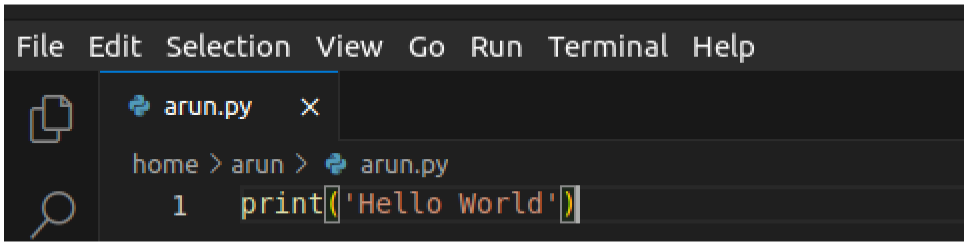
PYTHON

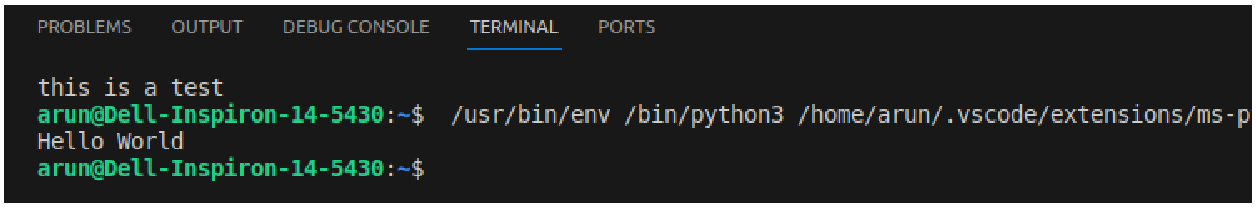
1. VS-code can be used as an interpreter for python and the file can be saved with an extension ".py"

2. First Program
print("First Program") --> A function to print the text written in quotes on the console.

3. To run a python file in terminal
Command: python3 file1.py*(file-name)*

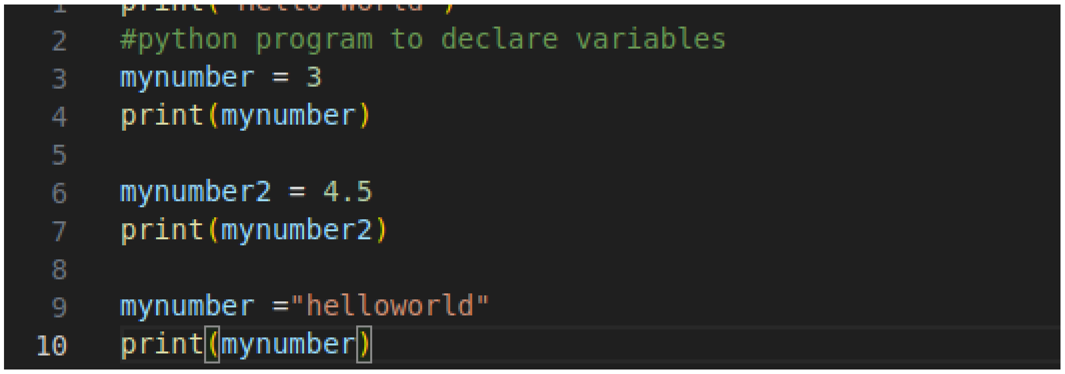


```python
print('Hello World')
```

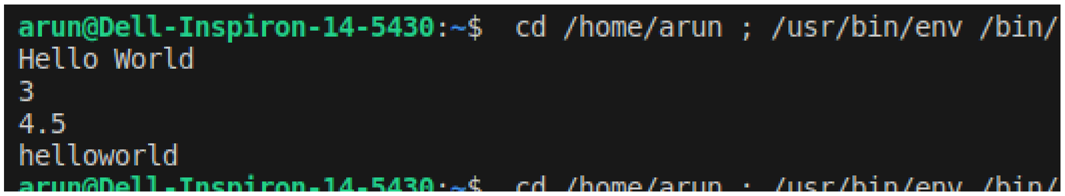


```
this is a test
arun@Dell-Inspiron-14-5430:~$  /usr/bin/env /bin/python3 /home/arun/.vscode/extensions/ms-p
Hello World
arun@Dell-Inspiron-14-5430:~$
```

4. Declaration --> No need to declare in Python.
myNumber = 3 --> It will take it as number
myNumber = 4.5 --> It will take it as float
nyNumber = "helloworld" --> It will take it as a string



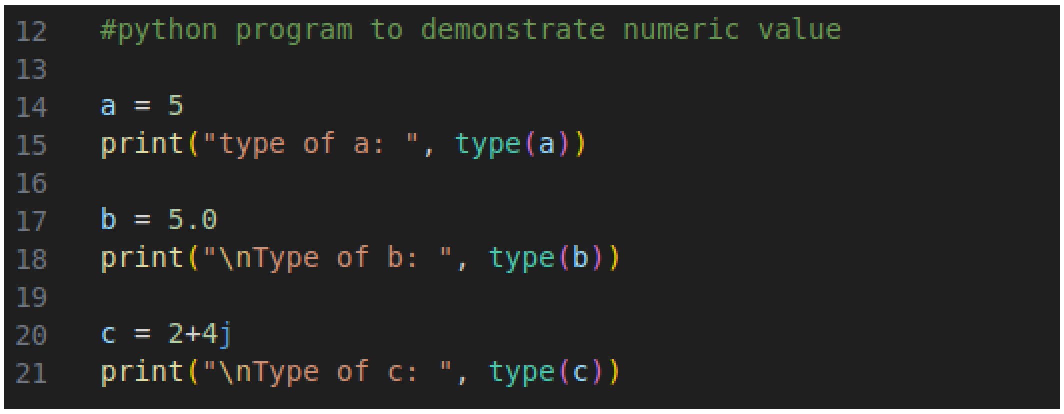
```python
#python program to declare variables
mynumber = 3
print(mynumber)

mynumber2 = 4.5
print(mynumber2)

mynumber ="helloworld"
print(mynumber)
```



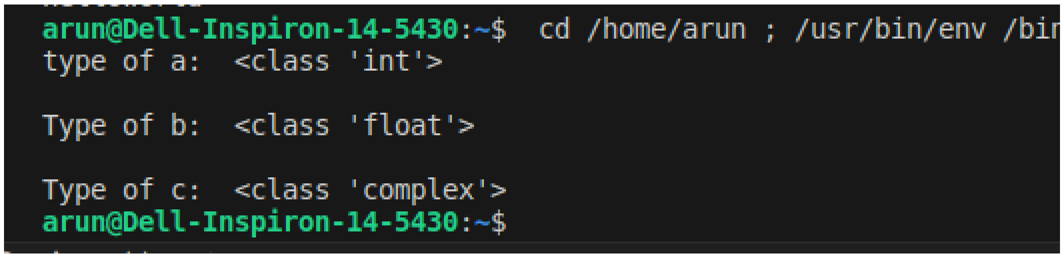
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/
Hello World
3
4.5
helloworld
```

5. Everything is a object in Python Programming and Data types is classes with variables as Instances.

6. Data Types of Python:
1. Numeric
a. Integer --> It contains positive or negative whole numbers (without fractions or decimals)
b. Float --> It is specified by a decimal point.
c. Complex Number --> It is specified as(real part) + (imaginary part)j.



```python
#python program to demonstrate numeric value

a = 5
print("type of a: ", type(a))

b = 5.0
print("\nType of b: ", type(b))

c = 2+4j
print("\nType of c: ", type(c))
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bir
type of a:  <class 'int'>

Type of b:  <class 'float'>

Type of c:  <class 'complex'>
arun@Dell-Inspiron-14-5430:~$
```

2. Boolean --> Data type with one of the two in-built values i.e., True or False.

3. Dictionary --> An unordered collection of data values in key : value pair.

```
Creating a Dictionary with Integer Keys
Dict= {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)
OUTPUT: Dictionary with the use of Integer Keys:
                  {1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

4. Set --> An unordered collection of data types that is iterable, mutable and has no duplicate elements. The order is undefined.

```
Creating a Set with the use of a String
set1 =set("GeeksForGeeks")
print("\nSet with the use of String: ")
print(set1)
OUTPUT: Set with the use of String:
                  {'F', 'o', 'G', 's', 'r', 'k', 'e'}
```

5. Sequence Type

a. Strings --> It is a collection of characters put in a single/double quote.

```
Creating a String
String1 ='Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)
```

b. List --> It is an ordered collection of data (just like arrays).

```
Creating a List
List= []
print("Initial blank List: ")
print(List)
```

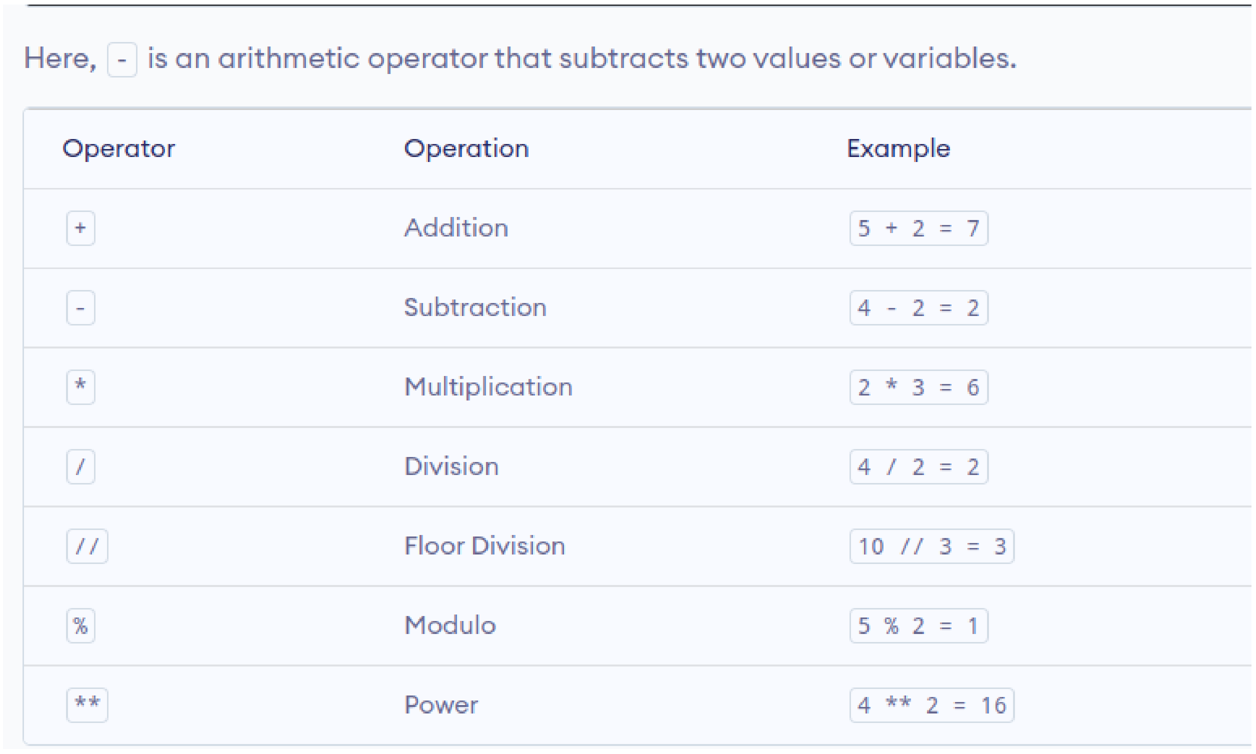
c. Tuple --> It is same as list but only difference is that it is immutable(cannot be changed after creation).

```
Creating a Tuple with the use of list
list1 =[1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))
```

7. To check the data type of any value --> type()

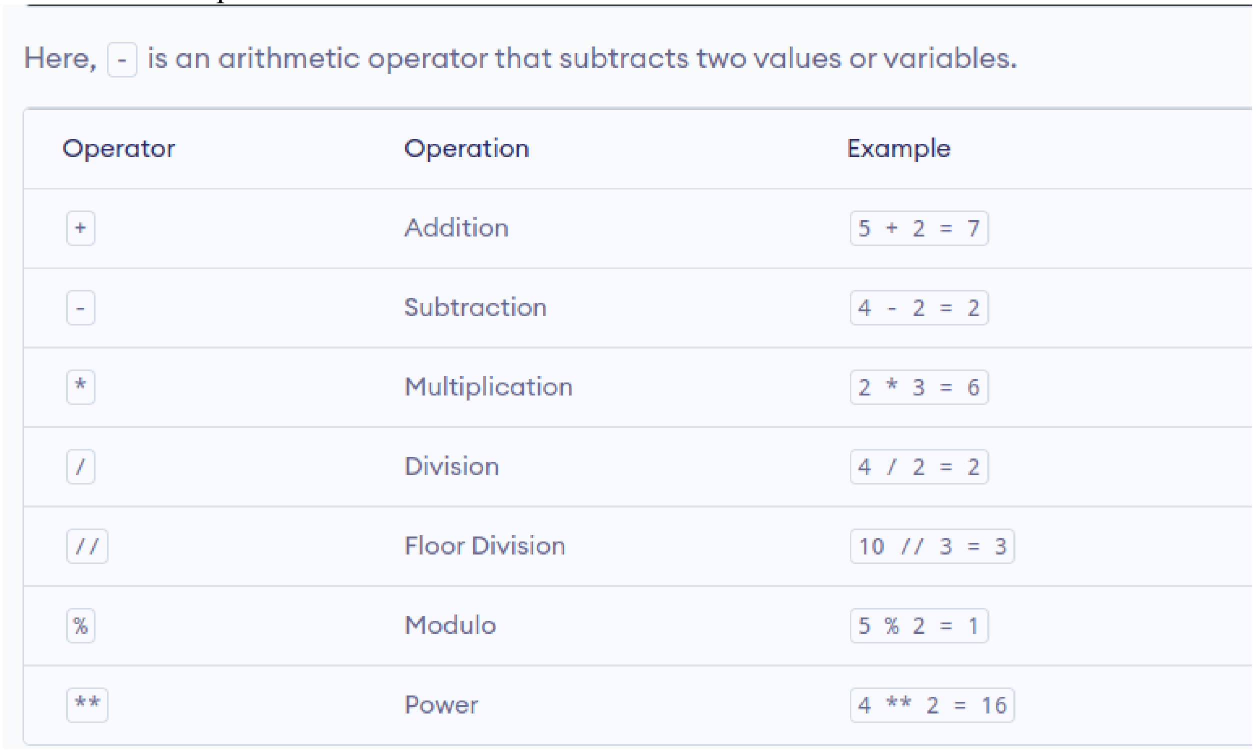
8. Math Functions: --> Python has an in-built module that can be used for mathematical tasks.
a. math.floor() --> Rounds a number down to the nearest integer.
b. math.isclose() --> checks whether two values are close to each other or not.
c. math.isqrt() --> Rounds a square root number downwards to the nearest integer.
Many more ....

9. Operator Precedence --> It describes the order in which operations are performed.


Here, `-` is an arithmetic operator that subtracts two values or variables.

Operator	Operation	Example
`+`	Addition	`5 + 2 = 7`
`-`	Subtraction	`4 - 2 = 2`
`*`	Multiplication	`2 * 3 = 6`
`/`	Division	`4 / 2 = 2`
`//`	Floor Division	`10 // 3 = 3`
`%`	Modulo	`5 % 2 = 1`
`**`	Power	`4 ** 2 = 16`

10. Arithmetic Operators



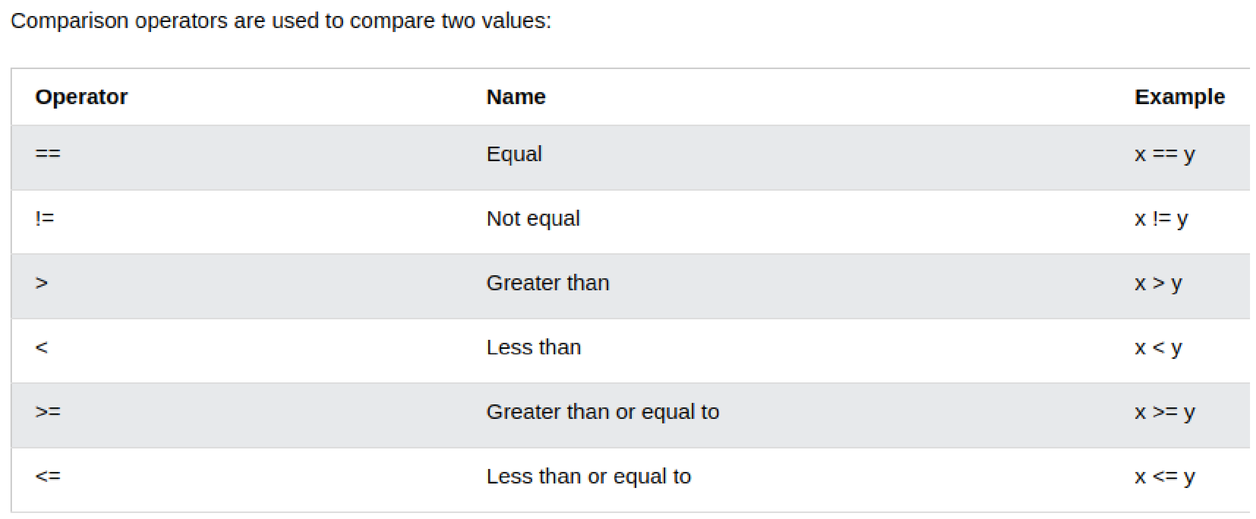
Here, `-` is an arithmetic operator that subtracts two values or variables.

Operator	Operation	Example
`+`	Addition	`5 + 2 = 7`
`-`	Subtraction	`4 - 2 = 2`
`*`	Multiplication	`2 * 3 = 6`
`/`	Division	`4 / 2 = 2`
`//`	Floor Division	`10 // 3 = 3`
`%`	Modulo	`5 % 2 = 1`
`**`	Power	`4 ** 2 = 16`

11. Assignment Operators

Operator	**Example**	**Same As**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

12. Comparison Operators



Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

13. Scope --> The region where a variable is accessed it the scope of that.

14. Local Scope --> A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

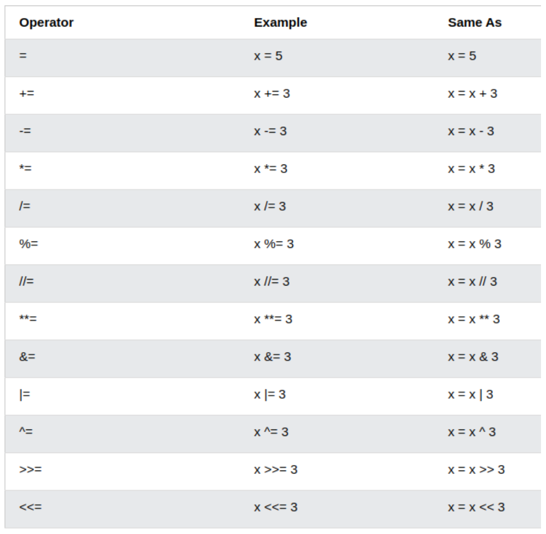
15. Global Scope --> A variable created in the main body of the Python code is a global variable and belongs to the global scope.

16. Global Keyword --> If you need to create a global variable, but are stuck in the local scope, you can use the globalkeyword.

17. Python Statement -->These are the whole structures which are declared.
Python Expression --> These can be assigned as a value or can used as operands.

PART-2

1. **Augmented Assignment Operators** -->
Here it combines the arithmetic operation with the assignment to a variable.



Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

2. **Strings** --> They are the operands which are
assigned to variable and are surrounded by single/double quotation marks.

```
print("Hello")
print('Hello')
```

Assigning string to a variable -->

```
a = "Hello"
print(a)
```

Length of a string --> **len()**

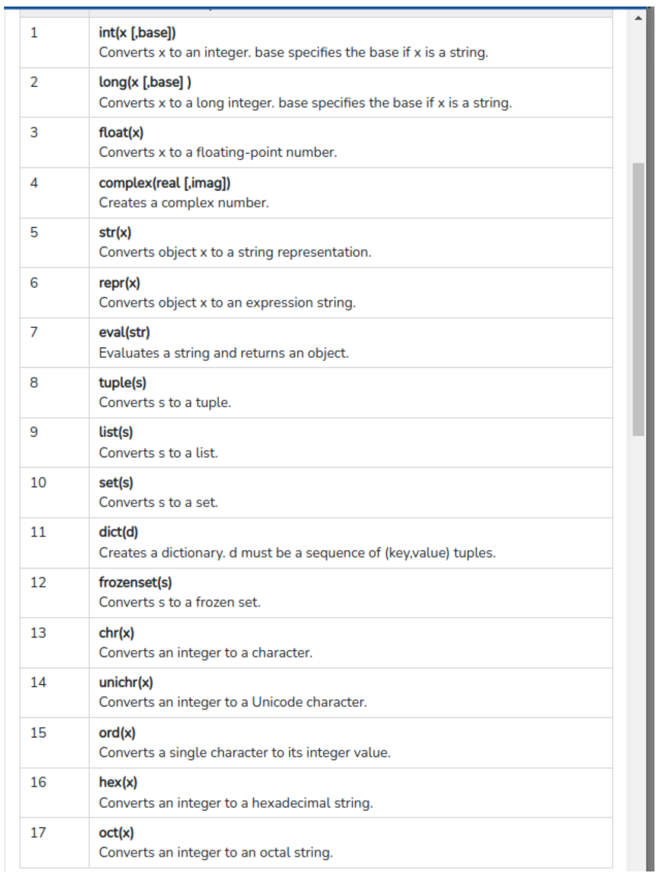
Check if certain phrase or character in string -->

```
txt = "The best things in life are free!"
print("free" in txt)
```

3. **String Concatenation** --> String concatenation means add strings together.

```
x ="Python is "
y ="awesome"
z = x + y
print(z)
```

4. **Type Conversion** --> There are several built-in functions to perform conversion from one data type to another.



1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

5. **Escaping Characters in Python** --> To insert characters that are illegal in a string, use an escape character.

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

6. **String formatting in Python**
There are five different ways to perform string formatting in Python

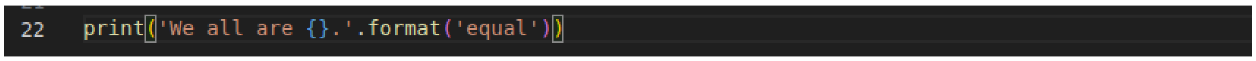
1. **Formatting with % Operator.**

a. print("The mangy, scrawny stray dog %s gobbled down" %'hurriedly' +
"the grain-free, organic dog food.")

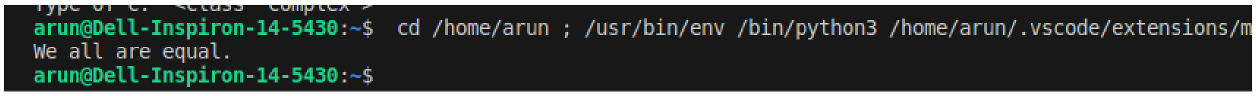
b. x = 'looked'
print("Misha %s and %s around"%('walked',x))
c. print('The value of pi is: %5.4f' %(3.141592)) - **Here 5.4 means 5 width and 4 decimal Places**

2. **Formatting with format() string method.**

a. `print('We all are {}.'.format('equal')) –> We all are equal.`

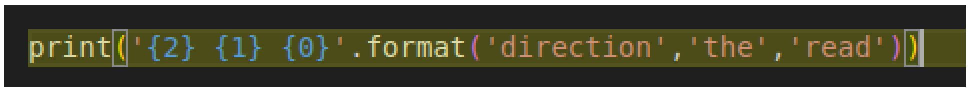


```
print('We all are {}.'.format('equal'))
```

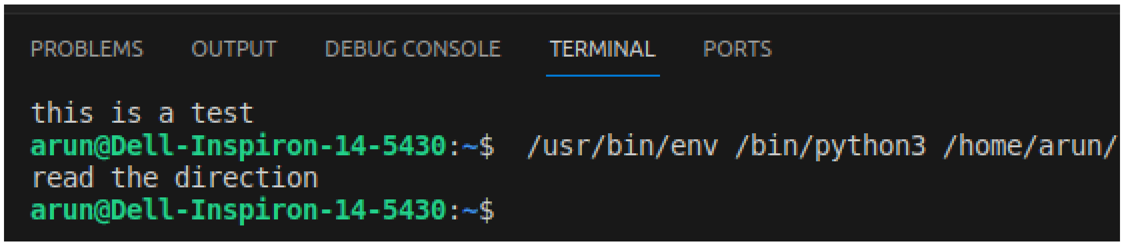


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode/extensions/m
We all are equal.
arun@Dell-Inspiron-14-5430:~$
```

b. `print('{2} {1} {0}'.format('directions','the', 'Read'))` –> Read the Instructions

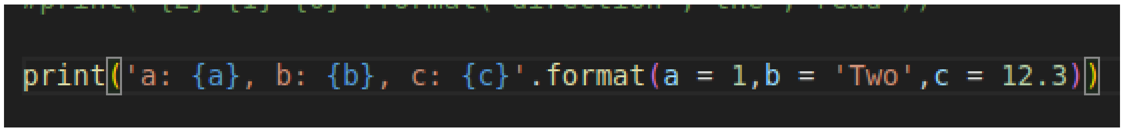


```
print('{2} {1} {0}'.format('direction','the','read'))
```

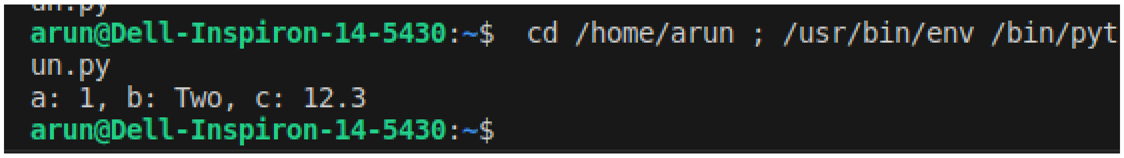


```
this is a test
arun@Dell-Inspiron-14-5430:~$  /usr/bin/env /bin/python3 /home/arun/
read the direction
arun@Dell-Inspiron-14-5430:~$
```

c. `print('a: {a}, b: {b}, c: {c}'.format(a = 1,b = 'Two',c = 12.3))` –> a:1, b: Two, c: 12.3

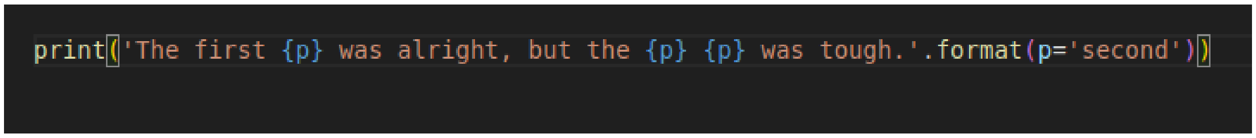


```
print('a: {a}, b: {b}, c: {c}'.format(a = 1,b = 'Two',c = 12.3))
```

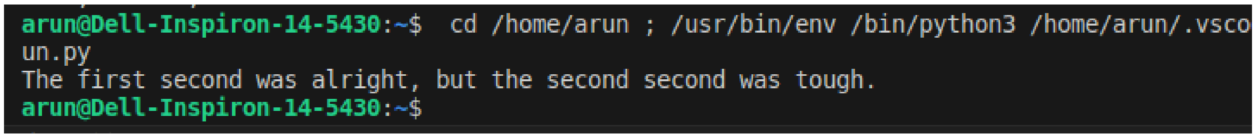


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/pyt
un.py
a: 1, b: Two, c: 12.3
arun@Dell-Inspiron-14-5430:~$
```

d. `print('The first {p} was alright, but the {p} {p} was tough.'.format(p='second'))` –> The first second was right, but the second second was tough.



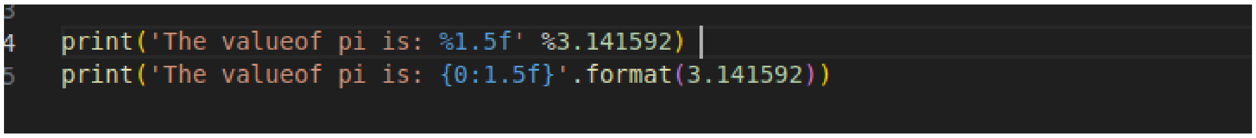
```
print('The first {p} was alright, but the {p} {p} was tough.'.format(p='second'))
```



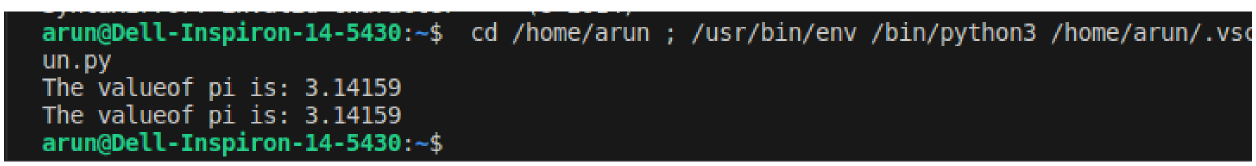
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vsco
un.py
The first second was alright, but the second second was tough.
arun@Dell-Inspiron-14-5430:~$
```

e. `print('The valueof pi is: %1.5f' %3.141592)` –> The value of pi is: 3.14159

f. `print('The valueof pi is: {0:1.5f}'.format(3.141592))` –> The value of pi is: 3.14159

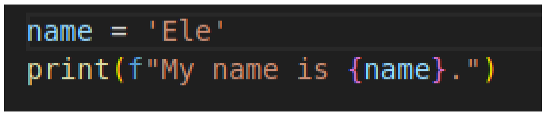


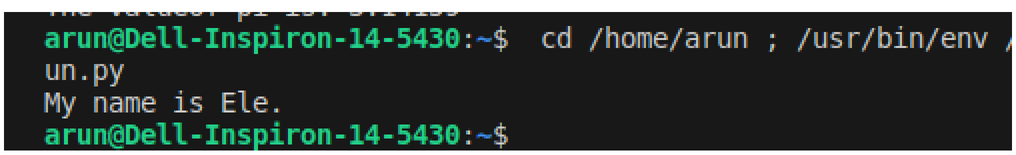
```
print('The valueof pi is: %1.5f' %3.141592)
print('The valueof pi is: {0:1.5f}'.format(3.141592))
```


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vsc
un.py
The valueof pi is: 3.14159
The valueof pi is: 3.14159
arun@Dell-Inspiron-14-5430:~$
```

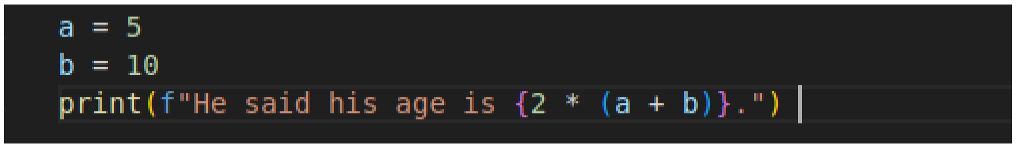
3. **Formatting with string literals, called f-strings.** – F-strings provide a concise and convenient way to embed Python expressions inside string literals for formatting.

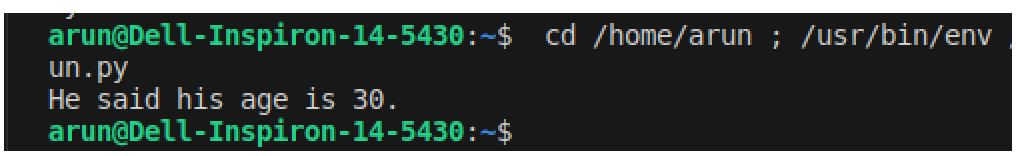
a. `name = 'Ele'`
   `print(f"My name is {name}.") -> My name is Ele.`
   (In this code, the f-string f"My name is {name}." is used to interpolate the value of the name variable into the string.)


```
name = 'Ele'
print(f"My name is {name}.")
```

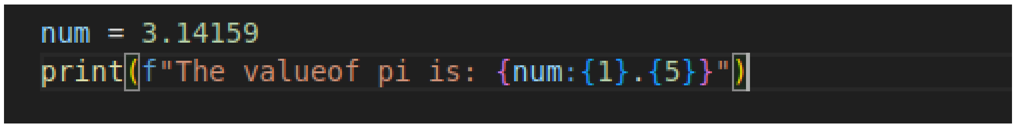

```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env
un.py
My name is Ele.
arun@Dell-Inspiron-14-5430:~$
```

b. `a = 5`
   `b = 10`
   `print(f"He said his age is {2 * (a + b)}.") -> He said his age is 30.`

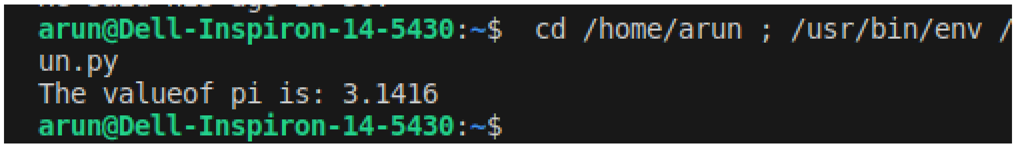

```
a = 5
b = 10
print(f"He said his age is {2 * (a + b)}.")
```


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env
un.py
He said his age is 30.
arun@Dell-Inspiron-14-5430:~$
```

```
c. num = 3.14159
   print(f"The valueof pi is: {num:{1}.{5}}") –> The valueof pi
   is: 3.1416
```



```
num = 3.14159
print(f"The valueof pi is: {num:{1}.{5}}")
```

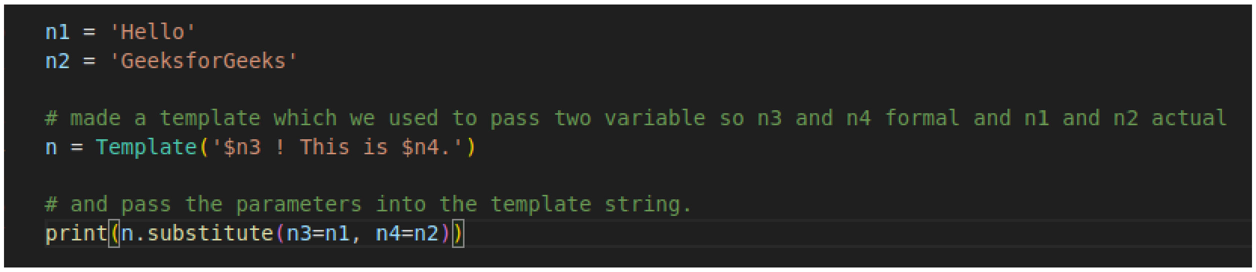


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /
un.py
The valueof pi is: 3.1416
arun@Dell-Inspiron-14-5430:~$
```

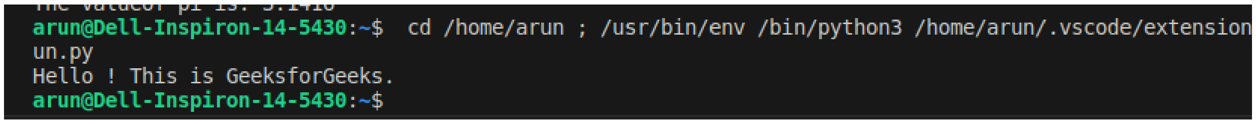
4. **Formatting with String Template Class**

```
from string import Template

n1 = 'Hello'
n2 = 'GeeksforGeeks'

# made a template which we used to pass two variable so n3 and n4
formal and n1 and n2 actual
n = Template('$n3 ! This is $n4.')

# and pass the parameters into the template string.
print(n.substitute(n3=n1, n4=n2))
```

OUTPUT: `Hello ! This is GeeksforGeeks.`



```
n1 = 'Hello'
n2 = 'GeeksforGeeks'

# made a template which we used to pass two variable so n3 and n4 formal and n1 and n2 actual
n = Template('$n3 ! This is $n4.')

# and pass the parameters into the template string.
print(n.substitute(n3=n1, n4=n2))
```

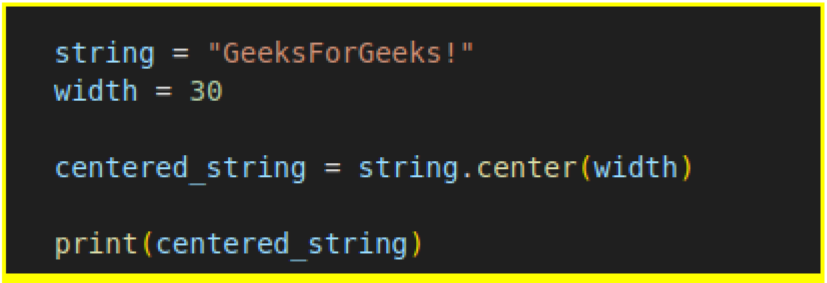


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode/extension
un.py
Hello ! This is GeeksforGeeks.
arun@Dell-Inspiron-14-5430:~$
```

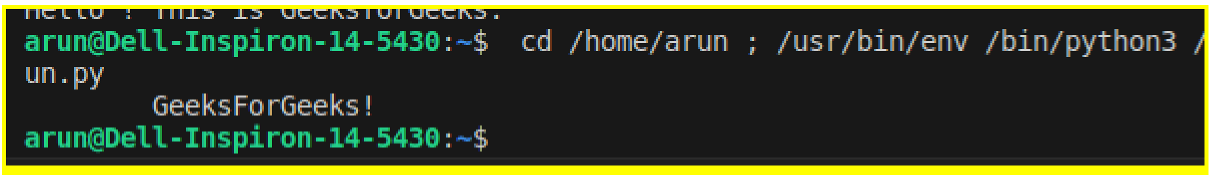
5. **Formatting with center() string method.**

```
string = "GeeksForGeeks!"
width = 30

centered_string = string.center(width)

print(centered_string)
```

OUTPUT: `        GeeksForGeeks!`



```
string = "GeeksForGeeks!"
width = 30

centered_string = string.center(width)

print(centered_string)
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /
un.py
        GeeksForGeeks!
arun@Dell-Inspiron-14-5430:~$
```

7. **Python String Index**

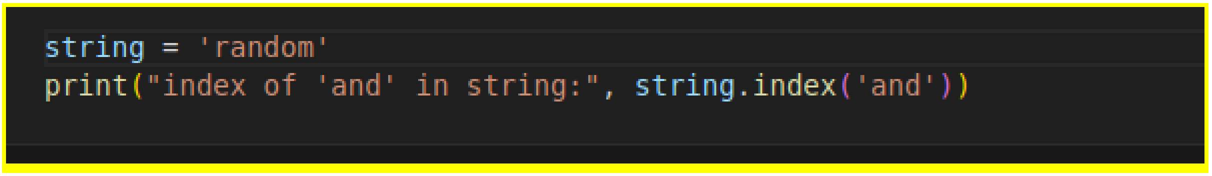
a.
```
string = 'random'
print("index of 'and' in string:", string.index('and'))
```

OUTPUT: `Index of 'and' in string: 1`

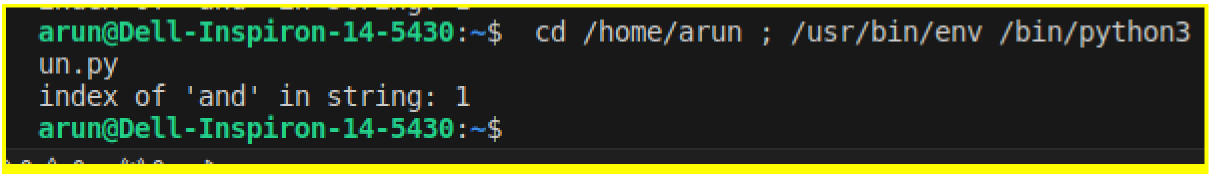
b. Python String Index() with Start Argument

```
# initializing target string
ch = "geeksforgeeks"

# initializing argument string
ch1 = "geeks"
```

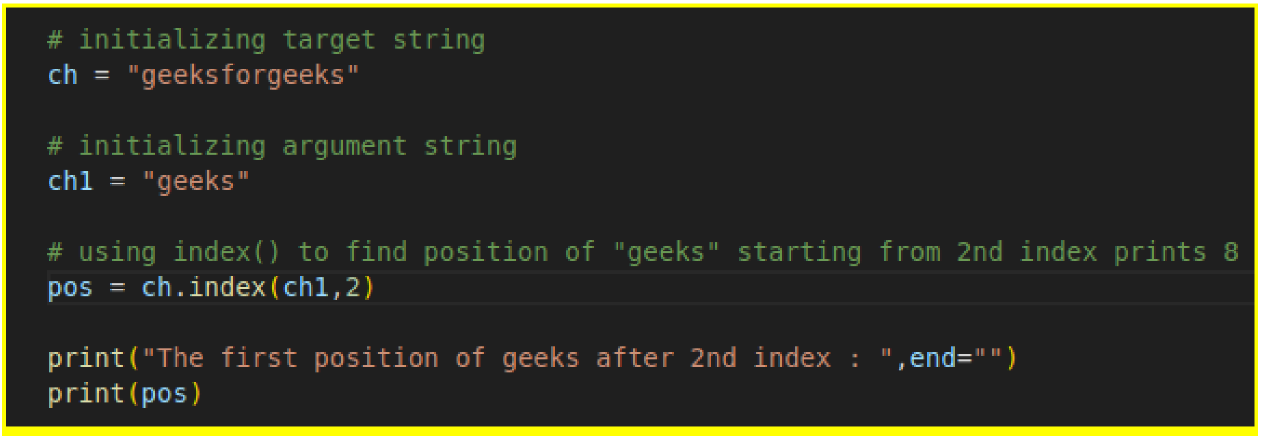
```
# using index() to find position of "geeks" starting from 2nd
index prints 8
pos = ch.index(ch1,2) -> '2" is used for slicing

print("The first position of geeks after 2nd index : ",end="")
print(pos)

OUTPUT: The first position of geeks after 2nd index : 8
```



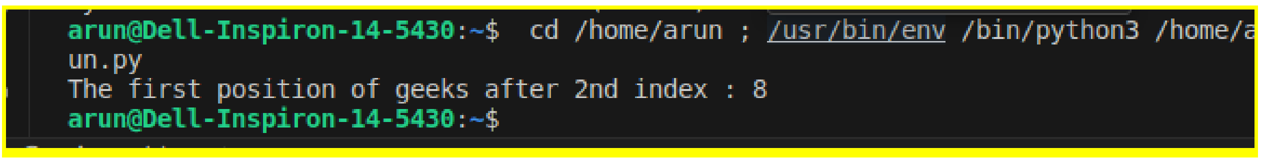
```
string = 'random'
print("index of 'and' in string:", string.index('and'))
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3
un.py
index of 'and' in string: 1
arun@Dell-Inspiron-14-5430:~$
```



```
# initializing target string
ch = "geeksforgeeks"

# initializing argument string
ch1 = "geeks"

# using index() to find position of "geeks" starting from 2nd index prints 8
pos = ch.index(ch1,2)

print("The first position of geeks after 2nd index : ",end="")
print(pos)
```



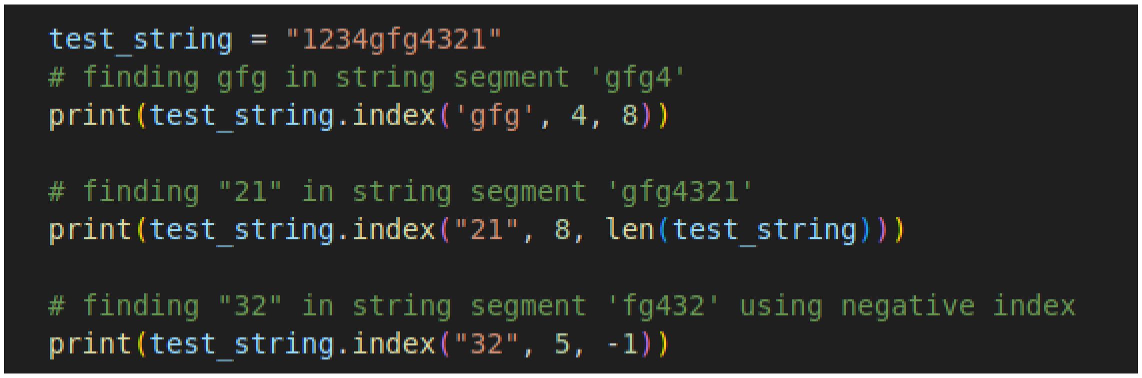
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/a
un.py
The first position of geeks after 2nd index : 8
arun@Dell-Inspiron-14-5430:~$
```

c. Python String Index() with Start and End Arguments

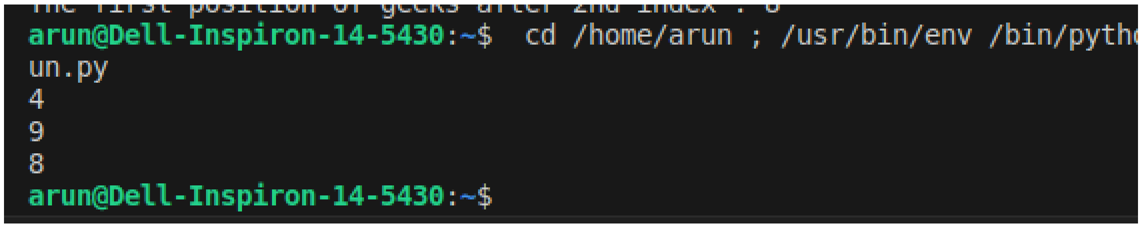
```
test_string = "1234gfg4321"
# finding gfg in string segment 'gfg4'
print(test_string.index('gfg', 4, 8))

# finding "21" in string segment 'gfg4321'
```

```
print(test_string.index("21", 8, len(test_string)))

# finding "32" in string segment 'fg432' using negative index
print(test_string.index("32", 5, -1))
```

```
OUTPUT:
4
9
8
```



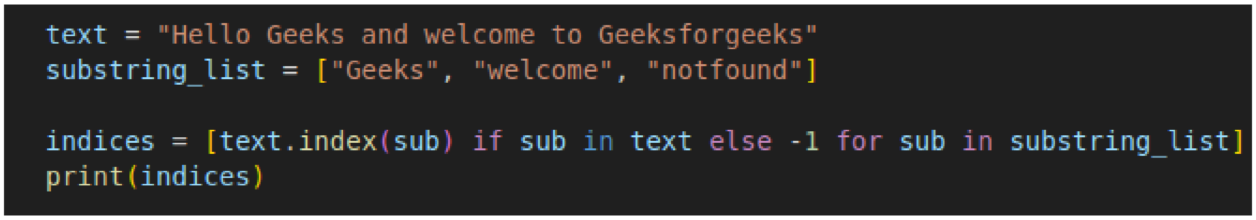
```
test_string = "1234gfg4321"
# finding gfg in string segment 'gfg4'
print(test_string.index('gfg', 4, 8))

# finding "21" in string segment 'gfg4321'
print(test_string.index("21", 8, len(test_string)))

# finding "32" in string segment 'fg432' using negative index
print(test_string.index("32", 5, -1))
```



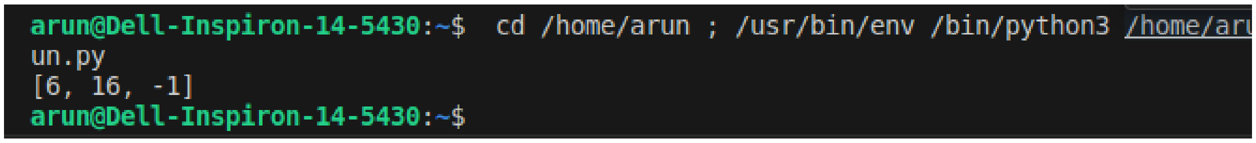
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/pyth
un.py
4
9
8
arun@Dell-Inspiron-14-5430:~$
```

```
d.text = "Hello Geeks and welcome to Geeksforgeeks"
substring_list = ["Geeks", "welcome", "notfound"]

indices = [text.index(sub) if sub in text else -1 for sub
in substring_list]
print(indices)
```

OUTPUT: [6,16,-1]



```
text = "Hello Geeks and welcome to Geeksforgeeks"
substring_list = ["Geeks", "welcome", "notfound"]

indices = [text.index(sub) if sub in text else -1 for sub in substring_list]
print(indices)
```



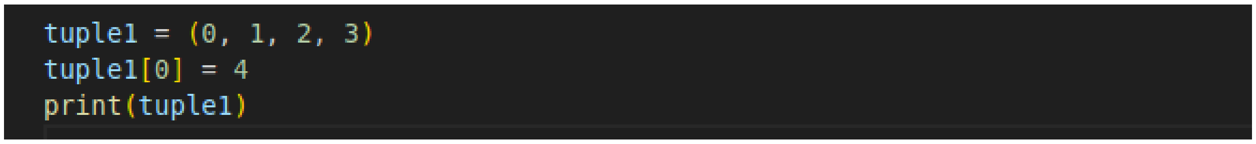
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/aru
un.py
[6, 16, -1]
arun@Dell-Inspiron-14-5430:~$
```

8. Immutability:-

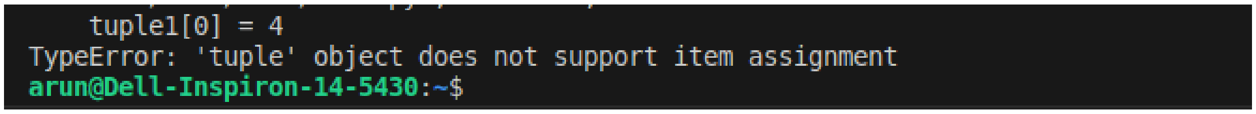
Immutable –> Int, float, bool, string, Unicode and tuple

a.
```
tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```

–> Will throw an error stating "assignment" not supported



```
tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```



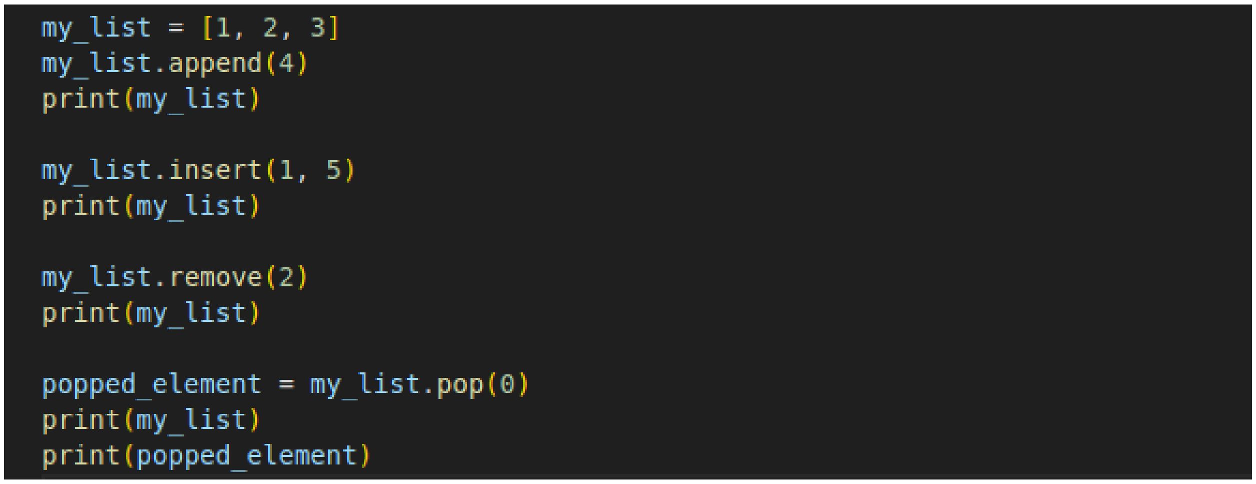
```
    tuple1[0] = 4
TypeError: 'tuple' object does not support item assignment
arun@Dell-Inspiron-14-5430:~$
```

Mutable –> list, dictionary, set

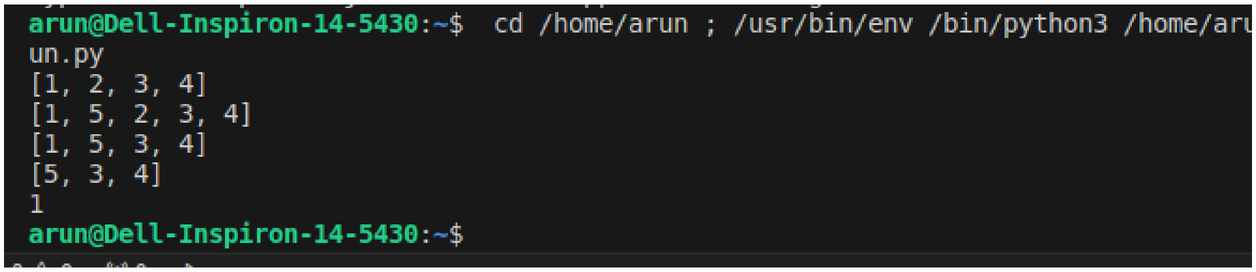
a.
```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)

my_list.insert(1, 5)
print(my_list)

my_list.remove(2)
print(my_list)

popped_element = my_list.pop(0)
print(my_list)
print(popped_element)
```

OUTPUT:

```
[1, 2, 3, 4]
[1, 5, 2, 3, 4]
[1, 5, 3, 4]
[5, 3, 4]
1
```



```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)

my_list.insert(1, 5)
print(my_list)

my_list.remove(2)
print(my_list)

popped_element = my_list.pop(0)
print(my_list)
print(popped_element)
```

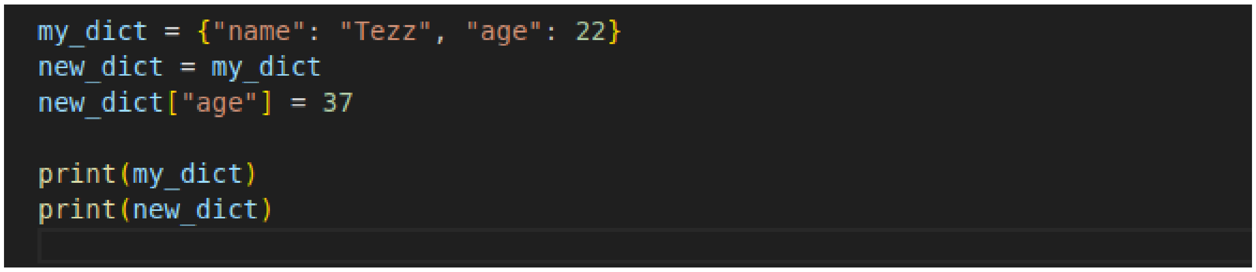


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/aru
un.py
[1, 2, 3, 4]
[1, 5, 2, 3, 4]
[1, 5, 3, 4]
[5, 3, 4]
1
arun@Dell-Inspiron-14-5430:~$
```

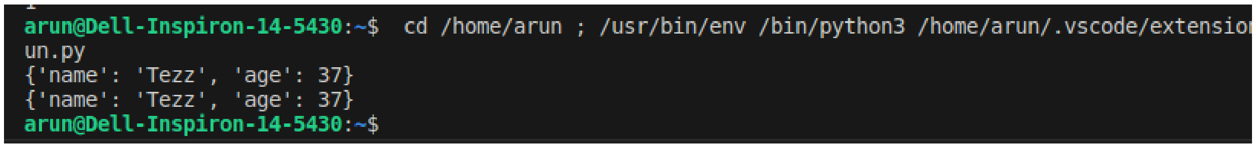
```
b.my_dict = {"name": "Tezz", "age": 22}
new_dict = my_dict
new_dict["age"] = 37

print(my_dict)
print(new_dict)
```

OUTPUT:

```
{'name': 'Ram', 'age': 37}
{'name': 'Ram', 'age': 37}
```

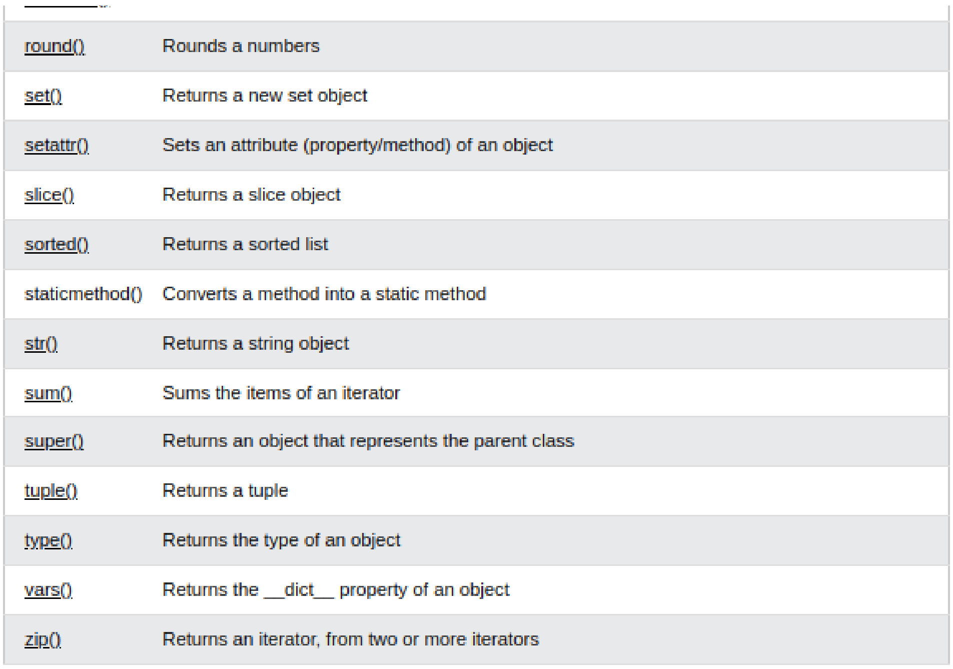


```python
my_dict = {"name": "Tezz", "age": 22}
new_dict = my_dict
new_dict["age"] = 37

print(my_dict)
print(new_dict)
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode/extensio
un.py
{'name': 'Tezz', 'age': 37}
{'name': 'Tezz', 'age': 37}
arun@Dell-Inspiron-14-5430:~$
```

10. **Built-In Functions + Methods**



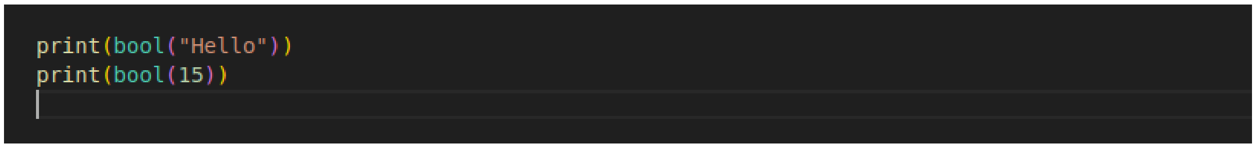
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
staticmethod()	Converts a method into a static method
str()	Returns a string object
sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary

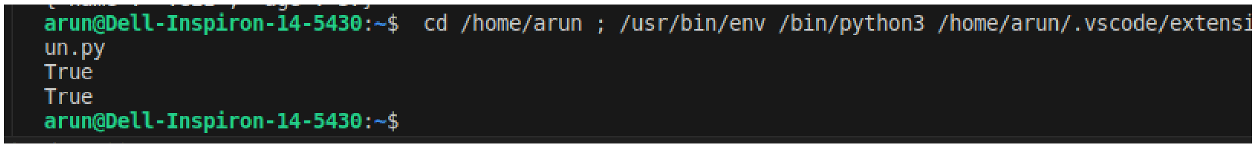
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator

11. Booleans :- Booleans represent one of two values i.e., True or False

a.
```
print(bool("Hello"))
print(bool(15))

OUTPUT:
True
True
```



```
print(bool("Hello"))
print(bool(15))
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode/extensi
un.py
True
True
arun@Dell-Inspiron-14-5430:~$
```

Note:

1. Almost any value is evaluated to `True` if it has some sort of content.
2. Any string is `True`, except empty strings.
3. Any number is `True`, except `0`.
4. Any list, tuple, set, and dictionary are `True`, except empty ones.

12. Lists : are used to store multiple items in a single variable. List is ordered, changeable and allow duplicate values.

```
mylist = ["apple", "banana", "cherry", “apple”]
```

a. To find length –> **len(myList)**

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

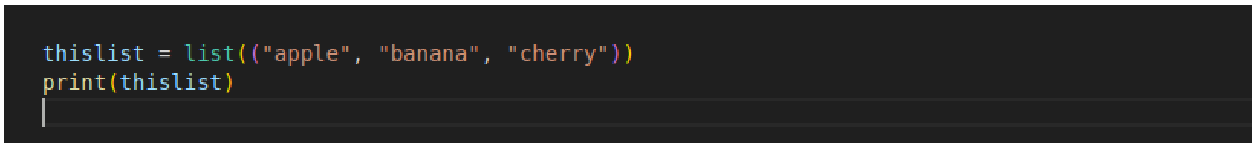
b. **List Constructor** to make a new list

```
thislist = list(("apple", "banana", "cherry"))
```

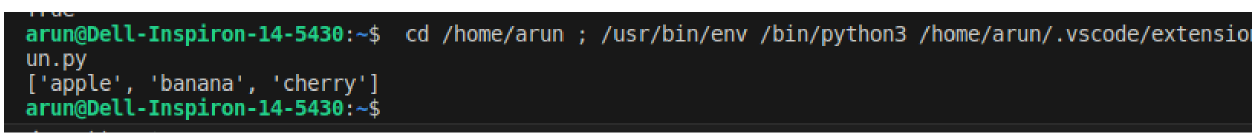
```
print(thislist)
```

OUTPUT:

```
["apple", "banana", "cherry"]
```



```
thislist = list(("apple", "banana", "cherry"))
print(thislist)
```



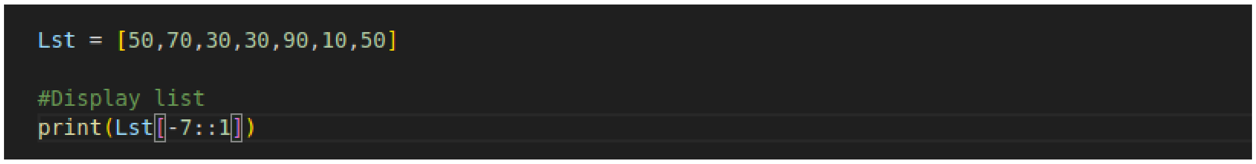
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode/extensio
un.py
['apple', 'banana', 'cherry']
arun@Dell-Inspiron-14-5430:~$
```

13. Python List Slicing

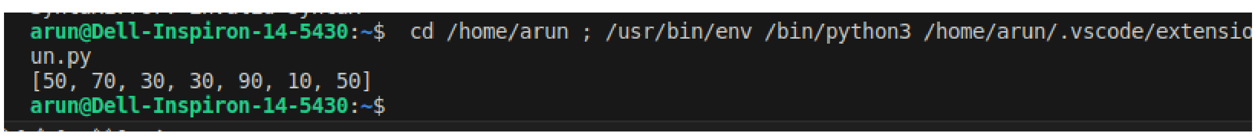
Syntax: List[Initial : End : IndexJump]

a. Positive Index: **0** →

b. Negative Index: **-1 from last**



```
Lst = [50,70,30,30,90,10,50]

#Display list
print(Lst[-7::1])
```



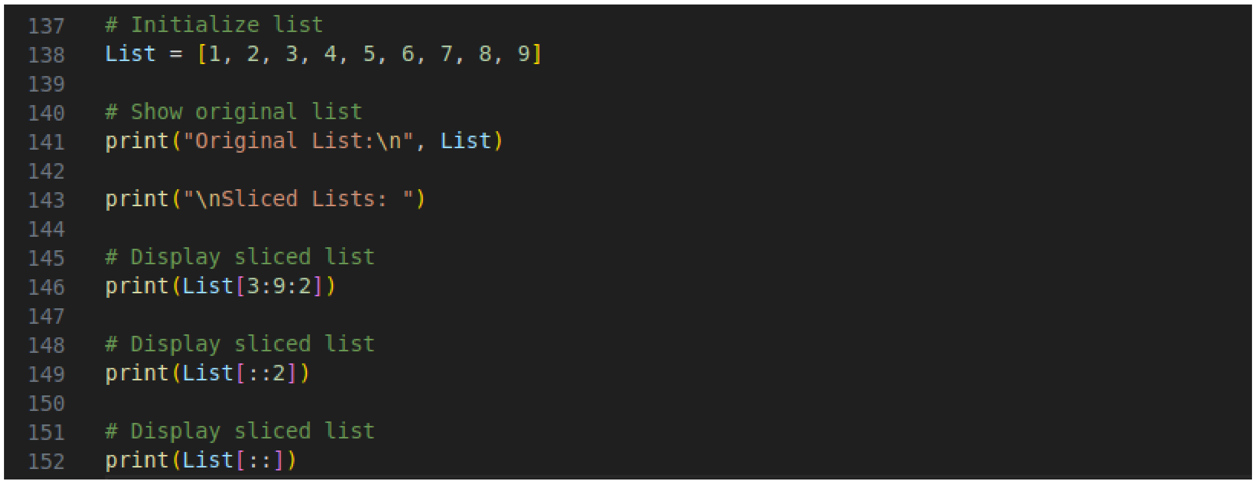
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode/extensio
un.py
[50, 70, 30, 30, 90, 10, 50]
arun@Dell-Inspiron-14-5430:~$
```

a.
```
# Initialize list
List = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Show original list
print("Original List:\n", List)

print("\nSliced Lists: ")

# Display sliced list
print(List[3:9:2])

# Display sliced list
print(List[::2])
```

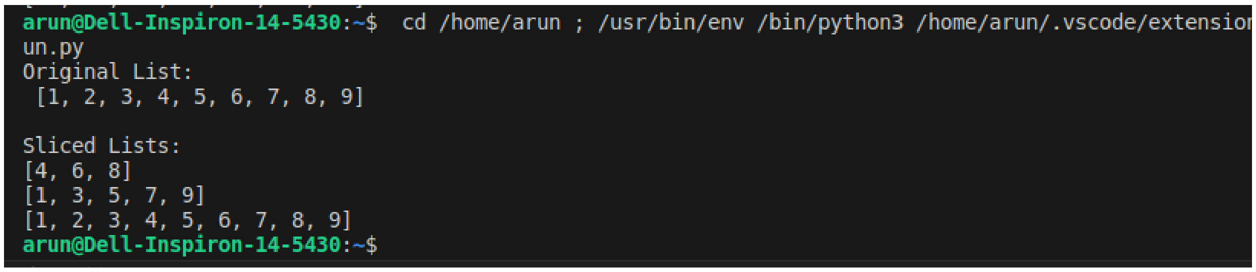
```
# Display sliced list
print(List[::])
```

OUTPUT:

```
Original List:
 [1, 2, 3, 4, 5, 6, 7, 8, 9]

Sliced Lists:
[4, 6, 8]
[1, 3, 5, 7, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```



```
# Initialize list
List = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Show original list
print("Original List:\n", List)

print("\nSliced Lists: ")

# Display sliced list
print(List[3:9:2])

# Display sliced list
print(List[::2])

# Display sliced list
print(List[::])
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode/extensior
un.py
Original List:
 [1, 2, 3, 4, 5, 6, 7, 8, 9]

Sliced Lists:
[4, 6, 8]
[1, 3, 5, 7, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
arun@Dell-Inspiron-14-5430:~$
```

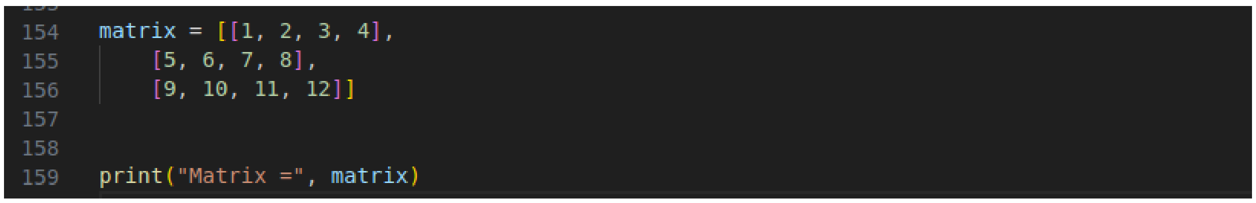
14. **Matrix:** A matrix is a collection of numbers arranged in a rectangular array in rows and columns.

a.
```
matrix = [[1, 2, 3, 4],
          [5, 6, 7, 8],
          [9, 10, 11, 12]]
```

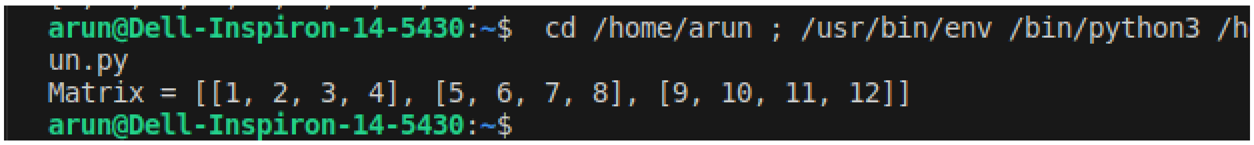
```
print("Matrix =", matrix)
```

OUTPUT:

```
Matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

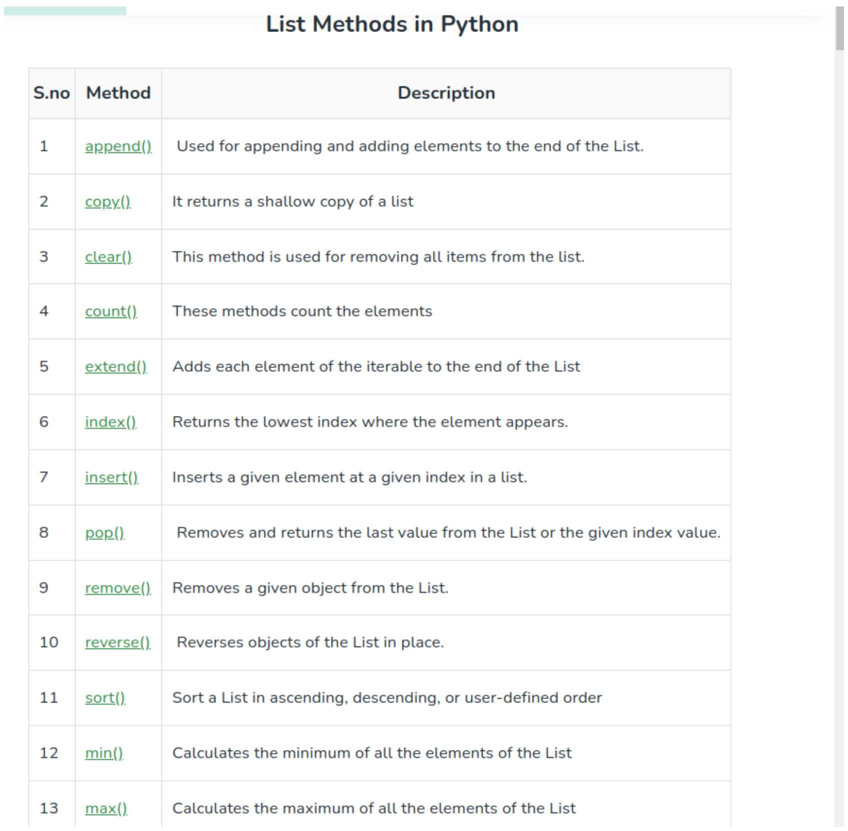


```
matrix = [[1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]]


print("Matrix =", matrix)
```



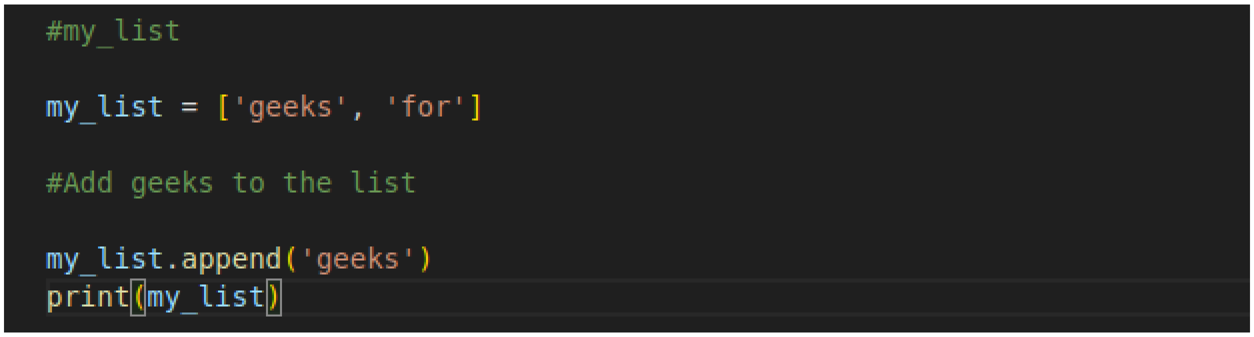
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /h
un.py
Matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
arun@Dell-Inspiron-14-5430:~$
```

15. **List Methods:**

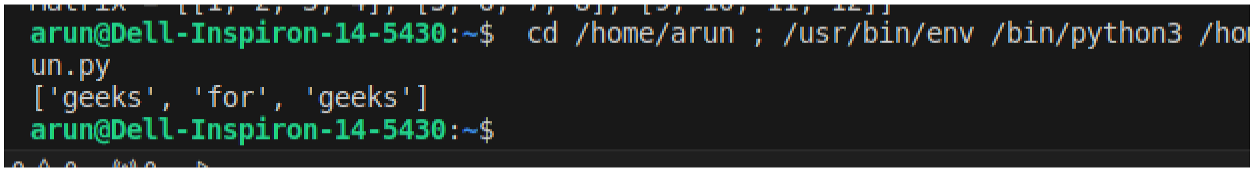


List Methods in Python

S.no	Method	Description
1	append()	Used for appending and adding elements to the end of the List.
2	copy()	It returns a shallow copy of a list
3	clear()	This method is used for removing all items from the list.
4	count()	These methods count the elements
5	extend()	Adds each element of the iterable to the end of the List
6	index()	Returns the lowest index where the element appears.
7	insert()	Inserts a given element at a given index in a list.
8	pop()	Removes and returns the last value from the List or the given index value.
9	remove()	Removes a given object from the List.
10	reverse()	Reverses objects of the List in place.
11	sort()	Sort a List in ascending, descending, or user-defined order
12	min()	Calculates the minimum of all the elements of the List
13	max()	Calculates the maximum of all the elements of the List



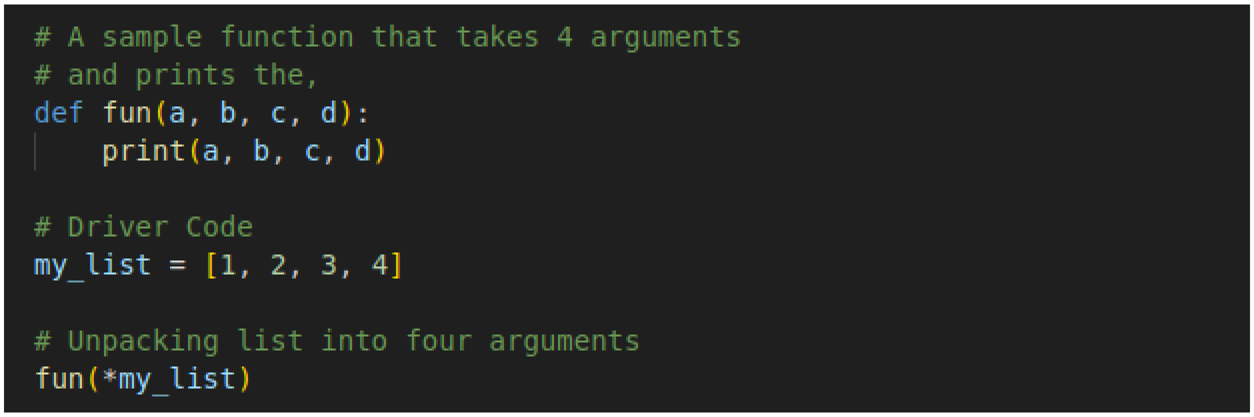
```
#my_list

my_list = ['geeks', 'for']

#Add geeks to the list

my_list.append('geeks')
print(my_list)
```



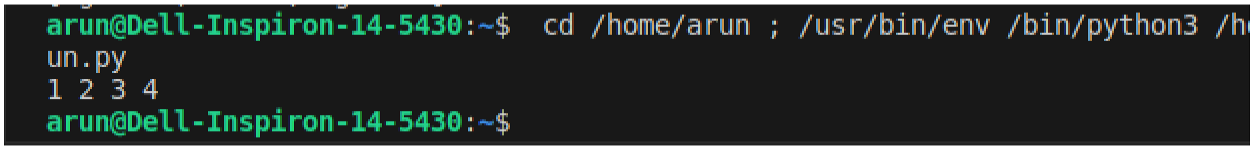
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
['geeks', 'for', 'geeks']
arun@Dell-Inspiron-14-5430:~$
```

16. Packing and Unpacking

a.
```
# A sample function that takes 4 arguments
# and prints the,
def fun(a, b, c, d):
    print(a, b, c, d)

# Driver Code
my_list = [1, 2, 3, 4]

# Unpacking list into four arguments
fun(*my_list)

OUTPUT:
(1,2,3,4)
```



```python
# A sample function that takes 4 arguments
# and prints the,
def fun(a, b, c, d):
    print(a, b, c, d)

# Driver Code
my_list = [1, 2, 3, 4]

# Unpacking list into four arguments
fun(*my_list)
```



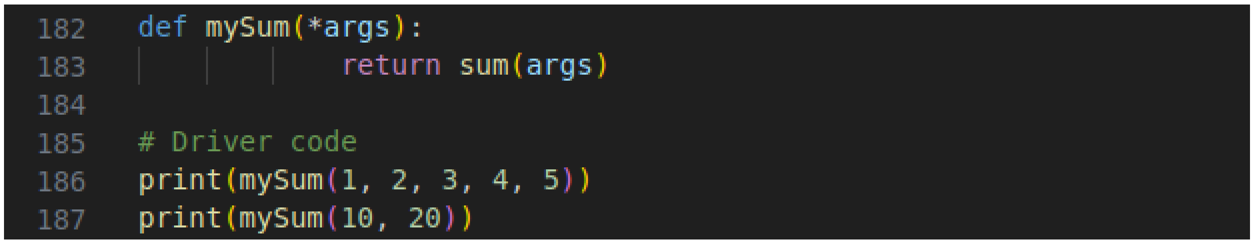
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /h
un.py
1 2 3 4
arun@Dell-Inspiron-14-5430:~$
```

1. **Packing:** When we don't know how many arguments need to be passed to a python function, we can use Packing to pack all arguments in a tuple.

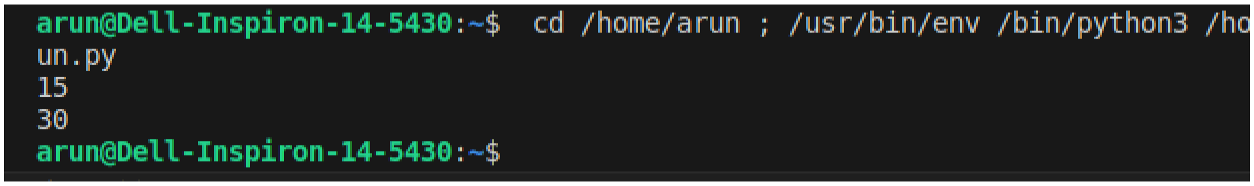
a.
```python
# A Python program to demonstrate use of packing
# This function uses packing to sum unknown number of arguments
def mySum(*args):
     return sum(args)
# Driver code
print(mySum(1, 2, 3, 4, 5))
print(mySum(10, 20))
```

OUTPUT:

```
15
30
```



```python
def mySum(*args):
        return sum(args)

# Driver code
print(mySum(1, 2, 3, 4, 5))
print(mySum(10, 20))
```

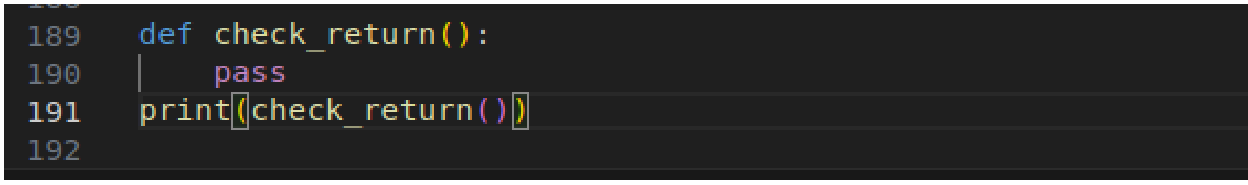


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
15
30
arun@Dell-Inspiron-14-5430:~$
```

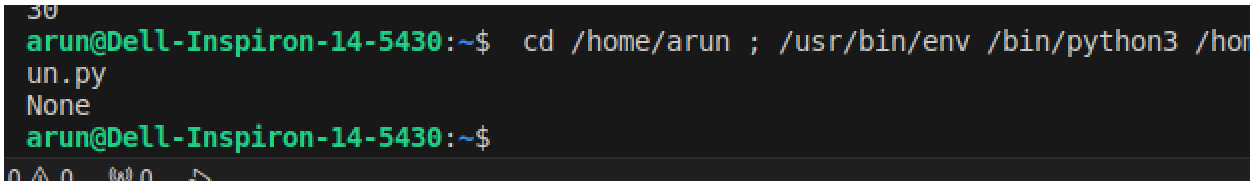
17. **None:-** None is used to define a null value or Null object in Python. It is not the same as an empty string, a False, or a zero. It is a data type of the class NoneType object.

a.
```
def check_return():
    pass
print(check_return())
```

OUTPUT: None



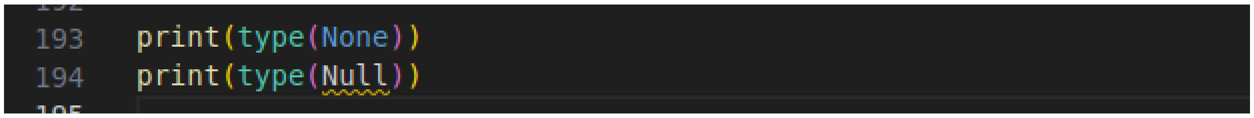
```
def check_return():
    pass
print(check_return())
```



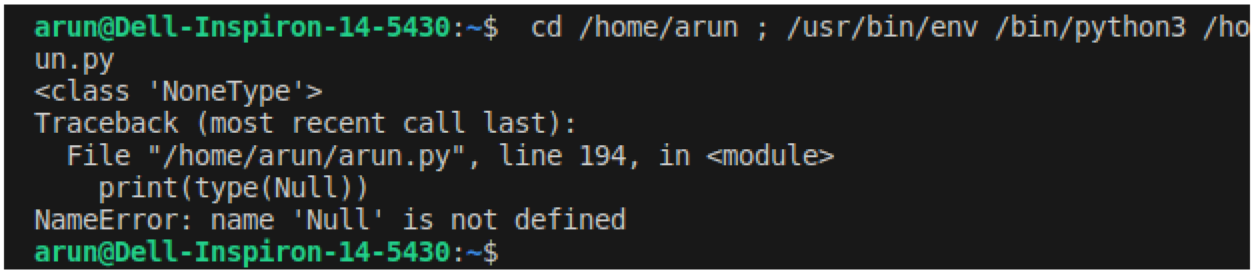
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /hom
un.py
None
arun@Dell-Inspiron-14-5430:~$
```

b.
```
print(type(None))

print(type(Null))
```

OUTPUT:
```
<class 'NoneType'>
NameError -> As there is no Null in Python
```



```
print(type(None))
print(type(Null))
```



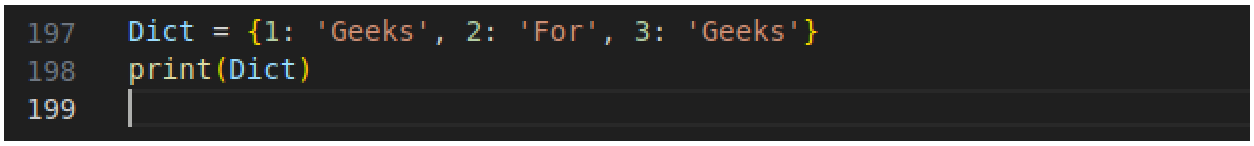
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
<class 'NoneType'>
Traceback (most recent call last):
  File "/home/arun/arun.py", line 194, in <module>
    print(type(Null))
NameError: name 'Null' is not defined
arun@Dell-Inspiron-14-5430:~$
```

c. **Note:** If a function does not return anything, it returns None in Python.

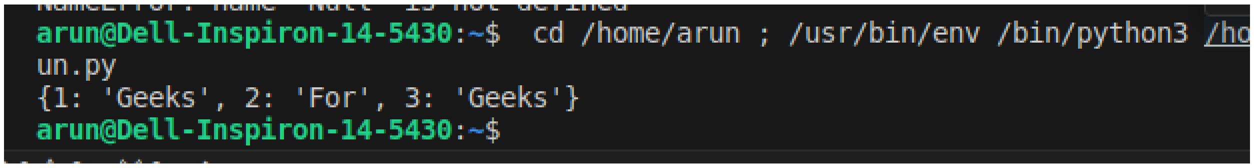
18. **Dictionary in Python** is a collection of keys values, used to store data values like a map.

a.
```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print(Dict)
```

```
OUTPUT:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```



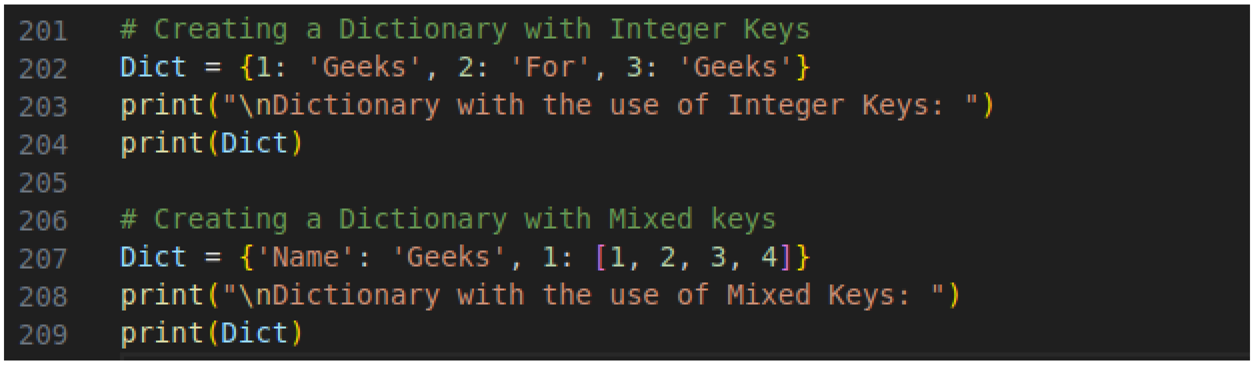
```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print(Dict)
```



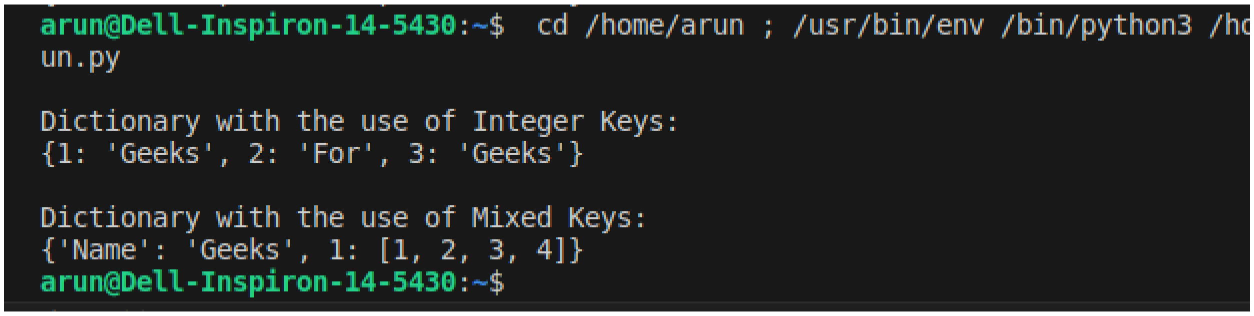
```
NameError: name 'Null' is not defined
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
arun@Dell-Inspiron-14-5430:~$
```

b.
```
# Creating a Dictionary with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

```
OUTPUT:
Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with the use of Mixed Keys:
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
```



```
# Creating a Dictionary with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

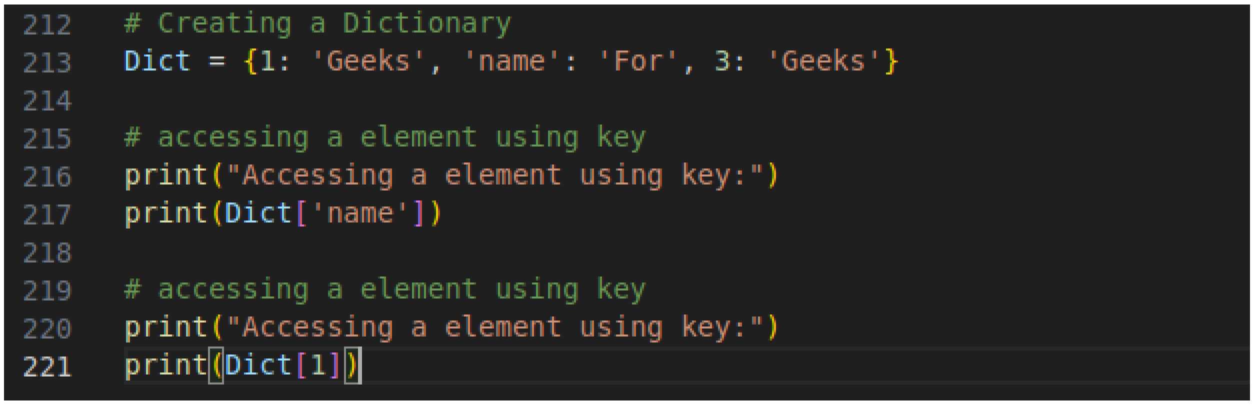


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py

Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with the use of Mixed Keys:
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
arun@Dell-Inspiron-14-5430:~$
```

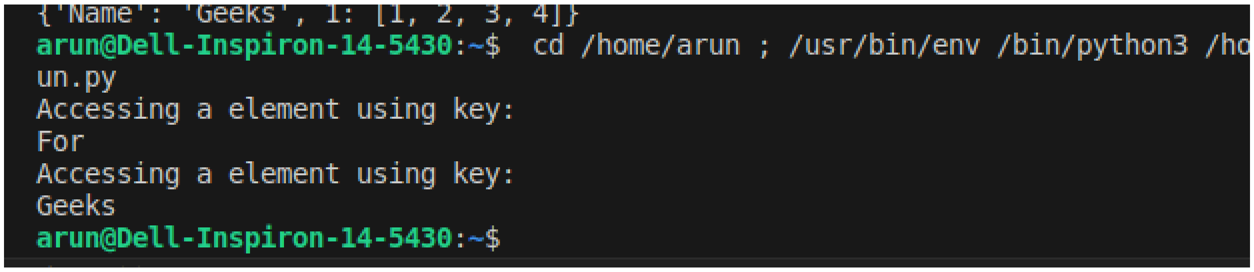
```
c.# Creating a Dictionary

Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])

# accessing a element using key
print("Accessing a element using key:")
print(Dict[1])
```

OUTPUT:

```
Accessing a element using key:
For
Accessing a element using key:
Geeks
```



```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])

# accessing a element using key
print("Accessing a element using key:")
print(Dict[1])
```



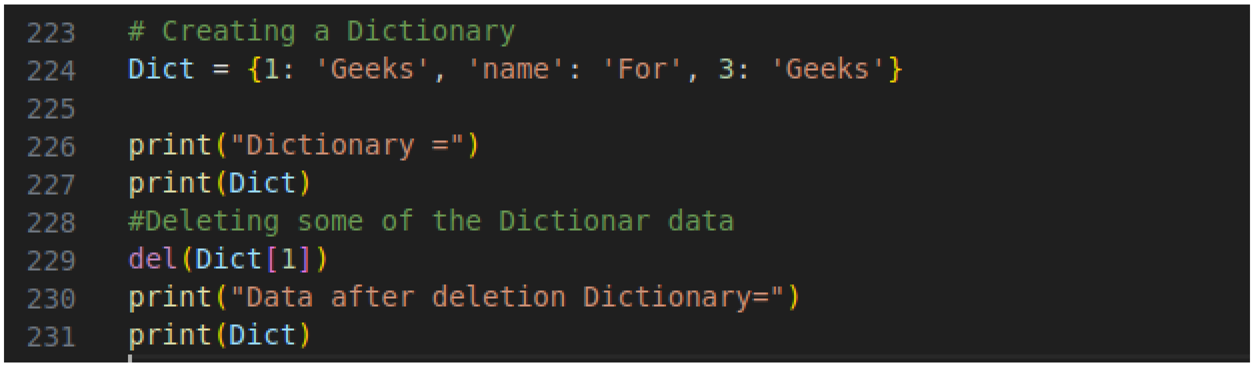
```
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
Accessing a element using key:
For
Accessing a element using key:
Geeks
arun@Dell-Inspiron-14-5430:~$
```

d. DELETING

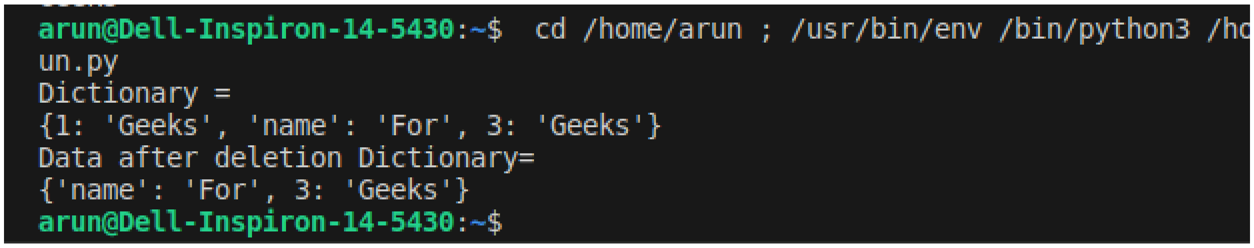
```
# Creating a Dictionary
     Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

     print("Dictionary =")
     print(Dict)
     #Deleting some of the Dictionar data
     del(Dict[1])
     print("Data after deletion Dictionary=")
     print(Dict)
```

OUTPUT:

```
Dictionary ={1: 'Geeks', 'name': 'For', 3: 'Geeks'}
Data after deletion Dictionary={'name': 'For', 3: 'Geeks'}
```

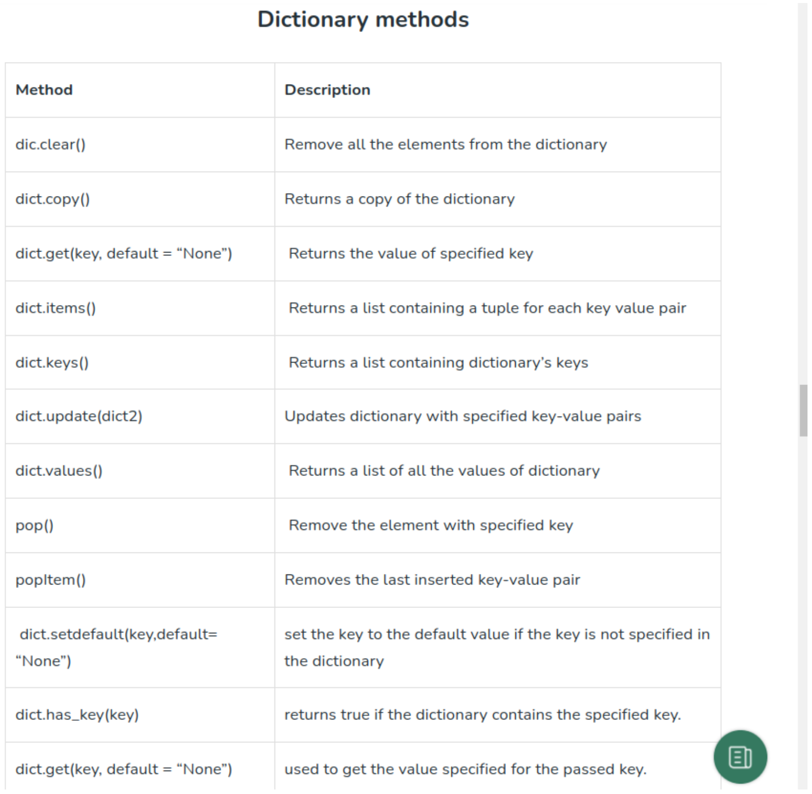


```python
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

print("Dictionary =")
print(Dict)
#Deleting some of the Dictionar data
del(Dict[1])
print("Data after deletion Dictionary=")
print(Dict)
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
Dictionary =
{1: 'Geeks', 'name': 'For', 3: 'Geeks'}
Data after deletion Dictionary=
{'name': 'For', 3: 'Geeks'}
arun@Dell-Inspiron-14-5430:~$
```

19. Dictionary Methods:



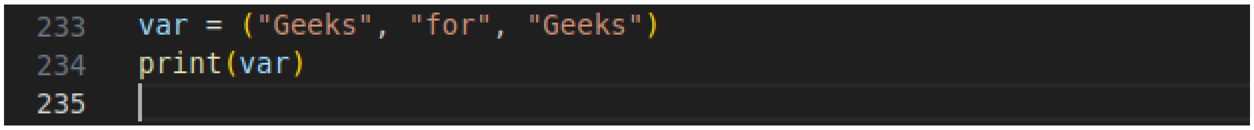
Dictionary methods

Method	Description
dic.clear()	Remove all the elements from the dictionary
dict.copy()	Returns a copy of the dictionary
dict.get(key, default = “None”)	Returns the value of specified key
dict.items()	Returns a list containing a tuple for each key value pair
dict.keys()	Returns a list containing dictionary’s keys
dict.update(dict2)	Updates dictionary with specified key-value pairs
dict.values()	Returns a list of all the values of dictionary
pop()	Remove the element with specified key
popItem()	Removes the last inserted key-value pair
dict.setdefault(key,default= “None”)	set the key to the default value if the key is not specified in the dictionary
dict.has_key(key)	returns true if the dictionary contains the specified key.
dict.get(key, default = “None”)	used to get the value specified for the passed key.

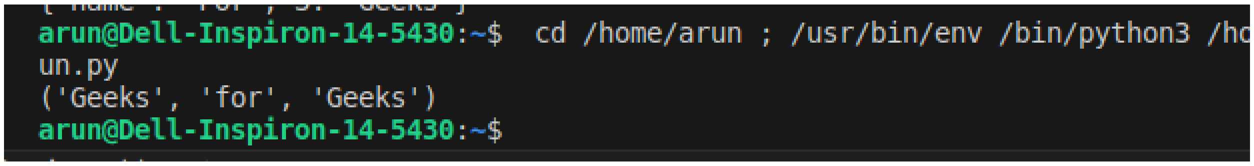
20. Tuple: Python Tuple is a collection of objects separated by commas.

a.
```
var = ("Geeks", "for", "Geeks")
print(var)
```

Output:
```
("Geeks", "for", "Geeks")
```



```
var = ("Geeks", "for", "Geeks")
print(var)
```



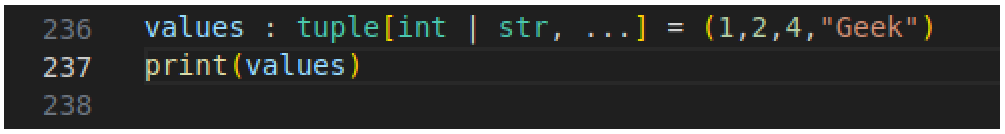
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
('Geeks', 'for', 'Geeks')
arun@Dell-Inspiron-14-5430:~$
```

b.
```
values : tuple[int | str, ...] = (1,2,4,"Geek")
print(values)
```

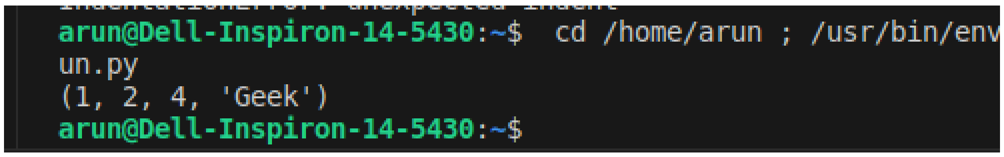
Output:

(1, 2, 4, 'Geek')

Here, in the above snippet we are considering a variable called values which holds a tuple that consists of either int or str, the '…' means that the tuple will hold more than one int or str.



```
values : tuple[int | str, ...] = (1,2,4,"Geek")
print(values)
```

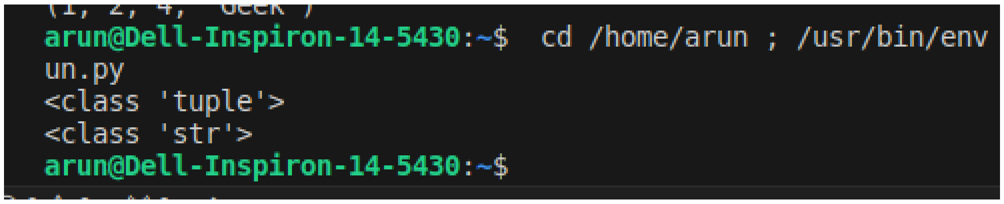


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env
un.py
(1, 2, 4, 'Geek')
arun@Dell-Inspiron-14-5430:~$
```

```
c.mytuple = ("Geeks",)
print(type(mytuple))

#NOT a tuple
mytuple = ("Geeks")
print(type(mytuple))
```

OUTPUT:

```
<class 'tuple'>
<class 'str'>
```

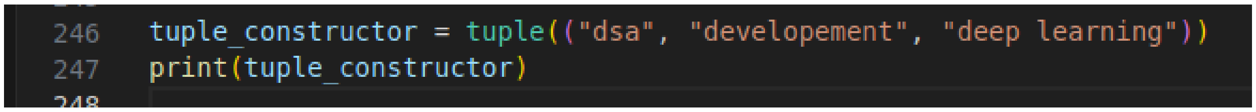


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env
un.py
<class 'tuple'>
<class 'str'>
arun@Dell-Inspiron-14-5430:~$
```

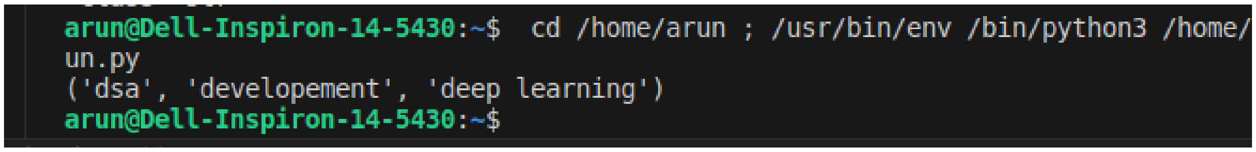
```
d.tuple_constructor = tuple(("dsa", "developement", "deep
learning"))

print(tuple_constructor)
```

OUTPUT:

```
('dsa', 'developement', 'deep learning')
```



```
tuple_constructor = tuple(("dsa", "developement", "deep learning"))
print(tuple_constructor)
```

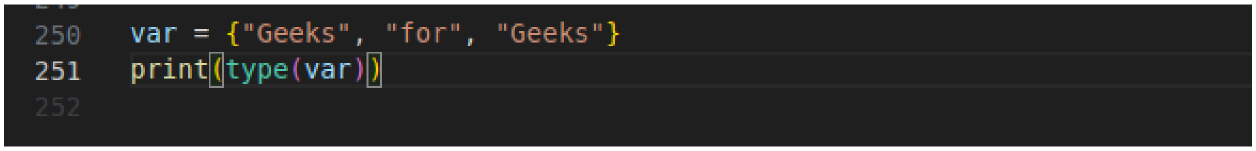


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/
un.py
('dsa', 'developement', 'deep learning')
arun@Dell-Inspiron-14-5430:~$
```

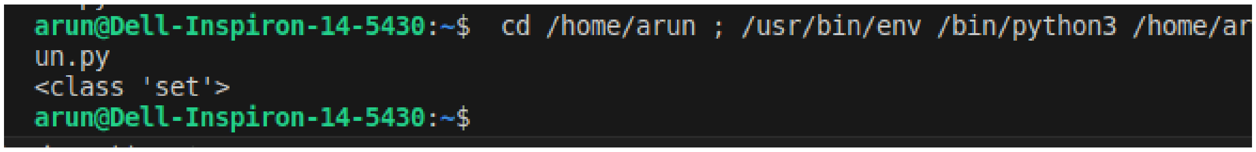
21. Sets: is an unordered collection data type that is iterable, mutable and has no duplicate elements

a.
```
var = {"Geeks", "for", "Geeks"}
print(type(var))
```

OUTPUT: Set



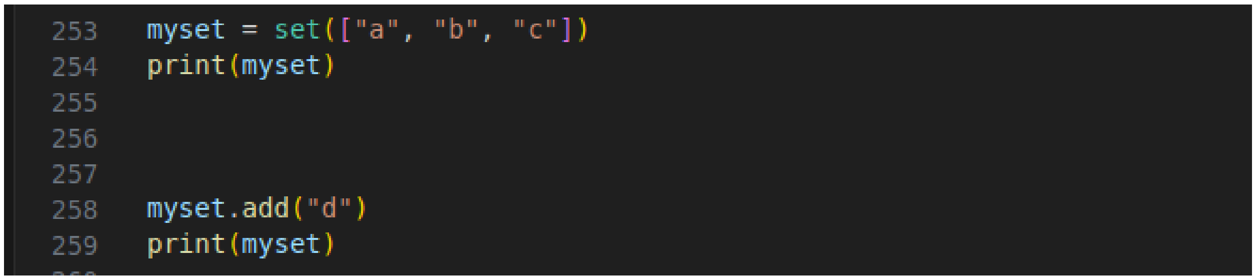
```
var = {"Geeks", "for", "Geeks"}
print(type(var))
```



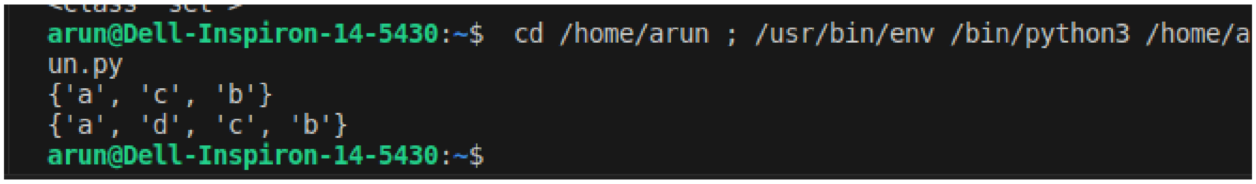
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/ar
un.py
<class 'set'>
arun@Dell-Inspiron-14-5430:~$
```

b.
```
myset = set(["a", "b", "c"])
print(myset)


myset.add("d")
print(myset)
```

OUTPUT:
```
{'c', 'b', 'a'}
{'d', 'c', 'b', 'a'}
```



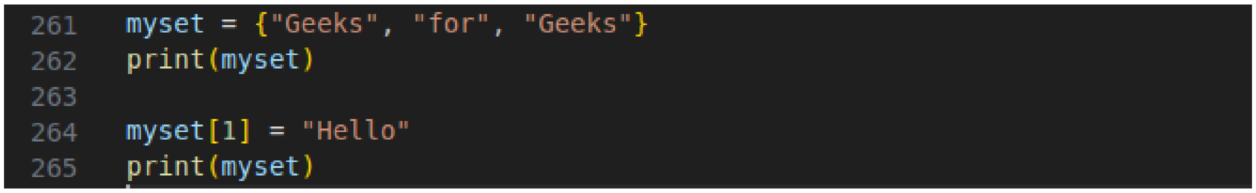
```
myset = set(["a", "b", "c"])
print(myset)



myset.add("d")
print(myset)
```



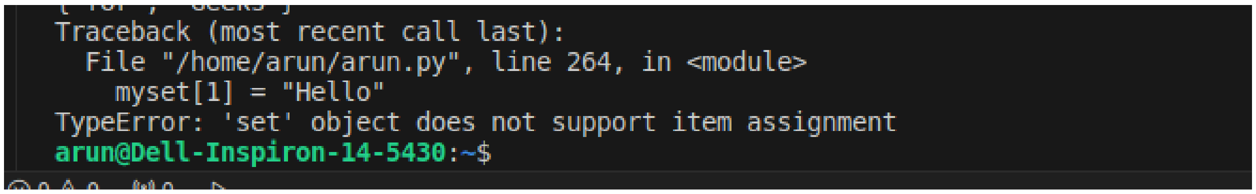
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/a
un.py
{'a', 'c', 'b'}
{'a', 'd', 'c', 'b'}
arun@Dell-Inspiron-14-5430:~$
```

c.
```
myset = {"Geeks", "for", "Geeks"}
print(myset)

myset[1] = "Hello"
print(myset)
```

OUTPUT:
```
{'Geeks', 'for'}
TypeError: 'set' object does not support item assignment
```



```python
myset = {"Geeks", "for", "Geeks"}
print(myset)

myset[1] = "Hello"
print(myset)
```

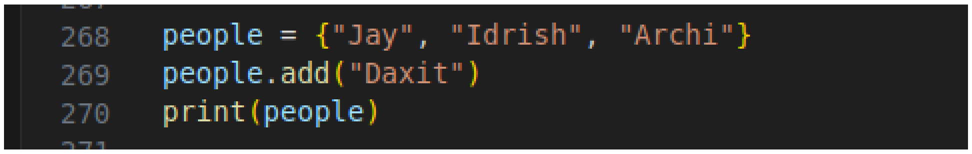


```
Traceback (most recent call last):
  File "/home/arun/arun.py", line 264, in <module>
    myset[1] = "Hello"
TypeError: 'set' object does not support item assignment
arun@Dell-Inspiron-14-5430:~$
```

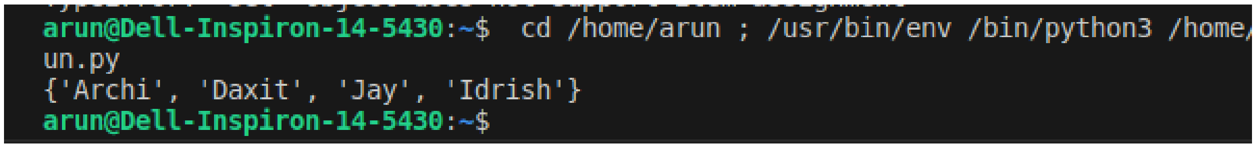
D. people = {"Jay", "Idrish", "Archi"}

```python
people.add("Daxit")
print(people)
```

OUTPUT:{"Jay", "Idrish", "Archi", "Daxit"} → IN ANY ORDER



```python
people = {"Jay", "Idrish", "Archi"}
people.add("Daxit")
print(people)
```



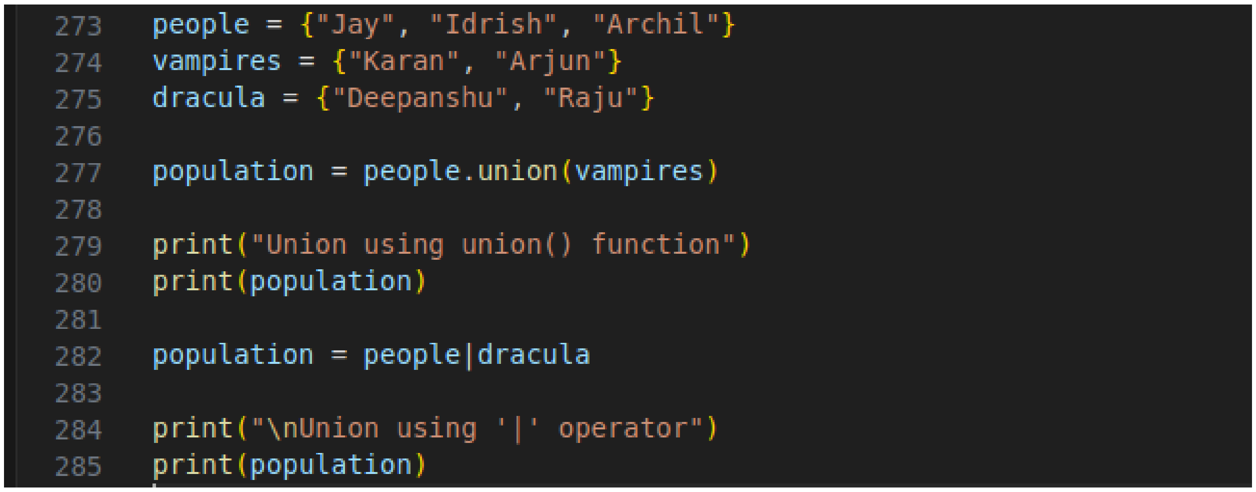
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/
un.py
{'Archi', 'Daxit', 'Jay', 'Idrish'}
arun@Dell-Inspiron-14-5430:~$
```

E. Adding two two sets using **UNION**

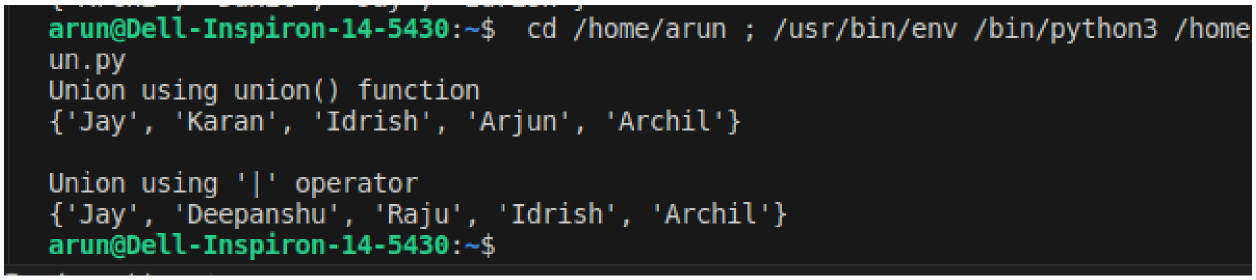
```python
people = {"Jay", "Idrish", "Archil"}
vampires = {"Karan", "Arjun"}
dracula = {"Deepanshu", "Raju"}

population = people.union(vampires)

print("Union using union() function")
print(population)

population = people|dracula

print("\nUnion using '|' operator")
print(population)
```

OUTPUT:

```
Union using union() function
{'Karan', 'Idrish', 'Jay', 'Arjun', 'Archil'}

Union using '|' operator
{'Deepanshu', 'Idrish', 'Jay', 'Raju', 'Archil'}
```



```
people = {"Jay", "Idrish", "Archil"}
vampires = {"Karan", "Arjun"}
dracula = {"Deepanshu", "Raju"}

population = people.union(vampires)

print("Union using union() function")
print(population)

population = people|dracula

print("\nUnion using '|' operator")
print(population)
```



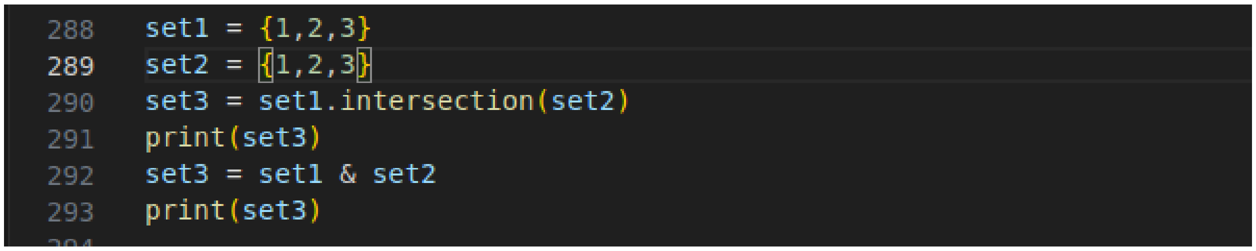
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home
un.py
Union using union() function
{'Jay', 'Karan', 'Idrish', 'Arjun', 'Archil'}

Union using '|' operator
{'Jay', 'Deepanshu', 'Raju', 'Idrish', 'Archil'}
arun@Dell-Inspiron-14-5430:~$
```

F. Selecting Common Elements

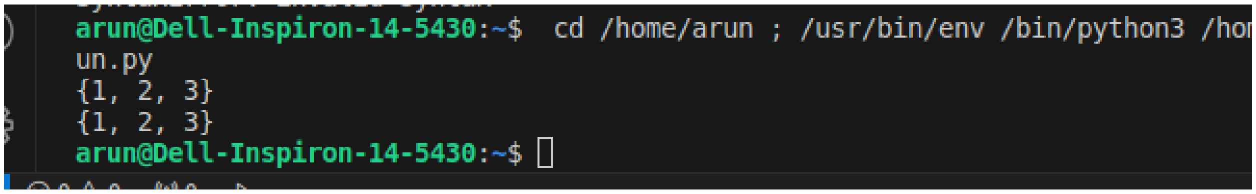
```
set1 = set()
set2 = set()
set3 = set1.intersection(set2)
print(set3)
set3 = set1 & set2
print(set3)
```

OUTPUT:

```
{1,2,3}
{1,2,3}
```



```
set1 = {1,2,3}
set2 = {1,2,3}
set3 = set1.intersection(set2)
print(set3)
set3 = set1 & set2
print(set3)
```

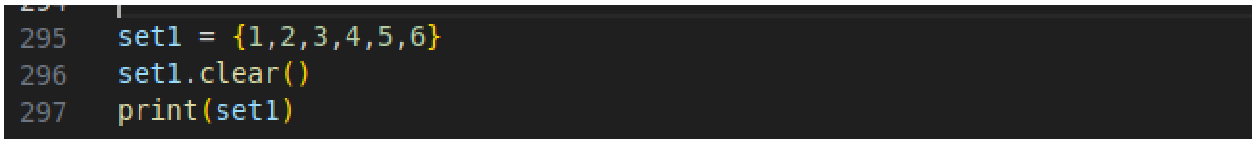


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
{1, 2, 3}
{1, 2, 3}
arun@Dell-Inspiron-14-5430:~$ 
```

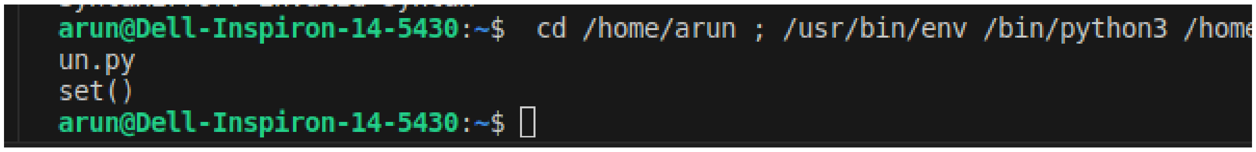
G. Clearing a set

```
set1 = {1,2,3,4,5,6}
set1.clear()
print(set1)
```

OUTPUT: set()



```
set1 = {1,2,3,4,5,6}
set1.clear()
print(set1)
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home
un.py
set()
arun@Dell-Inspiron-14-5430:~$ 
```

22. If-Else-Elif Statement :

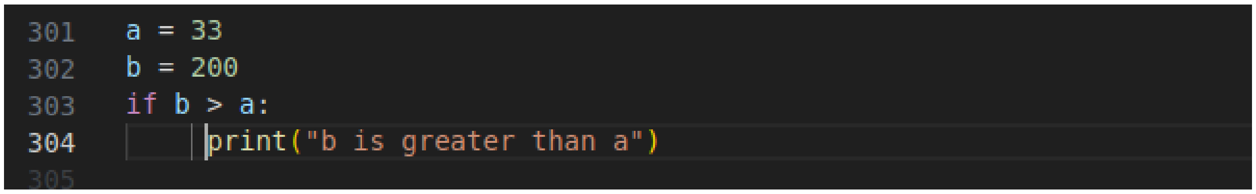
Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

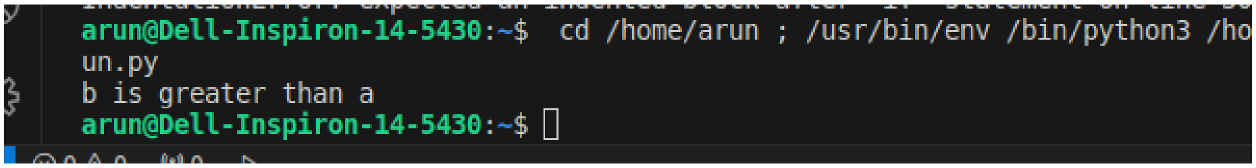
a. `a = 33`

```
b = 200
if b > a:
     print("b is greater than a")
```

OUTPUT: b is greater than a



```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

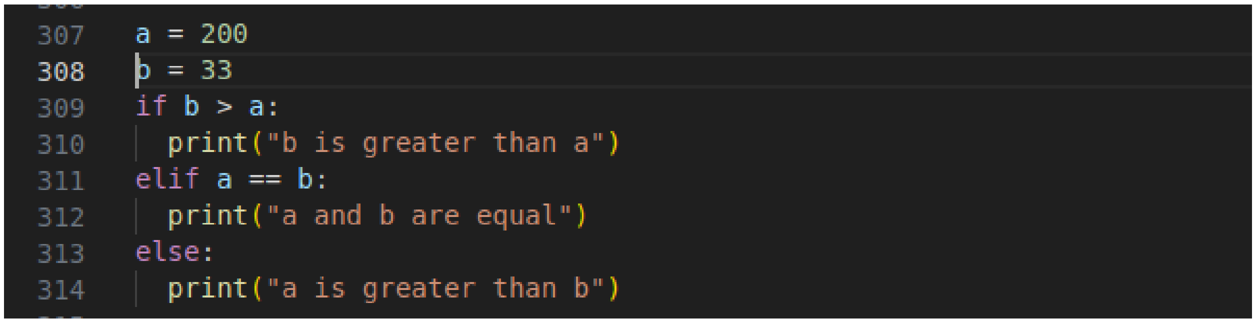


```
IndentationError: expected an indented block after 'if' statement on line 30
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /ho
un.py
b is greater than a
arun@Dell-Inspiron-14-5430:~$ 
```

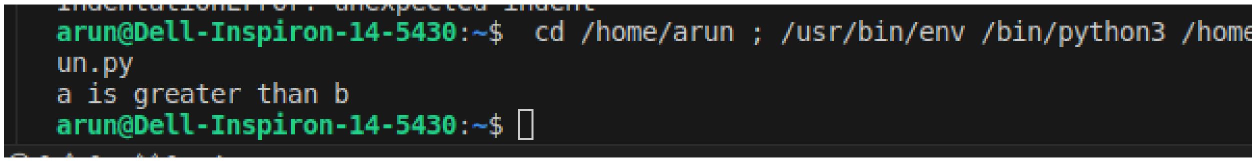
b.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

OUTPUT: a is greater than b



```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```



```
IndentationError: unexpected indent
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home
un.py
a is greater than b
arun@Dell-Inspiron-14-5430:~$ 
```

C. AND

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")

OUTPUT: Both conditions are True
```

D. OR

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")

OUTPUT:At least one of the conditions is True
```

E. NOT

```
a = 33
b = 200
if not a > b:
  print("a is NOT greater than b")

OUTPUT: a is NOT greater than b
```

F. Pass → if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

23. Truthly Vs Falsely: In Python, individual values can evaluate to either True or False. They do not necessarily have to be part of a larger expression to evaluate to a truth value because they already have one that has been determined by the rules of the Python language.
The basic rules are:

- Values that evaluate to False are considered Falsy. (0, None, Empty)
- Values that evaluate to True are considered Truthy.

24. Ternary Operator : The ternary operator in Python is simply a shorter way of writing an if and if…else statement

a.
```
a, b = 10, 20
min = a if a < b else b
print(min)
OUTPUT: 10
```
b.
```
a, b = 10, 20
print ("Both a and b are equal" if a == b else "a is greater than b"
  if a > b else "b is greater than a")

OUTPUT:b is greater than a
```

25. Short Circuiting Techniques: mean the stoppage of execution of boolean operation if the truth value of expression has been determined already.

–> An expression containing **and or** stops execution when the truth value of expression has been achieved. Evaluation takes place from left to right.

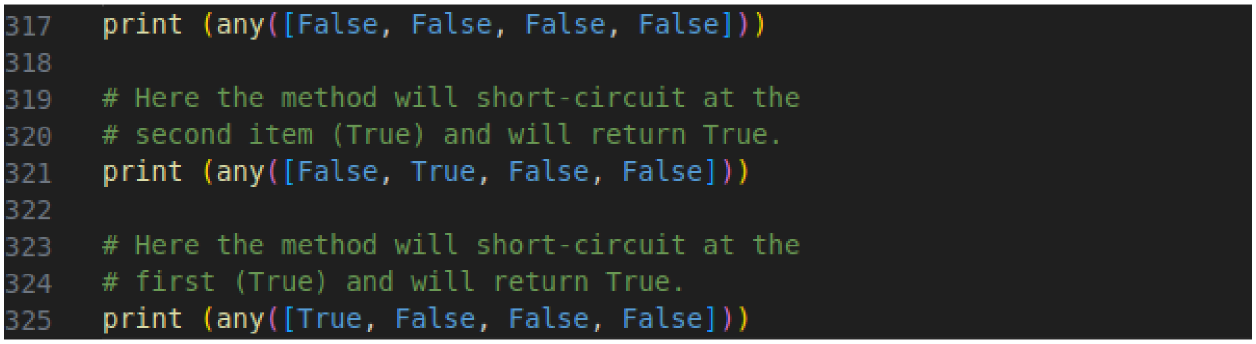
a. **X or Y** –> Y is executed only if X is false else if X is true, X is result.
b. **X and Y** –> Y is executed only if X is true, else if X is false, X is result.
c. **Not X** –> not has lower priority than non-booleans.

26. ANY expression :

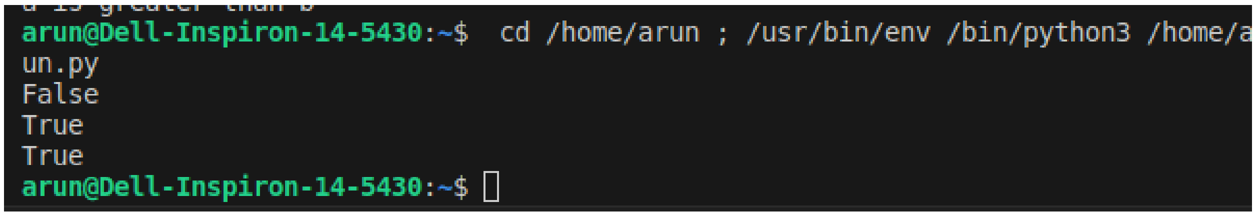
```
 # Since all are false, false is returned
print (any([False, False, False, False]))

# Here the method will short-circuit at the
# second item (True) and will return True.
print (any([False, True, False, False]))

# Here the method will short-circuit at the
# first (True) and will return True.
print (any([True, False, False, False]))
```

Output:

```
False
True
True
```



```
print (any([False, False, False, False]))

# Here the method will short-circuit at the
# second item (True) and will return True.
print (any([False, True, False, False]))

# Here the method will short-circuit at the
# first (True) and will return True.
print (any([True, False, False, False]))
```



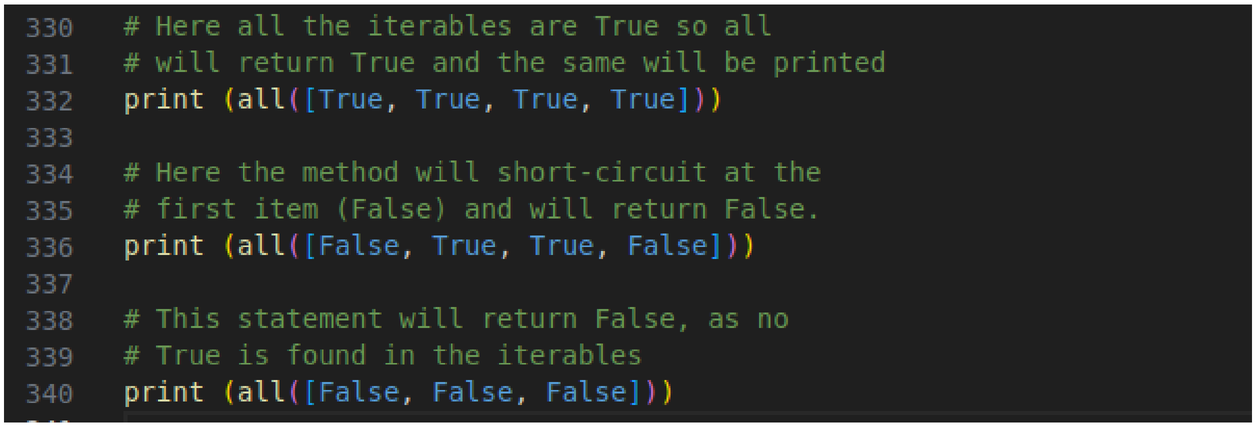
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/a
un.py
False
True
True
arun@Dell-Inspiron-14-5430:~$ 
```

27. All Expression :

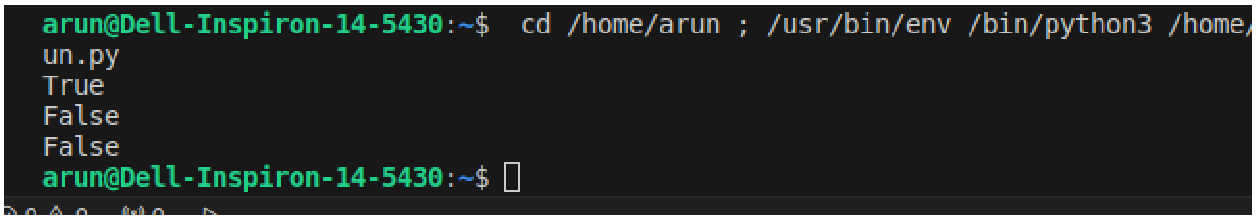
```
# Here all the iterables are True so all
# will return True and the same will be printed
print (all([True, True, True, True]))

# Here the method will short-circuit at the
# first item (False) and will return False.
print (all([False, True, True, False]))

# This statement will return False, as no
# True is found in the iterables
print (all([False, False, False]))
```

Output :

```
True
False
False
```

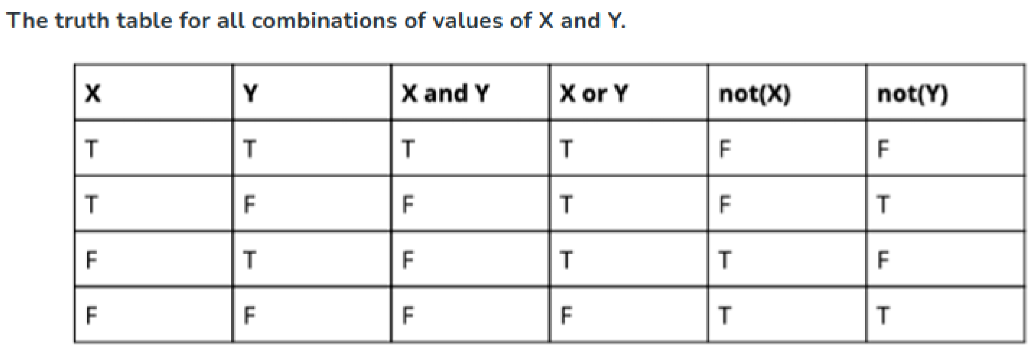


```python
# Here all the iterables are True so all
# will return True and the same will be printed
print (all([True, True, True, True]))

# Here the method will short-circuit at the
# first item (False) and will return False.
print (all([False, True, True, False]))

# This statement will return False, as no
# True is found in the iterables
print (all([False, False, False]))
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/
un.py
True
False
False
arun@Dell-Inspiron-14-5430:~$ 
```

28. Logical Operators:



The truth table for all combinations of values of X and Y.

X	Y	X and Y	X or Y	not(X)	not(Y)
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Truth Table

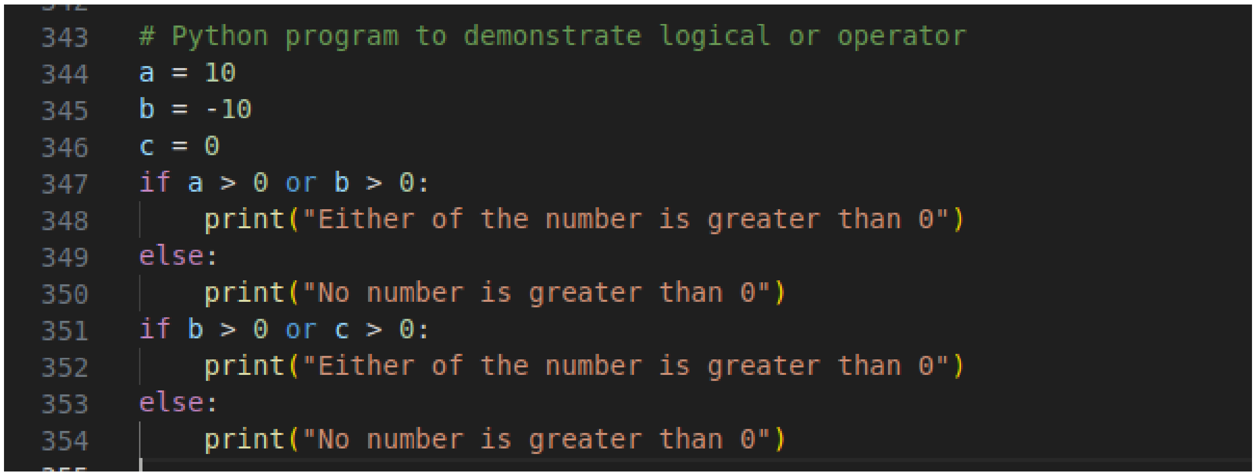
a.
```python
# Python program to demonstrate logical or operator
a = 10
b = -10
c = 0
if a > 0 or b > 0:
    print("Either of the number is greater than 0")
else:
    print("No number is greater than 0")
```

```
if b > 0 or c > 0:
    print("Either of the number is greater than 0")
else:
    print("No number is greater than 0")
```

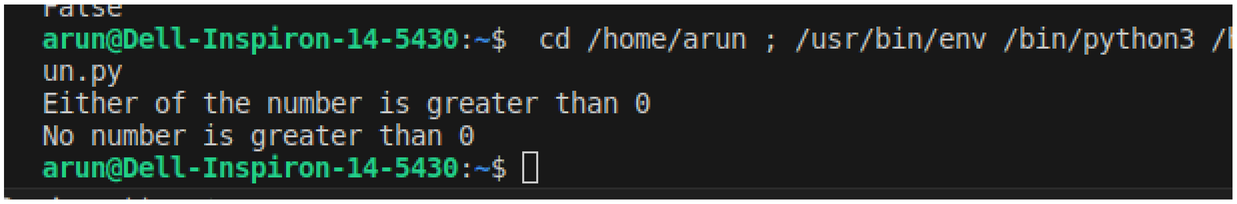
Output

Either of the number is greater than 0

No number is greater than 0



```
# Python program to demonstrate logical or operator
a = 10
b = -10
c = 0
if a > 0 or b > 0:
    print("Either of the number is greater than 0")
else:
    print("No number is greater than 0")
if b > 0 or c > 0:
    print("Either of the number is greater than 0")
else:
    print("No number is greater than 0")
```



```
False
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /
un.py
Either of the number is greater than 0
No number is greater than 0
arun@Dell-Inspiron-14-5430:~$ 
```

29. Is vs == Operator

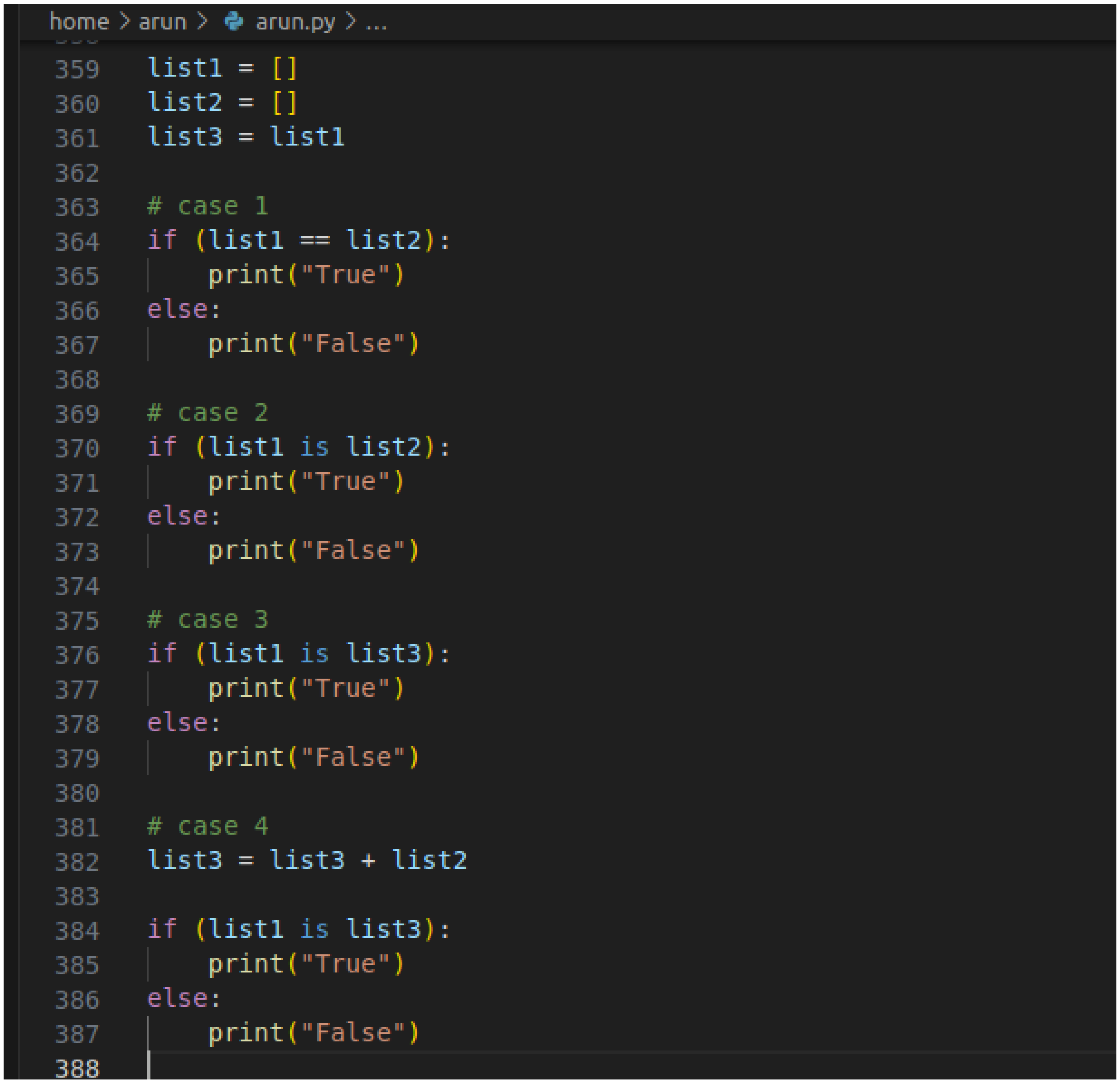
Parameters	is Operator	== Operator
Name	The 'is' is known as the identity operator.	The '==' is known as the equality operator.
Uses	When the variables on either side of an operator point at the exact same object, the **is** operator's evaluation is true. Otherwise, it will evaluate as False.	When the variables on either side have the exact same value, the == operator evaluation is true. Otherwise, it will evaluate as False.

a. list1 = []

```
list2 = []
list3 = list1

# case 1
if (list1 == list2):
    print("True")
else:
    print("False")

# case 2
if (list1 is list2):
    print("True")
else:
    print("False")

# case 3
if (list1 is list3):
    print("True")
else:
    print("False")

# case 4
list3 = list3 + list2

if (list1 is list3):
    print("True")
else:
    print("False")
```

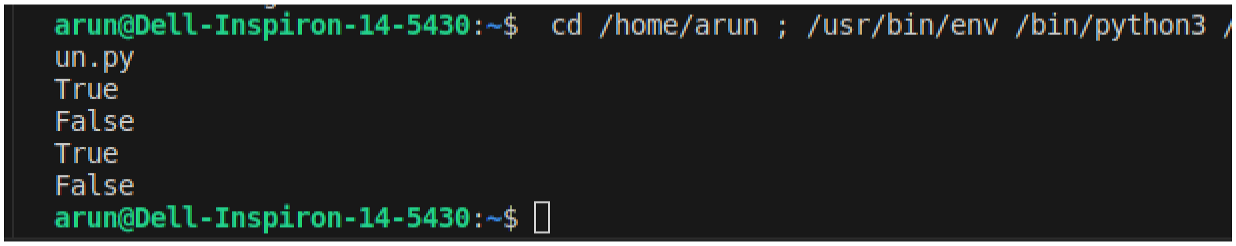
OUTPUT:

```
True

False

True

False
```



home > arun > arun.py > ...

```python
list1 = []
list2 = []
list3 = list1

# case 1
if (list1 == list2):
    print("True")
else:
    print("False")

# case 2
if (list1 is list2):
    print("True")
else:
    print("False")

# case 3
if (list1 is list3):
    print("True")
else:
    print("False")

# case 4
list3 = list3 + list2

if (list1 is list3):
    print("True")
else:
    print("False")
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /
un.py
True
False
True
False
arun@Dell-Inspiron-14-5430:~$ 
```

30. Loops in Python

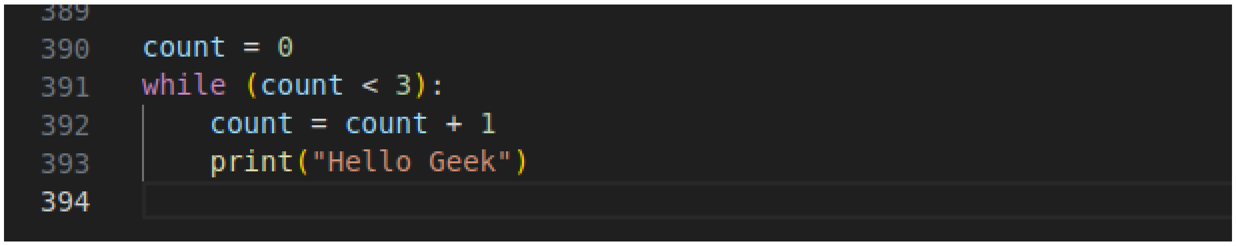
1. **WHILE Loop:** is used to execute a block of statements repeatedly until a given condition is satisfied.

Syntax:

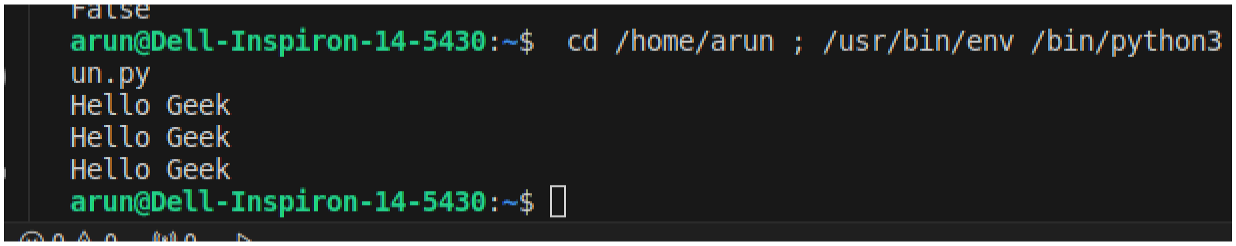
```
while expression:
        statement(s)
```

a.
```
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
```

```
OUTPUT:
Hello Geek
Hello Geek
Hello Geek
```



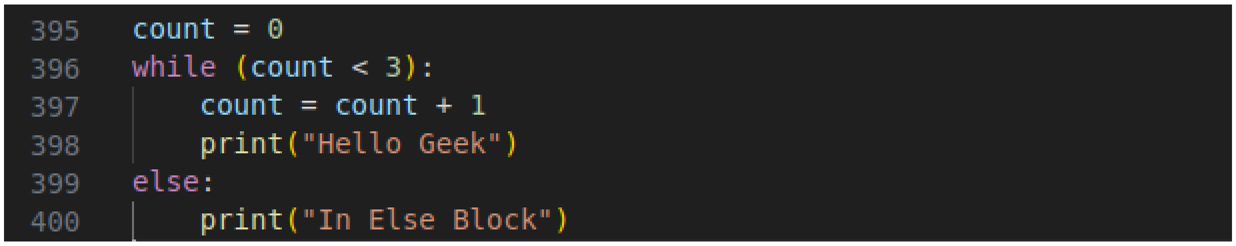
```
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
```



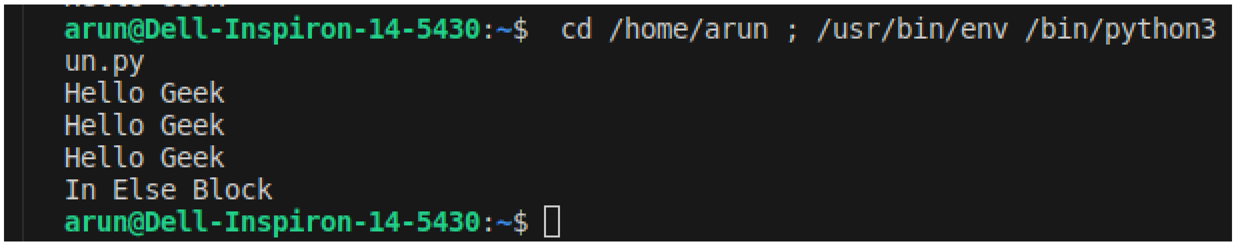
```
False
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3
un.py
Hello Geek
Hello Geek
Hello Geek
arun@Dell-Inspiron-14-5430:~$ 
```

b.
```
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
else:
    print("In Else Block")
```

```
OUTPUT:
Hello Geek
Hello Geek
Hello Geek
In Else Block
```



```
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
else:
    print("In Else Block")
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3
un.py
Hello Geek
Hello Geek
Hello Geek
In Else Block
arun@Dell-Inspiron-14-5430:~$
```

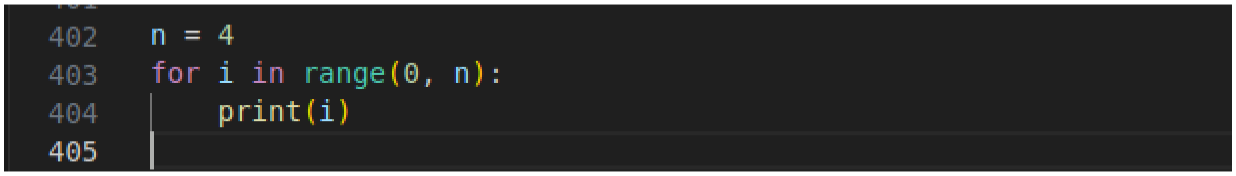
2. **FOR Loop:** are used for sequential traversal.

Syntax:

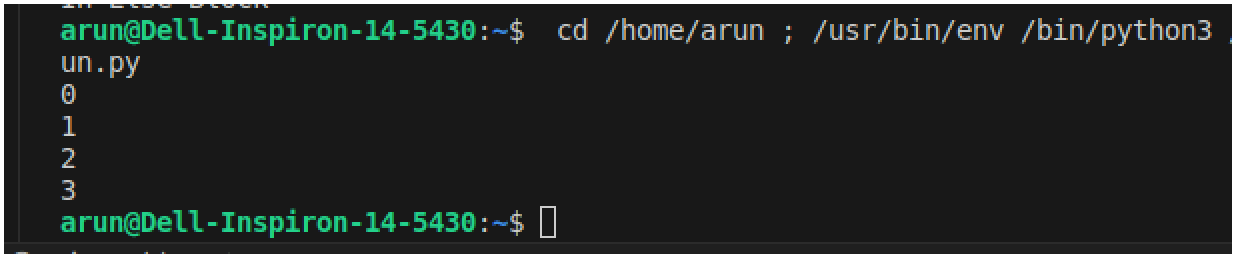
```
for iterator_var in sequence:

    statements(s)
```

a.
```
n = 4
for i in range(0, n):
    print(i)
```

```
OUTPUT:
0
1
2
3
```



```
n = 4
for i in range(0, n):
    print(i)
```

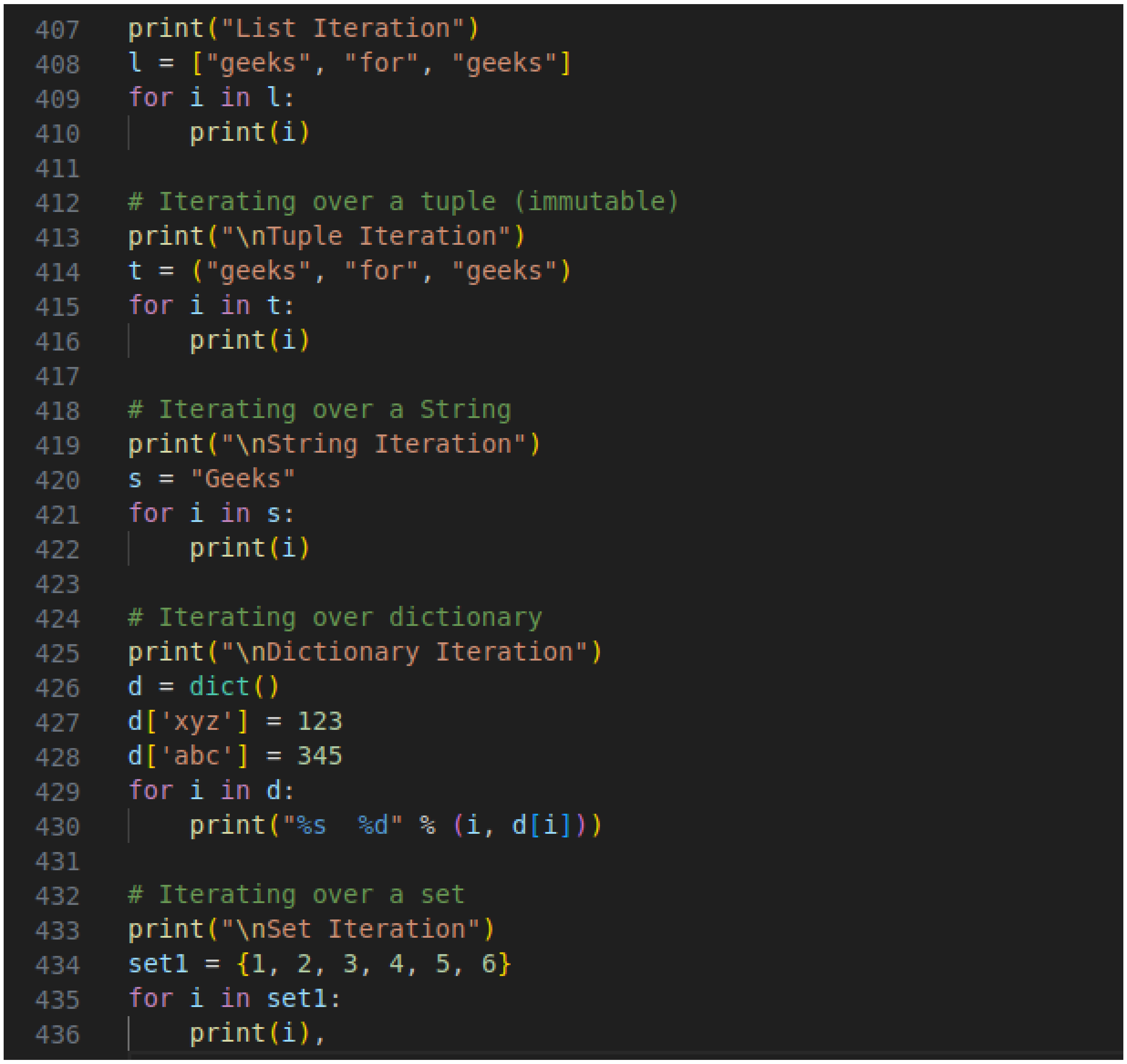


```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 
un.py
0
1
2
3
arun@Dell-Inspiron-14-5430:~$ 
```

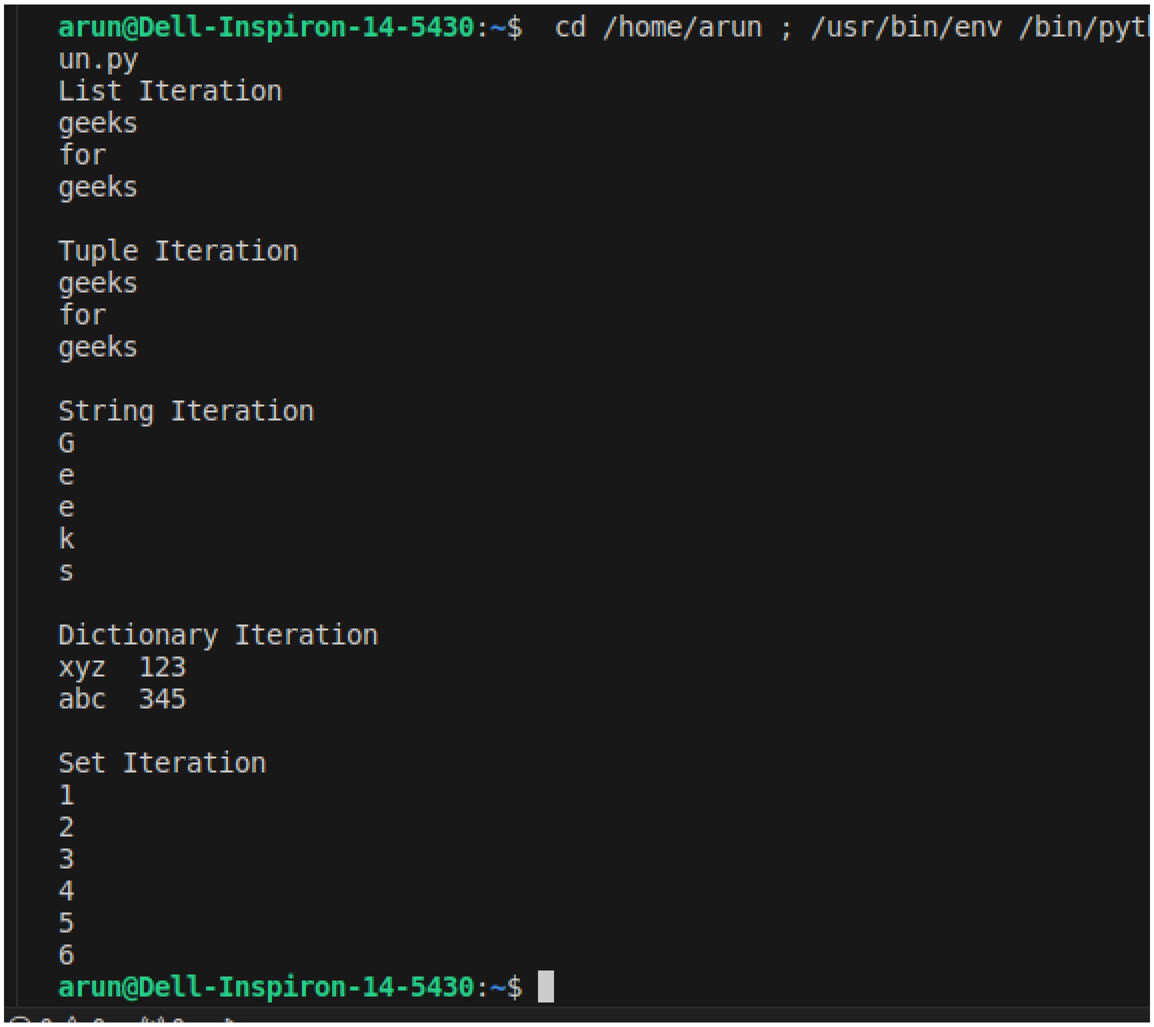
```
b.print("List Iteration")

     l = ["geeks", "for", "geeks"]
     for i in l:
         print(i)

     # Iterating over a tuple (immutable)
     print("\nTuple Iteration")
     t = ("geeks", "for", "geeks")
     for i in t:
         print(i)

     # Iterating over a String
     print("\nString Iteration")
     s = "Geeks"
     for i in s:
         print(i)

     # Iterating over dictionary
     print("\nDictionary Iteration")
     d = dict()
     d['xyz'] = 123
     d['abc'] = 345
     for i in d:
         print("%s  %d" % (i, d[i]))

     # Iterating over a set
     print("\nSet Iteration")
     set1 = {1, 2, 3, 4, 5, 6}
     for i in set1:
         print(i),
```

Output

```
List Iteration
geeks
for
geeks

Tuple Iteration
geeks
for
geeks

String Iteration
G
e
e
k
s

Dictionary Iteration
xyz  123
abc  345

Set Iteration
1
2
3
4
5
6
```



```python
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
    print(i)

# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("geeks", "for", "geeks")
for i in t:
    print(i)

# Iterating over a String
print("\nString Iteration")
s = "Geeks"
for i in s:
    print(i)

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d:
    print("%s  %d" % (i, d[i]))

# Iterating over a set
print("\nSet Iteration")
set1 = {1, 2, 3, 4, 5, 6}
for i in set1:
    print(i),
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/pyt
un.py
List Iteration
geeks
for
geeks

Tuple Iteration
geeks
for
geeks

String Iteration
G
e
e
k
s

Dictionary Iteration
xyz  123
abc  345

Set Iteration
1
2
3
4
5
6
arun@Dell-Inspiron-14-5430:~$
```

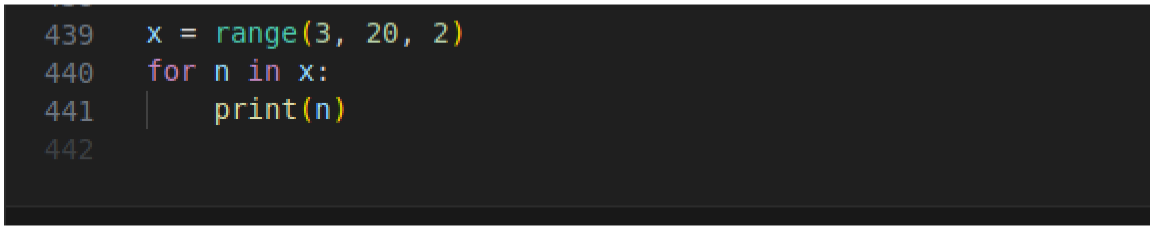
31. Python Iterators: An iterator is an object that contains a countable number of values.

32. Range() : The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

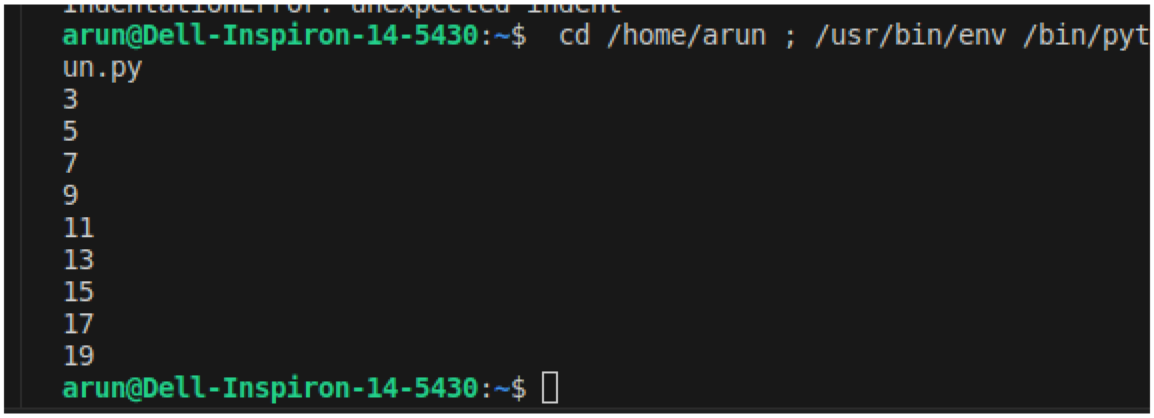
Syntax: **range(*start, stop, step*)**

```
a. x = range(3, 20, 2)
   for n in x:
     print(n)

   OUTPUT:
   3
   5
```

```
7
9
11
13
15
17
19
```



```python
x = range(3, 20, 2)
for n in x:
    print(n)
```



```
IndentationError: unexpected indent
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/pyt
un.py
3
5
7
9
11
13
15
17
19
arun@Dell-Inspiron-14-5430:~$ 
```

33. Enumerate(): The enumerate () method adds a counter to an iterable and returns it in the form of an enumerating object.

Syntax: *enumerate(iterable, start=0)*

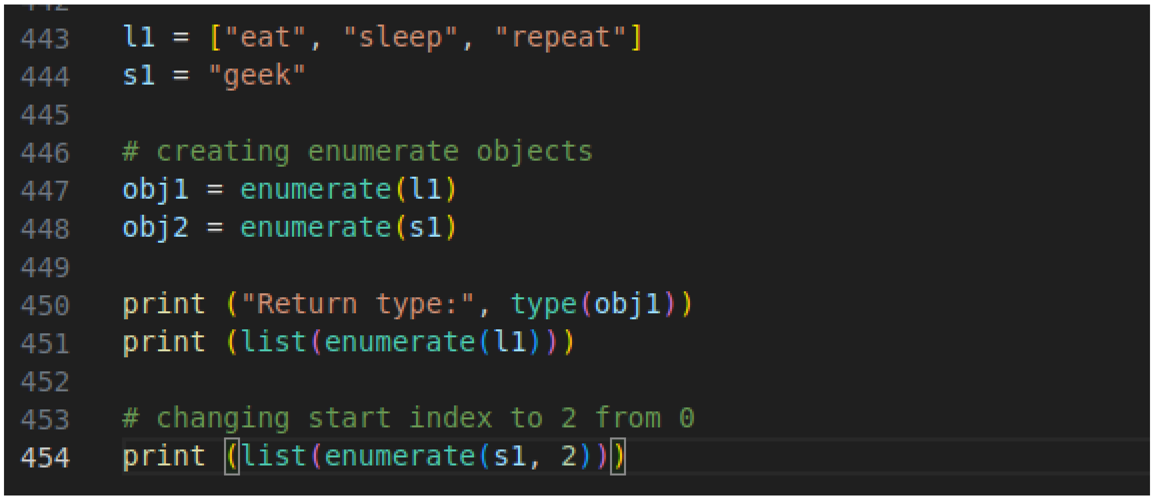
a.
```python
l1 = ["eat", "sleep", "repeat"]
s1 = "geek"

# creating enumerate objects
obj1 = enumerate(l1)
obj2 = enumerate(s1)

print ("Return type:", type(obj1))
print (list(enumerate(l1)))

# changing start index to 2 from 0
print (list(enumerate(s1, 2)))
```

Output:

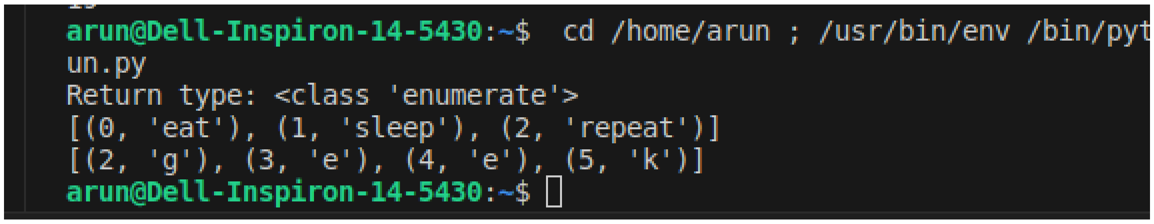
Return type: <class 'enumerate'>

[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]

[(2, 'g'), (3, 'e'), (4, 'e'), (5, 'k')]



```python
l1 = ["eat", "sleep", "repeat"]
s1 = "geek"

# creating enumerate objects
obj1 = enumerate(l1)
obj2 = enumerate(s1)

print ("Return type:", type(obj1))
print (list(enumerate(l1)))

# changing start index to 2 from 0
print (list(enumerate(s1, 2)))
```



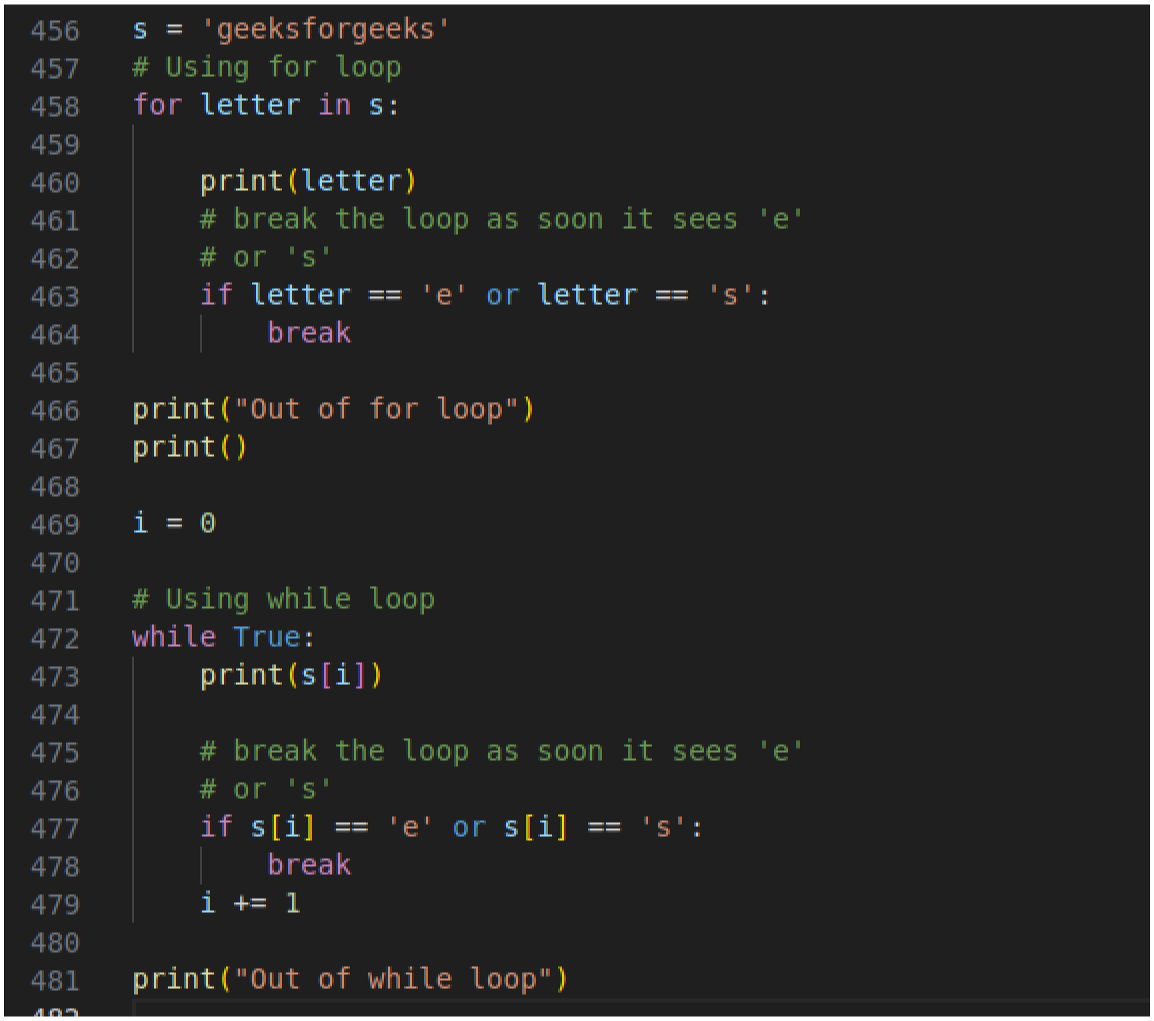
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/pyt
un.py
Return type: <class 'enumerate'>
[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]
[(2, 'g'), (3, 'e'), (4, 'e'), (5, 'k')]
arun@Dell-Inspiron-14-5430:~$ 
```

34. **Break Statement:** is used to terminate the loop or statement in which it is present.

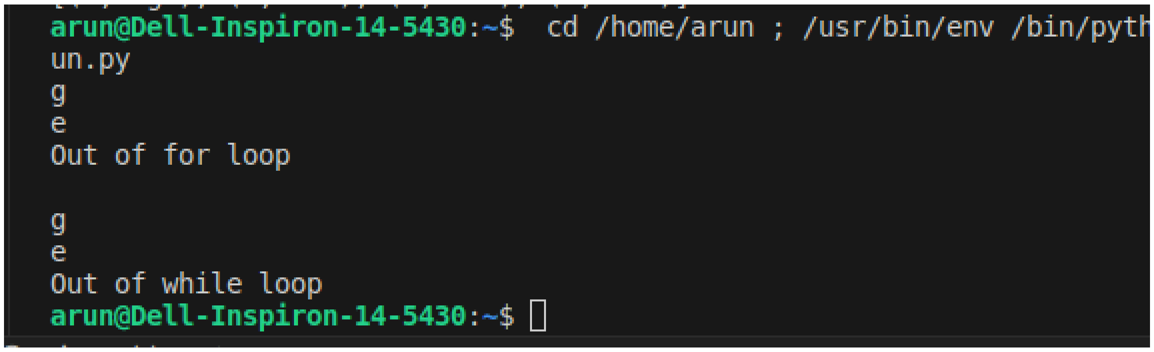
```python
s = 'geeksforgeeks'
# Using for loop
for letter in s:

    print(letter)
    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break

print("Out of for loop")
print()

i = 0

# Using while loop
while True:
    print(s[i])

    # break the loop as soon it sees 'e'
    # or 's'
    if s[i] == 'e' or s[i] == 's':
        break
    i += 1

print("Out of while loop")
```

OUTPUT:

```
g

e

Out of for loop


g

e

Out of while loop
```



```python
s = 'geeksforgeeks'
# Using for loop
for letter in s:

    print(letter)
    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break

print("Out of for loop")
print()

i = 0

# Using while loop
while True:
    print(s[i])

    # break the loop as soon it sees 'e'
    # or 's'
    if s[i] == 'e' or s[i] == 's':
        break
    i += 1

print("Out of while loop")
```



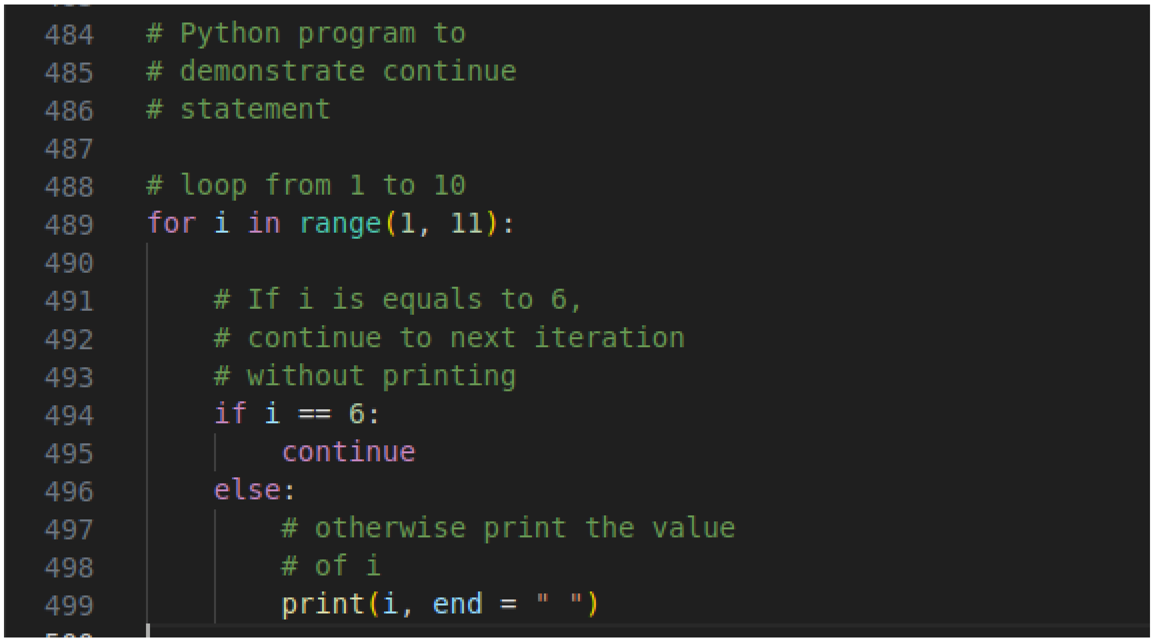
```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/pyth
un.py
g
e
Out of for loop

g
e
Out of while loop
arun@Dell-Inspiron-14-5430:~$
```

35. Continue Statememt: continue statement is opposite to that of the break statement, instead of terminating the loop, it forces to execute the next iteration of the loop.

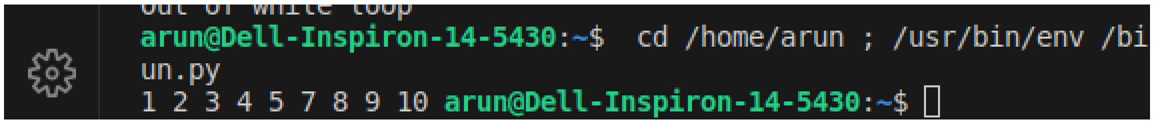
```python
# Python program to
# demonstrate continue
# statement
# loop from 1 to 10
for i in range(1, 11):

    # If i is equals to 6,
    # continue to next iteration
    # without printing
    if i == 6:
        continue
    else:
        # otherwise print the value
        # of i
        print(i, end = " ")
```

Output:

```
1 2 3 4 5 7 8 9 10
```



```python
# Python program to
# demonstrate continue
# statement

# loop from 1 to 10
for i in range(1, 11):

    # If i is equals to 6,
    # continue to next iteration
    # without printing
    if i == 6:
        continue
    else:
        # otherwise print the value
        # of i
        print(i, end = " ")
```



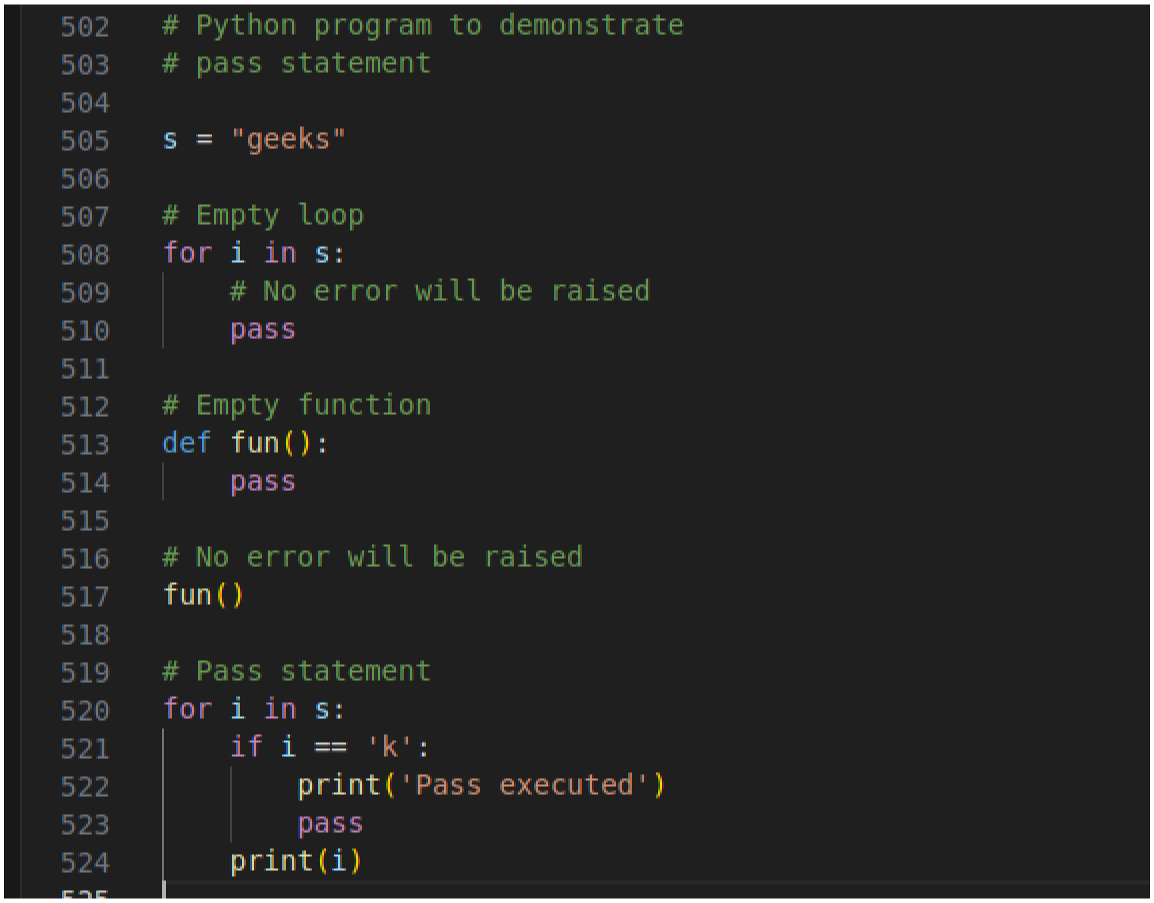
```
out of while loop
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bi
un.py
1 2 3 4 5 7 8 9 10 arun@Dell-Inspiron-14-5430:~$ 
```

36. Pass Statement: is used when a statement is required syntactically but you do not want any command or code to execute.

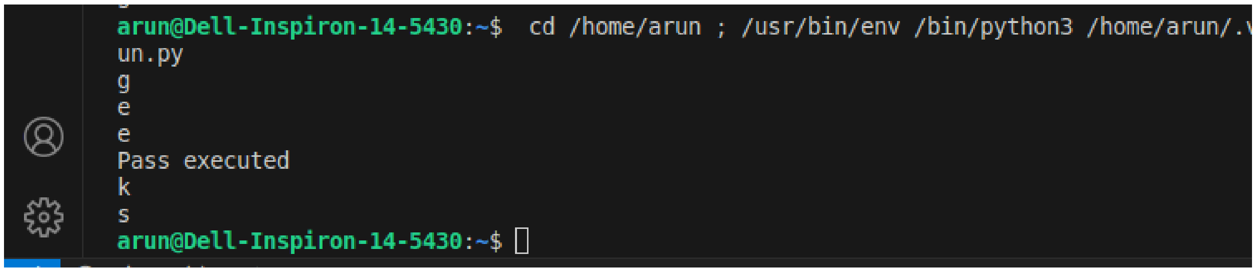
```python
# Python program to demonstrate
# pass statement

s = "geeks"

# Empty loop
for i in s:
    # No error will be raised
    pass

# Empty function
def fun():
    pass

# No error will be raised
fun()

# Pass statement
for i in s:
    if i == 'k':
        print('Pass executed')
        pass
    print(i)
```

Output:

```
g

e

e

Pass executed

k

s
```



```python
# Python program to demonstrate
# pass statement

s = "geeks"

# Empty loop
for i in s:
    # No error will be raised
    pass

# Empty function
def fun():
    pass

# No error will be raised
fun()

# Pass statement
for i in s:
    if i == 'k':
        print('Pass executed')
        pass
    print(i)
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.v
un.py
g
e
e
Pass executed
k
s
arun@Dell-Inspiron-14-5430:~$ 
```

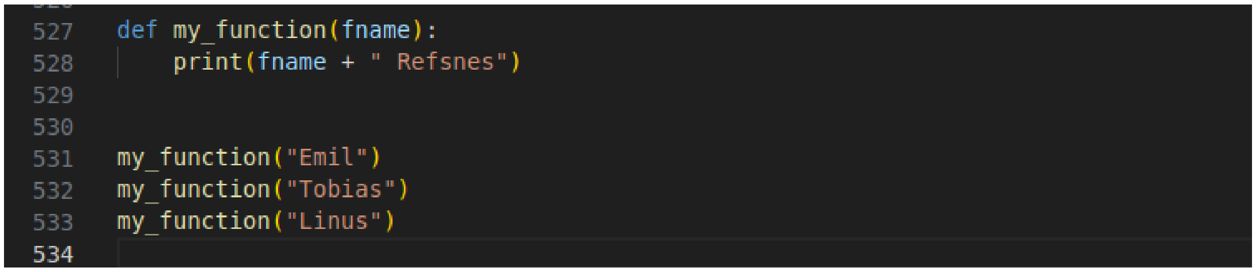
37. Functions : A function is a block of code which only runs when it is called.

Parameters: You can pass data, known as parameters, into a function.

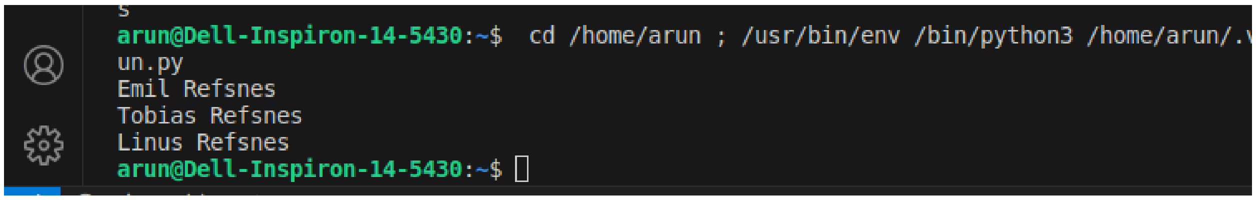
a.
```python
def my_function():
  print("Hello from a function")
```

Arguments: Information can be passed into functions as arguments.

a.
```
def my_function(fname):
    print(fname + " Refsnes")


my_function("Emil")
my_function("Tobias")
my_function("Linus")
```



```
def my_function(fname):
    print(fname + " Refsnes")


my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

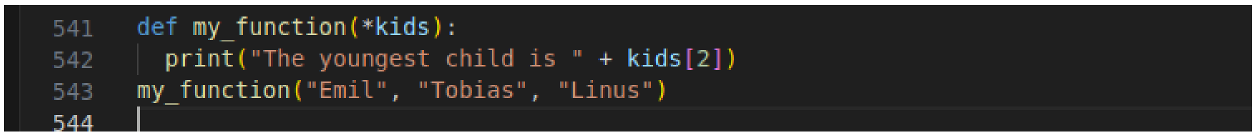


```
s
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.v
un.py
Emil Refsnes
Tobias Refsnes
Linus Refsnes
arun@Dell-Inspiron-14-5430:~$
```

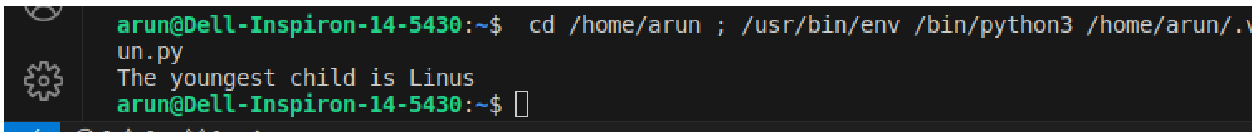
38. Arbitrary Arguments:(*args) If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

a.
```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

OUTPUT: `The youngest child is Linus`



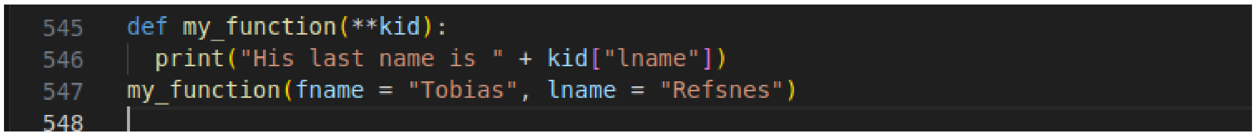
```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

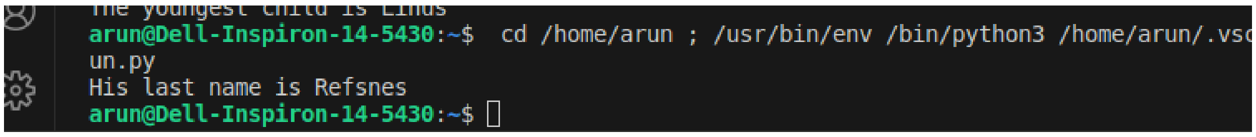

```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.
un.py
The youngest child is Linus
arun@Dell-Inspiron-14-5430:~$
```

39. **kwargs(Keyword Arguments): If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

a.
```
def my_function(**kid):
  print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

OUTPUT: His last name is Refsnes


```
def my_function(**kid):
  print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```


```
The youngest child is Linus
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vsc
un.py
His last name is Refsnes
arun@Dell-Inspiron-14-5430:~$
```

40. Default Parameter Value:

a. def my_function(country = "Norway"):

print("I am from " + country)

my_function("Sweden")

my_function("India")

my_function()

my_function("Brazil")

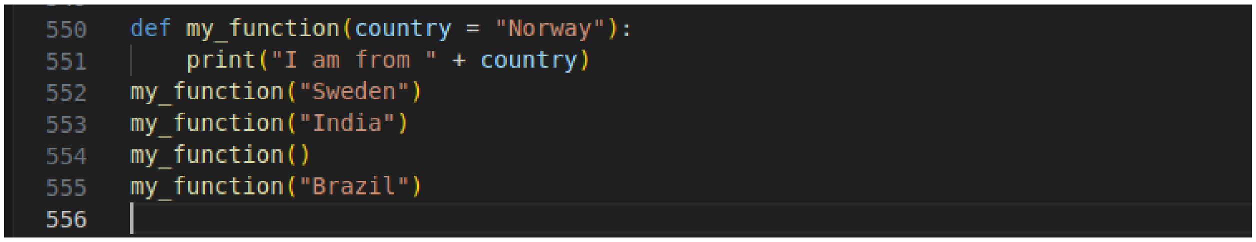
OUTPUT:

I am from Sweden

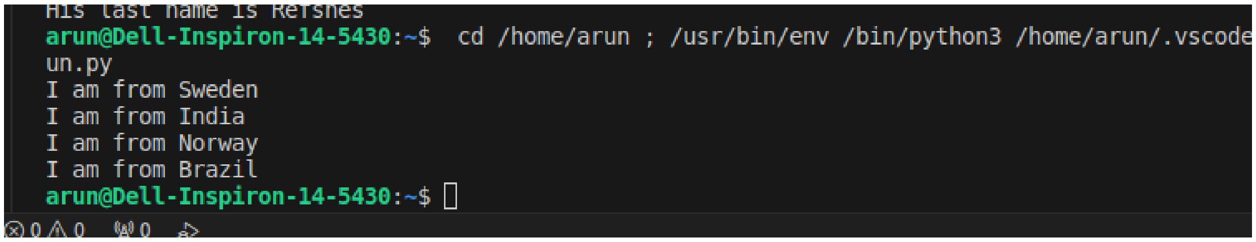
I am from India

I am from Norway

I am from Brazil



```
def my_function(country = "Norway"):
    print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```



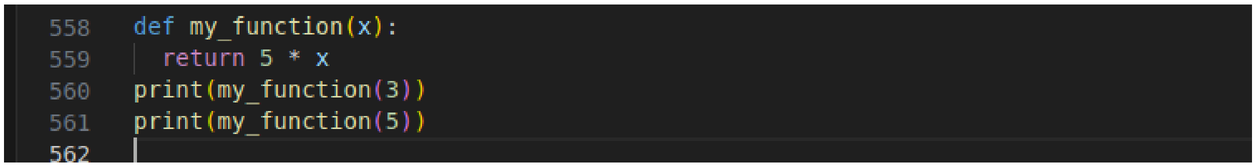
```
His last name is Refsnes
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscode
un.py
I am from Sweden
I am from India
I am from Norway
I am from Brazil
arun@Dell-Inspiron-14-5430:~$ 
```

41. Return Values → To let a function return a value, use the return statement.

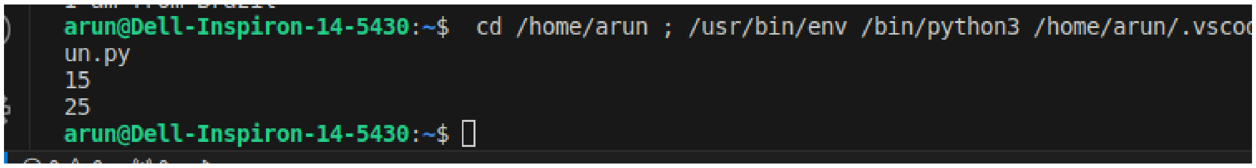
a.
```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

OUTPUT:

```
15

25

45
```



```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
```



```
arun@Dell-Inspiron-14-5430:~$  cd /home/arun ; /usr/bin/env /bin/python3 /home/arun/.vscod
un.py
15
25
arun@Dell-Inspiron-14-5430:~$ 
```

42. **DocStrings:** It's specified in source code that is used, like a comment, to document a specific segment of code.

a. **Declaring DocString:** The docstrings are declared using "'triple single quotes"' or """ triple double quotes """ just below the class, method, or function declaration. All functions should have a docstring.
b. **Accessing Docstrings**: The docstrings can be accessed using the **__doc__** method of the object or using the help function. The below examples demonstrate how to declare and access a docstring.

a.
```
def my_function():
    '''Demonstrates triple double quotes
    docstrings and does nothing really.'''

    return None

print("Using __doc__:")
print(my_function.__doc__)

print("Using help:")
help(my_function)
```

OUTPUT:

Using __doc__:

Demonstrates triple double quotes

    docstrings and does nothing really.

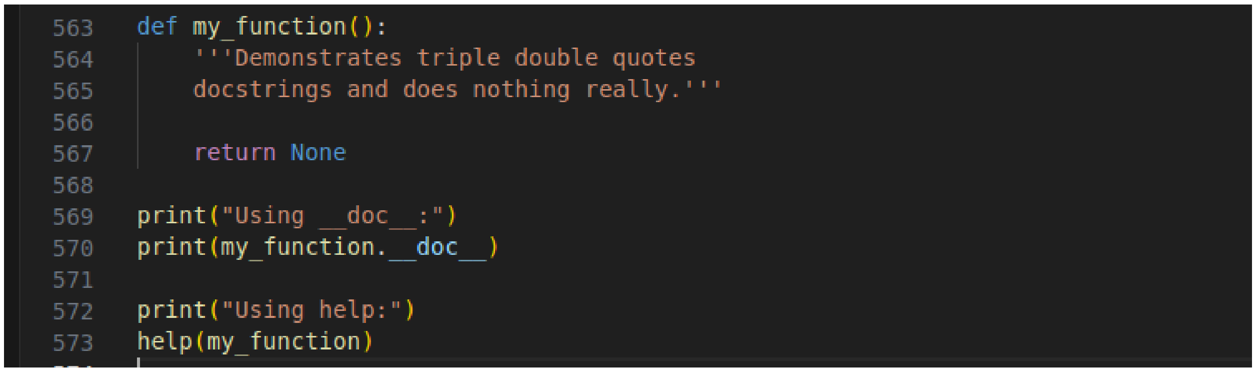
Using help:

Help on function my_function in module __main__:

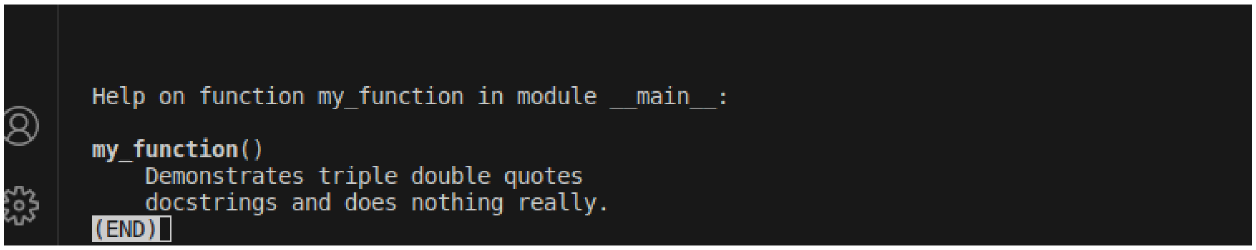
my_function()

    Demonstrates triple double quotes

    docstrings and does nothing really.



```python
def my_function():
    '''Demonstrates triple double quotes
    docstrings and does nothing really.'''

    return None

print("Using __doc__:")
print(my_function.__doc__)

print("Using help:")
help(my_function)
```

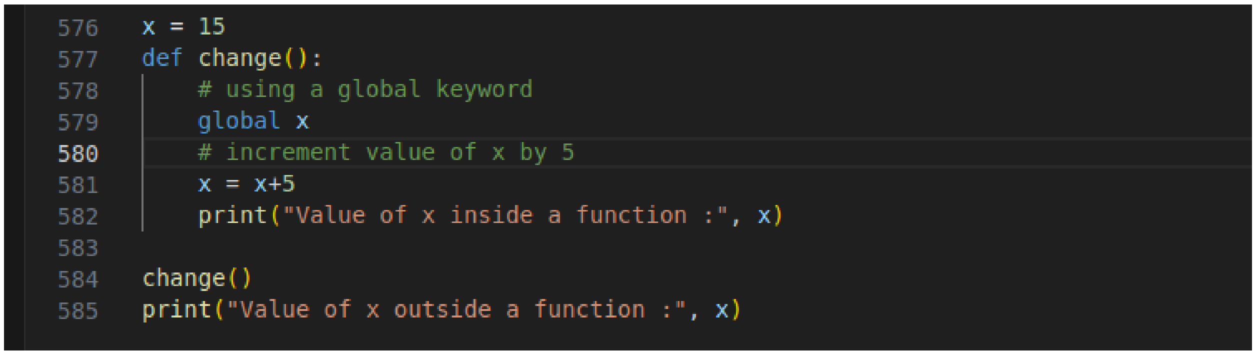


```
Help on function my_function in module __main__:

my_function()
    Demonstrates triple double quotes
    docstrings and does nothing really.
(END)
```

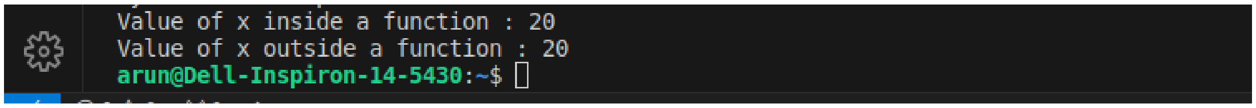
3. Global Keyword: A global keyword is a keyword that allows a user to modify a variable outside the current scope. It is used to create **global variables in Python** from a non-global scope, i.e. inside a function. Global keyword is used inside a function only when we want to do assignments or when we want to change a variable.

Output:

```
Value of x inside a function : 20
Value of x outside a function : 20
```



```python
x = 15
def change():
    # using a global keyword
    global x
    # increment value of x by 5
    x = x+5
    print("Value of x inside a function :", x)

change()
print("Value of x outside a function :", x)
```

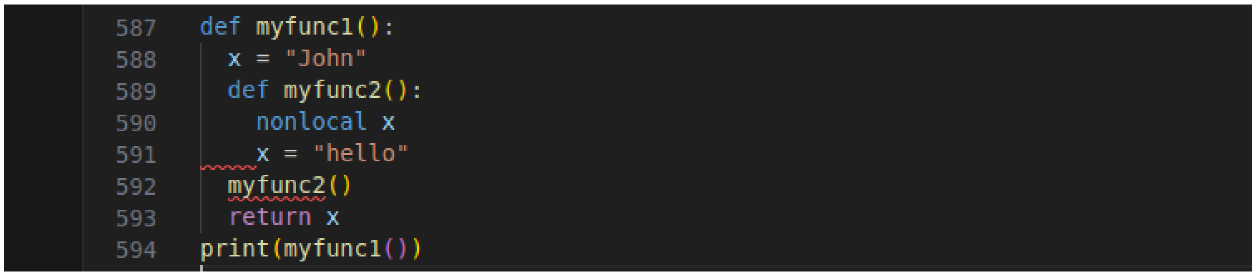


```
Value of x inside a function : 20
Value of x outside a function : 20
arun@Dell-Inspiron-14-5430:~$
```

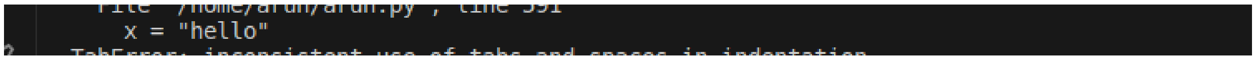
44. NonLocal Keyword: The nonlocal keyword is used to work with variables inside nested functions, where the variable should not belong to the inner function.

a.
```python
def myfunc1():
 x = "John"
 def myfunc2():
  nonlocal x
       x = "hello"
 myfunc2()
 return x
print(myfunc1())
```

OUTPUT: hello



```python
def myfunc1():
  x = "John"
  def myfunc2():
    nonlocal x
    x = "hello"
  myfunc2()
  return x
print(myfunc1())
```



```
  File "/home/arun/arun.py", line 591
    x = "hello"
TabError: inconsistent use of tabs and spaces in indentation
```