#______________________________________________________________________
#
#1.py Let's look at what functions do and how they work. If you understand
# this it will become clear to you why we use functions.

#Say program main() has many things to do. One of those things is to get the
#sum $S = 1^2 + 2^2 + 3^2 + 4^2$ ........ + $n^2$ for any given value of $n$

#Instead of main() doing it directly, main() calls another program called
#Larry() to do this task, because Larry() is specially written to do this
#no matter who calls Larry(). Note that any another program, say even
#some function called Moe() could also call Larry to do this task.

import sys

def wait():
    # we use this function only to step through the code
    x = input()  # during lecture since it waits for me to hit a key
    print(" ")

def Larry(j):
    # Larry expects something to come in, it's called j
    # because the name is not important, the position is
    # if there is more than one parameter
    sum = 0 # Larry initializes the sum he will compute and give
    # to whoever asked him for it

    for i in range(1,j+1,1): #remember 1**2 + 2**2 + ...... + j**2
        sum = sum + i*i #square each i and add to sum

    # at this point the sum has been computed. Return this value to whoever
    # called and asked for this task to be done
    return(sum) # if you forget to return sum, caller will get the
    # "None" which is of a special "NoneType", and NOT what
    # the caller wants

def Formula(j):
    s = j*(j+1)*(2*j+1)/6 
    return(s) #you could simply put the above expression here
def main():
    n = 999  # this could be any user input value, in general
    wait()
    answer = Larry(n)  # Larry(999) will also work, since n = 999

    # Things to know:
    #1. variable n is "local" to main(). Larry does not know it exists and so
    #   cannot even access it.
    #2. variable sum is "local" to Larry(). Main() does not know it exists and
    #   so cannot even access it.
    #3. Precisely when Larry() is called, j in Larry() gets the value 999 and
    #   the
    #   Larry() starts to run
    #4. When Larry finishes, it RETURNS a value (sum) to its caller. This value
    #   is placed in variable "answer" in Main(). Larry() has no access to this
    #   variable either. You can assume that Main() receives the value from
    #   Larry and stores it in variable "answer".

    print("The answer is: ", answer)

    #Actually, we know a formula for this sum, so we can check if Larry() is
    #doing this task right. So let's call function Formula() to tell us.
    wait()

    print("The closed form solution from Formula() is", Formula(n))

    # Note how Formula returns a FLOATING POINT number because it did division
    # If you want it to return an integer you must use "int()" as in the
    # call below

    wait()
    print("The closed form solution from Formula() is", int(Formula(n)))

    #Now suppose you wanted to compute
    # k**2 + (k+1)**2 + (k+2)**2 + ........ + n**2, for some integer k >= 1,
    # k<=n
    # Function Moe() can call Larry() twice to do it
wait()

m = Moe(900,1066)      #k = 900, n = 1066

print("Diff. of squares answer is ",m)

wait()

f = Formula(1066) - Formula(899)  # 899 because of how Formula works

print("and the formula says ", int(f))

#---------------------- end main() -------------------------------------

def Moe(k,n):    #Moe needs the "start" integer (k) and "stop" integer (n)

   if (k > n):
        print(" k:" ,k,"is greater than n:" ,n)
        sys.exit(0)
   if (k < 1):
        print(" Bad k-value" ,k)
        sys.exit(0)
   if (n < 1):
        print("Bad n-value" ,n)
        sys.exit(0)

#The above checks are just so that Moe does not get surprised by bad
#inputs. Catching things in this way will save a lot of trouble later
#if you have to look inside a lot of code into what went wrong

s = Larry(n) - Larry(k-1)     # subtract out first (k-1) squares

return(s)

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

# If you understand the above example completely, you understand how
# functions work

# They take in parameters, do something with those parameters and then
# they finish execution
# Usually they RETURN one or more values to the calling program
# Sometimes they do some work (say to some data structure in memory)
# and don't actualluy return any values

# The key idea is: they do SOME work, and you clearly define that work
# in a function and always call that function to do the work.

# When main() calls either Larry() or Moe(), main() simply waits for
# them to finish and return. When they finish, main() resumes execution
# from the place where the function call was made and finished.

# When you define variables, always keep in mind the SCOPE of variables.
# That is, which functions can see/access which variables? So far, all the
# variables you have seen were

# 1. "local" to the function in which you first defined them, and
# 2. If you make two consecutive calls to the same function you must
#    not assume the local variables will remember values you assigned
#    during the first call. They will not.

# 2.py

# When you define variables, always keep in mind the SCOPE of variables.
# That is, which functions can see/access which variables? So far, all the
# variables you have seen were

# 1. "local" to the function in which you first defined them, and
# 2. If you make two consecutive calls to the same function you must
#    not assume the local variables will remember values you assigned
#    during the first call. They will not.

def wait():
    print(" ")
    x = input()

def Larry(x):
    # assume first call to Larry is ALWAYS made with x <= 0
    # When Larry completes Larry(-1) it seems that count = 0
    if ( x > 0):
        print("Second call to Larry, x = ",x," count = ",count)
    if ( x <= 0):
        count = 0
    else:
        count = 1
def main():

    # first call to Larry, with parameter set to -1
    Larry(-1)

    # second call to Larry, with parameter set to 1
    Larry(1)  # this call fails. Python will complain that you are trying to
    # print a variable before even giving it a value

    # it "does not remember" variable count or it's value from
    # the previous call to Larry(0)

    # *if* it did remember, you might have expected it to
    # say "oh, count = 1" because of the previous call

    # But it does not!

    # Conclusion: functions do not remember local variable values
    # between calls (these are called "automatic
    # variables" -- they come and go)

# 3.py

# Functions can return multiple (i.e., many) values, and not just none

# Let's write a simple example of a main program that wants a list of
# the first names of people who will be coming to your "Post-cs177-midterm
# exam party." :) 

# We'll write a function that let's you input a guest-list, and your
# function will return many names

def wait():
    print(" ")
    x = input()


def variable_sized_guestlist(n):
    who = []
print("Enter the first names 
\n")

for i in range(0,n,1):
    who.append(str(input()))

return(who)

def guestlist_with_exactly_three_guests():
    print("Enter the first names 
\n")

    name1 = str(input())
    name2 = str(input())
    name3 = str(input())

    return(name1,name2,name3)

def main():

    wait()

    n = int(input("How many guests do you expect tonight? "))

    # When your variable sized guest-list function returns, the names should
    # go into the list "people"

    wait()

    people = variable_sized_guestlist(n)  #variable people gets a list

    wait()

    print("Program main() says OK, I got the list: ",people)

    #Now if you have a small number of parameters to return and you
    #don't care to put them in a list, you can do the following.

    #We'll assume there will only be three guests

    wait()
    print("Now let's try the guest-list with exactly 3 guests")
    wait()

    people = guestlist_with_exactly_three_guests()

    wait()

    print("Program main() says OK, I got the names: ",people)
wait()

print("Notice that in the first case it returned a list")
print("You can tell by the square braces [ ]")
print(" ")
print("In the second case the parameters came one by one")
print("embedded in parentheses ( )")

# #4.py

# Python ***"passes all parameters by value"***.

# What does this mean? It means a function works on the VALUEs of the # parameters it is given. Because of this, it CANNOT change the values # of variables in the calling program though it receives these variables # as parameters.

# It only use the values for something and (a) return some esult value(s) to # the caller, or (b) return nothing, if the caller wants nothing returned.

# Let's look at the addInterest example from page 186 of the textbook. # You want to add an r% interest to the balance on an account in a bank.

def addInterest(balance, rate):
    # account balance, rate of interest

    newBalance = balance + balance*(rate/100)  # r is passed as a percentage # hence we divide by 100

    balance = newBalance

    return  # returning nothing, hoping that balance's value has # been changed in main()

# You are hoping that when main() passes "balance" as a parameter, this # addInterest function will change its value by adding the r% interest. # But it's value CANNOT be changed in the main program. Remember "scope" ?

# We'll try it out and explain below why this happens

def main():

    amount = 1000  # your account has $1000 now
    r = 5  # you need to get 5% interest added to your account

    #Let function addInterest do the job
addInterest(amount, r)  # You could have also typed addInterest(1000, 5)

print("My updated balance is = \${0}".format(amount))

print(" ")

print("which of course means addInterest could not change its value!")

# Note: if addInterest() simply returns balance or newBalance, and amount
# accepts it as
#       amount = addInterest(amount, r)
# then amount will print the correct updated value. You already know this
# because you know that functions can return values to calling programs.

#
#_____ But why is it that "amount" cannot be changed by addInterest?_____
#

# Using a different "comment structure" below; it will work when its not
# the first thing in a class/function/module.

'''
A triple quote start of comment block

when program main() calls addInterest():

    amount ---------> 1000
         |           both amount and balance point
to the same value 1000

    r ---------> 5
         |           both r and rate point to
the same value 5

When addInterest(balance, rate) is called,

    balance gets the value 1000
    rate gets the value 5

and function addInterest has no access to variable "amount" and thus
cannot change its value

That is why Python's parameter passing is called "pass-by-value"

(now a triple quote to end the comment block)
Consider a list of account balances:

<table>
<thead>
<tr>
<th>amount</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000</td>
<td>1275</td>
</tr>
<tr>
<td>522</td>
<td>897</td>
</tr>
</tbody>
</table>

Now while things with r and rate remain the same as before, amount is now a LIST OBJECT of balances.

The entire list object gets passed from main() to addInterest, because the list object is the value of the list variable amount.

In addinterest, the variable balance has the entire list (the exact same list that amount is referring to).

Now addInterest() can walk through the list and change the value of each entry.
As it updates each entry by adding 5% to each, what actually happens is that the old values (1000, 1275, etc) are not erased. Instead each position in the list will refer to a new updated value, and the old values that sit around for a bit will be cleaned by by Python's "garbage collector." You can think of this as a separate function which you do not see, but one that runs every so often to collect all the space used up by unused values and return them to the pool of space the program needs.
#Note: whenever you pass a mutable object (list obj, graphics obj) to a
#function, the above idea enables the function to change its value
#so the calling programs gets the passed parameter value changed/updated

def addInterest(balance, rate):
    # balance is a list of accounts
    for i in range(0, 4, 1):
        balance[i] = balance[i] + balance[i] * rate / 100
    return  # returning nothing because balances will change.

# Now you know exactly why list amount will be updated as intended.

def main():
    amount = [1000, 1275, 522, 897]  # list of 4 accounts
    r = 5
    addInterest(amount, r)  # You could have also typed addInterest(1000, 5)

    print(" ")

    for i in range(0, 4, 1):
        print("Updated balance {0} is = ${1:0.2f}".format(i+1, amount[i]))

    print(" ")

    print("which means the list "amount" has been updated as intended")

    

From:
http://courses.cs.purdue.edu/ - Computer Science Courses

Permanent link:
http://courses.cs.purdue.edu/cs17700:spring16:week5_examples2

Last update: 2016/02/08 22:46