Scilab Tutorial

Variables

- **Integer** ตัวแปรประเภทจำนวนเต็ม เช่น 3, 14, 19, 23
- **Float** ตัวแปรประเภทจำนวนจริง เช่น 2.19, 3.14, 0.00001
- **Boolean** ตัวแปรที่เก็บค่าความจริงทางตรรกศาสตร์(true/false)
- **String** เป็นตัวแปรประเภท”สายอักขระ”นั่นคือข้อความ เช่น “Hello world”, “Cat is good” หรือแม้กระทั่ง “ “

Note ตัวแปรประเภทจำนวนจริงจะอาศัยการ”ประมาณค่า”ในการเก็บข้อมูลจึงควรใช้อย่างระมัดระวัง

Variables

การประกาศตัวแปรสามารถทำได้โดย ใช้รูปแบบ <ชื่อตัวแปร> = <ค่าที่ต้องการจะมอบ>

เช่น x = 5, str = “hello”, gpa = “3.97”, x = gpa (มีการประกาศตัวแปร gpa มาก่อน)

สังเกตว่าเครื่องหมาย “=“ ในการเขียนโปรแกรมไม่ใช่ความสัมพันธ์ แต่เป็นการมอบค่าฝั่งด้านขวา

ให้กับตัวแปรที่อยู่ด้านซ้ายเพราะฉะนั้นตัวอย่างต่อไปนี้ไม่สามารถคอมไพล์ได้

เช่น 5 = 20, 19 = p, หรือ p = s (เนื่องจากการนำค่าของตัวแปร s มาใช้จำเป็นจะต้องประกาศก่อนว่าตัวแปร s มีค่าเป็นเท่าใด)

Note ทุกตัวแปรและคำสั่งใน scilab เป็นประเภท case sensitive นั่นคือทุกคำที่ใช้ตัวสะกดต่างกันจะเป็นคนละคำ เช่นตัวแปร A จะเป็นคนละตัวแปรกับตัวแปร a หรือ คำสั่ง Disp จะใช้ไม่ได้

Input/Output

การรับค่าจากแป้นพิมพ์ ใช้คำสั่ง input โดยมีรูปแบบดังนี้

<ตัวแปรที่จะรับค่าที่ป้อน> = input(<ข้อความที่จะแสดง>)

ตัวอย่างเช่น

1. A = input(“Input A :”)
2. str = “input B:”

   B = input(str)

Input/Output

ในส่วนของการแสดงผล(output) จะสามารถใช้ได้สองคำสั่งคือ disp และ printf

คำสั่ง disp มีรูปแบบการใช้คือ disp(<ค่าที่ต้องการจะแสดง>)

ตัวอย่างเช่น disp(3), disp(3+5), disp(“this is string”)

คำสั่ง disp สามารถใช้ในการแสดงค่าของตัวแปรได้อีกด้วยเช่น

Birthday = 19.02

disp(Birthday)

Input/Output

คำสั่ง printf จะมีความซับซ้อนในการใช้งานมากกว่า disp แต่สามารถปรับแต่งได้มากกว่า

รูปแบบการใช้คือ printf(<สิ่งที่ต้องการแสดง>) เช่น printf(“hello world”)

แต่ในการแสดงค่าของตัวแปรจำเป็นจะต้องใช้รูปแบบพิเศษคือ

printf(“<format1> <format2>…<formatn>”, <var1>, <var2>, … <varn>)

โดยที่ format1, format2, …, <formatn> เป็นรูปแบบการแสดงผลตัวแปรของตัวแปรหรือค่า

var1, var2, …, varn **ตามลำดับ**

Input/Output

ใน scilab รูปแบบการแสดงผลของตัวแปร(format) สามารถแบ่งได้เป็นสามกรณีใหญ่ๆ

- %d **decimal** สำหรับการแสดงผลในรูปของจำนวนเต็ม(integer)
- %f **float** สำหรับการแสดงผลในรูปของจำนวนจริง(real values)
- %s **string** สำหรับการแสดงผลในรูปของข้อความหรือสายอักขระ

ตัวอย่างเช่น

- printf(“%d %f %s”, num, ratio, massage)
- Printf(“Price of %d eggs is %f baht”, quantity, price*quantity)

Input/Output

- ในคำสั่ง printf จะมีสิ่งที่เรียกว่า escaped character เพื่อแสดงผลเฉพาะสิ่งเช่น \n เป็นการเว้นบรรทัด \t เป็นการเว้นช่องว่างประมาณการกด tab

- เราสามารถกำหนดจำนวนจุดทศนิยมที่จะแสดงได้ (default คือ 6 ตำแหน่ง) โดยใช้รูปแบบเป็น %.nf โดย n คือจำนวนจุดทศนิยมที่ต้องการแสดงเช่น %.1f %.4f เป็นต้น

- หากเราต้องการแสดงค่าของตัวแปรจำนวนจริงให้อยู่ในรูปจำนวนเต็มสามารถใช้ %d แทนได้

Operator

- Arithmetic operator เป็นการดำเนินการทางคณิตศาสตร์ทั่วไปเช่น

+, -, * แทนการคูณ, ^ หรือ ** แทนการยกกำลัง

/ แทนการหารทางขวา เช่น x/y = xy^-1

\ แทนการหารทางซ้าย เช่น x\y = x^-1y

- Logical operator เป็นตัวดำเนินการของค่าความจริงเช่น

& (and), | (or), ~ (not), a == b (is equal), a ~= b (is not equal), a < b (is lower than),

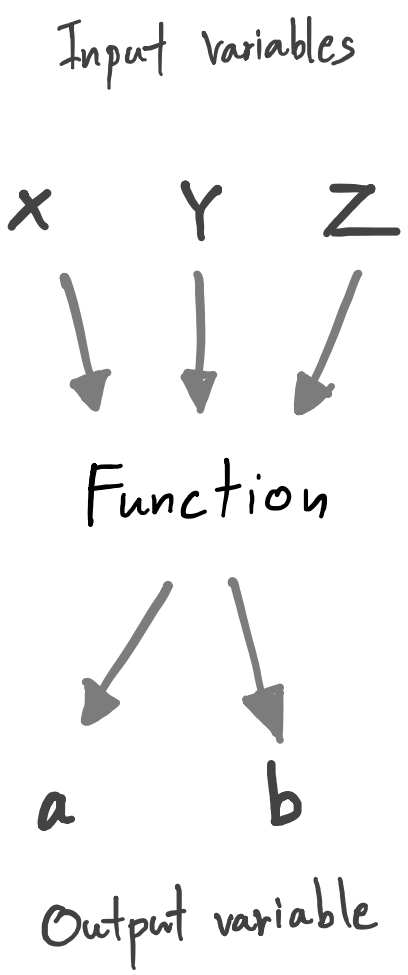
a > b (is greater than), a <= b (is lower or equal), a >= b (is greater or equal)

Function

การประกาศฟังก์ชันจะใช้รูปแบบดังนี้

```
function [out1, out2, ...] = <ชื่อฟังก์ชัน> (in1, in2, ...)
    //description
endfunction
```

- out1, out2, ... คือตัวแปรที่จะส่งออกจากฟังก์ชัน
- in1, in2, ... คือตัวแปรที่นำเข้ามาในฟังก์ชัน



Function

ตัวอย่าง ฟังก์ชันบวกเลข

```
function [sum] = addition (first_variable, second_variable)
    sum = first_variable + second_variable
endfunction
```

ตัวอย่างการเรียกใช้ฟังก์ชัน

```
sum = addition(19, 2)
disp(sum)
```

Function

ตัวอย่าง สร้างฟังก์ชันของการดำเนินการทวิภาค (binary operator)

```
function [result] = operate(first_var, second_var)
    result = (2*first_var) + 0.19 / (3.14*second_var)
endfunction
```

ตัวอย่างการเรียกใช้

```
Res =  operate(17, 49)
disp(Res)
```

Function

ตัวอย่าง สร้างฟังก์ชันของการสลับค่าของตัวแปร

```
function [out1, out2] = swap(in1, in2)
    temp = out1
    out1 = out2
    out2 = temp
endfunction
```

ตัวอย่างการเรียกใช้

```
[var1, var2] =  swap(var1, var2)
```

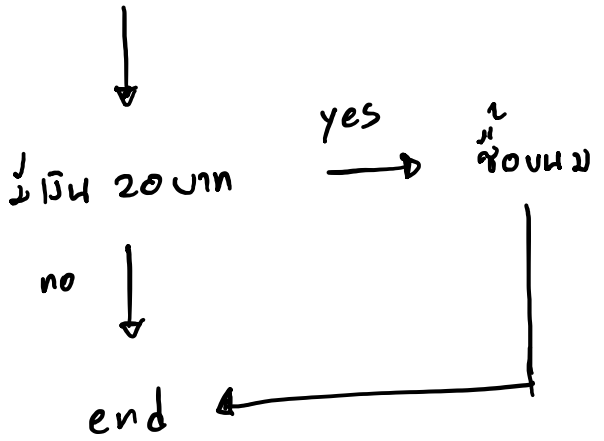
Branches (if/else/elseif statement)

คำสั่งประเภทนี้เป็นคำสั่งประเภทที่ทำการควบคุมการทำงานของโปรแกรม(control flow) โดยขึ้นกับค่าความจริงของเงื่อนไขหรือประพจน์ต่างๆ สมมติว่า **ถ้า** คุณมีเงิน 20 บาท **แล้ว** คุณจะซื้อขนม

แสดงว่าประพจน์คือ คุณมีเงิน 20 บาท ซึ่งค่าความจริงเป็นจริง คุณก็จะซื้อขนม

If มีเงิน 20 บาท

then ซื้อขนม



Branches (if/else/elseif statement

ในโปรแกรม scilab การสร้างการทำงานแบบนี้จะใช้คำสั่ง if ดังรูปแบบนี้

if <เงื่อนไข หรือ ประพจน์ที่มีค่าความจริง> then

    สิ่งที่จะทำถ้าค่าความจริงเป็นจริง

endif

ตัวอย่างเช่น

```
if money >= 20 then
      buy_snacks()
end
```

Branches (if/else/elseif statement

ในกรณีที่เราต้องการให้ทำอะไรบางอย่างในกรณีที่ค่าความจริงเป็นเท็จจะใช้คำสั่ง else

if <เงื่อนไข หรือ ประพจน์ที่มีค่าความจริง> then

    สิ่งที่จะทำถ้าค่าความจริงเป็นจริง

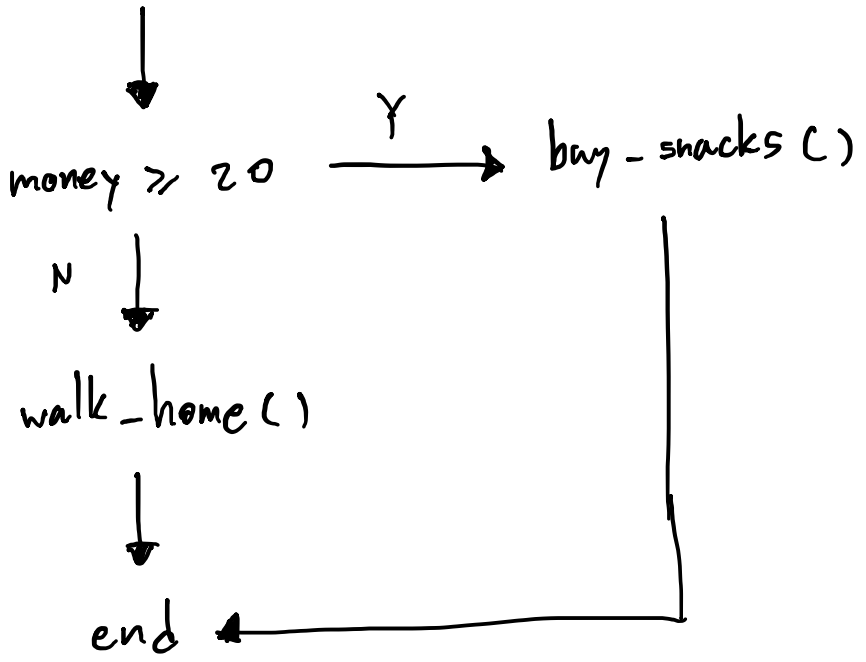
else

    สิ่งที่จะทำถ้าค่าความจริงเป็นเท็จ

end

ตัวอย่างเช่น

```
if money >= 20 then
     buy_snacks()
else
     walk_home()
end
```



Branches (if/else/elseif statement)

ชีวิตจะซับซ้อนขึ้นไปอีกขั้นด้วยคำสั่ง elseif ที่ใช้สำหรับทำงานในกรณีที่เป็นเท็จของเงื่อนไขก่อนหน้า และทำการ**ตั้งเงื่อนไขอีกครั้ง**

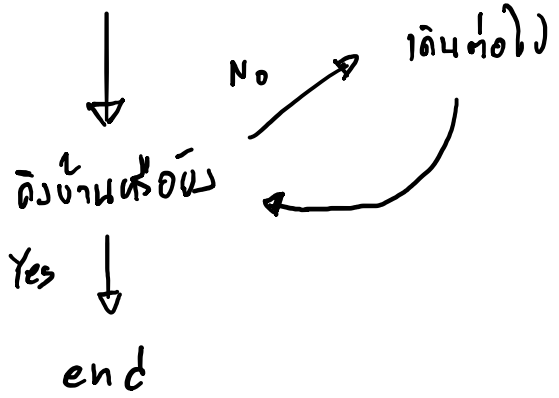
```
if <เงื่อนไข1> then
    //เงื่อนไข1 เป็นจริง
elseif <เงื่อนไข2> then
    //เงื่อนไข1 เป็นเท็จ แต่เงื่อนไข2 เป็นจริง
else then
    //เงื่อนไข 1 และเงื่อนไข2 เป็นเท็จ
end
```

Do it yourself!

- ลองเขียนโปรแกรมตัดเกรดด้วย scilab โดยอิงจาก pseudocode/flowchart ในใบงานที่1
- เขียนโปรแกรมทดสอบประเภทของตัวเลขว่าเป็น จำนวนเต็มบวก จำนวนเต็มลบ หรือ ศูนย์
- เขียนโปรแกรมทายตัวเลข
- และอื่นๆตามความสนใจ

Loop

สมมติว่าคุณกำลังเดินกลับบ้าน **คุณจะรู้ได้อย่างไรว่าคุณต้องเดินอย่างไร** อาจจะฟังดูตลกแต่จริงๆ แล้วเราคิดว่า **ตราบใดที่** ยังไม่ถึงบ้าน **แล้ว** ต้องเดินต่อไป จะพบว่าการกำหนดเงื่อนไขกลับมามีบทบาทอีกครั้งในคำสั่งประเภท loop



ตราบใดที่ ยังไม่ถึงบ้าน

เดินต่อไป

Loop

แน่นอนว่าใน scilab มีคำสั่งที่สนับสนุนการทำงานแบบนี้ในที่นี้คือคำสั่ง while ที่ใช้งานดังนี้

```
while เงื่อนไขการทำซ้ำ
    สิ่งที่จะทำเมื่อเงื่อนไขเป็นจริง
end
```

ตัวอย่างเช่น

```
while not_at_home
    Keep_walking()
end
```

Loop

ตัวอย่าง โปรแกรม countdown

```
Num = input(“Input number :”)
while Num > 0
    disp(Num)
    Num = Num – 1
end
```

Loop

จากคำสั่งแบบ loop เป็นการทำซ้ำภายใต้เงื่อนไขหนึ่ง โดยเราไม่รู้จำนวนแน่นอน แต่ถ้าเรารู้จำนวนครั้งแน่นอนของการทำซ้ำ scilab มีคำสั่ง **for** มาสนับสนุนการทำงานรูปแบบนี้

```
for var = start : step : end
    สิ่งที่จะทำเมื่อเป็นจริง
end
```

var เป็นชื่อตัวแปรที่จะเก็บค่า start ที่เป็นจุดเริ่มต้น โดยเปลี่ยนแปลงค่าในรูปแบบ var = var + step

จนกว่า var > end (นั่นคือสมมูลกับทำซ้ำตราบใดที่ var <= end)

Loop

ตัวอย่าง โปรแกรม countdown

```
Num = input(“Input number :”)
for current = Num : -1 : 0
    disp(current)
end
```

Do it yourself!

- สร้างโปรแกรมคูณเลขระหว่างสองจำนวน โดยใช้เป็นการบวกหลายๆครั้ง
- สร้างเกมทายตัวเลข(อีกครั้ง) โดยเราจะต้องป้อนเลขไปเรื่อยๆจนกว่าจะทายถูก
- ปรับปรุงโปรแกรมทายตัวเลขโดยมีการบอกเพิ่มเติมว่า เลขที่ทายมา มีค่ามากกว่า หรือน้อยกว่าตัวเลขจริงๆ
- สร้างโปรแกรมที่คำนวณค่า n!
- สร้างโปรแกรมที่แสดงตัวประกอบทั้งหมดของ n
- สร้างโปรแกรมที่หาค่า หรม/ ครน/ ทดสอบจำนวนเฉพาะ...
- สร้างโปรแกรมที่หาค่าลำดับที่ n ของลำดับ fibonacci

Recursive function

Recursive function เป็นฟังก์ชันที่ทำงานโดยมีการเรียกใช้ตัวเองซ้ำ ยกตัวอย่างเช่น factorial

Array

ในทางคณิตศาสตร์ ลำดับคือชุดของตัวเลขที่ต่อๆกัน

3, 14, 19, 23, 44, 112, ...

ใน scilab มีประเภทของข้อมูลที่สามารถจัดเก็บลำดับแบบนี้ได้ในรูปของ

[3, 14, 19, 23, 44, 112]

เราเรียกสิ่งนี้ว่า **อาร์เรย์ 1 มิติ**

Array

หากเรามองในมุมมองที่กว้างขึ้น ในมุมมองของเมทริกซ์

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

โปรแกรม scilab ก็สนับสนุนการทำงานแบบนี้เช่นกัน โดยเราจะเรียกว่า อาร์เรย์ 2มิติ

Note สามารถเขียนได้โดยใช้วิธีอีกแบบคือ A = [1 2 3 ; 4 5 6]

Access and update value in array

ใน scilab จะมองทุกอย่างเป็นเมทริกซ์ในทางคณิตศาสตร์ นั่นคือจะระบุตำแหน่งของสมาชิก(entry) ด้วยการใช้ แถว และ หลัก (row and column)

col 1 col 2 col 3

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

row 1

row 2

Access and update value in array

ในการเข้าถึงค่าในอาร์เรย์จำเป็นจะต้องมีชื่อของอาร์เรย์และตำแหน่งที่จะค้นหา โดยคำสั่งเป็นดังนี้ <ชื่ออาร์เรย์>(<แถวที่ต้องการค้นหา>, <หลักที่ต้องการค้นหา>) ยกตัวอย่างเช่น

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

ถ้าหากเราใช้คำสั่ง A(1, 1) จะได้ค่าเป็น 1 A(1, 3) จะได้เป็น 3 และ A(2, 2) ได้ค่าเป็น 5 แต่ถ้าเราใช้ B(1, 1) จะไม่สามารถใช้ได้ เนื่องจากยังไม่ประกาศตัวแปร B หรือถ้าเราใช้คำสั่ง A(5, 5) ก็ไม่สามารถใช้ได้เพราะ matrix ที่เราสร้างขึ้นมีขนาด 2x3

Access and update value in array

ในการแก้ไขค่า เราจะปฏิบัติเหมือน A(row, column) เป็นตัวแปรตัวหนึ่ง นั่นคือเราสามารถใช้ operator หรือทำการแก้ไขค่าได้อย่างอิสระ

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

disp(A(2, 2)) // ได้ค่าเป็น 5

A(2, 2) = 19 // แก้ไขค่าที่ แถวที่ 2 หลักที่ 2 เป็น 19

disp(A(2, 2)) // ได้ค่าเป็น 19

Access and update value in array

เราสามารถระบุถึงเมทริซ์ย่อย(submatrix) ได้อีกด้วยตัวอย่างเช่น

A(1, :) // สมาชิกทุกตัวในแถวที่ 1 ของ A

A(: , 3) // สมาชิกทุกตัวในหลักที่ 3 ของ A

A(2, 3 : 5) // สมาชิกหลักที่ 3 ถึง 5 ในแถวที่ 2 ของ A

Note จะเกิดอะไรขึ้นเมื่อคุณใช้คำสั่ง A(5, 5) = 1 ในอาเรย์ตัวอย่าง A