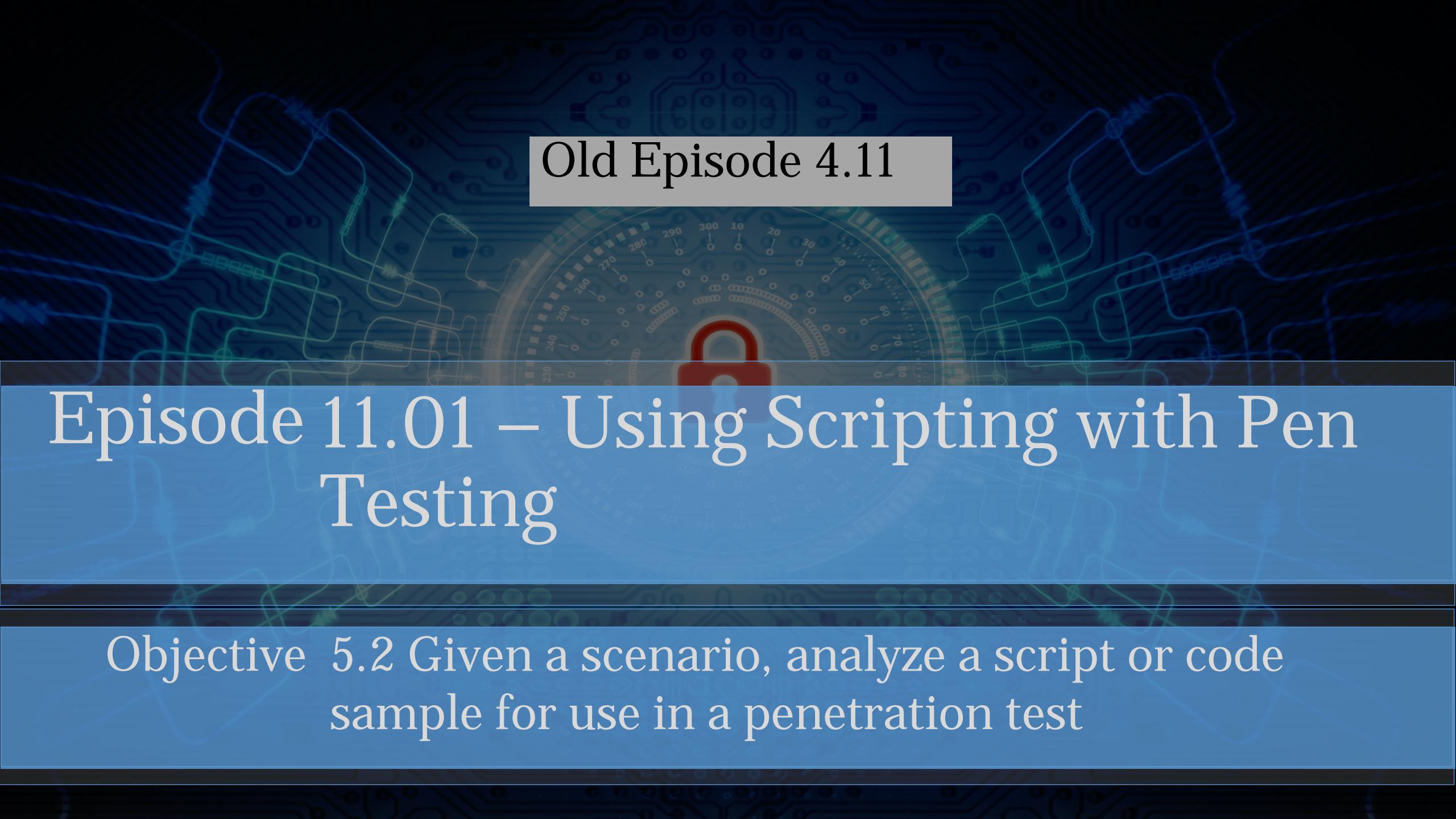


CHAPTER 11



Tools and Programming



Old Episode 4.11

Episode 11.01 – Using Scripting with Pen Testing

Objective 5.2 Given a scenario, analyze a script or code sample for use in a penetration test

SCRIPTING FOR PENETRATION TESTING

- Why bother with scripts?
 - Automate mundane/repetitive tasks
 - Faster
 - Less error prone
 - Repeatable
- What is a script?
 - Interpreted sequence of commands
 - Not compiled or assembled
 - Easy to code

COMMON SCRIPTING LANGUAGES

- Bash – Bourne Again Shell
 - Command shell for most Linux/MAC OS systems
 - Freely available version of the UNIX Bourne shell (sh)
- PowerShell – Windows-based admin and automation shell
 - Available in Windows since 2006
 - Powerful scripting language

COMMON SCRIPTING LANGUAGES

- Ruby – object-oriented high-level interpreted general purpose programming language
 - Influenced by Perl, Smalltalk, Ada, Lisp
- Python –object-oriented high-level interpreted general purpose programming language
 - Extensive available libraries
 - Great intro language

ADDITIONAL RESOURCES

- Bash
 - Curated list - <https://github.com/awesome-lists/awesome-bash>
 - <https://www.commonexploits.com/penetration-testing-scripts/>
 - <https://github.com/averagesecurityguy/scripts>
 - <https://github.com/bitvijays/Pentest-Scripts>
- PowerShell
 - <https://www.businessnewsdaily.com/10760-best-free-powershell-training-resources.html>
 - <https://blog.netwrix.com/2018/02/21/windows-powershell-scripting-tutorial-for-beginners/>

ADDITIONAL RESOURCES

- Ruby
 - <https://www.ruby-lang.org/en/>
 - <https://hackr.io/tutorials/learn-ruby>
 - <http://ruby-for-beginners.rubymonstas.org/index.html>
- Python
 - <https://learnpythonthehardway.org/>
 - <http://shop.oreilly.com/product/9781597499576.d0>

SCRIPTING

- Variables
Temporary data storage
- Substitutions
Input parameters and environment variables
- Common operations
Strings and comparisons
- Logic
Looping and flow control



SCRIPTING

- Basic I/O

Read input and write output (file, terminal, and network)

- Error handling

When things don't work

- Arrays

Simple data structure

- Encoding/decoding

Handling special characters

QUICK REVIEW

- Scripts help automate repetitive actions
- Scripts are good for standardizing testing activities
- Scripts also reduce typing errors and make tests repeatable, as well as help in documenting test activities

Old episode
4.12

Episode 11.02 Bash Scripting Basics

Objective 5.2 Given a scenario, analyze a script or code sample for use in a penetration test

COMMENTS

- Help you remember what you were thinking
 - All comments start with the ‘#’ character
 - Anything after ‘#’ is ignored by the interpreter
 - Ex: # This is a comment

• VARIABLES

- varName=value
 - Ex: name=Michael
- echo \$name
- Common to read data into variables, as opposed to hard coding too much
- Bash variables are untyped

SUBSTITUTIONS

- “\$” prefix refers to the contents of an identifier (ex. echo \$name)
- Can refer to
 - Variables \$name
 - Input parameters \$1
 - Environment variables \$PATH
 - Values from utilities \$\$(whoami)

SUBSTITUTIONS

- And, bash will set defaults when no other value is provided

```
JAVAPATH=${JAVAHOME:=/usr/lib/java}
```

```
OUTPUTDIR=${1:-/tmp} # IMPORTANT DIFFERENCE
```

COMMON OPERATIONS

- String operations
 - Concatenate var="Hello" ; var="\$var World"
 - Length \${#string} OR expr length \$string
 - Extract a substring echo \${string:position}
 - Replacing substring \${string/substring/replacement}
- Compound operations
 - AND: -a
 - OR: -o

COMPARISONS

- if [“\$varA” –eq “\$varB”]
- Equal: -eq OR ==
- Not equal: -ne OR !=
- Greater than, greater than or equal to: -gt OR >, -ge OR >=
- Less than, less than or equal to: -lt OR <, -le OR <=
- Not null (empty string): -n
- Null (empty string) : -z

LOGIC

- Looping – for

```
for var in list
```

```
do
```

```
    Statement(s)
```

```
done
```

- Examples

```
for i in 1 2 3 4 5
```

```
for i in $(seq 1 5)
```

FLOW CONTROL

```
if condition  
then  
    commands  
  
elif condition  
then  
    commands  
  
else  
    commands  
fi
```

```
if name=Michael  
    then <run some command>
```

If name doesn't equal Michael...

```
elif name=Mary  
    then <run some command>
```

If name doesn't equal Michael OR Mary...

```
else  
    then <run some command>  
fi
```

BASH if CONDITIONS

Expression	Description
-d file	True if file is a directory
-e file	True if file exists
-f file	True if file exists and is a regular file
-z string	True if string is a null (empty) string
-n string	True if string is not a null (empty string)
stringA = stringB	True if strings are equal
stringA != stringB	True if strings are not equal

BASH SCRIPTING

- test / []

```
if test -eq $name "Michael"
```

```
if [ $name = "Michael" ]
```

- break

- Exits the current loop iteration

- exit

- Exits a script and returns a value (exit code)

BASH PORT SCANNER



The image shows a terminal window titled "portscan.sh". The window has a standard OS X-style title bar with minimize, maximize, and close buttons. The menu bar contains "File", "Edit", "Search", "Options", and "Help". The main pane of the terminal displays the following Bash script:

```
#!/bin/bash

target=$1
minPort=$2
maxPort=$3

function scanports
{
for ((counter=$minPort; counter<=$maxPort; counter++))
do
    (echo >/dev/tcp/$target/$counter) > /dev/null 2>&1 && echo "$counter open"
done
}

scanports
```

QUICK REVIEW

- Bash is the default shell in Linux
- Bash makes it easy to combine multiple commands that can react to input
- Learn basic loops and conditional logic
- A few lines of a bash script can automatically execute many commands, such as scans

Old episode
4.13

Episode 11.03 Bash Scripting Techniques

Objective 5.2 Given a scenario, analyze a script or code sample for use in a penetration test

BASH SCRIPTING I/O

- I/O – File vs. terminal vs. network

- Input from a terminal

```
read -p "Enter your name:" name ; echo "Hi, " $name
```

- Input from a file

```
input="filePathName"
```

```
while IFS= read -r f1 f2 f3 f4
```

- Input from the network

```
while read -r inline < /dev/ttyS1
```

ERROR HANDLING

- Error handling
 - “\$?” is the exit status of a script we just ran

```
if [ "$?" = "0" ] then
```

ARRAYS

```
bashArray = (val1, val2, val3)
```

OR

```
declare -a bashArray = (val, val2, val3)
```

```
for i in 1 2 3  
do  
    echo ${bashArray[$i]}  
done
```

ENCODING/DECODING

- locale – shows local related environment variables
- Can change assignment of LANG for local character encoding
 - Allows bash to accept special characters (i.e. `LANG=da_DK.UTF-8`)

ENCODING/DECODING

- Can use openssl or base64 to encode and decode strings (base64)

Encoding:

```
echo string | base64
```



OR

```
base64 <<< string
```

Decoding:

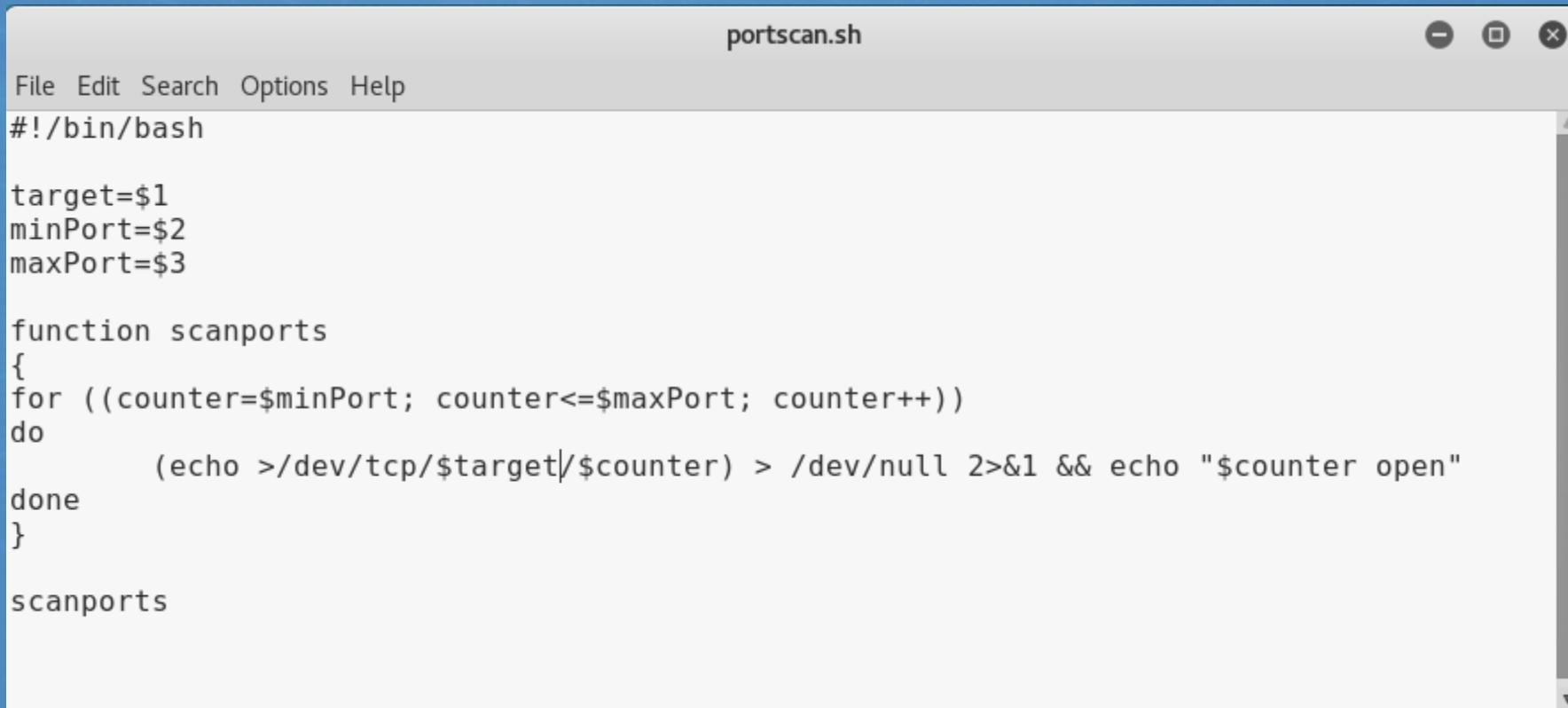
```
echo string | base64 --decode
```

OR

```
base64 -d <<< string
```

BASH: PUTTING IT ALL TOGETHER

- Port scanner in bash



The image shows a terminal window titled "portscan.sh". The window has a standard OS X style title bar with minimize, maximize, and close buttons. The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main content area contains the following Bash script:

```
#!/bin/bash

target=$1
minPort=$2
maxPort=$3

function scanports
{
for ((counter=$minPort; counter<=$maxPort; counter++))
do
    (echo >/dev/tcp/$target/$counter) > /dev/null 2>&1 && echo "$counter open"
done
}

scanports
```

QUICK REVIEW

- Redirecting input from stdin and output to stdout is the most common bash I/O technique
- Bash scripts can be used with Linux pipes
- Arrays can be useful, but aren't supported in older shells (make sure you're running bash and not sh)

Old episode
4.14

Episode 11.04 PowerShell Scripting Basics

Objective 5.2 Given a scenario, analyze a script or code sample for use in a penetration test

COMMENTS

- Helps you remember what you were thinking

Single line comments
start with the “#”
character

Multi line comments
look like this: <#
comment #>

VARIABLES

- Variables

- Variable names always start with “\$”
 - OR \$numberList = 1,3,5,7
- Write-Host \$name \$numberList
- gci variable # lists all defined variables
- Valid data types: [Array], [Bool], [DateTime], [Int], [Int32], [String] (and more)

ex. \$name = ‘Michael’

SUBSTITUTIONS

- Environment variable – Get-Item Env:varName
 - Reference with \$Env:varName
- Input parameters

```
param (  
    [string]$server = “10.10.10.0”,  
    [Parameter(Mandatory=$true)][string]$username,  
    [string]$password = (Read-Host “Input password,  
    please”)  
)
```

COMMON OPERATIONS

COMPARISONS

- if [“\$varA” –eq “\$varB”]
- Equal: -eq
- Not equal: -ne
- Greater than, greater than or equal to: -gt , -ge
- Less than, less than or equal to: -lt , -le
- Wildcard match: -like
- Match a portion of a string: -match
- Logical operators -and –or –not (or !)

LOGIC

- Looping – For, While, Do-While, Do-Until

```
For ($i=0; $i -lt $colors.Length; $i++) { cmd }
```

```
Foreach ($i in $range) { cmd }
```

```
While ($true) { cmd }
```

```
Do { cmd } While ($i -le 10)
```

```
Do { cmd } Until ($i -gt 10)
```

LOGIC

- Flow control

```
if (condition) {  
    statements  
} elseif (condition) {  
    statements  
} else {  
    statements  
}
```

I/O

- File vs. terminal vs. network

- Input from a terminal

```
$firstName = Read-Host -Prompt 'Enter first name'  
Write-Host $firstName
```

- Input from a file

```
$lines = Get-Content filename  
Out-File -FilePath filename -InputObject $lines -Encoding ASCII
```

- Input from the network

```
$socket = new-object System.Net.Sockets.TcpClient($ip, $port)  
If($socket.Connected) { }
```

ERROR HANDLING

- Try/catch

```
try {  
    Command  
}  
  
catch {  
    errorHandling commands  
}
```

ARRAYS

```
$PSarray=@(1.3.5.7.9);  
$PSarray.Length  
for ($i = 0; $i -lt $PSarray.Length; $i++) {  
    $PSarray[$i]  
}  
foreach ($element in $PSarray) {  
    $element  
}
```

POWERSHELL SCRIPTING

- Encoding/decoding

```
$OutputEncoding = [System.Text.Encoding]::Unicode
```

- Base64 encoding

```
$Text = 'Hello world'
```

```
$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text)
```

```
$EncodedText = [Convert]::ToBase64String.($Bytes
```

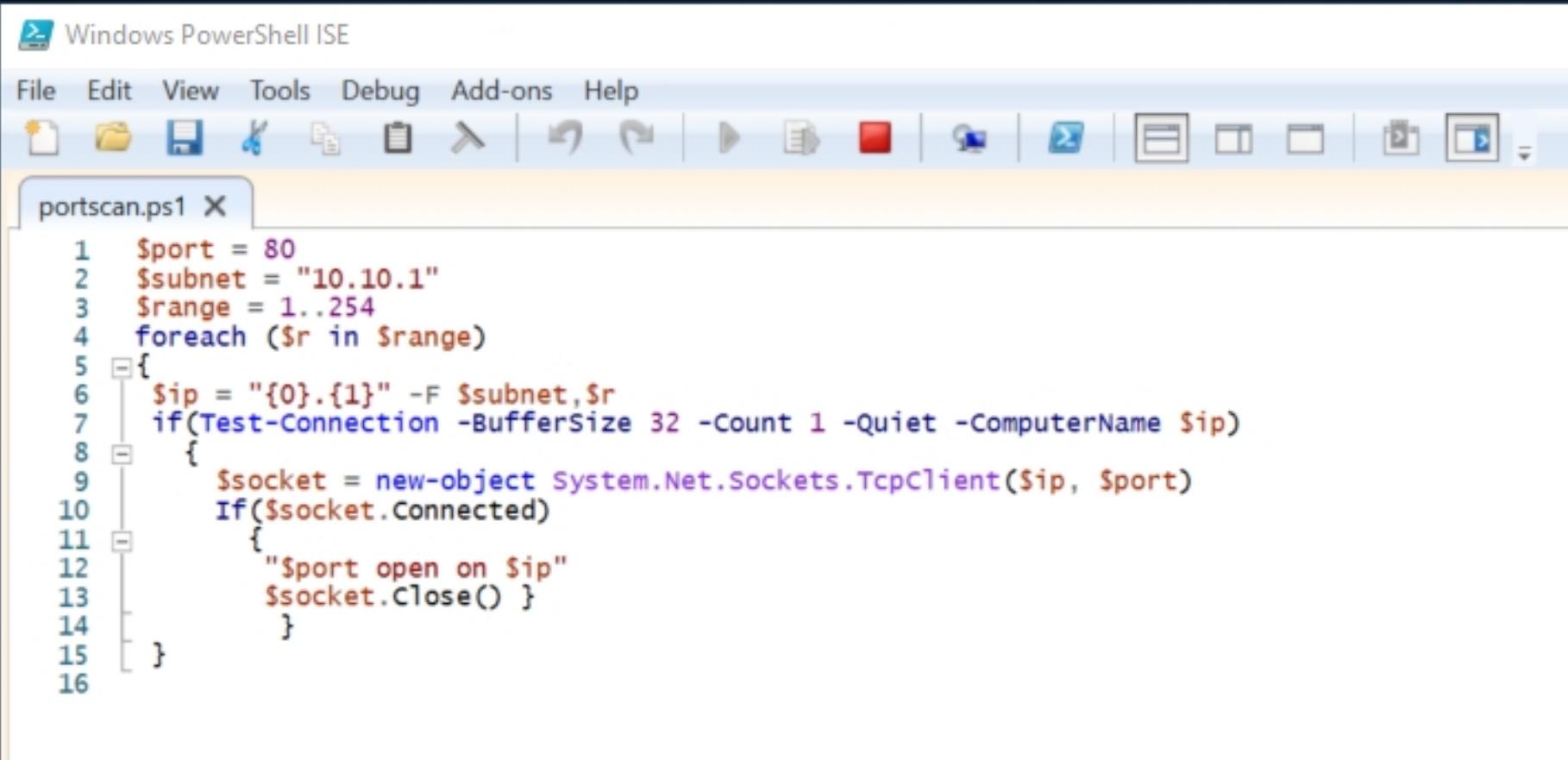
- Base64 decoding

```
$EncodedText = 'encodedString'
```

```
$DecodedText =
```

```
[System.Tet.Encoding]::Unicode.GetString([System.Convert]::FromBase  
64String($EncodedText))
```

PowerShell: Putting it all together



The image shows the Windows PowerShell Integrated Scripting Environment (ISE) window. The title bar reads "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main area displays a script file named "portscan.ps1" with the extension ".X". The script content is as follows:

```
1 $sport = 80
2 $subnet = "10.10.1"
3 $range = 1..254
4 foreach ($r in $range)
5 {
6     $ip = "{0}.{1}" -F $subnet,$r
7     if(Test-Connection -BufferSize 32 -Count 1 -Quiet -ComputerName $ip)
8     {
9         $socket = new-object System.Net.Sockets.TcpClient($ip, $sport)
10        If($socket.Connected)
11        {
12            "$sport open on $ip"
13            $socket.Close()
14        }
15    }
16 }
```

QUICK REVIEW

- PowerShell is currently open source and available for multiple operating systems
- PowerShell scripts are disabled in Windows by default

Old episode
4.15

Episode 11.05 Ruby Scripts

Objective 5.2 Given a scenario, analyze a script or code sample for use in a penetration test

HOW TO RUN RUBY SCRIPTING

- Download and install Ruby
 - <https://www.ruby-lang.org/en/downloads/>
 - Launch Ruby: irb (Interactive Ruby) (ctrl-D to exit)
 - Or, just run Ruby from a web browser -
<https://ruby.github.io/TryRuby/>
- Comments
 - '#' for single line comments, =begin comments =end (multi-line comments)
- Variables
 - name = “Michael”
 - number = 22
 - puts name, number
 - Valid data types: number, string, Boolean, symbol, array, hash

SUBSTITUTIONS

- Environment variables
- Input parameters

puts ENV['PATH']

ARG[0] ARG[1]

```
ARGV.each do |a|
```

```
  puts "Argument: #{a}"
```

```
end
```

- Ruby also has an OptionParser library

- Values from other utilities

```
`echo $PATH`
```

COMMON OPERATIONS

- String operations
 - Concatenation
“snow” + “ball”
 - Repetition
“hi” * 3
 - Length
“hello”.length
 - Substring (extract or replace)
“hello”[1..3]

COMMON OPERATIONS

- Comparisons
 - Equal `==`
 - Not equal `!=`
 - Greater than, greater than or equal to `>`, `>=`
 - Less than, less than or equal to `<`, `<=`
- Logical operations
 - and `&&`
 - or `||`
 - not `!`

LOGIC

- Looping – while, until, for

```
while condition do  
    statements  
end
```

```
until condition do  
    statements  
end
```

```
for var in expression do  
    statements  
end
```



LOGIC

- Flow control

```
if condition then  
    statements  
elseif  
    statements  
else  
    statements  
end
```



LOGIC

Case input

when “A”

statement

when “B”

statement

else

statement

end



I/O

- File vs. terminal vs. network

- Input from terminal
- Input from a file

```
name = gets
```

```
inFile = File.new("filename","r")
```

```
inFile.each_line{|line| puts}
```

```
"#{line.dump}" }
```

```
inFile.close
```

```
$stdout << 76 << "trombones" << "\n"
```

```
client = TCPSocket.open('hostname', 'port')
```

```
client.send("string",0)
```

- Output to a file
- Network I/O

ERROR HANDLING

- begin / end / rescue

begin

statements

rescue

statements if error occurred

else

statements if no error

end

ARRAYS

```
rubyArray = [ “val1”, “val2”, “val3” ]  
print rubyArray[1]  
print rubyArray.index(“val2”)  
print rubyArray.last OR print rubArray[-1]
```

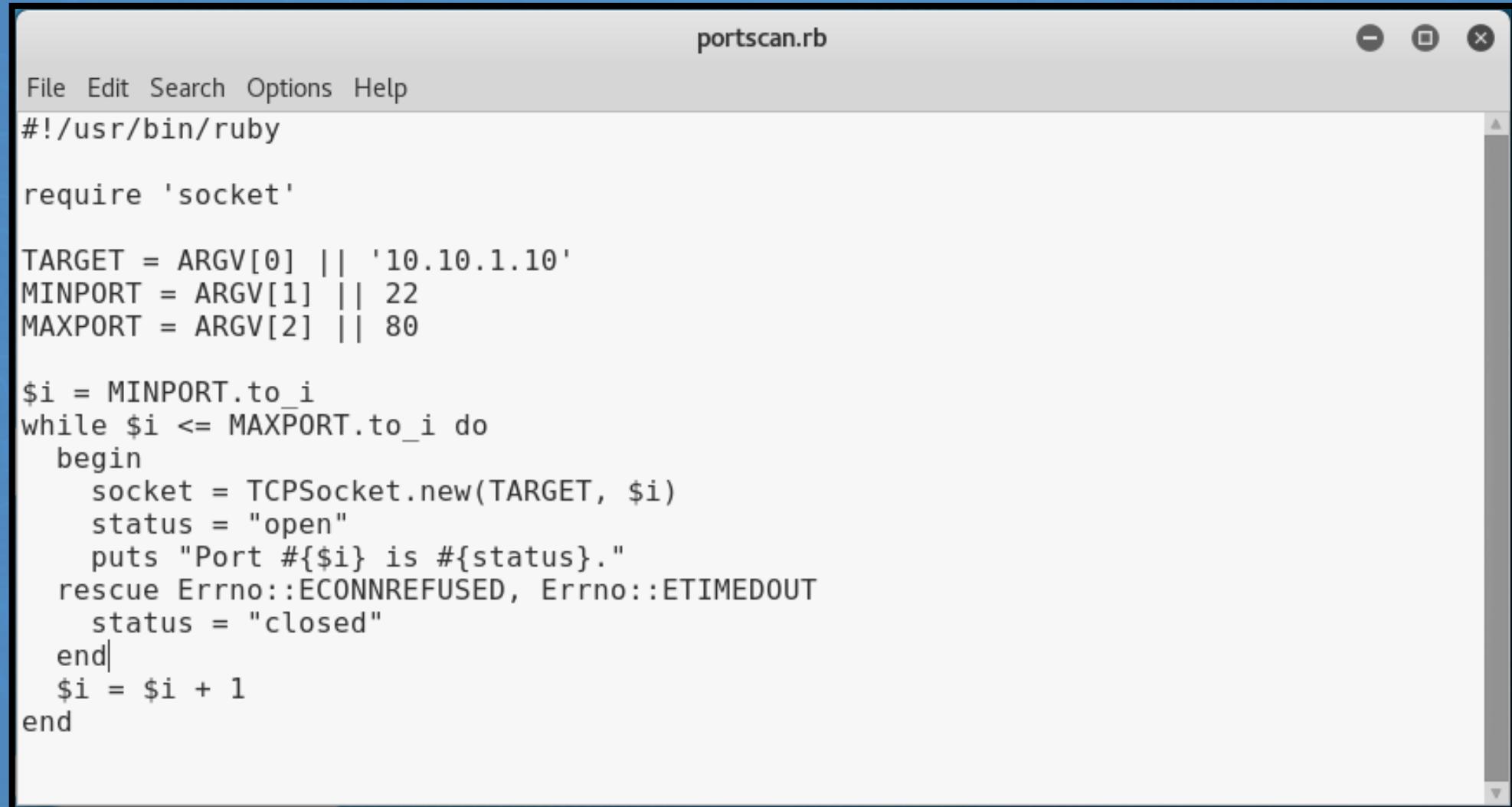
ENCODING/DECODING

```
Require "base64"
```

```
encString = Base64.encode64('Hello world!')
```

```
plaintext = Base64.decode(enc)
```

RUBY: PUTTING IT ALL TOGETHER



The image shows a terminal window titled "portscan.rb". The window has a standard OS X-style title bar with minimize, maximize, and close buttons. The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main pane contains the following Ruby script:

```
#!/usr/bin/ruby

require 'socket'

TARGET = ARGV[0] || '10.10.1.10'
MINPORT = ARGV[1] || 22
MAXPORT = ARGV[2] || 80

$i = MINPORT.to_i
while $i <= MAXPORT.to_i do
  begin
    socket = TCPSocket.new(TARGET, $i)
    status = "open"
    puts "Port #{$i} is #{status}."
  rescue Errno::ECONNREFUSED, Errno::ETIMEDOUT
    status = "closed"
  end
  $i = $i + 1
end
```

QUICK REVIEW

- Ruby is a powerful object-oriented language that can do far more than just scripting
- Ruby's popularity is related to the Ruby on Rails server-side web application framework written in Ruby
- Ruby treats everything as an object and relies heavily on methods and attributes

Old episode
4.16

Episode 11.06 Python Scripts

Objective 5.2 Given a scenario, analyze a script or code sample for use in a penetration test

PYTHON SCRIPTING

- Download and install Python
 - <https://wiki.python.org/moin/BEGINNERSGUIDE/Download>
 - Two versions in use: 2 and 3
 - Launch Python: python (ctrl-D to exit)
- Comments - all comments start with “#”
- Variables
 - name = “Michael”
 - number = 22
 - print(name + “ “ + str(number))
 - Valid datatypes: numbers, string, list, tuple, dictionary

SUBSTITUTIONS

- Input arguments (parameters)

```
import sys  
print ("Name of script:", sys.argv[0])  
print ("Number of arguments: ", len(sys.argv))  
print ("Arguments: ", str(sys.argv))
```

- Environment variables

```
import os  
extPath = os.environ['PATH']
```

COMMON OPERATIONS

- String operations
 - Concatenate
 - Length
 - Extract substring
 - Replace a substring

`string1 + string 2`

`len(string)`

`string[start:end+1]`

`string.replace(old, new,
count)`

COMMON OPERATIONS

- Comparisons
 - Equal `==`
 - Not equal `!=` OR `<>`
 - Greater than, greater than or equal to `>`, `>=`
 - Less than, less than or equal to `<`, `<=`
- Logical operations
 - and
 - or
 - not

LOGIC

- Looping – for, while

```
for i in range(1, 10):  
    print(i)
```

```
while x < 10:  
    print (x)  
    x += 1
```

LOGIC

- Flow control – if

```
If var == value:
```

```
    statements
```

```
elif var > value:
```

```
    statements
```

```
else:
```

```
    statements
```

- Notice indentation

I/O

- File vs. terminal vs. network
 - Input from a terminal
 - name = raw_input('Please enter your name') # map to simple datatype
 - toppings = input('Which toppings do you want on your pizza?') # maps to complex datatype
 - Input() will store data in the “best” datatype (i.e. list, etc.)

I/O

- Input from a file

```
f = open('inFile.txt','r')
```

```
for line in f:
```

```
    do something here
```

```
f.close()
```

I/O

- Output to a file

```
f = open('outFile.txt','w')  
for i in range(1,11):  
    print >> f, I  
f.close()
```

I/O

- Input from a network

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
If sock.connect_ex((remoteServerIP, port)) == 0:
    print ('Port {}: is Open'.format(port))
```

ERROR HANDLING

- Try / except / finally blocks

```
try:
```

```
    statements
```

```
    raise customErrorObject
```

```
except errorObject:
```

```
    statements
```

```
except customErrorObject:
```

```
    statements
```

```
finally:
```

```
    statements to clean up
```

ARRAYS

```
pythonArray = [10, 20, 30, 40, 50]
Print(pythonArray[1])      # -1 is last element index
len(pythonArray)
pythonArray.append(60)    # add 60 to the array
pythonArray.remove(30)    # remove element 30
pythonArray.pop(3)        # remove the 4th current element
```

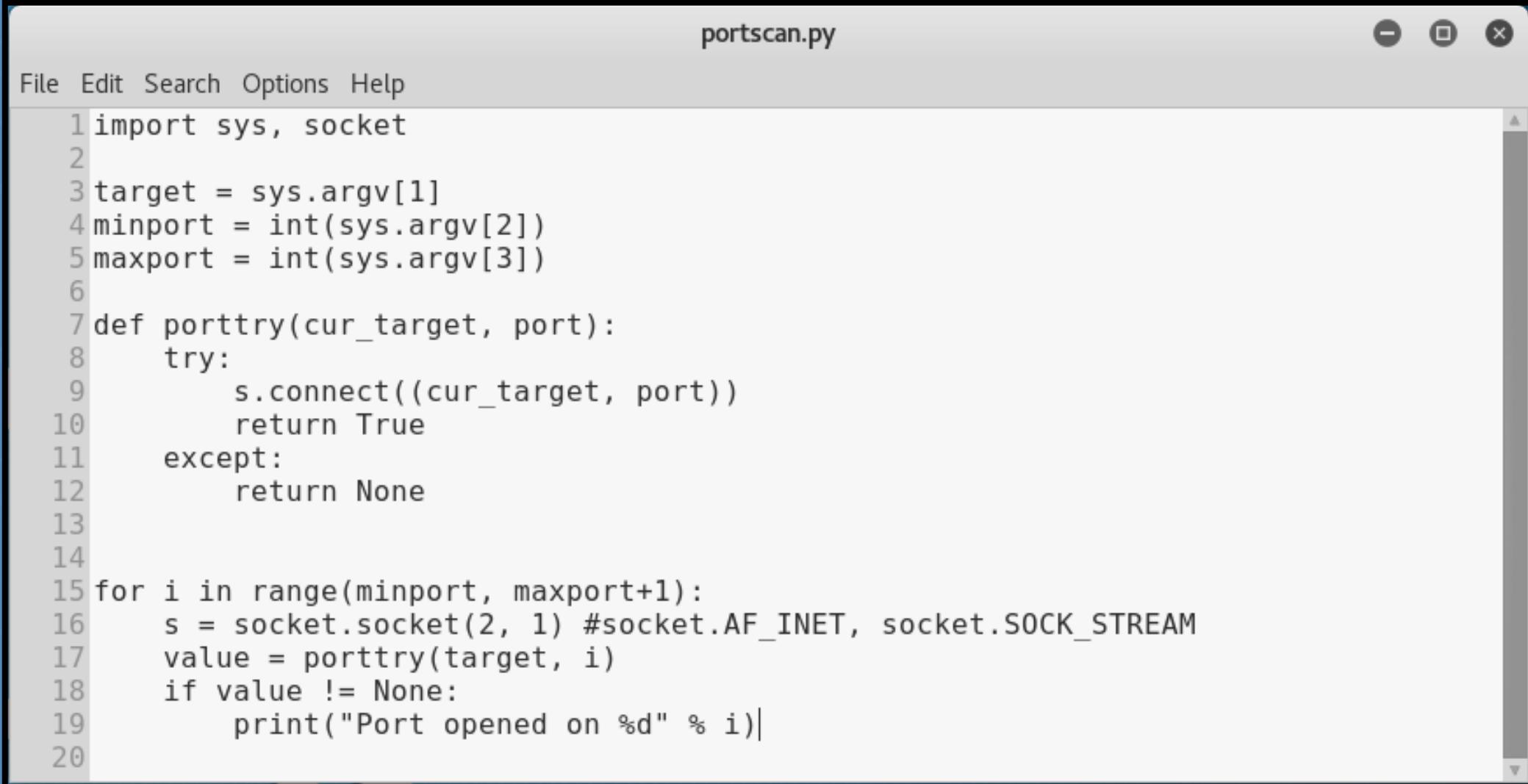
ENCODING/DECODING

```
Import base64
```

```
encString = base64.encodestring('Hello world!')
```

```
plaintext = base64.decodestring(encString)
```

Python: Putting it all together



```
portscan.py

File Edit Search Options Help
1 import sys, socket
2
3 target = sys.argv[1]
4 minport = int(sys.argv[2])
5 maxport = int(sys.argv[3])
6
7 def porttry(cur_target, port):
8     try:
9         s.connect((cur_target, port))
10        return True
11    except:
12        return None
13
14
15 for i in range(minport, maxport+1):
16     s = socket.socket(2, 1) #socket.AF_INET, socket.SOCK_STREAM
17     value = porttry(target, i)
18     if value != None:
19         print("Port opened on %d" % i)
20
```

QUICK REVIEW

- Python is another powerful object-oriented language
- Python is a popular language because it's easy to write very powerful programs in just a few lines of code
- Unlike many other languages, Python depends on indentation to define blocks

Old episode
4.17

Episode 11.07 Scripting Languages Comparison

Objective 5.2 Given a scenario, analyze a script or code sample for use in a penetration test

Comparing Scripting Languages

	Bash	PowerShell	Ruby	Python
Comments	#	# or <# #>	# or =begin =end	#
Variables – assign	varName=value	\$varName=value	varName=value	varName=value
Variables – display	echo \$varName	Write-Host \$varName	puts varName	print(varName)
Substitution – environment variables	\$env:varName	Get-item Env:varName	ENV['varName']	Os.environ['varName']

Comparing Scripting Languages

	Bash	PowerShell	Ruby	Python
String length	<code>\${#string}</code>	<code>(string).Length</code>	<code>string.length</code>	<code>len(string)</code>
String – substring	<code>\${string:position}</code>	<code>(string).Substring(start,end)</code>	<code>string[1..3]</code>	<code>string[start:end+1]</code>
String – replace substring	<code>\${string/substring/replacement}</code>	<code>(string).Replace(substr,replStr)</code>	<code>string[1..3] = replStr</code>	<code>string.replace(old, new, count)</code>
AND/OR	<code>-a / -o</code>	<code>-and, -or, -not !</code>	<code>and &&, or , not !</code>	<code>and, or, not</code>
Comparisons	<code>-eq (==), -ne (!=), -lt (<), -le (<=), -gt (>), -ge (>=)</code>	<code>-eq, -ne, -gt, -ge, -lt, -le</code>	<code>==, !=, >, >=, <, <=</code>	<code>==, != (<>), >, >=, <, <=</code>

Comparing Scripting Languages

	Bash	PowerShell	Ruby	Python
Looping	For	For, While, Do-While, Do-Until	while, until, for	for, while
Flow control	if condition then commands elif commands else commands fi	if (condition) { statements } elseif (condition) { statements } else { statements }	If condition then statements elsif statements else statements end	if condition: statements elif condition: statements else: statemenst

Comparing Scripting Languages

	Bash	PowerShell	Ruby	Python
Input – file	Input=“filename” While IFS=read –r f1 f2 f3	\$lines = Get-Content filename Out-File –FilePath filename –InputObject \$lines –Encoding ASCII	inFile = File.new(“filename”,”r”) inFile.each_line { line puts “#{line.dump}” } inFile.close	f = open(‘inFile.txt’,’r’) for line in f: do something here f.close()
Input – terminal	Read –p “Prompt:” var	\$firstName = Read- Host –Prompt ‘Enter first name’	name = gets	name = raw_input (‘Please enter your name’)

Comparing Scripting Languages

	Bash	PowerShell	Ruby	Python
Input – network	While read -r inline < /dev/ttyS1	\$socket = new-object System.Net.Sockets.Tcp Client(\$ip, \$port) if(\$socket.Connected) { }	client = TCPSocket.open (‘hostname’, ‘port’) Client.send(“strin g”,o)	sock = socket.socket(soc ket.AF_INET, socket.SOCK_ST REAM) If sock.connect_ex((remoteServerIP, port)) == 0: print (‘Port {}: is Open’.format(por t)

Comparing Scripting Languages

	Bash	PowerShell	Ruby	Python
Error handling	If [“\$?” = “0”] then Command } catch { errHandling commands }	try { Command } catch { errHandling commands }	begin statement rescue statement s if error occurred else statement s if no error end	try: statement s raise customErrorObject except errorObject: statement s except customErrorObject : statement s finally: statement s to clean up

Comparing Scripting Languages

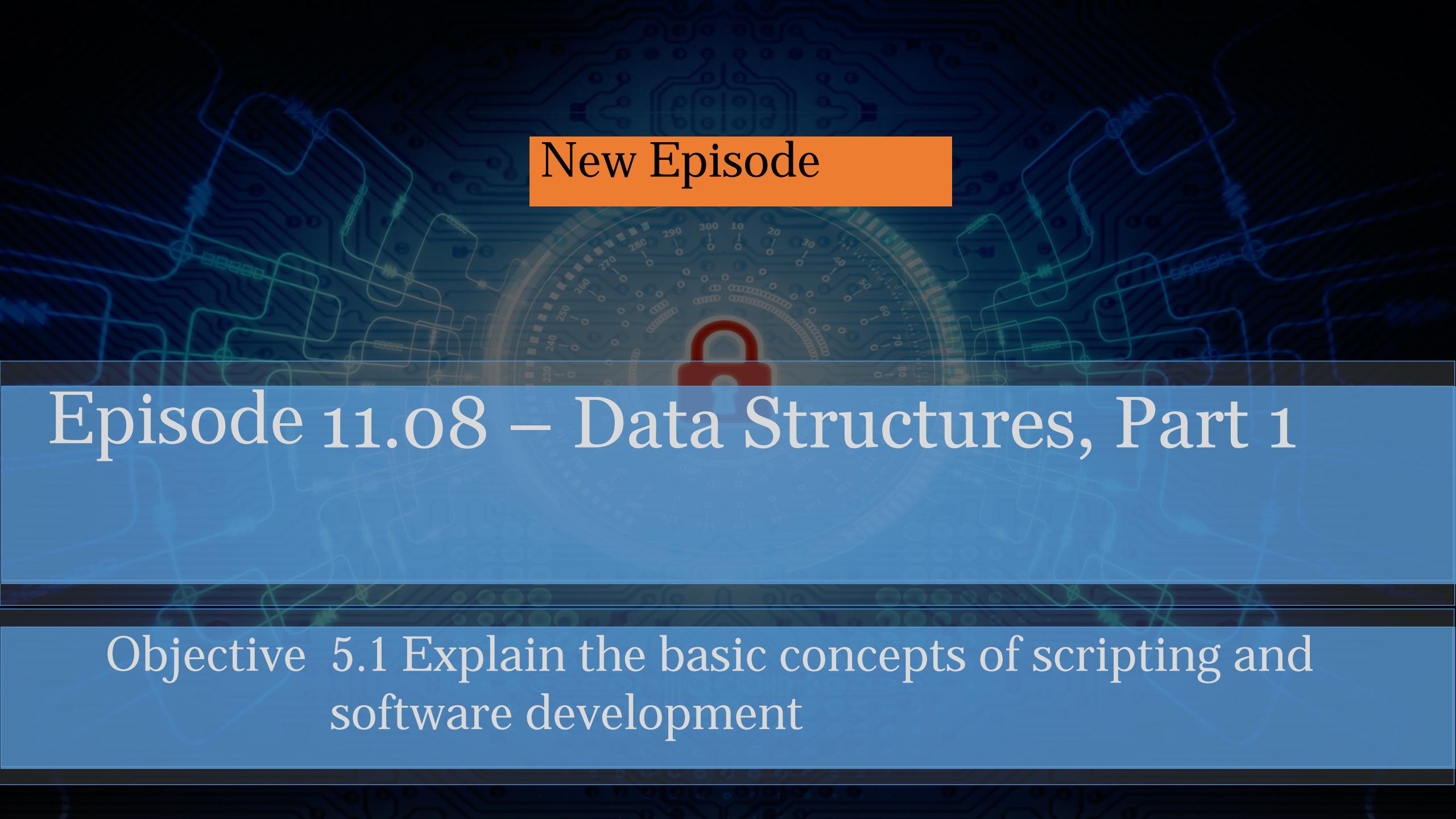
	Bash	PowerShell	Ruby	Python
Arrays	bashArray = (val1, val2, val3) For I in 1 2 3 Do echo \\$bashArray[\\$i] done	\$PSarray=@(1.3.5.7.9); for (\$i = 0; \$i -lt \$PSarray.Length; \$i++) { \$PSarray[\$i] } foreach (\$element in \$PSarray) { \$element }	rubyArray = [“val1”, “val2”, “val3”] print rubyArray[1] print rubyArray.index(“val2”))	pythonArray = [10, 20, 30, 40, 50] Print(pythonArray[1]) len(pythonArray)

Comparing Scripting Languages

	Bash	PowerShell	Ruby	Python
Encoding	Echo plainText base64	\$Text = 'Hello world' \$Bytes = [System.Text.Encoding]::Unicode.GetBytes(\$Text) \$EncodedText = [Convert]::ToBase64String.(\$Bytes)	Require "base64" encString = Base64.encode64("Hello world!")	Import base64 encString = base64.encodestring('Hello world!')
Decoding	Echo encString base64 --decode	\$EncodedText = 'encodedString' \$DecodedText = [System.Tet.Encoding]::Unicode.GetString([System.Convert]::FromBase64String(\$EncodedText))	plaintext = Base64.decode(enc)	plaintext = base64.decodestring(encString)

QUICK REVIEW

- Recognize unique Bash script syntax - output (echo), error-handling ("\$?")
- Recognize unique PowerShell script syntax - output (Write-Host), flow-control (elseif), error-handling (try/catch)
- Recognize unique Ruby syntax - output (puts), flow-control (elsif), error-handling (rescue)
- Recognize unique Python syntax - output (print), error-handling (try/except/finally)

A dark blue background featuring a glowing blue circuit board pattern. In the center, there is a semi-transparent orange rectangular overlay. Inside this overlay, the words "New Episode" are written in a white, sans-serif font.

New Episode

Episode 11.08 – Data Structures, Part 1

Objective 5.1 Explain the basic concepts of scripting and software development

SLATE

Clip: Roll 3 Clip 001

Chapter Name: 11 Tools and Programming

Proposed Episode #: 11.18

Episode Name: Data Structures, Part 1

Date: 3/10/22

New episode for 002

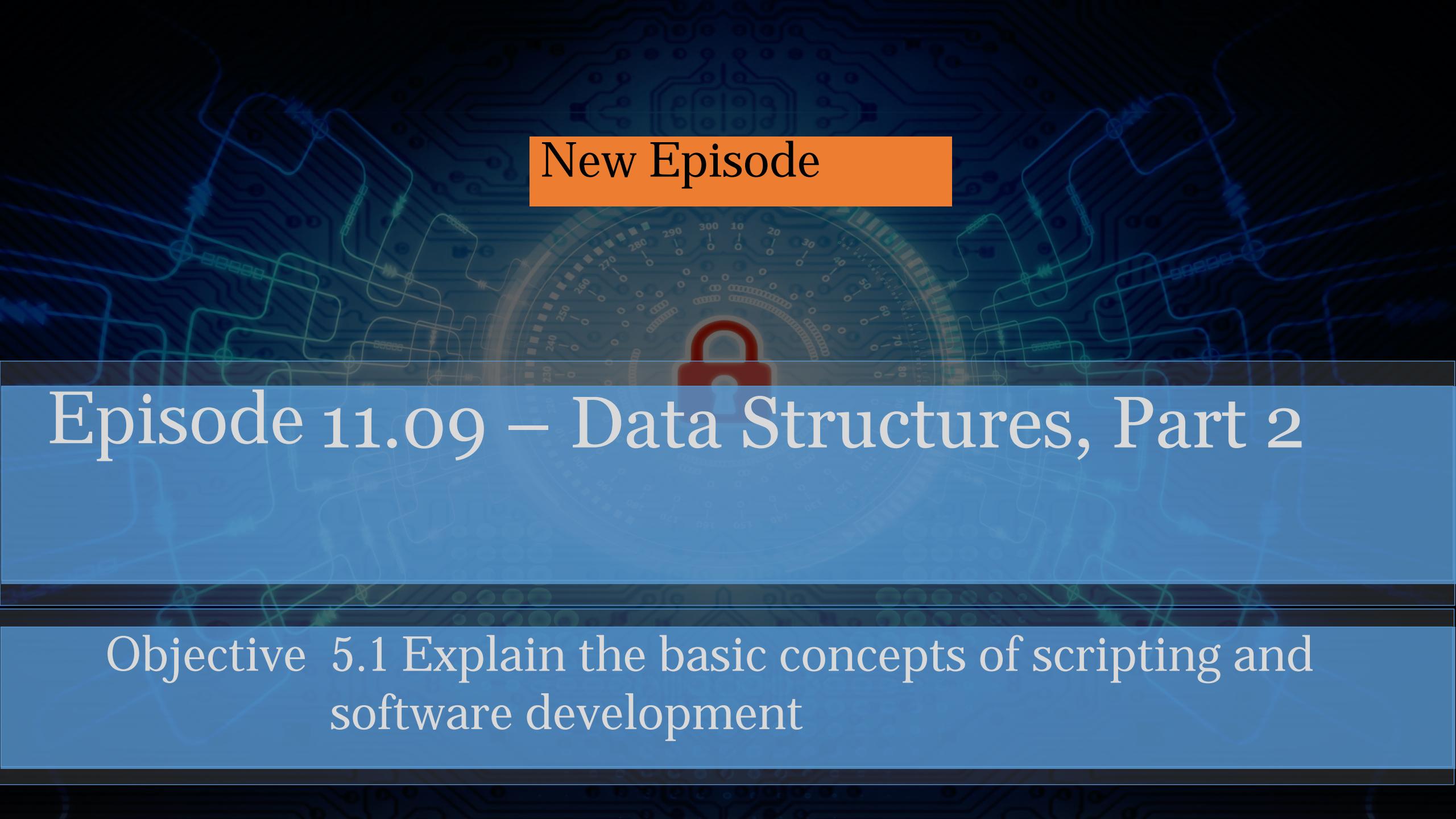
- Advanced scripting and software development

5.1.2a – Scripting/software dev - Data structures

- JavaScript Object Notation (JSON)
 - Show JSON example(s) in code and use
 - Explain why JSON is popular
- Key value
 - Python example
- Dictionaries
 - Python example

QUICK REVIEW

- JSON is a popular format for programs to exchange data
- JSON is readable by humans and easy to use for many languages
- Key-value pairs are easy ways to label data fields
- Dictionaries provide a way to store collections of key-value pairs

A dark blue background featuring a glowing blue circuit board pattern. In the center, there is a semi-transparent circular dial with numbers from 0 to 300. Overlaid on the dial is a red padlock icon. A solid orange rectangular button is positioned in the upper center, containing the text "New Episode".

New Episode

Episode 11.09 – Data Structures, Part 2

Objective 5.1 Explain the basic concepts of scripting and software development

SLATE

Clip: Roll 3 Clip 002

Chapter Name: 11 Tools and Programming

Proposed Episode #: 11.19

Episode Name: Data Structures, Part 2

Date: 3/10/22

New episode for 002

- Advanced scripting and software development

5.1.2b – Scripting/software dev - Data structures

- Comma-separated values (CSV)
 - Show example
 - Explain why csv is popular
- Lists
 - Explain/describe
- Trees
 - Explain types of trees and their uses

QUICK REVIEW

- CSV is a simple way to store data in files that many programs can read
- Individual data fields in a CSV are separated by commas
- A list is a convenient way to store multiple data values
- Trees are more complex data structures that support fast searching

A dark blue background featuring a glowing blue circuit board pattern. In the center, there is a bright orange rectangular button with the text "New Episode" in black. A red padlock icon is positioned in front of a circular dial with numerical markings (230, 240, 250, 260, 270, 280, 290, 300, 10, 20, 30, 40, 50, 60, 70, 80) and small square markers.

New Episode

Episode 11.10 – Libraries

Objective 5.1 Explain the basic concepts of scripting and software development

SLATE

Clip: Roll 3 Clip 003

Chapter Name: 11

Proposed Episode #: 11.20

Episode Name: Libraries

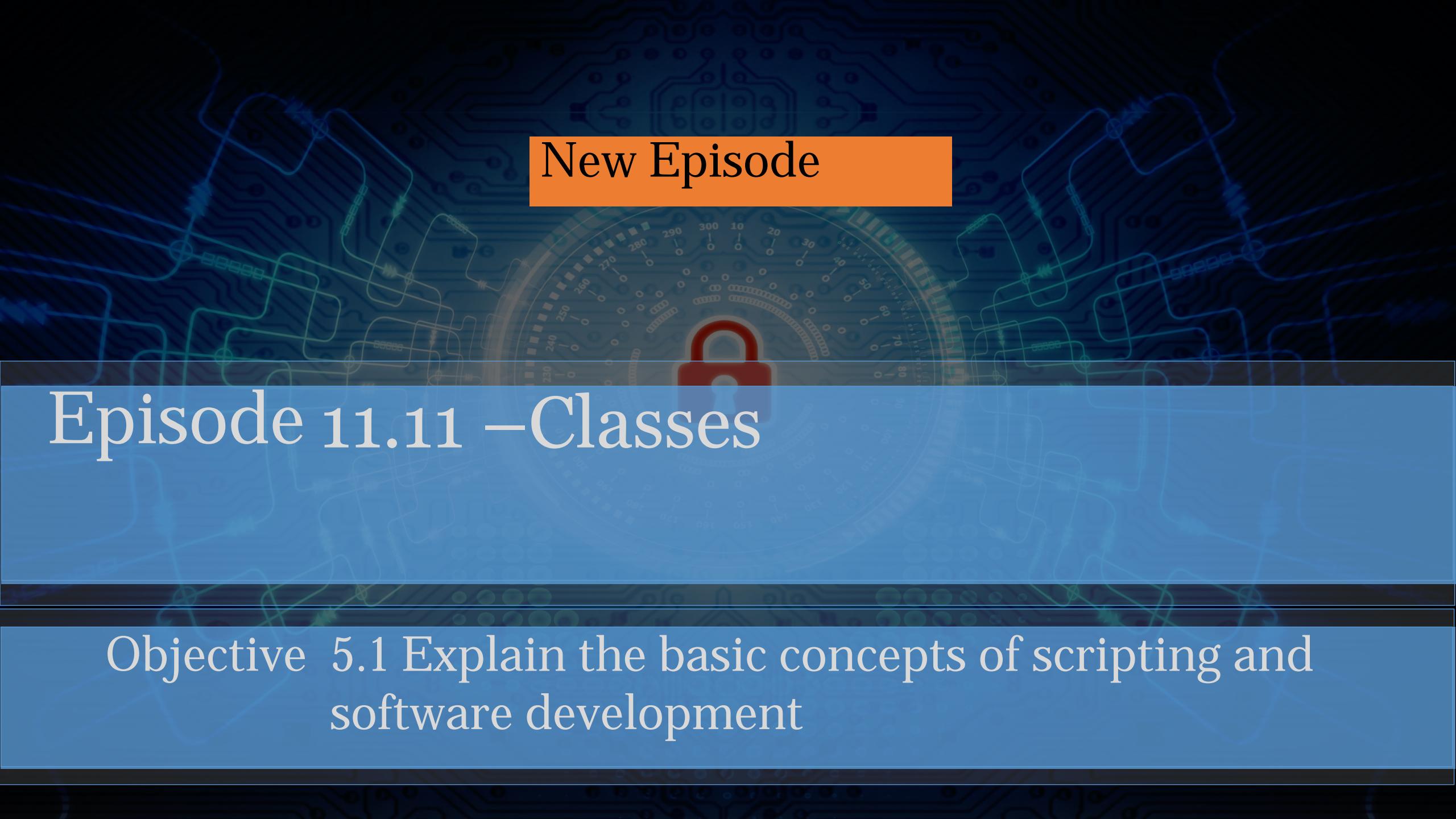
Date: 3/10/22

3.1.3 – Scripting/software dev - Libraries

- Libraries provide ready-made functionality
- Show Python example(s)
- Note the reliance on other developers for quality and security

QUICK REVIEW

- Libraries are collections of external code that are included in a program
- Libraries make it easy to draw on a large volume of code someone else wrote
- If a library contains vulnerabilities, any code that includes it will contain them, too

A dark blue background featuring a glowing blue circuit board pattern. In the center, there is a bright orange rectangular button with the words "New Episode" in black serif font.

New Episode

Episode 11.11 – Classes

Objective 5.1 Explain the basic concepts of scripting and software development

SLATE

Clip: Roll 3 Clip 004

Chapter Name: 11

Proposed Episode #: 11.21

Episode Name: Classes

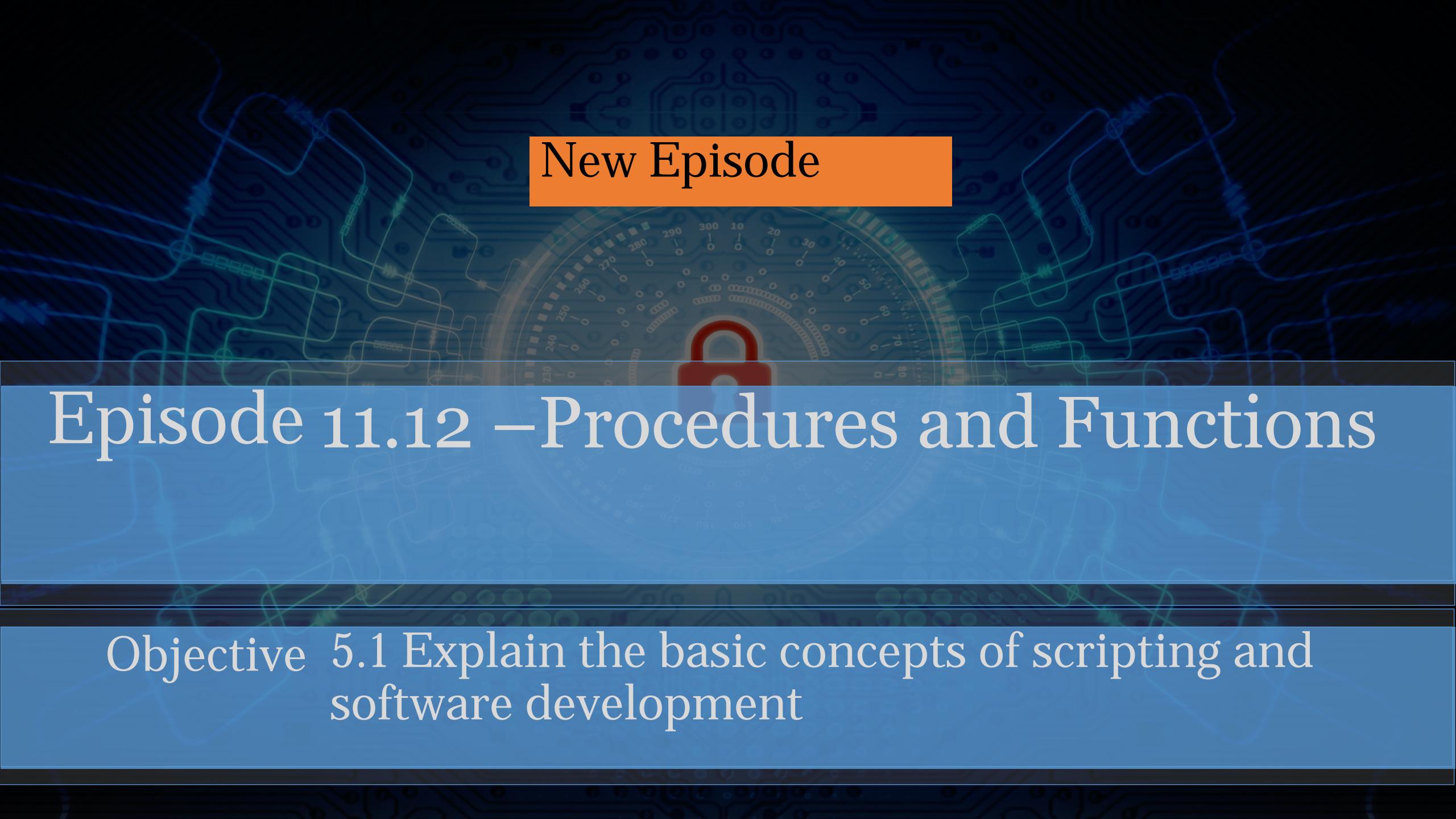
Date: 3/10/22

5.1.4 – Scripting/software dev - Classes

- Describe basic OO concepts
- Explain the value of classes
- Show Python class example

QUICK REVIEW

- A class is a blueprint, or template, that contains definitions used to create objects
- An object is an instance of a class that includes state data and functionality
- Each instance of an object is a unique entity
- Object Oriented (OO) programming supports a modular approach to handling data and functionality

A dark blue background featuring a glowing blue circuit board pattern. In the center, there is a bright orange rectangular button with the words "New Episode" in black serif font.

New Episode

Episode 11.12 – Procedures and Functions

Objective 5.1 Explain the basic concepts of scripting and software development

SLATE

Clip: Roll 3 Clip 005

Chapter Name: 11

Proposed Episode #: 11.22

Episode Name: Procedures and Functions

Date: 3/10/22

3.1.3 – Scripting/software dev - Procedures

- Define procedures
- Explain the difference between procedures and classes
- Show Python example

5.1.6 – Scripting/software dev - Functions

- Describe functions
- Explain the difference between functions and procedures
- Show Python examples

QUICK REVIEW

- Functions and procedures allow programmers to separate frequently used code and call it as needed
- Procedures only execute statements – no return value
- Functions execute statements and return a value to the caller