



Module 3 - System Administration Python

Session 2 - Operators and Flow Control

Presented by Tim Medin

© SANS, Cyber Aces, Red Siege. All Rights Reserved. Redistribution Prohibited.

YOUR GATEWAY TO CYBERSECURITY SKILLS AND CAREERS

Welcome to the Module 3, System Administration. In this sub-section we'll be discussing Python. In this session we'll discuss flow control in Python.

SANS CYBER ACES ONLINE TUTORIALS

YOUR GATEWAY TO CYBERSECURITY SKILLS AND CAREERS

1. Introduction to Operating Systems

- 01. Linux
- 02. Windows

2. Networking

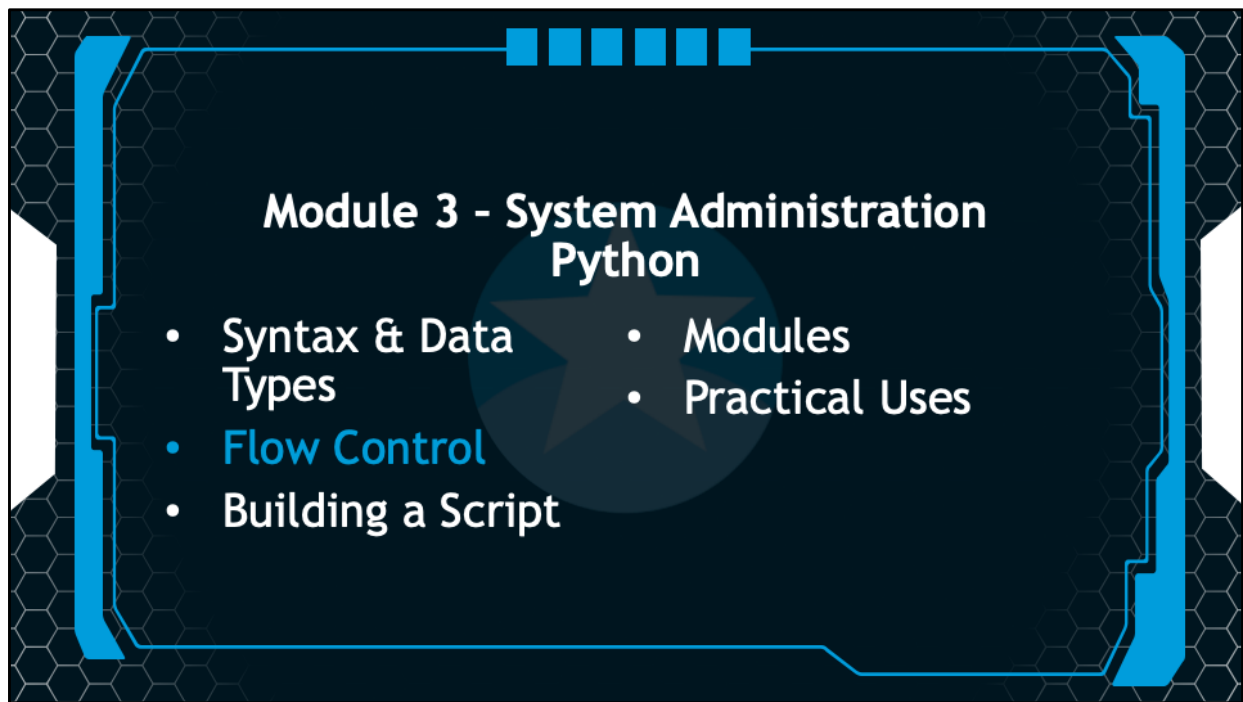
3. System Administration

- 01. Bash
- 02. PowerShell
- 03. Python

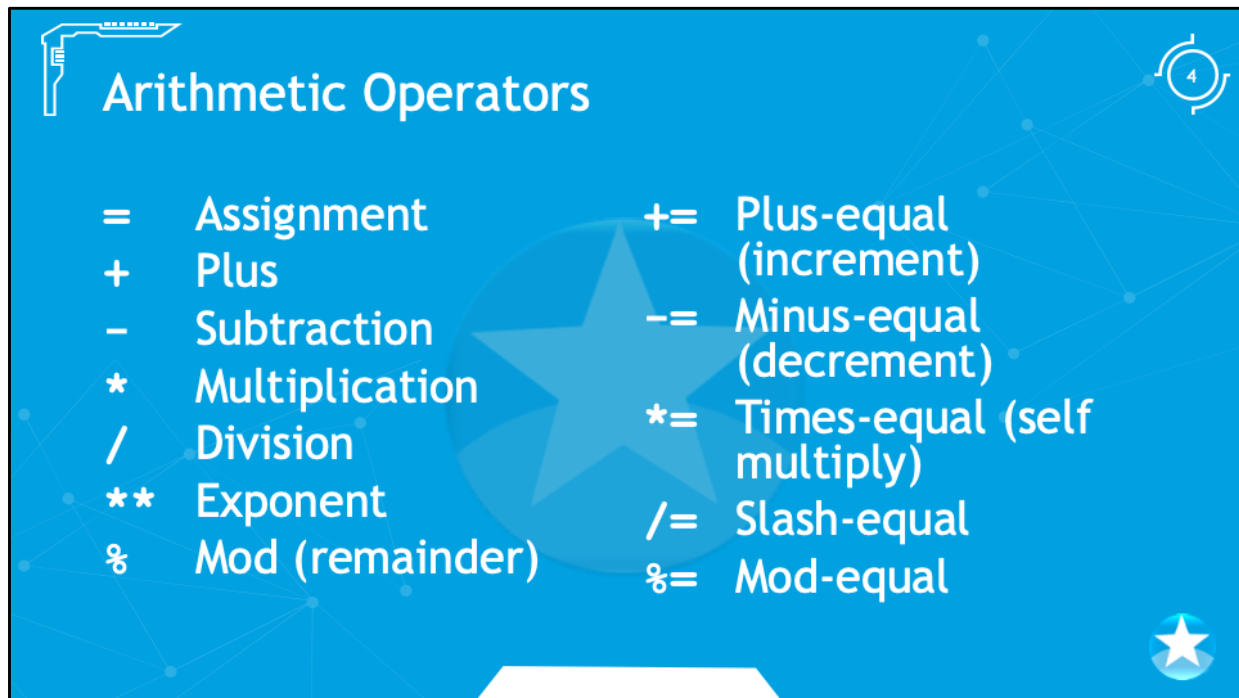
This training material was originally developed to help students, teachers, and mentors prepare for the Cyber Aces Online Competition. This module focuses on the basics of system administration and scripting. This session is part of Module 3, System Administration. This module is split into three sections, Bash, PowerShell, and Python. In this session, we will continue our examination of Python.

The three modules of Cyber Aces Online are Operating Systems, Networking, and System Administration.

For more information about the Cyber Aces program, please visit the Cyber Aces website at <https://CyberAces.org/>.



In this section, you will be introduced to flow control in Python.



Arithmetic Operators

=	Assignment	+=	Plus-equal (increment)
+	Plus	-=	Minus-equal (decrement)
-	Subtraction	*=	Times-equal (self multiply)
*	Multiplication	/=	Slash-equal
/	Division	%=	Mod-equal
**	Exponent		
%	Mod (remainder)		

The information here is the same as with Bash.

=	Assignment
+	Plus
-	Subtraction
*	Multiplication
/	Division
**	Exponent
%	Mod (remainder)
+=	Plus-equal (increment)
-=	Minus-equal (decrement)
*=	Times-equal (self multiply)
/=	Slash-equal
%=	Mod-equal



Comparison, Membership, and Logical Operators



<code>==</code>	Equals
<code>!=</code>	Not equals
<code><></code>	Not equals (similar to <code>!=</code>)
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to
<code>in</code>	True if value is in list or tuple
<code>not in</code>	Inverse of <code>in</code>
<code>and</code>	Logical AND
<code>or</code>	Logical OR
<code>not</code>	Logical NOT

The comparison operators are shown here:

<code>==</code>	Equals
<code>!=</code>	Not equals
<code><></code>	Not equals (similar to <code>!=</code>)
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to
<code>in</code>	True if value is list or tuple
<code>not in</code>	Inverse of <code>in</code>
<code>and</code>	Logical AND
<code>or</code>	Logical OR
<code>not</code>	Logical NOT



if...then...else



```
if x == 0:
    print('x is zero')
elif x >= 1 and x < 10:
    print('x is a single digit')
elif x > 10 and x < 100:
    print('x is two digits')
else:
    print('unknown condition')
```

Indentation must be consistent in code blocks

- Some use spaces (2, 3, 4, 5, 8) other use a single tab
- Tabs vs Spaces is a highly contested topic!

We can execute code based some condition using an "if" statement. We can do a simple statement like this:

```
if x == 0:
    print('x is zero')
```

We can have two options with this code:

```
if x == 0:
    print('x is zero')
else:
    print('unknown condition')
```

Or a more complex statement like this:

```
if x == 0:
    print('x is zero')
elif x >= 1 and x < 10:
    print('x is a single digit')
elif x > 10 and x < 100:
    print('x is two digits')
else:
    print('unknown condition')
```

The indention of each portion should be consistent. Some developers uses spaces (2, 3, 4, 5, 8) and some use tabs (just one) for each level of indention. The choice of tabs and spaces (and further, the number of spaces) is a hotly contested topic in the development community. What really matters is consistency in a development project. Space People and Tab People both agree that if you mix tabs and spaces you

are a terrible person.



Loops



There are two types of loops

- while - repeats code while a condition is true
- for - executes code multiple times (counter)

Special loop statements

- break - exit the loop and continue the code after the loop
- continue - skip the remainder of the loop and go to the next iteration

We use loops to perform an action a number of times. There are two types of loops in Python, the while loop and the for loop.

The while loop will execute code while a condition is true. The for loop is used to loop a predetermined number of times, such as with a counter or over a list (or tuple).

Inside the loop there are special statements to control execution. The break statement is used to leave the entire loop. The continue statement is used to skip the current iteration, perform the test condition (if applicable) and then go to the next iteration.



For loop



Often used with **range** where the "end" is exclusive

```
>>> for i in range(3):  
...     print(i)  
...  
0  
1  
2
```

Notice the prompt changes once you indent

`range(inclusive start, exclusive end, step)`

We can use the for loop with a counter. The range function will give us a series of numbers. The range statement can be used to give a simple range as shown above.

```
>>> for i in range(3):  
...     print(i)  
...  
0  
1  
2
```

We can also define the start, the stopping number, and step (increment) for each iteration. We could even use a negative number.

```
>>> for i in range(4, 10, 2):  
...     print(i)  
...  
4  
6  
8
```



For loop



Can also be used with a list

```
>>> weekdays = ['m', 't', 'w', 't', 'f']
>>> for day in weekdays:
...     print(day)
...
m
t
w
t
f
```

We can also iterate over a list, tuple or other sequence of objects.

```
>>> weekdays = ['mon', 'tues', 'wed', 'thurs', 'fri']
>>> for day in weekdays:
...     print(day)
...
mon
tues
wed
thurs
fri
```



While Loop



The while loop will check a condition before executing each iteration

```
>>> n = 1
>>> while n < 128:
...     print(n)
...     n *= 2
...
1
2
4
8
16
32
64
```



The while loop will run as long as the test condition is true. In this loop, we start with the number 1 and double it as long as it is less than 128.

```
>>> n = 1
>>> while n < 128:
...     print(n)
...     n *= 2
...
1
2
4
8
16
32
64
128
```



Input



Most programs will take some sort of input
To accept keyboard input we use the `input()` function

```
x = input('Do you like Python? y/n ')  
if x == 'y':  
    print('Excellent!')
```

Most programs will take some sort of input. To accept keyboard input we use the `input()` function. The code below will take input from the user and print "Excellent!" if the user enters "y" followed by the enter key.

```
x = input('Do you like Python? y/n ')  
if x == 'y':  
    print('Excellent!')
```



Break



Python doesn't have an unguarded loop

The while loop checks the condition on entry

To prevent multiple input and condition checks you will often see code like this:

```
while True:
    x = input('Pick a number between 1 and 100: ')
    if x.isdigit() and int(x) >= 1 and int(x) <= 100:
        break
```

Without break we would see something like this:

```
x = input('Pick a number between 1 and 100: ')
if not (x.isdigit() and int(x) >= 1 and int(x) <= 100):
    while not(x.isdigit() and int(x)>=1 and int(x) <=100):
        x = input('Pick a number between 1 and 100: ')
```

Python doesn't have an unguarded loop since the while loop checks the condition on entry. To prevent multiple input and condition checks you will often see code like this:

```
while True:
    x = input('Pick a number between 1 and 100: ')
    if x.isdigit() and int(x) >= 1 and int(x) <= 100:
        break
```

If we did this without the infinite loop we would have to do something like this:

```
x = input('Pick a number between 1 and 100: ')
if not (x.isdigit() and int(x) >= 1 and int(x) <= 100):
    while not(x.isdigit() and int(x)>=1 and int(x) <=100):
        x = input('Pick a number between 1 and 100: ')
```

As you can see, there are multiple checks and multiple input requests which is more prone to mistakes.



Continue



Skips the remainder of the code in the iteration and jumps to the top of the loop

```
>>> for letter in 'python':  
...     if letter == 'h':  
...         continue  
...     print(letter)  
...  
p  
y  
t  
o  
n
```

The "continue" will skip the rest of the code in the current iteration and jump back to the top of the loop. If it were a while loop the condition would be checked before execution. The code below demonstrates the usage of "continue".

```
>>> for letter in 'python':  
...     if letter == 'h':  
...         continue  
...     print(letter)  
...  
p  
y  
t  
o  
n
```



Review



Using what you've learned here so far, write Python that does the following:

- Asks the user if they like Python
- Only accepts "y" or "n" as input
- If "y", then print "Excellent"
- If "n", then print "Sorry"
- If any other entry, print "Invalid input"

Using what you've learned here so far, write Python that does the following:

- Asks the user if they like Python
- Only accepts "y" or "n" as input
- If "y", then print "Excellent"
- If "n", then print "Sorry"



Possible Answer



```
while True:
    a = input('Do you like Python? y/n ')
    if a == 'y':
        print('Excellent')
        break
    elif a == 'n':
        print('Sorry')
        break
    else:
        print('Invalid input')
```

One possible answer is shown here:

```
while True:
    a = input('Do you like Python? y/n ')
    if a == 'y':
        print('Excellent')
        break
    elif a == 'n':
        print('Sorry')
        break
    else:
        print('Invalid input')
```

Module 3 - System Administration Python

- Syntax & Data Types
- ✓ **Flow Control**
- Building a Script
- Modules
- Practical Uses

We've completed the discussion of flow control in Python. In the next session, we'll discuss reusing code in scripts and functions.