

Getting Started with Python

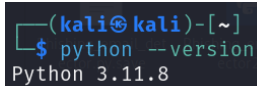
The Python scripts used here can be downloaded from this [GitHub Repository](#)

Verify if Python is Installed

The Python environment comes pre-installed in Linux operating system. As we are using Kali Linux, we don't have to worry about installation.

We can verify if python is installed using the following command: -

```
$ python --version
```



```
(kali㉿kali)-[~]  
$ python --version  
Python 3.11.8
```

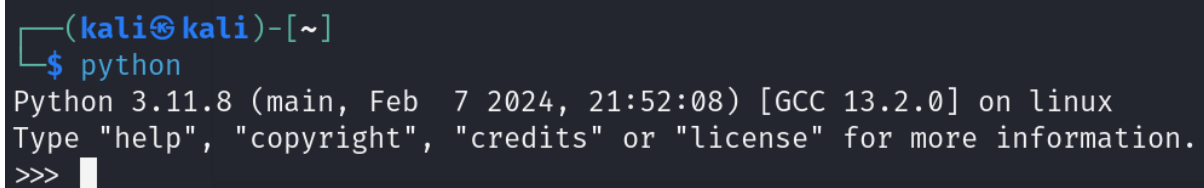
Here we see that the version 3.11.8 is installed.

Launch an Interactive Python Shell

By executing (running) the **python** command in your terminal, you are presented with an interactive Python shell.

This is also known as the Python Interpreter or a REPL (for 'Read Evaluate Print Loop').

```
$ python
```



```
(kali㉿kali)-[~]  
$ python  
Python 3.11.8 (main, Feb 7 2024, 21:52:08) [GCC 13.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```

">>>" This sign indicates that we are inside the Python interpreter.

To exit the Python interpreter press "CTRL and d" together.

If you want to cancel a command if you're in the middle of typing and get back to a clean command prompt, while staying inside the Interpreter shell, use "CTRL+c".

Writing Code Using a Text Based Editor (Nano Editor)

Simply type in nano followed by the file name you want to create and use the .py file extension.

```
$ nano file_name.py
```

Write or copy the required code inside this.

Finally save it using "CTRL+x" followed by entering "y". Then finally press enter.

We also need to convert this saved file to an executable file. This can be done by using the following command: -

```
$ chmod +x file_name.py
```

And finally execute the program using: -

```
$ python3 file_name.py
```

Printing any Value

For this use the `print()` command. Enter the value you want to display inside inverted commas, between the parentheses. We can also enter the variable name, this will print the value stored inside the variable.

```
(kali㉿kali)-[~]  
└─$ python  
Python 3.11.8 (main, Feb 7 2024, 21:52:08) [GCC 13.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello")  
Hello  
>>> █
```

Creating Variables and Assigning Values

To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it.

```
>>> <variable name> = <value>
```

Python uses = to assign values to variables. There's no need to declare a variable in advance (or to assign a datatype to it), assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.

```
>>> a=10  
>>> print (a)  
10  
>>> █
```

Variable assignment works from left to right. So, the following will give you a syntax error.

```
>>> 10=a
      File "<stdin>", line 1
        10=a
        ^^
SyntaxError: cannot assign to literal here. Maybe you meant '=' instead of '=?'
>>> █
```

You cannot use python's keywords as a valid variable name. You can see the list of keywords by:

```
import keyword
print(keyword.kwlist)
```

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
, 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try',
, 'while', 'with', 'yield']
>>> █
```

You can also assign a single value to several variables simultaneously.

```
a = b = c = 1
print(a, b, c)
```

```
>>> a=b=c=1
>>> print(a,b,c)
1 1 1
>>> █
```

When using such cascading assignment, it is important to note that all three variables a, b and c refer to the same object in memory, an int object with the value of 1. In other words, a, b and c are three different names given to the same int object. Assigning a different object to one of them afterwards doesn't change the others, just as expected:

```
a = b = c = 1 # all three names a, b and c refer to same int
object with value 1
```

```
print(a, b, c)
```

```
# Output: 1 1 1
```

```
b = 2 # b now refers to another int object, one with a value
of 2
```

```
print(a, b, c)
```

```
# Output: 1 2 1 # so output is as expected.
```

```
>>> a=b=c=1
>>> print(a,b,c)
1 1 1
>>> b=2
>>> print(a,b,c)
1 2 1
>>> 
```

Things are a bit different when it comes to *modifying* the object (in contrast to *assigning* the name to a different object, which we did above) when the cascading assignment is used for mutable types. Take a look below, and you will see it first hand:

```
x = y = [7, 8, 9] # x and y are two different names for the
same list object just created, [7,
```

```
8, 9]
```

```
x[0] = 13 # we are updating the value of the list [7, 8, 9]
through one of its names, x
```

```
in this case
```

```
print(y) # printing the value of the list using its other name
```

```
# Output: [13, 8, 9] # hence, naturally the change is
reflected
```

```
>>> x=y=[7,8,9]
>>> x[0]=5
>>> print(y)
[5, 8, 9]
>>> 
```

Nested lists are also valid in python. This means that a list can contain another list as an element.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list
```

```
print x[2]
```

```
# Output: [3, 4, 5]
```

```
print x[2][1]
```

```
# Output: 4
```

Lastly, variables in Python do not have to stay the same type as which they were first defined -- you can simply use = to assign a new value to a variable, even if that value is of a different type.

```
a = 2
```

```
print(a)
```

```
# Output: 2
```

```
a = "New value"

print(a)

# Output: New value
```

Taking in User Input

Interactive input

To get input from the user, use the `input()` function

```
>>> name=input("What is your name?")
What is your name?John Smith
>>> print(name)
John Smith
>>> █
```

Python Datatypes

Integers (int):

Integer

```
x = 5
```

```
print(type(x)) # Output: <class 'int'>
```

```
>>> x=5
>>> print (type(x))
<class 'int'>
>>> █
```

Large integers

```
big_num = 1234567890123456789
```

```
print(big_num) # Python handles large integers automatically
```

```
>>> big_num = 1234567890123456789
>>> print (type(big_num))
<class 'int'>
>>> █
```

Floating-point numbers (float):

Float

```
y = 3.14
```

print(type(y)) # Output: <class 'float'>

```
>>> y = 3.14
>>> print(type(y))
<class 'float'>
```

Scientific notation

z = 1.5e2 # 1.5 * 10²

print(z) # Output: 150.0

```
>>> z = 1.5e2
>>> print(z)
150.0
```

Complex numbers:

Complex number

c = 1 + 2j

print(type(c)) # Output: <class 'complex'>

print(c.real) # Output: 1.0

print(c.imag) # Output: 2.0

```
>>> c = 1 + 2j
>>> print(type(c))
<class 'complex'>
>>> print(c.real)
1.0
>>> print(c.imag)
2.0
```

Strings: Strings are immutable sequences of Unicode characters.

String operations





name = "Alice"

print(len(name)) # Output: 5

print(name[0]) # Output: A

print(name[-1]) # Output: e

```
>>> name = "Alice"
>>> print(len(name))
5
>>> print(name[0])
A
>>> print(name[-1])
e
>>>
```





Country	Name	Description
 United States	IANA-IPV4-MAPPED-ADDRESS	Internet Assigned Numbers Authority
 United States	NETS	American Registry of Internet Numbers
 India	AMAZON-BOM	Amazon Data Service
 India	AMAZON-BOM	Amazon Data Service

String methods

```
print(name.upper()) # Output: ALICE
```

```
print(name.replace('A', 'B')) # Output: Blice
```

```
>>> print(name.upper())
ALICE
>>> print(name.replace('A', 'B'))
Blice
>>>
```




 United States	IANA-IPV4-MAPPED-ADDRESS	Internet Assigned Numbers Authority
 United States	NETS	American Registry of Internet Numbers
 India	AMAZON-BOM	Amazon Data Service
 India	AMAZON-BOM	Amazon Data Service

String formatting

```
age = 30
```

```
print(f"{name} is {age} years old") # Output: Alice is 30 years old
```

```
>>> age = 30
>>> print(f"{name} is {age} years old")
Alice is 30 years old
>>>
```

 United States	NETS	American Registry of Internet Numbers
 India	AMAZON-BOM	Amazon Data Service
 India	AMAZON-BOM	Amazon Data Service

Booleans: Booleans represent truth values.

Boolean values

```
is_true = True
```

```
is_false = False
```





Boolean operations

```
print(True and False) # Output: False
```

```
print(True or False) # Output: True
```

```
print(not True) # Output: False
```

```
>>> is_true = True
>>> is_false = False
>>> print(True and False)
False
>>> print(True or False)
True
>>> print(not True)
False
>>> 
```





Country	Name	Description
 United States	IANA-IPV4-MAPPED-ADDRESS	Internet Assigned
 United States	NET65	American Registry
 India	AMAZON-BOM	Amazon Data Ser
 India	AMAZON-BOM	Amazon Data Ser

Comparison operators result in booleans

```
print(5 > 3) # Output: True
```

```
print(5 == 6) # Output: False
```

```
>>> print(5 > 3)
True
>>> print(5 == 6)
False
>>> 
```

Country	Name	Description
 United States	IANA-IPV4-MAPPED-ADDRESS	Internet Assigned
 United States	NET65	American Registry
 India	AMAZON-BOM	Amazon Data Ser
 India	AMAZON-BOM	Amazon Data Ser

NoneType: None is a special constant in Python that represents the absence of a value or a null value.





None

```
x = None
```

```
print(x) # Output: None
```

```
print(type(x)) # Output: <class 'NoneType'>
```

```
>>> x = None
>>> print(x)
None
>>> print(type(x))
<class 'NoneType'>
>>> 
```




Country	Name	Description
 United States	IANA-IPV4-MAPPED-ADDRESS	Internet Assigned
 United States	NET65	American Registry
 India	AMAZON-BOM	Amazon Data Ser
 India	AMAZON-BOM	Amazon Data Ser

Checking for None

```
if x is None:
```

```
    print("x is None")
```

```
>>> if x is None:
...     print("x is None")
...
x is None
>>> 
```

Country	Name	Description
 United States	NET65	American Registry
 India	AMAZON-BOM	Amazon Data Ser
 India	AMAZON-BOM	Amazon Data Ser

Type Conversion

Type Conversion: You can convert between different types using functions like `int()`, `float()`, `str()`, `bool()`, etc.

Type conversion

```
print(int("123")) # Output: 123
```

```
print(float("3.14")) # Output: 3.14
```

```
print(str(42)) # Output: "42"
```

```
print(bool(1)) # Output: True
```

```
>>> print(int("123"))
123
>>> print(float("3.14"))
3.14
>>> print(str(42))
42
>>> print(str(42))
42
>>>
```

Country	Name	Description
United States	IANA-IPV4-MAPPED-ADDRESS	Internet Assigned Num
United States	NET62	American Registry for
India	AMAZON-BOM	Amazon Data Service
India	AMAZON-BOM	Amazon Data Service

Immutability: Strings, tuples, and numbers are immutable in Python. Once created, their value cannot be changed. Any operation that seems to modify them actually creates a new object.

Dynamic Typing: Python is dynamically typed, meaning you don't need to declare the type of a variable when you create it. The interpreter infers the type based on the value assigned.

Other Datatypes: -

Lists: Lists are ordered, mutable sequences, useful for storing collections of related items that may change.

Cybersecurity use case: Storing a list of suspicious IP addresses.

```
$ nano IP_List.py
```

```
(kali㉿kali)-[~]
$ nano IP_List.py
```

In the screenshot below you can find the Python script for Storing a list of suspicious IP addresses. This can also be downloaded from

https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/IP_List.py

```
GNU nano 7.2 IP_List.py *
suspicious_ips = ['192.168.1.100', '10.0.0.25', '172.16.0.1']

# Add a new suspicious IP
suspicious_ips.append('192.168.0.50')

# Remove an IP that's no longer suspicious
suspicious_ips.remove('10.0.0.25')

# Check if an IP is in the suspicious list
ip_to_check = '172.16.0.1'
if ip_to_check in suspicious_ips:
    print(f"{ip_to_check} is suspicious!")
print(suspicious_ips)
```

```
$ chmod +x IP_List.py
```

```
$ python3 IP_List.py
```

```
(kali㉿kali)-[~]
$ chmod +x IP_List.py

(kali㉿kali)-[~]
$ python3 IP_List.py
172.16.0.1 is suspicious!
['192.168.1.100', '172.16.0.1', '192.168.0.50']

(kali㉿kali)-[~]
$
```

This list can be easily updated as new suspicious IPs are discovered or cleared.

Tuples: Tuples are ordered, immutable sequences, which means they can't be changed. It is ideal for data that shouldn't change.

Cybersecurity use case: Storing details of a security event.

```
$ nano security_event_tuple.py
```

```
(kali㉿kali)-[~]
$ nano security_event_tuple.py
```

Write the code given in the screenshot below (Available [here](#)): -

```
GNU nano 7.2 security_event_tuple.py *
# (timestamp, event_type, severity, source_ip)
security_event = ('2023-09-30 14:30:00', 'Unauthorized Access Attempt', 'High', '203.0.113.42')

# Accessing event details
print(f"Event Type: {security_event[1]}")
print(f"Severity: {security_event[2]}")

# Trying to modify the tuple (this will raise an error)
# security_event[2] = 'Medium' # This would raise a TypeError
```

```
$ chmod +x security_event_tuple.py
```

```
$ python3 security_event_tuple.py
```

```
(kali㉿kali)-[~]
$ chmod +x security_event_tuple.py

(kali㉿kali)-[~]
$ python3 security_event_tuple.py
Event Type: Unauthorized Access Attempt
Severity: High

(kali㉿kali)-[~]
$
```

Tuples are useful here because the details of a recorded security event shouldn't change.

Sets: Sets are unordered collections of unique elements, efficient for membership testing and eliminating duplicates.

Cybersecurity use case: Tracking unique usernames involved in failed login attempts.

Enter the command: -

```
$ nano login_sets.py
```

```
(kali㉿kali)-[~]
$ nano login_sets.py
```

Write down the code given below and save it (You can copy the code from https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/login_sets.py): -

```
GNU nano 7.2 login_sets.py *
failed_logins = {'alice', 'bob', 'charlie', 'alice', 'david', 'bob'}

# The set automatically removes duplicates
print(failed_logins) # Output: {'alice', 'bob', 'charlie', 'david'}

# Quick check if a username has had failed attempts
username_to_check = 'eve'
if username_to_check in failed_logins:
    print(f"{username_to_check} has had failed login attempts")
else:
    print(f"No failed attempts for {username_to_check}")

# Add a new failed attempt
failed_logins.add('eve')

# Set operations
allowed_users = {'alice', 'charlie', 'eve', 'frank'}
suspicious_users = failed_logins - allowed_users
print(f"Suspicious users: {suspicious_users}")
```

To convert this file to an executable file, use the command

```
$ chmod +x login_sets.py
```

And finally to run the Python script use the command: -

```
$ python3 login_sets.py
```

```
(kali㉿kali)-[~]
$ chmod +x login_sets.py

(kali㉿kali)-[~]
$ python3 login_sets.py
{'charlie', 'alice', 'david', 'bob'}
No failed attempts for eve
Suspicious users: {'david', 'bob'}

(kali㉿kali)-[~]
$
```

Sets are great for quickly checking membership and performing set operations, which can be useful in analyzing security data.

Dictionaries: Dictionaries are key-value pairs, useful for storing and retrieving data with a unique identifier.

Cybersecurity use case: Storing user access levels.

Create the file using the command: -

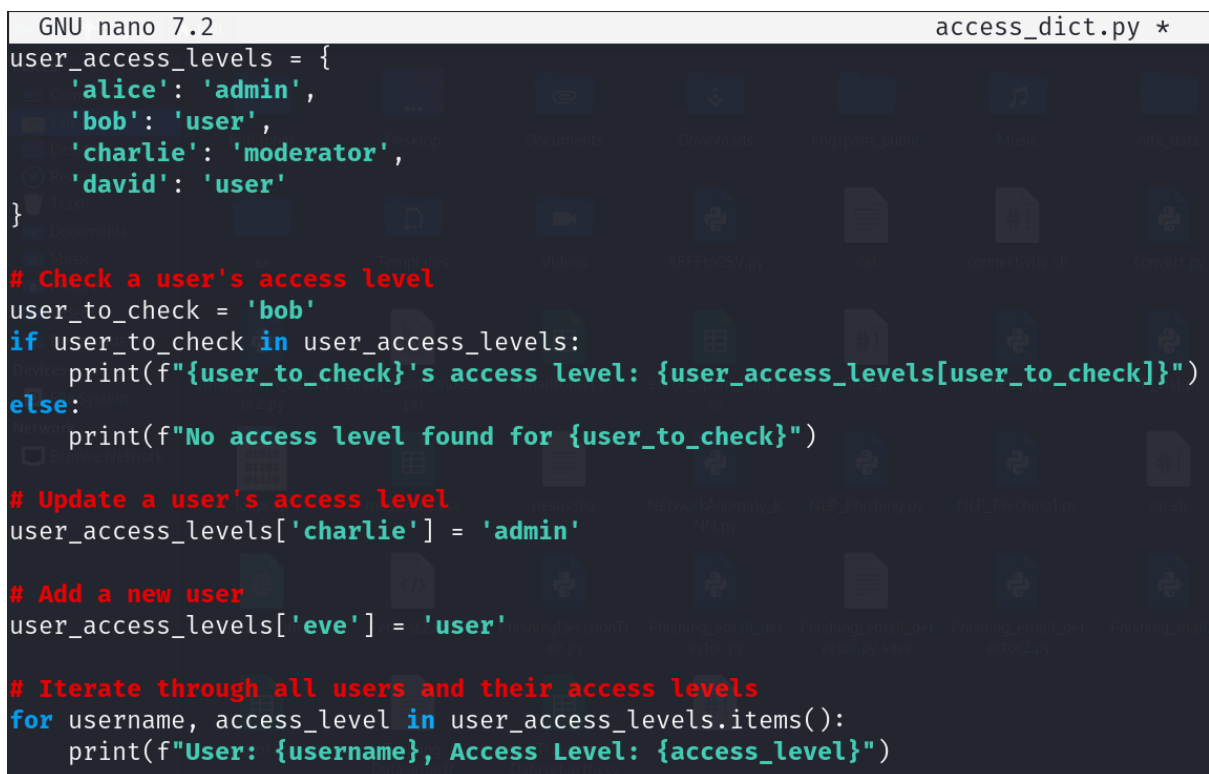
```
$ nano access_dict
```



```
(kali@kali)-[~]  
$ nano access_dict.py
```

In this write the code given in the screenshot below: -

This code is available [here](#)



```
GNU nano 7.2 access_dict.py *  
user_access_levels = {  
    'alice': 'admin',  
    'bob': 'user',  
    'charlie': 'moderator',  
    'david': 'user'  
}  
  
# Check a user's access level  
user_to_check = 'bob'  
if user_to_check in user_access_levels:  
    print(f"{user_to_check}'s access level: {user_access_levels[user_to_check]}")  
else:  
    print(f"No access level found for {user_to_check}")  
  
# Update a user's access level  
user_access_levels['charlie'] = 'admin'  
  
# Add a new user  
user_access_levels['eve'] = 'user'  
  
# Iterate through all users and their access levels  
for username, access_level in user_access_levels.items():  
    print(f"User: {username}, Access Level: {access_level}")
```

Now for converting the file to executable form use the command: -

```
$ chmod +x access_dict
```

And finally run the script using the command: -

```
$ python3 access_dict
```

```
(kali㉿kali)-[~]
$ chmod +x access_dict.py

(kali㉿kali)-[~]
$ python3 access_dict.py
bob's access level: user
User: alice, Access Level: admin
User: bob, Access Level: user
User: charlie, Access Level: admin
User: david, Access Level: user
User: eve, Access Level: user

(kali㉿kali)-[~]
$
```

Some Important Points On Python Programming

- **Case Sensitivity:** Python is case-sensitive. This means that variable names, function names, and keywords are sensitive to uppercase and lowercase letters.
- **Indentation:** Python uses indentation to define code blocks, unlike many other languages that use braces {}. Consistent indentation is crucial.
- **Comments:** Use # for single-line comments and triple quotes for multi-line comments.
- **Variables:** Variables are created when you first assign a value to them. There's no need to declare their type.
- **Data Types:** Python has several built-in data types including int, float, str, bool, list, tuple, dict, and set.
- **Arithmetic Operators:** Python supports standard arithmetic operators: +, -, *, /, // (integer division), % (modulo), and ** (exponentiation).
- **Comparison Operators:** Python uses ==, !=, <, >, <=, >= for comparisons.
- **Logical Operators:** and, or, and not are used for logical operations.
- **String Operations:** Strings can be concatenated with + and repeated with *.
- **Importing Modules:** Use the import keyword to use functionality from other modules.

Now let us use above concepts to see how Python can be utilised for Machine Learning Projects: -

Importing a common machine learning library:

```
import sklearn
print(sklearn.__version__)
```

```

(kali@kali)-[~] 74
$ python
Python 3.11.8 (main, Feb  7 2024, 21:52:08) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sklearn
>>> print(sklearn.__version__)
1.5.1
>>>

```

Creating a simple dataset for binary classification (e.g., malicious vs. benign):

```

x = [[0, 0], [1, 1], [1, 0], [0, 1]]
y = [0, 1, 1, 0]

```

```

>>> x = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 1, 1, 0]

```

Splitting data into training and testing sets:

```

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3, random_state=42)

>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
>>>

```

Training a simple classifier (Decision Tree):

```

from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()

clf.fit(X_train, y_train)

>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier()
>>> clf.fit(X_train, y_train)
DecisionTreeClassifier()
>>>

```

Making predictions:

```

predictions = clf.predict(X_test)

print(predictions)

>>> predictions = clf.predict(X_test)
>>> print(predictions)
[1 0]
>>>

```

Evaluating model accuracy:

```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, predictions)

print(f"Accuracy: {accuracy}")
```

```
>>> from sklearn.metrics import accuracy_score
>>> accuracy = accuracy_score(y_test, predictions)
>>> print(f"Accuracy: {accuracy}")
Accuracy: 1.0
>>>
```

Creating a confusion matrix:

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, predictions)

print(cm)
```

```
>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_test, predictions)
>>> print(cm)
[[1 0]
 [0 1]]
>>>
```

A confusion matrix is a table that sums up the performance of a classification model. It works for binary and multi-class classification. The confusion matrix shows the number of correct predictions: true positives (TP) and true negatives (TN) and incorrect predictions: false positives (FP) and false negatives (FN).

Implementing a simple neural network using NumPy:

```
import numpy as np
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
input_layer = np.array([0.1, 0.2, 0.3])
weights = np.array([0.4, 0.5, 0.6])
output = sigmoid(np.dot(input_layer, weights))
```



```
print(output)
```

```
>>> import numpy as np
>>>
>>> def sigmoid(x):
...     return 1 / (1 + np.exp(-x))
...
>>> input_layer = np.array([0.1, 0.2, 0.3])
>>> weights = np.array([0.4, 0.5, 0.6])
>>> output = sigmoid(np.dot(input_layer, weights))
>>> print(output)
0.5793242521487495
>>>
```

IP range	Country	Name	Description
IANA-IPV4-MAPPED-ADDRESS		IANA-IPV4-MAPPED-ADDRESS	Internet Assigned Numbers Authority
NETBS		NETBS	American Registry for Internet Numbers
AS2	India	AMAZON-BOM	Amazon Data Services
AS2	India	AMAZON-BOM	Amazon Data Services

Using natural language processing for text classification:

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
texts = ["This is a normal email", "You've won a prize!", "Urgent: Account suspended"]
```

```
vectorizer = CountVectorizer()
```

```
X = vectorizer.fit_transform(texts)
```

```
print(X.toarray())
```

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> texts = ["This is a normal email", "You've won a prize!", "Urgent: Account suspended"]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(texts)
>>> print(X.toarray())
[[0 1 1 1 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0 1 1 1]
 [1 0 0 0 0 1 0 1 0 0 0]]
>>>
```

Implementing a simple anomaly detection:

```
import numpy as np
```

```
def is_anomaly(data_point, mean, std_dev, threshold=2):
```

```
    z_score = (data_point - mean) / std_dev
```

```
    return abs(z_score) > threshold
```

```
data = [10, 12, 13, 15, 19, 20, 51]
```

```
mean = np.mean(data)
```

```
std_dev = np.std(data)
```

```
for point in data:
```

```
    print(f"{point} is anomaly: {is_anomaly(point, mean, std_dev)}")
```

```
>>> import numpy as np
>>>
>>> def is_anomaly(data_point, mean, std_dev, threshold=2):
...     z_score = (data_point - mean) / std_dev
...     return abs(z_score) > threshold
...
>>> data = [10, 12, 13, 15, 19, 20, 51]
>>> mean = np.mean(data)
>>> std_dev = np.std(data)
>>>
>>> for point in data:
...     print(f"{point} is anomaly: {is_anomaly(point, mean, std_dev)}")
...
10 is anomaly: False
12 is anomaly: False
13 is anomaly: False
15 is anomaly: False
19 is anomaly: False
20 is anomaly: False
51 is anomaly: True
>>>
```

Country	Name	Description
United States	IANA-IPV4-MAPPED-ADDRESS	Internet Assigned Number
United States	NET65	American Registry for Inte
India	AMAZON-BOM	Amazon Data Services Inc
India	AMAZON-BOM	Amazon Data Services Inc

We now understand the basic syntax of python and have seen some of its use cases in Machine Learning.

We have also seen that Python has many libraries that are extremely useful in Machine Learning. These libraries have predefined functions which eases the task of implementing ML systems.

We will see more about these libraries in the next section.