# Activity_Create more functions

April 9, 2024

# 1 Activity: Create more functions

## 1.1 Introduction

Built-in functions are functions that exist within Python and can be called directly. They help analysts efficiently complete tasks. Python also supports user-defined functions. These are functions that analysts write for their specific needs.

For example, patterns in login attempts could reveal suspicious activity. Python functions can help analysts work efficiently with lists of login attempts. Both built-in functions and user-defined functions in Python can help security analysts analyze login attempts.

## 1.2 Scenario

Working as a security analyst, I'm responsible for working with a list that contains the number of failed attempts that occurred each month. I'll identify any patterns that might indicate malicious activity. I'm also responsible for defining a function that compares the logins for the current day to an average and improving it by adding a `return` statement.

## 1.3 Task

I am provided with a list of the number of failed login attempts per month, as follows:

`119`, `101`, `99`, `91`, `92`, `105`, `108`, `85`, `88`, `90`, `264`, and `223`.

This list is organized in chronological order of months (January, February, March, April, May, June, July, August, September, October, November, and December).

This list is stored in a variable named `failed_login_list`.

In this task, I'll use a built-in Python function to order the list. I'll pass the call to the function that sorts the list directly into the `print()` function. This will allow me to display and examine the result.

[1]:
```
# Assign `failed_login_list` to the list of the number of failed login attempts␣
 ↪per month

failed_login_list = [119, 101, 99, 91, 92, 105, 108, 85, 88, 90, 264, 223]
```

```
# Sort `failed_login_list` in ascending numerical order and display the result

print(sorted(failed_login_list))
```

```
[85, 88, 90, 91, 92, 99, 101, 105, 108, 119, 223, 264]
```

To order the `failed_login_list` in ascending numerical order, I used the `sorted()` function. This is a built-in Python function that takes in a list, sorts its components, and returns the result.

## 1.4 Task

Now, I'll want to isolate the highest number of failed login attempts so I can later investigate information about the month when that highest value occurred. I'll use the function that returns the largest numeric element from a list. Then, pass this function into the `print()` function to display the result. This will allow me to determine which month to investigate further.

```
[2]: # Assign `failed_login_list` to the list of the number of failed login attempts␣
      ↪per month

failed_login_list = [119, 101, 99, 91, 92, 105, 108, 85, 88, 90, 264, 223]

# Determine the highest number of failed login attempts from␣
 ↪`failed_login_list` and display the result

print(max(failed_login_list))
```

```
264
```

To determine the highest number of failed login attempts from `failed_login_list`, I used the `max()` function. This is a built-in Python function that takes in a sequence, identifies the maximum value from the sequence and returns the result.

## 1.5 Task

Now I'll first define a function that displays a message about how many login attempts a user has made that day. I'll define a function named `analyze_logins()` that takes in two parameters, `username` and `current_day_logins`. Every time this function is called, it should display a message about the number of login attempts the user has made that day.

```
[4]: # Define a function named `analyze_logins()` that takes in two parameters,␣
      ↪`username` and `current_day_logins`

def analyze_logins(username, current_day_logins):

    # Display a message about how many login attempts the user has made that day
    print("Current day login total for", username, "is", current_day_logins)
```

### 1.6 Task

I will now call the function and add arguments for the two parameters(username and current_day_logins).

[6]:
```
# Define a function named `analyze_logins()` that takes in two parameters,␣
↪`username` and `current_day_logins`

def analyze_logins(username, current_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

# Call `analyze_logins()`

analyze_logins("dabaly", 15)
```

```
Current day login total for dabaly is 15
```

### 1.7 Task

Now, I'll need to expand this function so that it also provides the average number of login attempts made by the user on that day. Doing this will require incorporating a third parameter into the function definition.

In this task, I'll add a parameter called `average_day_logins`. The code will use this parameter to display an additional message. The additional message will convey the average login attemps made by the user on that day. Then, call the function with the same first and second arguments.

[7]:
```
# Define a function named `analyze_logins()` that takes in three parameters,␣
↪`username`, `current_day_logins`, and `average_day_logins`

def analyze_logins(username, current_day_logins, average_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has␣
↪made that day
```

```
    print("Average logins per day for", username, "is", average_day_logins)

# Call `analyze_logins()`

analyze_logins("dabaly", 15, 10)
```

```
Current day login total for dabaly is 15
Average logins per day for dabaly is 10
```

## 1.8 Task

In this task, I'll further expand the function. Include a calculation to get the ratio of the logins made on the current day to the logins made on an average day. Store this in a new variable named `login_ratio`. The function displays an additional message that uses this variable.

*Note that if `average_day_logins` is equal to `0`, then dividing `current_day_logins` by `average_day_logins` will cause an error. Due to the error, Python will display the following message: `ZeroDivisionError: division by zero`. For this activity, I assume that all users will have logged in at least once before. This means that their `average_day_logins` will be greater than `0`, and the function will not involve dividing by zero.

```
[8]: # Define a function named `analyze_logins()` that takes in three parameters,␣
 ↪`username`, `current_day_logins`, and `average_day_logins`

def analyze_logins(username, current_day_logins, average_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has␣
 ↪made that day

    print("Average logins per day for", username, "is", average_day_logins)

    # Calculate the ratio of the logins made on the current day to the logins␣
 ↪made on an average day, storing in a variable named `login_ratio`

    login_ratio = current_day_logins / average_day_logins

    # Display a message about the ratio

    print(username, "logged in", login_ratio, "times as much as they do on an␣
 ↪average day.")

# Call `analyze_logins()`
```

```
analyze_logins("dabaly", 15, 10)
```

```
Current day login total for dabaly is 15
Average logins per day for dabaly is 10
dabaly logged in 1.5 times as much as they do on an average day.
```

## 1.9 Task

I'll continue working with the `analyze_logins()` function and add a return statement to it. Return statements allow one to send information back to the function call.

In this task, I'll use the `return` keyword to output the `login_ratio` from the function, so that it can be used later.

I'll call the function with the same arguments used in the previous task and store the output from the function call in a variable named `login_analysis`. I'll then use a `print()` statement to display the saved information.

[14]:
```
# Define a function named `analyze_logins()` that takes in three parameters,␣
↪`username`, `current_day_logins`, and `average_day_logins`

def analyze_logins(username, current_day_logins, average_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has␣
↪made that day

    print("Average logins per day for", username, "is", average_day_logins)

    # Calculate the ratio of the logins made on the current day to the logins␣
↪made on an average day, storing in a variable named `login_ratio`

    login_ratio = current_day_logins / average_day_logins

    # Return the ratio

    return login_ratio

# Call `analyze_logins() and store the output in a variable named␣
↪`login_analysis`

login_analysis = analyze_logins("dabaly", 15, 10)
```

```
# Display a message about the `login_analysis`

print("dabaly", "logged in", login_analysis, "times as much as they do on an␣
↪average day.")
```

```
Current day login total for dabaly is 15
Average logins per day for dabaly is 10
dabaly logged in 1.5 times as much as they do on an average day.
```

### 1.10 Task

In this task, I'll use the value of `login_analysis` in a conditional statement. When the value of `login_analysis` is greater than or equal to 3, then the login activity will require further investigation, and an alert will be displayed.

```
[17]: # Define a function named `analyze_logins()` that takes in three parameters,␣
↪`username`, `current_day_logins`, and `average_day_logins`

def analyze_logins(username, current_day_logins, average_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has␣
↪made that day

    print("Average logins per day for", username, "is", average_day_logins)

    # Calculate the ratio of the logins made on the current day to the logins␣
↪made on an average day, storing in a variable named `login_ratio`

    login_ratio = current_day_logins / average_day_logins

    # Return the ratio

    return login_ratio

# Call `analyze_logins() and store the output in a variable named␣
↪`login_analysis`

login_analysis = analyze_logins("dabaly", 100, 10)

# Conditional statement that displays an alert about the login activity if it's␣
↪more than normal
```

```
if login_analysis >= 3:
    print("Alert! This account has more login activity than normal.")
```

```
Current day login total for dabaly is 100
Average logins per day for dabaly is 10
Alert! This account has more login activity than normal.
```

I called the updated `analyze_logins()` function and passed in `"dabaly"`, `100`, and `10` as the three arguments, in that order. This gives back a login_ratio of more than 3 hence the alert.

### 1.11 Conclusion

**What are your key takeaways from this lab?**

[- There are a variety of ways a function can be written. - It can be written to display information to the screen, or return information that can then be saved in a variable. - Also it can be written to take in any number of parameters, use the parameters to execute a series of tasks, and then return a result. - The `sorted()` function in Python is a built-in function that helps you sort the components of a list. - The `max()` function in Python is a built-in function that helps you identify the element with the maximum value in a list. - The `print()` function in Python is a built-in function that helps display information. It can also be used to directly display the output from another function call. - To display the output from another function call, one must place it inside a `print()` statement..]