Python Study Notes

February 12, 2022

1 Python Basics

This notebook serves as the sutdy notes for Python Language. The material is mainly follow the MIT courseware Introduction to Computer Science and Programming Using Python. This Chapter describes the basic `objects/while/for/if/logic/operations` concepts in Python language.

1.1 Scaler and Non-scalar Objects

`int`, `float`, `bool`, `NoneType` are built-in scalar obejcts (Python is an object-oriented language and eveything in python is an object of a class).

`list`,`tuple`, `list`, `dictionary` are non-scalar objects.

Can use `type()` to see the type (class) of an object.

We can directly convert object of one type to another.

[ ]:
```
print(type(5))
```

[ ]:
```
print(type(3.0))
```

[ ]:
```
print(float(3))
print(int(3.9))
```

1.2 Expressions in Python

Syntax for a simple expression `<object> <operator> <object>`

Common operators on `ints` and `floats` are `+,-,*,/`, **int division `//`**, **remainder `%`** and the **power `**`**.

Parentheses have the highest priority.

[ ]:
```
# 6/3 get 2.0 returns float 3.0, 5//2 returns integer 2
print(6/3)
print(5//2)
print(5%2)
```

1.2.1 Input/Output in Python

Keywords are `print(), input()`

`Text = input("Type anything...")`, input() takes string input and we can convert string to integer using `num=int(input("Type a number"))`

1.3 Binding variables and Values

Equal sign is an assignment of a value to a variable name. Re-bind variable names using new assignment statements. Previous value may still stored in memory but lost the handle of it.

1.4 Comparion Operators

Used for integers and floats `i>j`,`i>=j`, `i<j`, `i<=j`, **equality test `i==j`**, and **inequality test `i!=j`**.

1.5 Logic Operators

Used for bools `not a`, `a and b`, `a or b`.

1.6 Conditional Statement

`if (conditon): ... elif (condition): ... else:`, indentation matters in Python

[ ]:
```
# x = int(input('Enter an interger')) # input in Python
# COMPOUND BOOLEANS
x = 1; y =2; z = 3
if x<y and x<z:
    print('x is least')
elif y<z: # ELSE IF
    print('y is least')
else:
    print('z is least')
```

1.7 Strings

Strings can represent letters, special characters, spaces, digits. Strings are enclosed in double or single quotation marks.

1. Double quotation is handy and we can mainly use double quotes.

2. Use + to add (concatenate) strings together
3. Use " " as a blank space
4. String is a **non-scalar** object, meaning there are attributes associated with each string object.

[ ]:
```
hi = "Hello There!"
print(hi)
name = "eric!"
greeting = hi+" "+name
print(greeting)
```

1.7.1 String Operations

`Concatenation, successive concatenation, length, indexing, slicing, reverse, in`
(Note python uses a 0-based indexing system, while MATLAB uses the 1-based indexing system).

Strings are **immutable**; however, we can do re-assignment to modify the String.

[ ]:
```
hi = 'ab'+'cd' # CONCATENATION
print(hi)
hi1 = 3*'eric' # SUCCESSIVE CONCATENATION
print(hi1)
hi2 = len('eric') # THE LENGTH, ALSO INCLUDES THE SPACE
print(hi2)
hi3 = 'eric'[1] # INDEXING, BEGINS WITH INDEX 0, THIS RETURNS r
print(hi3)
hi4 = 'eric'[1:3] # SLICING, EXTRACTS SEQUENCE STARTING AT FIRST INDEX AND␣
 ↪ENDING BEFORE THE 3 INDEX
print(hi4)

# STRING OPERATION EXAMPLES
str1 = 'hello'
str2 = ','
str3 = 'world'

print('a' in str3) # bool, False, in/not in ARE TWO BASIC PYTHON MEMBERSHIP␣
 ↪OPERATORS
print('HELLO' == str1) # bool, False
str4 = str1 + str3 # STRING CONCATENATION
print('low' in str4) # bool, True
print(str3[:-1]) # string, worl, note -1 means the last element, -2 means the␣
 ↪second last element
print(str4[1:9:2]) # string, elwr, EXTRACT THE LETTERS WITH INDEX 1,3,5,7
print(str4[::-1]) # string, dlrowolleh, (REVERSE ORDER)
print(str4) # str4 itself is not changed in slicing operations
s = "hello"
s = "y" + s[1:len(s)] # strings are immutable, but we can re-assign the string.
print(s)
```

1.7.2 String Comparison Operations

`==, !=, >, >=, <, <=`

PYTHON COMPARES STRING LEXICOGRAPHICALLY (USING ASCII VLAUE OF CHARACTERS)

e.g. Str1 = "Mary", Str2 = "Mac", THE FIRST TWO CHARS ARE M = M, THE SECOND CHARS ARE THEN COMPARED a,a

ARE STILL EQUAL, THE THIRD TWO CHARS ARE THEN COMPARED r(ASCII 114) > c (ASCII 099)

A<B<C<…<Z<a<b<c<…<x<y<z

```
[ ]: print("tim" == "tie") # False
print("free" != "freedom") # True
print("arrow" > "aron") # True
print("right" >= "left") # True
print("teeth" < "tee") # False
print("yellow" <= "fellow") # False
print("abc">"") # True, NOTE THE EMPTY STRING "" IS SMALLER THAN ALL OTHER␣
 ↪STRINGS
```

1.7.3 String Method

1. EVERYTHING IN PYTHON IS AN OBJECT. OBJECTS ARE SPECIAL BECAUSE WE CAN ASSOCIATE SPECIAL FUNCTIONS, REFERRED TO AS OBJECT METHODS, WITH THE OBJECT.
2. More methods associated with Strings can be found here

```
[ ]: s = 'abc'
s.capitalize # returns the function type
s.capitalize() # invoke the function and returns Abc (need () to indicate a␣
 ↪method is invoked)
print(s.capitalize())
s.upper() # Return a copy of the string with all the cased chars converted to␣
 ↪uppercase
print(s.upper())
print(s.isupper()) # Return true if all cased characters in the string are␣
 ↪uppercase
           # and there is at least one cased character, false otherwise.
print(s.islower()) # similar to s.isupper
print(s.swapcase()) #Return a copy of the string with uppercase chars converted␣
 ↪to lowercase, vice versa.
print(s.find('e')) # Return the lowest index in the string where substring 'e'␣
 ↪is found,-1 if sub is not found
print(s.index('c')) # Like find(), but raise ValueError when the substring is␣
 ↪not found.
```

```
print(s.count('e')) # Return the number of non-overlapping occurrences of␣
 ↪substring e
print(s.replace('old','new')) # Return a copy of the str, all occurrences of␣
 ↪substr 'old' replaced by 'new'
```

1.8 While Loops

`while <condition>: <expression>`, note `<condition>` evaluates to a Boolean. If `<condition>` is `True`, do all the steps inside the while code block, and then check the `<condition>` again and repeat until `<condition>` is `False`.

Indentation matters!

[ ]:
```
# CONTRL FLOW while LOOPS , range(start,stop,step)
n = 0
while n<5: # CTRL + c IN THE CONSOLE TO STOP THE PROGRAM
    print(n)
    n = n+1
```

1.9 For Loops

`for n in range(5)`, is equivalent to n in [0,1,2,3,4]

`range(7,10)` starts at 7 stops at 10 (7,8,9) and `range(5,11,2)` starts at 5, stops at 11, step 2 (5,7,9)

`break` can be used for exiting the innermost loop (for,while)

`for` can loop through characters in strings

[ ]:
```
# break STATEMENT
mysum = 0
for i in range(5,11,2):
    mysum = mysum + i
    if mysum == 5:
        break
print(mysum)

# h, o ,l, a (for CAN LOOP CHARACTERS IN THE STRING)
for letter in 'hola':
    print(letter)
```

1.10 Iteration

Repeatedly use the same code. Need to set an iteration variable outside loop then test variable to determine when done and change variable within the loop.

Iterative algorithms allow us to do more complex things than simple arithmetic, one useful example are **guess and check** methods.

[ ]:
```
x = 3
ans = 0
itersLeft = x
while(itersLeft != 0):
    ans = ans +x
    itersLeft = itersLeft - 1
print(str(x)+'*'+str(x)+'='+str(ans))
```

1.11 Guess and Check Algorithm

We guess a solution and check iteratively. Guess a value for solution. Check if the solution is correct. Keep guessing until find solution or guessed all values. The process is exhaustive enumeration. Can work on problems with a finite number of possibilities.

[ ]:
```
# GUESS-AND-CHECK-cube root
cube = 28
for guess in range(abs(cube)+1):
    if guess**3 >= abs(cube):
        break
if guess**3 != abs(cube):
    print(cube, 'is not a perfect cube')
else:
    if cube < 0:
        guess = -guess
    print('Cube root of ' + str(cube) + ' is ' + str(guess))
```

2 Function/Iteration/Recursion/Modules/Files

This Chapter describes the Python `function/iteration/recrusion/modules/files`

- Bisection Search Algorithm
- Floats amd Fractions
- Newton-Rampson Root Finding Algorithm
- Functions
- Recursion
- Modules
- Files

2.1 Bisection Search Algorithms

We can use this algorithm to compute the monthly payment of a mortgage.

[ ]:
```
"""
BISECTION SEARCH - SQUARE ROOT
# REALLY RADICALLY REDUCES COMPUTATION TIME
"""
x = 25
epsilon = 0.01
```

```
numGuesses = 0
low = 1.0
high = x
ans = (high + low)/2.0

while abs(ans**2-x) >= epsilon:
    print('low = '+str(low)+' high = '+str(high)+' ans = '+ str(ans))
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0

print('numGuesses = '+ str(numGuesses))
print(str(ans) + ' is close to square root of '+ str(x))
```

[ ]:
```
"""
BISECTION SEARCH - CUBE ROOT
# THIS SCRIPT ALSO ADDRESSES THE CASES WHERE X IN (-1,1) AND X < 0
"""
x = -8
epsilon = 0.01
numGuesses = 0
low = 1.0
high = abs(x)

if abs(x) <= 1:
    low = 0
    high = 1

ans = (high + low)/2.0 # BISECTION METHOD

while abs(ans**3-abs(x)) >= epsilon:
    print('low = '+str(low)+' high = '+str(high)+' ans = '+ str(ans))
    numGuesses += 1
    if ans**3 < abs(x):
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0

if x < 0:
    ans = -ans

print('numGuesses = '+ str(numGuesses))
print(str(ans) + ' is close to cubic root of '+ str(x))
```

2.2 Floats and Fractions

1. Comupter represent numbers in binary format
2. Decimal number `302 = 3*100 + 0*10 + 2*1`
3. Convert an interger to binary form
4. For floats, IF WE MULTIPLE BY A POWER OF 2 (e.g 2^3) WHICH IS BIG ENOUGH TO CONVERT INTO A WHOLE NUMBER, CAN THEN CONVERT TO BINARY, AND THEN DIVIDE BY THE SAME POWER OF 2
    1. e.g. 3/8 = 0.375 = 3*10^-1 + 7*10^-2 + 5*10^-3; 0.375*(2**3) = 3 (DECIMAL), THEN CONVERT TO BINARY (NOW 11)
    2. THEN DIVIDE BY 2**3(SHIFT RIGHT) TO GET 0.011 (BINARY)
5. THERE ARE SOME PORBLEMS WITH COMPRAING TWO FLOAT POINTS BECAUSE COMPUTER TRIES TO SEE IF THE BINARIES ARE SAME.
    1. WE ALWAYS USE abs(x-y)< some small number, rather than x == y

[ ]:
```
#THE FOLLOWING PROGRAM CONVERTS INTERGERS TO BINARY FORMS
num = -10
if num < 0:
    isNeg = True
    num = abs(num)
else:
    isNeg = False
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
if isNeg:
    result = '-'+ result
print(result)
```

[ ]:
```
x = float(input('Enter a decimal number between 0 and 1:'))
p = 0
while ((2**p)*x)%1 != 0: # CONVERT TO A WHOLE NUMBER
    print('Remainder = '+str((2**p)*x-int((2**p)*x)))
    p += 1

num = int(x*(2**p))

result = ''
if num == 0:
    result = '0'
while num > 0: # CONVERT TO BINARY
    result = str(num%2) + result
    num = num//2

for i in range(p-len(result)):
```

```
    result = '0' + result

result = result[0:-p]+'.'+result[-p:]
print('The binary representation of the decimal '+str(x)+' is'+str(result))
```

2.3 Newton-Raphson

GENERAL APPROXIMATION ALGORITHM TO FIND ROOTS OF A POLYNOMIAL IN ONE VARIABLE P(X)=a_n x^n + a_n-1 x^n-1 + … + a_1 x + a_0 = 0

[ ]:
```
# WE USE THIS METHOD TO SOLVE p(x) = x^2 -24 = 0, WHREE x IS THE SQUARE ROOT OF 24
epsilon  = 0.01
y = 24.0
guess = y/2.0
numGuesses = 0

while abs(guess*guess - y) >= epsilon:
    numGuesses += 1
    guess = guess - (((guess**2)-y)/(2*guess)) # Intuitive explanation from wiki
print('numGuesses = '+str(numGuesses))
print('Square root of '+str(y)+' is about '+ str(guess))
```

2.4 Functions

1. Called/invokded/; `parameter/docstrings/body`; key word `def`; variable scope/global scope/function scope
2. Returns None if no return given
3. `printName(lastName = 'Huang',firstName = 'Zipeng', reverse = False)` (Most robust way with default value)

[ ]:
```
"""
FUNCTIONS
ARE NOT RUN IN A PROGRAM UNTIL THEY ARE CALLED/INVIKED
THEY HAVE: NAME, PARAMETERS (0, OR MORE), DOCSTRING(EXPLAIN WHAT A FUNCTION DOES) , BODY
"""
def is_even(i): # def IS A KEYWORD, IS_EVEN (NAME), i is PARAMETER/ARGUMENT
    """
    INPUT: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("hi")
    return i%2 == 0 # None, if no return given, only one return executed inside a function
```

```
                    # code insider function but after return statement noe␣
 ↪excuted

x = is_even(3) # x is False

def func_a(): # no parameter
    print('inside func_a')

def func_b(y):
    print('inside func_b')
    return y

def func_c(z):
    print('inside func_c')
    return z()

print(func_a())
print(5+func_b(2))
print(func_c(func_a)) # call func_c, takes one parameter, another function (a␣
 ↪func invokes another func)

# INSIDE A FUCNTION, CAN ACCESS A VARIABLE DEFINED OUTSIDE
# INSIDE A FUCNTION, CANNOT MODIFY A VARIABLE DEFINED OUTSIDE
def g(y):
    print(x)
    print(x+1) # x = x+1 is not valid
x = 5
g(x)
print(x)
```

[ ]:
```
"""
KWYWORD ARGUMENTS AND DEFAULT VALUES
"""
def printName(firstName,lastName,reverse):
    if reverse:
        print(lastName + ','+firstName)
    else:
        print(firstName,lastName)

# EACH OF RHESE ONVOCATIONS IS EQUIVALENT
printName('Zipeng','Huang',False)
printName('Zipeng','Huang',reverse = False)
printName('Zipeng',lastName = 'Huang',reverse = False)
# THE LAST INVOCATION IS RECOMMENDED SINCE IT IS ROBUST
printName(lastName = 'Huang',firstName = 'Zipeng', reverse = False)

"""
```

```
WE CAN ALSO SPECIFY THAT SOME ARGUMENTS HAVE DEFAULT VALUES, SO IF NO VALUE
SUPPLIED, JUST USE THAT VALUE  Default value
"""
def printName(firstName,lastName,reverse = False):
    if reverse:
        print(lastName + ','+firstName)
    else:
        print(firstName,lastName)

printName('Zipeng','Huang')
printName('Zipeng','Huang',True)
```

2.5 Recursion

1. DIVIDE AND CONQUER, `A FUNCTION CALLS ITSELF` (Mathematical Induction Reasoning)
    1. WE SOLVE A HARD PROBLEM BY BREAKING IT INTO A SET OF SUBPROBLEMS SUCH THAT:
    2. SUB-PROBLEMS ARE EASIER TO SOLVE THAN THE ORIGINAL
    3. SOLUTIONS OF THE SUB-PROBELMS CAN BE COMBINED TO SOLVE THE ORIGINAL
2. RECURSIVE STEP: THINK HOW TO REDUCE PROBLEM TO A SIMPLER/SMALLER VERSION OF SAME PROBLEM.
3. BASE CASE: KEEP REDUCING RPOBLEM UNTIL REACH A SIMPLE CASE THAT CAN BE SOLVED DIRECTLY.
4. ITERATION vs. RECURSION (DOES THE SAME THING)
    1. RECURSION MAY BE SIMPLER, MORE INTUITIVE
    2. RECURSION MAY BE EFFICIENT FROM PROGRAMMER'S POINT OF VIEW
    3. RECURSION MAY NOT BE EFFICIENT FROM COMPUTER POINT OF VIEW

[ ]:
```
# MULTIPLICATION-RECURSIVE SOLUTION
# MATHEMATICAL INDUCTION REASONING OF THE CODE
def mult(a,b):
    if b == 1: # BASE CASE
        return a
    else: # RECURSIVE STEP
        return a + mult(a,b-1)

# FACTORIAL
def fact(n):
    if n==1:
        return 1
    else:
        return n*fact(n-1)
```

[ ]:
```
# TOWERS OF HANOI (THINK RECURSIVELY! )
# SOLVE A SMALLER PROBELM/ SOLVE A BASIC PROBLEM
```

```
# fr: from stack (INITAL TOWER); to: to stack (FINAL TOWER); spare: (SPARE␣
 ↪TOWER);
def printMove(fr,to):
    print('move from'+str(fr)+'to'+str(to))

def Towers(n,fr,to,spare):
    if n==1:
        printMove(fr,to)
    else:
        # WE CAN HAVE MULTIPLE RECURSIVE CALLS INSIDE A FUNCTION
        # THINK IT RECURSIVELY
        Towers(n-1,fr,spare,to) # move the stack size of n-1 to a spare disc
        Towers(1,fr,to,spare) # move the bottom to the desired disc
        Towers(n-1,spare,to,fr) # move the n-1 back to the desired disc

print(Towers(3,'P1','P2','P3'))
```

[ ]:
```
# RECURSION WITH MULTIPLE BASE CASES
# FIBONACCI NUMBERS
def fib(x):
    """assumes x an int >=0, returns Fibonacci of x """
    if x == 0 or x ==1: # base cases
        return 1
    else:
        return fib(x-1)+ fib(x-2) # we have two recurisve functions calls in a␣
 ↪return
```

[ ]:
```
# RECURSION ON NON-NUMERICS (STRINGS)
def isPalindrome(s):
    def toChars(s): # convert string to all lower cases
        s = s.lower() # convert to lower case
        ans = ''
        for c in s: # remove all the punctuations/space
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s)<= 1: # recursive base case
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1]) # recursive step happens␣
 ↪here
                                                    # we compare the fisrt and␣
 ↪last letter then
                                                    # we convert the problem to␣
 ↪a smaller probelm
```

```
    return isPal(toChars(s))
```

2.6 Modules

A module is a `.py` file containing python definitions and statements

1. `import circle` (import a module), we can then use `circle.pi` to access a attribute/method inside a module
2. `from circle import *`
    1. from the module `circle` import everything
    2. we can then directly use `pi` variable defined in cirle without suing `circle.pi`
    3. Need to make sure no name collision when use importing method
3. `from circle import pi` (equivalent to `import module_name.member name`)
4. `import numpy as np` (import **module** as **another name**)

[ ]:
```
# the file circle.py contains
pi = 3.14159
def area(radius):
    return pi*(radius**2)
def circumference(radius):
    return 2*pi*radius

# then we can import and use this module
import circle
pi = 3   # can still define the pi in the shell
print(pi) # 3
print(circle.pi) # 3.14159, look for pi defined in the module
print(circle.circumference(3)) # 18.84953999

# if we don't want to refer to functions and vars by their module, and the␣
 ↪names don't
# collide with other bindings, then we can use:
from circle import* # means from the module, import everything (denoted by the␣
 ↪star sign)
print(pi)
print(area(3)) # we can refer them by calling their own name
```

2.7 Files

Python provides an operating-system independent means to access files, using a file handle.

`nameHandle = open('kids','w')` (open kids for write, r for read)

`name = input('Enter name: ')`

`nameHandle.write(name+ '\n')`

`nameHandle.close()`, (() means that we are referring to the close() function)

[ ]:
```
"""
WRITE/READ FILES
"""
nameHandle = open('kids','w') # 'kid': name of file; w: write command
for i in range(2):
    name = input('Enter name: ')
    nameHandle.write(name+ '\n')
nameHandle.close()

nameHandle = open('kids','r') # read
for line in nameHandle:
    print(line)
nameHandle.close()
```

3 Tuple/List/Mutability/Cloning/Dict

This Chapter introduces the concept of `tuple/list/dict` and `cloning/mutability`

- Tuples
- Lists
- Mutation, Aliasing, Cloning
- Functions as Objects
- Strings, Tuples, Ranges, Lists
- Dictionaries
- Global Variables

3.1 Tuples

1. `Immutable` same as strings (cannot change element values)
2. an ordered sequence of elements, can mix element types
3. represented with parentheses `()`
4. use index `t[0]` to access element, slice tuple as `t[1:2]`, `t[1]=2` gives error (cannot modify object)
5. `(x,y) = (y,x)` ease to swap variable values
6. used to return more than more value in a function
7. nested tupes. we can iterate over the **tuples** like **range/string**

[ ]:
```
te = () # empty tuple
t = (2,"one",3) # () indicates a tuple type
print(t)
ts = (5,) #When a tuple has only one element, you must specify with a comma
print(ts)
print(len(t)) # returns 3
print(t[0]) # evaluates to 2, using [] to index, we cannot change element␣
 ↪inside tuple, t[1] = 4 not work
print((2,"one",3)+(5,6)) # concatenate two tuples, evaluates (2,"one",3,5,6)
```

```
print(t[1:2]) # slice tuple, ("one",) also give a comma which tells us it's a␣
 ↪tuple
x = (1, 2, (3, 'John', 4), 'Hi') # a tuple inside a tuple (nested tuples)
print(x)
print(x[2][2]) # returns int 4, double index
print(x[-1][-1]) # returns string 'i', tricky one

# CONVENIENTLY USED TO SWAP VARIABLE VALUES (This is good)
x,y = 1,2
(x,y) = (y,x) # Swap varibles
print('x is '+ str(x)+', y is ' + str(y))
```

[ ]:
```
# USED TO RETURN MORE THAN ONE VALUE FROM A FUNCTION
def quotient_and_remainder(x,y):
    q = x//y
    r = x%y
    return(q,r)

(quot,rem) = quotient_and_remainder(4,5)
print('quotient is '+ str(quot) + ', remainder is '+str(rem))
```

[ ]:
```
# CAN ITERATE OVER TUPLES
def get_data(aTuple): # aTuple is a tuple of tuples
    nums = () # empty tuples
    words = ()
    for t in aTuple: # interate over a tuple
        nums = nums + (t[0],)
        if t[1] not in words:
            words = words + (t[1],)
    min_nums = min(nums) # find the min value inside the integer tuples
    max_nums = max(nums)
    unique_words = len(words) # return number of unique tuples
    return (min_nums,max_nums,unique_words)

(small,large,words) = get_data(((1,'mine'),(3,'yours'),(5,'ours'),(7,'mine')))
print('min is '+str(small)+', max is '+str(large)+', uniqe words is␣
 ↪'+str(words))
```

3.2 Lists

1. ordered sequence of information, accessible by index, denoted by square brackets `[]`
2. list is `mutable`, elements can be changed e.g. `L[1]=5`
3. we can iterate over lists
4. a list contains elements
    1. usually homogeneous (i.e., all integers)
    2. can contain mixed types (not common)

[ ]:
```
# Indices and ordering
a_list = [] # empty list, index starts from 0
b_list = [2,'a',4,True] # mixed element type
L = [2,1,3] # uniform element type
len(L) # 3
L[0] # 2
print(L[2]+1) # 4
print(L[2]) # 3
# L[3] # error

# CHANGING ELEMENTS (MUTABLE)
L[1] = 5 # L is now [2,5,3]
print(L)

# ITERATING OVER LIST, LIKE STRINGS, CAN ITERATE OVER LIST ELEMENTS DIRECTLY
# NOTE: INDEX 0 TO len(L)-1, range(n) GOES FROM 0 to n-1
total = 0
for i in L:
    total += i
print(total)
```

3.2.1 List Operations

[ ]:
```
# OPERATION ON LISTS (ADD)
L1 = [2,1,3]
L1.append(5) # L1 is now [2,1,3,5], only works for list
# lists are python objects, everything in python is an object
# objects have data/methods/functions/, we can access this info by object_name.
 ↪do_something()
L2 = [4,5,6]
L3 = L1 + L2 # [2,1,3,5,4,5,6]  concatenation +
print(L3)
L1.extend([0,6]) # mutated L1 to [2,1,3,5,0,6]
print(L1) # [2,1,3,5,0,6]
print(L3) # [2,1,3,5,4,5,6]

# OPERATION ON LISTS (REMOVE)
del(L1[3]) # delete element at a specific index
print(L1) # [2,1,3,0,6]
L = [2,1,3,6,3,7,0]
L.remove(2) # remove a specific element,[1,3,6,3,7,0]
L.remove(3) # L = [1,6,3,7,0], if an element appears multiple times, only␣
 ↪remove the first instance
del(L[1]) # l = [1,3,7,0]
L.pop() # returns 0 and  mutates L = [1,3,7] (remove element at end of list)
print(L)
```

```
# CONVERT LISTS to STRINGS AND BACK
s = 'I <3 cs' # string
print(list(s)) # returns ['I','', '<','3','','c','s']
print(s.split('<')) # retruns['I ', '3 cs'], splits on spaces if called without␣
 ↪a parameter
L = ['a','b','c'] # list
print(''.join(L)) # returns 'abc' (use join to join a list to string)
print('_'.join(L)) # returns 'a_b_c'

# OTHER LIST OPERATIONS
# MORE
L = [9,6,0,3]
print(sorted(L)) # returns sorted list, does not mutate
print(L)
L.sort() # mutates, L = [0,3,6,9]
print(L)
L.reverse() # mutates L = [9,6,3,0]
print(L)

# More list operations can be found from the link below
import webbrowser
webbrowser.open('https://docs.python.org/3/tutorial/datastructures.html')
```

3.2.2 Loops/Fucntions/range/list

1. range is a special procedure
2. range returns sth that behaves like a tuple! doesn't generate elements at once
3. rather it generates the first element, and provides and iteration method by which subsequent elements can be generated

[ ]:
```
range(5) # equivalent to tuple (0,1,2,3,4)
range(2,6) # equivalent to (2,3,4,5)
range(5,2,-1) # equivalent to (5,4,3)
for var in range(5):
    print(var)
for var in (0,1,2,3,4):
    print(var)
```

3.3 Mutation, aliasing, cloning

1. Important and tricky, Python Tutor is a good tool to sort this out.
2. listes are mutable and they behave differently than immutable types
3. cloning a list, `chill = cool[:]`
4. Nested lists, side effects still possible after mutation (avoid mutation in iteration)

[ ]:
```
# ALIASING
warm = ['red','yellow','orange']
```

```
hot = warm # warm points to exact address, different name, but points the same␣
 ↪thing
hot.append('pink')
print(hot) # returns ['red','yellow','orange','pink']
print(warm) # also returns ['red','yellow','orange','pink']

# if two lists print the same thing, does not mean they are the same structure
cool = ['blue','green','grey']
chill = ['blue','green','grey']

print(cool == chill) # return True, == returns True if the objects refereed to␣
 ↪by the varibales are equal
print(cool is chill) # return False, is returns True if two variables point to␣
 ↪the same object

print(cool) # ['blue','green','grey']
print(chill) # ['blue','green','grey']
chill[2] = 'blue'
print(cool) # ['blue','green','grey']
print(chill) # ['blue','green','blue']
```

[ ]:
```
#CLONG A LIST (Creat a new list and copy evryt element)
cool = ['blue','green','grey']
chill = cool[:] # clone cool to chill (This is recomended!)
chill.append('black')
print(chill) # ['blue','green','grey','black']
print(cool) # ['blue','green','grey']
```

[ ]:
```
# SORTING LISTS
# sort(), mutates the list, returns nothing
# sorted(), does not mutate list, must assign result to a variable
warm = ['red','yellow','orange']
sortedwarm = warm.sort() # note wortedwarm is none type, since sort does not␣
 ↪return anything
print(warm)
print(sortedwarm)
cool = ['grey', 'green', 'blue']
sortedcool = sorted(cool) # sorted returns the sorted version, thus should be␣
 ↪assigned to a variable
print(cool)
print(sortedcool)
```

[ ]:
```
# NESTED LIST, side effects still possible after mutation
warm = ['yellow','orange']
hot = ['red']
brightcolors = [warm]
```

```
brightcolors.append(hot) # list of list
print(brightcolors) # [['yellow','orange'],['red']]

hot.append('pink')
print(hot) # ['red','pink']
print(brightcolors) # [['yellow','orange'],['red','pink']], also mutates

print(hot+warm)
print(hot)
```

[ ]:
```
# MUTATION AND ITERATION
# avoid mutating a list as you are iterating over it
# remove duplicates from two lists
def remove_dups(L1,L2):
    for e in L1:
        if e in L2:
            L1.remove(e)

L1 = [1,2,3,4]
L2 = [1,2,5,6]
remove_dups(L1,L2) # This returns [2,3,4], not [3,4], you cannot iterate the␣
 ↪list while mutating it
                   # Python has an internal counter for the list, say you at 1,␣
 ↪you removed 1,
                   # it actually goes directly to the second element in L1 (3),␣
 ↪and skips 2.
print(L1)

def remove_dups_new(L1,L2): # This is the correct way to implement
    L1_copy = L1[:]  # we use clones
    for e in L1_copy:
        if e in L2:
            L1.remove(e)

L1 = [1,2,3,4]
L2 = [1,2,5,6]
remove_dups_new(L1,L2)
print(L1)
```

3.4 Functions as Objects

1. class object, can pass function as arguments of another function
2. can process a function operation on each element in a list
3. list of functions, pass a lost as an argument to a function
4. `map(abs,[1,-2,3,-4])`, a general purpose (in the sense of **iterable**) of **high-order-programming (HOP)**.

5. `map` is the key word. HOP is **useful** in analyzing high-dimensional data.

```
[ ]: # functions: have types
# particularly useful to use function as arguments when coupled with lists␣
 ↪(higher order programming)
def applyToEach(L,f):
    """assumes L is a list, f a function,
        mutates L by replacing each element, e, of L by f(e)"""
    for i in range(len(L)):
        L[i]=f(L[i])

L = [1,-2,3.4]
applyToEach(L,abs) # L = [1,2,3.4]
print(L)
applyToEach(L,int) # L = [1,2,3]
print(L)
```

```
[ ]: # LIST OF FUNCTIONS
def applyFuns(L,x):
     """L is a list of functions, x is argument"""
     for f in L:
         print(f(x))
applyFuns([abs,int],4) # 4 4
```

```
[ ]: # GENERALIZATION OF HOPS, map (produces an iterable, so need to walk down it)
for elt in map(abs,[1,-2,3,-4]): # simple form-a unary function and a␣
 ↪collection of suitable arguments
    print(elt) # map gives you a struture acts like a list, but in a way that␣
 ↪you have to iterate to
               # get all the vlaues, 1,2,3,4

L1 = [1,28,36]
L2 = [2,57,9]
for elt in map(min,L1,L2): # general form- an n-ary function and n collections␣
 ↪of arguments
    print(elt) # 1,28, 9
```

3.5 Strings, Tuples, Ranges, Lists

1. Only list is `mutable`
2. Common operations: `acces element/ lengh/ concatenation (not range)/ repeats(not range)/ slice/ in/ not in/ interate`
3. FOUR DIFF WAYS TO COLLECT THINGS TOGETHER INTO COMPOUND DATA STRUTURES `strings/tuples/ranges/lists`

```
[ ]: # COMMON OPERATIONS
seq, seq1, seq2 = 'example', 'example1', 'example2'
```

```
i = 1
print(seq[i]) #  i^th element of sequence
print(len(seq)) # length of sequence
print(seq1+seq2) # concatenation of sequences (not range)
n=2
print(n*seq) # sequence that repeats seq n times (not range)
# seq[start:end] # slice of sequence
print('e' in seq) # True if e contained in sequence
print('e' not in seq) # True if e is not contained in sequence
for e in seq: # iterates over elements of sequence
    print(e)

# PROPERTIES
#Type   Type of elements  Example           Mutable
#str    characters        '','a','abc'      No
#tuple  any type          (),(3,),('abc',4) No
#range  integers          range(10)         No
#list   any type         [],[3],['abc',4]   Yes
```

3.6 Dictionaries

1. Nice to index item of interest directly (store students grades, one data structure, no separate list)
2. Similar to cell array in MATLAB
3. Store pairs of data: `{key: value}` , e.g. `Grades = {''Ana':'B','John':'A+'}`
4. `Grades['John'] = 'A+'`, looks up the key and returns the value
5. Dictionaries are `mutable` data structures
6. `values` in dictionary: can be **any type**. `keys`: must be **unique**, **immutable type** (int, float, string, tuple, bool)

[ ]:
```
# messy if have a lot of diff info to keep track of
def get_grade(student,name_list,grade_list,course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list [i]
    return (course,grade)

# A better way and cleaner way - A dictionary
# nice to index (custom index) item of interest directly (not always int)
# key, value
my_dict = {} # cell array in Matlab
grades = {'Ana':'B','John':'A+','Denise':'A','Katy':'A'}
print(grades['John']) # returns 'A+'
```

3.6.1 Dictionary Operations

[ ]:
```
# DICTIONARY OPERATIONS
grades = {'Ana':'B','John':'A+','Denise':'A','Katy':'A'}
grades['Sylvan'] = 'A' # add an entry, only works in dict
print('John' in grades) # returns True, test to see if a key is in the␣
 ↪dictionary
print('Daniel' in grades) # returns False
del(grades['Ana']) # can remove an entry

print(grades.keys()) # grdaes.key is a method, need type () to call the method,␣
 ↪['John', 'Denise', 'Katy'] (returns an iterable list)
print(grades.values()) # ['A+', 'A', 'A'] (returns an iterable list)
grades2 = grades.copy() # copy the dictionary
print(grades.get('Huang',0)) # The safe way to get value from key 'Huang', if␣
 ↪'Huang' is
                     # not a key, then it returns 0

# values in dictionary: can be any type
# keys: must be unique, immutable type (int,float,string,tuple,bool, careful␣
 ↪with float keys)
# a list cannot be a key in dictionaies since lists are mutable in python
d = {1:{1:0},(1,3):"twelve",'const':[3.14,2.7,8.44]}

# list vs dict
# lsit: ordered sequence of elements,look up by an integer index, indices have␣
 ↪an order, index is an integer
# dict: matches "keys" to "values", look up one item by another item, no order␣
 ↪is guaranteed, key can be any immutable type.
```

3.6.2 Three Function to Analyze Song Lyrics

[ ]:
```
"""
EXAMPLE: 3 FUNCTIONS TO ANALYZE SONG LYRICS
"""
# CREATE A FREQ DICTIONARY MAPPING str:int
def lyrics_to_frequencies(lyrics):
    myDict = {}
    for word in lyrics: # iterate over the list
        if word in myDict: # if the word is in the dictionary
            myDict[word] += 1 # increase the value associated with it by 1
        else:
            myDict[word] = 1
    return myDict # returns a dictionary

# FIND WORD THAT OCCURS THE MOST AND HOW MANY TIMES
# 1. use a lsit, in case there is more than one word
```

```
# 2. return a tuple(list,int) for (words_list,highest_freq)
def most_common_words(freqs): # freqs is a dictionary
    values = freqs.values() # all ints  note it is a special type not a list
    if values: # find the maxium if values is not empty
        best = max(values)
    else:
         best = 0
    words = []
    for k in freqs: # can iterate over keys in dictionary
        if freqs[k]==best: # is the value is the best
            words.append(k) # append works for list only
    return (words,best) # returns a tuple


# FIND THE WORDS THAT OCCUR AT LEAST X TIMES
# let user choose "at least X times", return a lsit of tuples, each tuple is a␣
 ↪(list,int)
# containing the list of words ordered by their frequency
# IDEA: from song dictionary, find most frequent word, delete most common word,␣
 ↪repeat.
def words_often(freqs,minTimes):
    result = []
    done = False # an initial flag
    while not done:
        temp = most_common_words(freqs)
        if temp[1]>= minTimes: # do this untile the most common words appear␣
 ↪leass than minTimes
            result.append(temp)
            for w in temp[0]: # temp[0] is a list defined as words in␣
 ↪most_common_words function
                del(freqs[w]) # can directly mutate dictionary; makes it easier␣
 ↪to iterate
        else:
            done = True
    return result

lyrics = ['I','love','you','I','love','you','I']
freqs = lyrics_to_frequencies(lyrics)
freqs_copy = freqs.copy() # get a copy of the original dicitonary so that we␣
 ↪will not change the original one

print(words_often(freqs_copy,1))
```

3.6.3 Fibonacci with a Dictionary

1. Efficient , can store the computed fab number in a dict
2. Do a lookup first in case already calculated the value

3. Modify dictionary as progress through function cal (good for fft algorithm)

```
[ ]: """
FIBONACCI AND DICTIONARIES (VERY EFFICIENT)
GLOBAL VARIABLES/ TRACKING EFFICIENCY
"""
# ORIGINALLY, WE HAD THIS RESURSIVE FUNCTION
# TWO BASE CASES, CALL ITSELF TWICE, INEFFICIENT
def fib(n):
    global numFibCalls # global variable, we can access outside of the function
    numFibCalls += 1
    if n == 1:
        return 1
    if n == 2:
        return 2
    else:
        return fib(n-1)+fib(n-2)

# INSTEAD OF RECALCULATING THE SAME VALUES MANY TIMES
# WE COULD KEEP TRACK IF ALREADY CALCULATED VALUES (FIBONACCI WITH A DICTIONARY)
# USING A DICTONARY TO HOLD ON THE VALUES I HAVE ALREADY CALCULATED
def fib_efficient(n,d):
    # d is a base dictionary
    global numFibCalls # accessible from outside scope of function
    numFibCalls += 1
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1,d) + fib_efficient(n-2,d)
        d[n] = ans # strore the ans in a dictinary
        return ans

numFibCalls = 0
fibArg = 12

print(fib(fibArg))
print('function calls', numFibCalls)

numFibCalls = 0
d = {1:1,2:2} # base cases in a dictionary, memoization: create a memo for␣
 ↪youself
print(fib_efficient(fibArg,d))
print('function calls', numFibCalls)
print(d) # the base dicironary is updated
```

3.7 Global Variable

1. Accessible from outside scope of fucntion, can be dangerous to use

2. But can be convenient when want to keep track of info inside a function
3. The key word is `global`, e.g. `global numFibCalls` as in the fibonacci example above

4 Testing/Debugging/Exception/Assertion

- Exceptions
- Assertions

[ ]:
```
"""
TESTING, DEBUGGING
"""

"""
ERROR MESSAGES-EASY
"""
# IndexError, test = [1,2,3] then test[4]
# TypeError, int(test), '3'/4 (mixing data types)
# NameError, referencing a non-existent variable
# SyntaxError, a = len([1,2,3] (forgetting to close parenthesis, quotation, etc)
# IOError: IO system reports malfunction (e.g. file not found)
# AttributeError: attribute reference fails

"""
LOGIC ERRORS-HARD
"""
# think
# draw pictures
# explain the code to someone else /rubber ducky
```

4.1 Exceptions

1. What happens when procedure execution hits an unexpected condition (`SyntaxError/NameError/AttributeError/TypeError/ValueError/IOError`)
2. keywords: `try:, except:, else:, finally:, raise:`
3. Handling specific exceptions: `except ValueError`

[ ]:
```
# get an exception... to what was expected
# what to do with exceptions?
#  DEALING WITH EXCEPTIONS

# try to execute each of the instuctions in turn
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number"))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
# if a exception is raised, jump to here
```

```
except ValueError:  # separate except clauses to deal with a particular type of␣
 ↪exception
    print("Could not convert to a number")
except ZeroDivisionError:
    print("Can't divide by zero")
except: # for all other errors
    print("Something went very wrong.")

# OTHER EXCEPTIONS
# else: body of this is excuetd when execution of associated try body completes␣
 ↪with no exceptions
# finally: body of this is always executed after try, else and except clauses,␣
 ↪even if they raised
        # another error or executed a break, continue or return
        # useful for clean-up code that should be run no matter what else␣
 ↪happened
        # e.g. close a file
```

4.1.1 Exception Usages

[ ]:
```
"""
EXAMPLE EXCEPTION USAGE
"""
# 1st example
# Loop only exits when correct type of input provided
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError: # handles ValueError
        print("Input not an integer; try again")
print("Correct input of an integer!")

# 2nd example
# Contorl input
data = []
file_name = input("Provide a name of a file of data ")

try:
    fh = open(file_name,'r')
except IOError:
    print('cannot open', file_name)
else:
    for new in fh: # reading a new line
        if new != '\n':
```

```
            addIt = new[:-1].split(',') # remove trailing \n
            data.append(addIt)
    fh.close() # close file even if fail

gradesData = []
if data: # as long as got some data
    for student in data: # loop through the data
        try:
            name = student[0:-1]
            grades = int(student[-1]) # gives a vlaueError if the last element␣
 ↪is not number
            gradesData.append([name,[grades]])
        except ValueError:
            gradesData.append([student[:],[]])
```

4.1.2 Exception as Control Flow

[ ]:
```
"""
EXCEPTION AS CONTROL FLOW
"""
# WE CAN RAISE AN EXCEPTION WHEN UNABLE TO PRODUCE A RESULT CONSISTENT WITH␣
 ↪FUNCTIONS'S SPECIFICATION
def get_ratio(L1,L2):
    """ Assumes: L1 and L2 are lists of equal length of numbers
        Returns: a list containing L1[i]/L2[i]"""
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/float(L2[index]))
        except ZeroDivisionError:
            ratios.append(float('NaN')) # NaN = not a number
        except: # manage flow of program by raising own error
            raise ValueError('get_ratios called with bad arg')
    return ratios

# ANOTHER EXAMPLE
# GET A NEW LIST WITH AVERAGE MARKS
test_grades = [[['peter','parker'],[80.0, 70.0, 85.0]],[['bruce','wayne'],[100.
 ↪0, 80.0, 74.0]],
               [['captain','america'], [8.0,10.0,96.0]],[['deadpool'],[]]]

def get_stats(class_list):
    new_stats = []
    for elt in class_list:
        new_stats.append([elt[0],elt[1], avg(elt[1])])
```

```
    return new_stats

def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('no grades data') # no return for excetion, it actually assigns []
        return 0.0
```

4.2 Assertions

1. want to be sure that assumptions on state of computation are as expected
2. use an `assert statement` to raise an `AssertionError` exception if assumptions not met (functions end immediately if assertion not met)
3. an example of good defensive programming (make it easier to locate a source of a bug)
4. Keywords: `assert not len(grades)==0, 'no grades data'`

[ ]:
```
"""
ASSERTIONS (GOOD WAY OF DOING DEFENSIVE PROGRAM)
"""
# Prevent circumstances from leading to unexpected results
# Ensure that execution halts whenever an expected conditons not met
# typically used to check inputs to fucntions procedures, but can be used␣
 ↪anywhere
# can be used to check outputs of a function to avoid propagating bad values
def avg(grades):
    # function ends immediately if assertion not met
    assert not len(grades) == 0, 'no grades data'
    return sum(grades)/len(grades)
grades = [1.1,3.3]
print(avg(grades))
```

5 Object Oriented Programming

This Chapter introduces the concepts of object oriented programming (OOP) in Python. Everything in Python is an object and has a type. e.g. `1234` is an instance of an `int`, `a='hello'`, `a` is an instance of a string. Advantages of OOP include 1. bundle data into packages together with procedures that work on them through well-defined interfaces 2. divide-and-conquer development 3. classes make it easy to reuse code

- Classes
- Special Object Operators
- Getters and Setters
- Class Hiearchies
- Instance Variables and Class Variables
- A Complete Example (MIT Students)

- A complete Example (Gradebook)
- Generators

5.1 Basic Classes

1. Creating a class: define `name/attribute`. Keyword is `class`.
2. `Attributes` are **data** and **procedures (method)** associated with a class
3. `__init__(self,x,y)`, similar to constructors in C++
4. `print(c.distance(origin))` and `print(Coordinate.distance(c,origin))` are equivalent
5. Print representation of an object, `__str__` method
6. `isinstance(c,Coordinate)` check if c is an object of the class Coordinate

[ ]:
```
# ATTRIBUTES: data and procedures that belong to the class
class Coordinate(object): # In python, object is the superclass of Coordinate␣
↪class
    # Typically the fsirt function in a class
    def __init__(self,x,y): # double underscore, self->instance, this func␣
↪calls when you invoke
        self.x = x          # creation of an instance, self automatically␣
↪points the instance
        self.y = y

    # Methods are procudeure attributes that can manipulate the data attributes
    def distance(self,other): # other is another instance
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq+y_diff_sq)**0.5

    # Python calls the __str__ method when used with print on you class object
    # You choose what it does! e.g. print(c) -> <3,4>
    def __str__(self): # override internal print
        return "<"+str(self.x)+","+str(self.y)+">" # Must reetrun a string  for␣
↪this special method

    # Other special methods
    # https://docs.python.org/3/reference/datamodel.html#basic-customization
    def __sub__(self,other): # override the interal substraction for instances␣
↪in this class
        return Coordinate(self.x-other.x,self.y-other.y) # create a instance␣
↪(subtraction of two instances create a new instance)

    # eval(repr(c)) == c
    # Return a string containing a printable representation of an object (this␣
↪is typically used for debugging)
    # a string that looks like a valid Python expression that could be used to
    # recreate an object with the same value
    def __repr__(self):
```

```
        return "Coordinate({},{})".format(self.getX(),self.getY())

c = Coordinate(3,4) # create a new instance, c frame , like functions
origin = Coordinate(0,0) # creat another instance, origin fram

# These two calling methods are equivalent
print(c.distance(origin)) # print the distance to another instance, c is a frame
print(Coordinate.distance(c,origin)) # equivalent to aobve but in a different way

# PRINT REPRESENTATION OF AN OBJECT
print(c) # <3,4>, you control what prints out from an object instance
print(type(c)) # <class '__main__.Coordinate'>, prints the type of an object instance
print(Coordinate, type(Coordinate)) # <class '__main__.Coordinate'> <class 'type'>
print(isinstance(c,Coordinate)) # check if an object an instance of a class
print(c-origin) # <3,4>, object subtraction
```

5.2 Special Object Operators

1. `+,-,==,<,>,len(),print`, more information can be found here
2. Like `print`, we can override these to work with our class
3. Define them with double undersocres before/after
    1. `__add__` (self, other) self + other
    2. `__sub__`(self, other) self - other
    3. `__eq__`(self, other) self == other
    4. `__lt__`(self, other) self < other
    5. `__len__`(self) len(self)
    6. `__str__`(self) print(self)

5.3 Getters and Setters

1. we do not want to directly manipulate the attributes associated with those instances. In stead, we use **getters** and **setters**.
2. Good practice (for information hiding and avoiding potential bugs) to use **getters** (getting attributes) and **setters** (setting attributes)

[ ]:
```
"""
EXAMPLE: FRACTIONS
"""
class fraction(object):
    def __init__(self,numer,denom):
        self.numer = numer
        self.denom = denom
```

```
    def __str__(self):
        return str(self.numer)+'/'+str(self.denom)

    # getters, but why we use them? can be just directly use self.numer?
    # The reason is that we do not want to directly manipulate the attributes␣
 ↪associated with
    # those instances
    def getNumer(self):
        return self.numer
    def getDenom(self):
        return self.denom

    # objects addition
    def __add__(self,other): # use () to indicate a method
        numerNew = other.getDenom()*self.getNumer()+ other.getNumer()*self.
 ↪getDenom()
        denomNew = other.getDenom()*self.getDenom()
        return fraction(numerNew,denomNew)

    # objects subtraction
    def __sub__(self,other):
        numerNew = other.getDenom()*self.getNumer()- other.getNumer()*self.
 ↪getDenom()
        denomNew = other.getDenom()*self.getDenom()
        return fraction(numerNew,denomNew)

    def convert(self):
        return self.getNumer()/self.getDenom()

# initialize objects
oneHalf = fraction(1,2)
twoThirds = fraction(2,3)

print(oneHalf) # -> 1/2
print(twoThirds) # -> 2/3
print(oneHalf.getNumer())

new = oneHalf + twoThirds
print(new.convert()) # -> 7/6
```

[ ]:
```
"""
EXAMPLE: A SET OF INTEGERS
"""
class intSet(object):
    def __init__(self):
        self.vals = [] # empty list
```

```
    # Insert new integers to the our list , e is input argument
    def insert(self,e):
        if not e in self.vals:
            self.vals.append(e)

    # Test to see is e is our list
    def member(self,e):
        return e in self.vals

    # Try to remove e from our list
    def remove(self,e):
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e)+' not found') # raise my own particular␣
↪error

    # return an intersect instance, other is an instance here
    # return the intersection of two interger sets
    def intersect(self, other):
        result = intSet() # result is a new instance
        for i in self.vals:
            if other.member(i):
                 result.insert(i)
        return result

    # modify the print()
    def __str__(self):
        self.vals.sort()
        #  return '{' + ','.join([str(e) for e in self.vals]) + '}' does the␣
↪same job
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{'+ result[:-1] +'}'

    # modify the len()
    def __len__(self):
        return len(self.vals)

s = intSet()
print(s) # {}
s.insert(3)
s.insert(4)
print(s) #{3,4}
s.insert(3)
print(s.member(3)) # True
```

```
print(s.member(6)) # False
s.remove(3)
print(s) # {4}
# s.remove(3) # ValueError, 3 does not exist
```

[ ]:
```
"""
ANOTHER EXAMPLE - ANIMALS
"""
class Animal(object):
    def __init__(self,age):
        self.age = age
        self.name = None # define other data attribute even we dont pass them into an instance

    # getters
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name

    # setters, can change the binding for attributes
    def set_age(self,newage):
        self.age = newage
    def set_name(self,newname=""): # default argument is an empty string
        self.name = newname

    # special print
    def __str__(self):
        return "animal: "+str(self.name)+":"+str(self.age)

myAnimal = Animal(3)
myAnimal.set_name('foobar')
# why we want to use getters and setters:
print(myAnimal.age) # directly access the attribute
print(myAnimal.get_age()) # use the method and calling to access the attirbute (better)
                  # becasue we want to seperate the internal representation from access to that
                  # representation (it's called information hiding)

print(myAnimal) # animal: foobar: 3
```

[ ]:
```
"""
EXAMPLE-WHY USING GETTERS AND SETTERS
"""
# use a.get_age() NOT a.age: good style, easy to maintain code, prevent bugs
```

```
# always write a method to store the attributes inside it (setters()), to␣
 ↪access using getters()
class Animal(object):
    def __init__(self,age):
        self.years = age # class definition changes, self.age -> self.year (can␣
 ↪produce bugs if access age using myAnimal.age)
    def get_age(self):
        return self.years
```

5.4 Class Hiearchies

1. Parent class (superclass)/ child class (subclass)
2. Child class inherits all data and behaviors of parent class (But can add more info)
3. Can also add more behavior and override some behavior
4. `Class Cat(Animal):`, **cat** inherits from **Animal class**, a class heirachi tree is given in the figure shown below. Note that `object` is the superclass for all Python classes.

```
![A class heirarchi tree](images/class_heirarchic.png)
```

5. for an instance of a class, look for a method name in current class definition. if not found, look for method name up the hierarchy (in parent, then grandparent, and so on). use first method up the hierarchy that you found with that method name.

[ ]:
```
"""
HIERARCHIES
parent class (superclass), child class (subclass)
e.g. animal-> person, cat, rabbit. persion-> student
"""

class Cat(Animal): # inherits all attributes of Animal
    def speak(self): # add new functionality via new methods
        print("meow")
    def __str__(self): # overrides __str__ from Animal
        return "cat:"+str(self.name)+":"+str(self.age)

class Rabbit(Animal):
    def speak(self): # Note here, __init__ is not missing, Rabbit class just␣
 ↪uses the Animal Class's version
        print("meep")
    def __str__(self):
        return "rabbit"+str(self.name)+":"+str(self.age)

class Person(Animal):
    def __init__(self,name,age):
        Animal.__init__(self,age) # explicitly call Animal constructor, in the␣
 ↪superclass (self.age = age and self.name = None)
        Animal.set_name(self,name) # call Animal's method to change the name␣
 ↪associated with (self.name = name)
```

```
                                    # instance of a person
        self.freinds = [] # add a new data attribute (empty list)

    # define new methods (getter/setter)
    def get_friends(self):
        return self.friends
    def add_friend(self,fname):
        if fname not in self.friends:
            self.friends.append(fname)

    def speak(self):
        print("hello")

    def age_diff(self,other): # new method to give age diff in a user friendly␣
 ↪way
        diff = self.get_age()-other.get_age()
        if self.age > other.age:
            print(self.name,"is",diff,"years older than", other.name)
        else:
            print(self.name,"is",-diff,"years younger than",other.name)

    def __str__(self): # override Animal's __str__method
        return "person:"+str(self.name)+":"+str(self.age)


jelly = Cat(1)
jelly.set_name("JellyBelly")
print(jelly) # print cat: JeLLyBelly:1
# we can explicilty recover the underlying Animal method by
print(Animal.__str__(jelly)) # animal: JellyBelly:1, remeber Animal is the␣
 ↪super class here

eric = Person("eric",45)
john = Person("john",55)
eric.speak()
eric.age_diff(john)
```

[ ]:
```
"""
ANOTHER SUBSUBCLASS EXAMPLE Animal->Person->Student
"""
import random

class Student(Person):
    def __init__(self,name,age,major = None):
        Person.__init__(self,name,age) # call the person constructor
        self.major = major # student has a new attribute data (major)
```

```
    # change major method
    def change_major(self,major):
        self.major = major

    # override the speak method from Person class
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r <= 0.5:
            print("i need sleep")
        elif 0.5 <= r <= 0.75:
            print("i should eat")
        else:
            print("i am watching tv")

    # Override the __str__ method
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)

eric = Person('Eric',45)
fred = Student('Fred',18,'Course VI')
print(fred) # student:Fred:18:Course VI
print(Person.__str__(fred))
fred.speak()
fred.speak()
```

5.5 Instance Variables and Class Variables

1. `Instance variables` are specific to an insatnce (created for each instance, belongs to an instance)
2. `Class variables` are defined inside classes but outside any class methods (also outside of `__init__`). They are shared among all objects/instances of that class.

[ ]:
```
"""
CLASS VARIABLES
"""
# define inside the class definition, but outside of any of the methods
# shared among all objects/instances of that class

class Rabbit(Animal): # parent class
    # class variable
    tag = 1
    # Constrcutor
    def __init__(self,age,parent1 = None, parent2 = None):
        Animal.__init__(self,age)
```

```
        self.parent1 = parent1 # here parent1 and parent2 are two Rabbit␣
↪instances
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1 # incremeting class variable and changes it for all␣
↪instances that
                        # may reference it
    def get_rid(self):
        return str(self.rid).zfill(3) # method on string to pad the beginning␣
↪with zeros
    def get_parent1(self):            # e.g. 001 not 1
        return self.parent1
    def get_parent2(self):
        return self.parent2

    # Special class operations
    def __add__(self,other):
        return Rabbit(0,self,other) # modify the addition to create new instance
    def __eq__(self,other): # special method to compare two rabbits
        parents_same = self.parent1.rid == other.parent1.rid \
                       and self.parent2.rid == other.parent2.rid
        parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
        return parents_same or parents_opposite

peter = Rabbit(2) # rid = 1
peter.set_name('Peter')
hopsy = Rabbit(3) # rid = 2
hopsy.set_name('Hopsy')
cotton = Rabbit(1,peter,hopsy) # here peter and hopsy are instances, rid = 3
cotton.set_name('Cottontail')
mopsy = peter + hopsy # rid = 4
mopsy.set_name('Mopsy')

# Note,  __str__  method is defined in the Animal class
print(mopsy.get_parent1()) #animal: Peter:2,
print(mopsy.get_parent2()) # animal: Hopsy:3
print(mopsy==cotton) # True
```

5.5.1 Tips

- zfill() is a built-in `string` operation in python.
- \ is the explicit line break in Python

5.6 A Complete Example (MIT Students)

Explore in some detail an example of building an application that organizes info about people

```
![MIT students hierarchies](images/mit_student.PNG)
```

5.6.1 Bulding a Class

- start with a Person object
  - Person: name, birthday
  - get last name
  - sort by last name
  - get age

[ ]:
```
import datetime

class Person(object):
    def __init__(self,name):
        self.name= name
        self.birthday = None
        self.lastName = name.split(' ')[-1] # name is a string, so split into a␣
 ↪list of strings
                                            # based on spaces
    def getLastName(self):
        return self.lastName

    def setBirthday(self,month,day,year):
        self.birthday = datetime.date(year,month,day)

    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return (datetime.date.today()-self.birthday).days # convert into days/
 ↪years

    def __lt__(self,other): # lt = less than, sort people by last name
        if self.lastName == other.lastName: # if have same lastname, we sort␣
 ↪them by their
            return self.name < other.name   # fullname
        return self.lastName < other.lastName

    def __str__(self): # override the print method
        return self.name

# Initialize instances
p1 = Person('Mark Zuckerberg')
p1.setBirthday(5,14,84)
p2 = Person('Drew Houston')
p2.setBirthday(3,4,83)
p3 = Person('Bill Gates')
p3.setBirthday(10,28,55)
p4 = Person('Andrew Gates')
```

```
p5 = Person('Steve Wozniak')
#
personList = [p1,p2,p3,p4,p5] # is a list with each element a person object
# print them in order
for e in personList:
    print(e)
# sort the names in order
personList.sort() # sort() method in list use the __lt__ special class method
for e in personList:
    print(e)
```

5.6.2 Use Inheritance

- MITPerson: Person + ID Number
  - assign ID numbers in sequence
  - get ID number
  - sort by ID number

[ ]:
```
"""
USING INHERITANCE
"""
class MITPerson(Person): # Person is the superclass of the MITPerson class

    nextIdNum = 0 # next ID number to assign (class variable)

    def __init__(self,name):
        Person.__init__(self,name) # initilaize Person attributes
        self.idNum = MITPerson.nextIdNum # MITPerosn attribute: unique ID
        MITPerson.nextIdNum += 1 # update the class variable

    def getIdNum(self):
        return self.idNum

    def __lt__(self,other): # sorting them by ID numbers (override the __lt__␣
 ↪in Person class)
        return self.idNum < other.idNum

    def speak(self,utterance):
        return (self.getLastName()+" say: "+ utterance)

m3 = MITPerson('Mark Zuckerberg')
Person.setBirthday(m3,5,14,84)
m2 = MITPerson('Drew Houston')
Person.setBirthday(m2,3,4,83)
m1 = MITPerson('Bill Gates')
Person.setBirthday(m1,10,28,55)
#
```

```
MITPersonList = [m1, m2, m3]

# print outher the mit list
for e in MITPersonList:
    print(e)
# sort them by ID
MITPersonList.sort() # sort from largest to smallest (sort use __lt__ method)
for e in MITPersonList:
    print(e)

p1 = MITPerson('Eric')
p2 = MITPerson('John')
p3 = MITPerson('John')
p4 = Person('John')
#
print(p1 < p2) # = p1.__lt__(p2), True, comared on id numbers (compare use ids)

# use the __lt__ method associated with the type of p1, namely an MITPerson
# print(p1 < p4) # = p1.__lt__(p4), Attribute error, p4 does not have a id␣
 ↪attribute (compare use ids)

# use the __lt__ method associated with the type of p4, namely a Person (the␣
 ↪one that compares based on name
print(p4 < p1) # = p4.__lt__(p1), Flase , p1 has a name attribute, (compared␣
 ↪use names)
```

5.6.3 Add More Classes

- Students: several types, all MITPerson
  - undergraduate student: has class year
  - graduate student
- pass is a special keyword says there is no expression in the body

[ ]:
```
"""
ADD MORE CLASSES
"""
class Student(MITPerson): # better to create a student superclass that covers␣
 ↪all students (Also the class avariable)
    pass # inherits all the attributes from the MITPerson superclass

class UG(Student): # undergraduate student
    def __init__(self,name,classYear):
        MITPerson.__init__(self,name)
        self.year = classYear

    def getClass(self):
        return self.year
```

```
    def speak(self,utterance):
        return MITPerson.speak(self," Dude, "+utterance)

class Grad(Student):
    pass

class TransferStudent(Student):
    pass

def isStudent(obj): # check if is student
    return isinstance(obj,Student)

s1 = UG('Matt Damon',2017)
s2 = UG('Ben Affleck',2017)
s3 = UG('Lin Manuel Miranda',2018)
s4 = Grad('Leonard di Caprio')
s5 = TransferStudent('Robert deNiro')
#
print(s1)
print(s1.getClass())
print(s1.speak('Where is the quiz?'))
print(s2.speak('I have no clue!'))
```

5.6.4 Using Inherited Method

- add a Professor class of objects
  - also a kind of MITPerson
  - but has different behaviors
- Use as an example to see how one can leverage methods from other classes in the hierarchy
- Modularity helps, capture the behavior locally

[ ]:
```
"""
USING INHEREITED METHODS
"""
class Professor(MITPerson):
    def __init__(self,name,department):
        MITPerson.__init__(self,name)
        self.department = department

    def speak(self, utterance): # shadow the MIT speak method, but uses the MIT␣
 ↪speak method
        new = 'In course '+ self.department + ' we say '
        return MITPerson.speak(self,new + utterance)

    def lecture(self,topic):
        return self.speak('it is obvious that '+ topic)
```

```
faculty = Professor('Doctor Arrogant','six')
print(faculty.speak('Welcome'))
print(faculty.lecture('Math'))
```

5.7 A complete Example (Gradebook)

1. create class that includes instances of other classes within it
2. build a data structure that can hold grades for students
3. gather together data and procedures for dealing with them in a single structure, so that users can manipulate without having to know internal details

[ ]:
```
"""
EXAMPLE CLASS: GRADEBOOK
create class that includes instance of other classes within in
"""
class Grades(object):
    def __init__(self):
        self.students = [] # list of student object
        self.grades = {} # maps idNum -> list of grades
        self.isSorted = True # true if self.student is sorted

    def addStudent(self,student): # Here studen is another instance
        if student in self.students:
            raise ValueError('Duplicate student')
        self.students.append(student) # student is a list, we simply append it
        self.grades[student.getIdNum()] = [] # grades is a dict, student is␣
 ↪another instance
                                             # create empty list associated␣
 ↪with the key
                                             # student.getIdNum() from another␣
 ↪instance student
        self.isSorted = False

    def addGrade(self,student,grade):
        try: # student is another instance from other class
            self.grades[student.getIdNum()].append(grade) # is a list, then we␣
 ↪can append
        except KeyError: #  KeyError generally means the key doesn't exist
            raise ValueError('Student not in grade book')

    def getGrades(self,student):
        try:
            # return a copy, which means I can do things on that without␣
 ↪destroying original set
            return self.grades[student.getIdNum()][:]  # a safe thing to do
        except KeyError:
```

```
            raise ValueError('Student not in grade book')

    def allStudents(self): # sort all students
        if not self.isSorted: # if not sorted
            self.students.sort() # sort the list
            self.isSorted = True
        return self.students[:] # return a copy of list of students

    '''
    # a new version with generator
    def allStudents(self):
        if not self.isSorted:
            self.students.sort()
            self.isSorted = True # print(six00.allStudents().__next__())
        for s in self.students: # it gives the next one without generating the␣
↪entire list
            yield s
    '''

    # assume course is a type of Grade instance (equivalent to␣
↪gradeReport(self))
    # This practice is good for information hiding
    def gradeReport(course):
        report = [] # empty list
        for s in course.allStudents(): # loop  over all the students
            tot = 0.0
            numGrades = 0
            for g in course.getGrades(s): # run every grade in the gardes␣
↪associated with that student
                tot += g
                numGrades += 1
            try:
                average = tot/numGrades # report the average
                report.append(str(s)+'\'s mean grade is ' +str(average)) # \' =␣
↪'
            except ZeroDivisionError:
                report.append(str(s)+' has no grades')
        return '\n'.join(report) # join the list with carriage return (join␣
↪list into a string)

# student instance
ug1 = UG('Matt Damon', 2018)
ug2 = UG('Ben Affleck', 2019)
ug3 = UG('Drew Houston', 2017)
ug4 = UG('Mark Zuckerberg', 2017)
g1 = Grad('Bill Gates')
g2 = Grad('Steve Wozniak')
```

```
# six00 is Grade instance
six00 = Grades()
# add students instances from other classes
six00.addStudent(g1)
six00.addStudent(ug2)
six00.addStudent(ug1)
six00.addStudent(g2)
six00.addStudent(ug4)
six00.addStudent(ug3)

# Add students grades
six00.addGrade(g1,100)
six00.addGrade(g2,25)
six00.addGrade(ug1,95)
six00.addGrade(ug2,85)
six00.addGrade(ug3,75)

# add more grades
six00.addGrade(g1,90)
six00.addGrade(g2,45)
six00.addGrade(ug1,80)
six00.addGrade(ug2,75)

# Note here we did not pass any course instance. In this case, the gradeReport␣
 ↪function will just use six00 as its input argument.
print(six00.gradeReport())
print(Grades.gradeReport(six00)) # equivalent to the statement above
```

5.8 Generators

1. any procedure or method with `yield` statement called a **generator**
2. generators have a `__next()__` method which starts/resumes execution of the procedure.
3. Inside of generator:
    1. `yield` suspends execution and returns a value
    2. returning from a generator raises a `StopIteration` exception
4. Generators separates the concept of computing a very long sequence of objects, from the actual process of computing them explicitly. Allows one to generate each new objects as needed as part of another computation.
5. Have already seen this idea in range (`for I in range(4)`, same idea). Think of **iterations** with `range` when use generators.
6. System has some kind of memory associated with it.

[ ]:
```
"""
GENERATORS
solve the probelm - large list of student
"range" is same idea
```

```
"""
# any procedure or method with yield statement called a generator
# generators have a next() method which starts/resumes excution of the procedure
# yield suspends execution and returns a value
# returning from a generator raises a StopIteration
# It lets us known how far i go in the computation before i stop and return a␣
 ↪value
def genTest():
    yield 1
    yield 2

foo =genTest()
print(type(foo))
print(foo.__next__()) #-> 1
print(foo.__next__()) #-> 2
# foo.__next__() # Rasie StopIteration exception

# print out 1, 2 in turn (same as range(1,3))
# generators are useful in doing iterations (computation-efficient, check the␣
 ↪grade example using generators)
for n in genTest():
    print(n)
```

[ ]:
```
# Generate Fibonacci Numbers using generators
def genFib():
    fibn_1 = 1 # fib(n-1)
    fibn_2 = 0 # fib(n-2)
    while True:
        # fib(n) = fib(n-1) + fib(n-2)
        next = fibn_1 + fibn_2
        yield next
        fibn_2 = fibn_1
        fibn_1 = next

fib = genFib()
print(fib.__next__()) # 1
print(fib.__next__()) # 2
print(fib.__next__()) # 3
print(fib.__next__()) # 5

# for n in genFib():
#  print(n)
# will produce all of the Fibonacci numbers (an infinite sequence)
```

6 Algo Complexity, Searching and Sorting

6.1 Algorithm Complexity

constant<log<linear<nlogn<polynomial<expoential<factorial

n^2+2n+2 O(n^2)

log(n)+n+4 O(n)

n^2+10000n+3^10000 O(n^2)

0.0001*n*log(n)+300n O(nlog(n))

2n^30+3^n O(3^n)

[ ]:
```
# HOW TO EVALUATE EFFICIENCY OF PROGRAMS
# timer/count/order of growth(most appropriate way)

# TIMING A PROGRAM
# NOT SO GOOD - VARY DEPENDING ON COMPUTES/ALOGORITHM/IMPLEMENTATIONS
import time # bring that class into your own file

def c_to_f(c):
    return c*9/5 + 32

t0 = time.clock()
c_to_f(100000)
t1 = time.clock()-t0
print("t =",t0,":",t1,"s,")

# COUNTING OPERATIONS
def c_to_f(c):
    return c*9.0/5+32 # 3 ops

# total op = 1+(1+2)n  n-> iterations
def mysum(x):
    total = 0 # 1 op
    for i in range(x+1): # 1 op/ ite
        total += i # 2 ops /ite
    return total
```

[ ]:
```
# ORDER OF GROWTH
# want to evaluate program's efficiency when input is very big
# express the growth of program's run time as input size grows
# upper bound on growth/ don't have to be precise
```

```
# tyoes of orders of growth:␣
 ↪constant<log<linear<nlogn<polynomial<expoential<factorial

# O() is used to describe worst case - > whats is the behavior when as the␣
 ↪problem input size
# get really big

def fact_iter(n):
    """ assumes a an int >= 0, computes factorial """
    answer = 1
    while n>1:
        answer *= n
        n -= 1 # two steps
    return answer

# in toal 1+ 5n + 1 operations
# worst case asymptotic complecity: O(n)
    # ignore additive constants
    # ignore multiplicative constants
# n^2 +2n + 2 -> O(n^2)

for i in range(n):
    print('a')  # O(n)

for j in range(n*n):
    print('b')  # O(n*n)

# O(n) + O(n*n) = O(n+n^2) = O(n^2)
# O(f(n))*O(g(n)) = O(f(n)*g(n))
```

6.1.1 Logrithmic Complexity

- complexity grows as log of size of one of its inputs (very efficient)
- example:
  - bisection search
  - binary search of a list

6.1.2 Linear Complexity

- searching a list in sequence to see if an element is present
- Iterative algorithms, for loops
- Iterative and recursive factorial are the same order of growth

6.1.3 Log-linear Complexity

- many practical algorithms are log-linear
- very commonly used log-linear algorithm is `merge sort`

6.1.4 Polynomial Complexity

- most common polynomial algorithms are quadratic
- commonly occurs when we have nested loops or recursive function calls
- Computes n^2 very inefficiently
- When dealing with nested loops, look at the `ranges`

6.1.5 Exponential Complexity

- very expensive, many programs are exponential (will lead us to consider approximate solutions more quickly)
- recursive functions where more than one recursive call for each size of problem (Towers if Hanoi)

6.2 Search Algorithms

- `Brute force`
- `Bisection` if the list is sorted (O(log n))
- `linear search` can work for both sorted and unsorted lists.

[ ]:
```
"""
USE BISECTION SEARCH
"""
# a breaking and conqure problem
# IMPLEMENTATION 1 O(nlog(n))
def bisect_search1(L,e):
    if L == []: # L is a list, check to see if e in a sorted list L
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half]>e:
            return bisect_search1(L[:half],e) # copy requries O(n) operations
        else:
            return bisect_search1(L[half:],e)

# IMPLEMENTATION 2 (O(log n))
def bisec_search2(L,e):
    def bisect_search_helper(L,e,low,high):
        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 bisect_search_helper(L, e, low, mid-1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)

# SEARCH A  SORTED LIST -- n is len(L)
# Linear search, O(n)
# binary search, O(log n), but assume the list is sorted
# When does it make sense to sort first then search ?
        # SORT + O(log n) < O (n) -> SORT < O(n) - O(log n)
        # when sorting is less than O(n) -> never true !
```

6.3 Sorting Algorithms

- Monkey sort (stupid sort)
- Bubble sort O(n^2)
- Selection sort O(n^2)
- Merge sort (divide and conquer approach O(n*log(n))) is the fastest a sort can be.

6.3.1 Bubble Sort

- compare consecutive pairs of elements
- swap elements in pair such that smaller is first
- when reach end of list, start over again
- stop when no more swaps have been made

[ ]:
```
def bubble_sort(L):
    swap = False
    while not swap:
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

6.3.2 Selection Sort

- extract minimum element and swap it with element at index 0 (1st step)
- in remaining sublist, extract minimum element swap it with the element at index 1 (subsequent steps)
- keep the left portion of the list sorted
  - at ith step, first i elements in list are sorted
  - all other elements are bigger than first i elements

[ ]:
```
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L): # O(N)
        for i in range(suffixSt, len(L)): # O(N)
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt] # swap list values
        suffixSt += 1
```

6.3.3 Merge Sort

- MERGE SORT -> DIVIDE AND CONQUER -> Split list in half until have sublists of any 1 element
- if list is of length 0 or 1, already sorted
- if list has more than one element, split into two lists, and sort each
- merge sorted sublists
  - look at first element of each, move smaller to end of the result
  - when one list empty, just copy rest of other list

[ ]:
```
# Merge Two list in a ordered manner
def merge(left,right):
    result = []
    i, j = 0, 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    while (i < len(left)): # when right sublist is empty
        result.append(left[i])
        i += 1

    while (j < len(right)): # when left sublist is empty
        result.append(right[j])
        j += 1

    return result

# merge sort algorithm (most efficient sort algorithm)
def merge_sort(L):
    if len(L) < 2: # base case
        return L[:]
    else:
        middle = len(L)//2 # divide
```

```
        left = merge_sort(L[:middle]) # call the merge_sort function recursively
        right = merge_sort(L[middle:])
        return merge(left,right) # conquer with the merge step
```

7 Visualization of Data

- Pylab Tutorial
- Pyplot Tutorial
- pylab is a good package for plotting data in Pyhton. matpltlab is also a good one. Note `pylab` uses the `Matlab` plotting style.
- Matplotlib is the whole package; pylab is a module in matplotlib that gets installed alongside matplotlib; and matplotlib.pyplot is a module in matplotlib.
- Pylab combines the pyplot functionality (for plotting) with the numpy functionality (for mathematics and for working with arrays) in a single namespace, making that namespace (or environment) even more MATLAB-like. For example, one can call the sin and cos functions just like you could in MATLAB, as well as having all the features of pyplot.
- More details about `pylab,matpltlib,pyplot` here
- `PyLab` is a convenience module that bulk imports `matplotlib.pyplot` (for plotting) and `NumPy` (for Mathematics and working with arrays) in a single name space. Although many examples use `PyLab`, it is **no longer recommended**. Use `pyplot` instead.

7.1 Pylab Tutorial

- `plt.xlabel('xlabel')`, `plt.ylabel('ylabel')`, `plt.title('title')`
- `plt.clf()` clean the window, `plt.ylim(0,1000)`, `plt.legend(loc = 'upper left')`,
- Color/line style/line width similar to matlab plot
- Subpot/axis scale similar to matlab plot
- More documentation can be found from here

7.1.1 Basic Usage

[ ]:
```
# allows me to reference any library procedures as plt.<procName>
import pylab as plt # plt -> plot, imported pylab into the name plt

# generate some example data
mySamples = []
myLinear = []
myQuadratic = []
myCubic = []
myExponential = []

for i in range(0,30):
    mySamples.append(i) # x values
    myLinear.append(i) # y values
```

```
    myQuadratic.append(i**2)
    myCubic.append(i**3)
    myExponential.append(1.5**i)

# ALL PLOTS ON ONE GRAPH
plt.plot(mySamples,myLinear) # x,y
plt.plot(mySamples,myQuadratic)
plt.plot(mySamples,myCubic)
plt.plot(mySamples,myExponential)
```

7.1.2 Plot Multiple Figures and Labels

[ ]:
```
# PLOTS ONSEPARATE GRAPHS / PROVIDEING LABELS
plt.figure('lin')
plt.xlabel('sample points') # labels
plt.ylabel('linear function')
plt.plot(mySamples,myLinear)

plt.figure('quad')
plt.plot(mySamples,myQuadratic)

plt.figure('cube')
plt.xlabel('sample points')
plt.ylabel('cubic function')
plt.plot(mySamples,myCubic)

plt.figure('expo')
plt.xlabel('sample points')
plt.ylabel('exponential function')
plt.plot(mySamples,myExponential)

plt.figure('quad') # open the quad figure and we add label inside of it
plt.xlabel('sample points')
plt.ylabel('quadratic function')
```

7.1.3 Titles and Scales

[ ]:
```
# ADDING TITLES/ CHANGE SCALES
plt.figure('lin')
plt.clf() # clear the previous windows, such as x, y labels
plt.ylim(0,1000) # set limits on the axis or axes
plt.plot(mySamples,myLinear)
plt.title('Linear')

plt.figure('quad')
plt.clf()
```

```
plt.ylim(0,1000)
plt.plot(mySamples,myQuadratic)
plt.title('Quadratic')

plt.figure('cube')
plt.clf()
plt.plot(mySamples,myCubic)
plt.title('Cubic')

plt.figure('expo')
plt.clf()
plt.plot(mySamples,myExponential)
plt.title('Exponential')
```

7.1.4 More Options (color/label/title/legend/linewidth...)

[ ]:
```
# OVERLAYING PLOTS/ ADD LABELS / COLOR AND STYLE / LINE WIDETH
# very similar to matlab, see documentation for choices of color and style
plt.figure('lin quad')
plt.clf()
plt.plot(mySamples,myLinear,'b-',label = 'linear', linewidth=2.0) # add legend
plt.plot(mySamples,myQuadratic,'ro', label = 'quadratic', linewidth=3.0)
plt.yscale('log') # change to log scale
plt.legend(loc = 'upper left') # specify label location
plt.title('Linear vs. Quadratic')

plt.figure('cube exp')
plt.clf()
plt.plot(mySamples,myCubic,'g^',label = 'cubic', linewidth=4.0 )
plt.plot(mySamples,myExponential,'r--', label = 'exponential', linewidth=5.0)
plt.legend() # let python decides what is the best location, necessary
plt.title('Cubic vs. Exponential')
```

7.1.5 Subplots

[ ]:
```
# USING SUBPLOTS
plt.figure('lin quad')
plt.clf()
plt.subplot(211) # 2 rows 1 column 1st location
plt.ylim(0,900)
plt.plot(mySamples,myLinear,'b-',label = 'linear', linewidth=2.0) # add legend
plt.subplot(212)
plt.ylim(0,900)
plt.plot(mySamples,myQuadratic,'ro', label = 'quadratic', linewidth=3.0)
plt.legend(loc = 'upper left') # specify label location
plt.title('Linear vs. Quadratic')
```

```
plt.figure('cube exp')
plt.clf()
plt.subplot(121) # 1 row 2 columns 1st location
plt.ylim(0,140000)
plt.plot(mySamples,myCubic,'g^',label = 'cubic', linewidth=4.0 )
plt.subplot(122)
plt.ylim(0,140000)
plt.plot(mySamples,myExponential,'r--', label = 'exponential', linewidth=5.0)
plt.legend() # let python decides what is the best location, necessary
plt.title('Cubic vs. Exponential')
```

7.1.6 A Complete Example

[ ]:
```
"""
AN EXAMPLE
"""
def retire(monthly,rate, terms):
    savings = [0]
    base = [0]
    mRate = rate/12
    for i in range(terms):
        base += [i] # addition in list
        savings += [savings[-1]*(1+mRate)+monthly]
    return base, savings # returning two elements

# constant rate, varying monthlies
def displayRetireWMonthlies(monthlies, rate, terms):
    plt.figure('retireMonth')
    plt.clf() # clear frame for reuse
    for monthly in monthlies:
        xvals, yvals = retire(monthly, rate, terms)
        plt.plot(xvals, yvals, label = 'retire:'+str(monthly))
        plt.legend(loc = 'upper left')

displayRetireWMonthlies([500,600,700,800,900,1000,1100], 0.05, 40*12)

# constant monthly, varying rates
def displayRetireWRates(month, rates, terms):
    plt.figure('retireRate')
    plt.clf() # clear frame for reuse
    for rate in rates:
        xvals, yvals = retire(month, rate, terms)
        plt.plot(xvals, yvals, label = 'retire:'+str(month)+ ':
 ↪'+str(int(rate*100)))
        plt.legend(loc = 'upper left')
```

```
displayRetireWRates(800,[.03, .05, .07], 40*12)


def displayRetireWMonthsAndRates(monthlies, rates, terms):
    plt.figure('retireBoth')
    plt.clf()
    plt.xlim(30*12, 40*12) # focus on last 10 years
    # we can use in Matlab plot
    monthLabels = ['r','b','g','k']
    rateLabels = ['-','o','--']
    for i in range(len(monthlies)):
        monthly = monthlies[i]
        monthLabel = monthLabels[i%len(monthLabels)] # nice trick, remainder␣
 ↪func cycle those labels
        for j in range(len(rates)):
            rate = rates[j]
            rateLabel = rateLabels[j%len(rateLabels)]
            xvals, yvals = retire(monthly, rate, terms)
            plt.plot(xvals,yvals, monthLabel+rateLabel, label = 'retire:
 ↪'+str(monthly)+ ':'+str(int(rate*100)))
            plt.legend(loc = 'upper left')

displayRetireWMonthsAndRates([500,700,900,1100],[.03, .05, .07], 40*12)
```

7.2 Pyplot Tutorial

- The most commonly-used matplotlib plotting module
- A useful tutorial is given here and here. This document provides descriptions about all functions in `pyplot`

7.2.1 2D Plot

- The most important function in matplotlib is `plot`, which allows you to plot 2D data.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```

7.2.2 Titles, Axis Labels, Legends

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

7.2.3 Subplots

- You can plot different things in the same figure using the `subplot` function.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

8 Numpy Tutorial

Numpy is the fundamental package for scientific computing with Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy. More details can be found from the Numpy Reference

- Numpy Basics

8.1 Numpy Basics

The following content are from Python Numpy Tutorial

8.1.1 Arrays

- A numpy array is a grid of values, all of the same type, and is indexed by a `tuple` of nonnegative integers. The number of **dimensions** is the `rank` (number of most outside brackets)of the array; the `shape` of an array is a `tuple` of integers giving the **size** of the array along each dimension.
- We can initialize numpy arrays from nested Python lists, and access elements using square brackets.
- More methods about the array creation in Numpy can be found here

[ ]:
```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

[ ]:
```
# Numpy also provides many functions to create arrays:
a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
```

```
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

8.1.2 Array Indexing

- Numpy offers several ways to index into arrays.
- `slicing` Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:
- `Integer array indexing`
- `Boolean array indexing`
- More information about array indexing can be found here

Slicing

[ ]:
```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array. (This is different from Matlab)
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

[ ]:
```
# You can also mix integer indexing with slice indexing.
# However, doing so will yield an array of lower rank than the original array.
# Note that this is quite different from the way that MATLAB handles array␣
 ↪slicing:
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
```

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

Integer Array Indexing

[ ]:
```
# - When you index into numpy arrays using slicing, the resulting array view␣
 ↪will always be a subarray of the original array.
# In contrast, integer array indexing allows you to construct arbitrary arrays␣
 ↪using the data from another array.
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]]) # rank 2, shpae is (3,2)

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"

# Produce a new array
b = a[[0, 1, 2], [0, 1, 0]]
print(b) # [1 4 5]
```

```
# change value
b[0]=10 # does not change the value in a

print(b) # [10 4 5]
print(a) # unchanged
```

[ ]:
```
# One useful trick with integer array indexing is selecting or mutating one␣
 ↪element from each row of a matrix:
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]]) # rank 2, shape = (4,3)

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(np.arange(4)) # [0 1 2 3]
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10 # like generator?

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

Boolean Array Indexing

[ ]:
```
# Boolean array indexing lets you pick out arbitrary elements of an array.
# Frequently this type of indexing is used to select the elements of an array␣
 ↪that satisfy some condition.
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.
```

```
print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

8.1.3 Datatypes

Every numpy array is a grid of elements of the same type. **Numpy** provides a large set of **numeric datatypes** that you can use to construct arrays. **Numpy** tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

[ ]:
```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int32"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

8.1.4 Array Math

- Basic mathematical functions operate `elementwise` on arrays, and are available both as operator overloads and as functions in the numpy module.
- Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication.
- Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`
- You can find the full list of mathematical functions provided by numpy in the documentation
- reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix `T`.
- Numpy provides many more functions for manipulating arrays; the full list can be found from here

[ ]:
```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
```

```
# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

[ ]:
```
# Note that unlike MATLAB, * is elementwise multiplication, not matrix␣
 ↪multiplication.
# We instead use the dot function to compute inner products of vectors, to␣
 ↪multiply a vector by a matrix, and to multiply matrices.
# dot is available both as a function in the numpy module and as an instance␣
 ↪method of array objects:
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

[ ]:
```
# Numpy provides many useful functions for performing computations on arrays;␣
 ↪one of the most useful is sum
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

[ ]:
```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
# Python does not distinguish between a rank 1 row array or a column array
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

8.1.5 Broadcasting

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.
- Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.
- `np.tile(A,reps)` is desribed here

- Broadcasting two arrays together follows these rules:
  - If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
  - The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.

  - The arrays can be broadcast together if they are compatible in all dimensions.
  - After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
  - In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension
- More explanation about **array broascasting in Nunpy** can be found from here and here
- Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.
- Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

[ ]:
```
# For example, suppose that we want to add a constant vector to each row of a␣
 ↪matrix.
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
print(x)
```

[ ]:
```
# This works; however when the matrix x is very large, computing an explicit␣
 ↪loop in Python could be slow.
# Note that adding the vector v to each row of the matrix x is equivalent to␣
 ↪forming a matrix vv
# by stacking multiple copies of v vertically, then performing elementwise␣
 ↪summation of x and vv.
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
```

```
                        #          [1 0 1]
                        #          [1 0 1]
                        #          [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

[ ]:
```
# Numpy broadcasting allows us to perform this computation without actually␣
 ↪creating multiple copies of v.
# Consider this version, using broadcasting:
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
# The line y = x + v works even though x has shape (4, 3) and v has shape (3,)␣
 ↪due to broadcasting;
# this line works as if v actually had shape (4, 3), where each row was a copy␣
 ↪of v, and the sum was performed elementwise.
```

[ ]:
```
# Here are some applications of broadcasting:
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
```

```
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

9 Appendix

- Useful Python Libraries
- Usful Python Learning Resources

9.1 Useful Python Libraries

- Pyomo : A Python library for optimization (quadprog/linprog/mixed prog) ...
- random: generate randome numbers
- datetime: basic date and time types
- time: time access and conversions (a submodule of datetime module)
- matplotlib: visualization
- numpy: fundamental package for scientific computing in Python
- scipy: Fundamental algorithms for scientific computing in Python
- ffmpy: for converting video files from mp4 to avi. Follow here for installation.
- Python Control Systems Library: for control system design
  - Youtube Tutorial
- Note need to open the VS code app from Anaconda navigator to use Conda libraries

9.2 Usful Python Learning Resources

- Chemical Process Control : A process control course taught in Pyhton (model-

ing/PID/frequency domain/optimal control/linearization/optimization)