Week Three

CSOH| Python Study Group

Cloud Security Office Hours

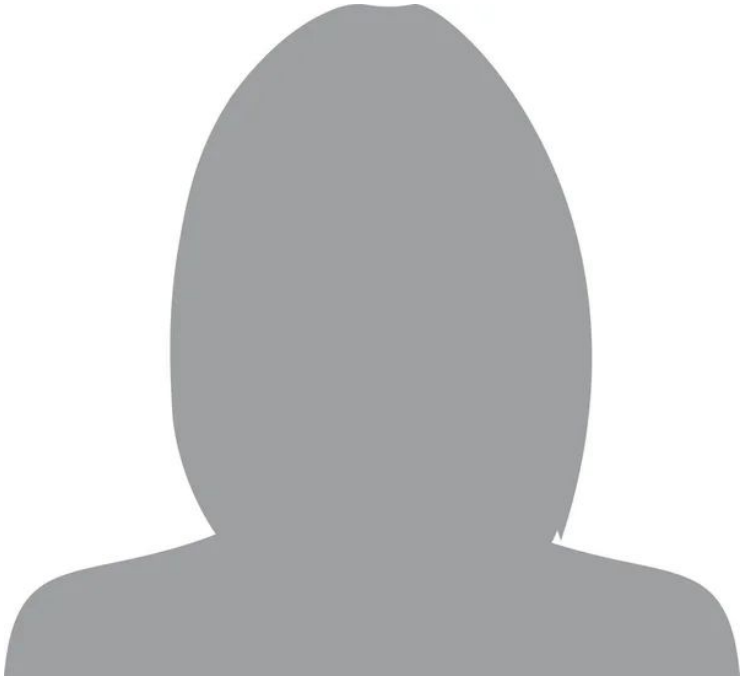
Thanks for the space!

Once a week, every Friday, we host a Zoom call that is an open forum for novices and experts in Cloud Security to collaborate and encourage each other in this field.

Fridays @ 10:00 AM
Same Zoom

https://csoh.org/

`id`



i am D (they/them) or Hetz

recent cybersecurity grad

interested in malware analysis and application security

not an expert, just an enthusiast tired of tutorial hell

and chihuahuas…lots of chihuahuas

Grounding Assumptions

1. **We're all here to learn.**
   Everyone starts somewhere, it's okay to ask questions and make mistakes. This is a space for us to work and learn together.

2. **No prior coding experience required.**
   We'll explain each project and ask our advanced group members for support, when needed.

3. **Python is forgiving.**
   Small errors happen! Syntax mistakes are part of learning how code "thinks."

4. **We'll focus on understanding, not memorizing.**
   Knowing *why* something works is more valuable than remembering every detail.

5. **Try it yourself.**
   The best way to learn coding is by typing, testing, and experimenting.

6. **Respectful collaboration.**
   Share ideas, help each other, and be patient- we're learning together.

7. **Keep it simple today.**
   We'll start with the basics and build up over time — no need to rush.

Schedule

1. Codespace Review (10 min)
2. Lesson Review (25 min)
3. Mini Project: Input & Output (30 min)
4. Show & Tell (25 min)

Focus & Objectives

Focus:

Learn how to use lists and dictionaries to structure and analyze data. Use loops to process mock event logs and count occurrences of event names - like a basic threat-hunting task.

Contributers

Special Thanks!

Kyle Ingersoll - Managing the Github Workspace

Sam M. - Recording each session

Neil - Wisdom and admin powers

https://csoh.org/

Data Structures

Built-in data types in Python used to store collections of data.

Tuple `mytuple = ("apple", "banana", "cherry")`	- used to store multiple items in a single variable - ordered and unchangeable
Set `myset = {"apple", "banana", "cherry"}`	- used to store multiple items in a single variable. - *unordered*, *unchangeable**, and *unindexed*. Set *items* are unchangeable, but you can remove items and add new items.

Dictionaries
`thisdict = {`
`  "brand": "Ford",`
`  "model": "Mustang",`
`  "year": 1964`
`}`
`print(thisdict)`
	- used to store data values in key:value pairs
	- a collection which is ordered*, changeable and do not allow duplicates
	As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

List `mylist = ["apple", "banana", "cherry"]`	- used to store multiple items in a single variable - Lists are created using square brackets

Lists & Dictionaries

Lists: Ordered collections used to store multiple values

```
mylist = ["apple", "banana", "cherry"]
```

Dictionaries: Key-value pairs

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

W3 Dictionaries

Lists

Items are indexed in lists:

[0] refers to the first element and [1] refers to the second element.

[-1] refers to the last element.
[-2] refers to the second-to-last element.

When we say that lists are **ordered**, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

- We can change, add, and remove items in a list after it has been created.
- Duplicates are allowed.

W3 Schools

List Methods

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']

fruits.count('apple') #Count number of values
2

fruits.index('banana') # Find first position that 'banana' exists
3
fruits.index('banana', 4)  # Find next banana starting at position 4
6
fruits.reverse() # Reverse the list values
fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']

fruits.append('grape') # Add a list item (to the end of list)
fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']

fruits.sort() # Sort alphabetically or in an ordered fashion
fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']

fruits.pop() # To retrieve an item from the top of the stack
'pear'
```

Dictionaries

Dictionary items are:

- ordered, changeable, and do not allow duplicates.
- presented in key:value pairs
- referred to by using the key name.

Ordered items have a defined order, and that order will not change.

Unordered means that the items do not have a defined order, you cannot refer to an item by using an index.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

- We can change, add or remove items after the dictionary has been created.
- Dictionaries cannot have two items with the same key
- Dictionaries can be of any data type (string, int, boolean etc.)

W3 Dictionaries

For Loops

Real Python

for loops are mostly used to iterate a *known* number of times, which is common when you're processing data collections with a specific number of data items.

- `for` is the keyword that initiates the loop header.
- `variable` is a variable that holds the current item in the input iterable.
- `in` is a keyword that connects the loop variable with the iterable.
- `iterable` is a data collection that can be iterated over.
- `<body>` consists of one or more statements to execute in each iteration.

reading line by line

```
for line in open("firewall.log"):
    print(line)
```

Output:

```
ALLOW 192.168.1.10 → 8.8.8.8

DENY 10.0.0.5 → 172.16.0.1

ALLOW 192.168.1.15 → 8.8.4.4
```

For Loops

Real Python

The break statement immediately exits the loop and jumps to the first statement after the loop.

The continue statement terminates the current iteration and proceeds to the next one.

For example, if you have a list of numbers and only want to process the even ones, you can use a continue statement to skip the odd numbers:

```
>>> numbers = [1, 2, 3, 4, 5, 6]

>>> for number in numbers:
...     print(f"{number = }")
...     if number % 2 != 0:
...         continue
...     print(f"{number} is even!")
...
number = 1
number = 2
2 is even!
number = 3
number = 4
4 is even!
number = 5
number = 6
6 is even!
```

Range Function

The built-in **range()** function returns an immutable sequence of numbers, commonly used for looping a specific number of times.

```
# Print all items by referring to their index number:
  thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

Create a sequence of numbers from 0 to 5, and print each item in the sequence:

```
# Create a sequence of numbers from 0 to 5, and print each item in the sequence
  x = range(6)
for n in x:
  print(n)
```

RangeFunction

Range Function

```
# Generates a sequence from 0 up to (but not including) 10, in steps of 2
numbers = range(0, 10, 2)
# The numbers generated are 0, 2, 4, 6, 8
print(list(numbers))
```

Syntax

`range(start, stop, step)`

Parameter Values

Parameter	Description
start	Optional. An integer number specifying at which position to start. Default is 0
stop	Required. An integer number specifying at which position to stop (not included).
step	Optional. An integer number specifying the incrementation. Default is 1

RangeFunction

Loops & Lists

Access both the index and the value simultaneously, especially for finding all occurrences of an item.

```
names = ["Alice", "Bob", "Charlie", "Bob"]
all_bobs = [i for i, name in enumerate(names) if name == "Bob"]
print(f"All Bob indices: {all_bobs}") # Output: All Bob indices: [1, 3]
```

You can also loop through the list items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable

```
# Loop through items in a list

thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)

# Print all items by referring to their index number:
  thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

Dictionaries & Lists

Python3- Dictionaries

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the **items()** method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Threat Hunting

Dictionaries:

You can access values using keys like event["user"].
Great for representing structured logs or JSON-like data.

```
# Example

  event = {
    "eventName": "login",
    "user": "admin",
    "sourceIP": "192.168.1.10",
    "status": "SUCCESS"
}

print("User:", event["user"])
print("Source IP:", event["sourceIP"])
```

Event logs often arrive as JSON, which Python naturally converts to dictionaries using the `json` module.

Threat Hunting

Github

```python
# get csv file from command line using argparse
parser = argparse.ArgumentParser()
parser.add_argument("csv_file", help="CSV file (needs to be in same directory)")
args = parser.parse_args()
csv_file = args.csv_file
# open csv file
with open(csv_file, 'r') as file:
    # create CSV reader object
    reader = csv.reader(file)
    # create a list to critical vulns (9+ cvss)
    critical_vulns = []
    # loop through rows
    for row in reader:
        # skip header row
        if row[4] == "CVSS":
            critical_vulns.append(row)
            continue
        # check cvss score
        if float(row[4]) >= 9.0:
            critical_vulns.append(row)
# create new CSV file containing only critical critical
filename = '[CRITICAL] ' + csv_file
with open(filename, 'w', newline='') as file:
    # create CSV writer object
    writer = csv.writer(file)
    # write critical vulns to file
    writer.writerows(critical_vulns)
print(f"Finished writing {len(critical_vulns) - 1} to {filename}")
```

Threat Hunting

Lists Example:IOC Matching
This simple list comparison forms the basis for many more complex, automated threat hunting scripts.

Resource:

Awesome Threat Detection

```python
# A simple Python list of malicious file hashes (IOCs)
known_malicious_hashes = [
    "a1b2c3d4e5f67890abcdef1234567890",
    "f6e5d4c3b2a10987654321abcdef01234",
    # ... more hashes from threat intelligence feeds
]

# A hash observed in your environment
observed_hash = "f6e5d4c3b2a10987654321abcdef01234"

# Check if the observed hash is in the list
if observed_hash in known_malicious_hashes:
    print(f"Potential threat detected: {observed_hash} is a known malicious hash.")
else:
    print(f"Hash {observed_hash} is not in the known malicious list.")
```

Projects (Suggestions)

- Loop through a list of fake event dictionaries and count how many times each eventName appears.
- Simple IOC Hunter
- Import CSV/JSON and filter information
- URL Reputation Checker

Optional Challenges

1. Use Pandas
2. Create functions
3. Use Regex filtering
4. Nested Loops

Share your work, if you'd like!