ECE-122
Final Review

Class Notes

Chapter 1: Getting Started with Python

1.1 Variables-Type 1.2 Input/Output (I/O)- Python Files- Programming mode

Chapter 2: Elements of Programming

2.1 Functions; 2.2 Comments- Conditional Statements (if/elif/else)
2.3 Notion of Algorithms – Modules; 2.4 Notion of Data-Structure – Lists
2.5 Iterations: for loops; 2.6 Iterations: while loops; 2.7 Data Objects
2.8 Data Objects Complement-Procedural Programming; 2.9 Application: Create quiz

Chapter 3: Object Oriented Programming with Applications

3.1 Method Objects; 3.2 Method Objects Complement; 3.3 More on I/O-Reading/Writing
3.4 More on I/O- Graphics; 3.5 Class Anatomy – Encapsulation;
3.6 Encapsulation and Properties; 3.7 Inheritance; 3.8 Polymorphism

Chapter 4:Scientific Computing in Python

4.1 Introduction; 4.2: Numpy arrays; 4.3 Random and Matplotlib, 4.4 Applications

Chapter 5: Data Structure and Algorithms-

5.1 Tuple-dictionary-set; 5.2 Unsorted vs Sorted Lists-Binary Search Algorithm;
~~5.3 Simple Sorting: bubble, selection, insertion; 5.4 Stacks and Queues;~~ 5.5 Recursion
Complement: Exceptions and discussion about objects in Python

Class Notes

Chapter 1: Getting Started with Python

1.1 Variables-Type   1.2 Input/Output (I/O)- Python Files- Programming mode

Chapter 2: Elements of Programming

2.1 Functions; 2.2 Comments- Conditional Statements (if/elif/else)
2.3 Notion of Algorithms – Modules; 2.4 Notion of Data-Structure – Lists
2.5 Iterations: for loops; 2.6 Iterations: while loops; 2.7 Data Objects
2.8 Data Objects Complement-Procedural Programming; 2.9 Application: Create quiz

Chapter 3: Object Oriented Programming with Applications

3.1 Method Objects; 3.2 Method Objects Complement; 3.3 More on I/O-Reading/Writing
3.4 More on I/O- Graphics; 3.5 Class Anatomy – Encapsulation;
3.6 Encapsulation and Properties; 3.7 Inheritance; 3.8 Polymorphism

Chapter 4:Scientific Computing in Python

4.1 Introduction; 4.2: Numpy arrays; 4.3 Random and Matplotlib, 4.4 Applications

Chapter 5: Data Structure and Algorithms-

5.1 Tuple-dictionary-set; 5.2 Unsorted vs Sorted Lists-Binary Search Algorithm;
~~5.3 Simple Sorting: bubble, selection, insertion; 5.4 Stacks and Queues~~; 5.5 Recursion
Complement: Exceptions and discussion about objects in Python

OOP: summary

- Three fundamental paradigms in OOP:
  - **Encapsulation**- introduce methods that operate on data
  - **Inheritance**- create relationships between classes
  - **Polymorphism**- includes concepts of overriding and overloading
- Advantages of OOP
  - Provide high-level interfaces to define, organize and operate data
  - Help developers focusing on “What to do” rather than “How to do it”
  - Produce elegant software that is well designed and easily modified
  - Usability and Portability
- Some drawbacks (compared to POP)
  - Increase level of abstraction- less straightforward implementation
  - Badly design interfaces will hurt efficiency and performance

OOP- **Example- up to midterm**

```
class Fraction:
    def __init__(self,n=1,d=1): #constructor
        # n,d : instance variables
        self.n=n
        self.d=d

    def __str__(self): # return a string
        return str(self.n)+"|"+str(self.d)

    def flip(self):
        # method that works in place (no return)
        self.n, self.d = self.d, self.n

    def product1(self,f):
        # methods with return statement
        return Fraction(self.n*f.n,self.d*f.d)

    @staticmethod
    def product2(f1,f2): # no self argument
        return Fraction(f1.n*f2.n,f1.d*f2.d)
```

```
# Main (test constructor and str)
f1=Fraction(2,3)   # instantiate object f1
f2=Fraction()      # instantiate object f2
print(f1.n,f1.d)
print(f1,f2)

f3=Fraction()
f3.n=f1.n*f2.n
f3.d=f1.d*f2.d
print(f3)
```

```
2 3
2|3 1|1
2|3
```

```
# Main (test methods)
f1,f2=Fraction(2,3), Fraction(2,3)
f1.flip() # works in place
print(f1)

f3=f1.product1(f2) #call instance method
print(f3)

f3=Fraction.product2(f1,f2) #call static
print(f3)
```

```
3|2
6|6
6|6
```

Lecture 3.6- **Encapsulation**

- Encapsulation consists of using getter and setter methods to access instance attributes.
- Setter methods can be used to set some properties for the attributes.
- Encapsulation is often associated with the use of private attributes (information hiding) leading to the concept of data abstraction (two underscores __name)

```
class Fraction:
  def __init__(self,n=1,d=1): #constructor
    self.set_n(n)
    self.set_d(d)

  def get_n(self):
    return self.__n

  def get_d(self):
    return self.__d

  def set_n(self,a): # works in place
    self.__n=a

  def set_d(self,a): # works in place
    if a!=0:
      self.__d=a
    else:
      print("Nope can’t do that, will use 1")
      self.__d=1

  def __str__(self): # return a string
    return str(self.__n)+"|"+str(self.__d)
```

```
# Main
f1=Fraction(2,0)
print(f1)

f1=Fraction(2,3)
f2=Fraction()
print(f1.get_n(),f1.get_d())
print(f1,f2)

f3=Fraction()
f3.set_n(f1.get_n()*f2.get_n())
f3.set_d(f1.get_d()*f2.get_d())
print(f3)
```

```
Nope can’t do that, will use 1
2|1
2 3
2|3 1|1
2|3
```

Lecture 3.6- **Encapsulation**

- In Python, it is possible to keep the instance attribute public and still make use of encapsulation, two approaches: **property** function or **decorators**

```
class Fraction:
    def __init__(self,n=1,d=1): #constructor
        self.set_n(n)
        self.set_d(d)

    def get_n(self):
        return self.__n
    def get_d(self):
        return self.__d

    def set_n(self,a): # works in place
        self.__n=a
    def set_d(self,a): # works in place
        if a!=0:
            self.__d=a
        else:
            print("Nope can't do that, will use 1")
            self.__d=1
    def __str__(self): # return a string
        return str(self.__n)+"|"+str(self.__d)

    n=property(get_n,set_n)
    d=property(get_d,set_d)
```

```
f1=Fraction(2,2)
f1.d=0
print(f1)

f1=Fraction(2,3)   # instantiate object f1
f2=Fraction()      # instantiate object f2
print(f1.n,f1.d)
print(f1,f2)

f3=Fraction()
f3.n=f1.n*f2.n
f3.d=f1.d*f2.d
print(f3)
```

```
Nope can't do that, will use 1
2|1
2 3
2|3 1|1
2|3
```

Lecture 3.7- **Inheritance**

- Allows to derive a new class from an existing one (parent-child relationship)
- A child class can inherit methods from parents as well as class attributes
- They can inherit instance attributes using the **super** operator

```
class Parent:
  def __init__(self,arg1,arg2,arg3):
      self.arg1=arg1
      self.arg2=arg2
      self.arg3=arg3
```

```
class Child(Parent):
  def __init__(self,arg1,arg2,arg3,arg4):
      super().__init__(arg1,arg2,arg3)
      self.arg4=arg4
```

- **Private attributes cannot be directly accessed** (need get/set methods or the use of **protected** attributed)
- A child class can access the parent methods using the **super** operator
- A child class can override an inherited method (and provide its own implementation). Method is said to be overridden if parameters, and return type are the same.
- Multiple inheritances also possible in Python (collisions problems need to be resolved)

Lecture 3.8- **Polymorphism**

- Method overriding, Method overloading (same name, different arguments/return types), Operator overloading using methods __add__, etc.
- Python uses some overloading by default:
  - no need to declare the type of the input argument or the type of the return output with methods.
  - basic operation such +,*,... can operate on multiple types
- If the number of same type arguments for same name methods is different, we can make use of the arguments *args. If the arguments have different types, we could also introduce **type** conditions.

```
def poly_add(*args):
    if type(args[0])  is int: # check the type of 1st argument
        result=0
    elif type(args[0])  is str:
        result= ""
    elif type(args[0])  is list:
        result=[]
    for item in args:
        result = result + item  # case of operator overloading
    return result
print(poly_add(3,4,5,6))
print(poly_add("A "," bright"," day"))
```

Lecture 3.8- **Polymorphism**

- Examples of Operator overloading

+	__add__	<	__lt__
-	__sub__	<=	__le__
*	__mul__	==	__eq__
**	__pow__	=>	__ge__
/	__truediv__	>	__gt__
//	__floordiv__	!=	__ne__

```
class Vector:
    def __init__(self,a,b):
        self.a=a
        self.b=b
    def __str__(self):
        return "Vector (%s, %s)"%(self.a, self.b)

    def __add__(self, vec): # overload “+”
        return Vector(self.a+vec.a,self.b+vec.b)

    def __lt__(self,vec):     # overload “<”
        if self.a**2+self.b**2<vec.a**2+vec.b**2:
            return True  #vector self ‘smaller’ than vec
        else:
            return False
```

```
# Main Progam
v1,v2= Vector(1,1),Vector(3,4)
v3=v1+v2      # it is calling v1.__add__(v2)
print(v3)
if v1<v2:       # it is calling v1.__lt__(v2)
   print(str(v1)+" is smaller than "+str(v2))
```

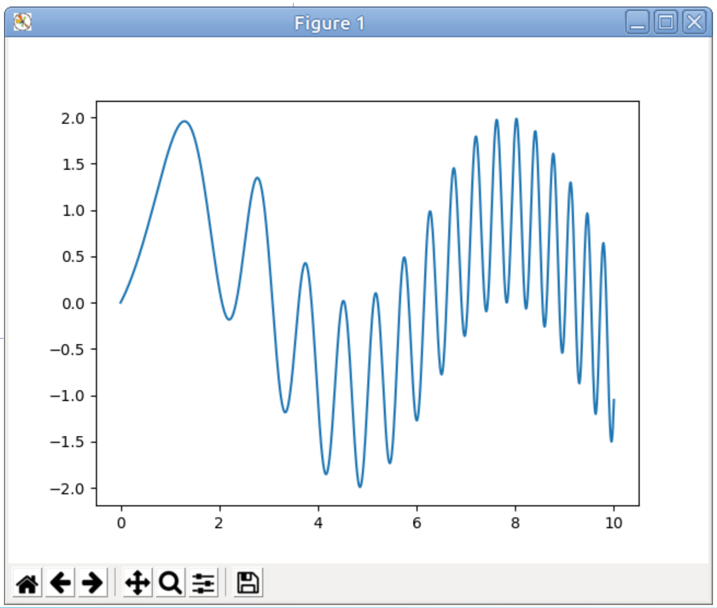
```
Vector (4, 5)
Vector (1, 1) is smaller than Vector (3, 4)
```

Lectures 4.1- 4.2 Numpy

- Python numerical computing framework: numpy, matplotlib, scipy
- Numpy arrays lead to vectorization and performance (while operating all items at once)
- Arrays could be 1d (vectors), 2d (matrices),...
- Lot of useful functions and methods to form arrays, fill-up with data, access elements (element by element or slices), etc.
- Important application: numerical representation of function. Example:
  f(x)=sin(x)+sin(x**2) 0<=x<=10

```
x = np.linspace(0,10,1000)   #consider 1000 grid points evaluation
f = np.sin(x) + np.sin(x**2)    #new vector f

#Visualization
import matplotlib.pyplot as plt
plt.plot(x,f)
plt.show()
```

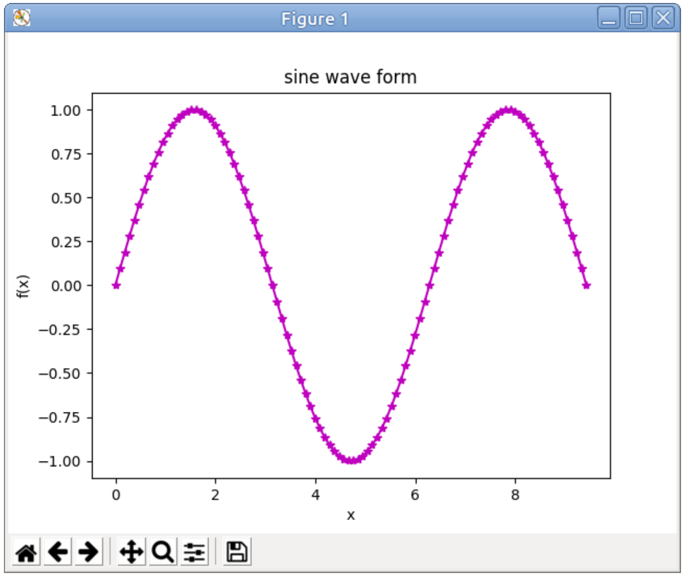


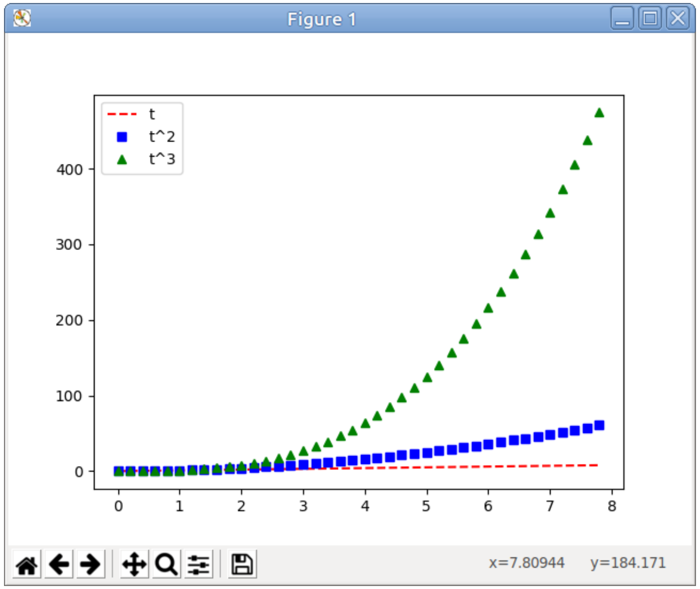
- multiple math methods available for statistics,

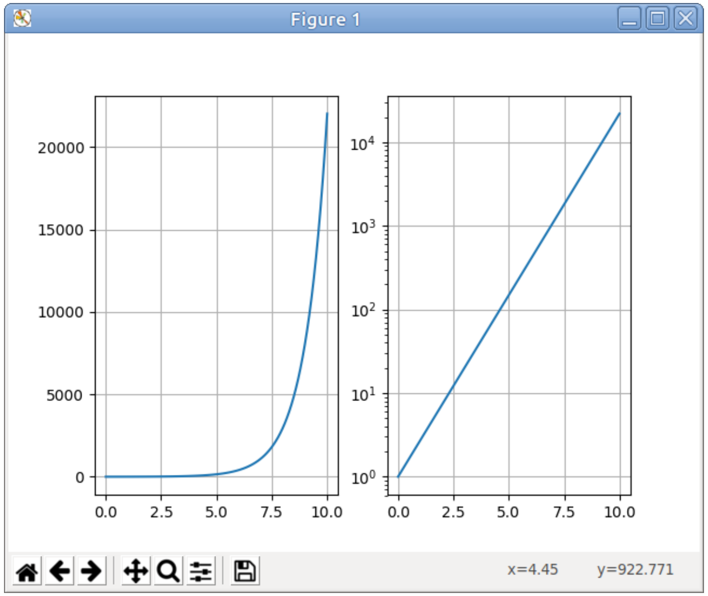
  linear algebra, etc.

Lecture 4.3 Random/matplotlib

- A pseudo-random generator is a useful and important numerical tool
- The random module offers a lot of functions: random, uniform, randint, choice, shuffle... It is also possible to use a seed for reproducibility.
- Numpy offers its own extension of the random module to generate large sample faster. Some functions: np.random.random, ...random_integer, etc.
- Matplotlib is a plotting library including pyplot with a lot functionalities







Lecture 5.1 Tuple/dict/set

- Tuple- immutable sequence of objects
  - use tuple instead of list if you do not plan to reassign the items
  - It is used everywhere within Python (internal construct)
  - function with multiple return values

```
tup1 = (1, 2, 3, 4, 5 )       # tuple of integer
tup2 = ("a", "b", "c", "d")  # tuple of string
tup3 = (True,False,True) # tuple of boolean
tup4 = ()                      #empty tuple
tup5 = (122,)                  #tuple with one item
```

- Dictionary
  - items are not identified by their indices but using a key
  - keys are immutable and unique
  - Multiple operations possible: search/accessing, delete, insert, regroup, scan

```
d1= {1:"Monday",2:"Tuesday",3:"Wednesday"}
d4 ={"Mon":1,"Tue":2,"Wed":3}
d5={} # an empty dictionary
# Add new values inside dictionary
d5["Mon"]="Monday"
d5["Tue"]="Tuesday"
d5["Wed"]="Wednesday"
```

- Set
  - sequence of values which is unordered and unindexed
  - set mutable but items are not
  - math operations (union, intersection, difference...)

```
s1 = {"apple", "banana", "cherry"}
s2 = {4,2,3,1,5}
s3=set() # empty set
s3.add("apple")
s3.add("banana")
s3.remove("apple")
print("cherry" in s3)
```

Lecture 5.2 Sorted list- binary search

- **Unsorted List**
  - fast insertion
  - slow search (linear search)- N/2 comparisons in average
  - slow deletion (items must be shifted)
- **Sorted List**
  - it offers very fast **binary search** $Log_2(N)$ steps:

    a good example of a customized data structure

    that can improve the efficiency of an algorithm
  - deletion still slow (items must be shifted)
  - slow insertion (items must be shifted)
- **Problems**: list must be sorted, insertion becomes very slow
- Sorted list useful in situations where once the list is ordered, search are very frequent but insertion/deletion are not -Example: search words in dictionary

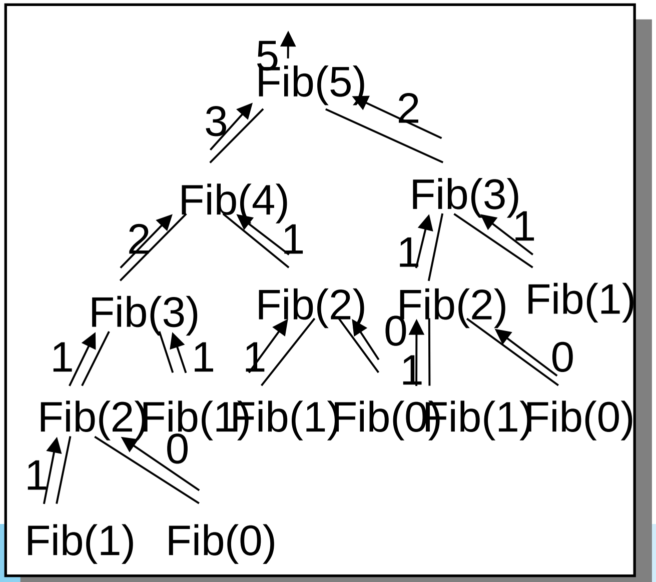
Lecture 5.5- Recursion

- **Recursion**
  - Programming technique in which a method calls itself
  - Uses **Base step+Recursive step**
  - Well-suited for the "**divide and conquer**" strategy

```
#function recsum
def recsum(n):
    if n==0:   # base step
        return 0
    else:      # recursive step
        return n+recsum(n-1)
```

- Simple Recursion:
  - Triangular number: **sum(n)=sum(n-1)+n** with **sum(0)=0**
  - Factorials: **fact(n)=fact(n-1)*n** with **fact(1)=1**
  - Binary search: keeps dividing in half
- Two recursions (divide and conquer)
  - Fibonacci (2 recursions) **fib(n)=fib(n-1)+fib(n-2)** with **fib(1)=1, fib(0)=0**
  - Hanoi Tower

```
def fib(n):
    if n<=1: # base step
        return n # 0 or 1
    else:     # recursive step
        return fib(n-1)+fib(n-2)
```



Lecture 6.1- **Exception**

- In Python, it is possible to include error handler which can both “catch” and "fix" the problem so the execution of a program can proceed.
- The programmer can then try to anticipate any potential errors using a **try** and **except** instruction block.

```
while True:
   try:
      a,b=map(float,input("Enter a,b to compute a/b: ").split())
      print(a/b)
      break
   except:
      print("Your b is 0, or you entered the wrong type")
```

```
Enter a,b to compute a/b: 3 0
Your b is 0,  or you entered the wrong type
Enter a,b to compute a/b: 3 a
Your b is 0,  or you entered the wrong type
Enter a,b to compute a/b: 3 1
3.0
```

Lecture 6.1 Passing to Functions

– Passing mutable objects (**list, dict, set, user-defined classes)**

```
def func1(b):
    b.append(1)

a=[2]
func1(a)
print(a)
```

```
def func2(b):
    b=b+[1]

a=[2]
func2(a)
print(a)
```

```
def func3(b):
    b+=[1]

a=[2]
func3(a)
print(a)
```

➡ [2,1]

➡ [2]

b=b+[1] is equivalent of using the __add__ overload operator method that does not work in-place (create a new object)

➡ [2,1]

b+=[1] is equivalent of using the __iadd__ overload operator method that works in-place (it is equivalent to the append method)

Activities Summary

	Lab7	Lab8	Lab9	Lab10	Project 3 OOP
Encapsulation	X				X
Inheritance	X				X
Polymorphism		X			X
Numpy+Random+ Matplotlib			X		X
Tuple/Dict/Set				X	
Binary Search				X	
Recursion				X	

Final s23

Final- ECE 122 – Spring 2023

Closed book/notes- no calculator- no phone- no computer

NAME:
ID:

Problem	**Score**
1- General questions (25pts)	
2- OOP (40pts)	
3- Numpy (17pts)	
4- Searching (8pts)	
5- Recursion (10pts)	
TOTAL (100pts)	

Final Words

- **Exam**: Monday May 22- [10:30am-12:30pm]
  - 2h in separate rooms: THOM104 or THOM106 (will send an email to clarify)
- Practice all labs, redo/understand all class note examples, be very well-prepared, and do your best
- 122 Material is comprehensive, lot of practices, you can keep using these class notes as references when needed
- Reminder about the course goals (syllabus):
  - √ Learn how to program (using a lot of programming practices)
  - √ Python syntax and fundamentals
  - √ Object oriented programming techniques
  - √ How to solve engineering/scientific problems with programming
  - √ Basic data structures and algorithms
- You should be very well-prepared for future challenges but keep practicing!
- Hope you enjoyed the Class, the Labs and the Projects!
- C'est la Fin, Merci.