Massive Information & Knowledge Engineering

Functions
in Python

204113 Computer & Programming

Arnon Rungsawang
Dept. of computer engineering
Kasetsart University
https://mike.cpe.ku.ac.th/204113

Version 2014

What is a function?

- A block of code that performs a specific task.
- Suppose, you need to create a program to draw a circle and color it. We can create two functions to solve this problem:
  - create a circle function
  - create a color function
- Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Types of function

- There are two types of function in Python programming:
  - Standard library functions - These are built-in functions in Python that are available to use.

  - User-defined functions - We can create our own functions based on our requirements.

```
def my_function():
  print("Hello from a function")
```


Python function declaration

The syntax to declare a function is:

```
# global variable
def function_name(parameters):
    # function body
    return
```

Here,

- `def` – keyword used to declare a function
- `function_name` – any name given to the function
- `parameters` – any value passed to a called function
- `return` (optional) – returns value from a function

Example of a defined function

```
def greet():
    print('Hello World!')

greet()

# code
```

Shell

```
>>> %Run -c $EDITOR_CONTENT
  Hello World!
>>>
```

Here,

- When the function is called, the control of the program goes to the function definition.
- All codes inside the function are executed.
- The control of the program jumps to the next statement after the function call.

This function doesn't have any arguments and doesn't return any values.

https://www.programiz.com/python-programming/function

Function arguments

```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum: ",sum)

# function call with two values
add_numbers(5, 4)
# Output: Sum: 9
add_numbers(num2=4, num1=5)
# Output: Sum: 9
```

- A function can also have arguments.
- An argument is a value that is accepted by a function.
- If we create a function with arguments, we need to pass the corresponding values while calling them.
- Here, `add_number(5,4)` specifies that arguments `num1` and `num2` will get values 5 and 4 respectively.
- We can also call the function by mentioning the argument name, **not** in order, as illustrated by the last statement.

`return` statement

```
# function definition
def find_square(num):
    result = num * num
    return result

# function call
square = find_square(3)

print('Square:',square)

# Output: Square: 9
```

- A Python function may or may not return a value.
- If we want our function to return some value to a function call, we use the `return` statement.
- The `return` statement also denotes that the function has ended. Any code after return is not executed.
- In the example, we have created a function `find_square()`.
  - The function accepts a number and returns the square of that number.

Library functions

```
1  import math
2
3  # sqrt computes the square root
4  square_root = math.sqrt(4)
5
6  print("Square Root of 4 is",square_root)
7
8  # pow() comptes the power
9  power = pow(2, 3)
10
11 print("2 to the power 3 is",power)
```

- In Python, standard library functions are the built-in functions that can be used directly in our program. For example,
  - `print()` - prints the string inside the quotation marks
  - `sqrt()` - returns the square root of a number
  - `pow()` - returns the power of a number
- These library functions are defined inside the module. And, to use them we must include the module inside our program.
  - For example, `sqrt()` is defined inside the `math` module.

Benefits of using functions

```python
# function definition
def get_square(num):
    return num * num

for i in [1,2,3]:
    # function call
    result = get_square(i)
    print('Square of',i, '=',result)
```

1. Code Reusable

- We can use the same function multiple times in our program which makes our code reusable.
- In the example, we have created the function named `get_square()` to calculate the square of a number.
- Here, the function is used to calculate the square of numbers from 1 to 3.
- Hence, the same method is used again and again.

2. Code Readability

- Functions help us break our code into chunks to make our program readable and easy to understand.

Function argument with default values

```python
def add_numbers( a = 7,  b = 8):
    sum = a + b
    print('Sum:', sum)


# function call with two arguments
add_numbers(2, 3)

#  function call with one argument
add_numbers(a = 2)

# function call with no arguments
add_numbers()
```

- In Python, we can provide default values to function arguments.
- We use the = operator to provide default values.
- In the example, we have provided default values 7 and 8 for parameters a and b respectively.
  - **add_number(2, 3)** Both values are passed during the function call. Hence, these values are used instead of the default values.
  - **add_number(2)** Only one value is passed during the function call. So, according to the positional argument **2** is assigned to argument a, and the default value is used for parameter b.
  - **add_number()** No value is passed during the function call. Hence, default value is used for both parameters a and b.

Keyword argument

```python
def display_info(first_name, last_name):
    print('First Name:', first_name)
    print('Last Name:', last_name)

display_info(last_name = 'Cartman',
    first_name = 'Eric')
```

- In keyword arguments, arguments are assigned based on the name of arguments.
- In the example, we have assigned names to arguments during the function call.
- Here, `first_name` in the function call is assigned to `first_name` in the function definition. Similarly, `last_name` in the function call is assigned to `last_name` in the function definition.
- In such scenarios, the position of arguments doesn't matter.

Function with arbitrary arguments

```python
# program to find sum of multiple
    numbers

def find_sum(*numbers):
    result = 0

    for num in numbers:
        result = result + num

    print("Sum = ", result)

# function call with 3 arguments
find_sum(1, 2, 3)

# function call with 2 arguments
find_sum(4, 9)
```

- Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments.
- Arbitrary arguments allow us to pass a varying number of values during a function call.
- We use an asterisk (*) before the parameter name to denote this kind of argument in Python.
- Note in the example that after getting multiple values, `numbers` behave as an array (i.e., a tuple) so we are able to use the for loop to access each value.

Recursion?

- Recursion is the process of defining something in terms of itself.
- A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python recursive function

```
def recurse():
    ...
    recurse()
    ...

recurse()
```

recursive call

- In Python, we know that a function can call other functions.
- It is even possible for the function to call itself. These types of construct are termed as recursive functions.
- Left figure shows the working of a recursive function called `recurse`.

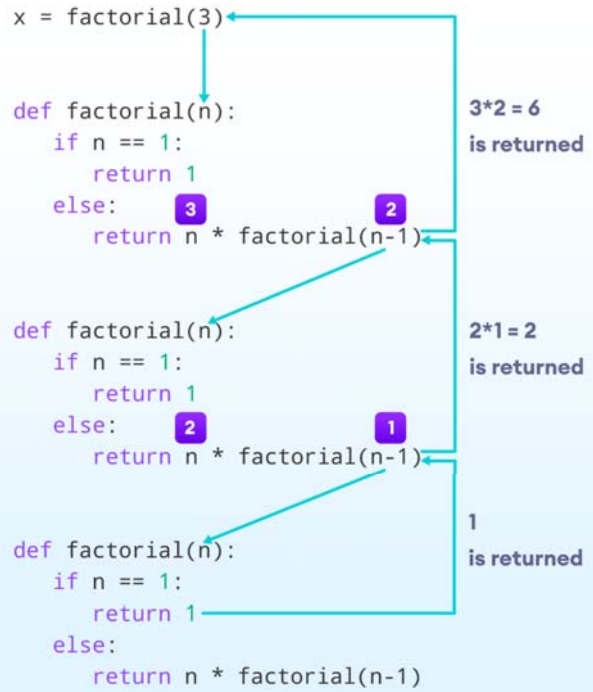
```
1 def recurse(n=5):
2     print(n, end=' ')
3     if n==1:
4         return
5     recurse(n-1)
6
7 recurse()
```


factorial()

```
1 def factorial(x):
2     """This is a recursive function
3     to find the factorial of an
          integer"""
4
5     if x == 1:
6         return 1
7     else:
8         return (x * factorial(x-1))
9
10
11  num = 3
12  print("The factorial of", num, "is",
      factorial(num))
```

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

- Factorial of a number is the product of all the integers from 1 to that number.
  - For example, the factorial of 6 (denoted as 6!) is 1*2*3*3*5*6 = 720.
- `factorial()` is a recursive function as it calls itself.
- When we call this function with a positive integer, it will recursively call itself by decreasing the number.
- Each function multiplies the number with the factorial of the number below it until it is equal to one.



Recursion depth

```
def recursor():
    recursor()
recursor()
```

Output

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

```
1 def recurse():
2     global n
3     n = n + 1
4     recurse()
5
6 n = 0
7 recurse()
```

- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000.
- If the limit is crossed, it result in `RecursionError`.

Pro vs. Con of recursion

- **Advantages of Recursion**

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

- **Disadvantages of Recursion**

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

`lambda` function

- In Python, a lambda function is a special type of function without the function name.

```
>>> lambda : print('Hello World')
<function <lambda> at 0x000001592FC2C430>
>>> a = lambda : print('Hello World')
>>> type(a)
<class 'function'>
>>> a
<function <lambda> at 0x000001592FC2C3A0>
>>> a()
Hello World
>>>
```

`lambda` function declaration

```
lambda arguments : expression
```

- We use `lambda` keyword instead of `def` to create a lambda function.
- Here,
  - `arguments` – any value passed to the lambda function.
  - `expression` – expression is executed and returned.

```
1 # lambda that accepts one argument
2 greet_user = lambda name : print('Hey there,',
      name)
3
4 # lambda call
5 greet_user('Delilah')
6
7 # greet_user()
```

- In the left example, we have defined a lambda function and assigned it to the `greet_user` variable.
- When we call the lambda function with a string argument, `print()` statement inside is executed.

Variable scope

- A variable scope specifies the region where we can access a variable.
- In Python, we can declare variables in three different scopes: local scope, global, and nonlocal scope.

```
def add_number():
    sum = 5 + 4
```

- Here, the `sum` variable is created inside the function, so it can only be accessed within it (local scope). This type of variable is called a local variable.

Local variables

```
1  def greet():
2
3      # local variable
4      message = 'Hello'
5
6      print('Local', message)
7
8  greet()
9
10 # try to access message variable
11 # outside greet() function
12 print(message)
```

- When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function.
- These types of variables are called local variables.
- In the left example, `message` variable is local the `greet()` function, so it can only be accessed within the function.
- That's why we get an error (`NameError` in the example) when we try to access it outside the `greet()` function.

Global variables

```
1  # declare global variable
2  message = 'Hello'
3
4  def greet():
5      # declare local variable
6      # message = 'CPE'
7      print('Local', message)
8      # message = 'CPE'
9
10 greet()
11 print('Global', message)
```

- In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.
- In the left example this time we can access the `message` variable from outside of the `greet()` function. This is because we have created the `message` variable as the global variable.

Nonlocal variables

```
1  # message = 'global'
2  # outside function
3  def outer():
4      # local to outer
5      message = 'local'
6
7      # nested function
8      def inner():
9
10         # declare nonlocal variable
11         nonlocal message
12
13         message = 'nonlocal'
14         print("inner:", message)
15
16     inner()
17     print("outer:", message)
18
19 outer()
20 print('main:', message)
```

- In Python, nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.
- In the left example, there is a nested `inner()` function. We used the `nonlocal` keyword to create a nonlocal variable.
- The `inner()` function is defined in the scope of another function `outer()`.
- Note that if we change the value of a nonlocal variable, the changes appear in the local variable.

global keyword

```
# global variable
c = 1

def add():
    print(c)

add()

# Output: 1
```

- `global` keyword allows to modify the variable outside of the current scope.
- It is used to create a global variable and make changes to the variable in a local context.
- In the left above example, we have accessed a global variable from the inside of a function.

```
# global variable
c = 1

def add():

    # increment c by 2
    c = c + 2

    print(c)

add()
```

- However, if we try to modify the global variable from inside a function, we will get an error.
- This is because we can only access the global variable but cannot modify it from inside the function.

```
Traceback (most recent call last):
  File "<string>", line 11, in <module>
  File "<string>", line 7, in add
ERROR!
UnboundLocalError: cannot access local
    variable 'c' where it is not associated
    with a value
```


Changing global variable inside a function

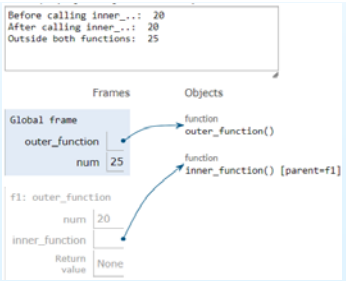
```
# global variable
c = 1

def add():
    # use of global keyword
    global c
    # increment c by 2
    c = c + 2

    print(c)

add()
```

- We here have defined `c` with `global` keyword inside `add()`.
- Then, we have incremented the variable `c` by 2, i.e., `c = c + 2`
- As we can see while calling `add()`, the value of the global variable `c` is modified from 1 to 3.

`global` in nested functions

```
def outer_function():
    num = 20

    def inner_function():
        global num
        num = 25

    print('Before calling inner_..: ', num)
    inner_function()
    print('After calling inner_..: ', num)

outer_function()
print('Outside both functions: ', num)
```

```
Before calling inner_..:  20
After calling inner_..:  20
Outside both functions:  25
```



- Inside `outer_function()`, `num` has no effect of the `global` keyword.
- Before and after calling `inner_function()`, `num` takes the value of the local variable, i.e., num=20.
- Outside of the `outer_function()` function, `num` will take the value defined in the `inner_function()` function, i.e., num=25.
- This is because we have used the `global` keyword to create a global variable `num` inside the `inner_function()` function (local scope). So, change inside the `inner_function()` has the effect on `num` outside the `outer_function()`.

Rules of `global` keyword

The basic rules for `global` keyword in Python are:

- When we create a variable inside a function, it is local by default.
- When we define a variable outside of a function, it is global by default. We don't have to use the `global` keyword.
- We use the `global` keyword to read and write a global variable inside a function.
- Use of the `global` keyword outside a function has no effect.

Function as an argument of a function

```python
from math import sin,cos,sqrt,pi

def my_fn_plus_one(f, x):
  return f(x) + 1

print(my_fn_plus_one(sin, pi/2))
print(my_fn_plus_one(cos, pi/2))
print(my_fn_plus_one(sqrt, 25))

a = my_fn_plus_one(lambda x: x + 2, 2)
print(a)
```


Sample Problem Solving

compute_circle_area

```python
import math
def compute_circle_area(radius):
    circle_area = math.pi*radius**2
    return circle_area

r = float(input("Enter a radius: "))
area = compute_circle_area(r)
print(f"Area of the circle is {area:.2f}")
```


myadder

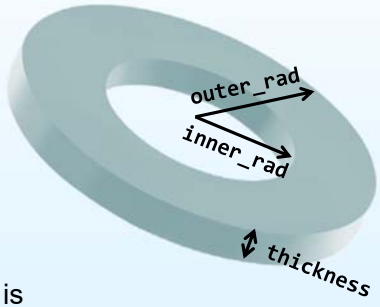
```python
def my_adder(a, b, c):
  # variable out has local scope in my_adder() fn
  out = a + b + c
  print(f'The value out within the function is {out}')
  return out

## main begins here
out = 1 # here variable out is in the global scope
# now we call the fn to see whether variable out changes
d = my_adder(1, 2, 3)
# verify that the variable out does not change
print(f'The value out outside the function is {out}')
# print(a,b,c)
```

Task: Flat_washers

- You work for a hardware company that manufactures flat washers. To estimate shipping costs, your company needs a program that computes the weight of a specified quality of flat washers.

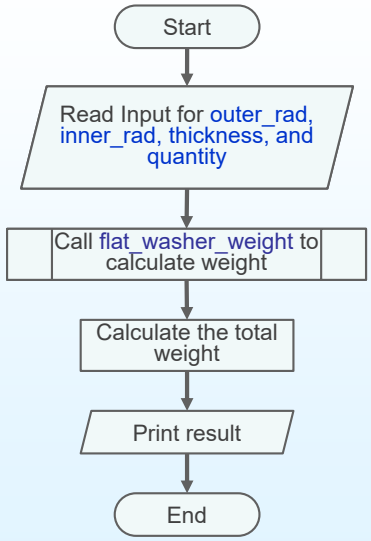
Flat_washers – Idea

- A flat washer resembles a small donut (see the figure).
- To compute the weight of a single flat washer, you need to know its *rim area*, *thickness*, and *density* of the material
  - Here, we can reuse `compute_circle_area()` function
- Requirements:
  - Radius of flat washer and hole
  - Thickness
  - Density
  - Quantity
- We will assume that the material used is aluminum, whose density is well-known.



Flat_washers – Steps

- Get the washer's outer radius, inner radius, thickness, and quantity
- Compute the weight of one flat washer
  - unit_weight = rim_area × thickness × density
- Compute the weight of batch of washers
  - total_weight = unit_weight × quantity
- Print the resulting weight of batch.



Flat Washers – Program

Notice how the variable MATERIAL_DENSITY is defined and used as a global variable

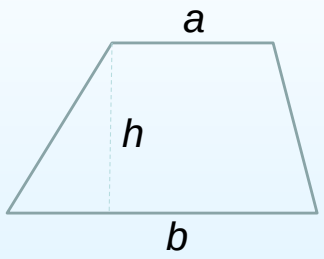
```
 1: import math
 2:
 3: MATERIAL_DENSITY = 2.70     # in g/cc
 4:
 5: def compute_circle_area(radius):
 6:     return math.pi*radius**2
 7:
 8: def flat_washer_weight(outer_r,inner_r,thickness):
 9:     rim_area=compute_circle_area(outer_r)-compute_circle_area(inner_r)
10:     return rim_area*thickness*MATERIAL_DENSITY
11:
12: outer_rad = float(input('Enter the outer radius (cm.): '))
13: inner_rad = float(input('Enter inner radius (cm.): '))
14: thickness = float(input('Enter thickness (cm.): '))
15: quantity = int(input('Enter the quantity (pieces): '))
16: unit_weight = flat_washer_weight(outer_rad,inner_rad,thickness)
17: total_weight = unit_weight * quantity
18: print(f'Weight of the batch is {total_weight:.2f} grams')
```

```
Enter the outer radius (cm.): 15
Enter inner radius (cm.): 10
Enter thickness (cm.): 3
Enter the quantity (pieces): 10
Weight of the batch is 31808.63 grams
```

Task: Trapezoid

- In Euclidean geometry, a convex quadrilateral with at least one pair of parallel sides is referred to as a trapezoid.

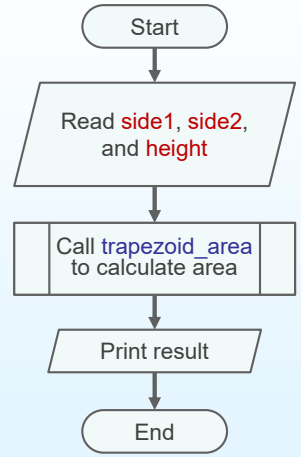
(ref: https://en.wikipedia.org/wiki/Trapezoid)



$$area = \frac{a + b}{2} h$$

Trapezoid – Steps

- Get three double values from the user:
  - (parallel) side1
  - (parallel) side2
  - height
- Calculate the trapezoid area
  - area = ((side1 + side2)/2) × height
- Print the resulting area



Trapezoid – Program

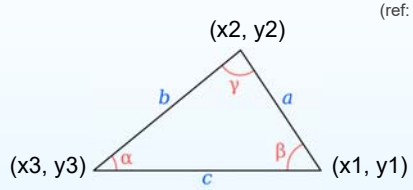
```
1:  def read_trapezoid():
2:      print("Enter the properties of your trapezoid.")
3:      a = float(input("Length of parallel side 1: "))
4:      b = float(input("Length of parallel side 2: "))
5:      h = float(input("Height: "))
6:      return a,b,h
7:
8:  def trapezoid_area(a,b,h):
9:      return 0.5*(a+b)*h
10:
11: # main program
12: a,b,h = read_trapezoid()
13: area = trapezoid_area(a,b,h)
14: print(f"Trapezoid's area is {area:.2f}")
```

```
Enter the properties of your trapezoid.
Length of parallel side 1: 10
Length of parallel side 2: 15
Height: 13
Trapezoid's area is 162.50
```


Task: Triangle Area (Heron)

- In geometry, Heron's formula (sometimes called Hero's formula), named after Hero of Alexandria, gives the area of a triangle by requiring no arbitrary choice of side as base or vertex as origin, contrary to other formulas for the area of a triangle, such as half the base times the height or half the norm of a cross product of two sides.

(ref: https://en.wikipedia.org/wiki/Heron's_formula)



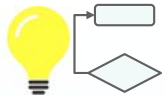
- Heron's formula states that the area of a triangle whose sides have lengths a, b, and c is

$$area = \sqrt{s(s-a)(s-b)(s-c)},$$

where s is the semiperimeter of the triangle; that is,

$$s = \frac{a + b + c}{2}$$

Triangle Area (Heron) – Ideas + Step



- Get the x-y coordinate of the triangle's 3 vertices
- Calculate the length of the lines *a*, *b*, and *c* which are connected to those 3 vertices
- Calculate the semiperimeter
- Calculate the triangle's area using the Heron's formula
- Print the resulting area

Triangle Area (Heron) – Program

```
1:  import math
2:
3:  def line_length(x1, y1, x2, y2):
4:      """
5:      Given X-Y coordiates of 2 points, compute the line length that
6:      joins them
7:      """
8:      return math.sqrt((x1-x2)**2+(y1-y2)**2)
9:
10: def triangle_area(x1, y1, x2, y2, x3, y3):
11:     """
12:     Given the 3 vertices, compute triangle area using Heron's Formula
13:     """
14:     a = line_length(x1, y1, x2, y2)
15:     b = line_length(x2, y2, x3, y3)
16:     c = line_length(x3, y3, x1, y1)
17:     s = (a+b+c)/2
18:     return math.sqrt(s*(s-a)*(s-b)*(s-c))
```

(The conde continues on the next page)

Triangle Area (Heron) – Program

```
Enter X-Y coordinates of the three vertices of triangle:
1st vertex:
x? 1
y? 1
2nd vertex:
x? 3
y? 2
3rd vertex:
x? 2
y? 3
area of the triangle is 1.50
```

```
19: def read_coordinates():
20:     x = float(input("x? "))
21:     y = float(input("y? "))
22:     return x,y
23:
24: def read_triangle():
25:     """
26:     Read X-Y co-ordinates of 3 vertices of a triangle
27:     """
28:     print("Enter X-Y coordinates of the three vertices of triangle:")
29:     print("1st vertex:")
30:     x1,y1 = read_coordinates()
31:     print("2nd vertex:")
32:     x2,y2 = read_coordinates()
33:     print("3rd vertex:")
34:     x3,y3 = read_coordinates()
35:     return x1,y1,x2,y2,x3,y3
36:
37: x1,y1,x2,y2,x3,y3 = read_triangle()
38: area = triangle_area(x1,y1,x2,y2,x3,y3)
39: print(f"area of the triangle is {area:.2f}")
```


Distance between 2D points

```
1  from math import sqrt
2
3  def my_dist_xyz(x, y, z):
4    """
5    x, y, z are 2D coordinates contained in a tuple
6    output:
7    d - list, where
8        d[0] is the distance between x and y
9        d[1] is the distance between x and z
10       d[2] is the distance between y and z
11   """
12   d0 = sqrt((x[0]-y[0])**2+(x[1]-y[1])**2)
13   d1 = sqrt((x[0]-z[0])**2+(x[1]-z[1])**2)
14   d2 = sqrt((y[0]-z[0])**2+(y[1]-z[1])**2)
15   return [d0, d1, d2]
16
17 ## main begins here
18 d = my_dist_xyz((0, 0), (0, 1), (1, 1))
19 print(d)
```

Distance between 2D points (2)

```python
def my_dist_xyz(x, y, z):
  '''x, y, z are tuples represent the point in 3D
    output - list, where
      d[0] is the distance between x and y
      d[1] is the distance between x and z
      d[2] is the distance between y and z
  '''
  def my_dist(x, y):
    '''x and y are tuples represent the point in 2D'''
    res = (x[0]-y[0])**2 + (x[1]-y[1])**2
    return res ** (1/2)
  d0 = my_dist(x, y)
  d1 = my_dist(x, z)
  d2 = my_dist(y, z)
  return [d0, d1, d2]

## main begins here
d = my_dist_xyz((0, 0), (0, 1), (1, 1))
print(d)
a,b,c = d
print(f'a={a:.2f}, b={b:.2f} and c={c:.2f}')
# d = my_dist((0, 0), (0, 1))
```


To be continue..

つづく