Introduction

Concrete Experance

Watch the video below

Objectives

- Understand programming's role as sequential instructions
- Explore types of programming languages
- Compare syntax of English and Python
- Learn the different approaches to following instructions, interactive vs. batch
- Understand the difference between interpreted and compiled programming languages
- Understand the difference between code and data
- Understand the differences between python/python2/python3
- Running Python programs from the command line
- Use the Python REPL: Read-Evaluate-Print-Loop

- Understand literals
- Find and access Python documentation
- Access help()
- Describe Python keywords and objects with dir()
- Exit the Python shell
- Using comments
- Using preferred style for Python programs

TLO Knowledge and Skills

Condition:

- Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of python fundamentals.

Knowledge:

- Python Overview
- Basic Python Syntax
- Common Python Grammar
- Differences between Interactive Read-Eval-Print Loop (REPL) Shell vs. Batch
- Execution of Python code from the Command Line
- Execution of Python code in an Interactive Read-Eval-Print Loop (REPL) Shell

Skills:

- Identify Python Documentation.
- Utilize Python Help functions.
- Understand Basic Python Syntax.
- Utilize Python from Command Line.
- Utilize Python from Interactive Read-Eval-Print Loop (REPL) Shell.

What is Python

- Python is a **high-level**, **interpreted** programming language known for its simplicity and readability. It has gained popularity due to its versatility and wide range of applications. Python is used in various domains such as web development, data analysis, artificial intelligence, cybersecurity, and more.

Benefits To Python

-**Powerful**: Python has a "batteries included" philosophy with thousands of 3rd party libraries as well

-**Fast**: Coding is fast, execution performance is reasonable for many application

-**Plays** well with others: C code can be called from Python and modules for Python can be written in C

-**Runs everywhere**: Windows, Linux/UNIX, Mac OSX, Other

-**Simple and Easy to learn**: Managed, interpreted, simple syntax

-**Open Source**: For personal and commercial use


DISA

Basic Syntax And Semantics

Python's **syntax** refers to the set of rules and structures that dictate how code is written in the language. It includes conventions for defining variables, writing statements, and organizing code. Python uses a combination of English keywords and punctuation symbols to construct valid expressions and statements

Errors:

When writing code, errors will arise quite often. These errors may cause the programs to crash, run forever or provide inaccurate results. These issues are usually caused by runtime, logic or syntax errors.

- **Runtime** - error occurs when a program runs instructions that the computer can't execute.
- **Logic error** - is a type of runtime error in the logic or design of the program.
- **Syntax error** - occurs when an instruction does not follow the syntax rules of the programming language.

Example: **SyntaxError**: EOL while scanning string literal

Semantics:

Python **semantics** defines how the code is executed and the behavior of various constructs. Semantics determines how expressions are evaluated, variables and objects are managed, control flow is handled, and how functions and classes are used. It also includes the rules for variable scope, function calls, object references, and how different data types behave.

Similar to the semantics of a sentence in english python semantics provide the meaning and interpretation of the programming language.

Semantics Errors

Semantic errors are more subtle and occur when the code is grammatically correct but doesn't produce the intended results due to logical or functional mistakes. Semantic errors ca

to detect and may lead to unexpected behavior in the program.

```
#Input:
# Creating 2 variables and storing the input of user
num1 = input('Enter a number: ')
num2 = input('Enter another number: ') user

# “Adding” the two number together, concatenating num1 and num2 together
sum = num1 + num2
# Printing out the sum
print('The sum of', num1, 'and', num2, 'is', sum)

#Output:
#The sum of 3 and 4 is 34
```

Case Sensitivity

Python is case sensitive meaning it considers uppercase and lowercase characters differently. Therefore, the following two variables are different.

```
# The “=” assigns the variable name the_Person to the string “Him”
the_Person = "Him"

# The “=” assigns the variable name the_person to the string “Her”
the_person = "Her"

print("the_Person: ", the_Person)
print("the_person: ", the_person)

#output:
#the_Person:  Him
#the_person:  Her
```

Python Grammar

White Space

In Python, **whitespace** refers to spaces, tabs, and newline characters that are used for formatting and structuring the code. White space assist with determining the structure and readability of Python programs. Unlike some other programming languages, Python uses whitespace for indentation and block delineation, which affects how code is executed.

- **Indentation**: Used to define blocks of code
- **Space in expressions**: Separates elements in a block of and expression
- **Blank lines**: used separate block of code for better readability

INDENTATIONS vs. TABS

Python uses **indentation** to define blocks of code, such as those inside loops, conditionals(if/else), functions, and classes. Python relies heavily on whitespace to determine the control flow structure of an application. As control structures are introduced, we will revisit the indentation rules. Consistent and proper indentation is crucial for the program's functionality. When creating “White Space” using the space bar is preferred over “**Tab**” indentation method. Tabs should be used solely to remain consistent with code that is already indented with tabs. Python disallows mixing tabs and spaces for indentation and it will produce the following error

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

Output:

```
IndentationError: expected an indented block
```

```
for i in range(1, 10):
   print(i, end=" ")
#Output:
1 2 3 4 5 6 7 8 9
```

Comments

- **Comments** are used to explain and label Python code and will be ignored by the Python interpreter. They are use to provide more information about the functionally of the code and ultimately to make Python code more human-readable.
- Comments assist with **collaboration and ease of use** allowing other users or developer edit your code.
- Comments can also be used to prevent execution when testing code. If you are troubleshooting your code, you can use comment blocks to organize the flow of execution so that you can identify run time errors or incorrect logic.

Example 1

```
1  # This is a comment!!
2  print("Hello, this will print!")
```

Output: Hello, this will print!

Example 2:

```
1  # This is a comment!!
2  # It is multilined
```

Statements

- In Python, **statements** are sequence of instructions or commands that perform specific actions.
  - Examples of statements include variable assignments, conditional statements (if-else), loops (for, while), and function definitions.
- All lines in a Python application must begin in the first column, unless the statement is in the body of a loop, conditional, function, or class definition.

Line / Statement Terminator

- In python, the **statement terminator** is marked by the end of the line. In the example below, we can see that two separate statements are split by a newline.

```
#Example 1
x = 5 + 5 # statement 1
print(x)  # statement 2
```

- This is not the case in other languages like C and C++, where every statement must end with a semicolon ; .

- Python 3 supports the use of a semicolon like its C based predecessors but it is not required. Some coders will use it for readability.
- In the example below, we can see that multiple statements are split by a ; [Semicolon] instead of a newline.

Example 1:

```
x = 5 + 5
print(x)
```

Example 2:

```
length = 10; width = 5
area = length * width
print(area)
```

- We can extend the statement beyond the end of line by using the backslash character [Backslash]
- When used in this fashion it is often referred to as the line continuation character.
- In the example below, we can continue with our statement on newlines without ending the statement.

Example 3:

```
result = 500 * 2 + \
400 * 3
```

```
3
4  print(result)
5
6  #Output:
7  #2200
```

Python Execution

Read-Evaluate-Print-Loop(REPL)

Python Interactive Mode

Python can work in an interactive mode, which uses a special program called a REPL (Read-Evaluate-Print-Loop). The Python REPL:

- **Reads** what the user types
- **Evaluates** the typed input as a Python expression
- **Prints** out the results of that evaluation if any
- **Loops** back to waiting for more input from the user

Batch

In Python, instead of typing out individual statements in a REPL, a programmer could use any text editor to create a file, usually with the extension .py, that lists all of the instructions in the proper order. The programmer could then provide the file to a special program to be translated to machine instructions as a unit.

Execution of code

Executing Python code can be done from the command line, Python interactive interpreter, or an Integrated Development Environment (IDE). To execute a Python program from the command line:

1. Create a Python script with a text editor and name it with the standard .py file extension.
2. Execute the file using one of the various python commands.
3. Python recommends using `python2` for Python 2, `python3` for Python 3, and `python` to refer to either Python2 or Python 3, depending on the platform.

Shebang

Many scripts include the "shebang" (#!) character sequence on the first line of each source file, such as:

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

On a Linux system, the shebang provides the pathway to the Python interpreter. The chmod u+x hello.py command is used to grant execution permissions to the user, and ./hello.py is used to execute the file. In Python 3, parentheses () are required for function arguments.

Execution

The Python interpreter, sometimes called an interactive Python shell, allows Python commands to be executed interactively. Interactively entering an expression will output the result of executing the expression without the need to call the print() function. This behavior is only available in the interactive shell and would not produce output if the same statement was run within a script.

Here is a small example session executing the Python interpreter in interactive mode:

```
developer:~/testing$ python3
Python 3.8.10 (default, May 26 2023, 14:05:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("This is a string")
This is a string
>>> print(50 / 3, 50 // 3, 50 % 3)
16.666666666666668 16 2
>>> exit()
```

Integrated Development Environment (IDE)

An IDE (Integrated Development Environment) is software that can edit, organize, and debug code. IDEs support many different programming languages and can contain more features than just editing, organizing, and debugging. They integrate several tools specifically designed for software development, including:

- Code Editing: A text editor that assists in writing software code with features such as syntax highlighting, auto-completion, and bug checking.
- Building / Debugging: A program for testing other programs, highlighting and identifying bugs, and building code.
- Source Control: Managing changes to code.


DISA

Help in Python

Python Documentation

There are various ways in which a Python programmer can get help from the Python documentation. A good starting point is one of the

following URLs:
Python Docs - Latest Version
Python Docs - By Version

Note

Built-in Help

In addition to the online Python documentation, the interactive Python shell can provide additional help. Typing `help()` at the Python shell prompt will start the Python help utility. For example:

```
developer:~/test$ python3
>>> help()
help> str
help> str.isalpha
help> int
help> math
help> math.degrees
```

Usefull Commands

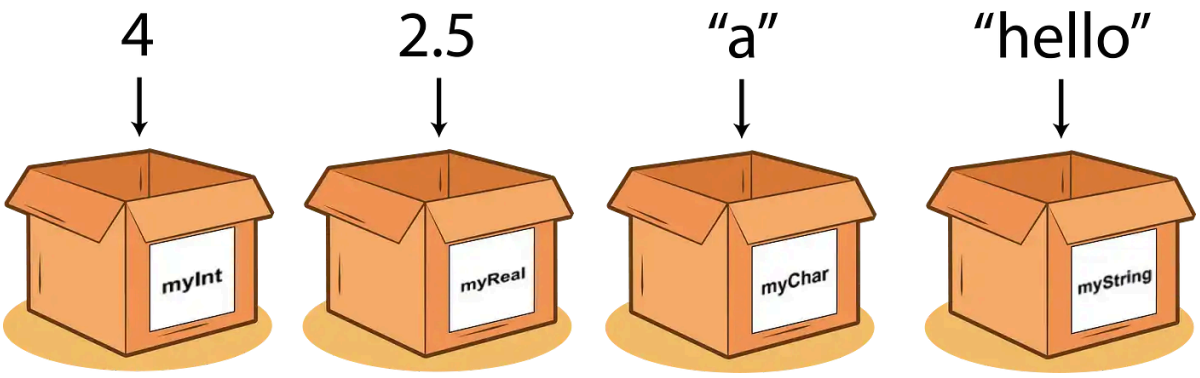
When working with the Python REPL, there may be times when a programmer needs more information about an expression, value, or object. The `dir()` function can be helpful in describing what properties a Python object may have.

```
dir()
dir(4)
exit()
```


DISA

Introduction

Concrete Experance



OBJECTIVES

- Explore basic data types of the Python language
- Use type() to find the data type a variable refers to
- Create sequences of characters as Python strings
- Get the length of a string with len()
- Use Boolean data types to represent True or False
- Understand the None type
- Cast data as one type or another
- Convert characters to their underlying values

TLO Knowledge and Skills

Conditions:

- Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force, students will demonstrate an understanding of data types, variables, and operators.

Knowledge:

- Identify the components that build program structure in Python
- Understand Python structure and the components used in scripting

Skills:

- Working knowledge of how to write and modify scripts in order to perform specific functions
- Knowledge of parsing through scripts and files

Primitive Data Types

- In programming, data type is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data.
- Python has many data types. Each of these types behave and do different things.
- There are builtin functions to convert from one type to another. (We will cover this later)
- If the source type cannot be converted to the target type, Python will show a TypeError.
- Broadly, these data types can be classified as Numbers, Strings, and Objects (more specifically, Sequence and Mapping types).

Integers

- Numbers can be an Integer called an int.
- In Python, integers are defined as type int.
- Numbers 1,2,3 and 4… etc are integers. An integer is any whole number with no decimal or fractional value.
- Python3 does not limit the maximum value of an integer, it is unbounded. Unbounded is a mathematical term for functions and values that have no maximum value.
- The size of the integer is limited to the size of the registers/memory allocated to the integer within the operating system.

```
1  x = 5
2  y = 4
3  z = -1
```

Floating Point

- Floating point (decimal) called a float.
- Floating point numbers, that is those numbers with decimal point places, are defined as float.
- A floating point number precision is dependent on the system architecture.

- Precision refers to the accuracy at which small numbers can be measured. The greater the precision the more decimal places can be represented by a value.

```
a = 2.0
b = 3.5
a = 1.1567
```

What can we do with numbers

Other Base Types

- When working with integers, it may be convenient to encode literal values in bases other than decimal.

```
x = 10

bin(10)
#'0b1010'
oct(10)
#'0o12'
hex(10)
#'0xa'
print(type(x))
#<class 'int'>
```

Base	Value	Decimal Value
Hexadecimal	`x = 0xff`	255
Octal	`y = 0o77`	63
Binary	`z = 0b111`	7

Numerical Operations

Operation	Result
x + y	Sum of x and y
x - y	difference of X and Y

Operation	Result
x * y	Product of X and Y
x / y	Quotient of x and Y
x // y	Floored Quotient of X and Y
x % y	Remainder of x / y
-x	x negated
+x	x unchanged
abs(x)	Absolute Value(or magnitude of x)
int(x)	X Converted to integer
float(x)	x Converted to floating point

- The = (equal sign) does not test if the two sides are equal, but sets the left hand side to refer to the right hand side.
- Python supports three distinct numeric data types.
- Integers, Floating Point Numbers, and Complex Numbers.
- Python's numerical operations are shown below, sorted by ascending priority.

Boolean

- The bool data type is named after Boolean values. In short, a bool data type can hold only the values 0 or 1.
- This can be thought of as a way to express True [1] or False [0]. For example, the bool value of 0 represents False.
- Any value of a variable other than 0 will result to True in Bool
  - example: x = 5; bool(x) >>> True

- example: x = 0; bool(x) >>> False

```
#And Truth Table:
#Output:
print(True and True)            True
print(True and False)           False
print(False and True)           False
print(False and False)          False
```

- Python supplies keywords to express True and False, these are unsurprisingly, True and False.
- The case is important.
- Both must start with a capital letter.
- Also make note that there are no quotation marks around True or False.

String

- A string is a sequence of characters (upper and lowercase alphabetical characters, numbers, or special characters) surrounded by quotes.
- A string may also be referred to as a string literal.
- Recall that a literal is a basic value.
- This is raw data that cannot be changed once initialized.
- Python will recognize this data type as str for string.
- A Python expert would describe strings as an “immutable sequence of zero or more characters.”
- Immutable means that strings cannot be changed once created.
- However, a variable referring to a string can be made to refer to a different string at another time in code

```
print("this is a string")
```

Literal Strings

- Literal string values may be enclosed in single ‘ ‘ or double quotes “ “.
- Since strings are a sequence of characters, the built-in len function can be used to determine the number of characters the string contains.

- Literal strings can span multiple lines in several ways.
- Using the line continuation character as the last character.
- Literal strings may also be prefixed with a letter r or R.
- These are referred to as raw strings and use different rules for backslash escape sequences.
- If a programmer needs a special character in a string, the programmer may need to use an escape sequence.
- The following is a list of the escape sequences that can be used in literal strings to represent special characters.

Character Meanings

Sequence	Character/Meaning
\newline	Line Continuation
\	Backslash
\'	single Quote
\"	Double Quote
\a	ASCII Bell(BEL)
\b	Backspace
\f	Form Feed
\n	Line feed
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

Sequence	Character/Meaning
\ooo	ASCCI Character (octal value ooo)
\xhhh	ASCCI Character (hex value hhh)
\uxxxx	Unicode Character with 16-bit hex value xxxx
\Uxxxxxxxx	Unicode Character with 32-bit hex value xxxxxxxx

```
#Input:                                  output:
print("this is a 'single quote' inside a double quote") #this is a 'single
quote' inside a double quote

print('this is a \'single quote\' inside a single quote')   #this is a
'single quote' inside a single quote

print('this is a \"double quote\" inside a single quote')   #this is a
"double quote" inside a single quote

print('this is a \t tab')                       #this is a      tab

print('this is a \n newline')                   #this is a
#newline
```

List

- Mutable sequence of items (not necessarily of hte same type)
- Creating a list:
- list()
- []

```
li = []
li2 = list()
li3 = ["string", 2, 5.0, True]
```

Tuple

- immutable sequence of items

- Creating a tuple:
- tuple()
- ()

```
    tup = tuple()
    tup2 = ()
    tup3 = (1, "string", 3.0)
```

None Type

- Another data type in Python is the NoneType which can be represented by using the None keyword.
- None is similar to null in other languages. In Python, None means nothing and will evaluate to False in a conditional statement.
- None is often used to check the result of some action or to ensure a variable has a value and is not nothing

Character Encoding

Coding Scheme

- In Python, characters are encoded in Unicode by default.
- To a computer, every character is a number, this includes the letters and other characters typed on the screen.
- These numbering schemes indicate which number is the "code" behind a character.
- There is a code for each case of a character as well.
- There is even a code for each emoji you see in a program.
- Other coding schemes exist besides Unicode. Some older encoding standards include ASCII and EBCDIC.

Ord() Function

- A programmer can find the code behind a character by using the ord() function. The ord() function provides the ordinal (code) behind the character provided as input.
- For example, the Unicode code point for “A” is 65. For “a”, the number is 97. The code for the exclamation point is 33.
- The quotes around the letters and the exclamation point are important.

```
>>> ord('A')
65
>>> ord('a')
97
>>> ord('!')
33
```

Chr() Function

- If a programmer initializes a string with a variable name, the variable name refers to a specific area in the computer’s memory.

- Python keeps track of the areas that have numbers (like integers and floats) and strings (which are really numbers too).
- In that way, Python keeps track of the data type of a value referred to by a variable and knows whether to use it as a number in a calculation or a number to represent a character.
- Python provides the chr() function to see which character is associated with a particular code.

```
>>> chr(97)
'a'
>>> chr(34)
'"'
>>> chr(33)
'!'
>>> chr(65)
'A'
```

Dealing With Variables

Naming Conventions

- A Python identifier is a name that is used to identify any of the following:
  - Variables, Functions/Methods, Classes, or Modules
- Often Python programming documentation will use the term name to describe an identifier. An identifier must start with:
  - A letter of the alphabet or the underline _ character.
- This can be followed by any number of letters, digits and/or the _ character.
- Identifier names cannot consist of any other characters:
  - Variable names and function names typically begin with a lowercase letter, while class names typically start with an uppercase letter.
- Remember that Python is a case-sensitive language.
  - This means mynumber, MyNumber, and Mynumber are three different variables.
- Most Python programmers would prefer another option, my_number.
- The use of underscores between multi-word names is called snake_case.
- Capitalizing the first letter of each word with no spaces between is called CamelCase.
- There are many different naming styles within the language.

Dyanmic Variable Declaration

What if We wanted to declare more then one variable at a time?

- The following program emphasizes the dynamic type system of Python.
- In a dynamically typed language, a variable is simply a value bound to a name.
- The value of what is being referenced by the variable has a type like int or float or str, but the variable itself has no type.
- Python uses pass by object reference. Other languages also have pass by value and pass by reference

```
s,x,y = 'hello', 10, 5.0
tab = " \t"
                                          #output:
print(s, tab, type(s), tab, id(s))        #hello  <class 'str'>
139954674414256
print(x, tab, type(x), tab, id(x))        #10     <class 'int'>
9764672
print(y, tab, type(y), tab, id(y))        #5.0    <class 'float'>
139954674401328
```

- The built-in type() function returns the data type of the object that is referenced by a particular variable.
- Notice that the type of the reference is bound dynamically as the program is executed.
- The built-in id() function returns a unique identifier of an object that is referenced by a particular variable.
- This is the address of a particular place in the computer's memory.
- It's not vital to know this address in normal Python programming but it can illustrate how Python stores and accesses data in memory. There is only a single int object 10 being created.

```
s,x,y = 'hello', 10, 5.0
tab = " \t"
                                          #output:
print(s, tab, type(s), tab, id(s))        #hello  <class 'str'>
139954674414256
print(x, tab, type(x), tab, id(x))        #10     <class 'int'>
9764672
print(y, tab, type(y), tab, id(y))        #5.0    <class 'float'>
139954674401328
y = 10
print(y, tab, type(y), tab, id(y))        #10    <class 'int'>
9764672
```

Data Conversion Function

What if we wanted to convert types of Varables such as a Int to a string?

- Often times, a programmer may need to treat a string as a numeric type or a number as a string within your code.
- Python provides several functions that allow these types of conversions to be performed.
- The `int()`, `float()`, and `str()` are methods called constructors that initialize an object as opposed to a function call.

- Some of the more common built-in conversion functions are:

`int()`	`str()`	`chr()`	`oct()`
`float()`	`ord()`	`hex()`	`bin()`

```
x = '101'
print(f"base 10: {int(x)} \tBase 2: {int(x, 2)} \tBase 8: {int(x, 8)} \tBase 16: {int(x, 16)} ")
#Output:
#base 10: 101   Base 2: 5       Base 8: 65      Base 16: 257

num = int(x)
print(f"Binary: {bin(num)} \tOctal: {oct(num)} \tHex: {hex(num)}")
#Output:
#Binary: 0b1100101      Octal: 0o145    Hex: 0x65
```

Augmented assignment Operators

Augmented assingment is when we want to perform a operation and capture the result of that operation inside our variable.

we can do so using the below operations.

Operation	Result
x += y	Sum of x and y assigned to x
x -= y	difference of X and Y assigned to x
x *= y	Product of X and Y assigned to x
x /= y	Quotient of x and Y assigned to x
x //= y	Floored Quotient of X and Y assigned to x
x %= y	Remainder of x / y assigned to x

Operation	Result
x **= y	x to the power of y assigned to x

Python Keywords

- Python keywords are reserved words within the language.
- In algebra class students learn to name values with names like x, y, and π (pi).
- Likewise, a programmer can assign names to values in Python as well, however it would be wise to avoid any Python keyword.
- These words cannot be used as the names of variables or functions.
- They are listed here for reference. This list can be generated by executing:
- `python help('keywords')` at the >>> prompt or
- keywords at the help> prompt.

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	`break`
`except`	`in`	`raise`		

Useful Commands

Python Input

- Use the input() method to allow a user to input data
- when one enters data into the terminal, it is of the string data type
- Can change data type of input by doing a type conversion

```
    name = input("Please enter a name: ")
    age = int(input("Please enter your age: "))
    print("Hello", name, "your age is", age)
```

Python Math

- A Python programmer can replace a value with a variable instead as a placeholder of sorts.
- For example, the following code would add two numbers together and display the results in the Python REPL.
- Recall that the Python REPL is a read-evaluate-print-loop.
- In the first statement, x = 3, the programmer assigned the value 3 (an integer) to the variable name x.
- In the second statement, y = 2, the programmer performed a similar task.
- It would better however to say that 2 has been assigned to y.
- In the third statement, x + y, the programmer is not asking to see an output like xy, but instead, like in algebra class, to take the value assigned to x and add it to the value assigned to y.
- A later statement, z = x + y, saves the value of the calculation for later use with a variable named z.

```
x = 3
y = 2

z = x + y

print(f'{x} + {y} = {z}')
```

```
#output
# 3 + 2 = 5
```

- At this point, there is no way to retrieve the result of 5 again.
- Now the programmer can use z to recall the results of the calculation later in code.
- Because the programmer is using the Python REPL, there is no need to use a Python keyword or function to output the results of a statement.
- In programs not designed to run in the Python REPL, a special Python function called print() will be necessary to tell Python to output the results.
- In the code, the programmer reassigned the value 3.3 to the variable f.
- This section of the code is repeated below for reference

```
x = 3
y = 2
z = x + y

print(f'{x} + {y} = {z}')
#output
#3  +  2  =  5

f = 3.3

n = f + z

print(f'{f} + {z} = {n}')
#output
# 3.3 + 5 = 8.3
```

- Any variable can be assigned to any value at any time in a program. It is the programmer's responsibility to ensure that a variable contains an appropriate value and data type to avoid unexpected results or errors.
- Repeatedly re-using the same variable for different values and data types is not a good development practice.
- Evaluation happens at the time of execution. Each line is executed as it is entered in the REPL.
- It is important to understand that Python is "right-handed" in that Python evaluates the right hand side of an assignment statement first and then assigns that result to the variable on the left hand side.

```
x = 5 * (2+2) -10 # correct

5 * (2+2) -10 = x # Incorrect
```

Variable Assignments

- Chained Assignment: A quirk of this right-handedness in Python is that a programmer can use this to assign multiple variables to the same value at one time. For example:

```
x = y = z = 2                           #output:
print(f'x: {x} y: {y} z: {z}')          #x: 2 y: 2 z: 2
```

- This results in three separate variables x, y, and z referring to the value 2.
- Multiple Assignment: Another quirk of Python syntax is that multiple variables can be assigned to multiple values in the same statement. For example:

```
x, y, z = 1,2,3                         #output:
print(f'x: {x} y: {y} z: {z}')          #x: 1 y: 2 z: 3
```

- This results in three variables like the previous example, however x refers to 1, y refers to 2, and z refers to 3.
- Augmented Assignment: The right-handedness of a Python assignment statement also gives a programmer another shortcut. Often programmers need to perform a calculation on a value and refer to the new value with the same variable name. A counter in a game is a typical example.

Counter

- Let's say a programmer wanted to count the number of times that an action had been performed, the programmer can make a variable called counter and assign it in this way:

```
counter = 0
```

- This is called initialization. A variable in Python must be both declared (that is, have a name assigned) and initialized (that is, have a value assigned) at the time of creation.

- A value is being assigned, but to what? Both steps happen when a variable is first created.
- Now, suppose a programmer wants to increase the value of counter.
- The counter variable has already been declared and initialized with counter = 0.
- Now a programmer could just reassign the counter variable to 1, such as:

```
counter = 1
```

- While that works, a programmer might not know the exact value to assign to the counter, but might just want to increase it by one, or increment it.
- The programmer could write:

```
counter = counter + 1
counter += 1
```

Introduction

TODO: introduction goes here

Objectives

- Utilize strings methods to modify the case of strings.
- Identify the number of characters within a string.
- Find specific characters within a string.
- Modify the contents of a string.
- Use escape sequences.
- Create triple quoted strings.
- Work with string slices.

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of intermediate strings.

Knowledge:

Identify the components that build program structure in Python Understand Python structure and the components used in scripting

Skills:

Working knowledge of to write and modify scripts in order to perform specific functions Knowledge of parsing through scripts and files


DISA

What are Intermediate Strings

- Strings in Python are an immutable sequence of zero or more characters.
- Literal string values may be enclosed in single or double quotes and can span multiple lines in several ways.
- This is achieved by using the line continuation character as the last character.
- Literal strings may also be prefixed with a letter r or R.
- These are referred to as raw strings and use different rules for backslash escape sequences.
- Strings in Python are represented using a class named str.
- A class can be thought of as a new data type and an object as an instance of that data type.
- The str class has an abundance of methods defined within it.
- Some of the methods return Boolean values such as True or False.
- Some methods return a modified copy of the string.
- A full listing of the string methods can be found by using help(str) from the Python interactive shell.

String Casing

- Python provides several methods that can be used to modify the casing of characters in a string.
- `upper()` returns a copy of the string with all cased characters converted to uppercase.
- `lower()` returns a copy of the string with all cased characters converted to lowercase letters.
- `casefold()` similar to lower(), is stronger, more aggressive, meaning that it will convert more characters into lower case. will find more matches when comparing two strings and both are converted using the casefold() method
- `title()` returns a copy of the string with the first letter of each word in the string in upper case and the rest of the characters in lower case.
- `capitalize()` returns a copy of the string with the first letter in upper case and the rest of the characters in lower case.

```
string = "python is fun"
#Output:
print("String is now upper:", string.upper())           #String is now upper:
PYTHON IS FUN
print("String is now lower", string.lower())            #String is now lower
python is fun
print("String is now casefold:", string.casefold())     #String is now
casefold: python is fun
print("String is now title:", string.title())           #String is now title:
Python Is Fun
print("String is now capitalize:", string.capitalize()) #String is now
capitalize: Python is fun
```


DISA

Finding Substrings

- A variety of methods can be used to find instances of characters or strings within another string.
- count(s) will return the number of times the value passed to the method is found within the string.
- index(s) will return the index position of the first instance of the value passed to the method.
- A ValueError exception is raised if the sub-string to find is not found within the string
- find(s) is similar to index but find will return a -1 when the sub-string is not found rather than raising an exception.

Note

that all these methods perform case sensitive searches.

```
string = "python is fun"

                                #Output:
print(string.count("n"))        #2
print(string.index("n"))        #5
print(string.find("n"))         #5
```

METHODS TO MODIFY STRINGS

- Methods can be used to clean up data or to replace or remove unwanted characters.
- strip() will remove any leading and trailing whitespace from the string.
- lstrip() will remove any leading whitespace (whitespace on the left)
- rstrip() will remove any trailing whitespace (whitespace on the right)
- replace(old, new) replaces all instances of the first parameter with all instances of the second parameter.
- An optional third parameter can be used to specify the number of times the replacement is to be made.

```
s = "\t 123-456-7890 \t"
                                          #Output:
format = s.strip()
print(s)#-------------------------->    123-456-7890
print(format)#-------------------------->123-456-7890
print(format.replace("-", " "))#-------->123 456 7890
```

is METHODS for Strings

- Methods can be used to to validate if a string contains a certain set of characters
- When the method validates, it will return either True or False

The methods are:

- isdigit(): will return True if all characters are digits
- islower(): will return True if all characters are lowercase
- isupper(): will return True if all characters are uppercase
- isalnum(): will return True if all characters are alphanumeric

```
s = "will this return true"
print(s.islower())
print(s.isdigit())
print(s.isupper())
print(s.isalnum())

# Ouput:
True
False
False
False
```

Escape Characters

sequence	Character/Meaning
\newline	Ignored
\	Bckslash
\'	Single Quote
\"	Double Quote
\a	ASCII Bell(Bel)
\b	Backspace
\f	form feed
\n	Line feed
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab
\ooo	ASCII Character(octal value ooo)
\xhhh	ASCII Character (Hex Value hhh)
\uxxxx	Unicode Character with 16-bit hex alue xxxx
\Uxxxxxxx	Unicode Character with 32-bit hex value xxxxxxx

```
single_quote = "This is a \'single quote\'" # The string will escape char for a single quote
new_line = "This will produce a newline: \n" # The string will produce a newline at the end of it
carriage_return = "This will produce a carriage return: \r" # The string will produce a newline at the end of it
tab = "This is a \t tab" # The string will add a tab
chess_piece = "\U0000265F" # Use unicode to create a chess piece


print(single_quote)             #This is a 'single quote'
print(new_line)                 #This will produce a newline:
print(carriage_return)          #
print(tab)                      # This is a     tab
print(chess_piece)              #♟
```

```
                                                        #Output:
print("Strings are \'easy\' in Python")                 #Strings are 'easy' in Python
print(r"Strings are \'easy\' in Python")                #Strings are \'easy\' in Python
print("Strings are \x27easy\x27 in Python")             #Strings are 'easy' in Python
print('\U0000265F','\U0000265F','\U0000265F','\U0000265F','\U0000265F')#--> ♟ ♟ ♟ ♟ ♟
```

Triple Quoted Strings

- Python supports the use of triple quotes (single or double) for defining a long (or short) string.

```
s = '''this is a string'''
s = """this is also a string"""
```

- Triple quoted strings can also be used as f-strings.
- Any white space in the triple quoted string is included in the stored data and will be part of the printed output.
- Triple quoted strings are often used as docstrings, which are usually included as descriptions of functions and classes.
- Triple quoted strings are often used as a workaround for multi-line comments.

Note

triple quoted strings and comments are two different things.

```
side_one = '''\U0000265F \U0000265F \U0000265F \U0000265F \U0000265F'''

side_two = '''\U0000265F \U0000265F \U0000265F \U0000265F \U0000265F'''

                        #Output:
print(side_one)         #♟ ♟ ♟ ♟ ♟
print(side_two)         #♟ ♟ ♟ ♟ ♟
```

String Indexing

- Strings, and other sequence types, have a set of operations that utilize a pair of square brackets [ ] in the syntax.
- All three of the following types of operations return a sub-string of the sequence the operation is performed on.
- Indexing: Involves a single integer placed inside of the brackets.

```
the_string[index]
```

Note

Variables containing an integer can be used.

- Slicing: Involves two integers separated by a colon in the brackets.
- Returns a sub-string from the start index to the end index.
- The end index is exclusive.

```
the_string[start:stop]
```

- Extended Slicing: Involves three integers separated by colons.
- Returns a sub-string from the start index to the end index using the step.
- The end index is exclusive.

```
the_string[start:stop:step]
```

```
string = "Python is fun"
                        #Output:
print(string[-3:])         #fun
print(string[7:9])         #is
print(string[::-1])         #nuf si nohtyP
```

String Formating

- Python has several ways to format the output rather than just printing space separated values.
- The newest and easiest way to format a string for output in Python uses formatted string literals or f-strings.
- Beginning with Python 3.6, f-strings are available for output formatting.
- To use a formatted string literal, begin the string with f or F before the opening quotation mark.
- This can be used in front of single, double, or triple quotation marks.
- Inside the string, the programmer adds braces such as { }

```
python = "Python"           #output
print(f'{python} is fun')   #Python is fun
```

- A programmer can define and merge f-strings to create a multi-line f-string.
- Notice that the {name} placeholder, or field can be repeated. Python evaluates the fields in the braces as Python expressions and looks for variables to match.
- The order of the fields does not matter and as stated earlier, a field can contain any valid Python expression. It is important to include the f or F in front of each line.

```
python = "Python"
students = "Students"

long_string = (
   f"{python} is fun. "
   f"{students} seem to enjoy learning Python"
)
print(long_string)
```

Output: Python is fun. Students seem to enjoy learning Python.

- When using f-strings the programmer can place curly braces in a literal string.
- The curly braces can contain a Python expression.
- Additionally, the programmer can specify additional formatting details for the result of the Python expression.

- A format specifier can be appended to the field after a colon : and inside the curly braces.
- Backslashes are not allowed within the formatting field and will cause a syntax error!

Type	Meaning
's'	String format. This is the default type for strings and may be omitted.
none	The same as 's'.
'b'	Binary Format. The Number is display as base 2.
'c'	Character. Converts the integer to a Unicode Character.
'd'	Decimal Integer. This converts the number to base 10.
'x' or 'X'	Hex format. Display the number in base 16. The case of the hex number will match the case of the specifier.
'f'	Fixed point number. Displays the number with a decimal point. Default precision is 6.
'%'	percentage. Multiplies the number by 100 and displays in fixed ('f') fromat with a percent sign.

```
a = 1000
b = "Hello"
c = 0b110010011
d = 90
e = 3.14159
f = 0x3f
g = .5

print(f"A is now binary: {a:b}") #  Converts to binary value
print(f"D is now unicode char: {d:c}") #  Converts to unicode character
print(f"C is now base 10: {c:d}") #  Converts number to base 10 decimal
print(f"F is now base 10: {f:d}") #  converts to base 10 decimal. In this
case it was a hex value
print(f"A is now hex: {a:x}") #  Converts to Hex
print(f"E is now 2 decimal places: {e:.2f}") #  this will show 2 digits aft
the decimal
print(f"G is now a percentage: {g:%}") #  This will display a value
percentage sign.
```

```
print(f"G is now 2 decimal places: {g:.2%}") #  This will display a value with only 2 digits after the decimal and the percentage sign.
```

OUPUT A is now binary: 1111101000 D is now unicode char: Z C is now base 10: 403 F is now base 10 hex: 63 A is now hex: 3e8 E is now 2 decimal places: 3.14 G is now a percentage: 50.000000% G is now 2 decimal places: 50.00%

- When the code is run, Python will output 7.29. The field for interest_rate in the print statement has a format specifier after the variable name. The .2f may be more easily understood from the right end.
- The 'f' indicates the programmer would like the output to be fixed-point. The '.2' indicates that the programmer would like two places after the decimal point, in other words, two places of precision.
- In the output, Python has rounded 7.288 to 7.29
- The rounding is for output only and does not change the underlying value of the interest_rate variable.

```
                                        #Output:
interest_rate = 7.288
print(f'{interest_rate:.2f}')           #7.29
number = 123456.78
print(f'value: {number:7,.3f}')         #123,456.780
```

- The following code example has four variables, each referring to an integer. The variables are labeled ip_add_octet1, ip_add_octet2, ip_add_octet3, and ip_add_octet4. Each one represents on octet of an IP address.

```
ip_1 = 192
ip_2 = 168
ip_3 = 0
ip_4 = 1

                                                #Output:
print(f'{ip_1}.{ip_2}.{ip_3}.{ip_4}')           #192.168.0.1
print(f'{ip_1:b}.{ip_2:b}.{ip_3:b}.{ip_4:b}')   #11000000.10101000.0.1
print(f'{ip_1:x}.{ip_2:x}.{ip_3:x}.{ip_4:x}')   #c0.a8.0.1
```

- Like f-strings, replacement fields are indicated with curly braces {}.
- However, the syntax of referencing the variables to be printed was different.

```
input = "easy"
output = "Formatting in python is {}".format(input)
print(output)
```

Output: Formatting in python is easy

- Much like f-strings, the str.format() method uses curly braces in a template string to mark where the output should be replaced.
- format() is a method to be used with strings.
- The syntax is to create a string literal with fields marked with curly braces {} where the programmer wants the output to appear.

```
format = "Formatting"
input = "easy"
output = "{} in python is {}".format(format, input)
print(output)
```

Output: Formatting in python is easy

- Each of the variables has an index within the method.

```
str.format(0, 1, 2)
```

```
format = "Formatting"
input = "easy"
output = "{0} in python is {1}".format(format, input)
print(output)
```

Output: Formatting in python is easy

- The programmer could use names instead of indices much like what has been shown with f-strings. This produces the same output but now the values for each field have been supplied as named arguments in the format() method.
- Used in this way, the order of the fields no longer matters.
- The same format specifiers used in f-strings will work with the format() method.

```
output = "{format} in python is {input}".format(format = "Formatting", input =
"easy")
print(output)
```

Output: Formatting in python is easy

- The colon : separates the index or name of the field from the format specifier.

```
                                                #Output:
number = 150
print(f"The number is: {number}")               #The number is: 150
print("The number is: {:X}".format(number))     #The number is: 96
print("The number is: {:b}".format(number))     #The number is: 10010110
```

Introduction

Objectives

- Use the indenting requirements of Python properly.
- Use the Python control flow constructs correctly.
- Understand and use Python's various relational and logical operators.
- Use if statements to make decisions within the Python code.
- Use for loops to iterate through iterables.
- Use while loops to perform repetitive operations.
- Utilize the range and enumerate functions.
- Control loop termination.

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of conditional expressions.

Knowledge:

Identify the components that build program structure in Python Understand Python structure and the components used in scripting

Skills:

Working knowledge of to write and modify scripts in order to perform specific functions Knowledge of parsing through scripts and files


DISA

Identify Pythonic Code Blocks

- Python provides a robust set of keywords and related items that control the flow of execution within an application.
- In this section, we will explore the various conditional execution options that Python provides.
- In addition, we will look at the various operators used by Python in control flow constructs.
- Python mandates the use of indenting within a compound statement. The first line of the compound statement is referred to as the header.
- All other statements within the compound statement are referred to as the suite or body and must be indented the same number of columns to be part of the same suite.
- The suite ends with the first statement that is “indedented” to the column of the header.
- One such type of compound statement is a control structure.
- Suites must be indented the same amount of white space from the starting column of the header.
- If tabs are used in the source code, a single tab is not equal to the number of spaces used in a tab.
- It is recommended that spaces be used over tabs.

```
1  if some_condition:
2     suite_statement_1
3     suite_statement_2
4
5  print("output")
```

Conditionals

IF and ELSE

- The fundamental decision making control structure in many programming languages is the if statement.
- Code inside an if suite is only executed if a given condition is true.
- The following examples demonstrate proper indenting when using the if statement and its variants.
- Also, notice the required use of the colon : to end the header portion of the if and the else.

ELIF

- When writing an if statement, there can be zero or more elif blocks, and the else block is optional.
- The keyword elif can be used to prevent the testing of multiple if statements when only one of the if statements can ever be True at a time.
- It can also sometimes prevent the need for nesting if statements inside of an else.
- Since Python does not have a “switch” statement found in other languages, elif is often a suitable substitute.
- elif gives the opportunity to have multiple conditional checks.
- When one conditional check results in True, the True part of the suite is executed and the rest of the elif statements are skipped.

```
                                                    #Output:
number = 25

if number <= 25:
   print("number is smaller or equal to 25")        #number is smaller or
equal to 25
elif number == 50:
   print("number is 50")
elif number <= 75:
   print("number is less than or equal to 75")
```

```
10  else:
11     print("number is greater than 75")
```

RELATIONAL VS. LOGICAL OPERATORS

RELATIONAL

Operator	Meaning
<	Strictly Less than
<=	Less than or Equal
>	Strictly Greater Than
>=	Greater than or Equal
==	Equal
!=	Not Equal
is	Object identity
is not	Negated Object identity

LOGICAL

Operator	Unary or Binary	The Result is True When...
not	unary	Operand is False
and	binary	Both Operands Must be True
or	binary	Only one Operand needs to be True

```
# Declaring two variables

num_1 = 10
num_2 = 0
                                          #Output:
if num_1 == 10 and num_2 == 0:
   print("num_1 is 10 and num_2 is 0")      #num_1 is 10 and num_2 is 0

if num_1 == 10 or num_2 == 10:
   print("num_1 is 10 or num_2 is 10")      #num_1 is 10 or num_2 is 10
```

- Both the and and the or are short-circuited operators.
- or: This means as soon as a condition in an or series of conditional tests evaluates to True, any remaining test conditions in the or series are not evaluated.
- and: Likewise, as soon as a condition in an and series of conditional tests evaluates to False, any remaining test conditions in the and series are not evaluated.
- The relational and logical operators yield results of either True or False.
- True: Any non-zero value.
- False: 0, Empty sets, dictionaries, tuples, or lists, empty strings, none.

X OR	**Y**	**Evaluates To:**
True	True	True
True	False	True
False	True	True
False	False	False

X AND	**Y**	**Evaluates To:**
True	True	True

X AND	Y	Evaluates To:
True	False	False
False	True	False
False	False	False

Loops

For Loops

- The for loop in Python is used to iterate over the items of any sequence, such as a string, list, dictionary, or any iterable object.
- The generic syntax of a for loop is: for target in sequence: suite
- Each item in the sequence is evaluated once and assigns the item to the target and then the suite is executed.
- The loop terminates after processing the last item in the sequence.

```
# Storing a string in a variable
string = 'Python is fun!'

# Using a for loop to print the string
for each_character in string:
   print(each_character, end="")
```

output: Python is fun!

While Loops

- The while statement causes Python to loop through a suite of statements if the test expression evaluates to True.
- The while statement uses the same evaluations for True and False as the if statement.
- Likewise, the same indentation rules are used for a while as they are for an if.
- Each time through the loop, if the condition in the while is True, then the body of the while loop is executed.
- When the condition is False, then the loop is complete and first statement after the while loop is executed.
- While loops often require the use of a counter variable.
- When this is the case, remember to increment the counter variable to prevent an infinite loop!

```
# Declaring a counter
counter = 1

# While the counter is less than or equal to 10 print counter
while counter <= 10:
   print(counter, end=" ")
   counter += 1
```

Output: 1 2 3 4 5 6 7 8 9 10

Range Function

- A range object can also be used to represent a sequence of numbers and then iterate over the range as a sequence with a for loop.
- range(start, stop[,step]) - The start parameter defines the number at which to start.
- If no value is provided for the start, it will default to zero.
- The stop parameter defines the number at which to stop.
- The stop is exclusive, meaning the last number the range returns is one less than the value of stop.
- The step parameter defaults to one.
- If a value is provided for the step, it must be greater than zero and the contents of the range are determined by the formula r[i] = start + step*i where i >= 0 and r[i] < stop
- Negative values can be specified for the step.
- The formula for negative steps is the same but the constraints are i >= 0 and r[i] > stop

```
# using range to print 1 - 10
for i in range(1,11):
   print(i, end=" ")            #1 2 3 4 5 6 7 8 9 10
print()

# using range to print even numbers
for i in range(0, 11, 2):
   print(i, end=" ")            #0 2 4 6 8 10
print()

# using range to print -1 - -10
for i in range(-1, -11, -1):
   print(i, end=" ")            #-1 -2 -3 -4 -5 -6 -7 -8 -9 -10
print()
```

ENUMERATE FUNCTION

- The enumerate function returns an enumerate object which is iterable. Each item returned from the enumerate object will be a tuple.
- enumerate(iterable, start=0)
- The tuple returned will contain two things, a count and the value from the iterable object passed to the function.
- If a value is not passed for the start parameter, it defaults to zero.
- In this case, the tuple will contain the index position and the value at that index position from the iterable object.

```
# storing a tuple in a variable
tup = (1,2,3,4,5)

# iterating through tup
for i in enumerate(tup):
   print(i)
```

Output: (0, 1) (1, 2) (2, 3) (3, 4) (4, 5)

Break and Continue

- Any looping construct can have its control flow changed through a break or continue statement within it.
- When a break is executed, control of the program jumps to the first statement beyond the loop.
- A break is often used when searching through a collection for the occurrence of a particular item.
- When a continue statement is executed, the rest of the suite is skipped for that iteration of the loop and control goes to the next iteration.

```
# Declaring count to be 1
count = 1

# Making a while loop continue on 5 but break on 8
while count <= 11:
   count += 1
   if count == 5:
       print("Count is 5")
       continue
```

```
    elif count == 8:
        break
print(count)
```

Output:

Count is 5

8

Loops with Else

- Both for loops and while loops can have an optional else clause.
- The else clause suite will execute when a loop is terminated normally.
- The else clause suite will not execute when the loop terminates as the result of a break statement.

```
# Declaring count to be 1 and total to 0
count = 1
total = 0

# creating a while loop and using else
while count <= 11:
   count += 1
   total = total + count
else:
   print("The total is:", total)
```

Output: the total is: 77

Introduction

Objectives

- Use lists to store ordered collections of objects.
- Unpack list items.
- Use built-in functions on lists.
- Add and delete list items.
- Sort lists.
- Utilize join and split.
- Access individual list items and slices of lists.
- Use dictionaries to store key/value pairs.
- Add and delete key/value pairs.
- Retrieve individual values.
- Retrieve all keys, values, and key/value pairs.
- Reverse and sort dictionaries.

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of python containers.

Knowledge:

- Identify the components that build program structure in Python
- Understand Python structure and containers.

Skills:

- Working knowledge of to write and modify scripts in order to perform specific functions
- Knowledge of how containers function

Python Objects

Python provides a list object, which is a sequence type that can hold a variety of objects. List objects can be described as: - An ordered collection of zero or more elements. - Similar to an array in other languages. - Dynamic in that items can be added and removed. - List objects support efficient element access using integer indices (slice notation).

List objects can be created in several ways: - Using the `list()` constructor. - Using a pair of square brackets to denote the empty list. Example:

```
li = list()
li2 = []
```

Overview of list objects:

- Using square brackets [ ], separating items with commas.
  `[1, 2, 3 ]`
- Elements in a list can be a mix of any types. `[1, 'strings', [1, 2, 3 ]]`
- Elements in lists are accessed by their index positions.
  `0 1 2`
- Index positions are **zero-based**.
- Elements can be retrieved and set by index.
- Slicing provides the ability to create a new list from a piece of an existing list.
- Using sub-script notation, specify the start and end index.
- The end index is exclusive. `a_list[start:stop:step]`
- All values for the start stop and step are optional.

```
# list of numbers
list = [1, 2, 3, 4, 5]

# printing every other element out
print(list[1:4:2])
#output [2, 4]
```

```
#Both empty list:
[] []
#The number list:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#The string list:
['Moose', 'Jynx', 'Duke']

#The number and string list combined:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'Moose', 'Jynx', 'Duke']

[1, 3, 5, 7, 9, 'Moose', 'Duke']
# Creating two empty lists
empty_list_1 = list()
empty_list_2 = []

# Creating a number and string list
number_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
string_list = ["Moose", "Jynx", "Duke"]

# Combining the lists together
number_list_and_string_list = number_list + string_list

# Printing both empty list
print("Both empty list: ", empty_list_1, empty_list_2)

# Printing both the number and string list
print("The number list: ", number_list)
print("The string list: ", string_list)

# Printing the combined list
print("The number and string list combined: ", number_list_and_string_list)

# Slicing the combined list
print(number_list_and_string_list[::2])
```

- Attempting to access elements beyond the bounds of the list will result in an Index Error being raised.
- The bounds of the list is the index position of the last element.
- The built-in len() function will return the number of elements in a list when a list is passed to it.

```
len(a_list)
```

```
# index out of range example:
list = [1, 2, 3, 4, 5]
print(len(list))
#ouptut: 5
print(list[-6])
#output: IndexError: list index out of range
```

Unpacking List

The elements of a list can be unpacked into individual variables that are often easier to understand rather than referencing them by index. When unpacking lists, the number of variables being assigned values to must be the same as the number of elements in the list. The variables are copies of the data in the list. Changes made to the variables do not affect the list.

Unpacking list into variables

```
numbers = [1,2,3]
one, *other = numbers
print(other)
```

Changing the variable

```
three = "three"
print(three)
print(numbers)

Output:
[2, 3]
three
[1, 2, 3]
```

List Methods

- `append(item)` - Adds a new element to the end of the list, increasing the length by one.
- `extend(iterable)` - Adds each element from the iterable passed in as new elements to the end of the list, increasing the length by the number of items in the iterable.
- `insert(index, item)` - Adds the specified item at the specified index position, shifting existing elements to make room for the new item, increasing the length by one.
- `pop([index])` - Deletes the element at the specified index or the end of the list if no index is provided, decreasing the length by one.
- `remove(value)` - Deletes the first instance of the specified value in the list, decreasing the length by one.

```
# Creating a list called numbers
numbers = [1,2,3,4,5]
extened_numbers = [7,8,9,10]

# Print length of numbers
print("The length of the list: ", len(numbers))
print("The list is: ", numbers)

# Adding elements
numbers.append(6)
print("An element was appended: ", numbers)
numbers.insert(0,0)
print("An element was inserted: ", numbers)

# Extending a list
numbers.extend(extened_numbers)
print("The numbers list extended the extended_numbers: ", numbers)
```

```
Output:
The length of the list:  5
The list is:  [1, 2, 3, 4, 5]
An element was appended:  [1, 2, 3, 4, 5, 6]
An element was inserted:  [0, 1, 2, 3, 4, 5, 6]
The numbers list extended the extended_numbers:  [0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 10]
```

```python
# Creating a list called numbers
numbers = [1,2,3,4,5,6,7,8,9,10]

# pop an element from the list
numbers.pop()
print("Pop last element from list: ", numbers)

# pop an element from the beginning of list
numbers.pop(0)
print("Pop first element from list: ", numbers)

# Remove the value of 5 in the list
numbers.remove(5)
print("Number 5 removed from list: ", numbers)
```

```
Output:
Pop last element from list:  [1, 2, 3, 4, 5, 6, 7, 8, 9]
Pop first element from list:  [2, 3, 4, 5, 6, 7, 8, 9]
Number 5 removed from list:  [2, 3, 4, 6, 7, 8, 9]
```

Sorting Lists

- There are some fundamental differences between the list.sort() method and the sorted(iterable) function.
- The list.sort() method is an instance method and belongs to a specific list object.
- list.sort() will modify the list object to which it belongs.
- list.reverse() will reverse the sort of the list

```python
# Creating a list called numbers
numbers = [5,4,2,1,3]

# Sort the list
numbers.sort()
print("Using the sort method: ", numbers)

# Reversing the list
numbers.sort(reverse=True)
print("Using the sort method to reverse: ", numbers)

# Using reverse method to reverse list
numbers.reverse()
print("Using the reverse method: ", numbers)
```

```
Output:
Using the sort method:  [1, 2, 3, 4, 5]
Using the sort method to reverse:  [5, 4, 3, 2, 1]
Using the reverse method:  [1, 2, 3, 4, 5]
```

```
# Creating a list called names
names = ["Jared", "Adam", "Bill", "Edward"]

# Using sorted function
sorted_name = sorted(names)
print("Using sorted: ", sorted_name)


# Using sorted function, but reversing
reverse_sorted_name = sorted(names, reverse=True)
print("Using sorted with reverse: ", reverse_sorted_name)
```

```
Output:
Using sorted:  ['Adam', 'Bill', 'Edward', 'Jared']
Using sorted with reverse:  ['Jared', 'Edward', 'Bill', 'Adam']
```

Split and Join

Join:

- str objects have methods that can be used to create a list from a str and concatenate str objects in a list to a string.
- The join method concatenates str objects from an iterable object and returns them as a single str.
- `str.join(values)`
- The separator in the returned str is the str to which this method belongs.
- A TypeError exception is raised if the iterable object contains non-string values.

Split

- The split method returns a list of the words in a str object.
- `str.split(sep=Non, maxsplit=-1)`

- If no value is specified for sep, then whitespace is used as the separator.
- The resulting list will not contain any empty strings.
- If a value is specified for sep, consecutive delimiters are not grouped which could result in empty strings in the returned list.
- maxsplit is used to determine the number of times the split should occur.
- If no value is supplied, all possible splits are made.

```
# Storing a string in a variable called string
string = "This is a string I want to turn into a list!"
print(string)

# Turning a string to a list using split
string_to_list = string.split()
print(string_to_list)

# Turning a list to a string using join
list_to_string = " ".join(string_to_list)
print(list_to_string)
```

```
Output:
This is a string I want to turn into a list!

['This', 'is', 'a', 'string', 'I', 'want', 'to', 'turn', 'into', 'a', 'list!']

This is a string I want to turn into a list!
```

Max/Min/Sum

Max

- max returns the largest item in an iterable.
- Key is an optional parameter.
- Default is also an optional parameter.
- If the iterable is empty and a default is NOT provided, a ValueError is raised.
- If the iterable is empty and a default IS provided, the default value is returned.
- `max(iterable, [key], [default])`

```
# Printing the Max of the list number
number = [1,2,5,9,10,100]
print(max(number))
```

```
Output:
100
```

Min

Min behaves the same as max except it returns the smallest item in an iterable. It has the same optional arguments with the same behavior. min(iterable, [key], [default])

```
# Printing the Min of the list number
number = [1,2,5,9,10,100]
print(min(number))
```

```
Output:
1
```

Sum

- Sums the value of start and each item in an iterable.
- Start is an optional parameter.
- The items in the iterable must be numeric.
- The MIN, MAX, and SUM functions will raise an TypeError if the iterable contains numbers and strings.
- `sum(iterable, start=0)`

```
# Printing the Sum of the list number
Number = [1,2,5,9,10,100]
print(sum(number))
```

ALL/ANY

- Another common problem to solve is determining if all elements in a list are True or if any element in the list is True. Python provides built-in functions for this purpose.
- `all(iterable)`
- The all function returns True if every element in the iterable is True or if the iterable is empty.
- `any(iterable)`
- The any function returns True if any element in the iterable is True. False will be returned if the iterable is empty.

```
# Printing all
number = [0,1,2,3]
print(all(number))

# Printing all
Number = [0,1,2,3]
print(any(number))
```

```
Output:
False
True
```

Lists Containing Lists

- Items within a list can be any Python object. This includes other collections, custom classes, etc.
- When working with a list as an item in a list, items can be accessed using indexing.
- If only one index is supplied then the inner list will be returned.
- Two sets of square brackets '[]' are required to access an item in the inner list.

```
# Creating a multi-list
multi_list = [[2,4,6,8,10], [1,3,5,7,9]]

# printing both lists
print("Both lists: ", multi_list)
print("First list: ", multi_list[0])
print("Second list: ", multi_list[1])

# Print the last element in the first list
print("Last element in first list: ", multi_list[0][4])
```

```
Output:
Both lists:  [[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
First list:  [2, 4, 6, 8, 10]
Second list:  [1, 3, 5, 7, 9]
Last element in first list:  10
```

Iterator Methods

- Many objects in python are classified as iterable. This simply means that can be iterated over and accessed sequentially in a iterative statement.
- Lists and other sequence objects in python can be iterated over using loops.
- You can utilize for or while loops to iterate over an object in Python.
- With a loop we can execute a set of statements for each item in a sequence.

For Loop

- The i variable in this loop raises a question, though: where did the variable name come from?
- We have not defined it previously in our code.
- Because for loops iterate through lists, tuples, etc. in sequence, this variable can actually be called almost anything. Python will interpret any variable name we put in that spot as referring to each list entry in sequence as the loop executes.
- We can name i whatever we want and the loop will take the current element in the sequence loop and assign it locally to that variable.

```
# Creating list
list = ["United", "States", "Army", "Cyber", "School"]

# Iterate through list
for i in list:
   print(i, end=" ")
```

Output:
United States Army Cyber School

Dictionaries

- A dict() (dictionary) is an ordered collection of entries.
- Each entry contains a 'key:value' pair.
- Dictionaries maintain their order of insertion - Last In, First Out (LIFO).
- Dictionary keys have to be both unique and hashable.
- Two ways to create a dictionary, using dict() or {}

```
# Creating an empty dictionary
dictionary = dict()
dictionary2 = {}
```

```
# Example 1 (using {}):
names = {"n1": "Bill", "n2":"Jill", "n3":"Gill"}
numbers = {1:1, 2:2, 3:3}


# Example 2 (using dict()):
also_names = dict(n1 = "Bill", n2 = "Jill", n3 = "Gill")
also_numbers = dict(one = 1, two = 2, three = 3)
```

Individal Keys

- Dictionaries are designed as a data structure that provide fast look-ups into the structure by key. Sub-script notation can be used to add new key/value pairs and to modify existing values for a given key.
- dictionary[key] will retrieve the value associated to the given key.
- If the key does not exist, a KeyError exception is raised
- dictionary[key] = 'value' will either add a new key value pair, or modify the value associated to the given key.
- Dictionaries have a get method that can be used to retrieve the value for a given ke

- `dictionary.get(key[, default])`
- If a value for default is not provided, None will be returned.

```
# Creating two dictionaries
names = {"n1": "Bill", "n2":"Jill", "n3":"Gill"}
numbers = {1:1, 2:2, 3:3}

# Getting the value of key 1
print("Number 1: ", numbers.get(1))
# Getting the value from key n1
print("name 1:", names.get("n1"))
# The key does not exist
print("Name 4:", names.get("n4", "Name does not exist"))
```

```
Output:
Number 1:  1
name 1: Bill
Name 4: Name does not exist
```

Removing Key Values/Pairs

- Removing elements from a dictionary can be done through the pop() and popitem() methods.
- The general syntax for the pop() method is as follows: value = dictionary.pop(key[,default])
- Key represents the key of the key/value pair to attempt to remove.
- Default represents an optional value to return if key does not exist.
- The general syntax for the popitem() method is as follows. a_tuple = dictionary.popitem()
- The popitem() method removes and returns the last key/value pair as a tuple, but raises a KeyError if obj is empty.
- A tuple is an immutable collection of ordered elements.
- The del command can also be used to remove a key/value pair from the dictionary or the entire dictionary.
- del dictionary[key] removes the specified key/value pair from the dictionary, the key/value pair is not returned.
- del dictionary deletes the entire dictionary.

```
# Creating two dictionaries
names = {"n1": "Bill", "n2":"Jill", "n3":"Gill"}
```

```
numbers = {1:1, 2:2, 3:3}

# Pop n1 off of names
remove_n1 = names.pop("n1")
print(names)

# popitem()
remove_3 = numbers.popitem()
print(numbers)

# pop something that does not exist
remove_n4 = names.pop('n4', 'Key does not exist')
print(remove_n4)
```

```
Output:
{'n2': 'Jill', 'n3': 'Gill'}
{1: 1, 2: 2}
Key does not exist
```

Dictionary Methods

- When there is a need to work with all key/value pairs in a dictionary, the following methods are useful.
- The keys() method returns a view of the dictionary keys.
- The values() method returns a view of the dictionary values.
- The items() method returns a view of the dictionary items.
- Each item in the view returned by items() is a 2 item tuple consisting of a key and a value.
- Each of the views can also be converted to a list or a tuple by passing the view to a list() or tuple() constructor.
- The views returned by keys() and items() are set-like views that can be converted to sets with the set() constructor.

```
# Creating two dictionaries
names = {"n1": "Bill", "n2":"Jill", "n3":"Gill"}

# Using items() to print keys and values
for k, v in names.items():
   print("The keys are:", k, " The values are:", v)
```

```
Output:
The keys are: n1  The values are: Bill
The keys are: n2  The values are: Jill
```

```
The keys are: n3  The values are: Gill

```

```
# Creating two dictionaries
numbers = {'one':1, 'two':2, 'three':3}

# Using key() and value()
for key in numbers.keys():
   print(key)

for value in numbers.values():
   print(value)
```

```
1
2
3
```

Finding Keys/Values

- The in and not in operators can be used to determine if a specific key or value exists in the dictionary.
- Both operators will return a True or False value.
- When used with the entire dictionary, the operators search for the value in the keys of the dictionary.
- The view returned by the values method can be used to search for a value in the values of the dictionary.

```
# Creating two dictionaries
numbers = {'one':1, 'two':2, 'three':3}

# Using in and not in
print("one" in numbers)
print("one" not in numbers)
```

```
Output:
True
False
```

Sorting Dictionaries

- The only way to sort a dictionary is to convert the keys or values to a list and call the sort method to sort the list.
- A sorted list of the keys will be returned by default. You can specify the items() or values() if you wish to have them sorted as well.
- To sort the resulting list objects by the values in the dictionary, use the get method of the dictionary as the key of the sort method and the sorted function.

```
# Creating two dictionaries
numbers = {'one':1, 'two':2, 'three':3}
      new_numbers = sorted(numbers.values())
      print(new_numbers)

# Turning the dictionary numbers to list
numbers_list = list(numbers)
print(numbers_list)

# sorting the list
numbers_list.sort()
print(numbers_list)
```

```
Output:
The sorted values: [1, 2, 3]
The unsorted list: ['one', 'two', 'three']
```

Sets and Tuples

Sets

- Sets are not ordered, do not allow for duplicates, and immutable.
- To create a set use set()
- `s = set()`
- Sets also allow for different data types.

```
s = {1, 2, 3, False, "String"}
s_two = set((1, 2, 3, False, "String"))
```

```
Output:
{False, 1, 2, 3, 'String'}
{False, 1, 2, 3, 'String'}
```

Tuples

- Tuples are immutable, meaning that they are not changeable.
- Tuples must be ordered.
- Tuples ALLOW duplicate values.
- Tuples are zero-based indexed.
- Can create an empty tuple using () or tuple()

```
tup = (1,1,2,3,4,5,6)

for i in tup:
    print(i, end=" ")
```

```
Output:
1 1 2 3 4 5 6
```

Introduction

Objectives

- How to open files for I/O operations
- How to close files automatically (the with statement)
- How to read text from a file
- How to write text to a file
- How to use the seek() and tell() file methods
- How to read and write raw (binary) data to a file

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of file inputs and outputs.

Knowledge:

Identify the components that build program structure in Python Understand Python input and output files.

Skills:

Working knowledge of to write and modify scripts in order to perform specific functions Knowledge of input and output works with files in Python.


DISA

File Overview

To handle files in Python, there are three key things to know:

`open()` `read()` or `write()` `close()`

```
# open file in write mode, and store it in a variable
file_obj = open("info.txt", "w")

# write to a file
file_obj.write("Hello World")

# close the file
file_obj.close()
```


DISA

Opening Files

- Performing file input/output operations is, arguably, one of the most important features of a programming language.
- Fortunately, Python has language features that greatly facilitate this important task.
- Use the `open()` function to prepare a file for I/O operations.

Specify:

- File name as string or path expression. The name/path may be absolute or relative.
- Remember, for Windows OS, the is the path separator versus the / which is the path separator for Linux/Mac.

File Modes

Mode options allow for the ability to read or write to a file. Some modes require two characters while others only require one. The list below shows all the available mode options:

Keyword	Description
r	reads from file, this is used by default
w	writes to a file, will overwrite if file exists
a	appends to a file, will not overwrite
x	creates a file if it does not exist
b	allows for reading or writing to binary
t	text mde, can read or write

Keyword	Description
w+ or r+	file must eist, can read and write

How to open a file in python

To open a file, use the open() function. Put either the file or the path in the parenthesis, followed by the mode. Below are examples of opening files and storing them in a variable.

```
# Create a file object and open file in read mode
file_obj = open("info.txt", "r")

# Create a file object and opens file in write mode
file_obj = open("info.txt", "w")

# Create a file object and opens file in append mode
file_obj = open("info.txt", "a")

# Create a file object and opens file in read binary mode
file_obj = open("info.dat", "rb")

# Create a file object and opens file in write binary mode
file_obj = open("info.dat", "wb")

# Create a file object and opens file in append binary mode
file_obj = open("info.dat", "ab")
```

Reading Files

To read one line at a time, open the file and use the `read()` method `read()` returns a string of all the contents from file

```
# open file in read mode, and store it in a variable
file_obj("info.txt")

# reading from a file
file_content = file_obj.read()

# close the file
file_obj.close()
```

`readline()` reads one line of the text file at a time.

```
# With automatically closes file
with open("info.txt") as file_obj:
   print(file_obj.readline())
```

`with` is used to do everything that has been taught but NOW there is no need to close because it **automatically** does it

The `readlines()` method returns the whole text file in the form of a list!

```
# Opening info.txt and reading the lines
with open("info.txt") as file_obj:
   file_content = file_obj.readlines()
   print(file_content)
```

- When using the readlines() method, it returns a list of strings from the file.
- The readlines will also return the new line character \n, but there is a way to remove it as shown below:

```
# Opening info.txt and reading the lines
with open("info.txt") as file_obj:
   file_content = file_obj.readlines()
   # Creating an empty list
   li = []
   # Iterate through the list
   for i in file_content:
       # Stripping the new line character
```

```
        li.append(i.strip())
    # Iterate through the new list
    for j in li:
        # print out the new list
        print(j)
```

Writing Text to a File

`write()` - writes a single string to a file `writelines()` - writes a list of strings to a file, does not add newlines

```
data = [("California", 39937489, "Sacramento"),
       ("Texas", 29472295, "Austin"),
       ("Florida", 21992985, "Tallahassee")]

with open("info.txt", "w") as data_obj:
       for state, population, capital in data:
           data_obj.write(f'{state}, {population}, {capital}\n')
```

Note: If info.txt was already created, using 'w' mode will overwrite the file. Using 'a' will not overwrite the file, it will add the contents to the end of the file.

```
# open file in write mode, and store it in a variable
file_obj = open("info.txt", "w")

# write to a file
file_obj.write("Hello World")

# close the file
file_obj.close()
```

```
# open file in append mode, and store it in a variable
file_obj = open("info.txt", "a")

# write to a file
file_obj.write("Hello World")

# close the file
file_obj.close()
```

Closing Files

- A Python program could open a file, do the I/O operations, and close the file by issuing Python functions.
- The `close()` method closes the file and may be explicitly coded as shown in the listing below.
- Python will automatically close files when I/O operations are done in concert with the with statement.
- Remember with closes the file automatically


DISA

Seek and Tell

- The Python functions `seek()` and `tell()` allow a program to navigate the read/write pointer in a file and to report on the current position of the read/write pointer.
- `seek( pos )` to position file at position pos for next I/O operation.
- For example, `seek(0)` positions the read/write pointer to the beginning of the file. The `seek()` function takes an optional second parameter called whence which is an integer specifying where to start seeking from.
- `seek( pos, whence )` when whence = 0 (the default value) tells Python to advance the pointer at the start of the file.
- For example, the call `seek( 100, 0 )` tells Python to advance the read/write pointer 100 characters from the start of the file.
- `seek( pos, whence )` when whence = 1 tells Python to advance the pointer at the current read/write position.
- For example, the call `seek( 100 )` followed by `seek(10, 1 )` tells Python to _first advance 100 characters from the start of the file followed by advancing 10 characters from the current position, which is 100.
- `seek( pos, whence )` when whence = 2 tells Python to advance the pointer first to the end of the file, then back up the read/write position by pos characters.
- In short, whence = 2 tells Python to move the read/write pointer from the back of the file.
- The `tell()` function returns an integer specifying the current position within the file of the read/write pointer.

```
# Open a file and print tell
with open("info.txt") as file_obj:
  print(file_obj.tell())
```

```
# Open a file and print tell
file_obj = open("info.txt")

# Setting the pointer to beginning of file
file_obj.seek(0,0)

# Printing the first line
print(file_obj.readline())
```

```
9
10  # Closing the file
11  file_obj.close()
```

Introduction

Objectives

- How to define a function
- What the term 'Name scope' means
- How to call functions
- How to code functions that return values
- What parameter and argument types can bes used with Python functions
- How to pass a varying number of positional parameters
- How to pass a varying number of keyword parameters
- How to use several parameter types in one function
- What a lambda function is and how to code them

TLO Knowledge and Skills

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of writing functions.

Knowledge:

Identify the components that build program structure in Python Understand writing functions.

Skills:

Working knowledge of to write and modify scripts in order to perform specific functions Knowledge of writing functions in Python.


DISA

Function Definitions

- To define a function, use the keyword def followed by a function name followed by a parenthesized list of parameters, if needed, followed by a colon.
- A function must be defined before it can be used (called).
- A function in Python defines a block. As with other blocks in Python (if/elif/else statements, and while/for statements) the code must be indented by using all spaces or all tabs.

```
1  def function_name(parameter1, parameter2): #  With parameters
2
3  def function_name():                       #  Without parameters
```

Calling Functions

A function call calls the function that is defined which will execute what is in the function. When a function is called, the arguments are the data passed into the method's parameters.

```
 1  # Creating a function called function_name
 2  def function_name(parameter1, parameter2):
 3     print(parameter1, parameter2)
 4
 5  # Creating two variables and calling the function
 6  parameter1 = ["This", "is", "a", "argument"]
 7  parameter2 = ("This is a argument")
 8  function_name(parameter1, parameter2) #  This is where the function is
 9  'called'
10      Output:
        ['This', 'is', 'a', 'argument'] This is a argument
```

Returning Data

- Functions often return values via the return statement.
- It is common to save returned values for later use BUT if it is not needed, the function call is equivalent to the returned value when used by the calling code.

```
# Creating a function and returning the output
def func_1(num):
   return num

print(func_1([1,2,3]))

    Output:
    [1, 2, 3]
```

Arguments

- Arguments passed to Python functions may be of any type.
- It is the programmer's responsibility to ensure that the type of data passed is appropriate for how the data is used in the function.

```
# Defining a function with a single parameter
def function(para1):
   print("An argument was passed to this function: ", para1)

# Function call and passing in an argument
function([1,2,3])
```

- TypeError is one of several Python runtime errors that get caught by compilers of strongly, statically typed languages (C#, C++, etc.)

- Argument data is assigned to parameters.
- Python treats assigning arguments to parameters as assignment, meaning that Python does not copy the argument data into the parameters (recall that in Python, assignment never copies data).
- Python is flexible in the composition of parameter lists for function definitions and allowable data passed to functions as arguments.
- Positional parameters are bound to argument data by their position in the function definition.

```
def func(num1, num2):
   print(num1,num2)

func(num1= 1, num2=2)
#ouput 1 2
```

Keyword Parameters

- Keyword arguments are arguments that can be bound to data by their parameter name.
- Required parameters must have argument data bound to the parameters.
- Optional arguments need not have argument data bound to the parameters.

- For optional arguments, the bound data may not be argument data; bound data may come from the function definition.
- A keyword argument is coded with the name of the corresponding parameter. The argument is coded with the name, not the parameter.

```
1  def func_1(para):
2     return(para)
3
4  print(func_1(para=100))
5  print(func_1(para = 'This is a string!'))
```

Do you see it

the first call to func_1() uses a keyword argument; the second call does not.

- Python allows the use of both keyword arguments and non-keyword arguments.
- However, the non-keyword argument must be coded before the keyword argument.

Optional Arguments

- Optional arguments depend on default parameter values.
- When the optional argument is not included when calling the function, the function uses the default value coded in the function definition.

```
1  def func_1(para1, para2):
2     return (para1, para2)
3
4  print(func_1(para1=100, para2 = "This is a string"))
5  print(func_1(para1 = 'This is a string!', para2 = [1,2,3]))
6
7  #output:
8  #(100, 'This is a string')
9  #('This is a string!', [1, 2, 3])
```

- Default arguments must follow non-default arguments.

- Python requires that default parameters must follow non-default (required) parameters as shown below:

```
def func_1(para1 = "This is a string", para2):
   return (para1, para2)

print(func_1(para1, para2))
#SyntaxError: non-default argument follows default argument
```

Arbitrary Arguments

- Python allows for function to treat a sequence parameter as a single argument or as a group of arguments corresponding to sequence elements.
- Code for a varying number of positional parameters of unknown number as follows:

```
def func_vary (*arbitrary_args):
```

- Python may pass several arguments to func_vary():

```
func_vary(12, "A string", [1,2,'a'])
```

```
def func(*arbitrary_args):
   print("Number of arguments passes: ", len(arbitrary_args))

func(1,2,3,4)
func("String")
func([1,2,3,4])

#ouput:
#Number of arguments passes:  4
#Number of arguments passes:  1
#Number of arguments passes:  1
```

```
def func(*arbitrary_args):
   print(arbitrary_args)

tup = (1,2,"string")
# unpacking the tuple
func(*tup)
# seeing it as 1-element tuple
func(tup)
#output:
#(1, 2, 'string')
#((1, 2, 'string'),)
```

- Python allows for function to treat a dictionary parameter as a single argument or as a group of arguments corresponding to name/value pairs.

- Code the function definition that accepts a varying number of keyword arguments as follows: `def func(**several_NV_pairs):`

```
def func(**dict_arg):
   print("The len of args: ", len(dict_arg))
   print("arguments: ", dict_arg)

func()
func(arg1 = "String", arg2 = [1,2,3], arg3= (1,2,3))
#output
#The len of args:  0
#arguments:  {}
#The len of args:  3
#arguments:  {'arg1': 'String', 'arg2': [1, 2, 3], 'arg3': (1, 2, 3)}
```

Name Scopes

- Functions provide a nested namespace (sometimes called a scope), which localizes the names they use, such that names inside the function won't clash with those outside (in a module or other function). We usually say that functions define a local scope, and modules define a global scope.

The four Python scopes:

1. The `local` scope are names created inside a function.
2. The `enclosed` scope are names created in a function that encloses (contains) another function. This section includes some examples of names defined in the enclosed scope.
3. The `global` scope are names created inside a script outside of any/all functions.
4. The `built-in` scope are names that Python reserves for its own internal workings.
5. When Python looks to resolve (use) a name, it examines the scopes 'inside-out'.
6. Python looks at the most restrictive scope first and works its way outward to increasingly larger scopes.
7. Pythonistas express this concept as the `LEGB` rule.
8. Python looks first in local scope, followed by enclosed, followed by global, followed by builtin (LEGB).
9. Each function contains its own scope; if a Python program has six functions that program contains six local scopes. The scope is activated when the function is called and disappears when the function terminates (either normally or abnormally).
10. A function cannot change the value of data bound to a name outside its scope (without some extra work). If a Python function uses the name defined in global (or any enclosing) scope, Python creates a new name known only in the local scope and binds data to it.

```
# Example of local scope
def func():
   within_function = "This is locally defined"
   print(within_function)

func()
# Output: This is locally defined
```

```
# Example of a Enclosed Scope
# Creating a function and defining num
def func_1():
   num = 20
   # Creating a function and printing num and num2
   def func_2():
       print(num)
       print(num2)
   func_2()
num2 = 50
func_1()
#output:
#20
#50
```

- The name `outside_func_scope` is defined (known) in the global scope. Functions may access names outside their scope provided the scope is an outer scope (E, G or B scopes).

```
#Global Scope
# Declaring a global variable
global_var = "I am global"

# Creating a function and changing global_var
def func_1():
   global_var = "New global"
   print(global_var)

# Creating a function and printing global_var
def func_2():
   print(global_var)

func_1()
func_2()

#output:
#New global
#I am global
```

Lambdas

- lambda p1, p2, … pn:
- lambda is a Python keyword
- p1, p2, …, pn are parameters to the lambda function
- is the code executing when the lambda is called.
- A lambda may contain only one statement.
- Lambdas cannot contain certain Python statements.
- In particular, the return, pass, assert, raise statements may not be coded in a lambda.
- Lambdas always return a value.

```
y = lambda x : x % 2
print(y(5))
# Output:
# 1
```

Introduction

Objectives

- Introduction
- Define the principles of Object Oriented Programming (OOP)
- Define a Python Class
- Describe a Constructor
- Identify Properties of a class
- Demonstrate Coding Class methods (PE)
- Summary

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of object oriented programming.

Knowledge:

Identify the components that build program structure in Python. Understand object oriented programming.

Skills:

Working knowledge of how to write and modify scripts in order to perform specific functions. Knowledge of object oriented programming.


DISA

Objects

Object Properties:

- Programmers speak of an 'OO style', or paradigm, of software development. The style involved organizes apps by creating modules called objects.
- An **object** in OO style is a code module that models an entity identified in the problem domain.
- The model stresses that the object is a happy amalgam of data and procedure, often expressed as object attributes and object behaviors.
- **Object attributes** are coded as Python names bound to values.
- **Object behaviors** are coded as Python functions (called methods in OO-speak).
- The OO methodology calls for breaking problems into code modules (objects!) that represent things (entities) in the problem domain.
- The different modules (each one representing something) communicate with each other during the app's execution.
- One module may send a message to another module which may cause the other module to react (execute a method corresponding to a behavior) which may change data in the object and/or send a message to another object, and so one.
- The objects changing their data, executing methods in accordance to their designed behaviors result in an end state of the completion of the application's work (the reason the app was developed in the first place).

Creating Objects

- In Python (and other OO languages) the programmer uses a special function called the class **constructor** (often referred to as the CTOR) to create objects.
- In the code defining the empty objects of class AccountCreation, Python uses a default constructor which is automatically run and creates empty objects.
- Since empty objects are of limited utility, the Python class CTOR may be coded to make objects more useful.

There are some attributes and behaviors that a AccountCreation could possess: - (Attribute) an account identifier - (Attribute) an account email - (Attribute) an account status (created or not) - (Attribute) an account username - (Attribute) an account password - (Behavior) closing an account - (Behavior) creating the account

- Tell Python what attributes/behaviors are used/possessed by the object.
- The programmer may assign attributes to an object by coding the attribute assignment in its class constructor.
- The class constructor is a special Python method coded as shown below:

```
def ___init___(*self*, att1_val, attr2_val, ..., attrn_val):
```

```
class Account:
    def __init__(self, email, username, password):
        self.email = email
        self.username = username
        self.password = password

def main():
    new_account = Account("email@email.com", "username", "password")
    print(f'{new_account = }', f'{type(new_account)}')

if __name__ == "__main__":
    main()

#new_account = <__main__.Account object at 0x7fa3587eb040> <class
'__main__.Account'>
```

The line of code below creates a BankAccount object:

```
new_account = Account("email@email.com", "username", "password")
```

As Python executes code in the CTOR to create the object new_account, Python uses the name self to reference the new_account object; the object currently being accessed.

- Any name attached to self with dot notation becomes an attribute of the object.
- In general, identify members of the object within the class definition by using the name self.
- The CTOR accepts data that will be assigned to object attributes like any data passed as arguments to Python functions.
- However, note the lines calling the CTOR and passing data:

```
    new_account = Account("email@email.com", "username", "password")
```

- Do not pass any value(s) for self!
- Python uses the self argument for its internal use, identifying members of the object.
- The remaining parameters are used to assign values to the object's attributes, which in this case are _email, _username and _password.
- So, "email@email.com" is assigned to _email, "username" to _username, and "password" to password.
- The output of the above is the same as what has been seen.
- This class, as coded, creates Account objects with attributes but, sadly, is still useless because there is no code that allows a user of the class to manipulate any Account object.
- The class needs methods to implement its behaviors (create and delete accounts)

```
class Account:
   def __init__(self, email, username, password):
       self.email = email
       self.username = username
       self.password = password

def main():
   new_account = Account("email@email.com", "username", "password")
   new_account2 = Account("email2@email.com", "username2", "password2")
   print(f'new_account created an account with username:
{new_account.username} and password: {new_account.password}')
   print(f'new_account created an account with username:
{new_account2.username} and password: {new_account2.password}')

if __name__ == "__main__":
   main()

'''
output:
new_account created an account with username: username and password: password
new_account created an account with username: username2 and password:
password2
'''
```

- Not all object attributes require data from the users of the object. - Account objects (as described above) have other attributes: an account identifier, date opened and status.

- These attributes are not set by the user; data used to set these attributes are not passed by the CTOR.
- Account holders of an account do not set their own account identifier.

```python
class Account:
    def __init__(self, email, username, password):
        self.email = email
        self.username = username
        self.password = password

        self.account_id = f"account_id: {randrange(1, 1_000_000)}"
        self.status = "Opened"
```

Classes

Class Properties:

- A Python class is defined in its own module and should be imported for use.
- A Python class also defines a data type which is the name of the class.
- The class allows a Python program to create objects of that class.
- The objects are the code modules containing data relevant to that object and methods (coded like functions) that describe what the object does; the object's behaviors.

Creating a class

```
class AccountCreation:
   pass

def main():
   my_account = AccountCreation()
   your_account = AccountCreation()
   print(f'{my_account}: ',f'{type(my_account)}')
   print(f'{your_account}: ',f'{type(your_account)}')

if __name__ == "__main__":
   main()



'''
<__main__.AccountCreation object at 0x7f8c2b5f0040>:  <class
'__main__.AccountCreation'>
<__main__.AccountCreation object at 0x7f8c2b5d7d30>:  <class
'__main__.AccountCreation'>
'''
```

- By convention, Python programmers code attributes with a leading underscore which signals any programmer viewing the code that these names are private - not to be used outside the class.
- Python does not enforce this; the programmer is free to use these names outside the class but is considered extremely poor practice to do so.

- With direct access to object attributes, the following code is legal:

```
    new_account.account_id = 302412
    print(f'new_account created an account with username:
{new_account.username} and accountID: {new_account.account_id}')
    '''
    Output: new_account created an account with username: username and
accountID: 302412
    '''
```

- The @property is known as a Python decorator and is used to tell Python that the name of the method may be accessed by an object using dot notation, as shown in the print statements above.
- The decorator @property tells Python that the method name is actually used to reference the attribute of the object.

```
    @property
    def new_email(self):
        return self.email

    @property
    def new_username(self):
        return self.username

    @property
    def new_password(self):
        return self.password
```

```
class Account:
    def __init__(self, email, username, password):
        self.email = email
        self.username = username
        self.password = password
    @property
    def new_email(self):
        return self.email

    @property
    def new_username(self):
        return self.username

    @property
    def new_password(self):
        return self.password

def main():
    new_account = Account("email@email.com", "username", "password")

```

```
    print(f'new_account created an account with username:
{new_account.new_username} and password: {new_account.new_password}')

if __name__ == "__main__":
    main()

'''
Output:
new_account created an account with username: username and password: passwo
'''
```

- Users of a class may need to change an object's attributes but the creator of the class should allow a way of users to change attributes without violating encapsulation.
- What is needed is a way of managing attribute changes so that any change in an attribute value follows some business process/rule.
- Python uses decorators to define methods that change attribute values often called setter methods.
- By defining a setter method to govern the changing of an attribute value, the method may check values that the attribute would assume to determine if the values meet the restrictions specified in the business rules.
- The setter method has guard code that enforces the allowable values of attribute values based on code implementing business rule.
- Python has another mechanism that programmers use in setter methods that guard against attributes containing invalid values.
- This mechanism is the try/except construct and is covered in the lesson on Python Exception Handling.
- Assume these business rules apply to the attributes of all Account objects:
- The Account ID must start with the letters "account_id" followed by one or more digits

```
  @acc_id.setter
   def acc_id(self, value):
       if isinstance(value, str) and value.startswith("account_id") and len(
value ) >= 1:
           self.acc_id = value
```

```
from random import randrange

class Account:
    def __init__(self, email, username, password):
        self.email = email
        self.username = username
        self.password = password
```

```
        self.account_id = f"account_id: {randrange(1, 1_000_000)}"
        self.status = "Opened"

    @property
    def acc_id(self):
        return self.account_id

    @property
    def new_username(self):
        return self.username

    @property
    def new_password(self):
        return self.password
    @acc_id.setter
    def acc_id(self, value):
        if isinstance(value, str) and value.startswith("account_id") and len
value ) >= 1:
            self.acc_id = value


def main():
    new_account = Account("email@email.com", "username", "password")
    print(f'new_account created an account with username:
{new_account.new_username} and account id: {new_account.acc_id}')

if __name__ == "__main__":
    main()

'''
Output:
new_account created an account with username: username and account id:
account_id: 242334
'''
```

- Methods describe what an object does.
- The syntax for coding methods is similar to that of coding properties but methods do not require any decorators.
- An object, or instance method is coded as a function that applies Python code to implement business rules (like a property setter).
- Python knows instance methods by having the name self as the first argument, again like property accessors.
- The code below shows implementing a 'close account' behavior.
- The business rule is only an open account may be closed; an attempt to close an already closed account results in an error (diagnostic):

```
    def close_acc(self):
        if self.status == "Opened":
            self.status = "Closed"
        else:
            print("Account already closed")
```

```
from random import randrange
class Account:
    def __init__(self, email, username, password):
        self.email = email
        self.username = username
        self.password = password

        self.account_id = f"account_id: {randrange(1, 1_000_000)}"
        self.status = "Opened"

    @property
    def acc_id(self):
        return self.account_id

    @property
    def new_username(self):
        return self.username

    @property
    def new_password(self):
        return self.password


    def close_acc(self):
        if self.status == "Opened":
            self.status = "Closed"
        else:
            print("Account already closed")

def main():
    new_account = Account("email@email.com", "username", "password")
    print(f'new_account created an account with username:
{new_account.new_username} and account id: {new_account.status}')
    new_account.close_acc()
    print(f'Account status is now: {new_account.status}')

if __name__ == "__main__":
    main()
'''
Output:
new_account created an account with username: username and account id: Opened
Account status is now: Closed
'''
```

Introduction

Objectives

- Use python to craft and receive TCP/IP packets.
- Utilize 3rd party python libraries to establish network connections.
- Use the socket library in Python.
- Create Python Sockets.
- Create and manage socket servers to exfiltrate data via python script.
- Create a continuous listening server with Python.
- Create a Port Scanner in Python.

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of Python Networking.

Knowledge:

Identify the components that build program structure in Python. Understand Python Networking.

Skills:

Working knowledge of how to write and modify scripts in order to perform specific functions. Knowledge of Python Networking.


DISA

Networking Overview

"An endpoint for communication."

```
Server: Generally a service on a remote server waiting to recieve
connections.

Client: Depending on the context, it can be either the device or the
application (i.e. a web browser) being used to connect to a server.

Stream Socket: Performs like streams of information. There are no record
lengths or character boundaries between data, so communicating processes must
agree on their own mechanisms for distinguishing information (i.e. connection
oriented). Stream sockets are most common because the burden of transferring
the data reliably is handled by TCP/IP, rather than by the application.

Datagram Socket: The datagram socket is a connectionless service. Datagrams
are sent as independent packets. The service provides no guarantees. Data can
be lost or duplicated, and datagrams can arrive out of order. The size of a
datagram is limited to the size able to be sent in a single transaction.

User Datagram Protocol (UDP). Connectionless communication method; sends
datagrams.

Transmission Control Protocol (TCP). Connection oriented communication
method; sends a data stream.
```

Primary Socket Methods

Some of the calls we make will be synchronous, which means they are blocking calls. They will stop the flow of the program until "something happens" (dependent on the call we made).

- .accept()

```
Accepts a connection. The socket must be bound to an address and listening
for connections. The return value is a pair (conn, address) where conn is
a new socket object usable to send and receive data on the connection, and
address is the address bound to the socket on the other end of the
connection.
```

- .bind(address)

```
Bind the socket to address. The socket must not already be bound.
```

- .close()

```
Mark the socket closed. The remote end will receive no more data (after
queued data is flushed).
```

- .connect(address)

```
Connect to a remote socket at address.
```

- .listen()

```
Enables server to accept connections
```

- .recv(bufsize)

```
Receive data from the socket. The return value is a bytes object
representing the data received. The maximum amount of data to be received
at once is specified by bufsize. This is a synchronous function.
```

- .recvfrom(bufsize)

```
Receive data from the socket. The return value is a pair (bytes, address)
where bytes is a bytes object representing the data received and address
is the address of the socket sending the data. This is a synchronous
function.
```

- .send(bytes)

```
TCP Method. Sends data to the socket. The socket must be connected to the
remote socket.
```

- .sendall(bytes)

```
TCP Method. Unlike send(), this method continues to send data from bytes
until either all data has been sent or an error occurs. None is returned
on success.
```

- .sendto(bytes, address)

```
UDP Method. Sends data to the socket. The socket should not be connected
to a remote socket, since the destination is specified by address
```

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

```
The Socket Object. Described in the next section
```

SOCKETS

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

1. The address family (AF_* constants) indicates the protocol used. AF_UNIX is used for interprocess communication. AF_INET and AF_INET6 for IPv4 and v6, respectively. This parameter loosely corresponds to the link and internet layers. For our course, we will only concern ourselves with IPv4 sockets, the default choice.
2. type (SOCK_* constants) indicates whether the socket will be streaming or datagram based (SOCK_STREAM, SOCK_DGRAM). This parameter loosely corresponds to the transport layer protocol. TCP vs UDP.
3. We will be using the proto and fileno default values and they can be ignored. proto is an option to change to a different protocol family, and fileno is a method of feeding in the socket information through a file descriptor.

   Only SOCK_STREAM and SOCK_DGRAM appear to be generally useful.

UDP Sockets

```python
import socket
def udp_echo_service():
    # set up the socket, and bind it to a port
    s = socket.socket(type=socket.SOCK_DGRAM)
    s.bind(('127.0.0.1',12345))
    # waiting to receive data; a blocking call
    print('receiving...')
    data, address = s.recvfrom(4096)
    # Received something!
    print('received',data,'from',address)
    # Now we echo it back to sender
    s.sendto(data,address)
    print('sent',data)

if __name__ == '__main__':
    udp_echo_service()
```

```python
import socket
def udp_echo_client():
```

```
    # set up the socket, no need to bind
    s = socket.socket(type=socket.SOCK_DGRAM)
    # Send something to the server
    print('sending...')
    s.sendto(b'Hello World',('127.0.0.1',12345))
    # Waiting to received data; a blocking call
    print('receiving...')
    data,address = s.recvfrom(4096)
    # Received something!
    print(data,'received from',address)

if __name__ == '__main__':
    udp_echo_client()
```

```
import socket

def udp_echo_service():
    s = socket.socket(type=socket.SOCK_DGRAM)
    s.bind(('127.0.0.1',12345))
    # the loop keeps the service alive
    while True:
        data, address = s.recvfrom(4096)
        print('received',data,'from',address)
        s.sendto(data,address)

if __name__ == '__main__':
    udp_echo_service()
```

```
import socket
def udp_echo_client():
    s = socket.socket(type=socket.SOCK_DGRAM)
    # the sendto was changed to prompt the user for input
    # Note the encoding; remember we have to send a bytes object
    s.sendto(input("Text to send: ").encode("ascii"),('127.0.0.1',12345))
    data,address = s.recvfrom(4096)
    print(data,'received from',address)

if __name__ == '__main__':
    udp_echo_client()
```

TCP Sockets

```
import socket

def tcp_qotd_server():
```

```
    s = socket.socket()
    s.bind(('127.0.0.1',12345))
    # Set up listener
    s.listen()
    # Perform 3-way handshake and get client information
    client_socket, address = s.accept()
    quote = b'Object oriented programs are offered as alternatives to corre
ones.'
    # Use client socket object created above to send data
    client_socket.send(quote)
    client_socket.close()

if __name__ == '__main__':
    tcp_qotd_server()
```

```
import socket

def tcp_qotd_client():
    s = socket.socket()
    s.connect(('127.0.0.1',12345))
    # Because we are getting a small message, the .recv(4096) is not an issue
    data = s.recv(4096)
    print(data)

if __name__ == '__main__':
    tcp_qotd_client()
```

```
import socket

def tcp_qotd_service():
    s = socket.socket()
    s.bind(('',12345))
    s.listen()
    # the loop makes it work continously; i.e. it is now a "service"
    while True:
        client_socket, address = s.accept()
        quote = b'Object oriented programs are offered as alternatives to
correct ones.'
        # .sendall() will divide up your message if it is larger than the
buffer,
        # it sends until complete
        client_socket.sendall(quote)
        client_socket.close()

if __name__ == '__main__':
    tcp_qotd_service()
```

```python
import socket

def tcp_qotd_client():
    s = socket.socket()
    s.connect(('127.0.0.1',12345))
    msg = bytearray() # <- A bytearray to store the parts of message
    chunk = s.recv(4) # <- Receive the first message piece
    while chunk:
        print(msg) # <- To see the message grow
        msg.extend(chunk) # <- adds to bytearray
        chunk = s.recv(4) # <- receives next chunk of msg
    print(msg) # <- prints the completed message

if __name__ == '__main__':
    tcp_qotd_client()
```

Socket's with exceptions

```python
import socket

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
   try:
       s.bind(('127.0.0.1',12345))
       s.settimeout(3)
       data, addr = s.recvfrom(1024)
       print ("received:",addr)
       print ("obtained:", data)
   except socket.timeout :
       print ("No connection!")
```

Introduction

Objectives

- Describe Try and Except
- What is raising an exception
- How to perform a sanity check
- Understanding logging and the different levels of logging
- How to log to a file
- How to debug a python file in VSCODE

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of Python debuggin.

Knowledge:

- Identify the components that build program structure in Python
- Understand Python structure and debugging.

Skills:

- Working knowledge of to write and modify scripts in order to perform specific functions
- Knowledge of how debugging function


DISA

Try and Except

- try: test code
- except: handles an error
- else: continues when no error
- finally: continues to execute without caring about the try and except

```
try:
  x = 1/0
except:
   print("Cannot divide by zero")
else:
   print(x)
finally:
   print("this is a finally block!")
```

```
Output:
Cannot divide by zero
this is a finally block!
```

Raising and Assertions

Raising Exceptions

- When code executes and the code is not valid, it will raise an error.
- When programming, programmers have the ability to raise an error in their code.
- This is a good way to debug functions when there are errors being produced in the function.
- The output of a raised exception is an error message.
- To Raise Exception, use the raise keyword, followed by the Exception() function, followed by a string in the Exception()
- `raise Exception(“Error Message”)`

```
def fun(num):
   if num < -1:
       raise Exception("Number needs to be above 0")
   else:
       print("Number is not negative")

fun(-5)
```

```
Output:
Exception: Number needs to be above 0
```

Assertions:

- Are a sanity check that make sure your program is not doing something wrong.
- Use assert keyword, a condition, and a string
- `assert <condition>, “string”`

```
num = -5
assert num == -5, "Number needs to be -5"

num = 5
```

```
4 assert num == -5, "Number needs to be -5"
5
```

```
1 Output:
2 AssertionError: Number needs to be -5
```

Logging

- Logging helps programmers understand what is happening in their code and what the code is doing. Typically, a programmer will use the print() function to test certain aspects of their program. Using the logging module can prevent having multiple print statements in their program.
- To enable logging, use import logging to import the logging module
- `basicConfig()` function is needed. This allows one to specify the details of the logs

```
import logging

logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
```

- **Level** allows one to define the level of logging they desire
- **format** allows one to define what format they want the log to look like

Levels of logging

- Debug: logging.debug(); used for details about code.
- Info: logging.info(); used for events that are general.
- Warning: logging.warning(): used to show a problem. This is the default level. Will write to stdout.
- error: logging.error(): used to show errors. Will write to stdout.
- critical: logging.critical(): used to show fatal errors. Will write to stdout.

```
1
2
3
4
5
```

```
import logging

logging.debug('debug')
logging.info('info')
logging.warning('warning')
logging.error('error')
logging.critical('critical')
```

```
import logging
# 'level=logging.<level>' can be changed to the level of logging you want to
capture.
#  the 'basicConfig' function can only be run ONCE! This is typically run in
the beginning.
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s
- %(message)s')

logging.debug("About to enter function")
def fun():
   for i in range(1,11):
       logging.debug(f"i is: {i}")
fun()
logging.debug("End of function")
```

```
Output:
 2023-11-21 09:16:14,065 - DEBUG - About to enter function
 2023-11-21 09:16:14,065 - DEBUG - i is: 1
 2023-11-21 09:16:14,065 - DEBUG - i is: 2
 2023-11-21 09:16:14,065 - DEBUG - i is: 3
 2023-11-21 09:16:14,065 - DEBUG - i is: 4
 2023-11-21 09:16:14,066 - DEBUG - i is: 5
 2023-11-21 09:16:14,066 - DEBUG - i is: 6
 2023-11-21 09:16:14,066 - DEBUG - i is: 7
 2023-11-21 09:16:14,066 - DEBUG - i is: 8
 2023-11-21 09:16:14,066 - DEBUG - i is: 9
 2023-11-21 09:16:14,066 - DEBUG - i is: 10
 2023-11-21 09:16:14,066 - DEBUG - End of function
```

Logging to a File

```
import logging

logging.basicConfig(filename="logs.txt", filemode='w', format=' %(asctime)s
%(levelname)s - %(message)s')

logging.warning("About to enter function")
def fun():
   for i in range(1,11):
       logging.warning(f"i is: {i}")
```

```
10
11
12  fun()
    logging.warning("End of function")
```

Introduction

Objectives

- Introduction
- Define compile
- Define Search and Findall
- Define Grouping
- Example of searching through a text file

TLO KNOWLEDGE AND SKILLS

Conditions:

Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of Python Regexes.

Knowledge:

Identify the components that build program structure in Python. Understand object Python Regexes.

Skills:

Working knowledge of how to write and modify scripts in order to perform specific functions. Knowledge of Python Regexes.


DISA

Python Regex

Importing re

- To use regexes in Python, one must first `import re`
- Without the import statement, Python will not know how to use regular expressions

```
1  import re
```

Compile and search

- Before diving into `re.compile()`, one needs to understand regexes and how to pattern match.
- Go to Regex Cheatsheet to learn more about how to build your pattern you are trying to match on.

Compile

- In Python, we must create a regex object that has the pattern one is trying to match.
- For example, lets say that I am wanting to match a cell phone number with this exact pattern `123-456-7894`.
- To do so we must create a regex object as seen below.

```
1  phone = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

NOTE: the `r` must be in from of string in order to ignore the backslashes and read the string as a regular expression.

Search

- Now that the regex is defined and its stored in the variable `phone`, lets search through a string using `search()`
- `search()` only matches on the first instance.
- The results of `search()` will be stored in a variable as seen below:

```
match_object = phone.search("The phone number is 123-789-7484")
```

NOTE: If one prints `match_object`, the output would be:

```
<re.Match object; span=(21, 33), match='123-789-7484'>
```

Group

- Using group, one can print out the match and can also match certain elements in the match.
- To print out the match from the example above, see the print below:

```
print(match_object.group())
```

Groups

- Another way to use `groups()` is to wrap some a portion of the regular expression with ().
- For example, lets say someone is looking for just the area code of the phone number. We could put () around the first three digits and we could print that out.

```
import re
phone = re.compile(r'(\d\d\d)-\d\d\d-\d\d\d\d')
match_object = phone.search("The phone numbber is 123-789-7484")
print(match_object.groups())
```

Findall

- Using `findall`, one can get all the matches returned as a list.
- An example is provided below:

```
import re
phone = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
match_object = phone.findall("The old phone number is 123-789-7484, My new
number is 236-789-8794")
print(match_object)
```

```
['123-789-7484', '236-789-8794']
```

Using findall to search through a file

```python
import re
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
with open("find.txt") as file:
    file = file.read()
    mat = phoneNumRegex.search(file)
    fall = re.findall(phoneNumRegex,file)
print(fall)
```

Cheatography

Regular Expressions Cheat Sheet

by Dave Child (DaveChild) via cheatography.com/1/cs/5/

Anchors

^	Start of string, or start of line in multi-line pattern
\A	Start of string
$	End of string, or end of line in multi-line pattern
\Z	End of string
\b	Word boundary
\B	Not word boundary
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\x	Hexadecimal digit
\O	Octal digit

POSIX

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion
?!	Negative lookahead
?<=	Lookbehind assertion
?!= or ?<!	Negative lookbehind
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Quantifiers

*	0 or more	{3}	Exactly 3
+	1 or more	{3,}	3 or more
?	0 or 1	{3,5}	3, 4 or 5

Add a ? to a quantifier to make it ungreedy.

Escape Sequences

\	Escape following character
\Q	Begin literal sequence
\E	End literal sequence

"Escaping" is a way of treating characters which have a special meaning in regular expressions literally, rather than as special characters.

Common Metacharacters

^	[	.	$
{	*	(	\
+	)	|	?
<	>		

The escape character is usually \

Special Characters

\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed
\xxx	Octal character xxx
\xhh	Hex character hh

Groups and Ranges

.	Any character except new line (\n)
(a|b)	a or b
(...)	Group
(?:...)	Passive (non-capturing) group
[abc]	Range (a or b or c)
[^abc]	Not (a or b or c)
[a-q]	Lower case letter from a to q
[A-Q]	Upper case letter from A to Q
[0-7]	Digit from 0 to 7
\x	Group/subpattern number "x"

Ranges are inclusive.

Pattern Modifiers

g	Global match
i *	Case-insensitive
m *	Multiple lines
s *	Treat string as single line
x *	Allow comments and whitespace in pattern
e *	Evaluate replacement
U *	Ungreedy pattern

* PCRE modifier

String Replacement

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string

Some regex implementations use \ instead of $.

By **Dave Child** (DaveChild)
cheatography.com/davechild/
aloneonahill.com

Published 19th October, 2011.
Last updated 12th March, 2020.
Page 1 of 1.