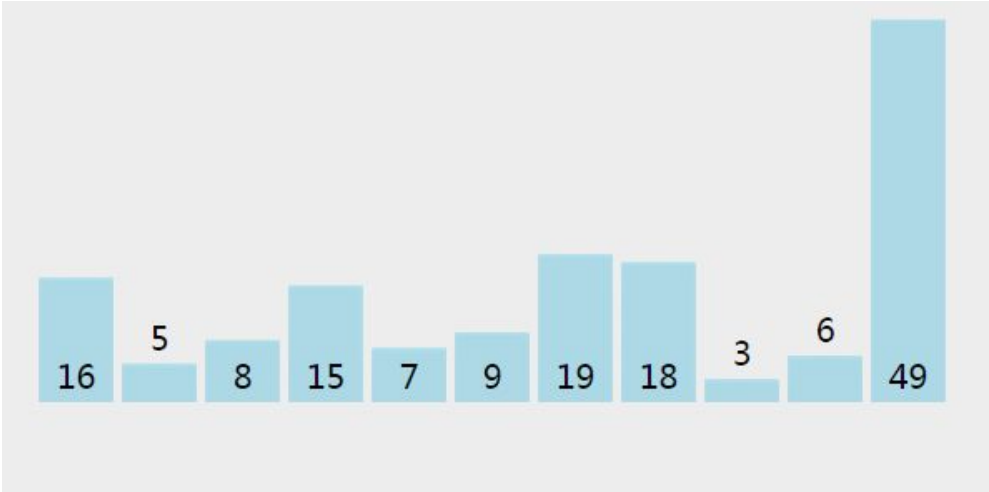
```
def a(n):
    for i in range(n+1):
            print(n-i)

def b(n):
    if n<0:
            return
    print(n)
    b(n-1)
```

Bubble Sort

```python
def bubble_sort1(my_list):
    for i in range(len(my_list)-1):
        for j in range(len(my_list)-1):
            if my_list[j] > my_list[j+1]:
                my_list[j], my_list[j+1] = my_list[j+1], my_list[j]
    return my_list


def bubble_sort2(my_list):
    N = len(my_list)
    for i in range(N-1):
        no_swap = True
        for j in range(N-1):
            if my_list[j] > my_list[j+1]:
                my_list[j], my_list[j+1] = my_list[j+1], my_list[j]
                no_swap = False
        if no_swap:
            break
    return my_list

# Optimized bubble sort function
def bubble_sort(my_list):
    N = len(my_list)
    for i in range(N-1):
        no_swap = True
        for j in range(N-1-i):
            if my_list[j] > my_list[j+1]:
                my_list[j], my_list[j+1] = my_list[j+1], my_list[j]
                no_swap = False
        if no_swap:
            break
    return my_list
```

- Swap if my_list[j] is greater than my_list[j+1]
- After i-th iteration, my_list[-i:] is sorted

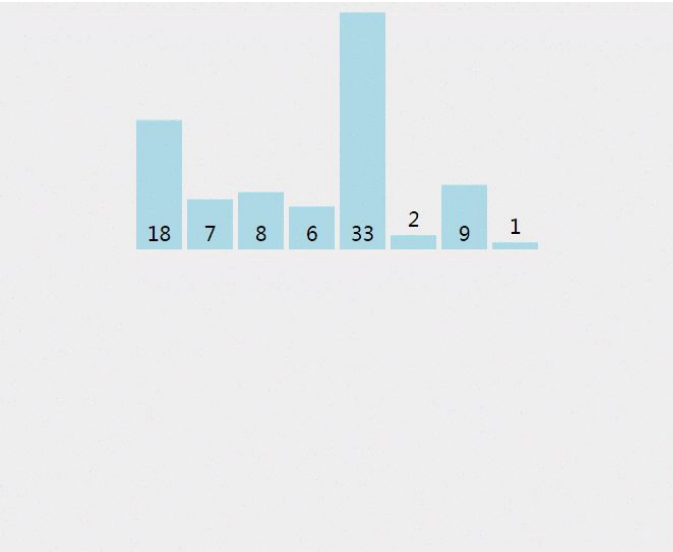
- Optimization: Stop the loop if my_list is sorted
- Runtime complexity: O(n^2)



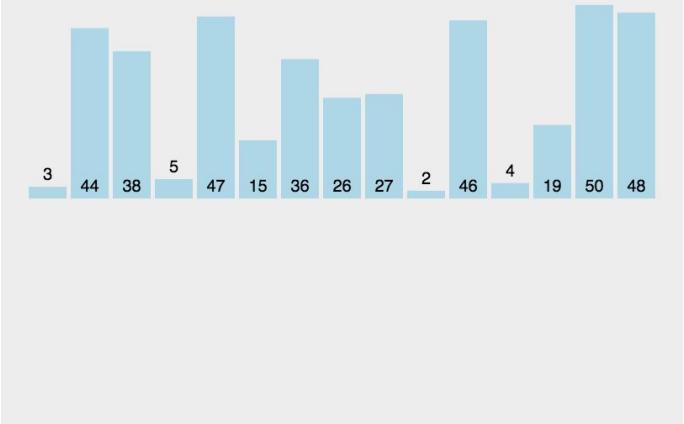
Merge Sort

```
# Merge function definition
def merge(left, right):
    result = []
#   comment out the line below and code the merge logic
    while left and right:
        if left[0] < right[0]:
            result.append(left[0])
            del(left[0])
        else:
            result.append(right[0])
            del(right[0])
    result += left + right

    return result


# Merge sort definition
def merge_sort(m):

    if len(m) <= 1:
        return m

    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]

    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)
```

- Recursion
- Runtime complexity: O(nlogn)



Insertion Sort / Quick Sort




```
def insert_sort(lists):
    for i in range(1,len(lists)):
        num=lists[i]
        j=i-1
        while j>=0:
            if lists[j]>num:
                lists[j+1]=lists[j]
                lists[j]=num
            j=j-1
    return lists
```

```
def add(left,nums,right):
    return left+nums+right
def quick_sort(num_list):
    if num_list==[]:
        return []
    if len(num_list)==1:
        return num_list
    num=random.choice(num_list)
    left,right,nums=[],[],[]
    for i in num_list:
        if i==num:
            nums=nums+[i]
    for i in num_list:
        if i<num:
            left=[i]+left
        elif i>num:
            right=right+[i]
    new_left=quick_sort(left)
    new_right=quick_sort(right)
    return add(new_left,nums,new_right)
```

Linear Search

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

Binary Search

```
def search(L, e):
    """Assumes L is a list, the elements of which are in
          ascending order.
       Returns True if e is in L and False otherwise"""

    def bSearch(L, e, low, high):
        #Decrements high - low
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bSearch(L, e, low, mid - 1)
        else:
            return bSearch(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return bSearch(L, e, 0, len(L) - 1)
```

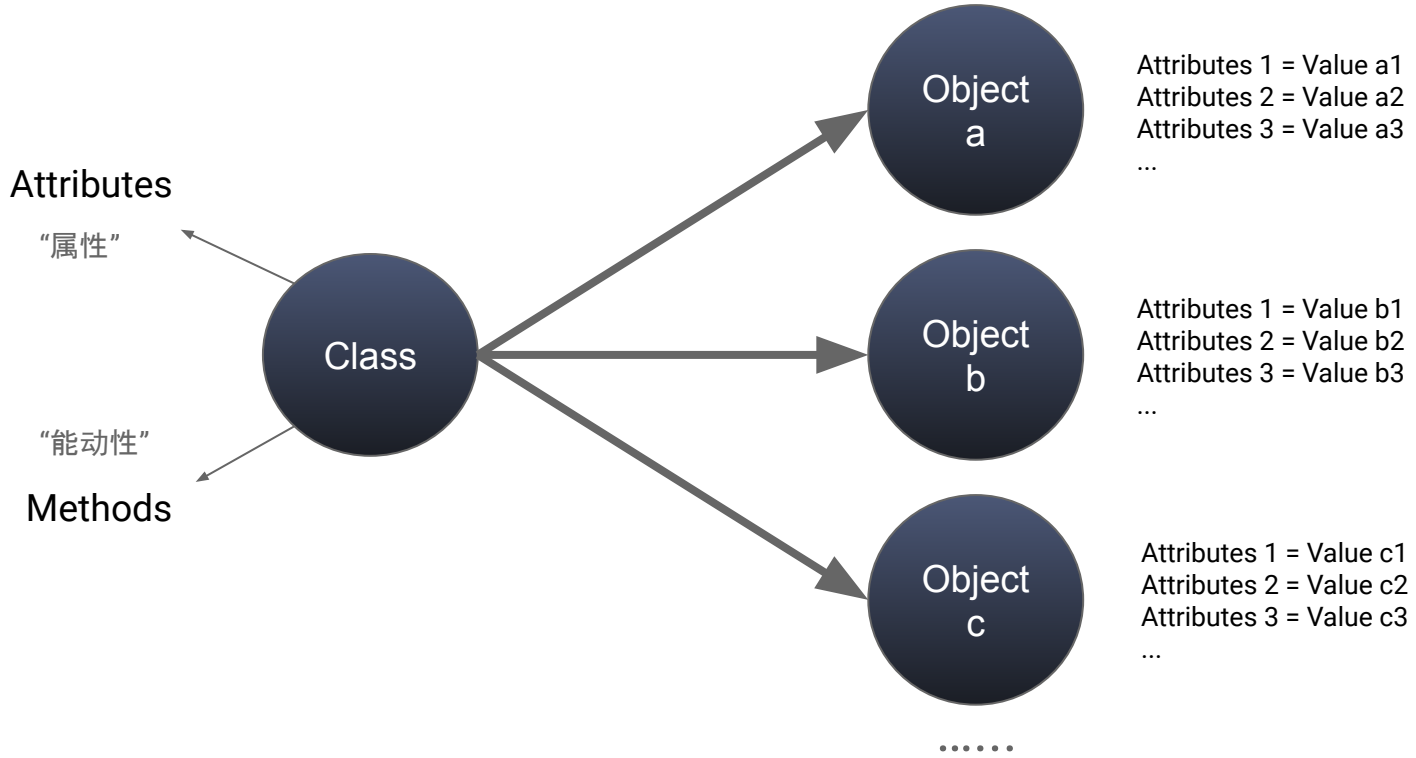
Dynamic Programming: Top-down approach of the problem

Knapsack Problem:

Knap(i, L)=max(Knap(i+1,L),Knap(i+1,l-w_i)+v_i)

```
num = inf
idx = 0
for i from 0 to (len(bills)-1):
    if b[i] <= value:
        sub_num = minBill(value-b[i])
        if (sub_num + 1) < num:
            num = sub_num + 1
            idx = i
```

OOP: A “Class”


Attributes
“属性”
“能动性”
Methods
Class
Object a
Attributes 1 = Value a1
Attributes 2 = Value a2
Attributes 3 = Value a3
...
Object b
Attributes 1 = Value b1
Attributes 2 = Value b2
Attributes 3 = Value b3
...
Object c
Attributes 1 = Value c1
Attributes 2 = Value c2
Attributes 3 = Value c3
...
.....

“self”

```
# create a class
class ClassName:

    # initialize
    def __init__(self,attribute1,attribute2,...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        ...

    # get method
    def get_attribute1(self):
        return self.attribute1

    # set method
    def set_attribute1(self,new_attribute1):
        self.attribute1 = new_attribute1

    # other methods
    def method1(self):
        pass

    def method2(self):
        pass

    ...

    # enable the "+" operator
    def __add__(self, other):
        pass

    # enable the "print()" function
    def __str__(self):
        pass

    # enable the "for...in..." statement
    def __iter__(self):
        yield
        pass
```

“getters” and “setters”

Why should we have getters?

--Encapsulation--

```
# create a class
class ClassName:

    # initialize
    def __init__(self,attribute1,attribute2,...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        ...

    # get method
    def get_attribute1(self):
        return self.attribute1

    # set method
    def set_attribute1(self,new_attribute1):
        self.attribute1 = new_attribute1

    # other methods
    def method1(self):
        pass

    def method2(self):
        pass

    ...

    # enable the "+" operator
    def __add__(self, other):
        pass

    # enable the "print()" function
    def __str__(self):
        pass

    # enable the "for...in..." statement
    def __iter__(self):
        yield
        pass
```

OOP: Operator Overloading

Predefined methods

“___***___”

___init___
___add___
___str___
___iter___
___next___
___eq___
___len___
___lt___
.....

the_sum = object_1 + object_2

print(object_1)

```
# create a class
class ClassName:

    # initialize
    def __init__(self,attribute1,attribute2,...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        ...

    # get method
    def get_attribute1(self):
        return self.attribute1

    # set method
    def set_attribute1(self,new_attribute1):
        self.attribute1 = new_attribute1

    # other methods
    def method1(self):
        pass

    def method2(self):
        pass

    ...

    # enable the "+" operator
    def __add__(self, other):
        pass

    # enable the "print()" function
    def __str__(self):
        pass

    # enable the "for...in..." statement
    def __iter__(self):
        yield
        pass
```

OOP: Inheritance

Inheritance allows a new class to extend an existing class.

The new class inherits the data attributes and methods of the class it extends.

```
class Human:

    def __init__(self,name,age):

        self.name = name
        self.age = age

    def get_name(self):

        return self.name

class Student(Human):

    def __init__(self,name,age,grade):

        super().__init__(name, age)
        self.grade = grade

    def get_grade(self):

        return self.grade
```

Understand “super()”.

```python
class Human:

    def __init__(self,name,age):

        self.name = name
        self.age = age

    def get_name(self):

        return self.name

class Student(Human):

    def __init__(self,name,age,grade):

        Human.__init__(self, name, age)
        self.grade = grade

    def get_grade(self):

        return self.grade
```

```python
class Human:

    def __init__(self,name,age):

        self.name = name
        self.age = age

    def get_name(self):

        return self.name

class Student(Human):

    def __init__(self,name,age,grade):

        super().__init__(name, age)
        self.grade = grade

    def get_grade(self):

        return self.grade
```

Some functions.

- dir()
- Function overloading
- isinstance(instance, class_name)

  issubclass(sub, super)

Now, you have learned all three pillars of OOP.

Inheritance! Encapsulation! Polymorphism!

OOP: “_” and “___”

“_”: Weekly hidden

“___”: Strongly hidden

Why do we have “_” and “___”?

Encapsulation!

```python
class Student:

    def __init__(self, name):

        self.__name = name

    def get_name(self):

        return self.__name

    def set_name(self,new_name):

        self.__name = new_name
```

```
billy = Student("Billy")
print(billy.__name)
```

```
---------------------------------------------------------------------
AttributeError                            Traceback (most rec
<ipython-input-37-c1263aae5188> in <module>
      1 billy = Student("Billy")
----> 2 print(billy.__name)

AttributeError: 'Student' object has no attribute '__name'
```

```
print(billy.get_name())
```

```
Billy
```

That's why we have getters and setters!

OOP: Decorators and Iterators

Decorators: @***

@property → getter

@property_name.setter

@staticmethod

.....

Iterators

```
class Object:

    def __init__(self, list_):

        self.list = list_

    def __iter__(self):

        self.idx = 0
        return self

    def __next__(self):

        self.idx += 1
        if self.idx == len(self.list)+1:
            raise StopIteration
        return self.list[self.idx-1]
```

```
class Object:

    def __init__(self, list_):

        self.list = list_

    def __iter__(self):

        for i in self.list:
            yield i
```

```
object_1 = Object([0,1])
for i in object_1:
    print(i, end= " ")
```

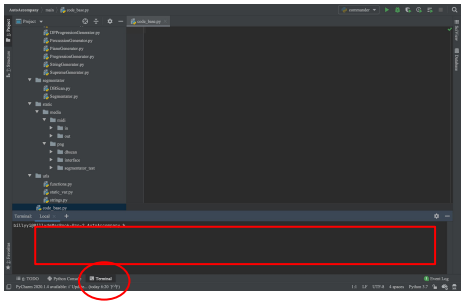
```
0 1
```

OOP: Matplotlib and Copy

Make sure you have installed Matplotlib!

>>>pip3 install matplotlib

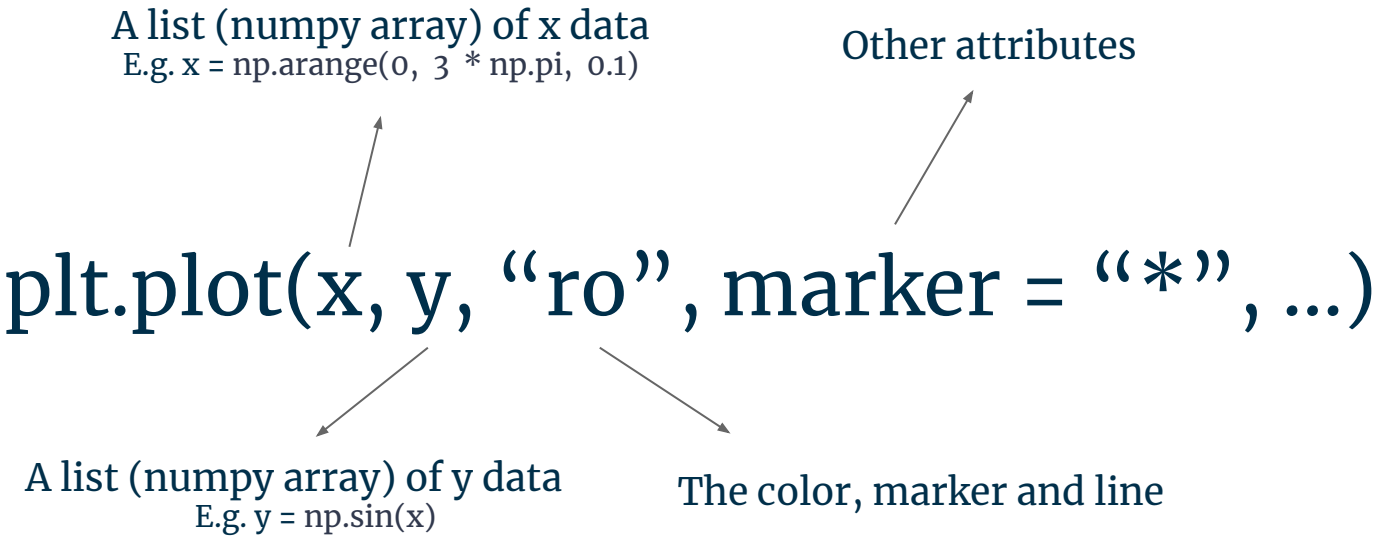
>>>If your IDE still tells you it cannot find Matplotlib after you have installed, try install Matplotlib in your IDE's terminal.



Probably no Matplotlib in midterm, but will be a important part in final!

No need to code by yourself, but have to understand.

```
>>>from matplotlib import pyplot as plt

>>>plt.plot(x, y, “ro”, marker = “*”, ...)

>>>plt.show()
```


A list (numpy array) of x data
E.g. x = np.arange(0, 3 * np.pi, 0.1)
Other attributes
plt.plot(x, y, “ro”, marker = “*”, ...)
A list (numpy array) of y data
E.g. y = np.sin(x)
The color, marker and line

Copy

```
>>>import copy
>>>object_1 = Object(“1”, [0, 1])
>>>object_2 = object_1
>>>object_3 = copy.copy(object_1)
>>>object_4 = copy.deepcopy(object_1)
```

```python
class Object:

    class ObjectInObject:

        def __init__(self):

            pass

    def __init__(self, str_, list_):

        self.str = str_
        self.list = list_
        self.inner_object = self.ObjectInObject()
```

	“=”	“copy”	“deepcopy”
Str (immutable)	same	different	different
List (mutable)	same	different	different
Inner object	same	same	different

“Same” means when you change an attribute of object 1, it will also change the corresponding attribute of object 2

Q & A

Thank you!

Do Well in Midterm!