
Subroutines

ECE 476 Advanced Embedded Systems

Jake Glower - Lecture #4

Please visit [Bison Academy](#) for corresponding lecture notes, homework sets, and solutions



Introduction:

Subroutines are programs you can call from other programs.

These go by various names

- Functions in Matlab
- Subroutines in C
- Definitions in Python

They all serve the same purpose:

- Break your program into smaller routines which can be tested
 - supports bottom-up and top-down programming
- Allow you to reuse code from program to program.

This lecture looks at

- How subroutines are defined in MicroPython and
 - How to return parameters to the main routine.
-

Subroutines in MicroPython

Subroutines are defined by the keyword *def*

- short for *define*

The simplest example would be a routine which

- is passed nothing,
- returns nothing, and
- simply prints 'hello' when called:

When you press the *run* command

- Python installs the subroutine *SayHello*
- It then runs the main routine
 - instruction following all of the definitions



The image shows a MicroPython IDE interface. At the top, there are several icons: a folder icon, a document icon labeled 'Open', a floppy disk icon labeled 'Save', a play button icon labeled 'Run', a red octagon icon labeled 'Stop', and a yellow square icon. Below these icons is a code editor window with the following Python code:

```
def SayHello():  
    print('hello')  
  
# Start of main routine  
SayHello()
```

shell

```
>>>  
hello
```

In this example, note that

- The subroutine is called *SayHello*
- Nothing is passed
 - Indicated by ()
- The definition is terminated with a colon
- The code within the subroutine is indented
 - Required Python standard

Also note

- *SayHello()* can be called from the shell window



```
def SayHello():  
    print('hello')  
  
# Start of main routine  
SayHello()
```

shell

```
>>>  
hello  
  
>>> SayHello()  
hello
```

Passing Parameters


You can pass parameters to subroutines.

Example: Display numbers from 1..N

- CountToN(N): Receives a number (N)
- N is used in a for-loop

The main routine passes the number 5

- Count from 1 to 5



```
def CountToN(N) :  
    for i in range(1,N+1) :  
        print(i)  
  
# Start of main routine  
CountToN(5)
```

shell

```
>>>  
1  
2  
3  
4  
5
```

Passing Multiple Parameters


You can pass multiple parameters

- Include them in the definition

Example: Write a subroutine which multiplies two numbers

- `Multiply(A, B):`

You can also call this subroutine from the shell window



```
def Multiply(A, B):  
    C = A * B  
    print(A, ' * ', B, ' = ', C)  
  
# Start of main routine  
Multiply(4, 6)
```

shell

```
>>>  
4 * 6 = 24  
  
>>> Multiply(8, 7)  
8 * 7 = 56
```

Returning One Parameter

Subroutines in Python variables.

That variable could be

- An array,
- A matrix, or
- A class object

Example: Return one parameter

- Indicated with *return(C)*

You can also call the subroutine from the shell window



```
# Example of Returning One Number
def Multiply(A, B):
    C = A * B
    return(C)

# Start of main routine
X = Multiply(4, 6)
print(X)
```

shell

```
>>>
24

>>> C = Multiply(8, 7)
>>> print(C)
56
```


Returning Several Parameters

Several parameters can be returned

- Pass them in an array

When you receive the array

- Each element can be pulled out separately



```
# Example of Returning four Numbers
def Operate(A, B):
    C0 = A + B
    C1 = A - B
    C2 = A * B
    C3 = A / B
    return([C0, C1, C2, C3])

# Start of main routine
X = Operate(4,6)
print(X)
```

shell

```
>>>
[10, -2, 24, 0.666667]

>>> C = Operate(8,7)
>>> print(C)
[15, 1, 56, 1.4142857]
```

Returning Several Parameters (cont'd)

You can also receive several parameters separately

- as four separate variables

or as an array

- and pull elements out separately



```
# Example of Returning Four Numbers
def Operate(A, B):
    C0 = A + B
    C1 = A - B
    C2 = A * B
    C3 = A / B
    return(C0, C1, C2, C3)

# Start of main routine
a, b, c, d = Operate(4,6)
print(a, b, c, d)
```

shell

```
>>>
10, -2, 24, 0.666667

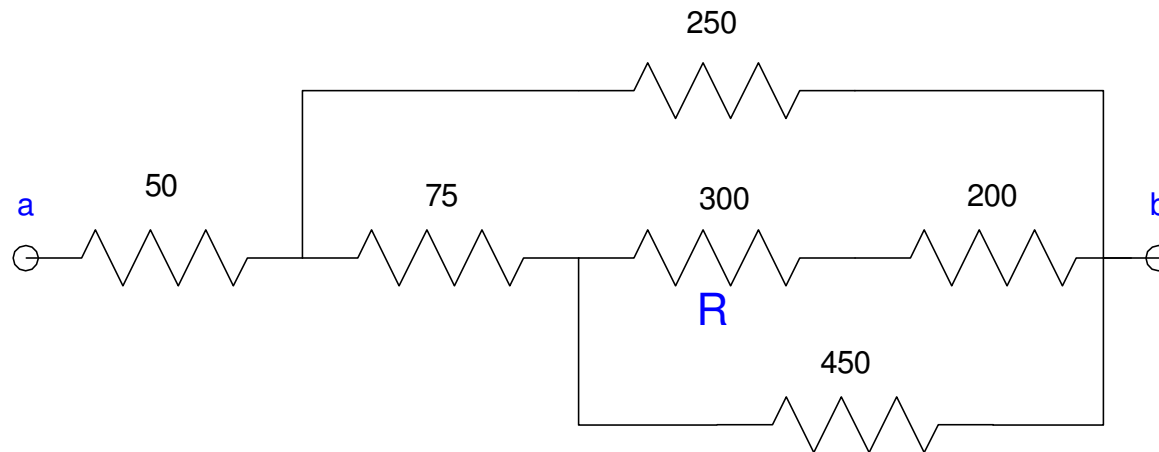
>>> a, b, c, d = Operate(8,7)
>>> print(a, b, c, d)
15, 1, 56, 1.4142857

>>> a = Operate(8,7)
>>> print(a)
(15, 1, 56, 1.4142857)

>>> print(a[2])
56
```

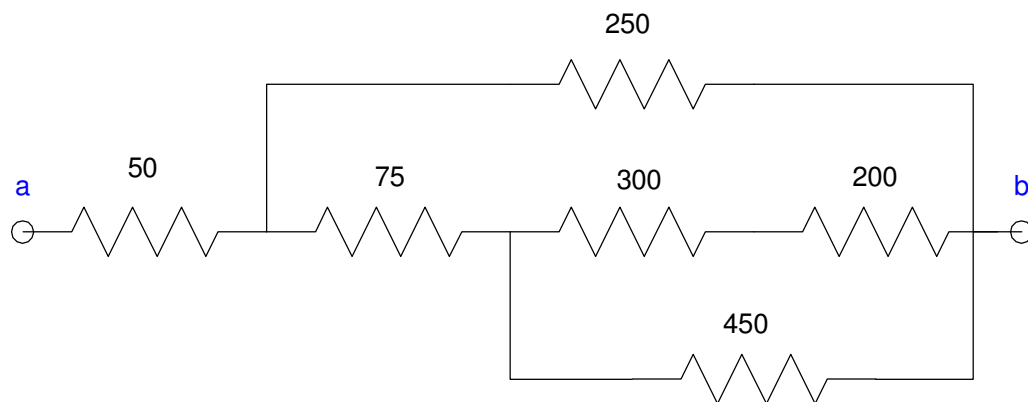
Fun with Subroutines: Resistors in Series & Parallel

As an example of where subroutines can be useful, let's write routines to add resistors in series and parallel. Using those routines, write a third routine to solve for R_{ab} if R is changed from 300 Ohms



Combine resistors in series and parallel to find Rab:

- Same as before, $R_{ab} = 188.7588$ Ohms



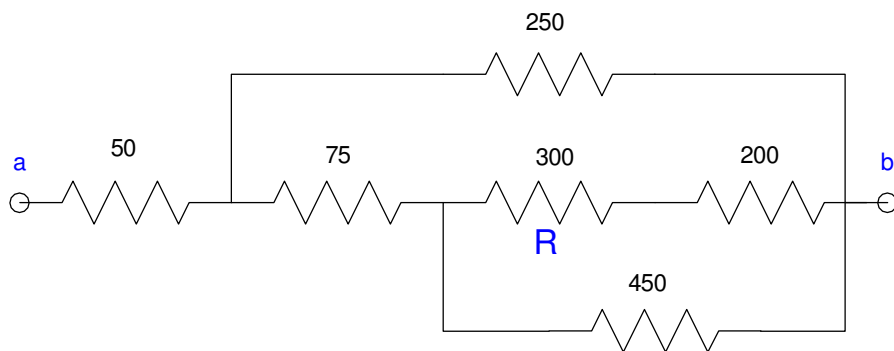
```
def Series(R1, R2):  
    Rnet = R1 + R2  
    return(Rnet)  
  
def Parallel(R1, R2):  
    Rnet = 1 / (1/R1 + 1/R2)  
    return(Rnet)
```

```
Ra = Series(300, 200)  
Rb = Parallel(Ra, 450)  
Rc = Series(Rb, 75)  
Rd = Parallel(Rc, 250)  
Rab = Series(Rd, 50)  
print('Rab = ', Rab)
```

Shell

```
>>>  
Rab = 188.7588
```

Find R_{ab} when the 300 Ohm resistor changes:



```
def Series(R1, R2):  
    Rnet = R1 + R2  
    return(Rnet)  
  
def Parallel(R1, R2):  
    Rnet = 1 / (1/R1 + 1/R2)  
    return(Rnet)  
  
def Circuit(R):  
    Ra = Series(R, 200)  
    Rb = Parallel(Ra, 450)  
    Rc = Series(Rb, 75)  
    Rd = Parallel(Rc, 250)  
    Rab = Series(Rd, 50)  
    return(Rab)  
  
for R in range(100, 400, 100):  
    Rab = Circuit(R)  
    print('R = ', R, ' Rab = ', Rab)
```

Shell

```
>>>  
R = 100    Rab = 176.2376  
R = 200    Rab = 183.5615  
R = 300    Rab = 188.7588
```

Fun with Subroutines: Convolution and Rolling Dice:

In the previous lecture, we looked at convolution and how it applies to rolling dice. Rather than having to write a convolution routine each time, let's create a subroutine which convolves two vectors.



Starting out, let's write a routine similar to Matlab's *linspace(a,dx,b)* which

- Creates a vector,
- Starting at a,
- Ending at b,
- With step size dx



```
def linspace(x0, dx, x1):  
    x = x0  
    A = []  
    while(x <= x1):  
        A.append(x)  
        x += dx  
  
def display(A):  
    n = len(A)  
    for k in range(0,n):  
        print(k, A[k])  
  
k = linspace(0,1,5)  
display(k)
```

shell

```
>>>  
0      0.000  
1      1.000  
2      2.000  
3      3.000  
4      4.000  
5      5.000
```

Now that this works, write a routine which

- Generates a uniform distribution
- Over the interval $[a, b]$:



```
def uniform(a,b):  
    A = []  
    N = b-a+1  
    for i in range(0,a):  
        A.append(0)  
    for i in range(a,b+1):  
        A.append(1/N)  
    return A  
  
print('4-sided die')  
d4 = uniform(1,4)  
display(d4)
```

shell

```
>>>  
4-sided die  
0      0.000  
1      0.250  
2      0.250  
3      0.250  
4      0.250
```

Now that this works, add a convolution routine

$$p(d_4 + d_6 = 7) = 0.167$$



```
def conv(A, B):
    nA = len(A)
    nB = len(B)
    nC = nA + nB - 1
    for n in range(0, nC):
        C.append(0)
        for k in range(0, nA):
            if ((n-k) >= 0) & ((n-k) < nB) & (k < nA):
                C[n] += A[k]*B[n-k]
    return C

d4 = uniform(1, 4)
d6 = uniform(1, 6)
d4d6 = conv(d4, d6)
print('d4 + d6')
display(d4d6)
```

shell

```
0      0.000
1      0.000
2      0.042
3      0.083
4      0.125
5      0.167
6      0.167
7      0.167
8      0.125
9      0.083
10     0.042
```

Ice Storm: $2d8 + 4d6$

With these routines, determine

- The pdf for the D&D spell *Ice Storm*
 - $2d8 + 4d6$
- The probability of doing 24 damage

There is an 8.06% chance of doing 24 damage

```
□  ↗ Open  □ Save  ▶ Run  ⬛ Stop  🇺🇸
def linspace(x0, dx, x1):
    :
def display(A):
    :
def uniform(a,b):
    :
def conv(A, B):
    :

d6 = uniform(1,6)
d8 = uniform(1,8)
d6x2 = conv(d6,d6)
d6x4 = conv(d6x2, d6x2)
d8x2 = conv(d8,d8)
IceStorm = conv(d6x4,d8x2)
print('p(24) = ', IceStorm[24])
```

shell

```
>>>
p(24) = 0.0806
```

Note: With this routine, you can also multiply polynomials


$$a(x) = 2 + 3x + x^2$$

$$b(x) = 7 + 6x + 5x^2 + 4x^3$$

$$y = a(x) \cdot b(x)$$

$$Y = [2, 3, 1] ** [7, 6, 5, 4]$$

$$y(x) = 21 + 32x + 34x^2 + 28x^3 + 13x^4 + 4x^5$$



```
def linspace(x0, dx, x1):  
    :  
def display(A):  
    :  
def uniform(a,b):  
    :  
def conv(A, B):  
    :  
  
A = [2, 3, 1]  
B = [7, 6, 5, 4]  
C = conv(A,B)  
display(C)
```

shell

```
>>>  
0      21.000  
1      32.000  
2      34.000  
3      28.000  
4      13.000  
5       4.000
```

Summary

Subroutines can be written in Python

- Similar to Matlab and C

You can pass parameters

- Similar to Matlab and C

You can return zero, one, or more parameters

- Similar to Matlab and C
- With a slightly different (and simpler) syntax

Subroutines are useful

- They allow you to create functions
 - Which can be used over and over again
 - Saving time in writing and debugging
-