Table Of Contents (TOC)

TOC - Ending The Book

—- Media Contacts

Instagram - **Totally_Not_A_Haxxer**
Email - **Totally_Not_A_Haxxer@proton.me**

About BHPM (Black Hat Python Manual)

This manual is the second ever written manual in the manual series, out of 10+ core languages, python became the second book I wanted to write. This booklet and manual are obviously going to be directed at the **Python** programming language and will focus on the deeper aspects of its use cases for cyber-security and will be held in the most used version of **Python** which at the time of writing is ***Python (3.9.2)***. The idea of this booklet and manual like all other manuals and books apart from this series are to actually make a good decent flip manual that you can flip to when you need help with a specific task, forget how to do something properly, or even need a small script idea to help sharpen your skills. I should also mention that similar to the ***black hat go manual (BHGM)***, the contents in this book are designed for smaller tasks and ideas and are not designed for production systems, so please do not take any of the code in this book as ***"secure coding"*** or ***"proper standard"*** because most likely it's not! This book also intends to give you a better idea of how Python is structured, its proper use cases, a little bit of a background on the language, what operations Python becomes best with, and also why you should use it as a primary scripting language for security-related tasks!

Primary Purpose of this booklet

As mentioned above, the core purpose of this book is to help you through specific situations where you may have forgotten how to execute a specific task. This may also be for hackers and security researchers who are looking for quick and simple ideas. This book will go over the following topics! Basic Python programming (loops, definitions, modules/classes, better programming, etc), file utilities and processes, general outputs and structure to code, third-party modules, backend systems such as PVM, packet parsing, data manipulation, data processing, web crawling, and other various topics to help you sharpen your skills! We may also go over some popular and higher-rated resources for learning Python, more advanced resources and systems, some popular frameworks and tools for hackers written using python3, and go over some other external resources that will for sure help you! Without much more to talk about, let's jump into this booklet!

About The Python Programming Language

The Python programming language is one of the most popular programming languages and well-known languages; both in and out of the development community. It is because of its popularity that the ideas of Python can be quite interesting to newcomers. For those who are new to the language: Python is a programming language designed and developed by Guido van Rossum and first released in 1991. The language is known as a dynamically typed, interpreted programming language. Before we go on about Python, I would like to get something straight right away that people ALWAYS ask me about and I figured it best to get it down right now more than ever. A ton of people always refer to Python as an interpreted programming language which I also just referred to it as in the pure definition. However, according to computer science, its backend systems and code; as well as its general operational structure; Python is a compiled programming language. No, it is not a machine code compiled language, instead, it acts like the secondary popular programming language Java which uses something known as a virtual machine and byte code compiler to compile the code and interpret it through the virtual machine. We can go two routes of classification here and call Python interpreted, however, it is mainly a compiled programming language because of PVM (Python Virtual Machine) and other various systems that make it compile code into bytecode. When the code is compiled into the bytecode it is then interpreted through the virtual machine. It has been a common argument amongst the community that whatever comes last in the step makes the languages classification, but for the booklet today and for general education; Python is a bytecode compiled programming language, thus earning the title **“byte code compiled”** even though the resulting byte code is interpreted, the direct raw source is not. However, you may see hybrid programming languages such as the Perl programming language that primarily uses standard code interpretation and evaluation but allows users to use the B compiler or perl compiler for bytecode compiling and execution. The purpose of this is so that the user has a choice to maximize or minimize performance since bytecode compilers can be very efficient in some cases, and not so efficient in others. For primary context, byte code execution can be beneficial in cases where execution efficiency is critical. Python is also an object-oriented programming language (OOP) which means that the entire language is object-based. This includes data types such as integers and strings and more complex data structures such as lists, dictionaries, and functions. Basically, an object is considered to be an instance of a class where each object has its own attributes (really just a fancy word for data) and methods (fancier word for functions). For example, if you create an integer variable you are creating an instance of the integer class, and that integer variable has its own set of methods such as `__add__` and `__sub__`. The whole object-oriented thing and methods are sometimes confusing to people, but I promise when you start using it you may get seriously used to it and more familiar with it then finally realize its major benefits! Some of the bigger parts about the Python programming language and why it gained its popularity is not just how easy it is to read to some people but also how the language supports multiple paradigms such as object-oriented, functional, and imperative programming! Python is also heavily used in the cyber security realm for these reasons, but why else may Python be used for security?

Why Python is Used For Security Purposes

Python, as mentioned above, grew insane in terms of its popularity for cybersecurity-related tasks but why? Something I did not mention that you most likely already know is that Python is a general-purpose programming language and has its prime points and tasks where it just succeeds and others where it just does not fit well. So what makes a general-purpose language such as Python good for security? Well, this is primarily because it is considered to be more of a scripting language and has the ability to easily manipulate sets of data making it great for exploit development, having a diverse community of developers makes it have an over-saturated amount of third-party libraries for all kinds of tasks, has a decent mathematical library which makes it great for analytics, is an easy language to extend which libraries such as PwnTools have even mentioned, and many many more other primary examples. You may also notice that popular tools such as SQLmap use and rely on Python, so does commix, Scapy, Wireshark (doesn't rely on but utilize it), Radare2, GEF (GDB plugin), and other various popular tools that you may use daily! There are many good reasons that Python can be used for tooling development, exploit development, and so on. Now, without a doubt, other languages such as Perl and Ruby may beat Python in some cases; but the primary reason hackers do not use perl or Ruby is because of how wackier the syntax is, and on Python's side of things, it's just easier for anyone to pick up. Now that we understand a bit about why it is used in comparison to languages such as Perl and Ruby, we can go ahead with installing the language and getting set up. Note that this booklet will be testing code on ParrotOS which is a popular Debian-based Linux distribution that is used for cyber security and comes with **Python 3.9.X** pre-installed in most versions. If not we will walk through the installation!

Black Hat Python Manual - Design and Development Note

<u>Design Note:</u> One thing I would like to throw out there before talking about Python and before fully diving into everything is the design of this book. I want you to know that this book was literally designed and written out of a ***"basement"*** if you will, it was made with some weird software and some wacky custom programs. So, the formatting of tables while being well off, and while they did come out well, you may notice some weird design choices being made for specific tables with longer sets of code. Instead of making the table half of the width of the page, tables that contain longer pieces of code will either shift left entirely or cover the entire width of the page. This is simply to ensure that readers can understand longer texts and to prevent issues from being made during the printing and editing process of the book! Also, each title has or starts a new part of a section. For example, the Python development primary section is split into multiple sub-sections, for example, ***6.0***. Each subsection is split up with the following syntax **`(Section number)[type] - title`**. The type is the type of section it is. The only sections that should have this are ones with ***`r`*** which indicate that it is a readable section and not a demonstration of code or modules. For example, when looking at module ***6.1*** it reads ***`(Section 6.1) Python - Modules To Look Into`***: as the name implies and the **`r`**, this section is a readable section that does not demonstrate libraries. Hopefully, this makes sense to you! If you have made it this far then I appreciate you reading this- man you are a nerd XD.

<u>Programmatic Note</u>: The code used in this book is not designed for production purposes and is only here for you to actually understand basic to advanced concepts within the language. Over time, people who know me directly reading this book know that I have grown a deep dislike for Python. But it does not mean I know the language- it simply means I have had to deal with the language too much. That being said, this book contains quite some interesting information about the language, expresses some viewpoints and etc that I have found either useful or harmful during my time spent in the language. For notes that are opinionated, I have labeled [***BIAS***]. These are sections that directly and uniquely talk about my opinion based on my experience working with the language itself. I want to have people who read this book not only with the will and intent to learn possibly something new or maybe even to help them remember something but also to read it with an open mind being open to specific opinions and viewpoints towards the world of programming. While I do not believe that an educational book should be biased, I do believe and from experience understand that bias rules the world and bias is the one thing that can motivate researchers to understand something they may not directly know about or can help raise their questions. Now of course, do not get me wrong, there is a ton of negative bias toward specific topics *(not in this book)* but that's what it is. Bias comes in two forms- negative and positive, at least in my world. Negative bias will be the one to corrupt people, has the intent to do damage and has the intent to destroy; meanwhile, positive bias has the intent to help and boost humans to better educate themselves on specific topics which is the type of bias that is included in this book. Please read the notes that are labeled [***BIAS***] with an open mind, and if you feel that said opinion is false: send over your research because the whole point of this book is to get you to look into things yourself in some shape or form! Now, again I hope the statement `definitely send over your research` did not come off as a bit “narcissistic” because it did not have the intent of being so- maybe there is a better way to back that statement? But I feel you ***(the reader***) are smart enough to understand the intent 😎

Informational Note: This should have been a part of the design note but I figured I would make a separate section for it. This book will include specific sections and parts of information that are just so simple it does not even need an explanation. The first real occurrence of this is in section 6.1 when we talk about libraries within Python that are standard and do not require any installation. So, what do I mean directly? This book aims for simplicity, so much simplicity that some things you may be told to Google or some things you may be told are so simple we do not need to directly explain the description. For example, if you are a tech nerd who understands what JSON is or even someone who may be reading this book; you should not have to explain what the JSON library is in Python. Should I? Yeah, probably. But I do not really-need to since the book by that time has already gone over its more advanced concepts. An example in terms of libraries is when we come across the “IP address” library. The library allows us to work with and properly validate IP addresses, as you would have probably imagined by the name hahaha! But anyway, a majority of the things in this book literally 99.9% of it we will explain but some things do not need to be explained in depth like others may. I would also like to mention that some sections in this book discuss my personal opinion and even advice or examples that confused me with wording, phrasing, or cold and annoying statements within Python. This is not a way to say I as a developer am better or stronger or the best and definitely not a way to degrade Python as a language or the libraries. I try to make the notes as obvious as possible that I am clearing up some wacky or fishy business that confused me as a beginner.

Book Wording/Phrasing Note: Some phrases in this book are going to be noted as good terms. For those who are not familiar, I run quite a tight community on Instagram, Discord, and various other platforms and have grown around using the term `nerd` positively. If you are reading this book, you are a nerd, that is that, accept it and own it. You could be rocking with a bunch of bulletproof vests and weapons in your hand but if you own this book- it does not matter, you are a nerd. This is good, the term `nerd` typically refers to someone who is smart and educational, someone who knows what they want in life, and someone who goes for it- keep doing you and keep going through with it.

Note that I might also drop some interesting and weird notes throughout the book that are counted as “**section break off’s”** because at the time of writing something just snaps in my head. For example, I may randomly start talking about space kittens and pizza and then tie it into some random obscure life lesson. I am sorry, but if you are thinking, “**Why**”, just- I mean- you bought a book from a kid who is 16, and spends his time trying to find vulnerabilities in cars for a living on top of someone whose online name is literally **Totally_Not_A_Haxxer…what** did you expect? Okay, I will shut it up. Let's actually get into this LOL.

Installing Python3 - Windows/Linux/MacOS

Installing Python for anyone on Windows and Mac seems to be a pain but I promise even now it's not that hard now. First, we will start off with the most simplistic install which is on Linux, then Windows and MacOS.

- **Linux Install:** In order to install Python3 on your Linux system (in this case again Debian-based systems) we can enter the following command.

```
sudo apt install python3
```

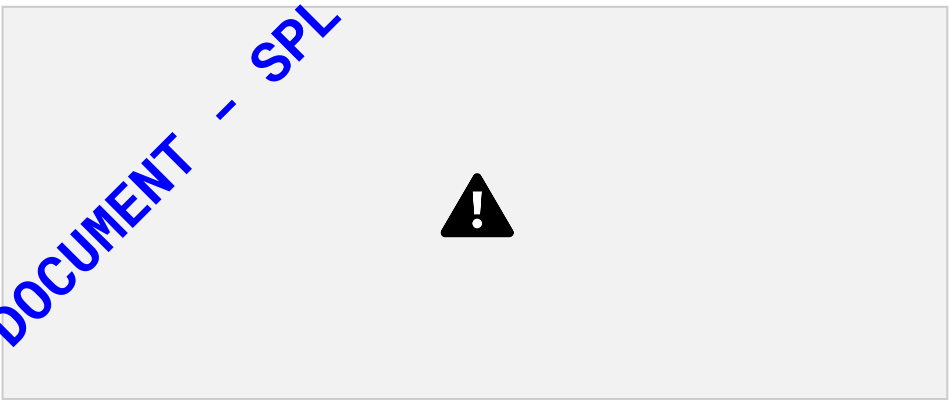
Once entered the install should then start up and you should after installation end up running Python in your system like the command below to check the version.

```
python3 –version
```

If it returns with version `**Python 3.9.2**` then that is a good enough version for the book today. There are other versions of Python, but pythons latest release is `**Python 3.11.4**` at the time of writing. Regardless, the scripts in this book are not extremely reliant on version-specific syntax since we try to eliminate that throughout.

Linux Install - Sub Note: It's important to note that not every Linux system supports apt since apt is a Debian-based package manager. If you use arch linux or Ubuntu you may need to use other package managers such as **yum, yarn, DNF,** and other package managers to install it. Now, the apt package manager is not exclusive to just Debian because other operating systems such as **Linux Mint or Ubuntu may also use APT** but other operating systems such as Fedora, Red Hat, and CentOS use DNF also known as DandifiedYUM which is a fork of APT. Arch Linux also has its own package manager known as Packman and SUSE ***Linux uses Zypper***. It is highly suggested that you look up how Python can be installed on those systems if you are running a different OS.

- **Windows Install (Win10) - The annoying way:** This is just a tell but as of 2023 (the year this book was written and printed), Microsoft now allows you to go to the Microsoft store and easily download and install Python. That is a pretty easy step and I won't be demonstrating that other than a simple image of it in the store since all you literally have to do is type **"python3"** in the terminal and then the Microsoft store will pop up and bring you to it OR you can go to the store app and just search **"python3"**.



- **MacOS Install:** To install Python on macOS it can be a bit tedious but still not horrible to install it. First, like everything else or another type of install, ensure you have the language installed by trying to call it with `` `python3 –version` `` and if the command does not exist try installing it with homebrew. If you do not have homebrew all you need to do is install it with the command below to fetch the install script.

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

When you fetch the install script and run it you can check if homebrew is installed when you run the command to install Python and it installs properly or you can run `which brew` or **`brew –version`** and then the version is outputted or its found. Anyway, in order to install Python we can run the command below

```
brew install python
```

Now that we are done installing Python and should have it working on the operating system we need to have it installed on, we can move on to general command line usage and explaining different flags and usages! See! I told you it was not going to be that bad :D- so now, we hop onto our next section!

Python3 - Command Line Usage And Modifications

I will say from experience, Python has a good range of abilities for its command line usage of things and when you start to use it, you may see the spark that Python has when you optimize it to its peak performance! Now, in this section, we will be going over anything from basic command line usage to advanced command line usage. We are going to start with a basic program, a useless but starting point hello-world program. Typically, I like starting with a good program that can actually do something but since we are only exploring command line options, we do not need anything super complex. Our Python contains a few lines of code which are shown in the boxon the next page.

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


def main(data=”Hello, World!”):
      print(data)

main()
```

I do not think I need to explain this code to anyone reading this book, but I will anyway just for extra reads for people who directly not know what we are doing here. So the very first line holds something known as a shebang which is also known as the hashbang. This is used to indicate the path of the python interpreted that should be used to execute the script especially if we want to run the script as a direct executable without having to call the interpreter. This line is typically placed at the very top of the script and is used primarily on Unix-like operating systems to specify the path. Our second line defines our main function, unlike C, C++, Go, Rust, and other programming languages; Python does not make you use a main function or definition but in this case, it's just good practice to do so. Our main function takes an argument known as data with a pre-assigned value of type String that reads **“Hello, World!”** which will be our message. Since we call the main function with no data to replace the assigned value to **“data”** we just output the pre-assigned message. So a simple program and the basic usage of this program is like so.

```
python3 hello_world.py
```

Or since we added the hashbang we can use the command set that follows on Linux.

```
chmod +x ./hello_world.py ; ./hello_world.py
```

Executing this script is as possibly basic as you can go with the interpreter, but let's explore some command line flags.

Flag	**Description / Short Understanding**
-b	This flag will issue warnings about possibly problematic code like mixing bytes with strings.
-bb	This is similar to the one above but instead of warnings the -bb switch will tell Python to turn those warnings into errors
-B	This flag will disable the creation of .pyc files also known as the compiled Python code from the bytecode compiler
-c <cmd>	Execute the python code as a given string which is useful for running short lines of code without creating or needing to create scripts
-d	Enable parser debugging output for developers which is only really useful for debug builds of Python/developer versions and again is not used for standard development.
-E	Ignore all Python environment variables
-h	Prints help
-i	Inspect interactively after running a specified set of scripts or script. This tells the interpreter to run basically in interactive mode even

	if the input does not appear to be a terminal
-I	Isolate python from the users environment which implies -E and -s flags
-m mod	Run a library as a module script
-0	Remove assert statements and debug dependant code for optimized code generation. Also sets environment variables such as PYTHONOPTIMIZE=x
-00	-0 flag but also discards docstrings
-q	Tells Python to not print version and copyright messages on interactive startup
-s	Do not add user site directory to the systems path (sys.path)
-S	Do not add site module during startup
-v	Verbose mode for better tracing systems
-V	Print version number and exit, if -VV then it will give more info on the build
-W	Control warning message based on specified format
-x	Skip the first line of source code allowing non-unix forms of hashbangs

These are the most well-used flags and all have a bit of a different meaning as mentioned before. I felt that some of them may be more confusing to some people despite a shorter representation so let's go through some of them. Starting from the last one on the list.

- **Flag (-x)**: This flag will skip the very first line of source code, so if we run a script like the one below.

```
print(“hello world”)

print(“hi”)
```

The only thing that will output from that is “**hi**”. This is also a good time to make a note on command line flags before we dig into everything. Command line flags must be specified before the file names you mention or they will not be counted as valid switches. So if we run the command

```
python3 main.py -x
```

Our output from the script above will still be ***“hello world” then “hi”*** because -x was not counted. But if we place it before, we only get ***“hi”*.**

- **Flag (-W):** This flag as mentioned in the table will allow for customization of warnings. Let's generate an example script that will produce warnings so we can actually use this flag. The script below will produce a simple warning. Note that this is a very very basic example and when we touch on error and warning handling in this book, we will go deeper into it.

```
import warnings

def insecure_function():
    password = "secret"
    print(f"Password: {password}")

def main():
    warnings.warn("Insecure password handling detected!",
category=RuntimeWarning)
    insecure_function()

if __name__ == "__main__":
    main()
```

When we run this file with the command shown below -

```
$python3 -W error WSwitchExample.py
```

we get an interesting error message

```
Traceback (most recent call last):
  File "/home/totallynotahaxxer/Desktop/PyBook/scripts/WSwitchExample.py", line
12, in <module>
    main()
  File "/home/totallynotahaxxer/Desktop/PyBook/scripts/WSwitchExample.py", line 8,
in main
    warnings.warn("Insecure password handling detected!", category=RuntimeWarning)
RuntimeWarning: Insecure password handling detected!
```

This is a good point to mention that this flag really will come in handy in more advanced development stages and tooling usages.

- **Flag (-v):** As we mentioned before this will output much more verbose data when running the program. When we use it running our other vulnerable or actually just warning example code that we ran above we get the following output.

```
import _imp # builtin
import '_thread' # <class '_frozen_importlib.BuiltinImporter'>
import '_warnings' # <class '_frozen_importlib.BuiltinImporter'>
import '_weakref' # <class '_frozen_importlib.BuiltinImporter'>
import '_frozen_importlib_external' # <class
'_frozen_importlib.FrozenImporter'>
import 'posix' # <class '_frozen_importlib.BuiltinImporter'>
import '_io' # <class '_frozen_importlib.BuiltinImporter'>
import 'marshal' # <class '_frozen_importlib.BuiltinImporter'>
# installing zipimport hook
import 'time' # <class '_frozen_importlib.BuiltinImporter'>
import 'zipimport' # <class '_frozen_importlib.FrozenImporter'>
# installed zipimport hook
#
/usr/lib/python3.9/encodings/__pycache__/__init__.cpython-
39.pyc matches /usr/lib/python3.9/encodings/__init__.py
# code object from
'/usr/lib/python3.9/encodings/__pycache__/__init__.cpython
-39.pyc'
# /usr/lib/python3.9/__pycache__/codecs.cpython-39.pyc
matches /usr/lib/python3.9/codecs.py
# code object from
'/usr/lib/python3.9/__pycache__/codecs.cpython-39.pyc'
import '_codecs' # <class '_frozen_importlib.BuiltinImporter'>
import 'codecs' #
<_frozen_importlib_external.SourceFileLoader object at
0x7fec8e73d160>

..............................[CUT]..................................
```

As you can see, the output provided above may come in handy when writing or trying to trace and debug specific parts of your program. This flag is pretty easy to understand but it's the output that may confuse people at times.

- **Flag (-S/-s):** Site modules are modules in Python that provide support for site-specific configurations and other various directories such as user site directories. These modules are used during the initialization of the Python virtual environment for important modules and extend the search path for additional directories. Generally speaking, the site module is automatically imported and executed during the startup of the interpreter. Typically, when Python wants to import this module, it searches for a file known as site.py within the standard library which will be the file used for configuration of the runtime environment as specified before. But why do flags -S and -s exist? Well, in some cases, you may want to run Python in optimized mode meaning you do not want it to take extra time to find that module thus disabling the automatic import of the site module. The idea of this is so that Python will not add the user site directory to the module search path and also starts up faster by avoiding importing potentially unnecessary or unused modules. The smaller flag -s will run Python without adding the user site directory to the module search path. Similar to -S the flag will prevent the user site directory from being considered during module imports.

- **Flag (-b/-bb):** One flag that is seriously not used well enough as it should be the **-b** and **-bb** flags. These flags are extremely helpful because they display in warning or error format possibly problematic code within the program. For example, say we have a small script below that is comparing bytes and string which will return with proper flags that there is a small warning.

```
bytes_var = b"Hello, World"
str_var = "Hello, Python"

if bytes_var == str_var:
    print("The bytes and str are equal.")
else:
    print("The bytes and str are not equal.")
```

This code should output no warning when run directly, but the issue here to python which could possibly be problematic in the future is displayed as a warning when **"-b"** is used and an error when **"-bb"** is used. Two examples are shown below

```
┌─[✗]─[totallynotahaxxer@TotallyNotAHaxxer]─[~/Desktop/PyBook/scripts]
└──╼ $python -bb ProblematicCode.py
Traceback (most recent call last):
  File "/home/totallynotahaxxer/Desktop/PyBook/scripts/ProblematicCode.py", line 4,
in <module>
    if bytes_var == str_var:
BytesWarning: Comparison between bytes and string


┌─[✗]─[totallynotahaxxer@TotallyNotAHaxxer]─[~/Desktop/PyBook/scripts]
└──╼ $python -b ProblematicCode.py
/home/totallynotahaxxer/Desktop/PyBook/scripts/ProblematicCode.py:4:
BytesWarning: Comparison between bytes and string
  if bytes_var == str_var:
The bytes and str are not equal.
```

Pretty easy to understand right? One just warns and the other throws an error, that's it! So now what do we do? Well, we have a decent amount of knowledge and decent examples shown above using command line flags. Well, we can now dive deeper into a more interesting realm of things and start talking about the development side of things. Note that I did not go over every flag, I only went over flags that some people may get confused about right now at this moment in time and when we talk about environment variables or more advanced optimization, we can move forth with everything. Now, let's move on to our very next section! Y e s, the nerds are progressing!

Python - Programming And Language Point

We have now reached the programming point of the book, in this section, we will talk about quite a lot so I am going to break the topics down in this section easily right now.

Environment variables, variables, classes, “**modules**”, third party packages, data serialization and deserialization, mathematics, standard libraries, etc.

And other various major hit points for this section, you can imagine that this is one giant tutorial for Python and quick reference. The reason we do this is to just make sure that everyone reading understands the code flow and idea behind the general **“quick references**” behind this book. Also do note that this book expects readers to understand basic code, but for this section, we will be treating you as someone who has never touched Python and wants to learn the general structure of the language.

(Section 1) Python - Common Keywords

Below you will find a table that lists a keyword and then the description of the keyword. These keywords are the top most used keywords in Python that are more common to see.

Keyword	**Description**
if	Standard conditional statement
elif	Else if - standard else if / conditional statement if the original condition is not satisfied
else	Usually added if the original condition before else has failed
for	For loop
while	While loop
True	Boolean true value
False	Boolean false value
with	creates a context manager that automatically sets up and tears

	down resources
as	used to assign the result of the __enter__() method of the context manager to a variable from “with”
def	Method definition
lambda	Defines small anonymous functions
global	Global variable declaration
and	Logical AND
or	Logical OR
not	Logical NOT
break	Break out of loops
continue	Continue through the loop
pass	Placeholder statement
try & except	try to execute something and catch an exception within that label

Now that we have some basic keywords down, we can go ahead and actually move onto the general structure and getting everything started. For the focus of this manual, we are going to start with operations. We will not only explain but be using these later! Section 1 was just an introduction to make sure that in the future you have a good summary of what those keywords do!

(Section 2) Python - Environment Variables

We mentioned this when we walked through the command line usage and the flags with the interpreter but we never actually got to talk about how Python works with environment variables. Since Python has the ability to work with environment variables internally and externally, we can represent them in the basic usage of the interpreter and its flags whilst also demoing flags by running a script which I will show shortly after. Below you will see two bullet points mentioning the standard environment variables used by the interpreter for customization and creating or checking for your own in scripts. I want to make a note that learning environment variables can be extremely helpful if you do not want to use configuration files for your projects and may be more user-friendly!

- **Python Environment Variables and Flags:** As we mentioned above, Python allows users to customize the way Python operates with interpretation or execution with the environment variables if they are used with flags. To show this and represent this a bit more, I have dropped a table of the most commonly used environment variables that you may use in Python for advanced development.

Environment Variable	**Meaning and Usage**
PYTHONPATH	The PYTHONPATH variable is used to extend the search path for Python modules that are going to be used in projects. By default, Python searches for these modules in the directories listed in the **`sys.path`** list but setting this variable to a specific value allows you to add custom directories to that list.
PYTHONVERBOSE	This variable is easy to understand but if you do not get it then it is simply a way of controlling the verbosity of the error messages or warnings Python may spit out during runtime. The easiest usage is to set 1 or True if you want Python to be verbose.
PYTHONIOENCODING	This variable sets the I/O (Input Output) encoding type or the default encoding for standard input (stdin) and standard output (stdout). You might be confused why this is helpful, but to put it in simpler terms; it becomes useful when dealing with non-ASCII characters on different sets of encodings. An example usage is setting it to ***utf-8*** like **`PYTHONIOENCODING=utf-8`**
PYTHONFAULTHANDLER	The fault handler variable is used to enable or disable the fault handler within Python. When this is set to a non-empty value which is either 1 for true or just "true" the fault handler is enabled and Python will make its superstar

	best attempt to provide additional information about segmentation faults and programmatic crashes. This becomes SUPER helpful at debugging stages and testing.
PYTHONHASHSEED	This variable will set the random seed used for hash randomization. This is relevant to applications that require a consistent hash value across different runs.
PYTHONDEVMODE	This variable is pretty clear in that it is stating whether or not you want to enable or disable developer mode. If you do not want dev mode, leave it alone; if else, then use “1” or “true” to turn developer mode on which will enable specific development features such as additional information during tracebacks!
PYTHONDUMPREFS	The dump reference variable is a variable that is used typically for debugging and understanding specific objects in the language that may stay alive at the end of a program. Basically, this will control whether or not Python will dump live object references during the shutdown. Sunukar to others, you can use “1” or “true” to enable or disable this.
PYTHONMALLOC	This variable depends on the version and platform on which Python is running. But basically, this variable is used to control which memory allocator Python uses for memory allocation. You can set the variable using values like **`malloc`, `pymalloc`, `dmalloc`** and other various allocators. This is a memory management function pretty much.
PYTHONUNBUFFERED	The python nun buffered is basically a variable that is used to force standard input, output, and error to be unbuffered. This becomes super helpful when you want to see output immediately, especially in cases where Python’s output may be redirected to a file or another process.
PYTHONDONTWRITEBYTECODE	This variable if you could not tell basically states that if true, Python will not write to **PYC files or python compiled (byte code)** files to the disk. If you do not want python to generate

	.pyc files when running scripts, this is a good example of what to use.

A simple table exists after this text showing the flag equivalents of some environment variables shown in the table above.

Variable	**Command Line Flag**
PYTHONDONTWRITEBYTECODE	-B or –dont-write-bytecode
PYTHONUNBUFFERED	-u or –unbuffered
PYTHONVERBOSE	-v or –verbose

Now you may want to also use environment variables on both platforms, so if you are not used to it, below are some notches that use them on both Windows and Linux or Unix systems.

- ***Standard Linux Use:*** When operating with environment variables in Linux systems, all you have to do is label **"export"** followed by the environment variable and its value as shown below!

```
export PYTHONMALLOC=pymalloc
```

- ***Standard Windows use:*** When working with Windows environment variables, it is quite similar to Linux systems but instead of export you use "set" as shown below.

```
set PYTHONFAULTHANDLER=1
```

When you have environment variables set or exported when running Python, the values or systems will change according to the way that you enabled or disabled and even set specific values on those variables without having to use individual flags. Environment variables are quite easy to knock down and if you use them frequently you may find yourself actually understanding how they work across more than just one language but any language that uses them in specific formats. But what if we wanted to set our own?

- **Using Internal Custom Environment Variables:** Say you are writing your own Python project and want to allow users to export or set their own environment variables to say customize the initiation of the script such as setting the verbosity level. Well, Python allows us to do this in our script using the `**os**` standard library! A script below can be run with a custom exported environment variable to set the verbose output of a script if a condition returns true.

```
import os

global Verbosity

x = 10


def CallCond():
    if x == 10:
        if int(Verbosity) >= 1:
            print("VARIABLE x=10\nCONDITION (if x == 10) RETURNED TRUE")
        else:
            print("x is 10!")


if __name__  == '__main__':
    Verbosity = os.environ.get('VERBOSELOGKEY')
    if str(Verbosity) == "":
        print("error when loading verbosity")
    else:
        print("Verbosity is: " + str(Verbosity))
    CallCond()
```

This program demo’s exactly the idea we had! If the verbosity variable is found and it is not set to anything the program outputs an error, if not it continues to output the verbosity level and then calls a function labeled `**CallCond**()` which stands for “**Call Condition**”. When the condition is checked for the variable x and x does equal 10, the program will convert the variable from type string to int using `**int**()` and then check if it is greater than or equal to one. If it is, the verbose output will be printed and if else the message that will output is “x is 10”. This is a quite simple example of what I mean to get at but regardless, it gives you a good idea of where I am headed. Running the program in two states is shown below!

- **State 1 (Variable set to 0)**

```
┌─[totallynotahaxxer@TotallyNotAHaxxer]─[~/Desktop/PyBook/scripts]
└──╼ $export VERBOSELOGKEY=0
┌─[totallynotahaxxer@TotallyNotAHaxxer]─[~/Desktop/PyBook/scripts]
└──╼ $python3 envvars.py

Verbosity is: 0
x is 10!
```

- **State 2 (Variable set to 1)**

```
┌─[totallynotahaxxer@TotallyNotAHaxxer]─[~/Desktop/PyBook/scripts]
└──╼ $export VERBOSELOGKEY=1
┌─[totallynotahaxxer@TotallyNotAHaxxer]─[~/Desktop/PyBook/scripts]
└──╼ $python3 envvars.py

Verbosity is: 1
VARIABLE x=10
CONDITION (if x == 10) RETURNED TRUE
```

Now that we have those out of the way, we can move onto the next section which is actually diving into actual python. I know I mentioned a script above that I wrote to show environment variables, but if you do not understand everything right now in that script, you will understand them in the next section(s).

(Section 3) Python - Data Types

Obviously, based on the title of this section which is labeled as section 2 we are going to focus on data types and operations. This includes the general and following range of operations. Arithmetic, assignment, comparison, logical, bitwise, membership, identity. We will also go deeper into data types in another section called *‘advancing data types*’ which will talk more about methods and complex use cases of data types such as dictionaries, lists, tuples, and more. I felt that this section was just meant to give you a good understanding of the data types, working with operations THEN we can move on to the whole advanced concepts of data types since operations are good to know when you are trying to understand before advanced data type usage.

So without much more to say and mention, let's dive head first! Since all the operations we will talk about will mostly rely on the data type we are using them with such as arithmetic operations, it's a good idea to get a sense of how Python works with data types. Following that, our next sub-section to Section 2 will be the data type section. Below is a table of all the data types that Python uses, following that; there will be more scripts that can demo those data types and how they work.

Data Type	**Description**
str	String data type
int	Integer data type (on the backend, int converts to an integer64 data type in C)
float	Float data type (on the backend, float converts to a float64 data type in C due to its maximum hold being approximately 1.7976931348623157 x 10^308)
complex	Complex data type (e.g., 2 + 3j)
set	Set data type, looks like a regular set, for example: {1,2,3}

frozenset	Immutable set data type
bool	Boolean data type
bytes	Immutable sequence of bytes
Bytearray	Mutable sequence of bytes
memoryview	View of memory of an object
dict	dictionary
None	Similar to null
list	Python's array
tuple	Tuple - example: (1, 2, 3)

These are the data types that are standard to Python without libraries such as NumPy. Now, generally speaking, Python only needs those data types to properly work but libraries such as NumPy add much more extendability to Python with extra data types. But for the current state of this section, we will not get into NumPy. Before we get into anything else and start talking about data-type examples, I would like to define something real quick. I have come across this a ton in my experience, but people often mix up what immutable and mutable are, so I am going to define it below.

- **Immutable**: Immutable data types in this case such as an integer, string, or tuple; are data types that do not allow you to change its state or value can not be changed after it is created. For instance, stating that **"x = 10"** in Python is telling Python that **X** can be modified but has an immutable value which is **10**. This means that while the variable **"x"** that can store a value can change after it is created, the value in this case an integer can not have its state changed.

- **Mutable:** Mutable values are values that can be modified or have their state modified. So this includes data types such as lists or dictionaries in Python. Mutable is the opposite of Immutable.

Below are some good points and programs that demo what these data types may look like in Python!

- **String Data Type (Str):** If you have prior programming experience, then you should easily remember or recall what a string may look like. In Python, a string is assigned to a variable like the example below that assigns to variables strings but does them in a different way than one another.

```
X   = “string1”
X1 = ‘string2’

print(type(X))  # output: <class ‘str’>
print(type(X1)) # output: <class ‘str’>
```

- **Integer Data Type:** The integer data type on the backend of Python translates to a **signed 64-bit integer data type** which means that it can hold approximately **9,223,372,036,854,775,807** as its **maximum or (2**63 - 1) and (-2**63) as its minimum**. Representing integers is pretty simple and you can easily even convert strings into integers using the int keyword which is shown below.

```
X   = 10    # int data type
X2 = “10” # string representing 10

print(type(X)) # output: <class ‘int’>

print(type(int(X2))) # output: <class ‘int’>
```

As you can see, we can easily convert the data type from string to int as long as it represents a number and also how easy it is to just simply assign an integer. This is just basic usage of ints and if we wanted to again, we could extend it for bigger use cases but we will get a bit more into it later. You can also represent integers with negative parts by using the `-` symbol before the number entry.

```
X = -10

print(X + 10) # output: 0 because -10 + 10 = 0
```

- **Float Data Type:** The float data type is pretty easy to understand but there are multiple avenues to go down so to break it down I have listed them into their own sub-points.

  - ***Basic Float Handling:*** In order to create a floating point number in Python, simply place a singular integer, followed by a period followed by another number after that. The program below prints the type of variable that is supposed to declare a float.

```
PI = 3.14

print(type(PI)) # output: <class ‘float’>
```

  - ***Rounding Floats:*** Founding floats is an operation you may find yourself doing quite often, and because it may be common; Python has implemented a standard function call known as `round` which will allow you to automatically round floats. The program below demonstrates this ability.

```
PI = 3.14

print(round(PI)) # output: 3
```

  - ***Floating Point Errors:*** Floating point errors may be common for you to come across if you are not properly handling the data type. An example program below shows how easy it may be to come across warnings or errors with floats without you realizing it!

```
X = 0.1 + 0.1 + 0.1
Y = 0.3

X == y  # Returns false due to the final conversion of X being
“0.30000000000000004”
```

- **Complex Data Type:** Complex data types are also quite interesting and used often within Python. The complex number data type within Python is used to represent quantities that involve both real and imaginary parts where the imaginary unit is denoted by ***`j`*** and it satisfies the equation ***j^2 = -1***. Below is a set of tags that have a program that works with complex numbers in different forms.

    - **Basic Usage:** There are many ways to work with complex data types even without libraries such as ***cmath***, but again we will hit that a bit more later on within the book. First, we need to learn how to create them.

```
z1 = complex(3, 4)      # Result: (3 + 4j)
z2 = 2 + 5j             # Result: (2+5j)
print(z2)               # ^
```

    - **Intermediate Usage:** This next approach is a bit more of an intermediate approach to using complex numbers. But in this case, we are going to run something known as a complex conjugate. For those who do not know, the complex conjugate of a complex number like `z = a + bj` is another complex number `z*` which is obtained by changing the sign of its imaginary part (changing `j` to `-j`. That is a decent way to explain it, but in mathematical terms, if `z = a + bj` then the complex conjugate `z*` is `a - bj`. We will also be grabbing the magnitude of the number using the built-in `abs()` function which stands for absolute.

```
z = 3 + 4j

conjugate_result = z.conjugate()
#Value: 3 - 4j
absolute_value = abs(z)
#Value: 5.0 (magnitude of the complex number)
```

- **Boolean Data Type:** The boolean data types are easy to get used to since Python uses both True and False to represent boolean values or 0 to 1. Python also has a built-in function for the boolean type called `**bool()**` which we can use to convert other data types to booleans. Below is an example program that represents the way booleans can be used for variable assignment and manipulation!

```
x1 = True
x2 = False
x3 = 0
x4 = 14342384342334

print(x1)        # true
print(x2)        # false
print(bool(x3)) # false
print(bool(x4)) # true
```

Note that this is just a basic example of these data types in Python and we will get deeper into their uses and possibilities later on as the point of this section is to just show how the data types work and what data types exist within Python.

- **List Type:** A ton of people get this confused and it’s the difference between an array and a list in Python. For starters, computer science will say that general arrays can only store one specific data type which means that arrays are a type of data structure that can store usually a fixed-size, contiguous, block of elements where each element is of the same data type. This is why in programming languages like Go, you may notice that arrays have to have a pre-defined type before they are declared or used, or in other languages, the data type of the array is determined based on the data type of the element or value in the 0th position of the array (the starting point of an array or start position of the first element). Lists in Python are a bit different, Lists instead represent a dynamic and ordered collection of elements, and unlike arrays; lists can store elements of all different data types making them more versatile and flexible for general use cases. In order to define an array data type in Python, there is a separate module for that where you must declare the type code followed by a list in the argument to the module call that creates an array type. Below is an example program that uses the list type.

```
list_o_froot = [
    "apple",
    "banana",
    "orange",
    True,
    False,
    1,
    1.4
]

# ---- indexing the list ----

# apple
print(list_o_froot[0])

# ---- slicing the list  ----

# banana
print(list_o_froot[1:2]) # get elements from 1 -> 2

# apple, banana, orange
print(list_o_froot[:3])  # index the list all the way until 3

# basic iteration

for value in list_o_froot:
    print(value) # apple \n banana \n orange [...cut...]
```

- **Set and FrozenSet Type:** The set data type is exactly what the name implies, it is a 'set' of data. This follows the mathematical idea of sets where a set is a collection of numbers known as elements. As the name and definition imply, this means that the values in a set data type within Python MUST be integers and if they are strings or other data types, they are converted to the proper data types if possible. Along with the set data type allows a secondary data type known as a frozen_set which is a set that can not be modified directly. The program shown below demos how sets work in Python both frozen and unfrozen.

```
# basic set usage and modification

z = {
    1, 2, 3
}
print(z) # {1, 2, 3}
z.add(4) # Add the number 4 to the set : z = {1, 2, 3, 4}
print(z) # {1, 2, 3, 4}

# frozen set

frozen_set_data = frozenset({10, 20, 30})
print(type(frozen_set_data)) # <class 'frozenset'>
```

- **Dictionary Type:** A dictionary in Python is essentially a map without specific data types to be static as input or key and output or value data. Dictionaries are extremely helpful especially when you do not really want to work around the idea of constant if-else statements and chaining data together. The script below is an example of how you can use dictionaries in Python.

```
def call():
    print("Hello, World!")


diction = {
    "hello": 1,
    1.4    : True,
    0      : False,
    10*20  : "good call!",
    "keypress@f": call,

}

# example usage
```

```
x = 10 * 20
print(diction[x])

# string representation
print(diction["hello"])

# method call from map
diction["Keypress@f"]()
```

This is a good example of how dictionaries can be used in multiple states and as you may have figured; because Python allows dictionaries to be any data type, you can easily embed methods and then call them directly from the map. This becomes EXTREMELY helpful when you are in deeper parts of development and need a performant way to call helper functions or other various systematic functions. There is much more you can do with dictionaries, but this is a basic example of what you could do with it!

- **Tuples:** Tuples are quite an interesting data type, they act like lists but the primary difference is that they can not be modified. They are pretty much constant a constant list. Side note: I would like to get into this later and I will, but Python has an extreme weirdness with constants, there are only very specific data types that are labeled to not be modified under specific states or not modified at all- but it seemed kind of pointless to me when they could have just added "constant" as a keyword or data type. Again I will explain this more when I get the chance but it's just something I wanted to bring up in the case that I was not the only one having an aneurysm trying to figure out why they did not do so. Anyway, this program below shows a good example of hows tuples work.

```
basic_tuple = (1, 2, 3, 4, "hello")

# indexing the tuple
print(basic_tuple[4]) # output: hello
```

```
# nested tuples

nested_tuple = (
    (1, 2, 3, 4),
    ("A", "B", "C", "D"),
    ("a", "b", "c", "d")
)

print(nested_tuple[2][0]) # Output: a
```

- **Binary Data Types:** Python has three primary data types for binary data and that is bytes, byte array, and memory_view. All of these are extremely helpful when it comes to data manipulation and in the case of the topic of this book security-related tasks such as reverse engineering or binary exploitation during payload creation. The bytes data type is a data type that represents an immutable sequence of bytes that is commonly used to store binary data or ASCII-encoded text. Python has a keyword called `` `bytes` `` to create a **bytes** type object. The **bytearray** type also follows a similar to the bytes type but instead is mutable which allows you to modify the bytes after creation. Also similar to the bytes type, the **bytearray** type has a keyword called `` `bytearray` `` to create a **bytearray** object. The final data type in the binary data type set with Python is the memoryview type which is a data type that provides developers or users of Python with a view of the memory of an object. This means that it allows you to access the data of an object directly in its raw memory representation. You might be wondering when this will be useful and I will give you a hint - its super useful when manipulating large amounts of binary data without copying the data ;). Below is a program that demonstrates the most basic use case of these data types.

```
# --- bytes data type, basic creation --- #
```

```
bin_data = bytes([65, 66, 67]) # b'ABC'

# python also allows you to use b'' to represent the bytes without
# using bytes()
bin_datarep = b'ABC'

# --- Byte array and basic creation --- #

mutable_bin_data = bytearray([
    65, 66, 67
])

# --- memory view example --- #

data = bytearray(b'Hello, World!')
view = memoryview(data)

# access the elements
print(view[0]) # 72 (ASCII value of 'H')
```

Now that we are done examining the basic data types of Python, we can move onto the operations those kinds of data types can work with! I know there were a ton of stopping points and confusing notes in there but I can assure you that we will focus on those when the time comes which is mainly towards the end of this segment of language development. If you are wanting to hop to see what I wanted to talk about, you can go ahead and check “Pythonic Weirdness” which is a section in this manual purely goes over weird things and behavior with Python that do not make sense to me personally. A note on that section as well, that section is purely based on my experience firsthand developing and designing multiple programming languages as well as helping design private languages in all different forms (byte code compiled, machine code compiled, and interpreted) all varying of different code designs. So it is important to note that the section that speaks about my confusion with Python is PURELY BIAS and if you want to explore some of the routes I took please do and I would LOVE to hear feedback on it! Without anything else to say about that and having to get it out of the way, we can move on to the next section which is the operations within Python.