NUMERICAL SOLUTION AND SIMULATION IN ENGINEERING WITH PYTHON

By:

Ayoob Salari

Chapter 3

Table of Contents

SECTION 1

Basics of Programming in Python

This chapter describes the basics of programming in Python. Topics discusses include functions and control flow.
The statements following a programming construct that are indented by the same number of spaces are grouped together as a single piece of code. Python statements are grouped using indentation.

1.1 Functions

A function is a block of code that only executes when called. Input data, known as arguments, can be supplied into a function, and the function will return data as a result. The purpose is to collect together a set of actions that are performed often or regularly, and then create a function for them, so that we can call that function with different inputs and reuse the code it contains. Function construction in Python is

Definition 1: Function

```
def function_name(input_parameters):
    Body of statements
    return result
```

Here is a simple example of a function file for sorting in descending order:

```
def decsort(x):
    import numpy as np
    y = np.sort(-np.array(x))
    y = -y
```

```
    return y
```

Here the python built-in function `sort` is used. We assumed `sort` only sorts in ascending order, and the negative of the input array is used for descending. It should be noted that the goal here is to learn how to define function, otherwise, we can use a predefined Python function `sorted(x, reverse=True)` for sorting in descending order.
A function can have two or more outputs (as here), separate by commas.

Example 1.1 Function

Define a function that accepts two array inputs and returns the array consisting of the element by element sum of their cubes.

Solution:

```
def sumcubes(x,y):
    import numpy as np
    res = np.power(x,3)+ np.power(y,3)
    return res
```

1.1.1 Inner Functions

An `inner function`, sometimes called a `nested function`, is a function that is declared inside of another function. This means that nested functions can use the variables defined in their parent function. The term "Encapsulation" can also be used to describe this procedure as well.

```
def outerFunction(msg):
    # This is the outer enclosing function
    def innerFunction():
        # This is the nested function
        print(msg)
    innerFunction()

 # We execute the function
outerFunction("Hello")
```

Closures `Closures` are function objects that remember values in enclosing scopes even if they are not in memory.

```
def outerFunction(msg):

    def innerFunction():
        print(msg)

    return innerFunction
```

```
    # Note we are returning function WITHOUT parenthesis

 # We execute the function
variable = outerFunction("Hello")
variable()
```

The value of this variable or function inside the enclosing scope will be preserved even if the variable or function is later declared out of scope or deleted. In the following, you can see that the returning function have a value in spite of the original function being removed.

```
del outerFunction
variable()
outerFunction("Hello")
```

1.2 Control Flow

There are a number of methods for controlling program flow in Python: `for` loops, `while` loops, `if-else` constructions and `try-except` blocks. Python, unlike any other programming language, lacks a switch or case statement. Those who have dealt with languages such as C, Java, or MATLAB will have little trouble understanding the control flow structures.

1.2.1 For Loops

The syntax of the for loop is

Definition 2: `for` Loop

```
for iterator_var in sequence:
    statements
```

To iterate through a range, you may utilise it with iterators. As an example, suppose that we need the values of the sine function at eleven evenly spaced points, for $n = 0, 1, \ldots, 10$. These numbers can be generate using the following `for` loop:

```
import numpy as np
x=[]
for n in range(0,11,1):
    x1=np.sin(n*np.pi/10)
    x.append(x1)
print('Sin-Vector:',x)
```

Exercise 1.1

Calculate the sin function at eleven evenly spaced points, for $n = 0, 1, \ldots, 10$ without using `for` loop.

Solution:

```
import numpy as np
n=np.arange(0,11,1)
print('Calculate Sin Without Loop: ', np.sin(n*np.pi/10))
```

Nested Loops allow for the usage of nested iterative structures.

Definition 3: Nested Loops

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements
        statements
```

The examples that follow will help to clarify the idea.

Example 1.2 Nested Loops

Create a 10 by 10 matrix A where

$$A(m,n) = \sin(m)\cos(n).$$

Using nested for loops.

Solution:

```
import numpy as np
A = np.zeros((10,10))
for m in range(1,11):
    for n in range(1,11):
        A[m-1,n-1] =np.sin(m)*np.cos(n)
```

If loop methods must be used to generate a matrix, it is good practice to create the entire matrix in one step, called **preallocation** as in the second command of the first method. Otherwise the matrix is produced incrementally and inefficiently.

Exercise 1.2

Consider the above nested loop example and calculate the matrix A without using loops.

Solution:

```
import numpy as np
k = np.arange(1,11,1)
A=np.transpose([np.sin(k)])*np.cos(k)
```

The technique of eliminating loops is known as **vectorization**. **It should be used whenever possible**.

1.2.2 While Loops

To constantly run code until a certain condition is met, programmers often turn to Python's while loop. The line of code immediately after the loop is performed when the condition no longer holds. The syntax is

Definition 4: `while` Loop

```
while expression:
    statements
```

Here is a simple problem that requires use of a `while` loop.

Example 1.3 While Loop

Suppose that the number π is divided by 2 and the resulting quotient divided again by 2, and so on, until the quotient is less than or equal to 0.01. What is the largest quotient that is less than 0.01 on the continue division of π by 2?

Solution:

```
import numpy as np
q=np.pi
while q>0.01:
    q=q/2
print('Largest quotient that is less than 0.01: ', q )
```

It is possible to use `else` in the `while` loop. As previously stated, a while loop performs the block till a condition is met. In the `while-else` loops, when your while condition fails, the else clause is performed. If you exit the loop or raise an exception, the code will not be performed.

Definition 5: `while-else` Loop

```
while condition:
    statements 1
else:
    statements 2
```

Following is an example that sheds lights on this type of loops:

```
i=3
while i < 6:
  print(i)
  i += 1  # Add one to the value of i
else:
  print("i>6")
```

Exercise 1.3

Rewrite the following code using a `while` loop to avoid using the break command:

```
import numpy as np
for k in range(1,11):
    x = 50 - k**2
    if x < 0:
        break
    y=np.sqrt(x)
```

Solution:

```
import numpy as np
k=0
x=50
while x>=0    :
    y = np.sqrt(x)
    k += 1
    x = 50-k**2
```

1.2.3 If-Else Construct

A general construct of `if-else` is:

Definition 6: `if-else` Loop

```
if  (condition 1):
    statements 1
elif (condition 2):
    statements 2
else:
    statements 3
```

As an example, we compute the coefficients of Chebyshev polynomials:

Example 1.4 Chebyshev polynomials

Consider the problem of calculating Chebyshev polynomials of the first kind $T_n(x)$, $n = 0, 1, 2, \ldots$. They can be defined recursively by:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

with

$$T_0(x) = 1 \quad \text{and} \quad T_1(x) = x.$$

Write a function that calculates the coefficients of Chebyshev polynomials

Solution:

```
# CHEB calculates coefficients of the Chebyshev polynomial of
the first kind of order n.
# c is the vector of coefficients in descending order
def Cheb(n):
    import numpy as np
    t0=np.array([1])
    t1=np.array([1,0])
    if n==0:
        c=t0
    elif n==1:
        c=t1
    else:
        for k in range(2,n+1):
            c=2*np.append(t1,0)-np.append(np.zeros((1,2)),t0)
            # Above equation matches the Chebyshev equation.
            t0=t1 #Reassignment to match T_(n-1)
            t1=c  # Reassignment to match T_(n)
    return c
```

The result of `Cheb(5)` is:

```
c = [ 16    0   -20    0    5    0 ]
```

that is $T_5 = 16x^5 - 20x^3 + 5x$.
You might be tempted to write `Cheb` as a *recursive* function (one that calls itself).

Example 1.5 Chebyshev polynomials

Consider the previous example and write a **recursive** function that calculates the coefficients of Chebyshev polynomials

Solution:

```
# Recursive version
def Cheb(n):
    import numpy as np
    if n==0:
        c=np.array([1])
    elif n==1:
        c=np.array([1,0])
    else:
        c=2*np.append(Cheb(n-1),0)-np.append(np.zeros((1,2)),
Cheb(n-2))
    return c
```

but this version is much slower because of the overhead in function calls.

Exercise 1.4

Python lacks switch-case statements, which are common in other programming languages. To replace switch cases, the if-else construct could be utilized. Write a script to input one angle, whose value can be 45°, −45°, 135° or −135°, and display its quadrant (1, 2, 3 or 4).
Hint: You need to write a switch-case function using a `if-else` block.

Solution:

```
angle = int(input("Enter the angle: "))
def switch(angle):
    if angle == 45:
        return "Quadrant 1"
    elif angle == 135:
        return "Quadrant 2"
    elif angle == -135:
        return "Quadrant 3"
    elif angle == -45:
        return "Quadrant 4"
    else :
        return "Check your input."
print(switch(angle))
```

1.2.4 Try-Except

Python will generally halt and display an error message if an error, or exception, occurs. The `try-except` statement can catch and manage these unexpected conditions. The `else` clause, which must come after all the except clauses in a `try-except` block, is another useful tool. If the try clause does not throw an error, the function continues to the else block. `Finally` keyword is used after any `try-except` blocks have completed. No matter what causes the try block to exit, the final block will always run thereafter.

A general construct of `try-except` is:

Definition 7: `try-except`

```
try:
    statements 1
except:
    # Executed if error in the try block
else:
    # Execute if no exception
finally:
    # Codes that always executed
```

As an example, we can consider division by zero.

```
try:
    for i in range(0,4):
        R=1/i
        print(R)
except ZeroDivisionError:
    print("Not-possible-to-divide-by-zero")
finally:
    print("This-is-always-executed")
```

```
Not-possible-to-divide-by-zero
This-is-always-executed
```

1.2.5 Loop Control Statements

As a result of the statements that control the loop, the usual order of operations is altered.

Continue Statement: It returns the control to the beginning of the loop.

```
 # Prints all numbers except 6 and 7
for i in range(1,11):
        if i == 6 or i == 7:
                continue
        print ('Current-number:', i)
```

Break Statement: It brings control out of the loop.

```
 # break the loop as soon it sees 6 or 7
for i in range(1,11):
        if i == 6 or i == 7:
                break
        print ('Current-number:', i)
```

Pass Statement: You may create a loop that does nothing by using the pass statement.

```
 # An empty loop
for i in range(1,11):
        pass
print ('Last-number:', i)
```

Example 1.6 Removing Duplicates

Write a Python function that takes a one-dimensional array of numbers (either a row or column vector), and removes all of the neighboring duplicate numbers. For example, the array `[1, 2, 2, 2, 3, 0, 1, 0, 0, 4]` becomes `[1, 2, 3, 0, 1, 0, 4 ]`. The function should return the answer as a one-dimensional array of numbers in the same format as the input. Your program should use a loop command.

Solution:

Example 1.7 Interleaving Arrays

Write a Python function that takes two row vectors *a* and *b*, not necessarily of the same length, and returns the row vector obtained by interleaving the two inputs. For example, if the first vector is `[1, 3, 5, 7, 0, 0]` and the second is `[-2, -5, 6]`, the output vector is `[1, -2, 3, -5, 5, 6, 7, 0, 0]`. Your function should work for an empty vector. Program loops are allowed.

Solution: