Call stacks:

Iterative functions use less memory than recursive functions

Recursive functions keep the calls in the memory and can slow down the code and can cause a max recursion error

Print/return

Return is an =, print isn’t

\*\*

STRINGS

Structural recursive design pattern

A string is:

An empty string “ “

Or a non-empty string comprising of two pieces, a head character, and a tale which is itself, another strgin

Goals:

Compute length, reverse of string

Helper functions:

Input string empty string(s)

Input string head(s)

Input tail(s)

Input concat(s1,s2) $ can use double quotes

sl = “this is a string 1”

s2 = ‘this is a string 2’ # single quotes’

s3 = “ I don’t want to use single quotes”

s4 = ‘I can\’t use \t this is because \n \\ I said so’

s5 = “”

print(s1)

print(s2)

print(s3)

#print(s1[25]) # python will give error if accessing character not in string

print(s1[0]) # access of character begins with index 0

midterm question about accessing characters in string, indexing begins with zero

print(s1[5])

print(s1[-3]) # python allows negative indexing, and it starts from one side to the next being index -1

\*\*\*\*

#slicing

Print(s1[3:7]# returns the substring beginning at inxed left, and ending, not including index right

Print(s1[:9]) # left argument is optional, if not included start at beginning

# can include last argument, include up to end

Print(s1[1:]) # right is also optional ` if not provided include up to ed

Print(s1[4:-2]) 3negative indices for slices as well

Print(1[7:5]) #not an error, returns empty string if left endpoint is right after right endpoint

Print(s1[40:42]) #also not an error to asl for error

\*\*

Concatenation

Newstr = s1+s2s3x

P

# strings are immutable

S1\_1 = ‘q’ # we cant assign new value to part of the string

Def isEmptyStr(s):

“””returns boolean indiciating ifa str is emptyt or not”””

If(s==””):

Return true

Else:

Return falst

Def head(s):

“””return the head character of the string – assume non empty”””

Def tails(s):

Return s[

def1:]

def length(s):

if is EmptyStr(s)):

return 0

else:

return 1+ length(tail(s))

def reverse(s):

if(isEmptyStr(s)):

return “ ”

else:

return reverse(tail(s)) + head(s)

len(s):

if empty

return –

else:

retun 1+len(len(tail)

subs trace:

len(‘abc’) = 1+len.bc.

= 1+(1+len(‘c’)

1+1+(1+(len(‘’)))

= 1+1+1+0=3

\*\*\*

For loops:

For el In S: # for “el” each individual element of the collection (eg. Characters in a string)

For s: to collectipn

Def lengthIter(s):

‘iteraticee version of length calculation’

Total = 0

For el in s:

Total +=1

Print(el) # el gives you access to each successive elemt in eaech iteration

Return total

Defreverse Iter(s):

Rev = ‘ ‘

For el in s:

Rev = el +rev

Return rev

Def testfunc():

If(length(‘’)==0)”

Print(“passed”)

Else:

Print(“failed”)

Print(‘abc’

\*\*\*

Reading from files:

Fp = open(“text.txt”)

#put files in same folder

For line in fp:

Def main():

Fp = open(‘text1.txt’)

For lie in fp:

# use strip function to remove whitespace fom left and right of str

If(line.strip()==(’word1’):

Printy (“match”)

Print(line, end=’ ‘) #print adds extra spaces between lines of pront

\*\*\*

Constant amount of work – order of 1

Independent of the size of input

Return s[0[[

Order of n notation – the amount of work is linear with the size of input – 0(n)

Order n squared – complexity explodes – alphabetize string