Module 9
Programming

Student Guide

Name: ______________________________ Class #: ______________

CIN: A-531-1900
Ver. 2020-01R5

Prepared for the U.S. Government by:

Table of Contents

Introduction

Programming challenges students to use knowledge from previous modules to develop useful tools. Students are introduced to Windows PowerShell and UNIX Shell scripting languages and they use them to perform basic system enumeration and analysis. Students must rely upon information from OS and networking modules, specifically sockets, to develop a port scanner tool using Python. Finally, students use regular expressions in Python and PowerShell as well as UNIX scripting techniques to parse large volumes of data. Programming ties many concepts together and teaches students how to automate cyber offensive and defensive work roles.

Safety/Hazard Awareness Notice

All personnel involved in operation and maintenance of electronic equipment must be thoroughly familiar with safety precautions as covered in Module 1.

Module Overview

This module provides hands-on experience with UNIX/Bash, PowerShell, and Python scripting languages. Students learn to identify program structures and functions and develop critical thinking skills through hands-on exercises.

Module Testing Practices

This module consists of the following graded events requiring a 75% or better score:

- One knowledge test
- One performance test

This module also utilizes a minimum of two quizzes and daily self-study quizzes in JCAC-eTC to reinforce the concepts required for obtaining passing scores in the above tests.

Students are not to discuss tests/quizzes with one another. Test discussion outside of formal review with instructor is considered cheating. As a reminder, cheating is reported.

External resources are not permitted unless expressly directed by the instructor.

Resources

- Programming Fundamentals Student Guide
- Operating Systems Student Guide
- Networking Concepts and Protocols Student Guide
- Windows Student Guide
- UNIX Student Guide

Module Objectives

Upon successful completion of this module, students will:

9.1 Understand fundamental programming concepts.

9.2 Explain programming's role in the CNO workforce.

9.3 Identify different language tiers and types of programming languages.

9.4 Identify and use program structural components.

9.5 Create procedural programs.

9.6 Explain Object-Oriented Programming concepts.

9.7 Analyze, create, and compile simple Python programs.

9.8 Describe the use of classes and objects in programming.

9.9 Identify and follow network programs.

9.10 Distinguish and create basic HTML web pages.

9.11 Use Python as a Common Gateway Interface (CGI) to process web pages.

9.12 Design, develop, and use regular expressions to parse data.

Objectives

Day 1

At the end of this training session, students will:

- ❖ Understand the differences between Interpreters and Compilers.
- ❖ Describe scripting structures (branching and repetition structures).
- ❖ Identify source code on a host.
- ❖ Identify program types.

Exercises

This training session includes the following exercises:

- ❖ Exercise 9-1, Basic Scripting
- ❖ Exercise 9-2, Understanding the Flow of Execution

JCAC-eTC

Complete self-study activities and assignments in the online training environment.

1 Compiled vs. Interpreted/Scripting Languages

Recall that compiled languages convert source code permanently into machine code stored within executable binary files. In contrast, scripting languages convert source code into machine code on the fly by another program called an interpreter. Scripting languages are easier to learn and coding is faster than in more structured, compiled languages such as C/C++.

1.1 Compiled Language: C/C++

When a compiler processes source code, it converts the source code to machine language and produces an executable binary or program. Bugs or errors found within the source code file(s) are listed at the end of the compilation process.

Note: Converting source code into an executable binary or program is known as the compilation process, or compiling a program.

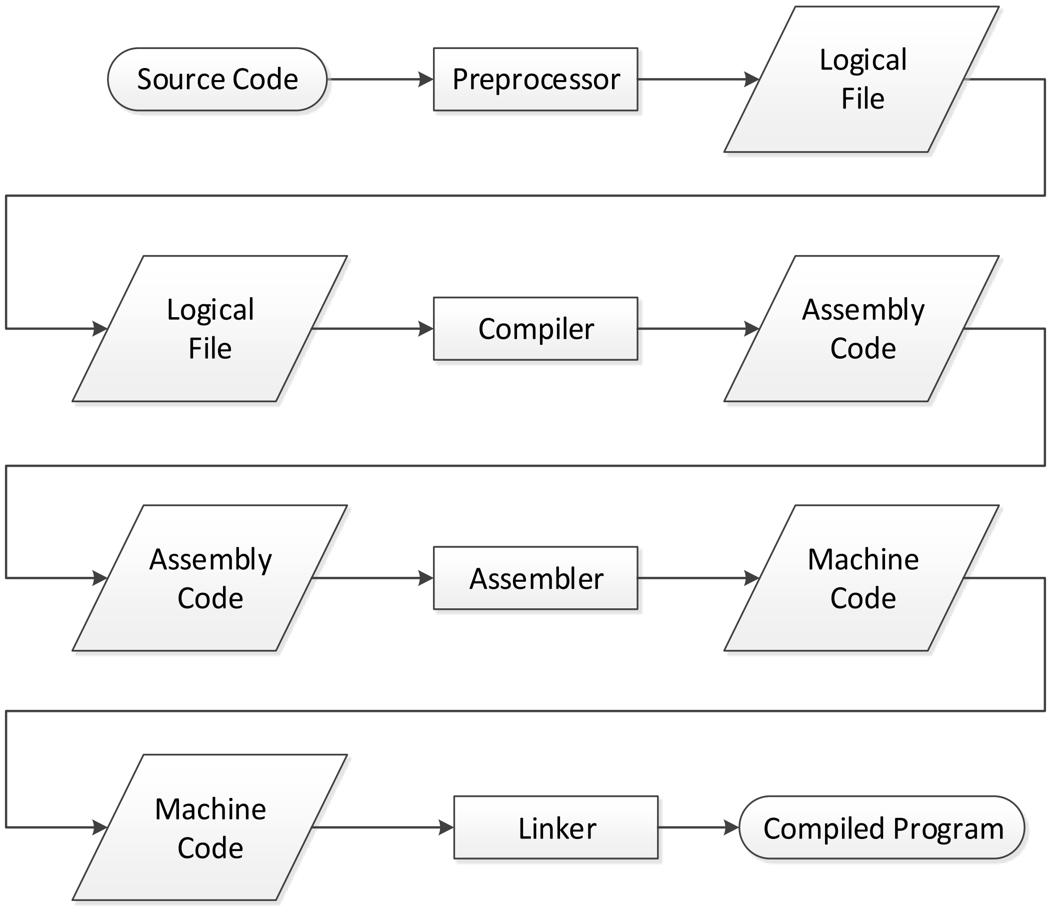


Figure 1. Compilation process.

1.2 Interpreted/Scripting Languages

An interpreter is a scripting language processor that converts source code into machine code and executes each line of code. Unlike a compiler, an interpreter reads one line of code, parses and understands the code, converts it into machine code, and then executes the line of code. The interpreter repeats the process until it reaches the end of the source code.

If an error occurs, the interpreter interrupts the interpretation process and identifies the location of the error in the source code. The scriptwriter corrects the identified source code error and then executes the program again.

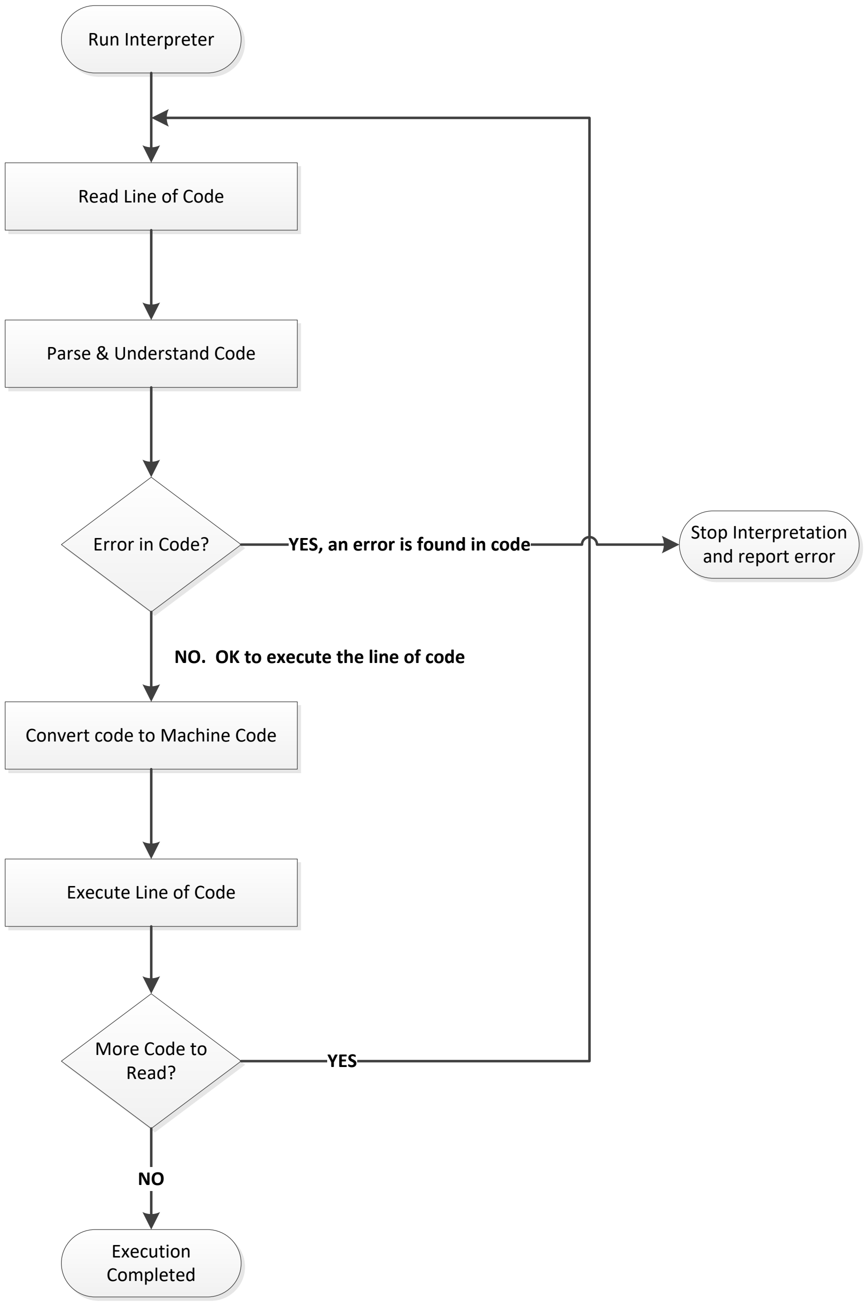


Figure 2. Code interpretation process.

2 Scripting

Scripting automates the execution of multiple tasks that would normally be executed one-by-one by a human operator on a computer system.

Common scripting languages include:

- Bash
- Python
- PowerShell

Scripts may be created and saved using a plaintext editor such as Notepad, NotePad++, PowerShell ISE, Visual Studio, Geany, vi, or Eclipse. Use descriptive filenames with file extensions to indicate a characteristic of the file contents or intended use.

Table 1. Common script file extensions.

Extension	Type of File	Example
.ps1	Windows PowerShell shell script	Get-PortListing.ps1
.py	Python source file	Echo_server.py
.sh	UNIX Bash source file	Get_Date.sh

2.1 Scripting Source Code

Text editors provide a means for writing and storing script files. A script's source code specifies the instructions a computer is to perform using interpreted programming languages. The script user invokes an interpreter to execute the code.

Sample PowerShell Scripting Language:

```
# Ask user to enter message and print it
Write-Host “Enter a short message: “ -NoNewLine
$shortMsg = Read-Host
Write-Host “The message you entered is: $shortMsg“
```

Sample Output:

```
Enter a short message: Mod 9 Class ROCKS!
The message you entered is: Mod 9 Class ROCKS!
```

See Information Sheet 9-1 in Student Workbook

Scripting Languages – Quick Reference Guides

Complete Exercise 9-1 in Student Workbook

Basic Scripting

2.1.1 Pseudocode

Pseudocode uses a structured, natural language (e.g., plain English) to leave a clear picture of the logical processes necessary to solve a problem. Pseudocode presents, in a clear and understandable way, the concept or algorithm it solves. It obscures the technical details and scripting syntax, so the reader is not distracted. Variables provide an easily referenced way to store information and are used in pseudocode as well as the actual scripts. Pseudocode is not a document meant for a computer to process but is helpful for individuals in understanding a program's design. Pseudocode is converted into actual instructions by a programmer for an interpreter to execute.

2.1.2 Comments

In scripting, a comment is an explanation or annotation in the source code of a computer program, usually written by the scriptwriter. Comments are added to make the source code easier for humans to understand and are generally ignored by compilers and interpreters.

Comments may be either multi-line comments or inline comments. In UNIX/Bash, PowerShell, and Python, comments begin with # (octothorpe), which causes the following characters up to a <newline> to be ignored by the interpreter. Comments may be on a line by themselves or share a line with a command.

2.2 Flow Control

Flow Control is the process of how a script executes. Scripting starts with understanding requirements. The design defines the steps taken to fulfill a script's requirements. It becomes the blueprint for a script, defining what the script accomplishes. This blueprint begins with pseudocode before progressing to actual script development. While in development, comments are used to ensure the source code is easier to understand.

Scripting techniques define flow control structures, like branching and looping statements, that determine which sections of code execute and how many times. Those statements are generally contained with one or several code blocks within the script. Variables are often used to store information for easy reference. The result is a script that is easy to read and understand and efficient in its operation.

2.2.1 Scripting Techniques

Understanding several scripting techniques and concepts is essential to solving a problem or designing an algorithm to produce a smooth, efficient script. Three basic scripting techniques are commonly used:

Sequential Action	A linear progression of tasks such as input, output, and computation.
Branching Structure	A selection between different courses of action.
Repetition Structure	A repetition structure to process the same sequence of statements (code block) multiple times.

Depending on the scripting language used, a **code block** is the grouping of code which, when executed, performs a task. Code blocks are used throughout all scripts. Code executing within a branching or repetition structure is in a code block. Functions and procedures contain all their code within a code block, usually enclosed by curly braces.

Sequential Action

Sequential action refers to computer instructions that execute one at a time, in a written order, from beginning to end. In the same way that a person reads a book (i.e., in a written order, without skipping paragraphs or pages), a computer executes a script comprised of instructions.

Example of sequential action:

Step 1: Receiving the first three octets of an IP address from the user.

Step 2: Calculating the next IP address to scan from input received.

Step 3: Sending a signal to the IP address.

Pseudocode authors should be consistent with verb usage to describe actions. Table 2 shows common verbs to describe sequential action in pseudocode.

Table 2. Sequential action verbs.

Technique	Term	Purpose
Input	READ, OBTAIN, GET	Receives input from a user or other data source such as the Internet.
Output	PRINT, DISPLAY, SHOW	Provides information to a user by displaying it to a screen or other output device such as a printer.
Compute	CALCULATE, COMPUTE, DETERMINE	Performs a mathematical operation.
Initialize	SET, INIT, CREATE	Sets the initial value of a variable.
Add One	INCREMENT, BUMP	Adds one to a value.

The following pseudocode shows variables declared and initialized with values:

```
SET VARIABLE IPAddr VALUE TO "192.168.122."
SET VARIABLE ValidIP VALUE TO invalid
SET VARIABLE IPList TO BE AN EMPTY LIST
GET VALUE OF VARIABLE Oct4 FROM USER
```

Branching

Branching allows a script to choose which series of instructions to execute. Using the key word **`IF`**, a computer uses a Boolean expression (i.e., condition) to decode whether certain instructions execute. This structure allows a script writer to "branch to" or "jump to" one or more paths of execution for the program to follow based on conditions.

An **`IF`** statement evaluates a Boolean condition and, if the resulting condition is true, executes statements located in the **`IF`** statement's code block. An **`IF`** structure is everything between the opening **`IF`** and the closing **`ENDIF`**.

The general form of an **`IF`** structure is as follows:

```
IF <condition 1> THEN
    Instruction when condition 1 is true
    Instruction when condition 1 is true
    Instruction when condition 1 is true
ELSE IF <condition 2> THEN
    Instruction when condition 2 is true
    Instruction when condition 2 is true
ELSE IF <condition 3> THEN
    Instruction when condition 3 is true
ELSE
    Instruction when above conditions are false
    Instruction when above conditions are false
    Instruction when above conditions are false
    Instruction when above conditions are false
ENDIF
```

When building an **`IF`** structure, only the first **`IF`** statement is necessary, the **`ELSE IF`** is optional and used when logic dictates that more than one action may be necessary for a given condition. Optionally, there can only be one **`ELSE`** statement for any given **`IF`** block, and the **`ELSE`** statement does not have a condition. An **`ELSE`** is optionally the last condition.

Multiple conditions can be combined using logical operators, such as AND and OR. The following simple **`IF`** statement triggers if the *Octet4* value is 0 or greater and less than 256:

```
GET Octet4 FROM USER
IF Octet4 IS GREATER THAN -1 AND Octet4 IS LESS THAN 256 THEN
    DISPLAY "You have a valid octet value…"
ENDIF
```

The following pseudocode performs a specific set of instructions by adding multiple **`ELSE IF`** statements:

```
SET VARIABLE MenuChoice VALUE TO “0”
DISPLAY TEXT “1.  Build list of valid IP Addresses”
DISPLAY TEXT “2.  Show list of valid IP Addresses”
DISPLAY TEXT “3.  Scan devices”
GET VALUE OF VARIABLE MenuChoice FROM USER

IF VALUE OF MenuChoice IS EQUAL TO “1” THEN
    DISPLAY TEXT “You chose menu option 1”
ELSE IF VALUE OF VARIABLE MenuChoice IS EQUAL TO “2” THEN
    DISPLAY TEXT “You chose menu option 2”
ELSE IF VALUE OF VARIABLE MenuChoice IS EQUAL TO “3” THEN
    DISPLAY TEXT “You chose menu option 3”
ELSE
    DISPLAY TEXT “You made an invalid menu choice”
ENDIF
```

Read the pseudocode above and answer the following questions:

What displays if the user types 1 on the keyboard? ___________________________

What displays if the user types 3 on the keyboard? ___________________________

What displays if the user types 5 on the keyboard? ___________________________

Repetition Structures

Repetition, or looping structures, cause a script to execute a set of actions multiple times. Like branching statements, loops use a Boolean condition to determine if its code block should execute. Some keywords for loops are: **`while`**, **`for`**, and **`foreach`**.

WHILE Statement

A **`WHILE`** statement checks a Boolean expression (condition) before it enters the code block of the loop. If the resulting Boolean condition returns a true value, the script executes the body of the loop. When the end of the loop is reached, script execution returns to the first line of the loop and checks the Boolean expression. If the condition is true, the script repeats the body of the loop. This cycle continues until the resulting Boolean condition is false, at which point execution of the script continues with statements after the loop.

The general form of a **`while`** loop is:

```
WHILE <condition>
    Instruction when condition is true
    Instruction when condition is true
    Instruction when condition is true
ENDLOOP
```

The following example represents a script that constructs an IP address, checks that it is valid, and adds only the valid IP addresses to a list:

```
WHILE VALUE OF VARIABLE Oct4 IS LESS THAN 256
   APPEND VALUE OF VARIABLE Oct4 TO END OF IPAddr
   TEST IPAddr FOR RETURN SIGNAL AND STORE RESULT IN ValidIP

   IF VALUE OF ValidIP VALUE IS valid THEN
      DISPLAY VALUE OF IPAddr AND “is valid”
      APPEND VALUE OF VARIABLE IPAddr TO END OF IPList
   ELSE
      DISPLAY TEXT IPAddr AND “is invalid”
   ENDIF

   INCREMENT VALUE OF VARIABLE Oct4 BY 1
   SET VALUE OF VARIABLE ValidIP TO invalid
   SET VALUE OF VARIABLE IPAddr VALUE TO “192.168.122.”’
END LOOP
```

FOR Statement

The **FOR** statement is another useful repetition structure; often called a counting loop because it iterates through a sequence of numbers. It is best to describe loop constraints using simple, understandable vocabulary that infers a loop’s meaning, such as the examples in Table 3.

Table 3. FOR loop statements.

Sample FOR Loop Statements
FOR Number IS A VALUE STARTING AT 1 AND ENDING AT 10
FOR Month IS A VALUE STARTING AT 1 AND ENDING AT 12
FOR IndexValue IS A VALUE STARTING AT 0 AND ENDING AT LENGTH OF STRING VALUE

The following pseudocode iterates through each number starting at 0 and ending at 255, builds an IP address, scans a computer for the given IP address, and saves the gathered information to a file.

```
FOR Octet4 RECEIVES A VALUE STARTING FROM 0 AND ENDING AT 255
   SET VALUE OF VARIABLE IPAddr TO “192.168.122.”
   APPEND VALUE OF VARIABLE Octet4 TO END OF IPAddr
   USE PING COMMAND TO TEST VALIDITY OF VALUE OF VARIABLE IPAddr

   IF VALUE OF VARIABLE IPAddr IS A VALID IP ADDRESS THEN
      SCAN COMPUTER USING VALUE OF IPAddr AND GATHER INFORMATION
      SAVE GATHERED INFORMATION IN A FILE
   ENDIF
ENDFOR
```

FOREACH Statement

A **FOREACH** statement is another repetition structure for traversing items in a collection. **FOREACH** statements do not use an explicit counter. The statements essentially say, "do this to everything in this set or list," rather than "do this *x* number of times." Table 4 provides some examples of pseudocode using **FOREACH** loop statements.

Table 4. Pseudocode examples of FOREACH loop statements.

Pseudocode Examples of FOREACH Statements
FOR EACH Filename IN THE CURRENT FOLDER
FOR EACH PortNumber IN THE PortNumber LIST
FOR EACH LetterValue IN THE STRING VALUE

The following pseudocode iterates through a list of IP addresses, scans a computer for a given IP address, and saves the gathered information to a file:

```
IPAddrList = [“192.168.122.20”, “192.168.122.24”]
FOREACH IPAddr IN LIST IPAddrList
   USE PING COMMAND TO TEST VALUE OF VARIABLE IPAddr

   IF VALUE OF VARIABLE IPAddr IS A VALID IP ADDRESS THEN
      SCAN COMPUTER USING VALUE OF IPAddr AND GATHER INFORMATION
      SAVE GATHERED INFORMATION IN A FILE
   ENDIF
ENDFOR
```

Break/Continue

Occasionally, a program needs to exit a loop body immediately, or immediately execute the next loop iteration without completing the current iteration. Scripting provides two statements to accomplish this task.

`break`	Terminate the current loop and immediately continue with the next statement after the loop body. Any code within the current loop, after the **`break`** statement, is skipped and ignored.
`continue`	Terminate the current loop iteration and immediately execute the next iteration of the loop without completing the current iteration.

Nested Loop

A nested loop is used for repeating over a list of objects. In the below example, the program repeats through a list of values from 0 through 59 for minutes and seconds. The nested loop on line 5 must iterate through all values in the data set before the main loop iterates to the next value.

```
1  SET Hours TO 0
2  SET Minutes TO 0
3  SET Seconds TO 0
4  FOR VALUE OF VARIABLE Minutes IS A VALUE FROM 0 TO 59
5     FOR VALUE OF VARIABLE Seconds IS A VALUE FROM 0 TO 59
6        DISPLAY TEXT “0:” + VALUE OF Minutes + “:” + VALUE OF Seconds
7     END FOR
8  END FOR
```

Output:

```
0:0:0
0:0:1
0:0:2
.  .  .
0:0:58
0:0:59
0:1:0
```

Note: Be sure to complete the entire inner loop before moving on to the next iteration of the outer loop. A common mistake among new programmers is only iterating one time through the innermost loop.

See Information Sheet 9-2 in Student Workbook

Pseudocode Translator

Evaluate the following pseudocode and answer the question below.

```
SET VARIABLE IPAddr VALUE TO “192.168.122.”
SET VARIABLE ValidIP VALUE TO invalid
SET VARIABLE IPList TO BE AN EMPTY LIST
GET VALUE OF VARIABLE Oct4 FROM USER
IF Oct4 IS GREATER THAN -1 AND Oct4 IS LESS THAN 256 THEN
   WHILE VALUE OF Oct4 IS LESS THAN 256
      APPEND VALUE OF Oct4 TO END OF IPAddr
      TEST IPAddr FOR RETURN SIGNAL AND STORE RESULT IN ValidIP

      IF VALUE OF ValidIP IS valid THEN
         DISPLAY VALUE OF IPAddr AND “is valid”
         ADD IPAddr VALUE TO END OF IPList
      ELSE
         DISPLAY VALUE OF IPAddr AND “is invalid”
      ENDIF

      INCREMENT VALUE OF VARIABLE Oct4 BY 1
      SET VARIABLE ValidIP TO invalid
      SET VARIABLE IPAddr VALUE TO “192.168.122.”
   END LOOP

   FOREACH IPAddr IN VARIABLE IPList
      SCAN COMPUTER AT VALUE OF IPAddr AND GATHER INFORMATION
      SAVE INFORMATION IN A FILE
   ENDFOR
ELSE
   DISPLAY TEXT “Invalid fourth Octet value”
ENDIF
```

What task does this pseudocode perform?

____

____

Complete Exercise 9-2 in Student Workbook

Understanding the Flow of Execution

Objectives

Day 2

At the end of this training session, students will:

- Describe scripting structures (variables and functions).
- Understand components of Object-Oriented Programming (OOP).
- Evaluate the components of Object-Oriented Programming (OOP), classes, and objects.
- Describe the purpose of regular expressions.
- Understand structure and use of regular expressions.
- Perform Text Manipulation using regular expressions.

Exercises

This training session includes the following exercises:

- Exercise 9-3, Understanding Variables
- Exercise 9-4, Understanding Functions
- Exercise 9-5, Basic Regular Expressions
- Exercise 9-6, Regular Expressions

JCAC-eTC

Complete self-study activities and assignments in the online training environment.

2.3 Variables

Variables store information that can be easily referenced and manipulated in a script and provide a way of labeling a type of data with a descriptive name, so scripts are easily understood by a reader. It is helpful to think of variable names as containers (e.g., boxes, buckets, cups) that hold different types of information. A variable's sole purpose is to label and store data values in memory, so they are accessible to a script.

For example, a bank account's balance is a variable because it has the parts of a variable. It has a name (e.g., acctBal) and an amount of money in the account (a changeable data value) that can change when the bankcard is used. When making a purchase with the bank card the value in the account goes down, and when making a deposit the value in the account goes up.

A script must declare variables before use. The syntax for declaring a variable differs slightly from one scripting language to another, but it is as simple as declaring a variable name and assigning an initial value to the variable. The interpreter uses the initialized value to determine what type of data is stored.

For the interpreted languages covered in this module, naming conventions state that identifiers contain only numbers, letters, and underscores, and cannot begin with a number. The capitalization of a single letter in two otherwise identical variable names indicates each is unique and each holds different values.

2.3.1 Data Types

A data type is a classification to specify the value of a variable and what type of mathematical, relational, or logical operations may be applied to the variable without causing an error. For example, a string data type classifies text, an integer data type classifies whole numbers, and a float data type classifies real or decimal-based numbers. Table 5 shows basic data types and their availability within specific scripting languages.

Table 5. Available data types by scripting language.

Data Type	Bash	Python	PowerShell
Boolean	⊗	✓	✓
Character	⊗	⊗	✓
Float	⊗	✓	✓
Integer	⊗	✓	✓
String	✓	✓	✓

2.3.2 Arrays/Lists

An array is an orderly arrangement of data values also known as a collection of values. In scripting, an array is a variable that contains multiple values, often of different data types. An array is zero-based indexed, which means indexing or accessing the values within an array always starts with a zero (0).

Two terms associated with arrays/lists are:

Index	References a position in an array/list. The first available index position is always index zero.
Element	The actual value in the array/list. The first element in the list is always element one.

Therefore, the value stored in element #1 is referenced by its index value of zero.

Note: UNIX/Bash and PowerShell use the term array, and Python uses the term list to refer to the concept of an array.

The pseudocode below shows the declaration of one array variable to hold the value of each port number.

```
SET PortList TO 21, 22, 25, 80, 143, 443
```

A memory representation of the declaration of the array variable shown above:

Element #	1	2	3	4	5	6
Values	21	22	25	80	143	443
Index #	0	1	2	3	4	5

2.3.3 Classes and Objects

Writing a script in PowerShell or Python requires constant work with classes and objects.
A class is the blueprint that defines the abstract details of a thing. When talking about the class of chairs in the classroom, for example, the conversation is about the fact that each chair (object) has legs, a seat, a color, and so on. The height may be different for each chair in the classroom, but that does not matter. When talking about a class of things, the focus is on the properties that each of these things possess. An object is known as an instance of a class and does not exist until the programmer creates or instantiates the object.

Structure of a typical Python class:

```
class ClassName( object ):
    def __init__( self ):
        Constructor Code
    def __repr__( self ):
        Representation Code
    def UserMethod( self ):
        User Method Code
```

Note: The keyword **`self`** represents the object itself, similar to the C++ keyword **`this`**. Python requires the keyword **`self`** as the first parameter of a method, unlike C++.

Classes are defined in a file, which usually contains only the class definition and its methods. The name of the file is usually the class name for clarity and is used when imported into the interpreter. Importing a file allows a programmer to call a class to create an object for use in the script. The object name can then be used to access the attributes and methods of the class.

Dot Notation

Dot notation provides a means to access the data and methods of an object. To use dot notation, supply the name of an object (i.e., variable) followed by a period (.) and the name of the object's attribute to access or the method name to execute.

<object>.<attribute>

```
chair.color = “black”;
```

<object>.<method_name>()

```
chair.setRoom(116);
```

Programming Fundamentals	Arrow notation from C/C++ is “dot notation” for pointers and uses the arrow operator to reference attributes and methods for an object (this->quarters, this->GetDimes(), etc.).

2.3.4 Typecasting

Typecasting is a method of temporarily changing a value from one data type to another to ensure a variable is processed correctly.

An example of typecasting is converting an integer to a string. Typecasting to create an IP address, where the IP subnet “192.168.0.”, is saved as a string (*IPSub*), and the 4th octet (*oct4*) is saved as an integer. When joining the two types of data, convert the *oct4* variable to a string and then concatenate (join) it to *IPSub* to create a complete IP address.

PowerShell example of typecasting:

```
# This is a PowerShell script
$oct4 = 0 # Creates an integer data type
while ($oct4 -lt 256)
{
   # Use type cast to convert oct4 from integer to string value
   $IP = “192.168.0.” + [string]$oct4
   $Result = Test-Connection -ComputerName $IP -Count 1 -Quiet
   if ($Result)
   {
      Write-Host “Valid IP: $IP”
   }
   $oct4 = $oct4 + 1
}
```

Note: Typecasting does not change a variable’s data type permanently. Type conversions require an assignment into a variable.

2.3.5 Scoping

The specific place in code where variables are declared has a direct impact on how functions in a script access the variables. Variable scope defines where a variable is accessible and how long its value is available. Scope is dependent upon where a variable is declared.

Programming Fundamentals	A compiled program has two main scopes, global and local. A global variable exists for the life of a program and is accessible by the entire program, while a local variable belongs to a specific block of code.

Global Variables

A variable defined in the main body of a source code file (i.e., outside of all functions) is a global variable that is readable and modifiable throughout the entire source code. Use global variables carefully. Only place object definitions intended for global use, like functions and classes, within a global scope.

Local Variables

A variable declared inside a code block is local to that block of code. It is readable and modifiable from the point at which it is declared until the end of the code block. It exists for as long as the code block is executing.

Shell Environments

In shell environments, local scope refers to variables declared within the current session of the shell interpreter, and their values are only accessible in that specific session. When opening a new session or starting a script none of the variables declared in the previous CLI session are present. Global scope refers to environmental values initiated by the OS within the environment.

UNIX	The **env** command displays all existing shell environmental variables (e.g., $PATH, $USERNAME, $HOSTNAME, $USERDOMAIN, etc.).

Complete Exercise 9-3 in Student Workbook

Understanding Variables

2.4 Functions

A function is a named section of a script that performs a specific task. Functions simplify programs by using procedures, sometimes referred to as routines, subroutines, or methods. To use a function, a script “calls on” a function by name to execute. Depending on a function declaration, a script may also pass values (arguments) necessary for the function to execute correctly. A function may be called at any point during a program's execution and may also be called by other functions. Creating functions divides a program into smaller steps, allowing programmers to create a manageable solution for a complex problem.

Parameter names in a function behave like local variables. They contain the values passed into the function when the function is called. The function cannot complete a task without this information.

An example of a function in Python:

```python
#!/usr/bin/env python
import os

tgtHosts = [0, 63, 127, 191, 255]

def pingNet(targetNet):
   for tgtHost in tgtHosts:
      os.system("ping " + targetNet + str(tgtHost) + " -c 1")

def main():
   tgtNet = raw_input ("Enter first 3 octets (###.###.###.): ")
   pingNet(tgtNet)

main()
```

A function may optionally send back a value in what is known as the function’s return value. Normally, a function exits when it reaches the end of its code block or a return statement. In PowerShell and Python, a return statement may be a return with no value or a return value of any data type. In UNIX/Bash, the return value must be an integer.

Table 6. Return statement examples.

Statement Syntax	Example	Explanation
`return`	`return`	Flow control returns to the line of code that called the function; no value is sent back.
`return` *<variable_name>*	`return total`	Returns as above, but replaces the function call with the value of the identifier sent back to the line that called the function.
`return` *<value>*	`return 3.14159`	Returns as above, but just sends the value given.

Example of a Python return statement:

```
1  #!/usr/bin/env python
2
3  def addition( num1 = 5, num2 = 10 ):
4     sum = num1 + num2
5     return sum
6
7  print “The result is: “, addition(120, 20)
8  print “The result is: “, addition(50)
9  print “The result is: “, addition()
```

Output:

```
The result is:  140
The result is:  60
The result is:  15
```

Note: Line 3 above provides an example of default parameters. Default parameters allow the calling of function parameters without requiring values passed as arguments. If no argument is passed, the variable is initialized to the specified value.

Example of Powershell return statement:

```
Function Addition( $num1, $num2)
{
   return ($num1 + $num2)
}
Write-Host “The sum of 10 and 20 is” (Addition 10 20)
```

Output:

```
The sum of 10 and 20 is 30
```

2.5 Redirection

Redirection commonly refers to directing input and output to files and devices other than default input/output (I/O) devices. Table 7 lists three standard data streams, the default devices for each, and stream numbers used for redirection operations.

Table 7. Data streams.

Data Stream	Default Device	Stream Number
STDIN	(Keyboard)	0
STDOUT	(Monitor)	1
STDERR	(Monitor)	2

Redirection operators can override these defaults so that a command or program accepts input from another device and sends output to a different device. A Linux/Windows shell uses redirection to move streams of data to different locations. For example, there may be times when a user wants to run a command and redirect error messages to a garbage file or a NULL hardware device, so they do not show up on the screen.
Redirecting a stream to a virtual hardware device (e.g., */dev/null* on Linux, *$NUL* on Windows) ignores incoming data.

Table 8 lists redirection symbols most shells recognize.

Table 8. Redirection symbols.

Redirection Symbol	Description
>	Redirect and write to a file. Defaults to the STDOUT data stream. Stream numbers can optionally specify a different source (e.g., ***2>*** redirects and writes STDERR to a file).
>>	Redirect and append to a file. Defaults to the STDOUT data stream. Stream numbers can optionally specify a different source (e.g., ***2>>*** redirects and appends STDERR to a file).
>&	Redirect one data stream to another (e.g., ***2>&1*** redirects both STDOUT and STDERR to the same place).
<	Redirect STDIN to come from a file. Not available in PowerShell.
|	Redirect STDOUT of the command on the left to the STDIN of the command on the right. Multiple redirects create a pipeline.

Complete Exercise 9-4 in Student Workbook

Understanding Functions

3 Regular Expressions

Regular expressions are strings that describe data or data patterns for which a user wishes to search. Regular expressions are useful for validating, parsing, and filtering data, as well as performing search-and-replace operations. Programmers generally use them to search strings, extract desired parts of strings, and compare to values in Boolean conditions.

Pattern matching is one of the most common uses of regular expressions. Examples include patterns typed into a search engine to find web pages and patterns used to validate data entered into a form.

3.1 Creating Regular Expressions

Regular expressions use sequences of characters and meta-characters to describe the data for which the programmer is searching. Combining multiple meta-character sequences, programmers can construct sophisticated expressions matching precise data.

Table 9. Regular expression meta-characters.

Meta-character	Name	Description
\	Backslash	Adds or removes meaning to the character that follows; also known as the escape character.
[ ]	Square Braces	Used to match only one out of several characters.
{ }	Curly Braces	Specifies the number of previous tokens to match.
*	Asterisk	Quantifier to indicate 0 or more of the previous token.
+	Plus Sign	Quantifier to indicate 1 or more of the previous token.
?	Question Mark	Quantifier to indicate 0 or 1 of the previous token.
^	Caret	The first symbol of expression to indicate no data can come before it or as a NOT if in square brackets.
$	Dollar Sign	Last symbol of expression to indicate no data can come after it.
.	Dot	Wildcard symbol used to match any character except a line break character (\n)
( )	Parentheses	Creates a group that may be referenced in other places of an expression.
|	Pipe	Specifies alternation where the regular expression on the left or the regular expression on the right is matched.

3.2 Escape Character (\ )

Backslash (\ ), known as the **escape character**, is used to turn off (escape) the special meaning of a meta-character or to confer special meaning to other characters.

Table 10. Regular expression escape character uses.

Escape Character Use	Description
`\\`	Match a backslash
`\n`	Match a <newline> by conferring special meaning to 'n'
`\t`	Match a <tab> by conferring special meaning to 't'
`\1 … \9`	References a group number 1 through 9
`\(     \)`	References actual open or close parenthesis
`\{     \}`	References actual open and close curly braces
`\[     \]`	References actual square brackets
`\.`	References actual period character
Look at the examples below and fill-in the match:	
Regular Expression	Match (fill-in)
`\(202\)456-1111`	
`\[Welcome\]`	
`C:\\Users\\John\.Doe`	
`name@whatever\.com`	
`192\.168\.122\.5`	

3.3 Character Class ([ ])

A **character class** (`[ ]`) is a list of values placed inside square braces to match a single character in data. A caret (^) as the first character negates the list, and anything NOT in the list is matched. A hyphen (-) is commonly used to denote a range of characters as in a-z, but can also be a character to match. If a hyphen is a character to match, it must be the first or last character in a character class. Table 11 shows several commonly used character classes and predefined shortcuts one could use in place of the standard long format.

Table 11. Regular expression character classes and shortcuts.

Character Class	Match	Shortcut
`[0-9]`	Any digit character	`\d`
`[^0-9]`	Any non-digit character	`\D`
`[A-Za-z0-9_]`	Any word character	`\w`
`[^A-Za-z0-9_]`	Any non-word character	`\W`
`[\t\r\n\f ]`	Any whitespace character	`\s`
`[^\t\r\n\f ]`	Any non-whitespace character	`\S`
Look at the examples below and fill-in the match:		
Regular Expression	Match (fill-in)	
`the`		
`[the]`		
`1[8-9]`		
`2[0-9]`		
`3[0-5]`		
`\d\d\d\.\d\d\d\.\d\d\d\.\d\d\d`		
`0x[0-9a-fA-F][0-9a-fA-F]`		
`[-+*/%]`		

3.4 Quantifier ({ })

A token is either a single character or a group of related characters. A **quantifier** (**{ }**) is used to indicate how many times the previous token should be repeated. When using quantifiers, do not use spaces between the curly braces. The letter ‘n' represents a numeric value.

{**n**}	Exactly *n* of the previous token
{n,}	At least *n* occurrences of the previous token
{,n}	At maximum *n* occurrences of the previous token
{n1,n2}	At least *n1* but not more than *n2* of the previous token

Table 12 shows quantifier ranges and predefined shortcuts used in place of the standard long format.

Table 12. Regular expression quantifier shortcuts.

Quantifier	Match	Shortcut
`{0,}`	Zero or more of the previous token	*
`{1,}`	One or more of the previous token	+
`{0,1}`	Zero or one of the previous token	?
Look at the examples below and fill-in the match:		
Regular Expression	Match (fill-in)	
`[0-9]{3}`		
`\w{5,10}`		
`\d{3,}`		
`\d*`		
`\s+`		
`[a-zA-Z]?\d+`		
`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`		

Complete Exercise 9-5 in Student Workbook

Basic Regular Expression

3.5 Start of String Anchor (^)

A caret (^) is used to match the start of a string, except when part of a character class. If present, a string must start with the pattern described by the regular expression.

Table 13. Regular expression start of string anchors.

Regular Expression	Match (fill-in)
`^\d{3}`	
`^\w{5,10}`	
`^[0-9]{3,}`	
`^\d*`	
`^[^a-z]`	

3.6 End of String Anchor ($)

The dollar sign ($) is used to match the end of a string. If present, a string must end with the pattern described by the regular expression.

Table 14. Regular expression end of string anchors.

Regular Expression	Match (fill-in)
`\d{3}$`	
`\w{5,10}$`	
`[0-9]{3,}$`	
`\d*$`	
`[^a-z]$`	

3.7 Word Boundary Anchor (\b)

A word boundary anchor (\b) is used to match a zero-length position before or after a word character, effectively matching whole words only anywhere in a line of text.

There are three different positions that qualify as word boundaries:

- Before the first character in the string, if the first character is a word character
- After the last character in the string, if the last character is a word character
- Between two characters in the string, where one is a word character and the other is not a word character

Table 15. Regular expression word boundary anchors.

Regular Expression	Description
`\b3\b`	Matches a single ‘3’ that does not have a word character before or after
`\b[0-9]dog`	Matches a single digit followed by ‘dog’ that does not have a word character before
`3\b`	Matches a ‘3’ that does not have a word character after

3.8 Single Character (.)

A dot (`.`) is used to match any single character, except <newline>. Depending on the regular expression processor, flags may be enabled to allow <newline> matching, if desired.

Table 16. Regular expression single character.

Regular Expression	Match (fill-in)
`m.+se`	
`\d{2}.\d{2}.\d{4}`	
`.art`	

3.9 Capturing Group ( ) and Backreferences

Parentheses ( ) are used to make a group of tokens that can be quantified as a unit or to create a list of string values from which to choose. If a match is successful, each captured group is accessible by the regular expression. Using **backreferences** (`\1 … \9`), data matched in a group may also be repeated in an expression.

For example, a palindrome is a word spelled the same forward as it is backward (e.g., racecar, radar, tacocat). The first letter matched must be the last letter in the string, the second letter matched must be the second to last, and so on. One way to accomplish this type of search is by using backreferences since the data of the string is used elsewhere in the expression.

Regular Expression	`(\w)`	`(\w)`	`(\w)`	`\w`	`\3`	`\2`	`\1`
String	R	A	C	E	C	A	R
Group #	1	2	3	-	3	2	1

Table 17. Regular expression group.

Regular Expression	Match (fill-in)
`([bch]at\s){2}`	
`(([0-9a-fA-F]{2}):?){6}`	
`(\d{1,3}\.){3}\d{1,3}`	
`([bchr]at)\s+\1`	
`M(iss)\1(i)(p)\3\2`	

Note: If the matched data does not need to be captured, a question mark and a colon specifies a non-capturing group (?:). This allows for the grouping of regular expressions without the overhead that results when parsing matched data into capture groups.

3.10 Or (|)

A pipe (|) is used to create a selection of strings from which to choose, thereby matching either regular expression on the left or right side.

Table 18. Regular expression or examples.

Regular Expression	Description
You (passed|failed)	Matches "You passed" or "You failed"
([cr]at|[mh]ouse)	Matches "cat" or "rat" or "mouse" or "house"
(1[8-9]|2\d|3[0-5])	Matches numbers in the range of 18 to 35

Complete Exercise 9-6 in Student Workbook

Regular Expressions

Objectives

Day 3

At the end of this training session, students will:

- ❖ Read and understand Bash scripts.
- ❖ Develop a script to parse information from a data set using regular expressions.
- ❖ Write, modify, and maintain simple Bash scripts to perform various actions.
- ❖ Use Bash scripting to perform text manipulation using regular expressions.
- ❖ Traverse, understand, and debug code within Bash scripting language.

Exercises

This training session includes the following exercises:

- ❖ Exercise 9-7, Bash Scripting: Variable Declarations and Output
- ❖ Exercise 9-8, Bash Scripting: Input and Branching Statements
- ❖ Exercise 9-9, Bash Scripting: IP Address and Port Scanner (Tool #1)
- ❖ Exercise 9-10, Using Tool #1 in a Network Environment

JCAC-eTC

Complete self-study activities and assignments in the online training environment.

4 Bash Shell Scripting

A **shell script**, in its basic form, is nothing more than a collection of UNIX commands put into a text file in a specific order of execution to accomplish a task.

4.1 Shell Interpreter

There are several shell interpreters available on UNIX systems. The Bourne Again Shell (Bash) is the focus in this section of programming.

The below UNIX/Bash script displays a message to the screen.

```
1   #!/bin/bash
2   # This script clears the screen and displays “Welcome to Mod 9!”
3
4   clear
5   echo “Welcome to Mod 9!”
```

Output:

```
Welcome to Mod 9!
```

Line 1: The shell declaration must appear on the first line of the script. Because UNIX shell interpreters are different, the shell declaration ensures use of the proper interpreter for which script was written.

Line 2: A comment indicating the purpose of the script.

Line 4: Any command that runs from a terminal prompt also executes within a script. Here, the `clear` command clears the terminal screen.

Line 5: The **echo** command sends text to the terminal screen through the STDOUT device.

Operating Systems

To execute a new script, first, change script file permissions to allow execution:

```
$ chmod 755 <scriptname.sh>
```

And then, execute the script:

```
$ ./<scriptname.sh>
```

4.2 Echo Command

The **echo** command sends arguments to STDOUT, which is usually the screen. When writing data to the screen, **echo** automatically adds a terminating <newline> character. Therefore, each time **echo** executes, information is output to the screen and then the screen cursor moves to the next line.

```
echo This is a test
echo “This is a test, too”
echo ‘This is another line as well.’
```

Output:

```
This is a test
This is a test, too
This is another line as well.
```

The escape character is the backslash character. In UNIX/Bash, **echo** requires the **-e** switch to enable interpretation of the escape sequence. The escape character gives special meaning to the character that follows. The following Bash script shows how to use the escape character and Table 19 lists the variety of escape characters.

```
echo -e “This is a double quote \””
echo -e “This is a single quote ’”
echo -e “This is a long line \c”
echo “that ends here.”
```

Output:

```
This is a double quote “
This is a single quote ’
This is a long line that ends here.
```

Table 19. Escape characters.

Character	Prints
\c	The line without a terminating <newline>
\n	<newline>
\	Backslash Character
\t	Tab Character
\$	Display $ Character
\”	Display the double quote character
\`	Displays the backquote/backtick/grave character

4.3 Variables

In UNIX/Bash, variables are declared by writing the name of the variable followed by the assignment operator (=) and the value to store in the variable. Spaces are not permitted on either side of the assignment operator. The shell has no concept of data types other than the string; therefore, everything in the shell environment is a string.

```
1   IPAddress=192.168.122.          # Assign 192.168.0.  to IPAddress
2   count=1                         # Assign character 1 to count
3   filename=script1.sh             # Assigns ‘script1.sh’ to filename
4   my_dir=/etc                     # Assigns directory path to my_dir
5   oct4=15                         # Assign the value 15 to oct4
6   IPAddress=$IPAddress$oct4       # Concatenate oct4 to IPAddress
7   portNumber=22                   # Assign 22 to portNumber
```

4.3.1 Accessing a Variable’s Value

The value of a UNIX/Bash variable is retrieved by placing a dollar sign ($) in front of the variable name, so the value of *IPAddress* is retrieved by **`$IPAddress`**. Variables may be assigned the value of other variables or a **`null`** value.

```
1   ping -c $count -W $count $IPAddress  # Ping 1 time, 1 sec timeout
2   echo "" > /dev/tcp/$IPAddress/$portNumber    # TCP conn to socket
3   IPAddress2=$IPAddress
4   count=1                              # Assign value 1 to count
5   echo IPAddress2 is: $IPAddress2
6   ls -l $my_dir                        # What does this command do?
7   find $my_dir -name “passwd” 2> /dev/null    # Redirect STDERR
8   mv $filename $my_dir/test/$filename1 # Why does this fail?
```

Sometimes, it is necessary to separate a variable name from the text around it. Constructs are a way to capture a variable value for use in concatenation by separating a variable name from the remainder of a string. Using the **`${variable-name}`** construct, line 6 above can be changed as follows:

```
1   mv $filename ${my_dir}/test/${filename}1   # Success!!!
```

The constructs in the preceding example isolate the variable names from the rest of the string, allowing a programmer to move the file to the specified directory with a new name. Without the construct, as shown on line 6 above, the file would be moved to the new directory without the new file name.

4.3.2 Arrays

UNIX/Bash arrays are zero indexed. The index is always an integer and can be an expression. All values are stored as strings with whitespace delimitation.

```
#!/bin/bash

myArray=(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32)

echo "Value at index 2 is: ${myArray[2]}"
```

Output:

```
Value at index 2 is: 3
```

Additionally, two constructs are very useful:

${name[*]}	When used, returns all the values in the array as a single value. The * can be replaced with an index number to return the value in that element.
${#name[*]}	When used, returns the number of elements in the array. The * can be replaced with an index number to return the length of that value.

```
#!/bin/bash

myArray=(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32)

echo "Values stored in the array: ${myArray[*]}"
echo "Number of values in the array: ${#myArray[*]}"
echo "Length of string element at index 12: ${#myArray[12]}"
```

Output:

```
Values stored in the array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Number of values in the array: 32
Length of string element at index 12: 2
```

4.3.3 Read-only Shell Variables

Read-only shell variables have values automatically assigned to them by the interpreter. Each read-only shell variable contains the value typed in the command line when executing a script or command.

Table 20. Read-only shell variables.

Shell Variable	Definition
$0	Stores name of command used to call current program.
$#	Stores number of arguments passed to script or function.
*$**	References all arguments passed to script or function.
$1, $2…$9	Stores first nine arguments in incrementing variables passed to script or function.
$?	Stores exit status of previous command.

```
#!/bin/bash
echo Display values for Read-only variables.
echo Read-only variables are the values which enter the
echo -e "script through the CLI.  \n"
echo "Sample Command Line:
#           ./pg42readOnlyShellVars_1.sh   192.168.0.  10    5
#           Name of script                 IPSub      oct4  count
#           $0                             $1         $2    $3
echo -e "\nREAD-ONLY Variable and how they are used in this script"
echo -e "\$0: Name of the script:  $0"
echo -e "\$1: The 1st argument value (IPSub):  $1"
echo -e "\$2: The 2nd argument value (oct4) :  $2"
echo -e "\$3: The 3rd argument value (count):  $3\n"
echo -e "\nAbout arguments (values) sent into script"
echo -e "\$#: Number of arguments passed into script: $#"
echo -e "\$*: All arguments passed into the script:   $*\n"
ping -c $3 $1$2
echo "Return status of ping: $?"
```

```
#!/bin/bash
IPAddr=”192.168.122.1”
portNumber=22
# Check if the IP Address and port number are available
# Send a “” to $IPAddr/$portNumber to see if we get a
# response back.  $? will equal 0 which is a valid IP
# otherwise IP and/or port are not available
(echo “” > /dev/tcp/$IPAddr/$portNumber) 2> /dev/null
echo "Return status of TCP connection: $?"
```

4.4 Quotes

Quotes allow the writing of string values with spaces, internal single and double quotes, and internal $ symbols. Quoting conventions are:

Double Quotes Substitutes variables and escaped characters with their actual values.

```
IPAddr=”192.168.122.”
Oct4=”25”
echo “IP Address is: $IPAddr$Oct4”
```

Output:

```
IP Address is: 192.168.122.25
```

Single Quotes Makes everything literal so the values of variables are not substituted.

```
IPAddr=”192.168.122.”
Oct4=”25”
echo ‘IP Address is: $IPAddr.$Oct4’
```

Output:

```
IP Address is: $IPAddr.$Oct4
```

Back Quotes Used to capture the output of a command. A command enclosed in back quotes is replaced by the STDOUT of that command. Shell variable values are substituted within back quotes.

```
today=`date +%A`
echo “Today is $today.”
```

Output:

```
Today is Tuesday.
```

Complete Exercise 9-7 in Student Workbook

Bash Scripting: Variable Declarations and Output

4.5 Read Command

The input of information is an essential part of any script. Scripts require a process to get data from the user, system, or files to perform their designed function. A script reads a line from the keyboard with the **`read`** command. The **`read`** command assigns the contents of the input to the variable listed. The following example shows two input styles.

```
echo -e “Enter an IP Address: (###.###.###.###) \c”
read IPAddress
```

```
read -p "Enter first 3 Octets (###.###.###): " subnet
read -p "Enter beginning address (###): " firstOctet
read -p "Enter final address (###): " lastOctet
echo "Scanning $subnet.$firstOctet through $subnet.$lastOctet"
```

The second example shows how one line of code can be used to effectively accomplish the same as the two lines of code used in the first example. The **`–p`** allows a double quoted prompt.

4.6 Sort Command

The **`sort`** command sorts the contents of a text file, line by line. **`sort`** is a simple and effective tool that rearranges the lines in a text file so that they are sorted, numerically and alphabetically. By default, the rules for sorting are:

- Lines starting with a number appear before lines starting with a letter
- Lines starting with a letter that appears earlier in the alphabet appear before lines starting with a letter that appears later in the alphabet
- Lines starting with a lowercase letter appear before lines starting with the same letter in uppercase

Table 21. Sort command flags.

SORT Flags	Description
-u, --unique	Display only lines that are unique input.
-n, --numeric-sort	Display according to string numerical value.
--help	Display a help message, and exit.

The following pipeline is an example of taking the contents of files with filenames starting with “net”, passing the file contents to **`grep`**, which filters for an IP address 192.168.122.xxx:xxxxx. Output is then sorted and only the unique values are displayed. This is a quick way of extracting specific data from large data files.

```
cat net* | grep -Po “192\.168\.122\.\d{1,3}:\d{4,5}”  | sort –u
```

4.7 Arithmetic Operations

The **expr** tool provides the ability to evaluate a wide range of arithmetic expressions. It evaluates its arguments as expressions and returns the result.

Syntax:

```
`expr <argument1> <operator> <argument2>`
```

expr allows the use of arithmetic operators to produce mathematical expressions. The spacing in an expression is critical. There must be a space between each element of an expression. The following symbols have special meaning to the shell and must be escaped:

*	Asterisk
(	Open parenthesis
)	Closed parenthesis
>	Right angle bracket
<	Left angle bracket

```
#!/bin/bash

read -p "Enter your age: " age
year=`date +%Y` # Output below reflects results from year 2021

echo "You are $age years old."
decAge=`expr $age + 10`
echo "In 10 years, you will be $decAge"

read -p "At what age do you plan to retire: " retAge
retYear=`expr \( $retAge - $age \) + $year`
echo "You plan to retire in $retYear.  Better start saving!"
```

Output:

```
Enter your age: 25
You are 25 years old.
In 10 years, you will be 35
At what age do you plan to retire: 67
You plan to retire in 2063.  Better start saving!
```

4.8 UNIX Scripting Flow-Control Structures

Flow-control structures allow programs to make decisions about what to do next, based on current conditions. In UNIX/Bash scripting, the primary flow-control structures are `if` statements, `while`, and `for` loops.

In UNIX, the `test` command is the basis for a Boolean condition of several flow-control structures and has two forms:

```
test <expression>
[ <expression> ]
```

Compare two strings or numeric values using comparison operators in Table 22.

Table 22. Comparison operators.

String	Numeric	Function
`= -or- ==`	`-eq`	Equivalent to
`!=`	`-ne`	Not equivalent to
`>`	`-gt`	Greater than
`<`	`-lt`	Less than
none	`-ge`	Greater than or equivalent to
none	`-le`	Less than equal to

It may be necessary to create compound statements to check multiple conditions. Table 23 lists useful logical operators.

Table 23. Logical operators.

Operator	Logic
&&	AND
||	OR
!	NOT

4.8.1 Branching Statements

Branching statements are used to choose which set of statements to execute using Boolean logic and conditional branches.

`if` Structure

- The `if` structure evaluates the test expression and returns control based on this status.
- The **`then`** statement marks the beginning of the `if`.
- The **`fi`** statement marks the end of the `if` (note that **`fi`** is `if` spelled backward).
- The **`elif`** (**`else if`**) allows for additional conditions in the `if` structure, and also begins with a **`then`** statement.

```
if [ expression ]
then
    commands
    commands
fi
```

```
if [ expression ]
then
    Commands
    Commands
else
    Commands
    Commands
    Commands
fi
```

```
if [ expression ]
then
    Commands
    Commands
elif [ expression ]
then
    Commands
    Commands
else
    Commands
fi
```

Branching example #1:

```
read -p “Enter a number: “ x
if [ $x -eq 10 ]
then
   echo -e “\$x is 10!”
elif [ $x -eq 5 ] || [ $x -eq 3 ]
then
   echo -e “\$x is 5 or 3!”
elif [ $x -gt 0 ]
then
   echo -e “\$x is positive!”
else
   echo -e “\$x is negative!”
fi
```

Branching example #2:

```
read -p “Enter the first octet:” oct1

if [ $oct1 -ge 0 ] && [ $oct1 -le 255 ]
then
    echo “Octet is valid!”
else
    echo “Octet is out of range!”
fi
```

What is the script’s purpose? ____________________________________

____

Complete Exercise 9-8 in Student Workbook

Bash Scripting: Input and Branching Statements

4.8.2 Loop Structures

The **while** and **for** loop structures are useful in UNIX/Bash scripting techniques to make repetitive tasks easy.

While Loop

```
while [ expression ]
do
      Commands
done
```

The body of a loop is defined between the **do** and **done** statements. The commands executed within a **while** loop must change the **test** expression value or an infinite loop results. The second example illustrates the use of indexing in a **while** loop.

```
read -p “Enter first 3 octets: (###.###.###)” subnet
read -p “Enter a starting octet range: ” startOctet
read -p “Enter an ending octet range: ” endOctet

# Assume we entered valid start and ending octet values
oct4=$startOctet

while [ $oct4 -le $endOctet ]
do
   IPAddr=$subnet.$oct4
   ping -c 1 $IPAddr
   if [ $? -eq 0 ]
   then
      echo Valid IP Address: $IPAddr
   fi
   oct4=`expr $oct4 + 1`
done
```

What is the script’s purpose? ______________________________

____

```
x=0
football=(Buckeyes Sooners Ducks “Fighting Irish”)
while [ $x -lt ${#football[*]} ]
do
   echo ${football[$x]}
   x=`expr $x + 1`
done
```

Output:

```
Buckeyes
Sooners
Ducks
Fighting Irish
```

For Loop

```
for element-item in array
do
   commands
done
```

The **FOR** loop in UNIX/Bash is a **FOREACH** loop as explained earlier.

```
IPAddr=192.168.122.1
portList=(20 21 22 23 25 43 53 69 79 80 88 110 111 443 543)
for portNumber in ${portList[*]}
do
   (echo > /dev/tcp/$IPAddr/$portNumber) 2> /dev/null
   if [ $? -eq 0 ]
   then
      echo “${portNumber} is open”
   fi
done
```

What is the script’s purpose? *Loop through each port number. Test IP and port number*

```
football=(Buckeyes Sooners Ducks “Fighting Irish”)
for team in ${football[*]}
do
   echo $team
done
```

Output:

```
Buckeyes
Sooners
Ducks
Fighting
Irish
```

See Information Sheet 9-3 in Student Workbook

How to use the Practice Environment to Build Tools

Complete Exercise 9-9 in Student Workbook

Bash Scripting: IP Address and Port Scanner (Tool #1)

Complete Exercise 9-10 in Student Workbook

Using Tool #1 in a Network Environment

Objectives

Day 4

At the end of this training session, students will:

- Read and interpret simple Python scripts.
- Identify and use program structural components.
- Write, modify, and maintain simple Python scripts to perform various functions.
- Use Python to parse through data files.

Exercises

This training session includes the following exercises:

- Exercise 9-11, Python: Input/Output, Variables, and Data Types
- Exercise 9-12, Python: Strings and Lists
- Exercise 9-13, Python: Branching and Looping
- Exercise 9-14, Password Cracking Using a Dictionary Attack (Tool #2)
- Exercise 9-15, Using Tools #1 and #2 in a Network Environment

JCAC-eTC

Complete self-study activities and assignments in the online training environment.

5 Python Programming Language

Python is an object-oriented interpreted language, used throughout CNO mission areas for building CNO analysis tools and testing applications. Python is an easy to learn programming language with decreased complexity, code-writing efficiency, limitless third-party libraries, and is available on most UNIX workstations.

Python scripts run on any OS platform with the Python interpreter installed. This negates the need for a programmer to write multiple versions of a program to run on different OS platforms.

NOTE: Python is quick and powerful. However, it is not suitable for time or memory sensitive applications.

A programmer can interact with Python in one of two ways:

The first is by entering one line at a time into the Python interpreter's interactive shell, which starts after entering the command **python** at the shell prompt. This method is quick and useful for testing small snippets of code without writing an entire script.

The second is by saving the source code to a file and instructing Python to process the file. The first line in every Python script indicates the path to the interpreter that should execute the script. This ensures the file is being run with the correct interpreter on a Linux system. On a Windows system, this is not as important since the file extension (*.py*) directs the script to the correct interpreter.

```
#!/usr/bin/env python
```

5.1 Print Statement

The **print** statement is the most common way to display information on the monitor using Python.

The following program displays two string objects on the monitor.

```
#!/usr/bin/env python

print “Welcome to Mod 9.”
print ‘Today, we are learning about Python.’
```

Output:

```
Welcome to Mod 9.
Today, we are learning about Python.
```

The **print** statement automatically inserts a **<newline>** once the string has been displayed. If desired, replace the **<newline>** with a single whitespace character using a comma so output from two or more print statements can share the same line. The comma also allows different data types to be separated within the same print statement, creating a space between each data output.

```
#!/usr/bin/env python

print “Welcome to Mod 9.”,
print ‘Today, we are learning about Python.’
```

Output:

```
Welcome to Mod 9.  Today, we are learning about Python.
```

5.2 Escape Character (\ )

The **escape character** is the backslash character (\ ). The quotation marks of the output (double or single) do not matter.

To use the escape character, type the backslash character followed by another symbol or letter. The escape character adds or removes meaning to characters in a string. It is used to display a symbol Python typically uses as a built-in symbol or add whitespace characters to a string.

Table 24. String format characters.

Character	Description
\n	<newline> character.
\t	Tab character. The width of the tab is set by the terminal program, not the Python script, although it is typically eight spaces.
\” or \’	Removes the meaning of the quote so it can be used as part of the string.
\b	Backspace. Removes previous character from string.

The program below demonstrates use of several common string format characters.

```
#!/usr/bin/env python

print "I \"like\":\nMod 9\nUNIX/Bash\nPowerShell\nand\tPython"
print "Potato" + "\b" + "er"
print "Mod 9 is\b\b\b awesome!"
```

Output:

```
I “like”:
Mod 9
UNIX/Bash
PowerShell
and   Python
Potater
Mod 9 awesome!
```

5.3 Variables

In Python, a variable is used to store, access, and manipulate a variety of data types in a memory location. The type of data can be integers, real numbers (floats), Booleans, strings, or more complex data, such as lists.

```
>>> port = 21
>>> PORT = 25
>>> print “Lowercase port is “, port
Lowercase port is 21
>>> print “Uppercase Port is “, PORT
Uppercase Port is 25
```

The following code declares the variable *port* to store an integer value, and *banner* to store a string value:

```
>>> port = 21
>>> banner = “Company FTP Server”
>>> print “[+] Checking for “ + banner + “ on port “ + str(port)
[+] Checking for Company FTP Server on port 21
```

An added benefit of data types is error checking. Python can report an error, called an exception, if an illegal operation is attempted between two or more different types of variables. For example, adding text to a number is illegal and generates an exception.

NOTE: Remember, variables are objects. As such, in addition to a value, variables have properties and methods. Objects of different data types have different properties and methods.

5.3.1 Typecasting

The Python functions listed in Table 25 perform data type conversions on given objects. A function raises a software-generated error (i.e., an exception) when an operation cannot be carried out successfully.

Table 25. Typecasting functions.

Function	Description
str(<object>)	Attempts to convert the provided object into a string.
int(<object>)	Attempts to convert the provided object into an integer.
float(<object>)	Attempts to convert the provided object into a float.
list(<object>)	Returns a sequenced list of elements from the object passed

The code below shows data conversion of the variable *value* from a string to integer, to float, and back to a string.

```
>>> value = “42”
>>> type(value)
<type ‘str’>
>>> value = int(value)
>>> type(value)
<type ‘int’>
>>> value = float(value)
>>> type(value)
<type ‘float’>
>>> list(str(value))
[‘4’, ‘2’, ‘.’, ‘0’]
```

5.3.2 Helpful Functions

Using Python interactive shell, a programmer can quickly query an object about what it can do, and then continue writing source code. The helpful functions listed in Table 26 are used in the Python interactive shell to get information or even a help page about an object.

Table 26. Python help functions.

Function	Description
type(<object>)	Returns the data type of the object passed.
exit(<object>)	Exits the Python script and returns a number to the terminal.
help(<object>)	Displays the help page from the Python documents.
len(<object>)	Displays the number of elements in a set, like a list or a string.

Example using **type** to display the data type of a variable:

```
>>> banner = “Company FTP Server”          # This is a string
>>> type(banner)
<type ‘str’>
>>> port = 21                              # This is an integer
>>> type(port)
<type ‘int’>
>>> portList = [21,22,25,80]               # This is a list
>>> type(portList)
<type ‘list’>
>>> portOpen = False                       # This is a Boolean
>>> type(portOpen)
<type ‘bool’>
```

5.4 raw_input() Function

`raw_input( )` : **`raw_input( )`** takes exactly what is typed and passes it back as a string. It does not interpret the user input. Whether an integer value or a list is entered, its type is of string only. When **`raw_input( )`** is encountered, the program waits until the user presses the ENTER key before returning the typed data and continuing with the next line of code.

`raw_input( )` can be used with or without a prompt as shown in the following program:

```
#!/usr/bin/env python

user_name = raw_input( “Please type your name: “ )
print “Hello,“, user_name

print “Type a message:”
message = raw_input( )

print user_name, “said:”, message
```

Output with input shown in bold:

```
Please type your name: Frank
Hello, Frank
Type a message:
JCAC is fun!
Frank said: JCAC is fun!
```

Values returned from **`raw_input()`** are stored as strings. A programmer must convert the string to the intended data type, such as a number, before use.

```
#!/usr/bin/env python

salesTax = 0.075
price = float(raw_input("How much is that dog in the window? " ))
totalPrice = price + ( price * salesTax)
print "The one with the waggly tail is $" + str(totalPrice)
```

Output with input shown in bold:

```
How much is that dog in the window? 50
The one with the waggly tail is $53.75
```

Complete Exercise 9-11 in Student Workbook

Python: Input/Output, Variables, and Data Types

5.5 Strings

In Python, strings are immutable, meaning once declared, the value of individual characters within a string cannot be changed. Like a list, strings use an index position to reference a character.

Note: Immutable behavior of strings is overcome by converting a string to a mutable data type and re-saving it as a string.

Create a string using either single or double quotes.

```
>>> myString1 = “Hello Mod 9 Students!”
>>> myString2 = ‘Hello Mod 9 Students!’
```

A string can also be created using triple quotes to create a string exactly as it is typed until it finds the closing triple quotes. This allows strings to span multiple lines in source code.

```
>>> longString = ’’’Why do programmers always
mix up Christmas and Halloween?
Because Dec 25 is Oct 31.’’’
```

Note: Sometimes a programmer uses triple quoted strings for a multi-line comment.

An individual letter can be retrieved from a string by specifying an index position between two square brackets [ ]. The example below stores the letter ‘M’ in the variable *letter*.

```
>>> letter = myString1[6]
>>> print letter
M
```

Note: A string object has methods that can change how the string is displayed or processed.

Table 27. String methods.

Method	Description
string.upper( )	Returns a copy of the string with each letter capitalized.
string.lower( )	Returns a copy of the string with each letter in lowercase.
string.find(<substring>)	Returns the first index position where the substring is found. Returns -1 if the substring is not found.
string.split()	Returns the string as a list delimited by the value passed as an argument, with a default of a whitespace.
string.strip()	Returns the string with leading and trailing characters removed; default is space characters.

Consider the following methods: **`upper( ), lower( ),`** and **`find( )`** and how they are used in the example:

```
>>> banner = “Company FTP Server”
>>> print banner.upper()
COMPANY FTP SERVER
>>> print banner.lower()
company ftp server
>>> print banner.find(‘FTP’)
8
```

String repetition is the process of repeating a series of characters multiple times. The code below adds 10 exclamation marks (`!`) to the end of the string “**`JCAC is fun`**”.

```
>>> myString1 = “JCAC is fun”
>>> myString2 = myString1 + ( “!” * 10 )
>>> print myString2
JCAC is fun!!!!!!!!!!
```

The `split()` method is used to separate values within a string of characters. In the following example, the variable userAcct is made up of two parts: user account and the Salt + Password. These two values are `split()` by the colon symbol (“:”) and returned as a list.

```
#!/usr/bin/env python

userAcct =”lazy.admin:$6$1561568054$abcccd280008f297a713f7d9791”
lstAcctDetails = userAcct.split(":")

print lstAcctDetails
```

Output:

```
[‘lazy.admin’, ‘$6$1561568054$abcccd280008f297a713f7d9791’]
```

5.6 Lists

In Python, a list data structure provides an excellent technique for storing an array of objects. Because it is an array of objects, a programmer can construct lists of any data type. Built-in methods exist for performing actions such as appending, removing, popping, indexing, and sorting lists. Lists are a mutable data set. The example below declares an empty list:

```
>>> portList = []
```

As Python is an object oriented scripting language, *portList* is an object containing methods that can manipulate its own data. Table 28 contains several common methods a list may utilize.

Table 28. List methods.

Method/Function	Description
list.append(<object>)	Appends an object to the end of the list.
list.remove(<object>)	Removes the first element from the list specified by the object value.
list.pop(<index>)	Deletes an element via index position.
list.sort()	Sorts all the items and updates the list at the same time.
list.index(<object>)	Returns the index position of the first item located within the list equal to the object value.

Consider the following example: a programmer can construct a list by appending items using the append method, printing the items, and sorting them before printing them again. The programmer can find the index of a particular element or remove a value.

```
>>> states = ['Ohio', 'Pennsylvania']
>>> states.append('Florida')
>>> states.append('Texas')
>>> print states
['Ohio', 'Pennsylvania', 'Florida', 'Texas']
>>> states.sort()
>>> print states
['Florida', 'Ohio', 'Pennsylvania', 'Texas']
>>> pos = states.index('Florida')
>>> print "Florida is at index " + str(pos) + " after sorting."
Florida is at index 0 after sorting.
>>> states.pop(2)
'Pennsylvania'
>>> states.remove('Texas')
>>> print states
['Florida', 'Ohio']
```

Complete Exercise 9-12 in Student Workbook

Python: Strings and Lists

5.7 Python Scripting Flow-Control Structures

In Python scripting, the flow-control techniques are the `if` branching structure, and the `while` and `for` loops.

To make compound conditions, perform comparisons, and determine membership, use the following operators:

Table 29. Comparison and logical operators.

Comparison Operators	
==	Equivalent to
!=	Not Equivalent to
>	Greater Than
>=	Greater than or equivalent to
<	Less than
<=	Less than or equivalent to
Membership Operators	
in	Is in the container
not in	Is not in the container

Logical Operators	
and	Both conditions must be true
or	One or the other condition must be true
not	Invert the Boolean value

5.7.1 Branching Statements

A Python **`if`** block is defined by an opening **`if`** and any **`elif`** or **`else`** statements and their bodies of code. Conditional statements in Python are terminated with a colon and the body of the statement is indented. To leave the body, return to the previous indention level.

Typical structure of an **`if`** block:

```
if condition:
    Body for if statements
elif condition:
    Body for elif statement
else:
    Body for else statements
```

An **`if`** block using comparison operators:

```
#!/usr/bin/env python

oct1 = int(raw_input(“Enter the first octet: ”))

if  oct1 >= 0 and oct1 <= 255:
    print “Octet is valid!”
else:
    print “Octet is out of range!”
```

What is the script’s purpose? ____

____

5.7.2 Loop Structures

While Loop

This loop structure executes a block of code repeatedly as long as the conditional expression evaluates to a true value. The condition of the loop terminates with a colon (**:**) and the body of the loop is indented.

```
#!/usr/bin/env python
count = 0
while count < 3:
   print “The value of count is: “ + str(count)
   count = count + 1

print “The value of count, after the loop is, “ + str(count)
```

Output:

```
The value of count is: 0
The value of count is: 1
The value of count is: 2
The value of count, after the loop is, 3
```

Commands executed within the **`while`** loop must change the test expression value or an infinite loop results. Line 5 is the change for the loop control variable, and line 7 makes use of the break statement to exit the loop.

```
1   #!/usr/bin/env python
2   count = 0
3   while True:
4      print “The value of count is: “ + str(count)
5      count = count + 1
6      if (count == 5):
7         break
8
9   print “The value of count, after the loop is: “ + str(count)
```

Output:

```
The value of count is: 0
The value of count is: 1
The value of count is: 2
The value of count is: 3
The value of count is: 4
The value of count, after the loop is: 5
```

```
#!/usr/bin/env python
import os

subnet = raw_input (“Enter first three octets: ”)
oct4 = int(raw_input(“Enter a starting octet: ”))
endOctet = int(raw_input(“Enter an ending octet: ”))

# Assume we entered valid data in variables
while oct4 <= endOctet:
   IPAddr = subnet + “.” + str(oct4)
   pingResponse = os.system(“ping -q -c 1 “ + IPAddr + “ >/dev/null”)
   if pingRespose == 0:
      print “Active IP: “ + IPAddr
   oct4 = oct4 + 1
print “End of program”
```

What is the script’s purpose? ____

____

for Loop

In Python, a **for** loop is used to iterate through objects that hold multiple values, like a string or a list, and processes each value individually. This is equivalent to the FOR EACH loop in pseudocode or the for loop in Bash.

```
#!/usr/bin/env python

football = ['Ohio State', 'Penn State', 'Clemson', 'Alabama',
'Oklahoma', 'Texas', 'LSU', 'USC', 'Michigan']

for team in football:
    if (team != 'Michigan'):
        print team
    else:
        msg = "\nO-H!"

print msg
```

Output:

```
Ohio State
Penn State
Clemson
Alabama
Oklahoma
Texas
LSU
USC

O-H!
```

Complete Exercise 9-13 in Student Workbook

Python: Branching and Looping

5.8 File I/O

File I/O allows programs to create, read, modify, and delete files within directories. Python uses *open()* to request a file handle from an OS. If the *open()* fails, an *IOError* exception is raised.

Syntax for opening a file:

```
fileObject = open( <filepath string>, <mode> )
```

There are several file open modes available (see Table 30).

Table 30. File open modes.

Mode	Description
r	Read mode
w	Write mode
a	Appends to the end of the file
b	Used in conjunction with any of the above modes to allow non-text files to be opened (e.g., image files like JPEGs or GIFs, audio files like MP3s, or binary document formats like Word or PDF)

Close files once the program is done with them to release consumed resources back to the OS. Syntax for closing a file:

```
fileObject.close()
```

```
#!/usr/bin/env python

filename = raw_input(“Enter a filename to edit: “)
fObj = open(filename, “w”)

Username = raw_input(“Enter your name: ”)
Age = raw_input(“Ener your age: “)
fObj.write(“Your name is: “ + Username + “\n”)
fObj.write(“Your age is: “ + Age + “\n”)

fObj.close()
```

What is the script’s purpose? ______________________________

____

5.8.1 File Output

Python uses the write (**“w”**) or append (**“a”**) mode to save information to a specified file.

When a file is opened in write mode, the contents of the file are overwritten if the file already exists. When a file is opened in append mode, additional data is written from the last write location and does not overwrite previous data. Once a file is closed, the write position in the file is reset. The *write()* method only accepts a string as an argument.

```
fileObject = open( “file.txt”, “w” )
fileObject.write( “Joe” + “\n” )
fileObject.write( str(1986) + “\n” )
fileObject.close()
```

Note: The <newline> is not automatically appended when data is written. It is up to the programmer to add it if necessary.

The below program attempts to open a file, specified by a string the user entered for their last name, and write several lines of content to the file.

```
#!/usr/bin/env python

lastName = raw_input(“Enter the your last name:\n> “)
age = int(raw_input(“Enter your age:\n> “))
fileObject = open( lastName + “.txt”, “w” )
fileObject.write( lastName + “\n” )
fileObject.write( str(age) + “\n” )
fileObject.close()
```

5.8.2 File Input

Once a file is open, there are several ways to read the data.

Reading Line by Line
To read one line at a time from a file, a `for` loop can be used. This allows very large files to be read without being loaded entirely into memory.

```
fileObject = open( “file.txt”, “r” )
for line in fileObject:
    print line,
fileObject.close()
```

Note: A <newline> is not automatically removed from a line when it is read, it must be removed by the programmer using the .strip() string method.

This program attempts to open a file specified by a user and display its contents to the screen.

```
#!/usr/bin/env python

print “Enter the name of the file you wish to open: “
filename = raw_input(“> “)
fileObject = open( filename, “r” )
for line in fileObject:
    print line,
fileObject.close()
```

What is the script’s purpose? ____

____

Reading Lines of Data into a List

Reading all lines of data into a list may be useful if data needs to be referred to throughout a program. Use caution because if the resulting list is large, the program consumes a lot of memory to hold its contents.

```
fileObject = open( “file.txt”, “r” )
fileLines = fileObject.readlines()
fileObject.close()
for line in fileLines:
    print line
```

This script attempts to open a file specified by the user and display its contents to the screen.

```
#!/usr/bin/env python

print “Enter the name of the file you wish to open: “
filename = raw_input(“> “)
fileObject = open( filename, “r” )
fileLines = fileObject.readlines()
fileObject.close()
for line in fileLines:
    print line.strip(‘\n’)
```

Complete Exercise 9-14 in Student Workbook

Password Cracking Using a Dictionary Attack (Tool #2)

Complete Exercise 9-15 in Student Workbook

Using Tools #1 and #2 in a Network Environment

Objectives

Day 5

At the end of this training session, students will:

- Discuss network protocol creation.
- Text manipulation and regular expressions.
- Create Procedural programs.
- Identify program relationships (Client/Server).

Exercises

This training session includes the following exercises:

- Exercise 9-16, Python: For Loop and Range Commands
- Exercise 9-17, Python: Functions and Modules
- Exercise 9-18, Python: Regular Expressions and Exception Handling
- Exercise 9-19, Python: Server Service Communications Using Ports
- Exercise 9-20, Python: Using Server Service Communications in a Network Environment

JCAC-eTC

Complete self-study activities and assignments in the online training environment.

5.9 range() Function

At times, it is necessary to iterate through a range of numbers. Python makes this easy by using *range()*. The *range()* function uses integers to indicate where to start, stop, and by how much it should change to create a list of integers.

Table 31. Range function arguments.

Arguments	Description
range(Stop)	Generates a list of every number between 0 and up to, but not including, the value specified by *Stop*. *Start* is defaulted to a value of 0 and *step* is defaulted to a value of 1.
range(Start, Stop)	Generates a list of every number between the value specified by *Start* and up to, but not including, the value specified by *Stop*. *Step* is defaulted to a value of 1.
range(Start, Stop, Step)	Generates a list of every number between the value specified by *Start* and up to, but not including, the value specified by *Stop*, incrementing by the value specified by *Step*. (*Step* can be a negative value)

If the numerical range is arithmetically invalid, *range()* creates an empty list, rather than raise an exception.

The following program uses *range()* to display every port number from portList List.

```
#!/usr/bin/env python

portList = [20, 21, 25, 80, 143, 443]
for index in range(len(portList)):
   print “Working with port number:”, portList[index]
print “\nEnd of program”
```

Output:

```
Working with port number: 20
Working with port number: 21
Working with port number: 25
Working with port number: 80
Working with port number: 143
Working with port number: 443

End of program
```

Complete Exercise 9-16 in Student Workbook

Python: For Loop and Range Commands

5.10 Functions

A function is a group of instructions (code) that, when executed, performs a task. Reducing code into functions allows the individual pieces to be tested and perfected. Once the programmer is satisfied the function operates as planned, it can be used repeatedly. When an issue is found, it is easier to track down the specific function and make the correction. Once corrected, all other code that uses the function is correct as well.

In Python, functions are defined with the following structure:

```
def function_name( [parameters[,parameters…]] ):
    <function_body>
    return
```

Table 32 describes each part of a function.

Table 32. Parts of a function.

Keyword	Description
def	Informs Python that the following block of code is a function definition.
function_name	A unique name by which a function is known. Following the same rules as variable naming (i.e., can contain letters, numbers, and underscores only, but may not begin with a number).
parameters	Data passed to the function. If a parameter contains an equal sign, the right-side is the default value in case no argument was passed. Each parameter must have a unique name.
function_body	The code to execute when the function is called.
return	Optional. Returns a value from the function to where it was called. A function without a return instruction automatically returns after all logically sequential instructions have been executed.

The program below defines a function to inform a user whether IP addresses from a list are valid or not. Lines 4 and 16 are function declarations and Lines 26 and 36 show what are known as function calls. Line 26 first calls the function and the function returns a value which is then assigned to the **`isValid`** variable.

```
1     #!/usr/bin/env python
2
3     # Check each octet to make sure it is valid
4     def CheckOctets(octList):
5         # for each octet in octList
6         for index in range(len(octList)):
7             if int(octList[index]) not in range(256):
8                 # Octet is not valid
9                 return False
10
11        # If this code is executing it is because
12        # all octets did not fail in line 7, therefore
13        # this means all octets are valid
14        return True
15
16    def main():
17        # Build list of IP Addresses
18        IPList = ['192.168.0.10', '192.168.256.15', '192.168.0.15']
19
20        # for each IP Address do the following
21        for IPAddr in IPList:
22            # Build list of octets from IPAddr
23            octList = IPAddr.split(".")
24            # Send list of octets to be checked
25            # function returns True or False
26            isValid = CheckOctets(octList)
27            # Check value of isValid, execute accordingly
28            if isValid == True:
29                print IPAddr, ": All octets are valid"
30            else:
31                print IPAddr, ": 1 or more octets are not valid"
32
33    # Python program starts here
34    main()
```

Output:

```
192.168.0.10 : All octets are valid
192.168.256.15 : 1 or more octets are not valid
192.168.0.15 : All octets are valid
```

5.10.1 Local Variable Scope

In Python, local variables are only accessible and modifiable by the function in which they are created. This is because no other function knows the variable's memory location. In the following example, there are two variables named *x* that do not affect each other because each variable is unique within its scope.

```
#!/usr/bin/env python
def localScopeTest():
    x = 10
    print "The value of x in localScopeTest() is", x
def main():
    x = 5
    print "The value of x in main() is", x
    localScopeTest()
    print "The value of x in main() is", x
main()
```

Output with input shown in bold:

```
The value of x in main() is 5
The value of x in localScopeTest() is 10
The value of x in main() is 5
```

5.10.2 Global Variable Scope

If a global variable is declared, Python can read it just fine. However, making a change to a global variable requires the keyword **`global`** before the variable, otherwise a new local variable is created.

```
#!/usr/bin/env python
x = 5   # This is a global variable
def globalScopeTest():
    global x  # Why do we need this line?
    x = 10
def main():
    global x
    print "The value of x in main() is", x
    x = 2
    print "The value of x in main() is", x
    globalScopeTest()
    print "The value of x in main() is", x
main()
```

Output with input shown in bold:

```
The value of x in main() is 5
The value of x in main() is 2
The value of x in main() is 10
```

5.11 Modularity

A Python module is a file that contains one or more functions and/or classes that may be reused to save time writing and debugging new source code. Dozens of first-party modules are installed automatically with the Python interpreter. Third-party modules are countless, can be found online by Python developers around the world, and can be used freely.

Python modules are similar to C/C++ header files, which are used with the include directive and the header file (e.g., **`#include <iostream>`**). For example, the following line of code imports all the functions that simplify network programming.

```
import socket
```

Python allows programmers to import someone else's code into their own program as a module to solve a task. This saves time when it comes to writing and debugging new code since the solution is already written. For example, to develop the 2nd Tool to be used during the Practical Exam include the import the Practical Python Library called "`mod9Prac`".

The structure of a module import:

```
import module_name
import mod9Prac
```

Note: For a list of installed modules, type the following in the Python Interpreter.

```
>>> help()
help> modules
```

See Information Sheet 9-4 in Student Workbook

Python Modules

See Information Sheet 9-5 in Student Workbook

Mod9Prac Python Library

Complete Exercise 9-17 in Student Workbook

Python: Functions and Modules

5.12 Using Regular Expressions

To use regular expressions, the following must be included in the source code:

```
import re
```

When constructing a regular expression string, it needs to begin with the letter ‘r’. This informs Python to not escape characters that have a backslash preceding them. This allows the regular expression engine to process them as part of the regular expression instead.

```
expression = r”\w+”
```

5.12.1 Flags

A flag modifies the way a regular expression examines data. Use the bitwise OR (|) to allow more than one flag at a time on an expression.

Table 33. Regular expression flags

Flags	Shortcut	Description
re.DOTALL	re.S	Make the `.` (match any character), include <newline>
re.IGNORECASE	re.I	Allow case insensitive searches on string
re.MULTILINE	re.M	Allows the expression to search for strings which span multiple lines

5.12.2 Search() Method

The *search()* method looks through the entire string for the pattern specified. If the pattern is found, a match object is returned, otherwise no value is returned.

```
searchObject = re.search( r”expression”, “string”, flags )
```

Each search object uses several methods to extract more information from the data that was matched. The most popular is the *group()* method, which isolates just that segment of the string.

The following program prompts a user to enter name and phone number. It then tries to extract the phone number from the string.

```
#!/usr/bin/env python
import re

def main():
    print “Enter your name and phone number then press <enter>.”
    data = raw_input( “> “ )
    matchObject = re.search( r”(\d{3}-\d{3}-\d{4})”, data )
    if matchObject:
        print “The phone number is”, matchObject.group(1)
    else:
        print “Could not extract a phone number.”

# Code started executing here
main()
```

Output with input shown in bold:

```
Enter your name and phone number then press <enter>.
> Susie Queue 555-867-5309
The phone number is 555-867-5309
Enter your name and phone number then press <enter>.
> Bill Jones 123.456.789
Could not extract a phone number.
```

5.12.3 findall() Method

The *findall()* method returns a list of matches to all non-overlapping values in a specified string. This is useful when there may be more than one instance of data to extract.

```
re.findall( r”expression”, “string”, flags )
```

Figure 3 shows the contents of a file, *configuration.txt*, containing a computer’s network configuration. The program below opens the pictured file and finds any series of numbers matching the format of an IPv4 address.

```
eth0      Link encap:Ethernet  HWaddr 00:0F:20:CF:8B:42
          inet addr:192.168.0.10  Bcast:192.168.0.63  Mask:255.255.255.192
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2472694671 errors:1 dropped:0 overruns:0 frame:0
          TX packets:44641779 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1761467179 (1679.7 Mb)  TX bytes:2870928587 (2737.9 Mb)
          Interrupt:28
```

Figure 3. Contents of *configuration.txt*

```
#!/usr/bin/env python
import re

def main():
    fileData = ""
    configFile = open( “configuration.txt”, “r” )
    for line in configFile:
        fileData += line
    configFile.close()

    expression = r’\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b’
    matchList = re.findall( expression, fileData, re.M )
    for ip in matchList:
        print ip

main()
```

Output:

```
192.168.0.10
192.168.0.63
255.255.255.192
```

5.13 Exception Handling

An exception occurs when a resource or event causes a major error within a program. Programmers should handle exceptions using the OS provided option whenever possible. In Python, the `try-except` block is provided as a built-in technique to handle errors.

`try-except`

A block of code defined as the body of a **`try`** statement executes. If an exception occurs in the program, the **`except`** clause of the statement immediately executes, allowing the programmer to handle the error before the OS or Interpreter. This has the effect of ignoring any remaining code in the **`try`** block, and mitigating the crash proactively. Muletiple

Typical **`try-except`** structure.

```
try:
    Code block for a successful operation
except ExceptionType:
    Code block for an unsuccessful operation
```

Table 34. Common exception types.

Exception Type	Triggering Situation
ArithmeticError	An illegal arithmetic operation, such as division by zero or a math result too large for the containing data type, has occurred.
IOError	When an input or output operation fails; opening a file that does not exist or when network socket operations fail.
IndexError	The subscripted index specified is out of range.
TypeError	An operation applied to an object of an inappropriate type.
ValueError	An operation or function received the correct data type, but the value given was inappropriate; trying to convert a string to a number. If the string contains anything other than numbers, this exception is raised.

```
>>> print 9/0
ZeroDivisionError: integer division or modulo by zero
 # ZeroDivisionError is part of the base ArithmeticError class
>>> open("fakeFile.txt", "r")
IOError: [Errno 2] No such file or directory: 'testFile.txt'

>>> str1 = "Hello"
>>> print str1[5]
IndexError: string index out of range

>>> print str1 + 50
TypeError: cannot concatenate 'str' and 'int' objects

>>> int(str1)
ValueError: invalid literal for int() with base 10: 'Hello'
```

Example using **`try-except`** block:

```
try:
   portNum = int(raw_input("Enter a Port Number: "))
   if portNum in [20, 21, 22, 143, 443]:
      print "Valid Port Number: ", portNum
   else:
      print “Port number “, portnumber, “ not found.”
except ValueError:
   print "You must enter a whole number or integer value"
```

Output Example #1:

```
Enter a Port Number: 143
Valid Port Number: 143
```

Output Example #2

```
Enter a Port Number: ABC
You must enter a whole number or integer value
```

Output: Example #3

```
Enter a Port Number: 25
Port Number 25 not found
```

A script can throw multiple exceptions during execution, and Python allows a programmer to account for all potential exceptions by using multiple except clauses. The program below demonstrates the use of multiple exceptions in a **`try-except`** block:

```
#!/usr/bin/env python

try:
    number1 = float(raw_input(“Enter a number: \n”))
    number2 = float(raw_input(“Enter another number: \n”))
    number3 = number1 / number2
    print number1, “divided by”, number2, “is”, number3
except ValueError:
    print “One of the values entered is not a number!”
except ArithmeticError:
    print “Illegal arithmetic operation!”
```

Output with input shown in bold:

```
Enter a number:
10
Enter another number:
5
10.0 divided by 5.0 is 2.0
```

Complete Exercise 9-18 in Student Workbook

Python: Regular Expressions and Exception Handling

6 Network Programming

Network programming is required in programs that access a network resource or connect to another computer. Using well-established protocols, computer programs can reliably communicate across vast distances. Communication is done from a client to a server or vice versa using sockets. Sockets are objects that allow software to communicate and send data across a network connection via access to a computer's NIC. Sockets require an active port and valid IP address to function.

6.1 Socket

The creation of a socket allows a program to send/receive data across a network. Similar to how a file handle provides access to a hard drive, a socket provides direct access to a network card. By default, all Python sockets are TCP connections. Raw sockets can be configured for UDP, ICMP, and even IPv6!

6.2 Socket Module

Figure 4 shows the typical flow of events for a connection-oriented socket session, and includes the sequence of issued APIs.

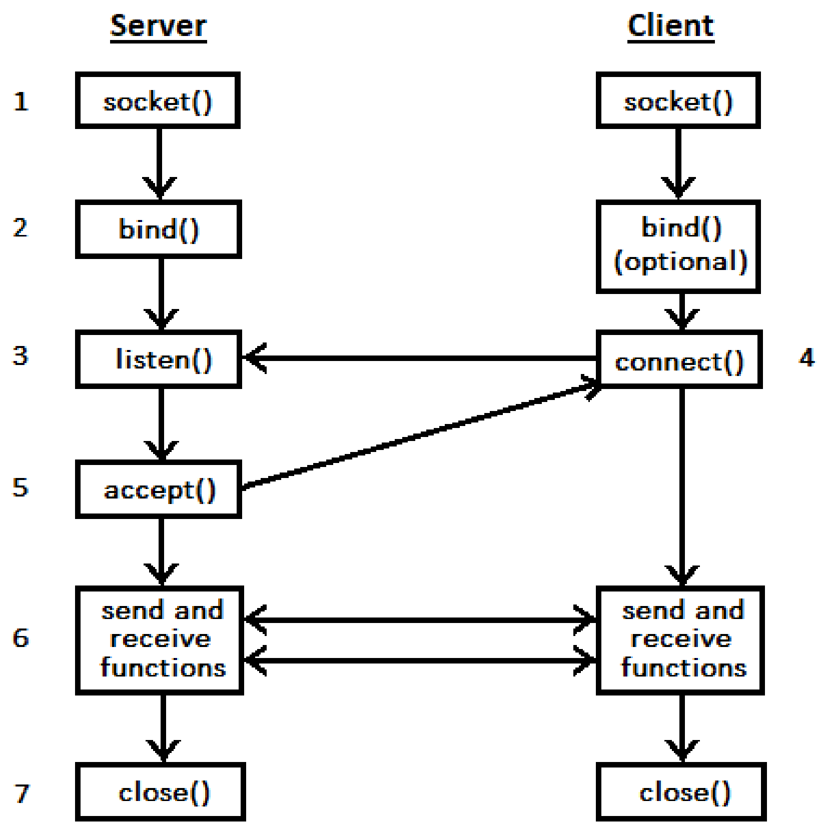


Figure 4. Server-client communication.

Note: Between steps 5 and 6, a connection socket is established and the actions are on that object.

Step 1: *socket()* method creates an endpoint for communications and returns a socket object representing the endpoint.

Step 2: Application uses socket object to bind a unique name to the socket. Servers must bind a name and port number to be accessible from the network.

Step 3: *listen()* method indicates a willingness to accept client connection requests. When a *listen()* method is issued for a socket, that socket cannot actively initiate connection requests. The *listen()* method is issued after a socket is allocated with a *socket()* method and the *bind()* method binds a name to the socket. A *listen()* method must be issued before an *accept()* method is issued.

Step 4: Client application uses a *connect()* method on a stream socket to establish a connection to the server.

Step 5: Server application uses the *accept()* method to accept a client connection request. Server must issue *bind()* and *listen()* methods successfully before issuing an *accept()* method.

Step 6: When a connection is established between stream sockets (client and server), any of the data transfer methods may be used. Clients and servers have many data transfer methods from which to choose, to include *send()* and *recv()*.

Step 7: Server or client stops operations by issuing a *close()* method to release any system resources acquired by the socket.

Table 35. Socket object methods.

Methods	Description
socket.socket()	Creates socket object used for endpoint communication.
<socket>.gethostname()	Returns a string containing the local computer name.
<socket>.bind()	Attempts to bind the socket to a physical interface.
<socket>.listen()	Listen for incoming connections.
<socket>.connect()	Initiates connection from the client.
<socket>.accept()	Accepts an incoming connection, returns a new object to transmit data and the address of the requesting socket
<socket>.close()	Closes the socket.
<socket>.setsockopt()	Specifies optional hardware/socket configuration.

Table 36. Connection object methods.

Methods	Description
<socket>.recv(N)	Reads string data from the socket, returns nothing if client closed the connection. N specifies number of bytes to read at one time, required.
<socket>.send(s)	Sends string data to the client.
<socket>.close()	Closes the connection.

6.3 Echo Server

An echo server replies to a client with the same message it received and is typically used to allow the client and server to successfully communicate.

Below is the source code for an echo server:

```
#!/usr/bin/python
import socket

def main():
   # Determine the computers host name
   host = socket.gethostname()

   # Indicates the port that the server will listen on
   port = 1337

   # Create a socket object and bind to hardware
   try:
      sObj = socket.socket()
      # Eliminate [Error 98].  Allows the socket to be re-useable
      sObj.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
      sObj.bind( (host, port) )
   # If socket object failed to create
   except IOError:
      sObj.close()
      print “Socket creation failed.”
      exit( 1 )

   # Specify maximum number of queued connections
   sObj.listen( 1 )

   # Infinitely wait for new connections
   while True:
      print “Server waiting for client on ” + host + “:” + str(port)
      conn, addr = sObj.accept()
      print 'Connection from', addr
      # Loop until the client disconnects
      try:
         while True:
            # Wait until data receive from the client
            data = conn.recv(1024)
            # Client disconnected
            if not data:
               # Close the client connection
               conn.close()
               # Leave the loop
               break
            # Send the data received back to the client
            conn.send(data)
      # Client disconnected unexpectedly
      except socket.error, msg:
         print '--Terminating Server--'
         print msg

   # Close the socket
   sObj.close()

# Code starts executing here
main()
```

Complete Exercise 9-19 in Student Workbook

Python: Server Service Communications Using Ports

Complete Exercise 9-20 in Student Workbook

Python: Using Server Service Communications in a Network Environment

Objectives

Day 6

At the end of this training session, students will:

- Use PowerShell to parse information from a data set.
- Read and interpret simple PowerShell scripts.
- Traverse, understand and debug code using PowerShell scripting language.
- Recall Windows scripting commands.
- Write, modify and maintain simple PowerShell scripts to perform various tasks.
- Recognize Encryption / Decryption of text.

Exercises

This training session includes the following exercises:

- Exercise 9-21, PowerShell Scripting: Cmdlets and Pipelines
- Exercise 9-22, PowerShell Scripting: Variable Declarations and Output
- Exercise 9-23, PowerShell Scripting: Decisions and Looping Structures
- Exercise 9-24, Decryption of Encrypted Text Using Caesar Cipher (Tool #3)

JCAC-eTC

Complete self-study activities and assignments in the online training environment.

7 Windows PowerShell

Windows PowerShell is Microsoft's task automation and configuration management framework consisting of a command-line shell with a built-in scripting language. PowerShell allows administration of local and remote Windows OS systems using small programs called cmdlets to simplify configuration and administration of both local and remote Windows systems.

7.1 Commands within PowerShell

Cmdlets are PowerShell specific commands. A cmdlet consists of a verb-noun pair separated by a hyphen. The verb specifies what action to carry out and the noun states the object on which to perform the action.

Table 37. Common PowerShell cmdlets

cmdlet	Description
`Get-Command`	Displays all commands that are installed on the computer, including cmdlets, aliases, functions, filters, scripts, and applications
`Get-Help`	Displays information about PowerShell concepts and commands, including cmdlets, functions, aliases, and scripts.
`Get-Member`	Displays the members, properties, and methods of objects.
`Get-Alias`	Displays the aliases available in the current session.
`Get-Service`	Displays objects that represent the services on a computer, including running and stopped services.
`Get-Process`	Displays a list of all active processes running on the local computer.
`Get-Childitem`	Displays the items in one or more specified locations.
`Get-Content`	Displays the content of the item at the location specified by the path, such as the text in a file or the content of a function.
`Test-Connection`	Sends pings to one or more remote computers and returns the echo response replies.
`Test-Path`	Determines whether all elements of a path exist.
`Read-Host`	Reads a line of input from the console.
`Write-Host`	Sends a specified object to the host/screen.
`Write-Output`	Sends a specified object down the pipeline to the next command.
`Where-Object`	Selects objects that have particular property values from the collection of objects that are passed to it
`Sort-Object`	Sorts objects in ascending or descending order based on object property values

The example below shows usage of PowerShell cmdlets.

```
Get-Service
Get-Help Get-Content
Get-Process | Get-Member
Get-ChildItem -Path C:\Windows\System32
```

7.2 Build a Pipeline of Cmdlets

Cmdlets are connected with pipes (|) to form more complex commands. A common order for processing cmdlets is **Command | Filter | Sort**. This order takes the output from the **Command**, and then **Filters** it to reduce the number of items to be processed, and finally takes that output and **Sorts** the resulting filtered data so that it can be displayed to the monitor or saved to a file.

7.2.1 Filter

In the event that a cmdlet results in a large amount of output, an optional filter is used to reduce the output to more specific results. A user can pipe the output of the first command to a second cmdlet named **`Where-Object`**.

```
Where-Object { $_.<property> <comparison> <value> }
```

`Where-Object` comparisons can be very complex and made of either Boolean or regular expression statements to filter on specific data.

Table 38. Pipeline syntax.

Component	Name	Description
`{ }`	Where Clause	The search expression is written between a set of curly braces.
`$_`	Default Object	Automatically references each object passed to the Where-Object cmdlet via the pipeline. Being an object itself, individual properties can be referenced to build the filter by using the dot $_.<property>
`<comparison>`	Comparison	Refers to an argument or operator to perform a Boolean or regular expression match.
`<value>`	Value	Refers to any value compared to the default object.

Table 39. Comparison and logical operators.

Comparison Operators	
-eq	Equivalent to
-ne	Not Equivalent to
-gt	Greater Than
-ge	Greater than or equivalent to
-lt	Less than
-le	Less than or equivalent to
-in	Is in the container
-notin	Is not in the container
-match	Regular Expression

Logical Operators	
-and	Both conditions must be true
-or	One or the other condition must be true
-xor	One or the other but not both conditions must be true
-not	Invert the Boolean value

The services returned previously can be filtered to allow a user to display only those services that are currently running.

```
Get-Service | Where-Object { $_.Status -eq "Running" }
```

Output:

```
Status   Name               DisplayName
------   ----               -----------
Running  AdobeARMservice    Adobe Acrobat Update Service
Running  Altiris Deploym...  Altiris Deployment Agent
Running  AppIDSvc           Application Identity
Running  Appinfo            Application Information
Running  AudioEndpointBu...  Windows Audio Endpoint Builder
Running  Audiosrv           Windows Audio
.  .  .  .  .
```

In the next example, the System32 directory is filtered to display file objects with executable file extensions that start with "win."

```
Get-ChildItem -Path C:\Windows\System32 |
Where-Object { $_.Extension -eq ".exe" -and $_.Name -match "^win" }
```

Output:

```
Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         9/29/2017   8:41 AM        359584 wininit.exe
-a----         9/13/2019   1:21 AM       1212224 winload.exe
-a----          9/4/2019  12:15 AM        716288 winlogon.exe
-a----         9/13/2019   1:21 AM        925064 winresume.exe
-a----         9/29/2017   8:42 AM         48640 winrs.exe
-a----         9/29/2017   8:42 AM         28160 winrshost.exe
-a----         9/29/2017   8:42 AM       2841088 WinSAT.exe
-a----         9/29/2017   8:42 AM         58880 winver.exe
```

Complete Exercise 9-21 in Student Workbook

PowerShell Scripting: Cmdlets and Pipelines

7.2.2 Sort

Sorting allows for ordering a set of objects in an acceptable format after the filtering of objects is complete. Sorting creates a more manageable dataset.

The **`Sort-Object`** cmdlet takes a piped list of objects for input and, at minimum, an argument that defines the property on which to sort. Sorting provides information in ascending order by default, but can organize it in descending order, by case, by uniqueness, or by using multiple comma separated properties.

```
Sort-Object <property[,property]> [-CaseSensitive] [-Descending] [-Unique]
```

The following example displays processes which have over 50 megabytes of memory being used, and sorts the processes in descending order with the largest memory usage first.

```
Get-Process | Where-Object { $_.WorkingSet -gt 50000000 } |
Sort-Object WorkingSet -Descending
```

Output:

```
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    705      77   209856     270840      81.25  11644   1 firefox
   1462     102   178380     249408     115.88  10356   1 firefox
   1285      87   241784     233248     375.55   3908   1 WINWORD
   1273      71   191036     233168      26.94  10480   1 powershell_ise
   1103      71    79452     142024       1.47  10104   1 SearchUI
   3627      90    71404     132344      32.69   7404   1 explorer
    648      52    72576     125444      10.44  11048   1 firefox
    209      17    96036     101592              2512   0 svchost
    498      40    30116      66584       2.36  10992   1 firefox
.  .  .  .  .
```

The next example displays files in the Windows directory filtered to log files and sorted by the time the file was last written to in ascending order, with the least recent write time being first.

```
Get-ChildItem -Path C:\Windows |
Where-Object { $_.Extension -eq ".log" } |
Sort-Object LastWriteTime
```

Output:

```
Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        2/20/2018   1:30 PM           1380 lsasetup.log
-a----        2/20/2018   4:12 PM              0 setuperr.log
-a----         2/6/2019   1:30 PM          24736 DPINST.LOG
-a----        2/13/2019   3:42 PM          27255 DtcInstall.log
-a----       10/16/2019  11:32 PM          59628 PFRO.log
-a----       10/18/2019   3:21 PM          11877 setupact.log
-a----       10/24/2019   4:42 PM            276 WindowsUpdate.log
```

7.3 Create a Script

A PowerShell script is a simple text file with an extension of *.ps1* and contains the commands the user wishes to run in sequential order. It is prudent to test each command individually and then move them into the script once the correct output is achieved. The dot backslash (.\ or ./) placed in front of the script name allows the script to run from a relative directory. The full file path could also be used instead of the dot back slash.

Example:

```
PS /home/student> pwsh .\myscript.ps1
PS /home/student> pwsh mystript.ps1
```

7.4 Write-Host Command

The **`Write-Host`** command sends arguments to STDOUT and automatically adds a terminating <newline> character. **`Write-Output`** is similar, but can be redirected; **`Write-Host`** only displays to the monitor.

```
Write-Host This is a test
Write-Host “This is a test, too”
Write-Host ‘This is another line as well.’
```

Output:

```
This is a test
This is a test, too
This is another line as well.
```

If a script requires that **`Write-Host`** not go to the next line on the screen after displaying the text, include the **`–nonewline`** argument as shown in the below code**:**

```
Write-Host This is a test
Write-Host “This is the beginning and ” -nonewline
Write-Host ‘this is the ending of the same line.’
```

Output:

```
This is a test
This is the beginning and this is the ending of the same line.
```

The **escape character** is the grave accent character (`).

```
Write-Host “This is a double quote `””
Write-Host “This is a single quote ’“
Write-Host “This is a long line `n”
Write-Host “which ends here.”
```

Output:

```
This is a double quote “
This is a single quote ‘
This is a long line

which ends here.
```

Table 40. Escape characters.

Character	Prints
`n	<newline>
``	Grave Accent Character
`t	Tab Character
`$	Display $ Character
`”	Display the double quote character

Note: The escape character does not work between single quotes since single quotes give literally what is between the quotes. A greater explanation of quotes appears later in the module.

7.5 Variables and Arrays

In PowerShell, variable names always start with a dollar sign (`$`). The dollar sign tells PowerShell that it should retrieve or set a value. All variables in PowerShell are mapped to underlying classes. This means that variables are objects and have methods that are used to manipulate their values.

Variables are declared by writing the name of the variable followed by the assignment operator (`=`) and the value to store in the variable. Spacing in PowerShell variable declarations does not affect the creation of the variable.

Example of variable use in PowerShell scripts:

```
$PhysicalAddr = “e0-1a-ea-02-20-0f”
Write-Host $PhysicalAddr.toupper()
Write-Host ${PhysicalAddr}.tolower() # Similar to UNIX/Bash syntax
Write-Host “Your MAC is $($PhysicalAddr.ToUpper())”
```

Output:

```
E0-1A-EA-02-20-0F
e0-1a-ea-02-20-0f
Your MAC is E0-1A-EA-02-20-0F
```

```
$IPAddress = “192.168.0.”       # Assign 192.168.0.  to IPAddress
$count = 1                      # Assign integer 1 to count
$filename = “script1.ps1“       # Assigns ‘script1.ps1’ to filename
$my_dir = “C:\Windows\System32” # Assigns directory path to my_dir
$oct4 = 15                      # Assign the value 15 to oct4
$IPAddress = $IPAddress + $oct4 # Concatenate oct4 to IPAddress
```

```
$my_dir = “C:\Windows\System32”
$IPAddress = “192.168.0.15”
$count = 1
ping -n $count $IPAddress            # What does this command do?
$IPAddress2 = $IPAddress
$count = $null                       # Assign null to count
Write-Host IPAddress2 is: $IPAddress2
Get-ChildItem -Path $my_dir          # What does this command do?
```

PowerShell determines the data type of each variable based on the value it was assigned. In line 1 in the example below, since the value of **`$IPAddress`** is enclosed in a set of quotation marks, PowerShell determines that data type to be a string. Use the **`GetType()`** variables method to identify the data type for a variable.

```
1   $IPAddress = “192.168.0.”       # Assign 192.168.0.  to IPAddress
2   $IPAddress.GetType()
3   $Number = 10
4   $Number.GetType()
5   $Pumpkin = 3.14
6   $Pumpkin.GetType()
7   $Knight = $True
8   $Knight.GetType()
```

Output:

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
True     True     Int32                                    System.ValueType
True     True     Double                                   System.ValueType
True     True     Boolean                                  System.ValueType
```

Note: PowerShell calls data types that we understand to be floats a double.

Typecasting, however, can force a variable to assume another data type. Line 3 and line 8 use typecasting to convert the data type of the respective variables.

```
1   $count = 45
2   Write-Host $count.GetType()
3   $count = [string]$count
4   Write-Host $count.GetType()
5
6   $year = "1775"
7   Write-Host $year.GetType()
8   $year = [int]$year
9   Write-Host $year.GetType()
```

Output:

```
System.Int32
System.String
System.String
System.Int32
```

Arrays in PowerShell are zero indexed. The index is always an integer and can be an expression. Remember that each element can store data that can be a different data type (e.g., char, int, string, float, etc.).

```
# Create an empty array called fileList
$fList = @()  # @ is only needed for empty arrays
# Create array of files from the C:\Windows\System32 folder
$fList = Get-ChildItem -Path $my_dir -File
# Two additional ways to create an array of IP Addresses
$IPList = (“192.168.122.1”, “192.168.122.2”)
$IPList = “192.168.122.1”, “192.168.122.2”, “192.168.122.3”
```

To access the value in a particular element of the array, use the format as shown below.

```
$Items = (“10.10.120.1”, 80, “115-129-WS08”)
Write-Host $Items[2]
```

Output:

```
115-129-WS08
```

Additionally, two methods can be very useful:

`$Items`	Returns all the values in the array as a single value.
`$Items[0]`	The value within the square brackets is the index number used to return the value in that element.
`$Items.length`	Used to return the number of elements in the array.

```
$IPAddr = (“192.168.122.1”,”192.168.122.20”,”192.168.122.23”)
Write-Host $IPAddr
Write-Host $IPAddr[2]
Write-Host $IPAddr.length     # How many elements in the array?
Write-Host $IPAddr[1].length  # How long is value in element two?
```

Output:

```
192.168.122.1 192.168.122.20 192.168.122.23
192.168.122.23
3
14
```

7.6 Quotes

Quotes allow the writing of string values with spaces, internal single and double quotes, and internal $ symbols. Quoting conventions are:

Double Quotes Substitutes variable and escaped characters with their actual values.

```
$Subnet = ”192.168.122”
$Oct4 = ”53”
Write-Host IP Address is: “$Subnet.$Oct4”
```

Output:

```
IP Address is: 192.168.122.53
```

Single Quotes Makes everything literal so the values of variables and escape characters are not substituted.

```
$Subnet = ”192.168.122”
$Oct4 = ”53”
Write-Host IP Address is: ‘$Subnet.$Oct4’
```

Output:

```
IP Address is: $Subnet.$Oct4
```

Complete Exercise 9-22 in Student Workbook

PowerShell Scripting: Variable Declarations and Output

7.7 Read-Host Command

The script reads a line from STDIN with the **`Read-Host`** command. The **`Read-Host`** command assigns the contents of the input to the variable listed as a string data type.

Examples:

```
Write-Host “Enter an IP Address ” -nonewline
$IPAddress = Read-Host
Write-Host “Enter a number ” -nonewline
$count = [int](Read-Host)
```

```
$IPAddress = Read-Host -Prompt "Enter an IP Address"
$count = [int](Read-Host -Prompt “Enter a number”)
```

The second example shows how one line of code can be used to effectively accomplish the same as two lines of code used in the first example. The **`-Prompt`** parameter is optional and specifies the text of the prompt. PowerShell also appends a semicolon (:) to the prompt text.

7.8 PowerShell Scripting Flow-Control Structures

In PowerShell scripting, the primary flow-control structures include the **`if`** statement and the **`while`** and **`foreach`** loops.

Flow control uses Boolean conditions (True or False conditions) to determine whether to execute a block of code. The Boolean condition values are created by using comparison operators in conjunction with values. Compound conditions can be created by joining two condition statements together with a logical operator. Refer to Table 38 and Table 39 for PowerShell comparison and logical operators.

7.8.1 Branching Statements

Branching statements are used to choose which set of statements to execute using Boolean logic and conditional branches, and use the same syntax as C++. A **`$True`** condition causes the associated code block to execute.

If Structure

- The **`if`** structure evaluates the expression and returns control based on this status.
- The open curly bracket (**`{`**) marks the beginning of the code block of the **`if, elseif,`** or **`else`** statements.
- The close curly bracket (**`}`**) marks the end of the **`if, elseif,`** or **`else`** statements.
- The **`elseif`** allows construction of a nested set of **`if-then-else`** structures.
- The Boolean value of TRUE is represented as **`$True`**
- The Boolean value of FALSE is represented as **`$False`**

```
if ( expression )
{
    Commands
    Commands
    Commands
    Commands
    Commands
}
```

```
if ( expression )
{
    Commands
    Commands
}
else
{
    Commands
    Commands
    Commands
}
```

```
if ( expression )
{
    Commands
    Commands
}
elseif ( expression )
{
    Commands
    Commands
}
else
{
    Commands
}
```

Branching Examples:

```
[int]$oct1 = Read-Host –Prompt “Enter the first octet”

if ( $oct1 -ge 0 -and $oct1 -le 255 )
{
    Write-Host “Octet is valid!”
}
else
{
    Write-Host “Octet is out of range!”
}
```

What is the script’s purpose? ____

____

```
$number = Read-Host -Prompt "Input a number"
if ($number -match "^-?\d+$")
{
    Write-Host "$number is a number!"
    [int]$number = $number
    if ($number -gt 0)
    {
        Write-Host "$number is positive."
    }
    elseif ($number -lt 0)
    {
        Write-Host "$number is negative."
    }
    else
    {
        Write-Host "$number is zero."
    }
}
else
{
    Write-Host "Invalid.  $number is not a number!"
}
```

Sample Output:

```
Input a number: 83
83 is a number!
83 is positive.
```

Sample Output:

```
Input a number: ten
Invalid.  ten is not a number!
```

7.8.2 Loop Structures

Loop structures make repetitive tasks easy. The focus of the next section is on **while** and **foreach** loops.

While Loop

The **while** loop executes a block of code repeatedly *if its conditional expression evaluates to a true value*. It also has the same syntax as in C/C++. This structure is used when the scriptwriter does not know how many times the loop needs to execute.

```
while ( expression )
{
    Commands
    Commands
    Commands
}
```

The body of the loop is defined between the open and closed curly braces. The commands executed within the **while** loop must change the test expression value or an infinite loop results.

```
$subnet = Read-Host -Prompt "Enter 1st three octets (###.###.###):"
[int32]$startOctet = Read-Host -Prompt "Enter a starting octet: "
[int32]$endOctet  = Read-Host -Prompt "Enter an ending octet: "

#Assume we entered valid data in variables
$oct4 = $startOctet
$IPList = @()
while ( $oct4 -le $endOctet )
{
   $IPAddr = "$subnet.$oct4"
   $result = Test-connection –Quiet –count 1 $IPAddr
   if ( $result –eq $True )
   {  # We have a valid IP Address append to IPList
      $IPList += $IPAddr
      Write-Host "Valid IP Address:" $IPAddr
   }
   $oct4 += 1
}
```

What is the script's purpose? ____

____

ForEach Loop

The **foreach** loop is an iterative **for** loop that iterated through each value in a set or container type object such as an array.

```
foreach($item in $set)
{
    Commands
    Commands
    Commands
}
```

```
$original = "abcdefghijklmnopqrstuvwxyz"
$original += $original.ToUpper()
foreach ( $letter in $original.ToCharArray() )
{
      Write-Host “Letter is: $letter”
}
```

Output:

```
Letter is: a
Letter is: b
...
Letter is: Z
```

PowerShell can also use the **ForEach-Object** cmdlet to perform operations against each item in a collection of input objects. The examples below use a pipeline to isolate a list of properties from a set of objects.

cmdlet | ForEach-Object { statement }

```
Get-ChildItem -Path C:\Windows | ForEach-Object {$_.Name}
```

Output:

```
addins
appcompat
apppatch
.  .  .
write.exe
```

```
Get-Process | Where-Object {$_.Name -match "^win"} | ForEach-Object
{$_.Name} | Sort-Object -unique
```

Output:

```
wininit
winlogon
WINWORD
```

Complete Exercise 9-23 in Student Workbook

PowerShell Scripting: Decisions and Looping Structures

8 Encryption/Decryption of Text

Cryptography is the analysis and practice of concealing information and securing sensitive data. Encryption and decryption of data supports the Information Assurance (IA) concepts of confidentiality, integrity, and availability. Cryptography is a means of manipulating or scrambling plaintext into cipher-text and transforming cipher text back into plaintext. Modern cryptography uses two keys—PUBLIC and PRIVATE—to encrypt and decrypt a message.

A cryptanalyst is someone who hacks, breaks, or decrypts a ciphered text. In this context, decrypt means to convert ciphered text to plaintext. Cryptanalysis is the art of breaking codes and ciphers.

Caesar-Shift Cipher

Caesar-shift cipher is one of the earliest methods of cryptography. This cipher hides the message from unauthorized readers by shifting the letters of a message by an agreed number. Upon receiving a message, the recipient shifts the letters back by the same number agreed upon earlier. A cryptanalyst can decrypt the message by observing the cipher method.

The Caesar-shift cipher is also one of the easiest ciphers to break. Since the shift must be a number between 1 and 25 (0 or 26 would result in an unchanged plaintext) an analyst can try each possibility and see which one results in readable text. Identifying a piece of the cipher text allows immediate identification of the key.

If this is not possible, a more systematic approach is to calculate the frequency distribution of the letters in the cipher text. This consists of counting how many times each letter appears in the encrypted text. Natural English text has a very distinct distribution that can be used to help crack codes as shown in Figure 5.

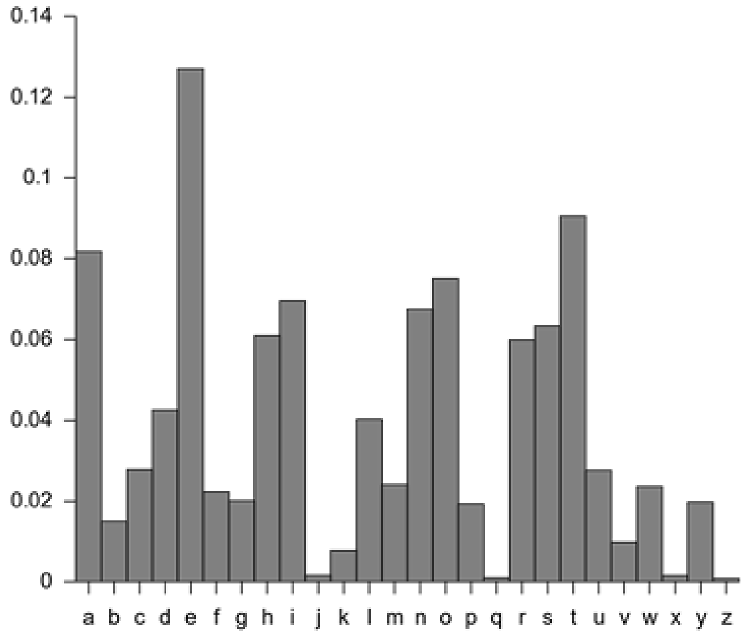


Figure 5. Distribution of letters in the English language.

This means that the letter “e” is the most common, and appears almost 13% of the time, whereas “z” appears far less than 1 percent of time. Application of the Caesar cipher does not change these letter frequencies; it merely shifts them (for a shift of one, the most frequent cipher text letter becomes “f”). A cryptanalyst must find the shift that causes the cipher text frequencies to match up closely with the natural English frequencies, and then decrypt the text using that shift. This is one of the methods used to break Caesar ciphers by hand.

Analyze and decode the following text: “pnrfne pvcure vf gur rnfvrfg gb qrpbqr”

What is the decoded text? ____

The following example uses bitwise XOR to encrypt a message input by the user. Line 8 changes the data type of each letter in the string to an 8-bit unsigned character. This is necessary to do use bitwise operations like bitwise XOR.

```
 1  try
 2  {
 3      $string = Read-Host -Prompt "Enter a string"
 4      [int]$key = Read-Host -Prompt "Enter the key"
 5      $cryptoString = ""
 6      foreach ($letter in $string.ToCharArray())
 7      {
 8          $cryptoletter = [byte] $letter -bxor $key
 9          $cryptoString += [char]$cryptoletter
10      }
11      Write-Host "The encryption is: $cryptoString"
12  }
13  catch
14  {
15      Write-Error "Invalid key value."
16  }
```

Output:

```
Enter a string: How neat is that?
Enter the key: 7
The encryption is: Ohp'ibfs'nt'sofs8

Enter a string: That's pretty neat.
Enter the key: 8
The encryption is: \`i|/{(xzm||q(fmi|&
```

Complete Exercise 9-24 in Student Workbook

Decryption of Encrypted Text Using Caesar Cipher (Tool #3)

Objectives

Day 7

At the end of this training session, students will:

- ❖ Understand basic web page HTML elements.
- ❖ Understand security risks associated with an HTML iframe.
- ❖ Integrate CGI Script with forms.

Exercises

This training session includes the following exercises:

- ❖ Exercise 9-25, Format an HTML Document

JCAC-eTC

Complete self-study activities and assignments in the online training environment.

9 HyperText Markup Language (HTML)

HTML is the language of the World Wide Web and is how most people access information on the Internet. The World Wide Web Consortium (W3C) standardized HTML in 1996 and all major browsers support the HTML standard. HTML is the most used markup language in the world for displaying web pages and other information in a web browser. A web browser reads HTML text-based documents and composes them into visible or audible web pages. A browser uses HTML elements to interpret the content of a page.

HTML elements form the building blocks for web pages by allowing images and objects to be embedded into documents. HTML allows the creation of interactive web pages and gives structure to a document. Headings, paragraphs, lists, links, quotes, and citations are some of the semantic structures of HTML. HTML allows actions or behaviors to perform within a document using scripting languages like Python or JavaScript.

HTML uses metadata elements to format documents and render information as intended. The elements work behind the scene and are not seen by the end-user. The hypertext in HTML represents its ability to connect relevant documents via links.

Table 41. Common HTML elements.

Element	Description
`<html>`	Represents the root (top-level element) of an HTML document. All other elements must be descendants of this element.
`<head>`	Contains metadata about the document (i.e., title, scripts, style sheets, etc.).
`<body>`	Represents the content of an HTML document.
`<title>`	Defines the document's title that is shown in a browser's title bar or a page's tab.
`<h1>,<h2>,<h3>` `<h4>,<h5>,<h6>`	Represent six levels of section headings. <h1> is the highest section level and <h6> is the lowest.
`<p>`	Represents a paragraph.
`<br>`	Produces a line break in text (carriage-return).
`<hr>`	Represents a thematic break between paragraph-level elements (e.g., a shift of topic within a section).
`<img>`	Embeds an image into the document.
`<u>,<i>,<b>`	Inline text for underline, italics, and bold.
`<font>`	Defines the font size, color and face for its content.
`<table>`	Represents tabular data (e.g., information presented in a two-dimensional table comprised of rows and columns of cells containing data).
`<tr>`	Defines a row of cells in a table.
`<td>`	Defines a cell of a table that contains data.

The text below is an example of an HTML document.

```html
<html>
   <head>
      <title>Mod 9's Fancy Web Page</title>
   </head>

   <body bgcolor="grey">
      Hello, World!
      <h1>Heading 1</h1>
      <h2>Heading 2</h2>
      <h3>Heading 3</h3>
      <h4>Heading 4</h4>
      <h5>Heading 5</h5>
      <h6>Heading 6</h6>
      <font size="1" color="purple">Purple 1</font>
      <font size="2" color="yellow">Yellow 2</font>
      <font size="3" color="fuchsia">Fuchsia 3</font>
      <font size="4" color="teal">Teal 4</font>
      <font size="5" color="Red">Red 5</font>
      <font size="6" color="mahogany">Mahogany 6</font>
      <font size="7" color="vermillion">Vermillion 7</font>
      <br> <b><i>This is a new sentence without a paragraph
      break, in bold italics.</i></b>
      <p> This is a new paragraph! </p>
      <p> <b>This is a new paragraph!</b> </p>
      <br><br>
      <a href="http://cnn.com">Click to display CNN Site</a>
      Send mail to: <a href="mailto:support@yourcompany.com">
      support@yourcompany.com</a>.
   </body>
</html>
```

9.1 Overview of a Web Page

A web page is a browser's rendition of a text document known as an HTML source file. HTML markup elements in a source file describe the contents of a web page. All HTML elements that contain content have both an opening and closing tag. The browser interprets the HTML source file and presents it to the user. HTML source files usually have the file extension *.htm* or *.html*.

Below is an example of a simple HTML document that displays the phrase "Hello World!" in a browser.

```
1  <html>
2      <head>
3          <title> Hello World web page </title>
4      </head>
5  
6      <body>
7          <h1>Hello World!</h1>
8      </body>
9  </html>
```

Line 1:	Opening tag, also called a start element; signifies the beginning of an HTML document. Its accompanying closing tag is on line 9. All HTML documents start with an <html> opening tag and end with a </html> closing tag.
Line 2:	<head> is the opening tag for an HTML header element. Known as the header, it is not rendered by the browser, but contains additional information about the document.
Line 3:	Document title element. <title> is the opening tag for the element and </title> is the closing tag. The document title contained between the tags is displayed at the top of the browser or on the browser tab if using tabbed browsing. If this tag is omitted, the web page URL displays in its place.
Line 4:	Closing tag for the document header element.
Line 6:	The opening tag of the document body element. All of the visible content of a web page is located in the body element of an HTML document.
Line 7:	<h1> is a text formatting tag that specifies the size of text and makes the text bold.
Line 8:	Closing tag for the document body element.
Line 9:	Closing tag, also called an ending tag; signifies the ending of an HTML document.

9.1.1 Document Formatting

Web page formatting is essential to organize the layout of data. The layout makes a web site readable and improves the overall appearance. For example, tables organize data into rows and columns.

<table>...</table>	Specifies a table. Tables contain table rows elements, which in turn contain table data elements.
<tr>...</tr>	Specifies a table row. Table rows should only contain table data elements. Text in a row aligns on both horizontal (align) and vertical (valign) using attributes: <tr align=”center” valign=”middle”>….</tr>
<td>...</td>	Specifies table data. This is where the information contained in the table goes. Each table data element is a table cell. The number of columns in a table is determined by the row with the most cells. Cells can span multiple columns using the attribute *colspan.* Text in a cell can be aligned both horizontally and vertically using attributes: <td align=”right” valign=”middle”>….</td>

Table example:

```
<table width="100%" border="0">
    <tr>
        <td align="left"> John Doe</td>
        <td align="right">123 Mockingbird Lane</td>
    </tr>
    <tr>
        <td colspan="2"><hr/></td>
    </tr>
</table>
```

9.1.2 iFrame and Security Risk

Web application security is always an important topic to discuss because web sites seem to be the first target of malicious hackers. Hackers use web sites to spread malware and compromised web sites for spamming and other purposes. Staying abreast of the latest web application vulnerabilities is one way to stay ahead of potential attackers.

An **`iframe`** element is part of HTML and is a technique used in web page development to embed files, such as documents, video, audio, and images in the same HTML document. A simple way to explain **`iframe`** is that **`iframe`** is a technique to display information from another web page within the same web page. The use of iframe can pose a security risk to a web page by making it vulnerable to cross-site attacks. For example, by using an iframe in a web page a malicious user could embed a form to collect a user's personal data, hijack a user's mouse clicks, or even hijack a user's keystrokes.

The below HTML element shows how to display another web site within a web site using the iframe element.

```
<iframe src=”http://www.companywebsite.com”></iframe>
```

In the below example, the width and height of the iframe is defined. However, the frame visibility is hidden, and therefore there is no physical presence of the company's web site. An attacker does not typically use this technique because the frame, although hidden, occupies an area defined by the width and height attributes.

```
<iframe src=’http://companywebsite.com/’ width=’500′ height=’600′
style=’visibility: hidden;’></iframe>
```

By setting the width and height attributes to a value of 1 in the following example, an attacker is able to not only embed a web site on a web page but also keep it from displaying and not take up any visible space when the web page is displayed in the web browser.

```
<iframe src=’http://companywebsite.com/’ width=’1′ height=’1′
style=’visibility: hidden;’></iframe>
```

Obfuscated iframe Injection Attacks

Generally, if a web site has been compromised by using an **`iframe`** injection attack, it is easy to find and locate the injection code because the code is easy to read. Alternately, if obfuscation (a way to hide the true content of the injected code, making it difficult to read) is used, then it is very difficult to detect and find the malicious injection code on a web site. The aim of this type of attack is to trick a user and then redirect to a third-party web page to further exploit the user.

Consider an example where a company’s compromised web site redirects to or displays another web page within a page to sell some products. The web site visitor trusts the web site because it looks the same as the one from which they usually purchase products.

To maintain the security of customers, the company’s web site administrator needs to protect the web site from attack. A simple way is to periodically review the index page for `iframe` and redirect code containing the URL of a third-party web site. However, sometimes a malicous attacker uses obfuscation tactics to hide code and evade detection.

Suppose there is code that looks like:

```
++++%23wp+/+GPL%0A%3CScript+Language%3D%27Javascript%27%3E%0A++++%3C%21--
%0A++++document.write%28unescape%28%273c696672616d65207372633d27687474703a
2f2f696e666f736563696e737469747574652e636f6d2f272077696474683d273127206865
696768743d273127207374796c653d277669736962696c6974793a2068696464656e3b273e
3c2f696672616d653e%27%29%29%3B%0A
++++//--%3E%0A++++%3C/Script%3E
```

The code seems to be normal, but in reality, it is the root cause of the problem. Decode it by using the java decoding function and the result is:

```
#wp / GPL
<Script Language='Javascript'>
    <!--
document.write(unescape('3c696672616d65207372633d27687474703a2f2f696e666f7
36563696e737469747574652e636f6d2f272077696474683d273127206865696768743d273
127207374796c653d277669736962696c6974793a2068696464656e3b273e3c2f696672616
d653e'));
    //-->
    </Script>
```

At this stage it still appears to be a legitimate piece of code because the attacker has obfuscated the malicious portion by using the terms “GPL” “wp” and “Java”. However, the numbers and letters appear to be hexadecimal.

```
3c696672616d65207372633d27687474703a2f2f696e666f736563696e737469747574652e
636f6d2f272077696474683d273127206865696768743d273127207374796c653d27766973
6962696c6974793a2068696464656e3b273e3c2f696672616d653e
```

Performing a hexadecimal decode results in the following, revealing a third-party URL:

```
<iframe src=’http://mycompanysite.com/’ width=’1′ height=’1′
style=’visibility: hidden;’></iframe>
```

This example exposes layers of obfuscation and the difficulty in finding and securing web sites against `iframe` injection attacks.

Complete Exercise 9-25 in Student Workbook

Format an HTML Document

9.2 Common Gateway Interface

The Common Gateway Interface (CGI) provides a means of delivering dynamic web content using executable files or scripts. CGI is supported by many different programming languages but is most commonly used by interpreted scripts such as Perl, PHP, and Python. Variables are typically passed using the HTTP **GET** or **POST** methods. To help maintain security, executable files are typically restricted to the *cgi-bin* directory of a web server.

9.2.1 Forms

A form on a web page allows a user to enter data to be processed either by server-side or client-side scripts. Forms are used by search engines, online stores, and web-enabled databases. They generally resemble paper-based forms and often include text fields, check boxes, and radio buttons.

Forms processed on the server have an *action* attribute that specifies the name of the script that processes the form. These forms also have a *method* attribute, either GET or POST, to specify how information is to be sent to the server.

<form>...</form>	Specifies a form element. Forms use input elements to hold user specific data. The input elements can be organized using a table. <form action="processMe.py" method="GET">….</form>
<input />	Specifies an input field. A type attribute must be used to indicate the type of input. Input types:

Text	A simple text box to allow input of a single line of text. An alternative, password, is used for security purposes. The characters typed in are invisible or replaced by symbols such as an asterisk (*).
Checkbox	A checkbox that can be selected.
Radio	A radio button, or a group of buttons in which only one can be selected.
File	A file select control for uploading a file.
Reset	A reset button that, when activated, tells the browser to restore the values to their initial (usually blank) values.
Submit	A button that tells the browser to take action on the form (typically, send it to a server).
Button	A button that can be associated with a client-side script using the *onclick* attribute.

<textarea>…</textarea> Specifies a multiline text input field. It has the attributes, rows, and columns that dictate the size of the text area. It is similar to the text input field except a text area allows for multiple rows of data to be shown and entered.

```
<textarea rows="2" cols="20">This is the default value of the textarea</textarea>
```

The form element in the following example specifies that the *processMe.php* script uses the POST method to send data to a server. Line 16 uses the input element to create a submit button. The input element includes the name attribute that is effectively a variable to be passed to the script as an argument. When the user presses the submit button, this is effectively a function call.

```
1   <form method="POST" action="processMe.php">
2       <table border="1">
3           <tr>
4               <td>Your name</td>
5               <td>
6                   <input type="text" name="name" size="20">
7               </td>
8           </tr>
9           <tr>
10              <td>Your E-mail address</td>
11              <td>
12                  <input type="text" name="email" size="25">
13              </td>
14          </tr>
15      </table>
16      <input type="submit" value="Submit" name="submitBtn">
17  </form>
```

9.2.2 CGI Scripts

CGI is used by specifying a form action to be associated with the click of a submit button. An HTML form is typically used to submit data to a CGI script and can be a separate HTML file or can be generated by the script itself. In a CGI script, STDOUT is redirected to the web page response. A Python script uses print statements to generate a web page.

To pass information to a server using a form, the following example attempts to send the *username* input field data to the *cgi_example.py* script on the local web server.

```
<html>
    <head>
        <title>This is HTML</title>
    </head>
    <body>
        <form action=”cgi-bin/cgi_example.py”>
            Type your name: <br/>
            <input type=”text” name=”username” />
            <br/>
            <input type=”submit” value=”Submit Data”/>
        </form>
    </body>
</html>
```

The previous HTML page can be produced with the following Python script. This file should be saved as *index.py* in the */var/www/cgi-bin* directory of a web server with execute permissions.

```
#!/usr/bin/python
import cgi

def main():
    print "Content-type: text/html \n\n"
    print ‘’‘
    <html>
        <head>
            <title>This is HTML</title>
        </head>
        <body>
            <form action=”cgi-bin/cgi_example.py”>
                Type your name: <br/>
                <input type=”text” name=”username” />
                <br/>
                <input type=”submit” value=”Submit Data”/>
            </form>
        </body>
    </html>
    <head><title>This is HTML</title></head>
    ‘’’

main()
```

The Python class *cgi.FieldStorage* allows access to variables passed to a CGI script. Notice the below program instantiates the *fieldstorage* class and uses *getvalue()* to obtain data from individual fields in the form.

```
#!/usr/bin/env python
import cgi

def main():
    form = cgi.FieldStorage()
    username = form.getvalue('username')
    print “Content-type: text/html\n\n”
    print ‘<html><body>’
    print “Hello ” + username + “!”
    print ‘</body></html>’

main()
```

Note: The previous script echoes back to the user any data they type into the input field on the web page. The script must be in the same directory as the *index.py* script and requires execute permissions. Pictures and other assets can be referenced from the /www/icons/ directory.

Objectives

Day 8

At the end of this training session, students will:

- ❖ None

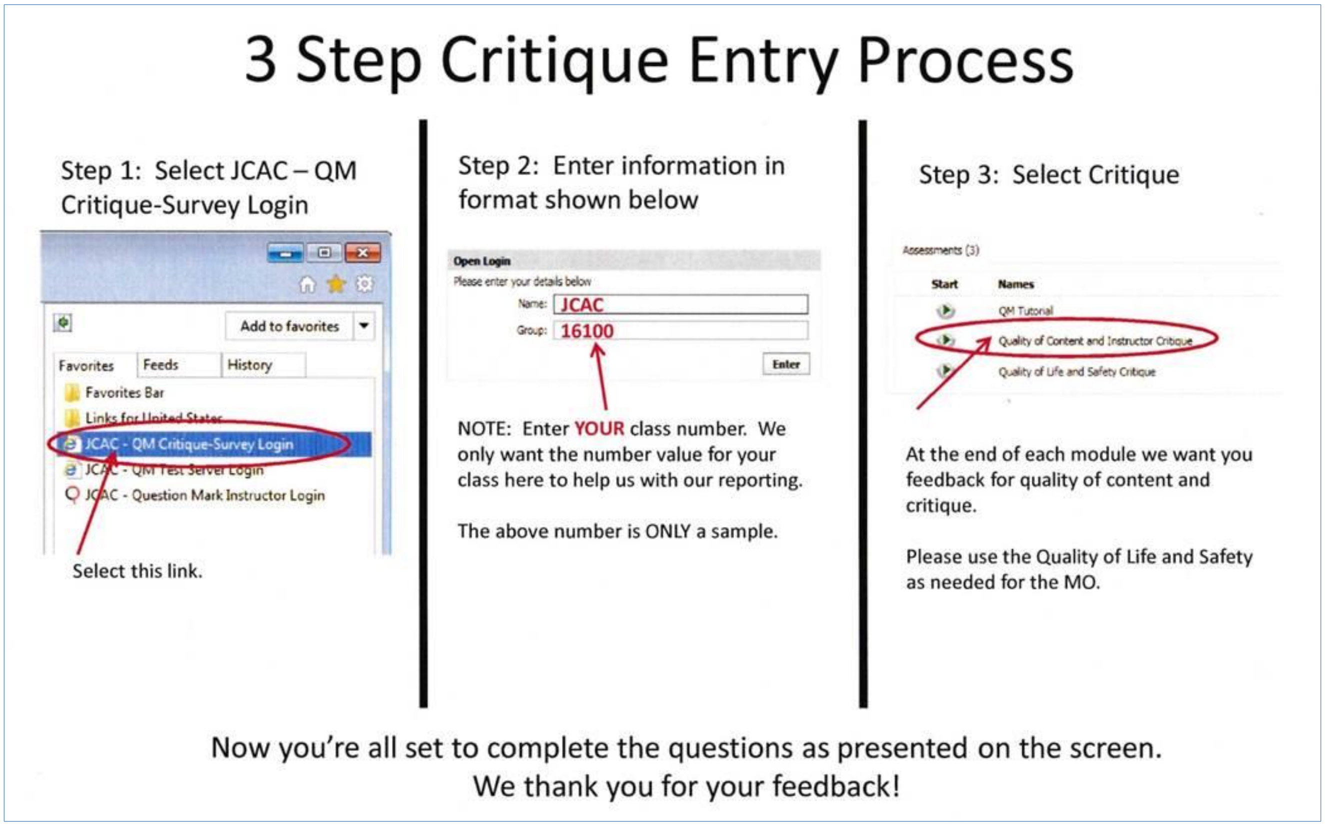
Exercises

This training session includes the following exercises:

- ❖ None

Critiques

- ❖ Submit JCAC-eTC Critique
- ❖ Submit QM Instructor and Course Critique



3 Step Critique Entry Process

Step 1: Select JCAC – QM Critique-Survey Login

Add to favorites
Favorites
Feeds
History
Favorites Bar
Links for United States
JCAC - QM Critique-Survey Login
JCAC - QM Test Server Login
JCAC - Question Mark Instructor Login

Select this link.

Step 2: Enter information in format shown below

Open Login

Please enter your details below

Name: JCAC

Group: 16100

NOTE: Enter **YOUR** class number. We only want the number value for your class here to help us with our reporting.

The above number is ONLY a sample.

Step 3: Select Critique

Assessments (3)
Start
Names
QM Tutorial
Quality of Content and Instructor Critique
Quality of Life and Safety Critique

At the end of each module we want you feedback for quality of content and critique.

Please use the Quality of Life and Safety as needed for the MO.

Now you're all set to complete the questions as presented on the screen.
We thank you for your feedback!