

School of Computing, Creative Technology and Engineering

**Module: Fundamentals of Computer Programming**

**Academic Year: 2023/24**

**Level 4: Semester 1**

**Assignment Title: Practical - 04**

**Date Due: Jan 16, 2025**

**Tutor: Saurav Gautam**

**Student Name: Sange Doma Tamang**

**Student ID: 10260**

GitHub link: <https://github.com/Sangedoma/Programming-Portfolio/tree/main/portfolio>

**Introduction to Programming**

**Lab Worksheet**

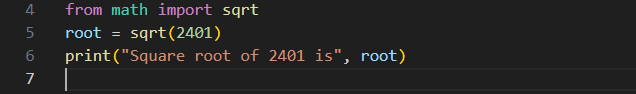
**Week 4**

**Importing and Using Functions**

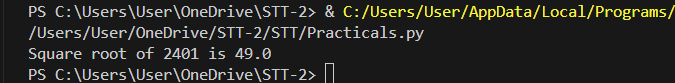
In our Python programs, we can import functions from modules in several different ways. First of all, we can import all functions that are defined within a module as follows - import math This would import all of the functions (and other elements) defined within the math module. Once imported, we can call the functions as usual. However, when the whole module has been imported (as above) we need to prefix the function name with the module name, e.g. result = math. Sin (6.2)

Task: Write some code that imports the math module, then calculates and prints the square root of the number 2401. Use the sqrt () function provided by the math module.

Code:

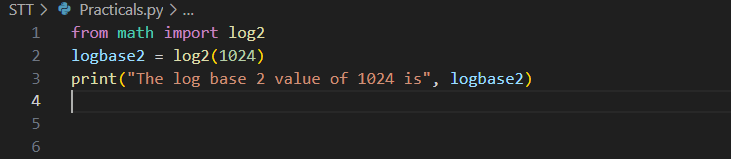


Output:

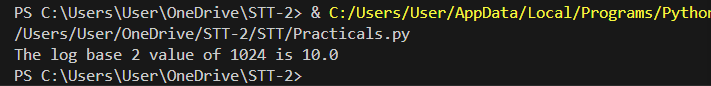


TASK: Write some code that imports only the log2() function from the math module, then call this function to calculate the log base 2 value of 1024. Print the result to the screen.

Code:

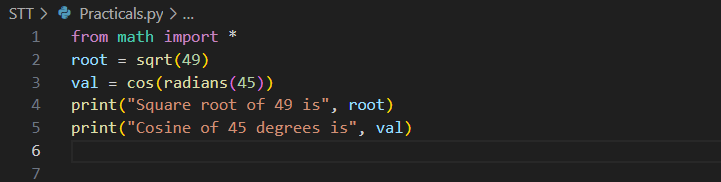


Output:

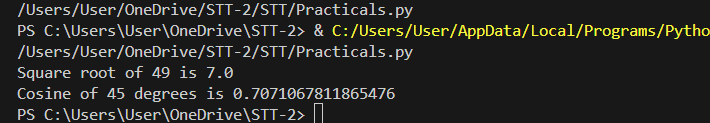


Finally, we can directly import all module content using a wildcard (\*). In this case all the functions can be called without the need for a prefix. This approach is not recommended since it pollutes the namespace, however it is convenient when using the interpreter in interactive-mode, like so:

Code:

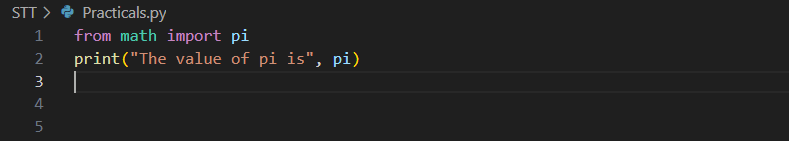


Output:

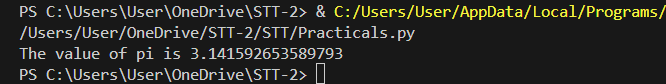


It is also worth noting that we can import other elements from modules as well as functions. For example, we can also import constant values:

Code:



Output:



**Defining Functions**

Within Python we define our own functions using the def keyword, followed by the function name, the names of the expected arguments (formal parameters), and the block of code to be executed when the function is called. For example, we could define a function that prints the given message out twice to the screen as follows -

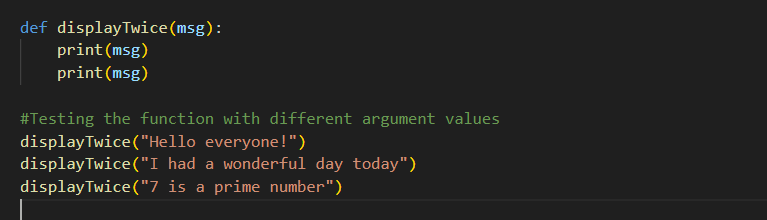
def displayTwice(msg):

print(msg)

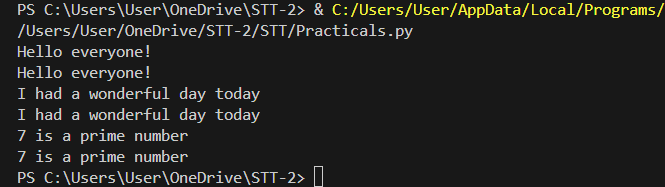
print(msg)

TASK: Input the above function definition. Once that is done make several calls to the function passing different argument values for testing.

Code:

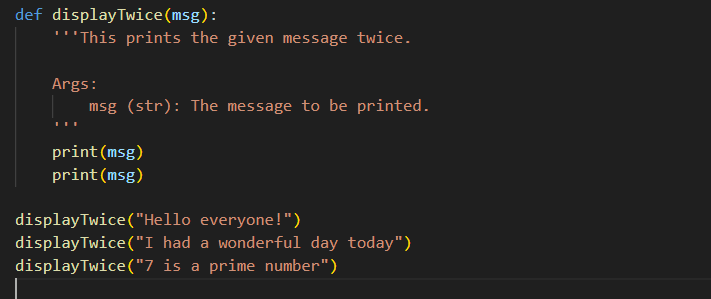


Output:



**Docstrings**

TASK: Re-Input the above function definition, but this time add a docstring that includes a description of the function’s purpose. Once that is done enter a command such as help(displayTwice) and see what it displays.



**Returning a value**

The following is an example of a function that returns a value.

def findMax(a,b):

"""Finds the maximum of two values."""

if ( a > b ):

max = a

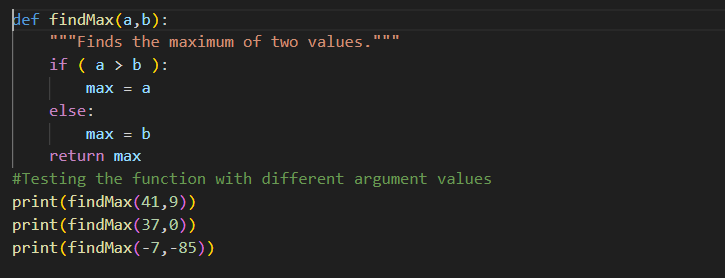
else:

max = b

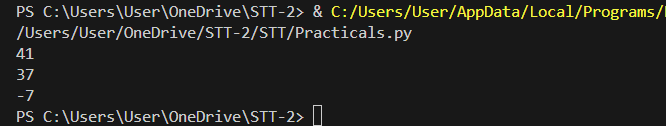
return max

TASK: Input the above function definition. Once that is done make several calls to the function passing different argument values and displaying the returned value.

Code:



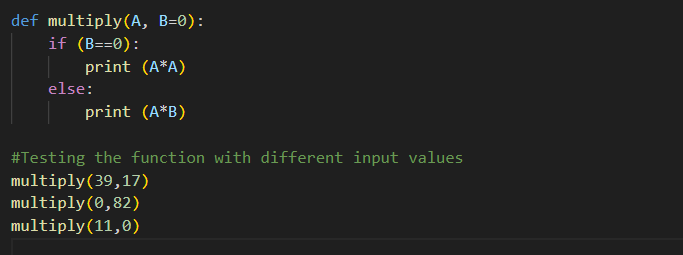
Output:



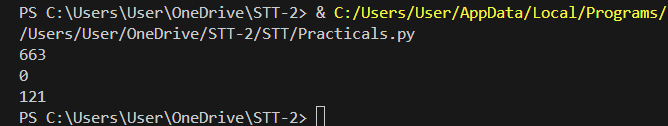
**Default Arguments**

TASK: Define a function that takes two numeric values, multiplies them together then returns the result. If the function is called with only a single argument however, then the value should be multiplied by itself. Once the function is defined, call it several times and display the returned values for testing purposes.

Code:



Output:



**Keyword Arguments**

Up until now all the function calls made have assumed that the passed formal parameter list appears in the same order as the actual parameters (arguments) specified within the function definition. So given the following function:

def someFunc(x, y, z):

print("x is", x, "\ny is", y, "\nz is", z)

The following call would result in:

someFunc(1,2,3)

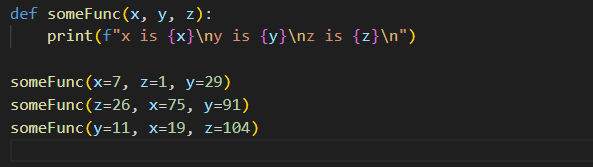
x is 1

y is 2

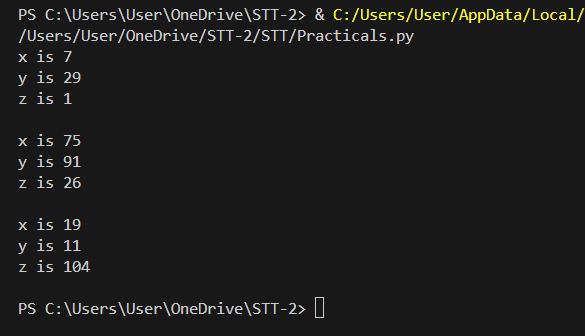
z is 3

TASK: Enter the example function shown above, then try calling it using the keyword arguments in several different orders.

Code:

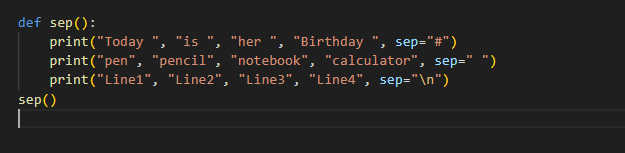


Output:

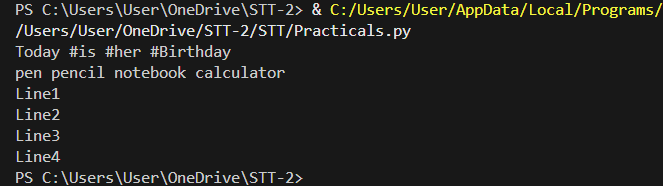


TASK: The built-in print() function supports a keyword argument called sep. This is used to decide what character to display between each of the provided positional parameters. Write some code that makes several calls to the print() function while setting the sep argument to values other than a space (which is the default).

Code:



Output:



**Arbitrary Length Argument Lists**

For example, we could define a function that calculates the average of the given parameters as follows:

def calcAve(\*numbers):

total = 0

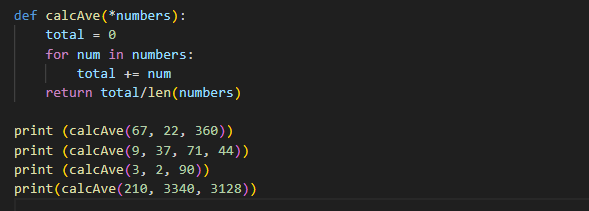
for num in numbers:

total += num

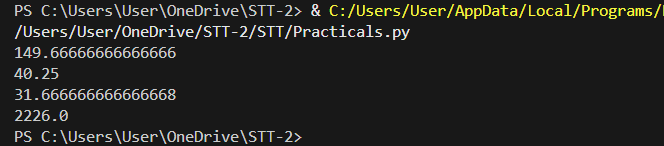
return total/len(numbers)

TASK: Enter the example function shown above, then try calling it several times, passing a different number of numeric arguments each time.

Code:



Output:



**Lambda Expressions**

The following example uses a lambda expression to define a simple anonymous function (this assumes the math module has already been imported).

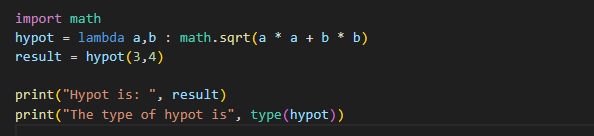
hypot = lambda a,b : math.sqrt(a \* a + b \* b)

Since this expression was assigned to the hypot variable it can now be called using that identifier, in the same way as a regular function:

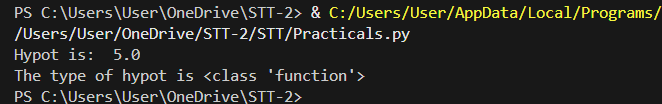
>>> hypot(3,4) 5.0

TASK: Enter the example lambda expression shown above, then find out the data type of the hypot variable using a call to the type () function. Notice the result.

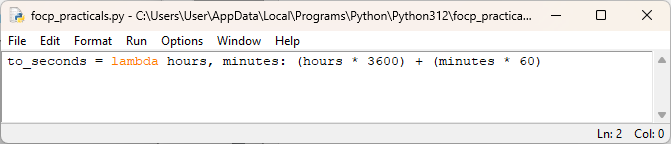
Code:



Output:



TASK: Write a lambda expression that takes two formal parameters, hours and minutes. The expression should calculate and return the total number of equivalent seconds. Assign the expression to a variable called to\_seconds, then call the function several times for testing.



Output:

