Execution Flow

with python3

TheMalteseSailor

Outline

1. Introduction
2. Conditional Statements
3. Loops
4. Scope

1. Introduction

Introduction

- What is the purpose of computer programming?
  - All computer programs essentially do 3 things:
    - Get data input
    - Process that data
    - Output that data somewhere
- Why use python to explain these topics?
  - Python3 is an easy-to-understand and follow language.
- What is scope?
  - We'll get to that later.

Introduction

- Getting input: (examples)
- 1. file input

```
raw_file_contents = “”
with open(“LatLong.csv”, ‘r’) as _f:
        raw_file_contents = _f.read()

binary_file_contents = b”
with open(“shellcode.bin”, ‘rb’) as_f:
       binary_file_cotents = _f.read()
```

- 2. user input

```
user_input = “”
user_input = input(“>> ”)
```

```
# The variable is created here because if not when you assign it a value
# within the scope of the subsequent “with open” call the variable
# will cease to exist once you exit the scope of that call.
# In python scope is easier to assess using the tabs/spaces
```

```
# I’m instantiating this variable here because typically user inputs are
# obtain from within a loop. this means that the value should be cleared
# after / at the beginning of each loop iteration.
```

Introduction

- Data Processing: (examples)
- 1. Data manipulation

```
split_data = raw_file_contents.split(“,”)      # raw_file_contents is of type “string”
                                               # split_data is of type “list” populated with strings.
                                               # Arrays in python are called lists.

‘ ‘.join(text.split())                         # This replaces all of the multiple spaces with single spaces.

‘,’.join(list_of_strings)                      # This method takes a list of strings and concatenates them together
                                               # with commas in between the individual items of the list.
```

Introduction

- Outputting data: (examples)
- 1. file output

```
contents = "LARP Cyber"
with open("LatLong.csv", 'w') as _f:
        raw_file_contents = _f.read()

binary_file_contents = b'558BEC…'
with open("shellcode.bin", 'wb') as_f:
        binary_file_cotents = _f.read()
```

```
# The variable is created here because if not when you assign it a value
# within the scope of the subsequent "with open" call the variable

# The value is a binary string. It's displayed as a string, but the value is bytes.
# If you don't open the file with the 'b' it will error on writing a bytes type.
```

- 2. terminal output

```
contents = "LARP Cyber"
print(contents)
```


2. Conditional Statements

Conditional Statements

- Conditional statements are where decisions are made in a program. In python the 'if' statement is 'it'. However, in other languages this isn't the case.
- 'if' statements are constructed as such:

```
if <bool expression>:                # The result from the expression must result in a Boolean value of True or False.
        …                            # Perform action.
elif <bool expression>:
        …
elif <bool expression>:
        …
else:                                # This is the catch-all/fall-through. If no other condition is met it will execute this.
        …
```

- Each 'if' statement requires an associated 'else', but can have any number of 'elif' statements.

Conditional Statements

- ‘if’ statements can have function calls as their expression if it returns a bool value.

```
if function():                  # Here the statement assumes that the function will return True or False.
        …

if function() == ‘derp’:        # If the function returns other values you can assess them to produce a True or False.
        …                       # The ‘==‘ indicates that you are assessing whether both sides of the ‘==‘ are equal.

if dice_roll() >= 3:            # This is a basic math greater than or equal to assessment. It’s either True or False.
        …
```

Conditional Statements

- 'if' statement example:

```
def retFalse():
        return False                    # Always returns False
def retTrue():
        return True:                    # Always returns True

if retFalse():
        print("This will never execute.")    # This never executes because retFalse always returns False.
elif retTrue():
        print("This will always execute.")
else:
        print("This will never execute.")    # This will never execute because the 'elif' expression always returns True.
```

3. Loops

Loops

- The purpose of a loop is to iterate over a data set to perform some form of processing.
  - This allows you to sequentially move through the data to:
    - Manipulate – modifications to the data or formatting to make it more consumable to a person or software
  - This allows you to force an encounter with a desired conditional statement
    - User input / output
    - Extract / Identify occurrences of a subset of data

Loops

- A basic while loop:

```
while <condition>:
        <code>
```

- while loop example:

```
while True:                                     # "True" is a Bool value that is equal to one (1)
        usr_input = input(">>> ")               # This is a built-in that grabs user input from the terminal
        print(usr_input.lower())

people = 5                                      # This defines the value to be iterated over outside of the scope of the loop so the data
while  people > 0:                              # processing persists after the loop exits.
        print("Person {0} likes this training.".format(people))
        people -= 1                             # "-=" is equivalent to "people = people -1"
```

Loops

- for loops:

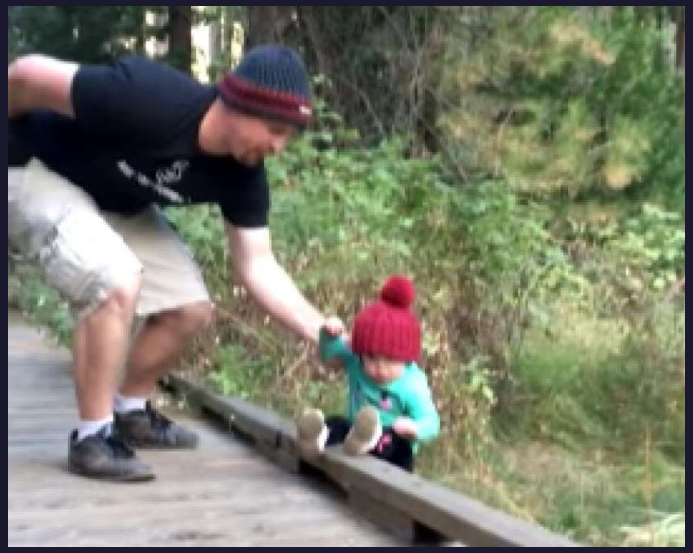
```
for <condition>:
        <code>                  # The code inside this for loop will run every time the loop iterates.
```

- while loop example:

```
people = ["Mike","Tina","Jim","Frank","Barbra"]     # I'm defining the data source outside the scope of the loop.
for person in people:                               # The for loop will iterate through the list 'people'
        print(person)

people = ["Mike","Tina","Jim","Frank","Barbra"]
for person in people:
        if 'i' in person.lower():                   # Because user input can take any form calling the '.lower()' method can/will reduce
                print(person)                       # the number of conditions that need checking for.
```

Exception Handling



4. Scope

Scope

- Scope essentially describes where a variable retains its value.
  - Scope passes down into loops and condition statements but doesn't pass out.
  - What does that mean?
    - A variable that is defined outside of a loop will still have meaning inside the loop.
    - A variable defined inside of a loop / conditional statement does not retain its meaning once the loop iteration or conditional statement ends

Scope

- Scope example:

```
1. from random import randint
2. kids = [["Janet","female",14,0],["Frank","male",12,0],["Nick","male",9,0]]
3. for index, kid in enumerate(kids):
4.     randnum = randint(100000,999999)
5.     kids[index][3] = randnum
6.     print("{0}, age {1}, was assigned customer ID {2}.".format(kid[0],kid[2],kids[index][3]))
```

- The variable 'kids' has value inside and outside of the subsequent for loop. It also retains its value through each iteration of the for loop.
- The variable 'randnum' becomes 'undefined' again once the loop restarts.

Scope

- Understanding scope is critical to understanding both functional and Object Oriented Programming.
- In the same manner that an acronym may have one meaning in one organization but mean something completely different in another. It may have no value in another.
- Just as acronyms such as IRS, DOD, and IHOP are largely recognized across groups so too variables can retain its value across a code base.

Questions?

This training was created to be used in tandem with the follow-on training, "Pillars of Object Oriented Programming (OOP)"

Thank You

TheMalteseSailor

email@email.com

website.internet