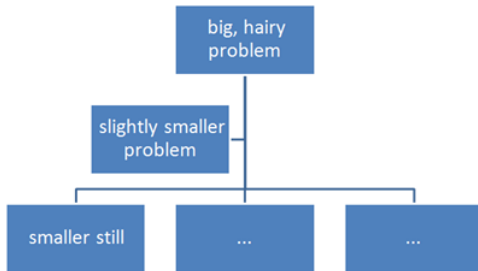


FUNCTIONS



TRIANGLE FORMATION PROBLEM

Given three points, write an algorithm and the subsequent Python code to check if they can form a triangle. Three points can form a triangle, if they do not fall in a straight line and length of a side of triangle is less than the sum of length of other two sides of the triangle.

For example, the points (5,10), (20,10) and (15,15) can form a triangle as they do not fall in a straight line and length of any side is less than sum of the length of the other two sides

PSEUDO CODE FOR TRIANGLE FORMATION

Read the three points

If the three points fall on a straight line then print "No Triangle" and break

Otherwise find length of all three sides

If length of one side is greater than the sum of length of the other two sides then print "Triangle" and print "No Triangle" otherwise

PSEUDO CODE FOR FALL IN STRAIGHT LINE

input : X and Y coordinates of the three points

IF (pt1.x==pt2.x==pt3.x) THEN

RETURN true

ELIF (pt1.y==pt2.y==pt3.y) THEN

RETURN true

ELSE

RETURN false

PSEUDOCODE FOR DISTANCE BETWEEN TWO POINTS (LENGTH OF A SIDE IN A TRIANGLE)

```
input : X and Y coordinates of the two points
Distance = sqrt((pt1.x-pt2.x)**2 - (pt1.y-pt2.y)**2)
Return distance
```

PSEUDOCODE FOR CHECKING LENGTH CONSTRAINT

```
input : Length of three sides l1, l2, and l3
if l1 > l2+l3 or l2 > l1+l3 or l3 > l1+l2:
    return false
else
    return true
```

Bigger Problems

- If you are asked to find a solution to a major problem, it can sometimes be very difficult to deal with the complete problem all at the same time.
- For example building a car is a major problem and no one knows how to make every single part of a car.
- A number of different people are involved in building a car, each responsible for their own bit of the car's manufacture.
- The problem of making the car is thus broken down into smaller manageable tasks.
- Each task can then be further broken down until we are left with a number of step-by-step sets of instructions in a limited number of steps.
- The instructions for each step are exact and precise.

Top Down Design

- Top Down Design uses the same method to break a programming problem down into manageable steps.
- First of all we break the problem down into smaller steps and then produce a Top Down Design for each step.
- In this way sub-problems are produced which can be refined into manageable steps.

Top Down Design for Real Life Problem

PROBLEM: To repair a puncture on a bike wheel

ALGORITHM:

- 1. remove the tyre
- 2. repair the puncture
- 3. replace the tyre

Step 1: Refinement:

1.Remove the tyre

- 1.1 turn bike upside down
- 1.2 lever off one side of the tyre
- 1.3 remove the tube from inside the tyre

Step 2: Refinement:

2.Repair the puncture Refinement:

- 2.1 find the position of the hole in the tube
- 2.2 clean the area around the hole
- 2.3 apply glue and patch

Step 3: Refinement:

3.Replace the tyre Refinement:

- 3.1 push tube back inside tyre
- 3.2 replace tyre back onto wheel
- 3.3 blow up tyre
- 3.4 turn bike correct way up

Still more Refinement:

MORE REFINEMENT

Sometimes refinements may be required to some of the sub-problems, for example if we cannot find the hole in the tube, the following refinement can be made to 2.1:-

Step 2.1: Refinement

2.1 Find the position of the hole in the tube

- 2.1.1 WHILE hole cannot be found
- 2.1.2 Dip tube in water
- 2.1.3 END WHILE

Python Functions

- A function has a name that is used when we need for the task to be executed.
- Asking that the task be executed is referred to as "calling" the function.
- Some functions need one or more pieces of input when they are called. Others do not.
- Some functions give back a value; others do not. If a function gives back a value, this is referred to as "returning" the value.

Why Functions?

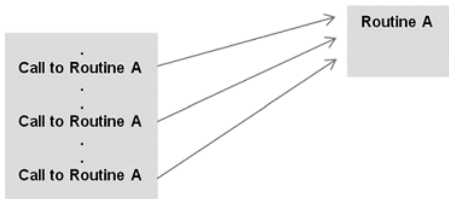
- To reduce code duplication and increase program modularity.
- A software cannot be implemented by any one person, it takes a team of programmers to develop such a project.

Term	Number of Lines of code (LoC)	Equivalent Storage
KLOC	1000	Application Programs
MLOC	1000000	Operating Systems / smart phones
GLOC	1000000000	Number of lines of code in existence for various programming languages

- In order to manage the complexity of a large problem, it is broken down into smaller sub problems.
- Then, each sub problem can be focused on and solved separately.
- Program routines provide the opportunity for code reuse, so that systems do not have to be created from "scratch"

What Is a Function Routine?

- A **function** or **routine** is a named group of instructions performing some task.
- A routine can be **invoked** (called) as many times as needed in a given program
- When a routine terminates, execution automatically returns to the point from which it was called.



Defining Functions

- Functions may be designed as per user's requirement.
- The elements of a function definition are given

```

Function Header ➤ def avg(n1, n2, n3):
Function Body   { -----
(suite)         { -----
                { -----
                { -----
    
```

- The number of items in a parameter list indicates the number of values that must be passed to the function, called actual arguments (or simply "arguments"), such as the variables num1, num2, and num3 below.
- `>>> num1 = 10`
- `>>> num2 = 25`
- `>>> num3 = 16`
- `>>> avg(num1, num2, num3)`
- Every function must be defined before it is called.

Parameters

- **Actual parameters:**

- we call simply → arguments
- values passed to functions to be operated on.

- **Formal parameters**

- we call simply → placeholder
- the placeholders names for the arguments passed.

- Actual parameters are matched with formal parameters by following the assignment rules

Operation	Interpretation
spam = 'Spam'	Basic Form
spam, ham = 'yum', 'YUM'	Tuple Assignment (Positional)
[spam, ham] = ['yum', 'YUM']	List Assignment (Positional)
a, b, c, d = 'spam'	Sequence Assignment (Generalized)
a, *b = 'spam'	Extended Sequence unpacking
spam = ham = 'lunch'	Multiple-target Assignment
spams += 42	Augmented Assignment (Equivalent to spams = spams + 42)

Assignment statement recap

- `>>> [a,b,c] = (1,2,3) → Assign tuple of values to list of names`
- `>>> a, c → check output → (1,3)`
- `>>> (a,b,c) = "ABC" → Assign string of characters to tuple`
- `>>> a, c → check output → ('A', 'C')`
- `>>> seq = [1,2,3,4]`
- `>>> a,b,c,d = seq`
- `>>> print(a,b,c,d) → check output → 1 2 3 4`
- `>>> a, b = seq → ValueError: too many values to unpack (expected 2)`
- `>>> a, *b = seq`
- `>>> a → check output → 1`
- `>>> b → check output → [2,3,4]`
- `>>> *a, b = seq`
- `>>> a → check output → [1,2,3]`
- `>>> b → check output → 4`

Assignment statement recap

- `>>> a, *b, c = seq`
- `>>> a` → check output → 1
- `>>> b` → check output → [2,3]
- `>>> c` → check output → 4
- `>>> a, *b = 'spam'`
- `>>> a, b` → check output → ('s', ['p','a'],'m')
- `>>> a, *b, c = 'spam'`
- `>>> a, b, c` → check output → ('s',['p','a'],'m')
- `>>> a, *b, c = range(4)`
- `>>> a, b, c` → check output → (0,[1,2],3)

Example Functions

- **def printer(message):**
 print('Hello' + message)
- **def adder(a,b=1,*c):**
 return(a+b+c[0])
- **def times(x,y):** → creates and assign function
 return x * y → Body executed when called
- When Python reaches and runs this def, it creates a new function object that packages the function's code and assigns the object to the name times.

Calls

- **>>> times(2,4)** → Arguments in parenthesis → 8
- **>>> x = times(3.14,4)** → saved the result object-x → 12.56
- **>>> times('Ni',4)** → Functions are typeless → check out →
 'NiNiNiNi'

FUNCTIONS

```
def intersect(seq1, seq2):  
    res = [] # start empty  
    for x in seq1: #scan seq1  
        if x in seq2: # common item  
            res.append(x) #add to end return  
    res
```

- >>> s1 = "SPAM"
- >>> s2 = "SCAM"
- >>> intersect(s1,s2) → output → ['S', 'A', 'M']

Equivalent Comprehension

- >>> [x for x in s1 if x in s2] → check output → ['S','A','M']
- works for list also:
- x = intersect([1,2,3],(1,4)) → mixed type data → check output → [1]

Def Statements

- creates a function object and assigns it to a name
- **def is a true executable statement:** when it runs, it creates a new function object and assigns it to a name
- Because its a statement, a def can **appear anywhere** a statement can even nested in other statements

Def Statements

if test:

 def func(): # Define func this way

else:

 def func(): # Or else this way...

func() # Call the version selected and built

Function definition in selection statement Example

```
a = 4
if a%2==0:
    def func():
        print ('even')
else:
    def func():
        print('odd')
func()
```

Output:

even

Function definition in selection statement Example

```
a = 4
if a%2==0:
    def func():
        print ('even')
else:
    def func():
        print1('odd')
```

error no function print1 is defined

```
func()
```

Error in only condition not satisfied item is found.
Otherwise code execute normal

Output:

even

Function definition in selection statement Example

```
a = 5
if a%2==0:
    def func():
        print ('even')
else:
    def func():
        print1('odd') # error no function print1 is defined
func()
```

Output:

error

Function Call through variable

```
def one():  
    print('one')  
def two():  
    print('two')  
def three():  
    print('three')
```

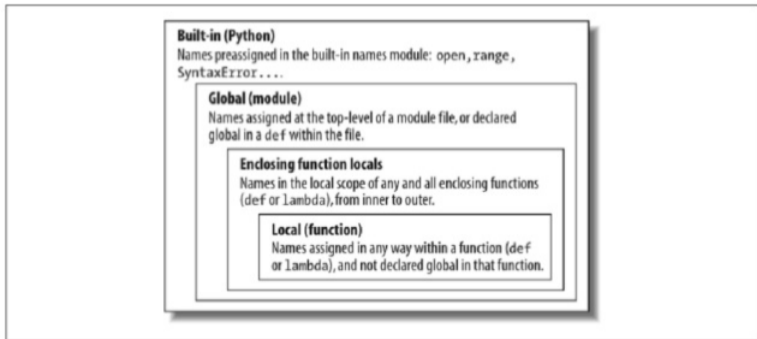
Example

```
a = 3  
if a == 1:  
    call_Func=one  
elif a == 2:  
    call_Func=two  
else:  
    call_Func=three  
call_Func()
```

Scope of Variables

- Enclosing module is a global scope
- Global scope spans a single file only
- Assigned names are local unless declared global or nonlocal
- Each call to a function creates a new local scope

Name Resolution: The LEGB Rule



The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned usually determines its scope. In Python 3, nonlocal declarations can also force names to be mapped to enclosing function scopes, whether assigned or not.

Scope Example

- # Global scope
X = 99
X and func assigned in module: global
- def func(Y):
 # Y and Z assigned in function: locals
 # Local scope
 Z = X + Y # X is a global
 return Z
func(1) # func in module: result=100
- Global names: X, func
- Local names: Y, Z
X = 88 # Global X
def func():
 X = 99
 # Local X: hides global
func()
print(X) # Prints 88: unchanged

Accessing Global Variables

```
X = 88      # Global X
def func():
    global X
    X = 99   # Global X: outside def
func()
print(X)    # Prints 99
```

Accessing Global Variables

```
y, z = 1, 2      # Global variables in module
def all_global():
    global x      # Declare globals assigned
    x = y + z
# No need to declare y, z: LEGB rule
```

Global Variables and Global Scope

- The use of global variables is generally considered to be bad programming style.

Nested Functions

```
X = 99      # Global scope name: not used
def f1():
    X = 88   # Enclosing def local
    def f2():
        print(X)
    # Reference made in nested def
    f2()
f1()         # Prints 88: enclosing def local
```

Return Functions

- Following code defines a function that makes and returns another function
- ```
def f1():
 X = 88
 def f2():
 print(X)
 # Remembers X in enclosing def scope
 return f2 # Return f2 but don't call it
action = f1() # Make, return function
action() # Call it now: prints 88
```

## Value-Returning Functions

- Program routine called for its return value, and is therefore similar to a mathematical function.
- Function `avg` takes three arguments (`n1`, `n2`, and `n3`) and returns the average of the three.
- The function call `avg(10, 25, 16)`, therefore, is an expression that evaluates to the returned function value.
- This is indicated in the function's return statement of the form `return expr`, where *expr* may be any expression.

## Function Definition

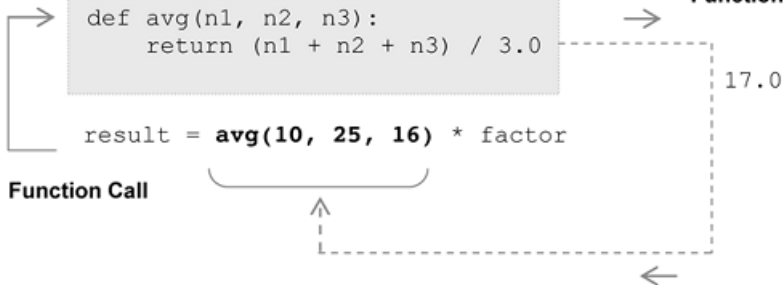
```
def avg(n1, n2, n3):
 return (n1 + n2 + n3) / 3.0
```

## Function Value

17.0

```
result = avg(10, 25, 16) * factor
```

## Function Call






## Non-Value-Returning Functions

- A non-value-returning function is called not for a returned value, but for its side effects.
- A side effect is an action other than returning a function value, such as displaying output on the screen.

### Function Definition



```
def displayWelcome():
 print('This program will convert between Fahrenheit and Celsius')
 print('Enter (F) to convert Fahrenheit to Celsius')
 print('Enter (C) to convert Celsius to Fahrenheit')

main
.
displayWelcome()
```

- In this example, function display Welcome is called only for the side-effect of the screen output produced.

## Returning Multiple Values

- ```
>>> def multiple(x, y):  
    x = 2      # Changes local names only  
    y = [3, 4]  
    return x, y  
# Return multiple new values in a tuple
```

Thank
you