Objectives

* Timing execution time for a function.
* Unpacking lists to a function (a pseudo-polymorphism attempt)

Due Date

This assignment is due on Sunday, February 4, 2024, by 6:00 pm.

**Remarks:**

* When you are asked to hand in code, you cut-and-paste your code as text and paste into this Word document immediately following the prompt for your code.
* Include with your code several test case examples and your results.
* This lab is cooperative - talk with your classmates as you go through, and make sure you are progressing together.

Value

This assignment is worth 10 points.

Activities

You will need to write two files: one with the function to time how long it takes to execute a function and another test script that will test that function.

1. Start by writing a basic function time\_function(func, arg) that returns the number of seconds to run any function func with a single arg. Make use of the timing examples provided in the text.
2. Store your function time\_function(func, arg) in a filename TimeFunctions.py. Insert the code for t time\_function(func, arg) below this line.

import time  
  
def time\_function(func, arg):  
 *'''Takes a function and argument input and outputs the time for a process to finish in seconds'''* start\_time = time.time()  
 func(arg)  
 end\_time = time.time()  
 return (end\_time - start\_time)

1. Create a test script Test\_TimeFunctions.py and import your TimeFunctions.py and other necessary modules.
   * Define a function test\_func(L) that accepts a list and iterates through that list multiplying every number in the list by 2. The function does not return anything. The function does not print out anything.
   * Run your time\_function(func, arg) with a list that contains the values 0, 1, …, 49.
   * Run your time\_function(func, arg) with a list that contains the values 0, 1, …, 59.
   * Print out your results formatted as shown in the string below where x.xxx is the time in *milliseconds* to three decimal places:  
     "t(L1) = x.xxx ms and t(L2) = x.xxx ms”
2. Provide your listing of your Test\_TimeFunctions.py below this line.

from TimeFunctions import time\_function  
  
  
def test\_func(L):  
 *"""Multiplies each number in the list L by 2."""* for i in range(len(L)):  
 L[i] \*= 2  
  
  
L1 = list(range(50)) # List that contains the values 0, 1, ..., 49  
L2 = list(range(60)) # List that contains the values 0, 1, ..., 59  
# Test cases  
time\_L1 = time\_function(test\_func, L1)  
time\_L2 = time\_function(test\_func, L2)  
# Formatted Print Statement  
print(f't(L1) = {time\_L1 \* 1000:.3f} ms and t(L2) = {time\_L2 \* 1000:.3f} ms')

1. Modify your time\_function(func, arg) to accept another parameter n\_trials with a default value of 10. The function will keep track of the results of all n\_trials and return the minimum amount of execution time across all of the n\_trials. We use the minimum of multiple trials because it is assumed that interrupts from other computer processes are the cause for deviations in time. The minimum gets you to the closest value for just the function and not the interrupts. Note it may help to use the operator float(‘inf’) which represents infinity (a very, very large number). Update your test script in Test\_TimeFunctions.py modifying the results print statement to, “The minimum execution time is x.xxx ms.” Where x.xxx is the number of *milliseconds* to three decimal places. Paste your updated code for time\_function below this line.

import time  
  
  
def time\_function(func, arg, n\_trials=10):  
 *"""Takes a function and argument input and  
 will keep track of the results of all n\_trials and  
 return the minimum amount of execution time"""* min\_time = float('inf')  
 for i in range(n\_trials):  
 start\_time = time.time()  
 func(arg)  
 end\_time = time.time()  
 min\_time = min(min\_time, end\_time - start\_time) # Minimum Time modification  
 return min\_time

1. Adjust your testing routine by modifying test\_func to incorporate calling the new time\_function(func, arg, n\_trials). Conduct some tests to measure the minimum value of running the function. Provide the printed results of what you confidently believe to be the minimum amount of time to run your test\_func(L) with a list of 1,000 numbers.

The minimum execution time for L1000: 0.000 ms

1. Now, modify test\_func(L) to now include returning the list. But, don’t do anything with the value being returned. Just return the list of values multiplied by 2. What is the difference in time if test\_func returns a single list of all the numbers multiplied by 2? Again, provided the printed results of what you confidently believe to be the minimum amount of time to run test\_func(L).

The minimum execution time for L1000: 0.000 ms

1. Adjust your testing routine by modifying test\_func so that it returns nothing but it prints out to the screen a listing of each value of the list multiplied by 2. Provide the printed results again for what you believe to be the minimum time now.

The minimum execution time for L1000: 5.013 ms

1. What could you say comparing the different version of the test functions?

Adding the print operation over the test\_func iterator added measurable amount of compute time.

1. Copy and modify time\_function to accept yet another argument and call it time\_function\_flexible(func, args\_list, n\_trials) with the parameters:
   * func – a function that accepts multiple arguments (more than one)
   * arg\_list – a *list* of elements that will be unpacked and passed as individual arguments to func. (You can assume the user provides the proper length list for the function.)
   * n\_trials – the number of trials to time test with a default value of 10 and the minimum time is returned.
2. Provide a listing of your new function time\_function\_flexible(func, n, args\_list, n\_trials) below this line.

def time\_function\_flexible(func, args\_list, n\_trials=10):  
 *"""Takes a function and list of elements input and  
 will keep track of the results of all n\_trials and  
 return the minimum amount of execution time as float"""* min\_time = float('inf')  
 for i in range(n\_trials):  
 start\_time = time.time()  
 func(\*args\_list)  
 end\_time = time.time()  
 min\_time = min(min\_time, end\_time - start\_time) # Minimum Time modification  
 return min\_time

1. Add to Test\_TimeFunctions.py two test functions: add\_2(x, y) which adds two numbers given to it and add\_3(x, y, z) which adds the three numbers given to it. Provide a listing of your test scripts and the results.

def add\_2(x,y):  
 *"""Adds two numbers."""* return x + y  
  
def add\_3(x,y,z):  
 *"""Adds three numbers"""* return x + y + z  
  
time\_add\_2 = time\_function\_flexible(add\_2, [10, 20])  
time\_add\_3 = time\_function\_flexible(add\_3, [10, 20, 30])  
  
  
print(f"Time for add\_2: {time\_add\_2 \* 1000:.3f} ms")  
print(f"Time for add\_3: {time\_add\_3 \* 1000:.3f} ms")

Time for add\_2: 0.000 ms

Time for add\_3: 0.000 ms

1. Going back to the original test\_func(L) that accepts a list and iterates through that list multiplying every number in the list by 2 and not returning anything or printing out anything. Run some test cases and determine what type Big-O function is represented? Provide your tests, results, and description supporting the Big-O function you believe it to be.

from TimeFunctions import time\_function  
  
def test\_func(L):  
 *"""Multiplies each number in the list L by 2."""* for i in range(len(L)):  
 L[i] \*= 2  
  
# Testing with increasing sizes  
sizes = [100000, 1000000, 10000000, 50000000]  
for size in sizes:  
 L = list(range(size))  
 time\_taken = time\_function(test\_func, L)  
 print(f"Size: {size}, Time taken: {time\_taken \* 1000:.3f} ms")

Size: 100000, Time taken: 7.015 ms

Size: 1000000, Time taken: 75.408 ms

Size: 10000000, Time taken: 753.089 ms

Size: 50000000, Time taken: 3775.612 ms

Due to how time is increasing by a steady factor, this is an (O(n)) linear Big-O function.

1. Develop a test\_func(L) function that has a quadratic Big-O function (*O*(*n*2)). Provide that test function, the results of the testing, and a description supporting why you believe it to be a quadratic Big-O function.

def test\_func\_quadratic(L):  
 *"""Performs a quadratic operation on the list L."""* for i in range(len(L)):  
 for j in range(len(L)):  
 L[i] = L[i] \* 2  
  
  
# Testing with increasing sizes  
sizes = [100, 1000, 1020, 1050, 1150]  
for size in sizes:  
 L = list(range(size))  
 time\_taken = time\_function(test\_func\_quadratic, L)  
 print(f"Size: {size}, Time taken: {time\_taken \* 1000:.3f} ms")

Size: 100, Time taken: 0.918 ms

Size: 1000, Time taken: 101.374 ms

Size: 1020, Time taken: 104.773 ms

Size: 1050, Time taken: 127.988 ms

Size: 1150, Time taken: 158.436 ms

If plotted, these times resemble a quadratic parabola as the size of the list increases. The first for i in range operation counts as an O(n) operation, then the modifier for j in range multiplies this operation by a factor of n, resulting in O(n^2).