Python Study Notes

February 10, 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.

- Scalar and Non-scalar Objects
- Expressions in Python
- Binding Variables
- Comparison Operators
- Logic Operators
- Conditional Statement
- Strings
- While Loops
- For Loops
- Iteration
- Guess and Check

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.

[ ]:
```python
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.

[ ]:
```python
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)

[ ]:
```python
"""
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**

[ ]:
```python
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
```

```python
# 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␣
 ↪yousel f
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

[ ]:
```
"""
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))
```

```
2.2
```

5 Appendix

- Useful Python Libraries

5.1 Useful Python Libraries