Command processor (stubs)

def Command\_Open(): # These are "stub" functions:

return # Functions that currently don't

# do anything, but eventually they

def Command\_Save(): # will. There is one function

return # for each "command" in the

# command processor.

def Command\_Print(): # If the stub functions were not

return # present, the program would crash

# trying to call a function that

def Command\_New(): # did not exist. Once we figure

return # out what to do, we will fill in

# all the stubs with the correct code.

def Main():

MoreToDo = True

while MoreToDo:

Command = input("Enter a command --- ")

if Command == "QUIT": MoreToDo = False

elif Command == "OPEN": Command\_Open() # Call stub

elif Command == "SAVE": Command\_Save() # Call stub

elif Command == "PRINT": Command\_Print() # Call stub

elif Command == "NEW": Command\_New() # Call stub

else: print ("Illegal Command")

return

Command Processor (Pass)

def Command\_Open():

return

def Command\_Save():

return

def Command\_Print():

return

def Command\_New():

return

# If we were too lazy or too busy to create the stub

# functions, we can use the "pass" statement instead.

# The pass statement doesn't do anything, but keeps

# Python happy if there isn't anything to do (or if

# you don't yet know what to do) after a while, for,

# if, elif, or else statement.

def Main():

MoreToDo = True

while MoreToDo:

Command = input("Enter a command --- ")

if Command == "QUIT": MoreToDo = False

elif Command == "OPEN": Command\_Open() # Call stub

elif Command == "SAVE": Command\_Save() # Call stub

elif Command == "PRINT": Command\_Print() # Call stub

elif Command == "NEW": Command\_New() # Call stub

elif Command == "EDIT": pass # Do nothing, replace with stub call later

elif Command == "COPY": pass # Do nothing, replace with stub call later

else: print ("Illegal Command")

return

Command Processor (string upper)

def Command\_Open():

return

def Command\_Save():

return

def Command\_Print():

return

def Command\_New():

return

# The problem with previous versions of the command processor

# is that command "OPEN" is legal but command "open" is not.

# By using the .upper() method on strings, we can capitalize the

# user's input, and test against only the upper case version.

# The user could enter "open", "OPEN", "Open", "oPeN", or any

# other variant, and it will always be treated as "OPEN".

def Main():

MoreToDo = True

while MoreToDo:

Command = input("Enter a command --- ")

Command = Command.upper() # Capitalize string

if Command == "QUIT": MoreToDo = False

elif Command == "OPEN": Command\_Open() # Call stub

elif Command == "SAVE": Command\_Save() # Call stub

elif Command == "PRINT": Command\_Print() # Call stub

elif Command == "NEW": Command\_New() # Call stub

elif Command == "EDIT": pass # Do nothing, replace with stub call later

elif Command == "COPY": pass # Do nothing, replace with stub call later

else: print ("Illegal Command")

return

Command Processor (writing a new function)

def Command\_Open():

return

def Command\_Save():

return

def Command\_Print():

return

def Command\_New():

return

# Here we write a new function to get a string from

# the user and always return the capitalizd version.

# We can call the function anywhere such a string is

# desired. Notice that the call to MyInput passes it

# the actual parameter "Enter a command --- ", which

# inside MyInput comes in as formal parameter Message,

# which is then passed straight on to the input function.

def MyInput (Message): # Function to get a string and capitalize it

S = input(Message)

return S.upper()

def Main():

MoreToDo = True

while MoreToDo:

Command = MyInput("Enter a command --- ")

if Command == "QUIT": MoreToDo = False

elif Command == "OPEN": Command\_Open() # Call stub

elif Command == "SAVE": Command\_Save() # Call stub

elif Command == "PRINT": Command\_Print() # Call stub

elif Command == "NEW": Command\_New() # Call stub

elif Command == "EDIT": pass # Do nothing, replace with stub call later

elif Command == "COPY": pass # Do nothing, replace with stub call later

else: print ("Illegal Command")

return

Command Processor (strings that span lines)

def Command\_Open():

return

def Command\_Save():

return

def Command\_Print():

return

def Command\_New():

return

def MyInput (Message): # Function to get a string and capitalize it

S = input(Message)

return S.upper()

# Python supports more different types of strings than

# just "double-quoted" or 'single-quoted' strings. If

# you surround a string with """triple-double""" or

# '''triple-single''' quotes, the string is allowed to

# span line breaks, and the line breaks will be included

# in the string itself. Notice that variable Message

# is assigned a four-line string, where the first three

# lines might contain instructions to the user, and the

# last contains the prompt indicating the place to type

# in a response. This is passed to MyInput (and then

# on to input), so the entire four-line string is printed

# each time the user is requested for input.

def Main():

Message = """Line #1 of instructions

Line #2 of instructions

Line #3 of instructions

Enter a command --- """

MoreToDo = True

while MoreToDo:

Command = MyInput(Message)

if Command == "QUIT": MoreToDo = False

elif Command == "OPEN": Command\_Open() # Call stub

elif Command == "SAVE": Command\_Save() # Call stub

elif Command == "PRINT": Command\_Print() # Call stub

elif Command == "NEW": Command\_New() # Call stub

elif Command == "EDIT": pass # Do nothing, replace with stub call later

elif Command == "COPY": pass # Do nothing, replace with stub call later

else: print ("Illegal Command")

return

Time and rand

import time

import random

print ("Number of seconds since January 1, 1970: ", time.time())

print ()

print ("Local Time (as list): ", list(time.localtime()))

print ("[YEAR,MONTH,DAY,HOUR,MINUTE,SECOND,DAYOFWEEK,DAYOFYEAR,DAYLIGHTSAVINGTIME]")

print ()

print ("Local Time (as text): ", time.asctime(time.localtime()))

# Use time.time() to profile how long a function will take to

# execute. Does Task1, which does nothing 10 million times,

# take longer time or shorter time than Task2, which uses

# time.sleep to wait exactly one-half second?

def Task1 ():

for I in range(10000000): pass # Do nothing 10 million times (busy wait)

return

def Task2 ():

time.sleep(0.5) # Do nothing for exactly one-half second

return

def Main():

# Profile how long Task1 takes.

Before = time.time()

Task1()

After = time.time()

Elapsed = After - Before

print ()

print ("Task 1 takes ", Elapsed, " seconds")

print ("This will be different based on computer's speed.")

print ("A faster computer will have a smaller number,")

print ("and a slower computer will have a larger number.")

# Profile how long Task2 takes.

Before = time.time()

Task2()

After = time.time()

Elapsed = After - Before

print ()

print ("Task 2 takes ", Elapsed, " seconds")

print ("This will be roughly the same for every computer.")

Main()

print ()

for I in range(20): print ("Random number 0<=N<1: ",random.random())

print ()

for I in range(20): print ("Six-Sided Die #1: ", int(random.random()\*6)+1)

print ()

for I in range(20): print ("Six-Sided Die #2: ", random.randrange(6)+1)

print ()

for I in range(20): print ("Six-Sided Die #3: ", random.randint(1,6))

print ()

for I in range(20): print ("Six-Sided Die #4: ", random.choice([1,2,3,4,5,6]))

Test Dr. Bill’s library

# William T. Verts

from DrBillsInputOutput import \*

def Main():

Filename = pickAFile("Please enter a file name")

showInformation (Filename)

return

Reading and writing to files

import math

from DrBillsInputOutput import \*

def ReadFileAsOneString(Filename):

Infile = open(Filename, "r")

X = Infile.read()

Infile.close()

return X

def ReadFileAsListOfStrings(Filename):

Infile = open(Filename, "r")

X = Infile.readlines()

Infile.close()

return [C.rstrip("\n") for C in X]

def Main():

Limit = requestIntegerInRange("Enter a limit", 1, 1000)

Filename = pickAFolder() + "/MyResults.txt"

print (Filename)

MyFile = open(Filename, "w")

for I in range(Limit):

MyFile.write (str(I) + " " + str(math.sqrt(I)) + "\n")

MyFile.close()

return

Chr and their numbers

# Copyright © 2019 W. T. Verts

def Main():

print ("Copyright © 2019 W. T. Verts")

for I in range(32, 256):

print (I, " = ", chr(I))

return

Int to str and inttostr bases

#----------------------------------------------------------------------

# This is how the str function works when given an int argument.

#----------------------------------------------------------------------

def IntToStr(N): # N is the number for which we want a printable string.

S = "" # String S is where we will be building the string result.

Negative = (N < 0) # Will be True if N<0, False otherwise.

N = abs(N) # Make sure N >= 0 for the while-loop.

while (N > 0): # As long as N isn't zero yet...

Remainder = N % 10 # ...carve off the right decimal digit of N, then...

N = N // 10 # ...shift N right by one decimal digit, then...

Digit = chr(ord("0") + Remainder) # ...convert the remainder digit into a character, and...

S = Digit + S # ...add the character to the left end of string S.

if S == "": S = "0" # If N was 0, S would still be empty, so we need to correct for that.

if Negative: S = "-" + S # If N was originally negative add a minus sign to the string S.

return S # Return the resulting string.

#----------------------------------------------------------------------

# This is how we can convert an int N to a string for some other base

# than base 10.

#----------------------------------------------------------------------

def IntToStrBases(N,Base=10):

S = ""

Negative = (N < 0)

N = abs(N)

while (N > 0): # As long as N isn't zero yet...

Remainder = N % Base # ...carve off the right digit of N in the target base, then...

N = N // Base # ...shift N right by one digit in the target base, then...

if Remainder > 9: # ...if the digit is not in 0...9, we...

Digit = chr(ord("A") + Remainder - 10) # ...use the letters starting at "A"

else: # ...otherwise the digit is in 0...9, so...

Digit = chr(ord("0") + Remainder) # ...convert it as we did before, and...

S = Digit + S # ...add the character to the left end of string S.

if S == "": S = "0"

if Negative: S = "-" + S

return S

Motivation for dictionaries

#----------------------------------------------------------------------

# Motivation for dictionaries

#----------------------------------------------------------------------

#----------------------------------------------------------------------

# WITHOUT DICTIONARIES

#----------------------------------------------------------------------

# Remember Lab 1? We did something like the following:

def Process (Name,Score):

# Do something fun with Name and Score, like:

print (Name, Score)

return

def Main1():

Process("Fred",78)

Process("Sam",34)

Process("Mary",98)

Process("Tom",57)

return

# This is not an efficient approach, because

# each name and score is "burned in" to the

# program. Here's a solution that separates

# out the data a little more cleanly, then

# the code to run through all the data is more

# efficient and much more general:

def Main2():

Data = [["Fred",78],["Sam",34],["Mary",98],["Tom",57]]

Index = 0

while (Index < len(Data)):

Student = Data[Index] # A list of a name and a score

Process(Student[0],Student[1])

Index = Index + 1

return

# Here's a different way of doing the same thing, since

# we really don't need to know \*where\* in Data each student

# is located:

def Main3():

Data = [["Fred",78],["Sam",34],["Mary",98],["Tom",57]]

for Student in Data:

Process(Student[0],Student[1])

return

# But what about searching? Here's one approach:

def Main4():

Data = [["Fred",78],["Sam",34],["Mary",98],["Tom",57]]

Search = input("Enter a student name --- ")

Index = 0

Found = False

while (Index < len(Data)) and not Found: # Same as: ...and Found == False

Student = Data[Index]

if Search == Student[0]:

print (Student[1])

Found = True # Stop the loop because we found a match

else:

Index = Index + 1 # Otherwise go to the next item to compare

return

# Here's a better search:

def Main5():

Data = [["Fred",78],["Sam",34],["Mary",98],["Tom",57]]

Search = input("Enter a student name --- ")

for Student in Data:

if Search == Student[0]:

print (Student[1])

break # Stops the loop

return

#----------------------------------------------------------------------

# WITH DICTIONARIES (See: Companion page 250)

#----------------------------------------------------------------------

def Main6():

Data = {"Fred":78, "Sam":34, "Mary":98, "Tom":57}

Search = input("Enter a student name --- ")

if Search in Data:

print (Data[Search])

return

def Main7():

Data = {"Fred":78, "Sam":34, "Mary":98, "Tom":57}

for Name in Data.keys():

Process (Name, Data[Name])

return

#----------------------------------------------------------------------

Polynomials

#------------------------------------------------------------

# Copyright (C) 2017-2019 Dr. William T. Verts

#------------------------------------------------------------

# See pages 272-275 in the Companion for a variation of

# these functions

#------------------------------------------------------------

#------------------------------------------------------------

# Clean up a polynomial by enforcing the following rules:

#

# (1) The polynomial is never empty.

# (2) Any floats with fractions really close to zero

# get converted to integers.

# (3) Any high-order coefficients of zero get deleted,

# except that if the polynomial is entirely zeroes

# the result returned is [0].

# Examples:

# polyNormalize([2,0,3,4,0,5]) returns [2,0,3,4,0,5]

# polyNormalize([2,0,3,0,0,0]) returns [2,0,3]

# polyNormalize([2.0, 3.5, 4]) returns [2, 3.5, 4]

# polyNormalize([0,0,0,0,0,0]) returns [0]

# polyNormalize([]) returns [0]

#------------------------------------------------------------

def polyNormalize (P): # P is a list of coefficients

Q = [0 for I in range(max(1,len(P)))] # Create a zero polynomial Q the same size as P (or [0] if P==[])

for I in range(len(P)): # For every entry in P

Value = P[I] # Grab the coefficient

if (abs(Value - float(int(Value))) < 1.0E-14): # Close enough to integer to call it one

Q[I] = int(Value) # Convert (possible) float to int

else: #

Q[I] = Value # Have to keep it as a float

while (len(Q) > 1) and (Q[-1] == 0): del Q[-1] # Delete any high-order zero entries (but leave one if all were 0)

return Q # Return the cleaned-up polynomial equivalent to P

#------------------------------------------------------------

# Add/Subtract two or three polynomials.

#------------------------------------------------------------

def polyAdd(P0,P1,P2=[]): # P0, P1, and P2 are lists of coefficients

P = [0 for I in range(max(len(P0),len(P1),len(P2)))] # P will be my answer, length assumed to be same as longest polynomial

for I in range(len(P0)): P[I] = P[I] + P0[I] # Add P0 to result

for I in range(len(P1)): P[I] = P[I] + P1[I] # Add P1 to result

for I in range(len(P2)): P[I] = P[I] + P2[I] # Add P2 to result

return polyNormalize(P) # Return cleaned-up result

def polySubtract(P0,P1,P2=[]): # P0, P1, and P2 are lists of coefficients

P = [0 for I in range(max(len(P0),len(P1),len(P2)))] # P will be my answer, length assumed to be same as longest polynomial

for I in range(len(P0)): P[I] = P[I] + P0[I] # Add P0 to result

for I in range(len(P1)): P[I] = P[I] - P1[I] # Subtract P1 from result

for I in range(len(P2)): P[I] = P[I] - P2[I] # Subtract P2 from result

return polyNormalize(P) # Return cleaned-up result

#------------------------------------------------------------

# Multiply two polynomials. This is like multiplying two

# large numbers by hand where there are partial products

# that must be added to the correct digit position. In this

# case, the position of a partial product is determined by

# the new exponent. For example, if P0 = [1,4,2] = 2X^2+4X+1

# and P1 = [7,3,8] = 8X^2+3X+7, then the partial product of

# P0[1] and P1[2] represents the 4X term times the 8X^2 term,

# which is 32X^3, so result of multiplying P0[1] and P1[2]

# (the 32) adds into slot 1+2 of the result, or P[3]. That

# is, if I=1 and J=2, then P[I+J] = P[I+J] + P0[I] \* P1[J]

# is the same as P[3] = P[3] + P0[1] \* P1[2].

#------------------------------------------------------------

def polyMultiply(P0,P1): # P0 and P1 are lists of coefficients

P = [0 for I in range(len(P0)+len(P1))] # P will be my answer, length assumed to be total of terms in both polynomials

for I in range(len(P0)): # Step through all coefficients in P0,

for J in range(len(P1)): # For each coefficient in P0 step through all coefficients in P1,

P[I+J] = P[I+J] + P0[I] \* P1[J] # Add partial product to correct slot in the answer

return polyNormalize(P) # Return cleaned-up result

#------------------------------------------------------------

# Differentiate P(X) with respect to X. In this code the

# P[0] term is the constant, which is ignored, and the result

# will have one fewer terms than the source. All other terms

# multiply the exponent (I) times the coefficient (P[I]) to

# get the coefficient of the next lower term. Notice that

# the code to differentiate a polynomial is significantly

# smaller and simpler than multiplication!

#------------------------------------------------------------

def polyDifferentiate(P):

Q = [0 for I in range(len(P)-1)]

for I in range(1,len(P)): Q[I-1] = I \* P[I]

return polyNormalize(Q)

#------------------------------------------------------------

# Convert a polynomial from its internal representation into

# a string that has the appearance of a standard mathematical

# polynomial. That is, a polynomial such as [-1, 2, 0, 5]

# returns the string 5X^3 + 2X - 1 (notice that the X^1 term

# suppresses the exponent, the X^0 term uses just the

# coefficient, and the X^2 term has a zero coefficient and

# is therefore ignored).

#------------------------------------------------------------

def polyStr(P):

S = ""

for Exponent in range(len(P)-1,-1,-1): # Step through coefficients from highest exponent to lowest

Coefficient = P[Exponent]

if Coefficient != 0: # We only care about non-zero coefficients

if Coefficient < 0:

if S != "":

S = S + " - " # String not empty and coefficient was negative (internal term)

else:

S = S + "-" # String was empty but coefficient was negative (first term)

else:

if S != "":

S = S + " + " # String not empty and coefficient was positive (internal term)

# The + is suppressed if first term was positive.

if (Exponent == 0) or (abs(Coefficient) != 1): # No need to include coefficient if it is 1 (unless it's the X^0 term)

S = S + str(abs(Coefficient)) # Include the coefficient without a sign (handled earlier)

if (Exponent >= 1): # Include X for all but the X^0 term

S = S + "X"

if (Exponent > 1): # Include the exponent for all terms X^2, X^3, X^4, etc., but not X^0 or X^1

S = S + "^" + str(Exponent)

if S == "": S = "0" # If the string comes up empty, it's zero.

return S

#------------------------------------------------------------

# Tests

#------------------------------------------------------------

Frog = [-1,7,0.5,0,-3]

Toad = [5,-3,2.5,2,-3]

print ("Frog = " + polyStr(Frog))

print ("Toad = " + polyStr(Toad))

print ("Frog + Toad = " + polyStr(polyAdd(Frog,Toad)))

print ("Frog - Toad = " + polyStr(polySubtract(Frog,Toad)))

print ("Frog \* Toad = " + polyStr(polyMultiply(Frog,Toad)))

print ("d/dx(Frog) = " + polyStr(polyDifferentiate(Frog)))