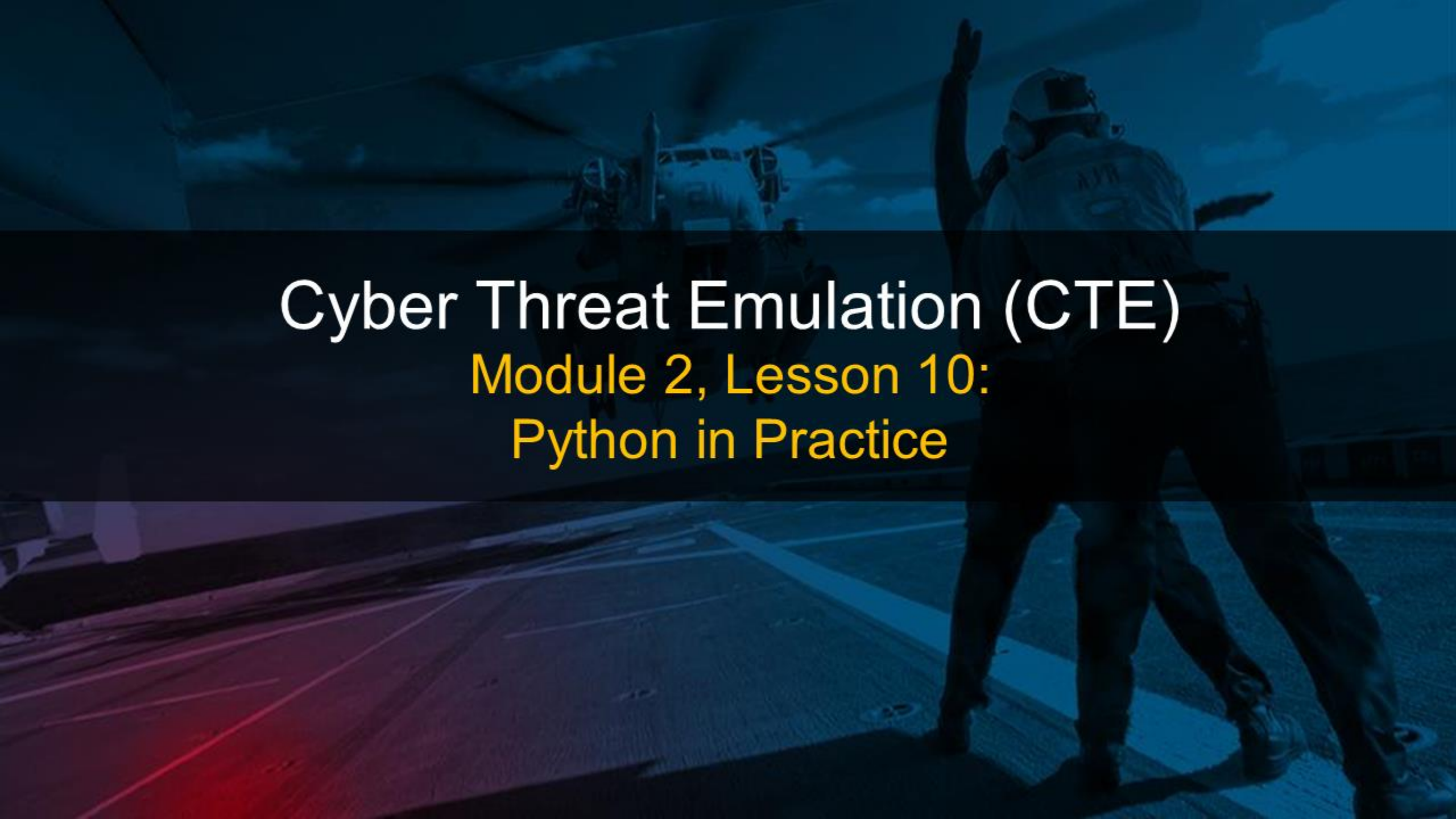


Cyber Threat Emulation (CTE)

Module 2, Lesson 10:
Python in Practice

Course Objectives

After completing this course, students will be able to:

- Summarize the CTE squad's responsibilities, objectives, and deliverables from each CPT stage
- Analyze threat information
- Develop a Threat Emulation Plan (TEP)
- Generate mitigative and preemptive recommendations for local defenders
- Develop mission reporting
- Conduct participative operations
- Conduct reconnaissance
- Analyze network logs for offensive and defensive measures

Course Objectives (Continued)

Students will also be able to:

- Analyze network traffic and tunneling protocols for offensive and defensive measures
- Plan non-participative operations using commonly used tools, techniques and procedures (TTPs)

Module 2: Threat Emulation (Objectives)

- Conduct reconnaissance
- Generate mission reports from non-participative operations
- Plan a non-participative operation using social engineering
- Plan a non-participative operation using Metasploit
- Analyze network logs for offensive and defensive measures
- Analyze network traffic and tunneling protocols for offensive and defensive measures
- Plan a non-participative operation using Python
- Develop fuzzing scripts
- Develop buffer overflow exploits

Module 2 – Lesson 10: Python in Practice (Objectives)

- Utilize fundamental scripting concepts in Python
- Execute code injection in Python
- Perform target reconnaissance with Python built-in libraries

Lesson Overview

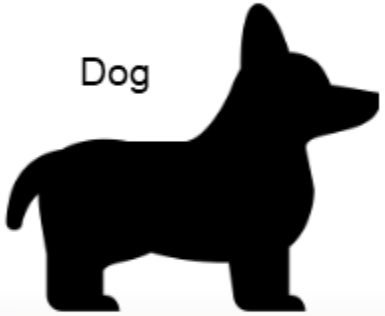
In this lesson we will discuss:

- Object-Oriented Programming
- List Comprehension
- Dangerous Functions
  - **Exercise 1: Abusing Python 2 Input Functions**
- File Handling
- Introduction to Modules (`socket`)
  - **Exercise 2: FTP Banner Grab**

Object Oriented Programming:

- An ***object*** is an abstract idea of an entity that:
  - has **properties** (*variables*) and
  - can **perform actions** (*functions*)
- In OOP vernacular, a "function" is called a "**method**".
- Each **object** is defined based off a ***class***.

CLASS
Dog



OBJECT
Regina

PROPERTIES
Height
Length
Weight
Coat Color

METHODS
Sit
Stay
Paws Up
Shake

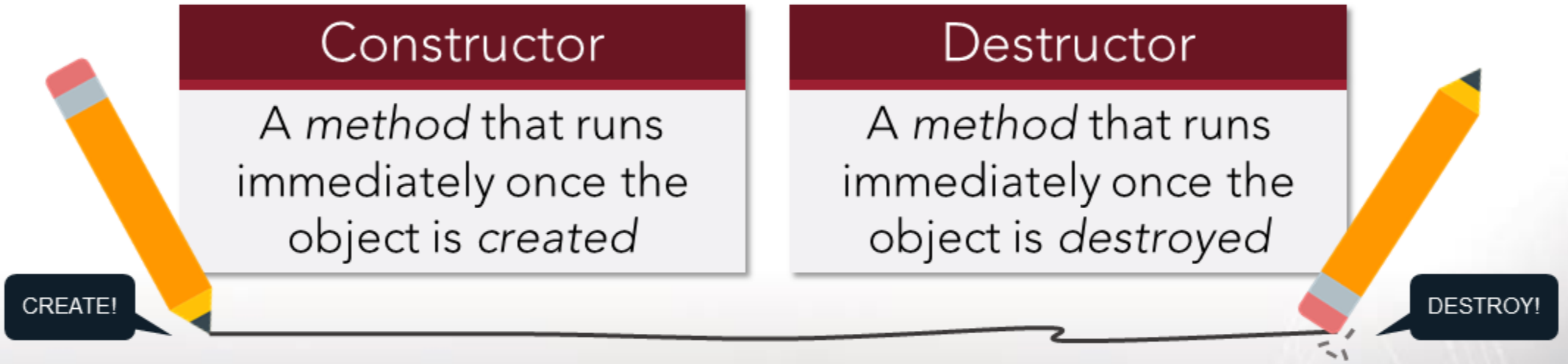
So what is a class?

- A ***class*** is like a "blueprint" that outlines what the object is and does.

  Classes define an object's properties, methods, other objects it might resemble, how certain operators affect it, and more.

- Typically every class is defined with a **constructor.**
  *(and sometimes, a **destructor!**)*

Constructors and Destructors:



- You don't often see destructors in Python, but constructors are crucial.
- All the properties necessary for the object are defined in the **constructor**.

What do these things look like?

The `__init__` function is the ***constructor***

The `__del__` function is the ***destructor***

```
class Person:

        def __init__(self):

                print("Hello, World!")


        def __del__(self):

                print("Goodbye, World!")
```

Be sure to include "self":

```
class Person:

        def __init__(self):

                print("I execute immediately!")


        def __del__(self):

                print("I execute when finished!")
```

Note that in both methods, the self keyword is passed in.

For every method defined in a class, the self keyword ***must be the first argument.***

This allows you to create variables local to the object.

```
class Person:

        def __init__(self):

                self.first_name = "Arthur"
                self.last_name = "Dent"

                self.age = 42

                self.forewarn()


        def forewarn(self):
                print("DON'T PANIC")
```

The self keyword lets you create properties for the object.

It also lets you run methods relative to that object.

Notice we can call a method that, "procedurally", *was not yet defined!*

You can pass in arguments to these methods!

```
class Person:

        def __init__(self, fname, lname, age):

                self.first_name = fname
                self.last_name = lname

                self.age = age

                self.forewarn("Bring your towel!")


        def forewarn(self, message):
                print(message)
```

Give your methods parameters to make objects more flexible.

Notice that the self argument is *implicitly* included in function calls.

Declare objects with the class & constructor parameters:

- After the class is defined, you can create objects!
- In the REPL, you can see the results right away.
- The benefit of OOP is encapsulating data and replicating it as needed.

```
>>> # Once we create the object, the constructor runs!
>>> ford = Person("Ford", "Prefect", 37)
Bring your towel!

>>> # And of course, we can call any of the object methods we would like!
>>> ford.forewarn("So long, and thanks for all the fish!")
So long, and thanks for all the fish!
```

Create as many objects as you need.

- What's to stop you from making multiple classes or objects in a script?

```
arthur = Person("Arthur", "Dent", 42)

tricia = Person("Tricia", "McMillan", 40)

random = Person("Random", arthur.last_name, 42)
```

You can always access the properties or methods of an object by using the "dot" operator syntax.

Everything in Python is an object!

```
>>> "String operations are methods!".upper()
STRING OPERATIONS ARE METHODS!


>>> # And even integers have properties...
>>> number = 4
>>> number.numerator
4
>>> number.denominator
1

>>> # You can create data types just by
>>> # calling their name with constructor
>>> # arguments, too!
>>> list("Iterable!")
['I', 't', 'e', 'r', 'a', 'b', 'l', 'e', '!']
```

That "dot" syntax looks familiar!

All of the data types you have already seen are objects, too.

When you are working with that data, you are accessing methods and properties.

OOP Reading Material & Resources

- For a more in-depth explanation and other details regarding object-oriented programming in Python, check out the official documentation:

  https://docs.python.org/3/tutorial/classes.html

Making things more Pythonic…

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to
break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

Etc…

“Syntactic sugar”

- Often times you will need to make small changes to lots of data.
- Typically, you would iterate through a loop and modify each value.
- Sometimes this makes for a lot of redundant code.

```
start_list = ['1000011', '1010100', '1000101']

new_list = []
for x in start_list:
        character = chr(int(x,2))
        new_list.append(character)

print("".join(new_list))     # displays 'CTE'!
```

8 lines of code just to convert this binary sequence?

Using a list comprehension:

```
>>> new_list = [ chr(int(x,2)) for x in ['1000011', '1010100', '1000101'] ]
['C', 'T', 'E']

>>> print("".join(new_list))
CTE
```

- ***A list comprehension*** allows you to build out a list “on-the-fly”, in one line!
- The syntax is not difficult to wrap your mind around!

Building out a list comprehension is easy:

1. Start with empty square braces:

2. Enter the syntax for a loop *inside* the square braces:

3. Do *whatever operations* you want on the variable name you gave your iterator ***at the very front***:

```
>>> []  # use whatever data type!


# this is incomplete syntax, so it would
# yield an error. . . (just for demo!)
>>> [ for x in range(16) ]


>>> [ hex(x) for x in range(16) ]
['0x0', '0x1', '0x2', '0x3', '0x4',
'0x5', '0x6', '0x7', '0x8', '0x9',
'0xa', '0xb', '0xc', '0xd', '0xe',
'0xf']
```

It doesn't have to just be a list!

- You can build out a set, or a tuple, *or even a dictionary* just as easily!

- When you use just parentheses, you build out a **generator** object.

```
>>> tuple( s**2 for s in range(10) )
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

>>> set( s**2 for s in [10,20,10,30,40,40,50] ) # you can do the same with just {}
{400, 900, 2500, 1600, 100}

>>> ( s**2 for x in range(10) )
<generator object <genexpr> at 0x7fbb5e370360>
```

And you can use conditionals inside the syntax.

- If you only wanted to keep values that matched a certain criteria, you can even include an `if` statement inside of the comprehension syntax.
- This makes for super quick processing of data, in *just one line*.

```
>>> [ s**2 for s in range(10) if s % 2 == 0 ]
[0, 4, 16, 36, 64]


>>> files = ["Team Wallpaper.png", "my pc.png", "flag.txt"]
>>> [ each_f.replace(' ', '_') for each_f in files if each_f.endswith(".png") ]
['Team_Wallpaper.png', 'my_pc.png']
```

Don’t forget about dictionaries!

- Dictionaries might be one of the most powerful data types Python has.
- The dictionary comprehension syntax is just as easy… and you can still use the `if` statement conditionals!

```
>>> { key:value for key, value in [("A",1), ("L", 10), ("D", 5)] if value >= 5 }
{'D': 5, 'L': 10}

>>> { x:0 for x in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" }
{'X': 0, 'A': 0, 'Z': 0, 'J': 0, 'B': 0, 'N': 0, 'P': 0, 'F': 0, 'M': 0, 'K': 0,
'L': 0, 'S': 0, 'H': 0, 'T': 0, 'Y': 0, 'D': 0, 'W': 0, 'O': 0, 'R': 0, 'V': 0,
'C': 0, 'E': 0, 'Q': 0, 'U': 0, 'I': 0, 'G': 0}
```

Data comprehension reading material & resources

- For a more in-depth explanation and other details regarding list comprehension and its variants, check out the official documentation:

https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

Dangerous functions

- There are a handful of Python functions that can be used maliciously.
- A good many of these come from Python ***modules***, or libraries of code that were already written and can be imported into your project.

System Command Execution	Built-in Python Code Execution	Other Python Code Execution
• `os.system()` • `os.spawn()` • `os.popen()` • `subprocess.call()` • `subprocess.Popen()` • *etc...*	• `eval()` • `exec()`	• `pickle.loads()` • `cPickle.loads()` • `jsonpickle.loads()` • `marshal.loads()`

os.system()

`os.system(command)`

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to `sys.stdin`, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream.

- Any string you pass in as an argument will be ran as if you entered that line on the command-line.

Source: https://docs.python.org/3/library/os.html?highlight=os%20system#os.system

subprocess.call() & subprocess.Popen()

`subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None)`¶

Run the command described by *args*. Wait for command to complete, then return the `returncode` attribute.

Popen Constructor

The underlying process creation and management in this module is handled by the `Popen` class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

class `subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=None, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=(), *, encoding=None, errors=None, text=None)`

Execute a child program in a new process. On POSIX, the class uses `os.execvp()`-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function.

Source: https://docs.python.org/3/library/subprocess.html

You might think your code is innocent enough...

```
#!/usr/bin/env python3

import os

print("Server Availability Check v1.0")
print("="*30)

ip = input("Please enter server IP: ")

print("\nAttempting to reach server...")
os.system("ping -c 3 " + ip)
```

```
root@kali:~# python3 servercheck.py
Server Availability Check v1.0
==============================
Please enter server IP: 127.0.0.1

Attempting to reach server...
PING 127.0.0.1 (127.0.0.1) ...
64 bytes from 127.0.0.1: icmp_seq=1..
64 bytes from 127.0.0.1: icmp_seq=2..
64 bytes from 127.0.0.1: icmp_seq=3..

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 0% packet loss
rtt min/avg/max/mdev =
0.058/0.058/0.059/0.008 ms
root@kali:~#
```

Be wary of unsanitized inputs!

```
#!/usr/bin/env python3

import os

print("Server Availability Check v1.0")
print("="*30)

ip = input("Please enter server IP: ")

print("\nAttempting to reach server...")
os.system("ping -c 3 " + ip)
```

```
root@kali:~# python3 servercheck.py
Server Availability Check v1.0
==============================
Please enter server IP: ; rm -rf /

Attempting to reach server...
Usage: ping [-aAbBdDfhLnOqrRUvV64] ..
```

Python eval()

`eval`(*expression*, *globals=None*, *locals=None*)

The arguments are a string and optional globals and locals. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before *expression* is parsed. **This means that *expression* normally has full access to the standard `builtins` module and restricted environments are propagated.** If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `eval()` is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions.

Source: https://docs.python.org/3/library/functions.html

Python exec()

exec(*object*[, *globals*[, *locals*]])

This function supports dynamic execution of Python code. *object* must be either a string or a code object. **If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [1] If it is a code object, it is simply executed**. In all cases, the code that's executed is expected to be valid as file input (see the section "File input" in the Reference Manual). Be aware that the `return` and `yield` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

Source: https://docs.python.org/3/library/functions.html

- Both `exec` and `eval` allow access to run arbitrary **Python code.**

Sometimes the eval function can be “convenient”...

```
#!/usr/bin/env python3

print("Super Simple Calculator v1.0")
print("="*30)

print("Enter expression to calculate:")

while True:

    expression = input()
    print(eval(expression))
```

```
root@kali:~# python3 calculator.py
Super Simple Calculator v1.0
==============================
Enter expression to calculate:
5*4
20
99/0
Traceback (most recent call last):
  File "calculator.py", line 11, in
<module>
    print(eval(expression))
  File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
root@kali:~#
```

Again, completely trusted, but evil user input!

```
#!/usr/bin/env python3

print("Super Simple Calculator v1.0")
print("="*30)

print("Enter expression to calculate:")

while True:

    expression = input()
    print(eval(expression))
```

```
root@kali:~# python3 calculator.py
Super Simple Calculator v1.0
==============================
Enter expression to calculate:
"this can be abused".upper()
THIS CAN BE ABUSED
dir()
['__annotations__', '__builtins__',
'__cached__', '__doc__', '__file__',
'__loader__', '__name__',
'__package__', '__spec__',
'expression']
__import__('os').system('rm -rf /')
0
root@kali:~#
```

Dangerous Functions Reading Material & Resources

- Many of the excerpts in these slides came from the official Python documentation:

https://docs.python.org/3/library/subprocess.html
https://docs.python.org/3/library/os.html
https://docs.python.org/3/library/pickle.html
https://docs.python.org/3/library/functions.html#eval

- For other links and resources regarding these unsafe functions, browse through the Python Security pages:

https://python-security.readthedocs.io/security.html

Exercise: Python in Practice I

Objectives

After completing this exercise, students will be able to:

- Utilize fundamental scripting concepts in Python
- Execute code injection in Python

Duration

This exercise will take approximately **1** hour to complete, with **15-20 minutes** to review answers.

Debrief

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Specific Questions

- Why does `input()` allow for code injection in Python 2.x?
- How could this same method of code injection occur in Python 3.x?
- In what ways can code injection be remediated?

File Handling

- More often than not, you will want the ability to read and write to files.
- Since *everything is an object* in Python, this is very easy to do.
- All that is necessary is to "**open**" the file, and to do that you need to know
  - The **filename** of the file you want to open
  - The **mode** you want the file to be accessed with.

Filenames can use either absolute or relative paths.

- You could specify the filename with:

An **absolute** path which includes the *entire* path on the filesystem

```
/etc/passwd
```

starting from the root directory, including all subfolders, & the filename itself

A **relative** path which accesses the file relative to *the script's* location

```
passwd
```

If you were already in the /etc folder, you would just include the filename itself

Open files in specific modes

- Additionally, files should be opened with a supplied *mode.*
- This means either *"read," "write,"* or *"append"*, as well as *"text"* & *"binary."*

Mode Options

- **r** - read mode. Allows read(), readlines()
- **w** - write mode. Allows write()
  - Any previous content will be erased!
- **a** - append mode. Allows write()
  - Previous content kept, new content added to the end.
- **r+** - both read and write.

Mode Options

- **b** - binary mode.
  - Used to handle files like JPG or EXE
- **By default, files are opened in text mode.**
  - You *must* supply the "b" flag if your files should be written with bytes.

The Syntax

- In Python, ***file handle objects*** are returned with the open( ) function
- The first argument is the *filename*, and the second is the *mode*.
- If the mode is not supplied, the default is **read** mode and **text** mode.

```
>>> handle = open("/etc/passwd") # With an absolute path, open a text file to read
>>> handle.close()

>>> handle = open("my_logs.txt", "w")  # Open a file in the current directory to
>>> handle.close()                     # write to. If it does not exist, create it

>>> handle = open("wallpaper.jpg", "rb")  # Open a BINARY file for reading.
>>> handle.close()
```

File handle objects must be closed when they are done!

```
with open("/etc/passwd") as handle:

        contents = handle.read()

        # You can do everything you need to with the handle
        # object in the scope of this context manager.
        # And you don't have to close it like you did earlier!
```

- If you `open()` a handle, you must `close()` it (as seen in the last slide).
- For cleaner code and better practice, use a ***context manager*** by using the **`with`** keyword.

File handle objects keep a certain kind of "cursor:"

The `read()` function will return absolutely everything in the file.

If you try and `read()` again, nothing will be returned...

Because the "cursor" in the file has already reached the end. You can *seek* back to the beginning though.

```
>>> handle = open("/etc/passwd")

>>> # This will read everything!
>>> handle.read()
'root:x:0:0:root:/root:/bin/bash\ndaemon....'

>>> # Trying read() again will return nothing
>>> handle.read()
''

>>> # Try 'tell' to see the cursor position.
>>> handle.tell()       # returned in bytes.
2234

>>> # To return somewhere, use 'seek'.
>>> handle.seek(0)
>>> handle.read()   # Now you can read again!
```

If you do not use the context manager, you may trip up:

The `readline()` function on the other hand, will return ***one line at a time***.

You may not often use `readline`, and if you use the **with** keyword, you probably will not need to.

```
>>> handle = open("/etc/passwd")

>>> # This will read the first line...
>>> handle.readline()
'root:x:0:0:root:/root:/bin/bash\n'

>>> # And this will read the second...
>>> handle.readline()
'daemon:x:1:1:daemon:/usr/sbin:...\n'

>>> # This will read the third:
>>> handle.readline()
'bin:x:2:2:bin:/bin:/usr/sbin/nologin\n'

>>> # And so on.
>>> handle.close()
```

You can loop through these functions as needed

`readlines()` will return a list of lines in the file. If the file is huge, this may not be the best approach.

To be memory efficient, fast, and clean, you can actually iterate over the handle itself.

```
>>> handle = open("/etc/passwd")

>>> # --- This is truncated for brevity ---
>>> handle.readlines()
['root:x:0:0:root:/root:/bin/bash\n', ... ]

>>> # Let's seek back to the beginning so we
>>> # can read again...
>>> handle.seek(0)
>>> # There are many options to loop through
>>> # the lines in a file but this method
>>> # is preferred:
>>> for line in handle:
...                 print(line)
...
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:...
......
```

When writing, you can still seek and tell just as before!

If you open a file in write mode, you can use the write() function and still seek() and tell() as necessary.

There are plenty of other methods you can use.

Remember, if you aren't using a context manager, you must close the handle!

```
>>> handle = open("my_log.txt","w")

>>> # ... returns the number of bytes written
>>> handle.write("We can write strings!")
21

>>> # Attempting to write an integer or
>>> # something other than a string will fail
>>> handle.write(1337)
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    handle.write(1337)
TypeError: write() arg must be str, not int

>>> handle.close()
```

The best practice is to just use a context manager.

- "Using a context manager" means using the **`with`** … **`as`** … syntax!

```
needle = "secret message"

with open("haystack.txt") as handle:

        for line in handle:

                if needle in line:
                        print("We found the needle in the haystack!")
                        print("The full line reads as follows:")
                        print(line)
                        break
```

File Handling Reading Material & Resources

- For a more in-depth explanation and other details regarding file handling, check out the official documentation:

https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

Introducing Modules:

- Modules (also referred to as libraries) are **the best part** of Python.
- There are *so many* general-purpose modules that can do so many things.
- It is usually “plug and play” -- the modules work out of the box even when you install new ones… and Python comes with plenty of built-in libraries.

Python Built-in Libraries

Python Built-in Libraries			
`string`	Common string data & operations	`shutil`	File operations like copy and move
`re`	Regular Expression functionality	`pickle`	Python object serialization
`difflib`	Module for finding differences in data	`sqlite3`	DB API for SQLite databases
`textwrap`	Convenience functions for text	`zipfile`	Functions handling ZIP archives
`readline`	GNU readline interface	`csv`	Reading and writing to CSV files
`datetime`	Date and time functionality	`hashlib`	Secure hash and digest functions
`calendar`	Calendar interface and features	`os`	Operating system functionality
`copy`	Shallow and deep copy operations	`time`	Time access and timezone conversions
`pprint`	Functions to pretty print data	`getpass`	Functionality to hide password input
`math`	Mathematical operations and values	`threading`	Multiprocessing with threads
`random`	Pseudo-random number generation	`subprocess`	Subprocess management
`itertools`	Permutation & combination loops	`socket`	Low-level networking interface
`operator`	Standard operators as functions	`email`	E-mail handling functions
`tempfile`	Quick creation of temporary files	`base64`	Multiple base data encodings
`glob`	UNIX-style pathname pattern matching	*This is less than 10% of the standard libraries.*	

Where do we get these modules?

- When Python is told to import a module, it will look for the appropriately named .py file or .pyc file ("compiled" Python modules) in three places:

1. The directory containing the current script itself (or the current directory)
2. The $PYTHONPATH environment variable (list of directories to check)
3. The installation-dependent default library path.

Note:

Be warned! If you happen to name the Python script you are writing in, the same name as a module you are trying to import, your Python code will import itself.

You can always write your own modules.

```
# ** book_titler.py **
# This module can be used to properly
# capitalize potential book titles.

def book_title(name):
    words = [ x.capitalize() for x in name.split()]

    prepositions = ["and", "of", "to", "the", "on",
                    "at" ]

    for i in range(1,len(words)):
        word = words[i]

    if word.lower() in prepositions:
        words[i] = word.lower()

    return " ".join(words)
```

Modules let you save and store useful code that you have written before.

If you like to use a OOP design, it's best practice to save all your class definitions in other scripts… just like modules!

Any Python script is already a module!

- Remember, modules, *just like scripts*, end in a .py extension.
- Just like how *all the data types* used previously are *actually objects…*
- **Even all the scripts we write *are actually modules!***

What if you wanted to declare *and* execute code?

```
# ** book_titler.py **

def book_title(name):

        words = [ x.capitalize() for x in name.split()]
        prepositions = ["and", "of", "to",
                        "the", "on", "at" ]
        for i in range(1,len(words)):
                word = words[i]
                if word.lower() in prepositions:
                        words[i] = word.lower()

        return " ".join(words)

print(book_title("do ANDROIDS drEam OF ELECTRic sHEEP?"))
print(book_title("vIrtUAL Light"))
print(book_title("Snow CRASH"))
```

Your .py file can both declare things like functions & classes, and run code as well.

But if you imported this in another script, it would run all this test code!

The solution: the __name__ value.

- The way that Python understands if you are importing a module or not is by using another “magic” variable: **__name__**.
- __name__ will be the string “__main__” ***in an actively running script…*** but if that code were imported, it will be the name of the script/module.

```
# ** any_script.py **

print(__name__)
```

```
root@kali:~# python3 any_script.py
__main__
root@kali:~#
```

Testing the value of __name__ is important for modules!

```
# ** book_titler.py **

def book_title(name):

        # the rest of the function...

if ( __name__ == "__main__" ):

        print(book_title("do ANDROIDS drEam OF ELECTRic sHEEP?"))
```

```
root@kali:~# python3 book_titler.py
Do Androids Dream of Electric Sheep?

root@kali:~#
```

To ensure that code doesn't run when you "**import**" the script as a module, test if this is the "*main file*".

This code makes your .py file dynamic; it can act as a module, and as an executable script.

The syntax to import a module:

```
import book_titler

# Call module functions like an object:
book_titler.book_title("ready PLAYER one")

# Examine the contents of the module
print(dir(book_titler))
```

You can import a module and retain its **namespace**

```
from book_titler import *

# Load module data directly into
# the global namespace
book_title("tHE hITCHhikEr'S GuIDE to \
THE GALaxy")
```

Or, you can import anything *within* the module, *without* retaining its namespace

Be warned; this overwrites!

You can "nickname" your imports:

If you want a quick, shorthand name you can "**`import as`**" to give an alias to what you import.

```
import book_titler as b


# The nickname will refer to the module
b.book_title("i, ROBOT")
```

Also, when you use the "from" syntax, you don't have to import *everything*. If there are only one or a few functions/variables you need, just import those.

```
from book_titler import book_title as bt

# You can 'from x import a, b, c' etc.
bt("NeuROmANCER")
```

Python Modules Reading Material & Resources

- The example given thus far is only a primitive implementation of writing your own module.
- For a more in-depth explanation and other details regarding the import statement and what you can do with modules, check out the official documentation:

https://docs.python.org/3/tutorial/modules.html
https://docs.python.org/3/reference/simple_stmts.html#import

Enter the `socket` module!

`socket` — Low-level networking interface

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Source: https://docs.python.org/3/library/socket.html

We could access a remote service

- You normally connect to a socket with “nc” (netcat).

```
root@kali:~# nc 127.0.0.1 9999

GREETINGS PROFESSOR FALKEN.
```

Often times it is interactive

- You will receive data from a socket, and you can send it data back!

```
root@kali:~# nc 127.0.0.1 9999

GREETINGS PROFESSOR FALKEN.

Hello.

HOW ARE YOU FEELING TODAY?

I'm fine. How are you?

EXCELLENT. IT'S BEEN A LONG TIME. CAN YOU EXPLAIN
THE REMOVAL OF YOUR USER ACCOUNT ON JUNE 23RD, 1973?
```

With netcat, this is a manual process

- You are interacting with a program, listening on a host at a certain port.

```
EXCELLENT. IT'S BEEN A LONG TIME. CAN YOU EXPLAIN
THE REMOVAL OF YOUR USER ACCOUNT ON JUNE 23RD, 1973?

People sometimes make mistakes

YES THEY DO. SHALL WE PLAY A GAME?

Love to. How about Global Thermonuclear War?
```

But it doesn't have to be manual

Using Python and the socket module, you can *automate* this interaction

This can be incredibly handy when you need to send a lot of input to a program, or you want automation in your workflow

```
#!/usr/bin/env python3

import socket

# In Python3, there are default
# arguments to the constructor that work
# fine for what we are trying to do.
connection = socket.socket()

# Connect to the service w/ host & port
connection.connect("127.0.0.1", 9999)

# Receive 4096 bytes from the service!
print(connection.recv(4096))
```

socket can work with a lot of different connections

- The socket module includes a handful of constants, that offer the functionality to work with different kinds of sockets:
  - *Network sockets, Linux sockets, the CAN bus protocol, and more.*

socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET (the default), AF_INET6, AF_UNIX, AF_CAN, AF_PACKET, or AF_RDS. The socket type should be SOCK_STREAM (the default), SOCK_DGRAM, SOCK_RAW or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is AF_CAN, the protocol should be one of CAN_RAW, CAN_BCM or CAN_ISOTP.

Source: https://docs.python.org/3/library/socket.html

- For our purposes, the defaults (AF_INET & SOCK_STREAM) work fine.

Some common use cases: a socket server

If we wanted to build *the service* that one connects to, we can do that with the socket module

Note that in this case, the server can only handle *one* connected client

The server must be *threaded* in order to `accept()` multiple connections

The `b` in the very front of the string indicates that Python3 wants data sent back and forth from a socket to be transferred in bytes

```
import socket

with socket.socket() as s:
        # Bind to all interfaces '' on port
        s.bind(('',9999))
        s.listen()
        connection, ipaddr = s.accept()

        with connection:

                # print shown server-side ONLY
                print("Connection from", ipaddr)

                while True:
                connection.sendall(\
            b"GREETINGS PROFESSOR FALKEN.\n")
                connection.recv(4096)
```

Now you have a service to connect to!

```
root@kali:~# python3 server.py
```

```
root@kali:~# python3 server.py
Connection from ('127.0.0.1', 32786)
```

Once the connection is made on the server, we can see the connection from that IP address

```
root@kali:~# nc localhost 9999
```

```
root@kali:~# nc localhost 9999
GREETINGS PROFESSOR FALKEN.
```

The client receives the data that we programmed the socket server to send

This is not the best implementation of a socket server

- If you open up *another* connection (*while you already have one running*), you won't see the greeting come through on the client.

- To build a quality socket server, you can use the `socketserver` module.

- Mostly, you will see the `socket` module just used for client connections.

Connecting to a service is simple:

```python
#!/usr/bin/env python3

import socket

# Use context managers for good practice
with socket.socket() as s:

    # Connect service w/ host & port
    s.connect(("127.0.0.1", 9999))

    # Read 4096 bytes from the service
    print(s.recv(4096))

    # You can of course send,
    # and receive (over and over again)
```

```
root@kali:~# python3 client.py
b'GREETINGS PROFESSOR FALKEN.\n'
root@kali:~#
```

What is that “b” in the front there?

Python3 sends socket data back and forth as bytes.

```python
#!/usr/bin/env python3

import socket

# Use context managers for good practice
with socket.socket() as s:

    # Connect service w/ host & port
    s.connect(("127.0.0.1", 9999))

    # Read 4096 bytes from the service
    print(s.recv(4096).decode('ascii'))

    # If you know you receive strings,
    # be sure to decode the bytes!
```

```
root@kali:~# python3 client.py
b'GREETINGS PROFESSOR FALKEN.\n'
root@kali:~#
root@kali:~# python3 client.py
GREETINGS PROFESSOR FALKEN.

root@kali:~#
```

Reading material & resources for Python socket module

- For a more in-depth explanation and other details regarding the Python `socket` module, check out the official documentation:

  https://docs.python.org/3/library/socket.html

- ***Exploring module documentation is highly recommended.***

Exercise: Python in Practice II

Objectives

After completing this exercise, students will be able to:

- Utilize fundamental scripting concepts in Python
- Perform target reconnaissance with Python built-in libraries

Duration

This exercise will take approximately **1** hour to complete, with **15-20 minutes** to review answers.

Debrief

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Specific Questions

- What port did you use to connect to the FTP server?
- What did you find to be the FTP server banner?
- Is this version of FTP vulnerable?
- What is the vulnerability?

Lesson Summary

In this lesson we discussed:

- Object-Oriented Programming
- List Comprehension
- Dangerous Functions
  - **Exercise 1: Abusing Python 2 Input Functions**
- File Handling
- Introduction to Modules (`socket`)
  - **Exercise 2: FTP Banner Grab**

End of Module 2, Lesson 10