4. Subroutines

Introduction:

Subroutines are programs you can call from other programs. These go by various names (funcitons in matlab, subroutines in C, definitions in Python, etc) but they all serve the same purpose:

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

Usually, subroutines are passed parameters. After some computations, they can also return parameters.

This lecture looks at how subroutines are defined in MicroPython as well as how to return parameters to the main routine.

Subroutines

In MicroPython, subroutines are defined by the keyword *def*, sort for *define*. Ths 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 defed as *SayHello*
- It then runs the main routine (instruction following all of the definitions)

```
def SayHello():
    print('hello')

# Start of main routine
SayHello()
```

shell

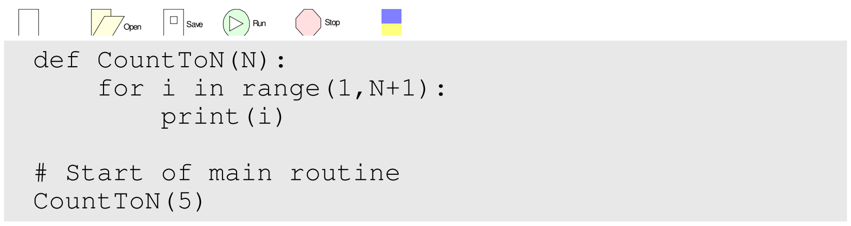
```
>>>
hello
```

In this example, note that

- The subroutine is called *SayHello*
- Nothing is passes to this routine as indicated by the ()
- The definition is terminated with a colon (:)
- The code within the subroutine must be indented as per the Python standard

Also note that once you run the routine, the function *SayHello()* is available to call from the shell window.

You can pass parameters to subroutines. For example, if you want to display numbers from 0..N, you could write a routine like the following:

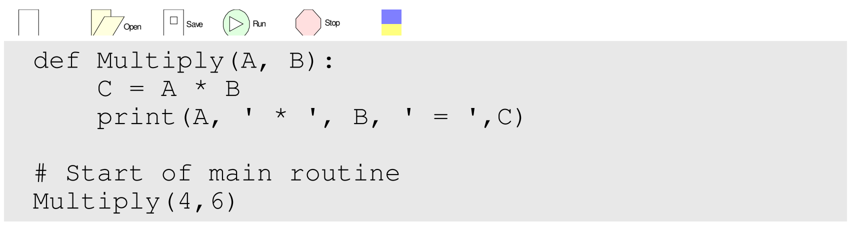


```
def CountToN(N):
    for i in range(1,N+1):
        print(i)

# Start of main routine
CountToN(5)
```

Thonny Program Window

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

You can also pass multiple parameters by simply including them in the definition



```
def Multiply(A, B):
    C = A * B
    print(A, ' * ', B, ' = ',C)

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

Thonny Program Window

```
>>>
4 * 6 = 24

>>> Multiply(8,7)
8 * 7 = 56
```

Returning Parameters

Subroutines in Python can only return zero or one variable. That variable could be an array, a matrix, or a class object however - so that's not very limiting.

Example where one number is returned:

```
# Example of Returning One Number
def Multiply(A, B):
    C = A * B
    return(C0)

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

Thonny Program Window

```
>>>
24

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

Example where four numbers are returned in a matrix:

```
# 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)
```

Thonny Program Window

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

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

Example where four numbers are returned separately

- If you receive four variables, the results are unpacked into each variable separately
- If you receive one variable, the results are unpacked into a vector containing four numbers

Open Save Run Stop

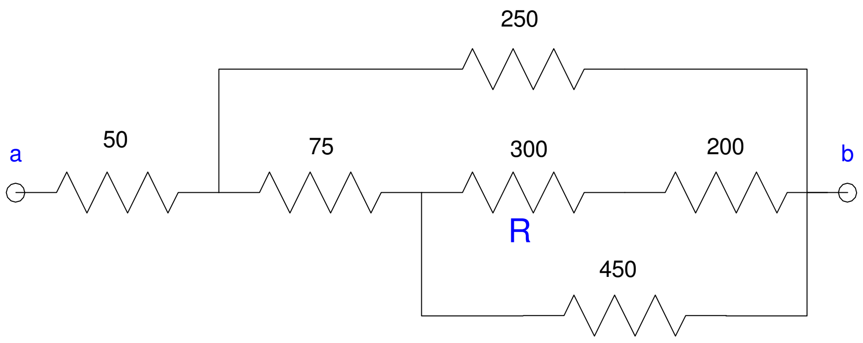
```
# Example of Returning Four Numbers
def Operate(A, B):
    C0 = A + B
    C1 = A - B
    C2 = A * B
    C3 = A / B
    return(C0, C1, C2, C3)

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

Thonny Program Window

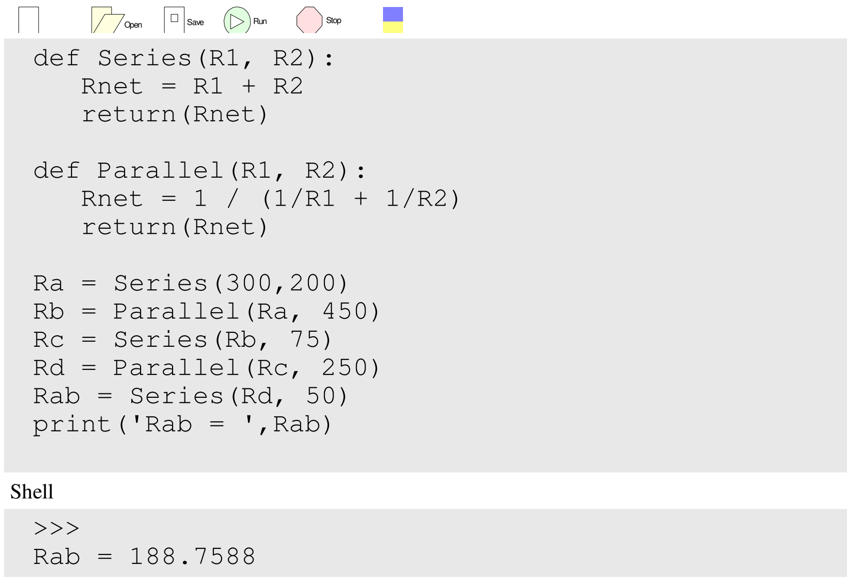
```
>>>
10, -2, 24, 0.666667

>>> 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 Rab if R is changed from 300 Ohms



Starting out, define routines for series and parallel resistors:



```
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)
```

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

Same as before, Rab = 188.7588 Ohms

You could also create a third program to find Rab when the 300 Ohm resistor changes:

Open Save Run Stop

```
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

Open Save Run Stop

```
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('{: 4.0f}'.format(k),'{: 10.3f}'.format(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 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('{: 4.0f}'.format(k),'{: 10.3f}'.format(A[k]))

def uniform(a,b):
    A = []
    N = b-a+1
    for i in range(0,a):
        A.append(0)
    for i in range(a,b+0.01):
        A.append(1/N)
    return(A)

print('4-sided die')
d4 = uniform(1,4)
display(d4)
print('6-sided die')
d6 = uniform(1,6)
display(d6)
```

shell

```
>>>
4-sided die
   0      0.000
   1      0.250
   2      0.250
   3      0.250
   4      0.250
6-sided die
   0      0.000
   1      0.167
   2      0.167
   3      0.167
   4      0.167
   5      0.167
   6      0.167
```

Now that this works, add a confolution routine

```
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('{: 4.0f}'.format(k),'{: 10.3f}'.format(A[k]))

def uniform(a,b):
    A = []
    N = b-a+1
    for i in range(0,a):
        A.append(0)
    for i in range(a,b+0.01):
        A.append(1/N)
    return(A)

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

```
>>>
d4 + d6
   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
```

The probability d4+d6 = 7 is 0.167

Note: With this routine, you can also multiply polynomials

Find

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

This is a convolution problem

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

Open Save Run Stop

```
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('{: 4.0f}'.format(k),'{: 10.3f}'.format(A[k]))

def uniform(a,b):
    A = []
    N = b-a+1
    for i in range(0,a):
        A.append(0)
    for i in range(a,b+0.01):
        A.append(1/N)
    return(A)

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)

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
```

The answer is

$$21 + 32x + 34x^2 + 28x^3 + 13x^4 + 4x^5$$