

# INTRODUCTION TO PROGRAMMING - II

DEPARTMENT OF MATHEMATICS

MSC - INTEGRATED

RAJ ZALAVADIA

- Machine Learning Engineer  
@ Social Orbit  
- Full Stack Developer  
@ Greycell Labs

Mo : +91 7041645834 / 7600299260

Email : raj.zalavadia@greylabs.com



# **UNIT – 4**

## **CONTENT**

- **Basics of Python**
- **Variables**
- **Data Types**
- **Basic Python Operation**
- **Decision Making**
- **File I/O**



# python



# Python

- Python is a simple, general purpose, high level, and object-oriented programming language.
- Python is an interpreted scripting language also.
- **Guido Van Rossum** is known as the founder of Python programming.
- Python is easy to learn yet powerful and versatile scripting language, which makes it attractive for Application Development.



# Python History & Version

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.

# Python History & Version

- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- ABC programming language is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.



# Python Features

## 1) Easy to Learn and Use

- Python is easy to learn and use.
- It is developer-friendly and high level programming language.

## 2) Expressive Language

- Python language is more expressive means that it is more understandable and readable.



# Python Features

## 3) Interpreted Language

- Python is an interpreted language i.e. interpreter executes the code line by line at a time.
- This makes debugging easy and thus suitable for beginners.



# Python Features

## 4) Cross-platform Language

- Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc.
- So, we can say that Python is a portable language.



# Python Features

## 5) Free and Open Source

- Python language is freely available at official web address.
- The source-code is also available. Therefore it is open source.

## 6) Object-Oriented Language

- Python supports object oriented language and concepts of classes and objects come into existence.



# Python Features

## 7) Extensible

- It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.

## 8) Large Standard Library

- Python has a large and broad library and provides rich set of module and functions for rapid application development.



# Python Features

## 9) GUI Programming Support

- Graphical user interfaces can be developed using Python.

## 10) Integrated

- It can be easily integrated with languages like C, C++, JAVA etc.



# Python Application

## 1) Web Applications

- We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feed parser etc.
- It also provides Frameworks such as Django, Pyramid, Flask etc. to design and develop web based applications.
- Some important developments are: Python Wiki Engines, Pocoo, Python Blog Software etc.



# Python Application

## 2) Desktop GUI Applications

- Python provides Tk GUI library to develop user interface in python based application.
- Some other useful toolkits wxWidgets, Kivy, pyqt that are useable on several platforms.
- The Kivy is popular for writing multitouch applications(Android, web & Desktop application).



# Python Application

## 3) Software Development

- Python is helpful for software development process.
- It works as a support language and can be used for build control and management, testing etc.

## 4) Business Applications

- Python is used to build Business applications like ERP and e-commerce systems.



# Python Application

## 5) Scientific and Numeric

- Python is popular and widely used in scientific and numeric computing.
- Some useful library and package are SciPy, Pandas, IPython etc.
- SciPy is group of packages of engineering, science and mathematics.



# Python Application

## 6) Console Based Application

- We can use Python to develop console based applications.
- For example: **IPython**.

## 7) Audio or Video based Applications

- Python is awesome to perform multiple tasks and can be used to develop multimedia applications.
- Some of real applications are: TimPlayer, cplay etc.



# Python Application

## 8) 3D CAD Applications

- To create CAD application Fandango is a real application which provides full features of CAD.

## 9) Enterprise Applications

- Python can be used to create applications which can be used within an Enterprise or an Organization.
- Some real time applications are: OpenErp etc.



# Python Application

## 10) Applications for Images

- Using Python several application can be developed for image. Applications developed are: image processing, opencv, image & Video manipulation etc.
- There are several such applications which can be developed using Python

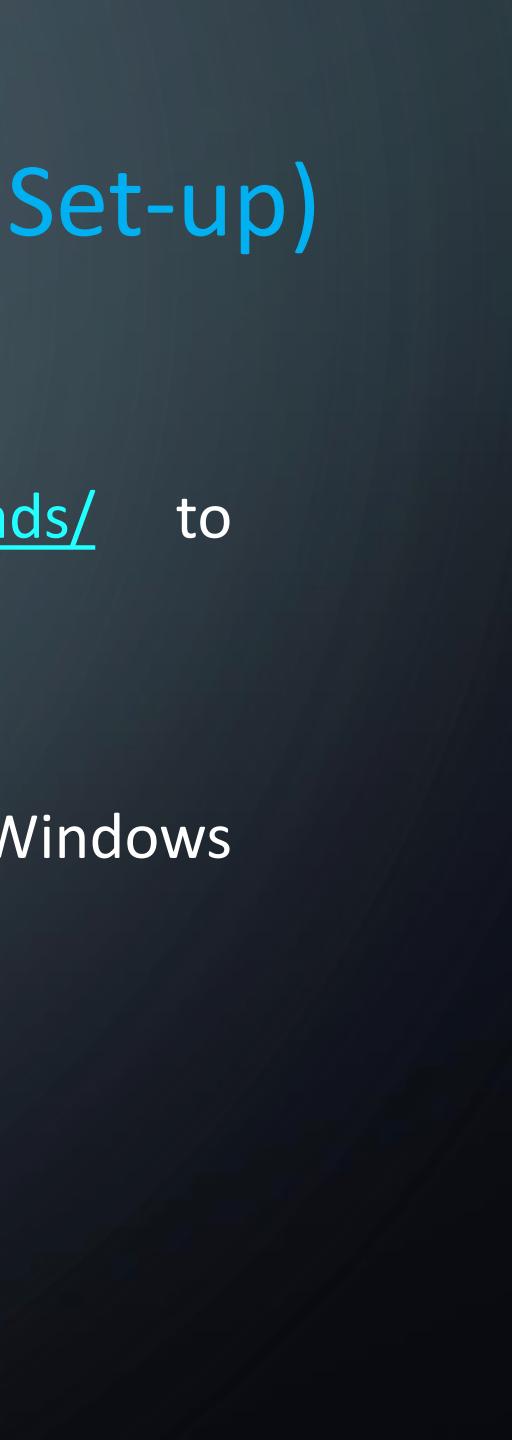


# Python Installation

- You can use as...
  - Terminal (Notepad++, Atom, Sublime, etc.)
    - Making “.py” file and run as script
  - Local Host (Jupyter notebook)
  - IDE(Pycharm, VS Code, Eclipse, etc.)
- 



# How to Install Python (Environment Set-up)

- Visit the link <https://www.python.org/downloads/> to download the latest release of Python.
  - In this process, we will install Python 3.6.7 on our Windows operating system.
- 

Help the PSF raise \$30,000 USD by November 21st!

Participate in our Recurring Giving Campaign

## Looking for a specific release?

Python releases by version number:

Release version	Release date	Click for more
<a href="#">Python 3.6.7</a>	2018-10-20	 <a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.5.6</a>	2018-08-02	 <a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.4.9</a>	2018-08-02	 <a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.7.0</a>	2018-06-27	 <a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.6.6</a>	2018-06-27	 <a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 2.7.15</a>	2018-05-01	 <a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.6.5</a>	2018-03-28	 <a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.4.8</a>	2018-02-05	 <a href="#">Download</a> <a href="#">Release Notes</a>

[View older releases](#)

# Install Python 3.7.0 (32-bit)

Select **Install Now** to install Python with default settings, or choose **Customize** to enable or disable features.



python  
for  
windows

## Install Now

C:\Users\developer\AppData\Local\Programs\Python\Python37-32

Includes IDLE, pip and documentation

Creates shortcuts and file associations

## Customize installation

Choose location and features

Install launcher for all users (recommended)

Add Python 3.7 to PATH

Cancel

## Optional Features

Documentation

Installs the Python documentation file.

pip

Installs pip, which can download and install other Python packages.

tcl/tk and IDLE

Installs tkinter and the IDLE development environment.

Python test suite

Installs the standard library test suite.

py launcher  for all users (requires elevation)

Use Programs and Features to remove the 'py' launcher.



python  
for  
windows

Back

Next

Cancel

## Advanced Options

- Install for all users
- Associate files with Python (requires the py launcher)
- Create shortcuts for installed applications
- Add Python to environment variables
- Precompile standard library
- Download debugging symbols
- Download debug binaries (requires VS 2015 or later)

### Customize install location

Browse

python  
for  
windows

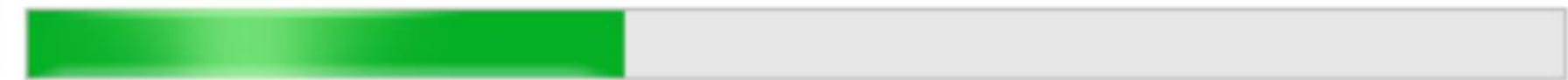
Back **Install**Cancel

# Setup Progress



Installing:

Python 3.7.0 Development Libraries (32-bit)



python  
for  
windows

Cancel

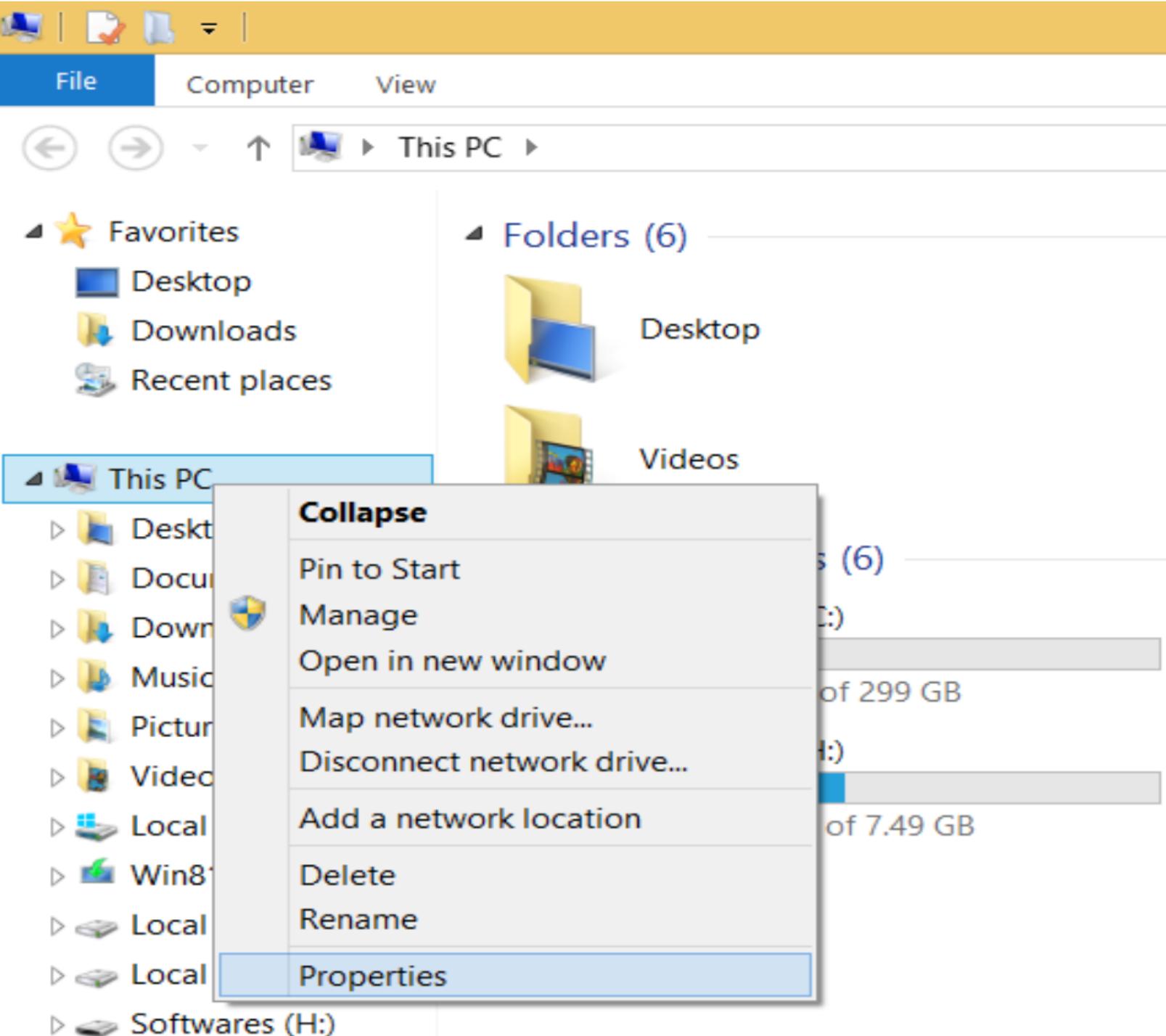
- Now, try to run `python` on the command prompt. Type the command **python** in case of `python2` or `python3` in case of **python3**.
- It will show an error as given in the below image. It is because we haven't set the path.

Microsoft Windows [Version 6.3.9600]  
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\developer>python  
'python' is not recognized as an internal or external command,  
operable program or batch file.

C:\Users\developer>\_

- To set the path of python, we need to right click on "my computer" and go to Properties → Advanced → Environment Variables.



# System Properties

X

Computer Name    Hardware    Advanced    System Protection    Remote

You must be logged on as an Administrator to make most of these changes.

## Performance

Visual effects, processor scheduling, memory usage, and virtual memory

[Settings...](#)

## User Profiles

Desktop settings related to your sign-in

[Settings...](#)

## Startup and Recovery

System startup, system failure, and debugging information

[Settings...](#)

[Environment Variables...](#)

OK

Cancel

Apply

## Environment Variables

x

## User variables for developer

Variable	Value
PATH	C:\Program Files\Java\jdk-10.0.2\bin
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

New... Edit... Delete

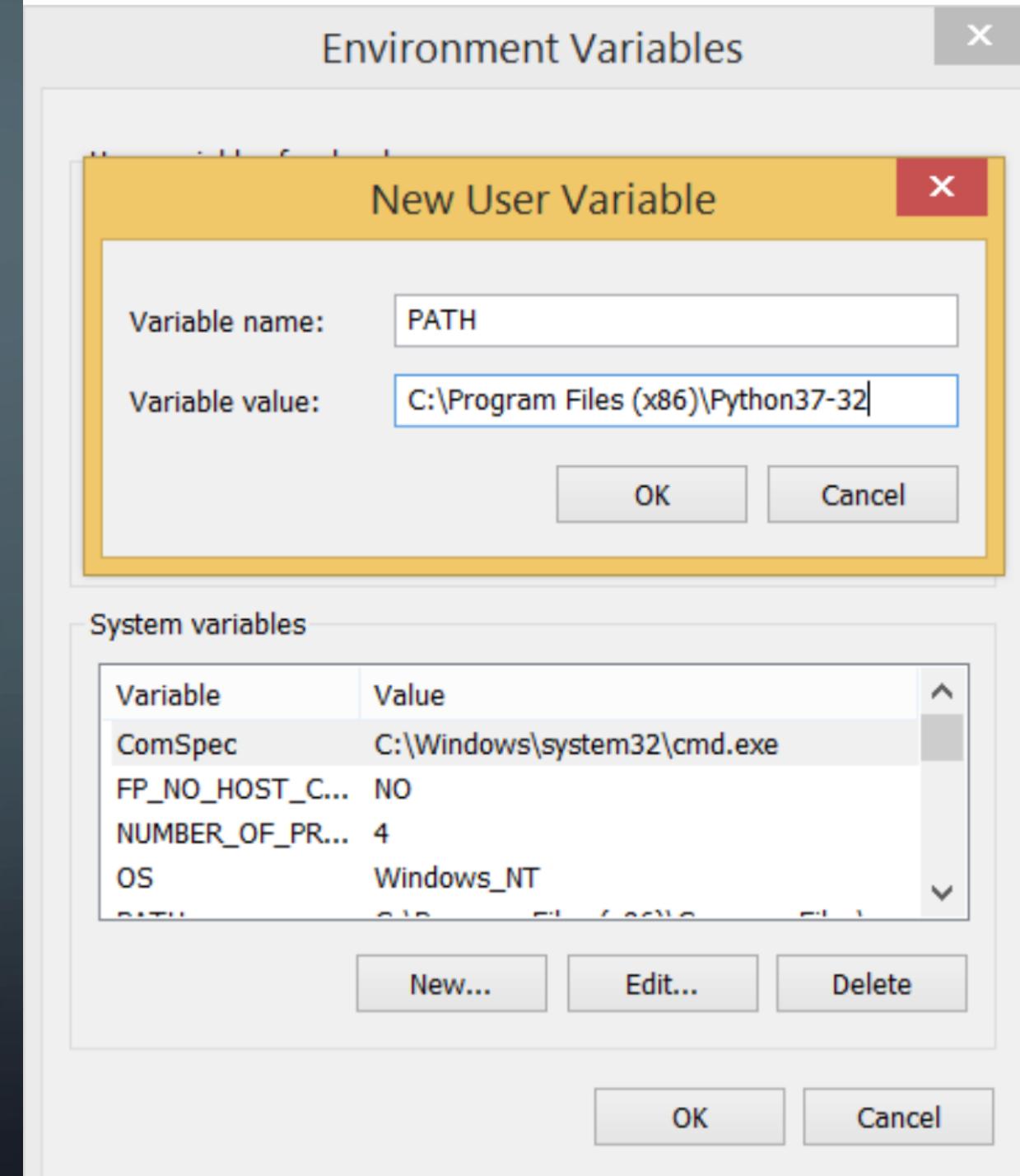
## System variables

[New...](#) [Edit...](#) [Delete](#)

OK | Cancel

- Add the new path variable in the user variable section.

Type **PATH** as the variable name and set the path to the installation directory of the python shown in the below image.



- Now, the path is set, we are ready to run python on our local system.
- Restart CMD, and type **python** again. It will open the python interpreter shell where we can execute the python statements.

Command Prompt - python

Microsoft Windows [Version 10.0.18362.356]

(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\dvachhani>python

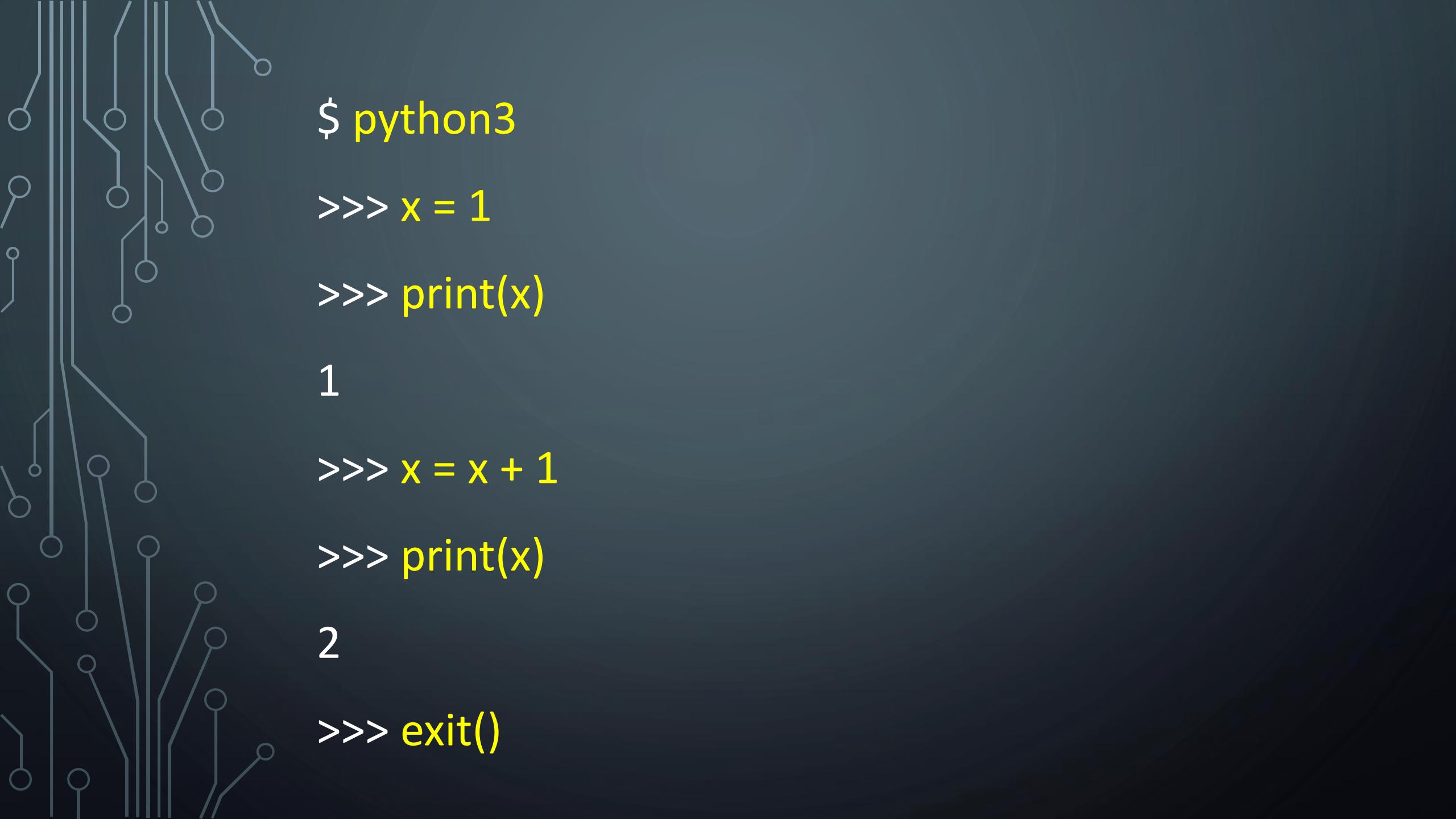
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32

Warning:

This Python interpreter is in a conda environment, but the environment has  
not been activated. Libraries may fail to load. To activate this environment  
please see <https://conda.io/activation>

Type "help", "copyright", "credits" or "license" for more information.

>>>

A faint, light-grey circuit board pattern serves as the background for the slide.

```
$ python3
```

```
>>> x = 1
```

```
>>> print(x)
```

```
1
```

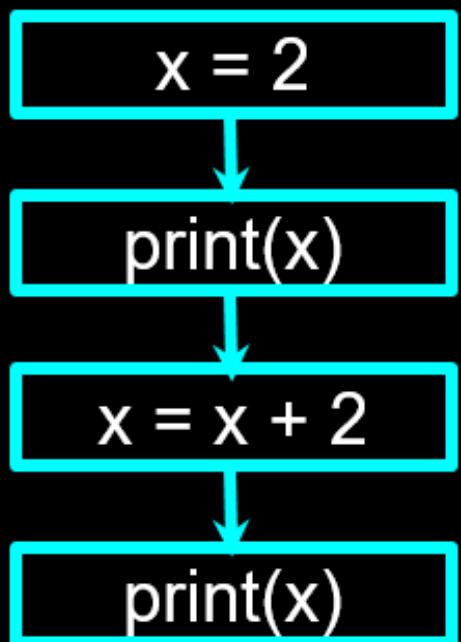
```
>>> x = x + 1
```

```
>>> print(x)
```

```
2
```

```
>>> exit()
```

# Sequential Steps



Program:

x = 2

print(x)

x = x + 2

print(x)

Output:

2

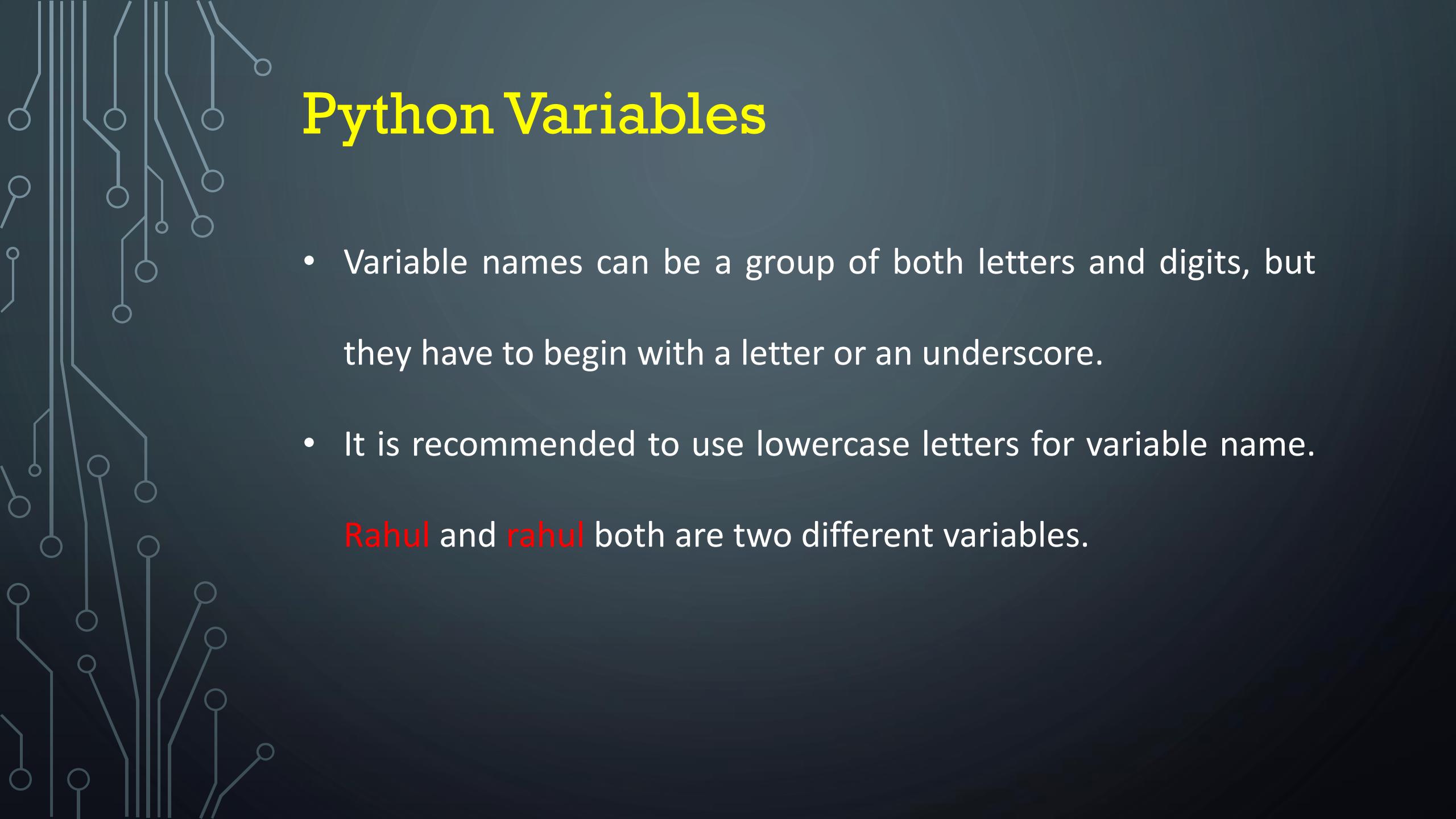
4

When a program is running, it flows from one step to the next. As programmers, we set up “paths” for the program to follow.



# Python Variables

- Variable is a name which is used to refer memory location.
- Variable also known as identifier and used to hold value.
- In Python, we don't need to specify the type of variable because Python is a type infer language and smart enough to get variable type.



# Python Variables

- Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore.
- It is recommended to use lowercase letters for variable name.  
**Rahul** and **rahul** both are two different variables.

In [3]: ► A = 10  
print(a)

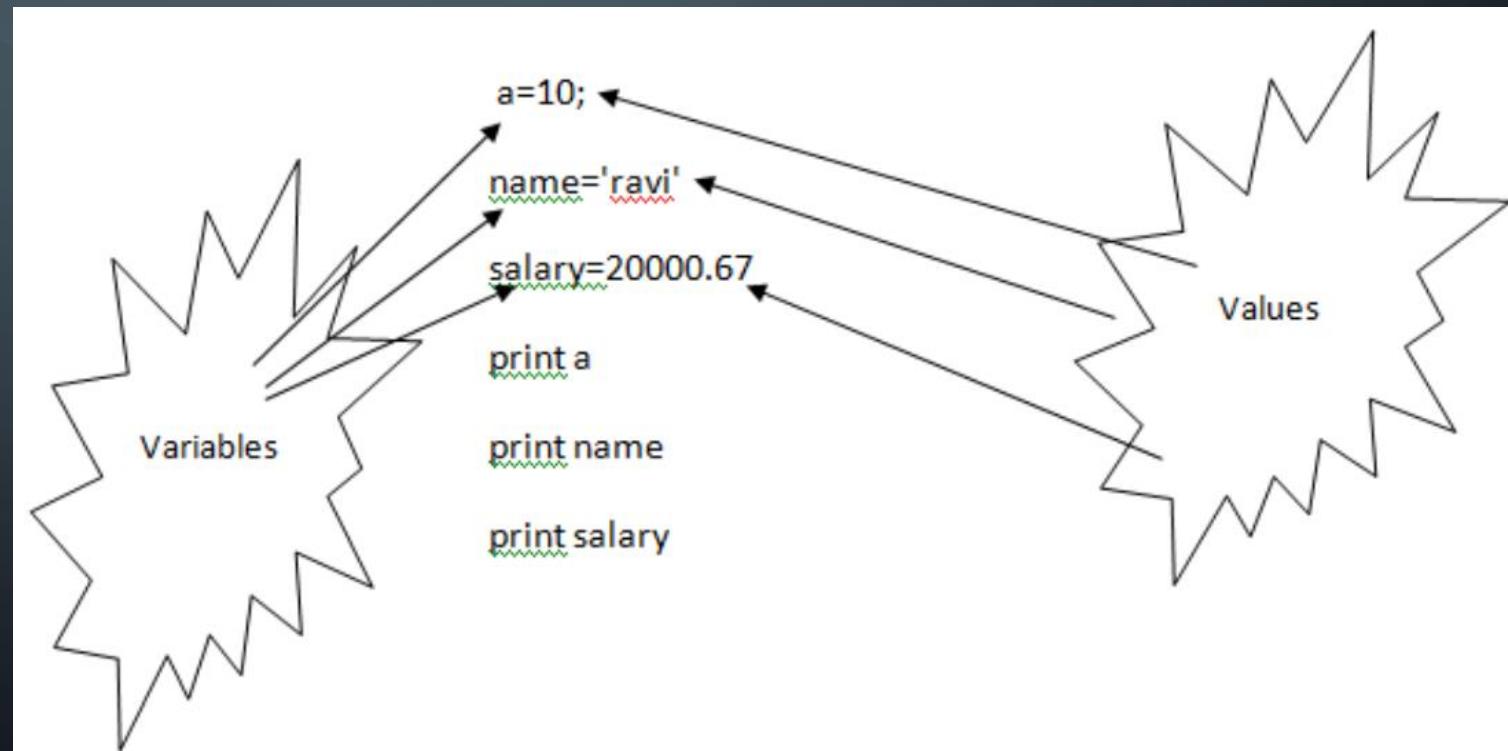
```
-----  
NameError                                     Traceback (most recent call last)  
<ipython-input-3-c12ea66e89bc> in <module>  
      1 A = 10  
----> 2 print(a)  
  
NameError: name 'a' is not defined
```

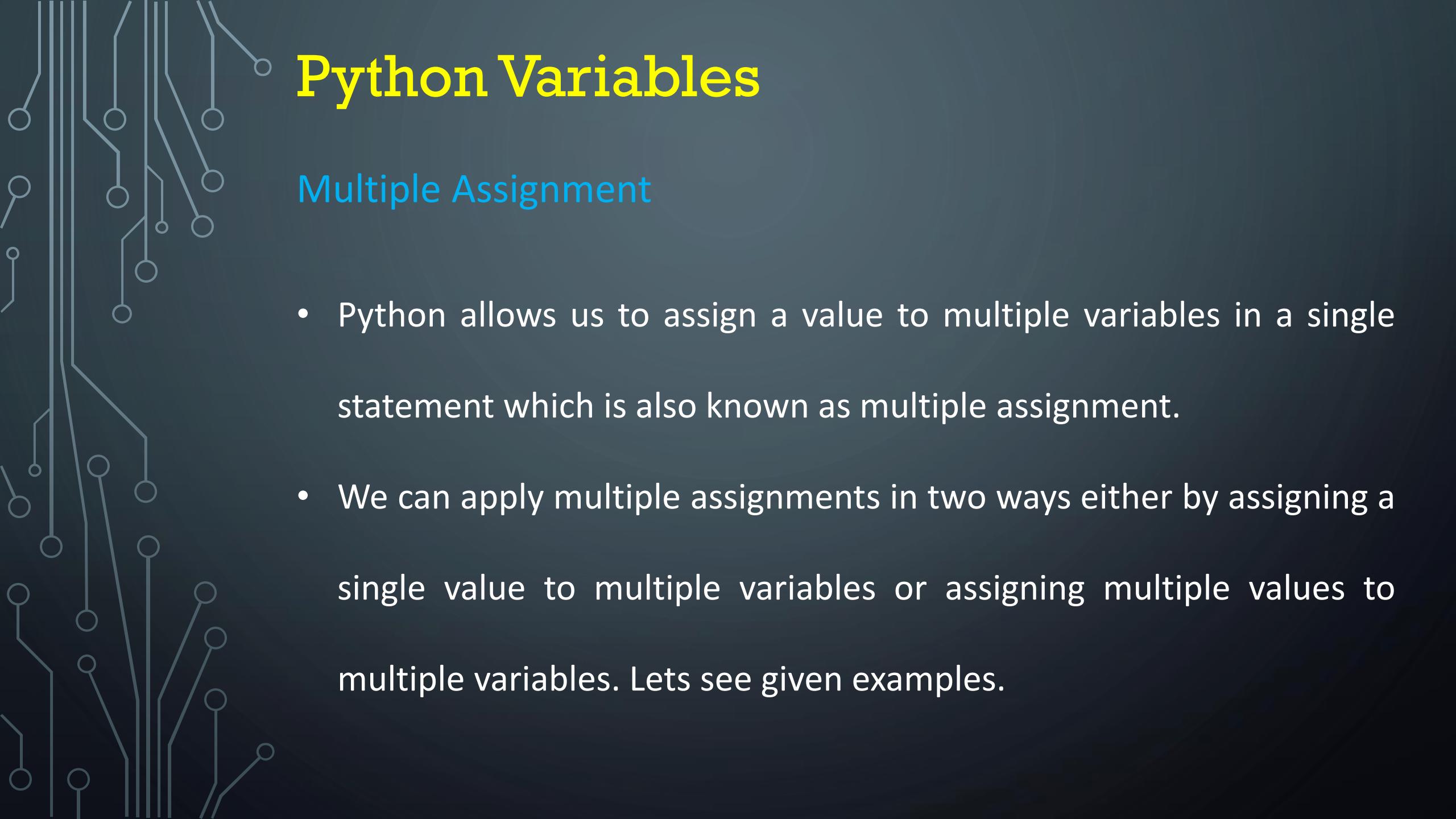
In [4]: ► Rahul = 100  
print(rahul)

```
-----  
NameError                                     Traceback (most recent call last)  
<ipython-input-4-ee679276ac45> in <module>  
      1 Rahul = 100  
----> 2 print(rahul)  
  
NameError: name 'rahul' is not defined
```

# Python Variables

- When we assign any value to the variable that variable is declared automatically.
- The equal (=) operator is used to assign value to a variable.
- This is also called **single assignment**

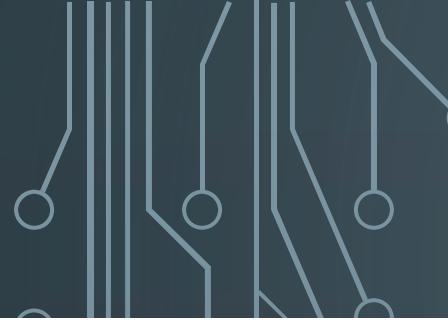




# Python Variables

## Multiple Assignment

- Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment.
- We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Lets see given examples.



# Python Variables

## Multiple Assignment

### Assigning single value to multiple variable

In [1]: ► `x=y=z=50`

In [2]: ► `print(x)`  
`print(y)`  
`print(z)`

50

50

50



# Python Variables

## Multiple Assignment

### 2. Assigning multiple values to multiple variables:

Eg:

```
a,b,c=5,10,15
```

```
print a
```

```
print b
```

```
print c
```

Output:

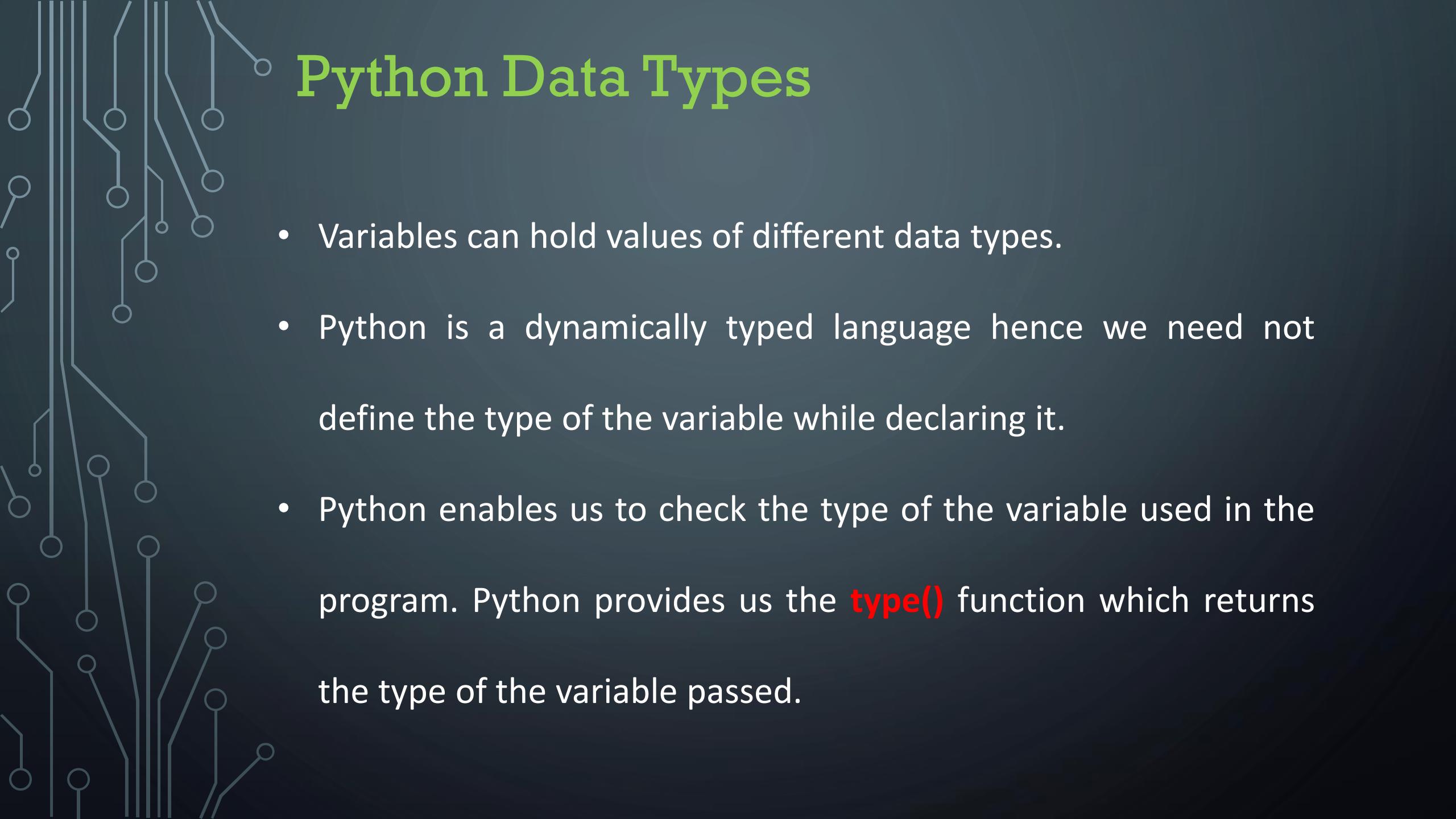
```
>>>
```

```
5
```

```
10
```

```
15
```

```
>>>
```



# Python Data Types

- Variables can hold values of different data types.
- Python is a dynamically typed language hence we need not define the type of the variable while declaring it.
- Python enables us to check the type of the variable used in the program. Python provides us the **type()** function which returns the type of the variable passed.



# Python Data Types

```
In [3]: ➜ A = 10  
        B = "Hi Python"  
        C = 10.5
```

```
In [4]: ➜ type(A)  
        type(B)  
        type(C)
```

Out[4]: float

```
In [5]: ➜ print(type(A))  
        print(type(B))  
        print(type(C))
```

```
<class 'int'>  
<class 'str'>  
<class 'float'>
```



# Python Standard Data Types

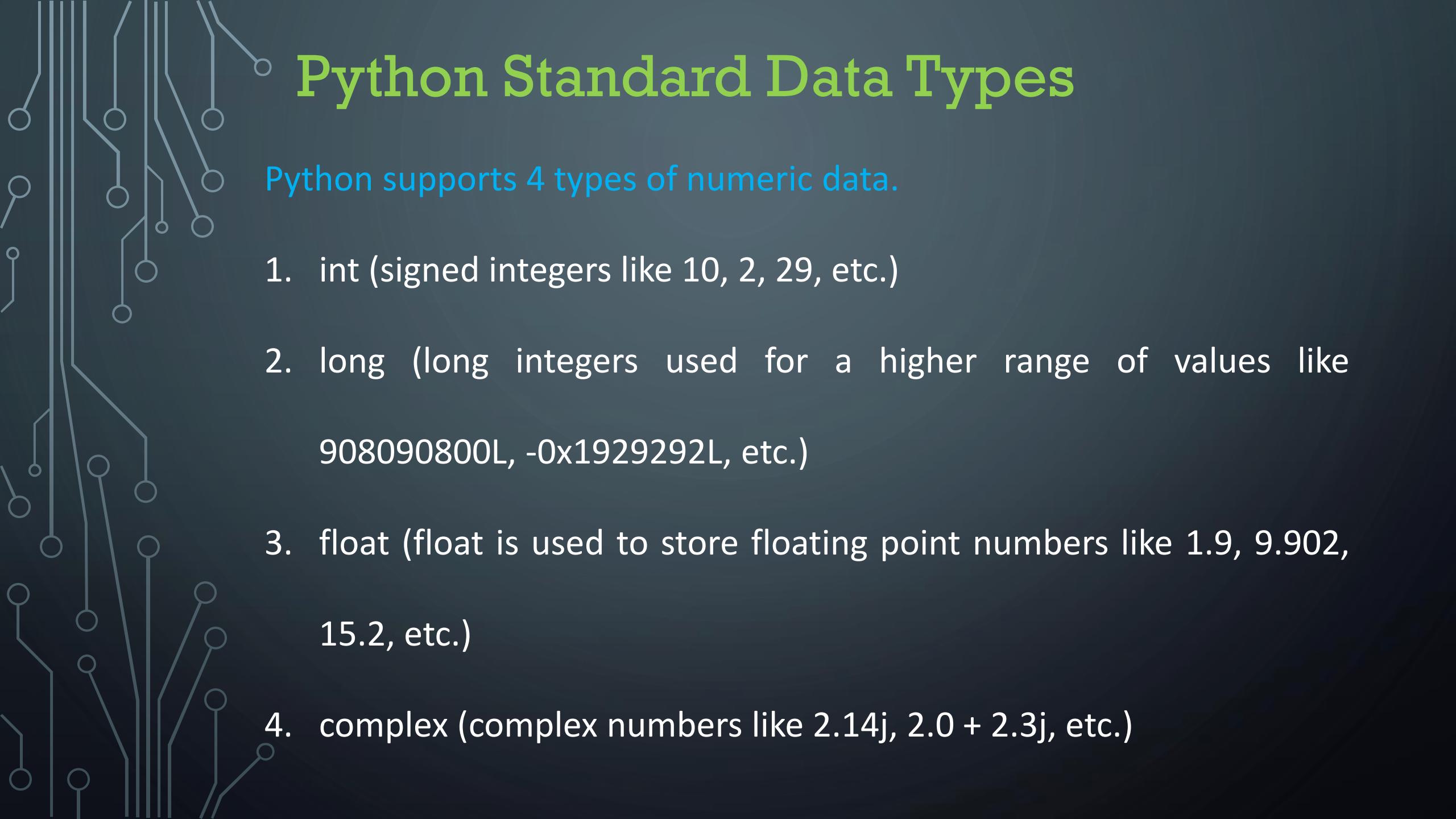
- Numbers
- String
- List
- Tuple
- Dictionary



# Python Standard Data Types

## Numbers

- Number stores numeric values.
- Python creates Number objects when a number is assigned to a variable.
- `a = 3 , b = 5 #a and b are number objects`



# Python Standard Data Types

Python supports 4 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)
3. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)
4. complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)



# Python Standard Data Types

## Strings

- The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.
- String handling in python is a straightforward task since there are various inbuilt functions and operators provided.
- In the case of string handling, the operator + is used to concatenate two strings as the operation "`hello`"+"`python`" returns "`hello python`".
- The operator \* is known as repetition operator as the operation "`Python`" \*2 returns "`Python Python`".

# Python Standard Data Types

## Strings

```
str1 = 'hello javatpoint' #string str1  
str2 = ' how are you' #string str2  
print (str1[0:2]) #printing first two character using slice operator  
print (str1[4]) #printing 4th character of the string  
print (str1*2) #printing the string twice  
print (str1 + str2) #printing the concatenation of str1 and str2
```

### Output:

```
he  
o  
hello javatpointhello javatpoint  
hello javatpoint how are you
```



# Python Standard Data Types

## List

- Lists are similar to arrays in C. However; the list can contain data of different types.
- The items stored in the list are separated with a comma (,) and enclosed within square brackets [].
- We can use slice [:] operators to access the data of the list.
- The concatenation operator (+) and repetition operator (\*) works with the list in the same way as they were working with the strings.

# Python Standard Data Types

## List

```
l = [1, "hi", "python", 2]  
print (l[3:]);  
print (l[0:2]);  
print (l);  
print (l + l);  
print (l * 3);
```

### Output:

```
[2]  
[1, 'hi']  
[1, 'hi', 'python', 2]  
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2]  
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2, 1, 'hi', 'python', 2]
```



# Python Standard Data Types

## Tuple

- A tuple is similar to the list in many ways.
- Like lists, tuples also contain the collection of the items of different data types.
- The items of the tuple are separated with a comma (,) and enclosed in parentheses ().
- A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

# Python Standard Data Types

## Tuple

```
t = ("hi", "python", 2)
print (t[1:]);
print (t[0:1]);
print (t);
print (t + t);
print (t * 3);
print (type(t))
t[2] = "hi";
```

### Output:

```
('python', 2)
('hi',)
('hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)
<type 'tuple'>
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```



# Python Standard Data Types

## Dictionary

- Dictionary is an ordered set of a key-value pair of items.
- It is like an associative array or a hash table where each key stores a specific value.
- Key can hold any primitive data type whereas value is an arbitrary Python object.
- The items in the dictionary are separated with the comma and enclosed in the curly braces { }.



# Python Standard Data Types

## Dictionary

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'};  
print("1st name is "+d[1]);  
print("2nd name is "+ d[4]);  
print (d);  
print (d.keys());  
print (d.values());
```

### Output:

```
1st name is Jimmy  
2nd name is mike  
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}  
[1, 2, 3, 4]  
['Jimmy', 'Alex', 'john', 'mike']
```



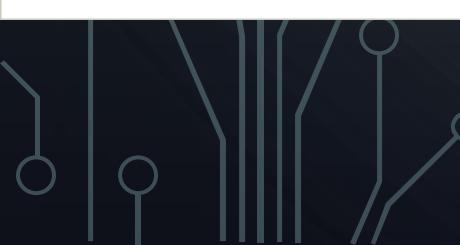
# Python Keywords

- Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter.
- Each keyword have a special meaning and a specific operation. These keywords can't be used as variable.



# Python Keywords

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





# Python Operators

- The operator can be defined as a symbol which is responsible for a particular operation between two operands.
- Operators are the pillars of a program on which the logic is built in a particular programming language.



# Python Operators

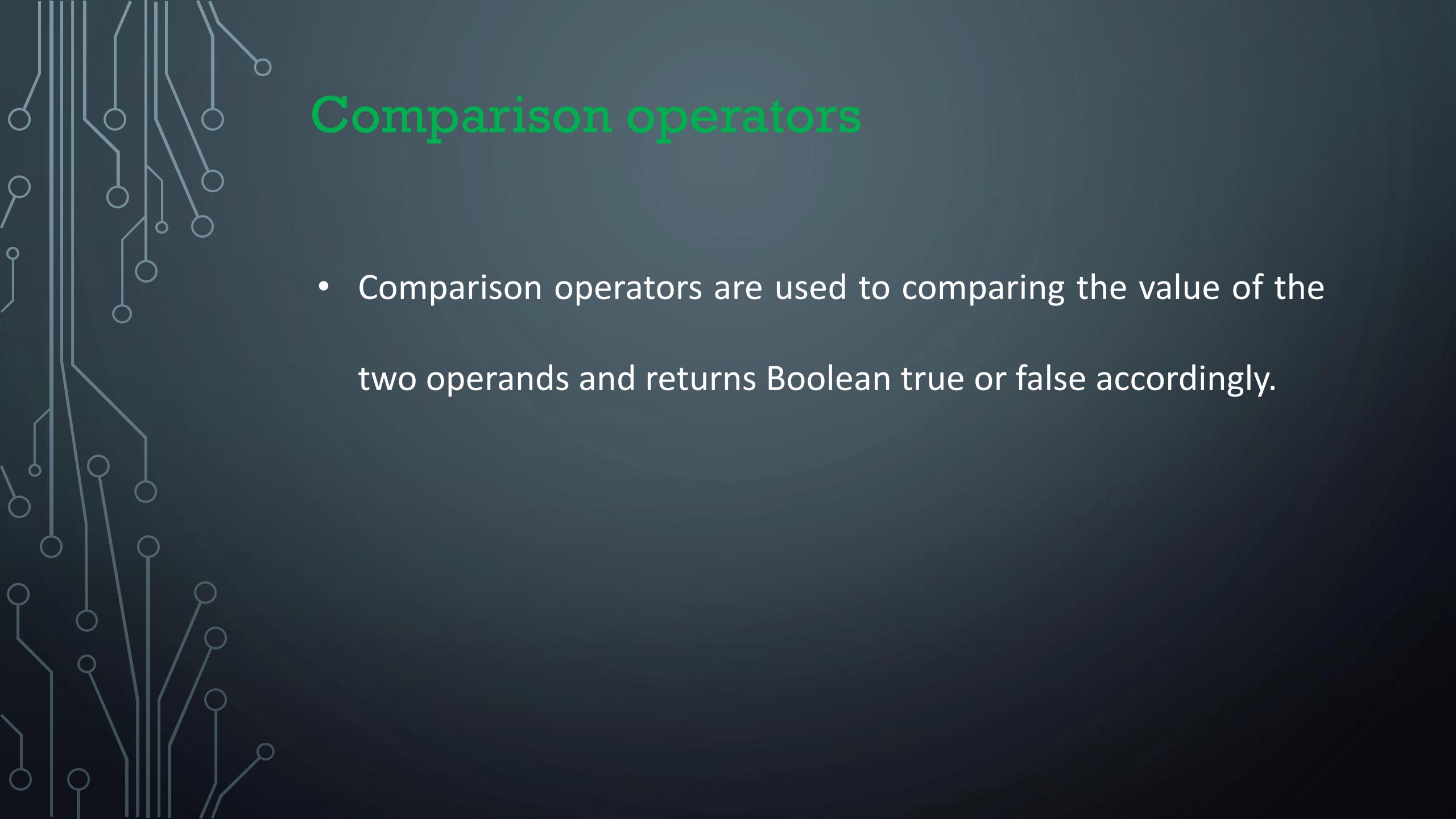
1. Arithmetic operators
2. Comparison operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators



# Arithmetic operators

- Arithmetic operators are used to perform arithmetic operations between two operands.
- It includes +(addition), - (subtraction), \*(multiplication), /(divide), %(remainder), //(floor division), and exponent (\*\*).

<b>Operator</b>	<b>Description</b>
<b>+ (Addition)</b>	It is used to add two operands. For example, if $a = 20$ , $b = 10 \Rightarrow a+b = 30$
<b>- (Subtraction)</b>	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if $a = 20$ , $b = 10 \Rightarrow a - b = 10$
<b>/ (divide)</b>	It returns the quotient after dividing the first operand by the second operand. For example, if $a = 20$ , $b = 10 \Rightarrow a/b = 2$
<b>*</b> <b>(Multiplication)</b>	It is used to multiply one operand with the other. For example, if $a = 20$ , $b = 10 \Rightarrow a * b = 200$
<b>% (remainder)</b>	It returns the remainder after dividing the first operand by the second operand. For example, if $a = 20$ , $b = 10 \Rightarrow a \% b = 0$
<b>** (Exponent)</b>	It is an exponent operator represented as it calculates the first operand power to second operand.
<b>// (Floor division)</b>	It gives the floor value of the quotient produced by dividing the two operands.



# Comparison operators

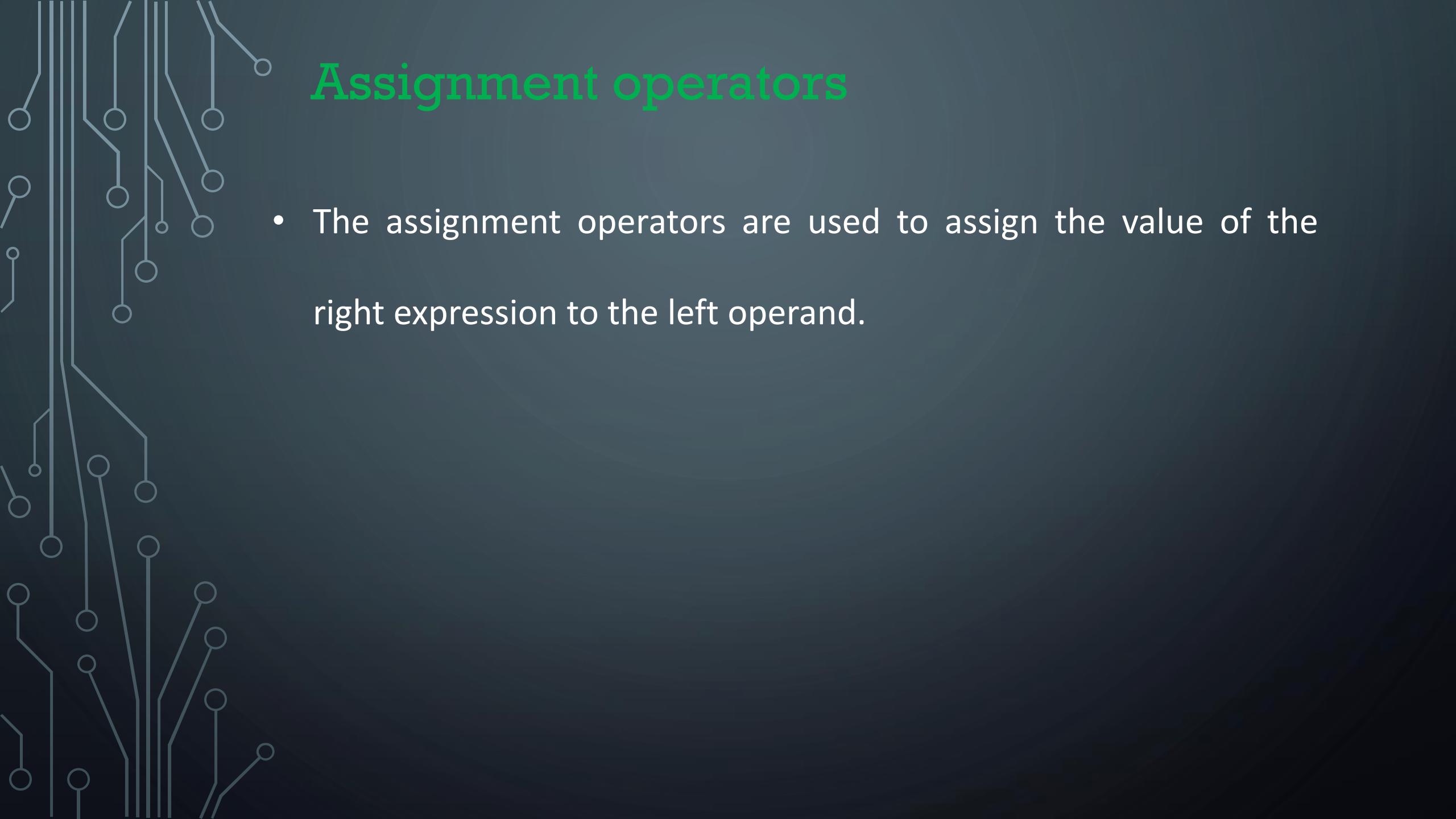
- Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly.



# Comparison operators

Operator	Description
<code>==</code>	If the value of two operands is equal, then the condition becomes true.
<code>!=</code>	If the value of two operands is not equal then the condition becomes true.
<code>&lt;=</code>	If the first operand is less than or equal to the second operand, then the condition becomes true.
<code>&gt;=</code>	If the first operand is greater than or equal to the second operand, then the condition becomes true.
<code>&lt;&gt;</code>	If the value of two operands is not equal, then the condition becomes true.
<code>&gt;</code>	If the first operand is greater than the second operand, then the condition becomes true.
<code>&lt;</code>	If the first operand is less than the second operand, then the condition becomes true.

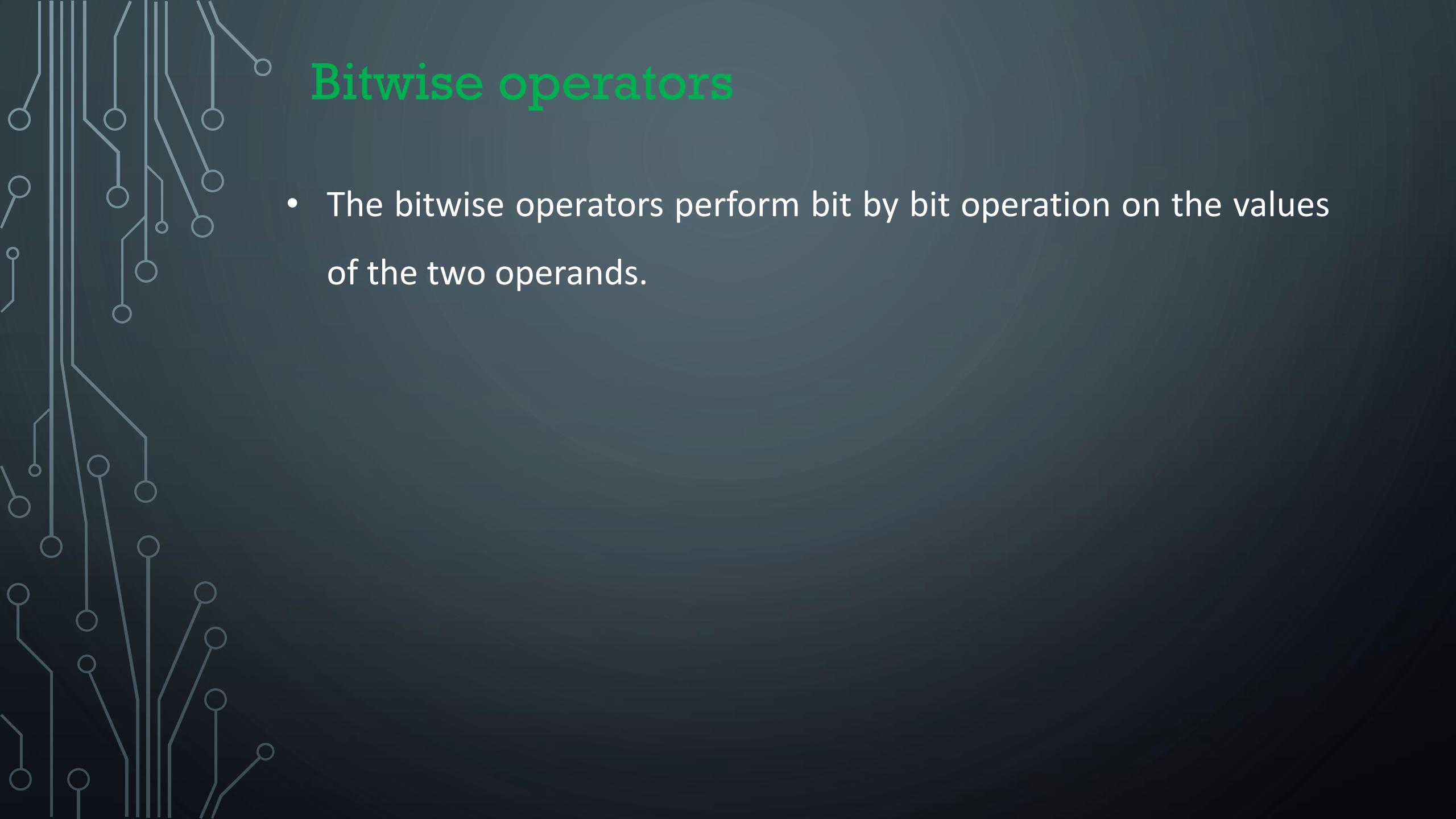




# Assignment operators

- The assignment operators are used to assign the value of the right expression to the left operand.

<b>Operator</b>	<b>Description</b>
=	It assigns the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 10$ , $b = 20 \Rightarrow a+ = b$ will be equal to $a = a + b$ and therefore, $a = 30$ .
-=	It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 20$ , $b = 10 \Rightarrow a- = b$ will be equal to $a = a - b$ and therefore, $a = 10$ .
*=	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 10$ , $b = 20 \Rightarrow a* = b$ will be equal to $a = a * b$ and therefore, $a = 200$ .
%=	It divides the value of the left operand by the value of the right operand and assign the remainder back to left operand. For example, if $a = 20$ , $b = 10 \Rightarrow a \% = b$ will be equal to $a = a \% b$ and therefore, $a = 0$ .
**=	$a**=b$ will be equal to $a=a**b$ , for example, if $a = 4$ , $b = 2$ , $a**=b$ will assign $4^{**}2 = 16$ to $a$ .
//=	$A//=b$ will be equal to $a = a// b$ , for example, if $a = 4$ , $b = 3$ , $a//=b$ will assign $4//3 = 1$ to $a$ .



# Bitwise operators

- The bitwise operators perform bit by bit operation on the values of the two operands.



# Bitwise operators

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
(binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.



# Logical operators

- The logical operators are used primarily in the expression evaluation to make a decision.

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, $a \rightarrow \text{true}, b \rightarrow \text{true} \Rightarrow a \text{ and } b \rightarrow \text{true}$ .
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, $a \rightarrow \text{true}, b \rightarrow \text{false} \Rightarrow a \text{ or } b \rightarrow \text{true}$ .
not	If an expression <b>a</b> is true then <b>not (a)</b> will be false and vice versa.



# Membership operators

- Python membership operators are used to check the membership of value inside a data structure.
- If the value is present in the data structure, then the resulting value is true otherwise it returns false.

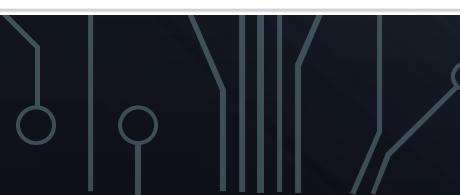
Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).



# Identity operators

- Python identity operators check both the side of operands and its return true and false after comparing the operands.

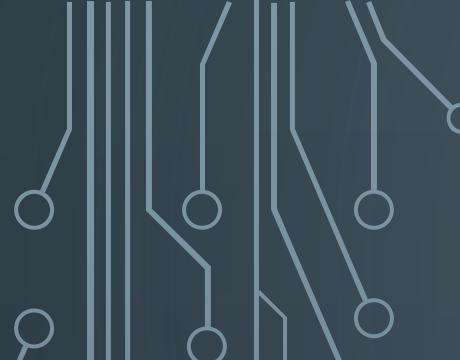
Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both side do not point to the same object.





# Python Comments

- Comments in Python can be used to explain any program code. It can also be used to hide the code as well.
- Comments are the most helpful stuff of any program. It enables us to understand the way, a program works.
- In python, any statement written along with # symbol is known as a comment.
- The interpreter does not interpret the comment.
- Comment is not a part of the program, but it enhances the interactivity of the program and makes the program readable.



# Python Comments

## 1) Single Line Comment:

In case user wants to specify a single line comment, then comment must start with `#`?

**Eg:**

```
# This is single line comment.
```

```
print "Hello Python"
```

**Output:**

```
Hello Python
```

## 2) Multi Line Comment:

Multi lined comment can be given inside triple quotes.

**eg:**

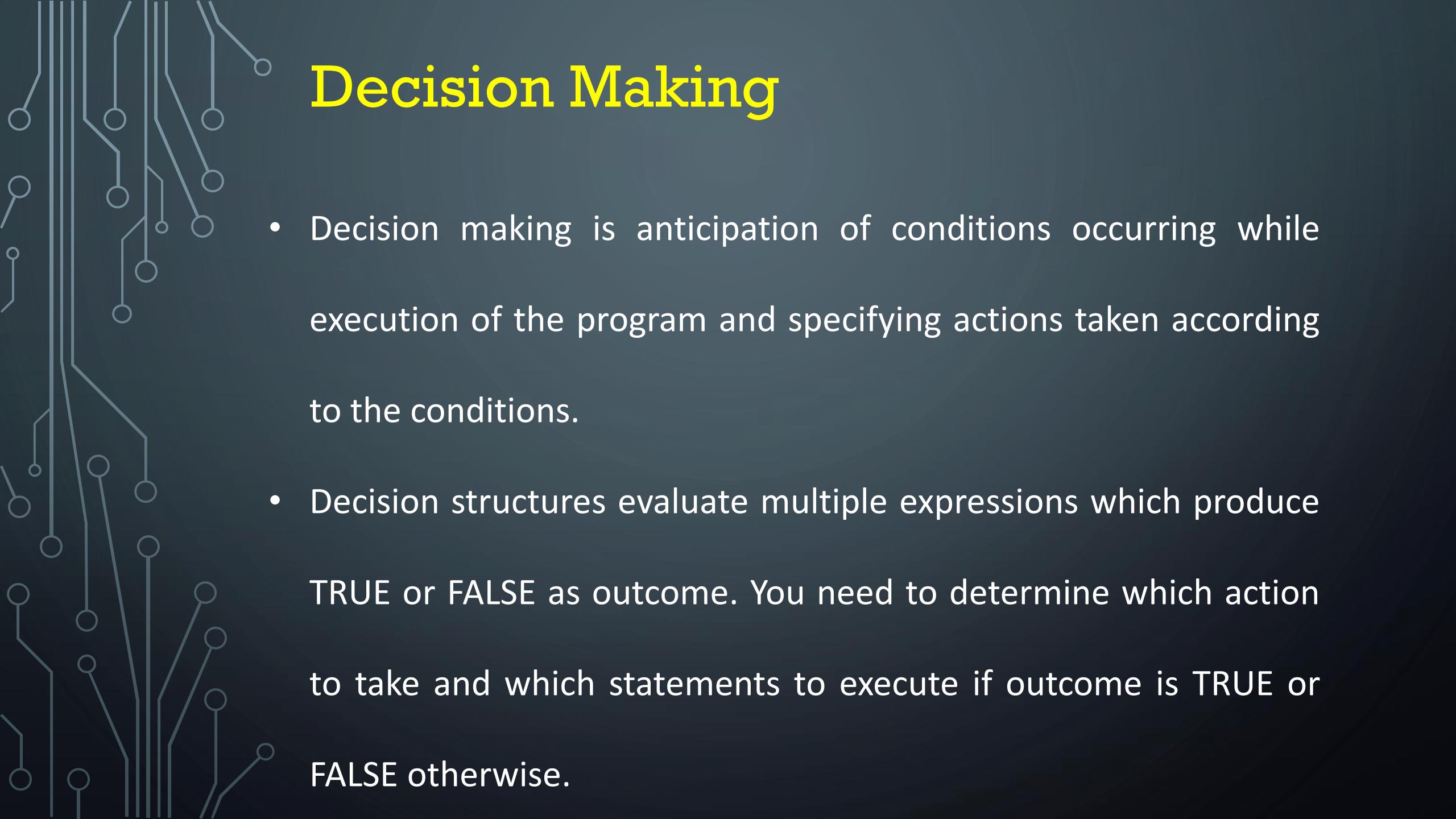
```
""" This  
Is  
Multipline comment""
```

**eg:**

```
#single line comment  
print "Hello Python"  
"""This is  
multiline comment""
```

**Output:**

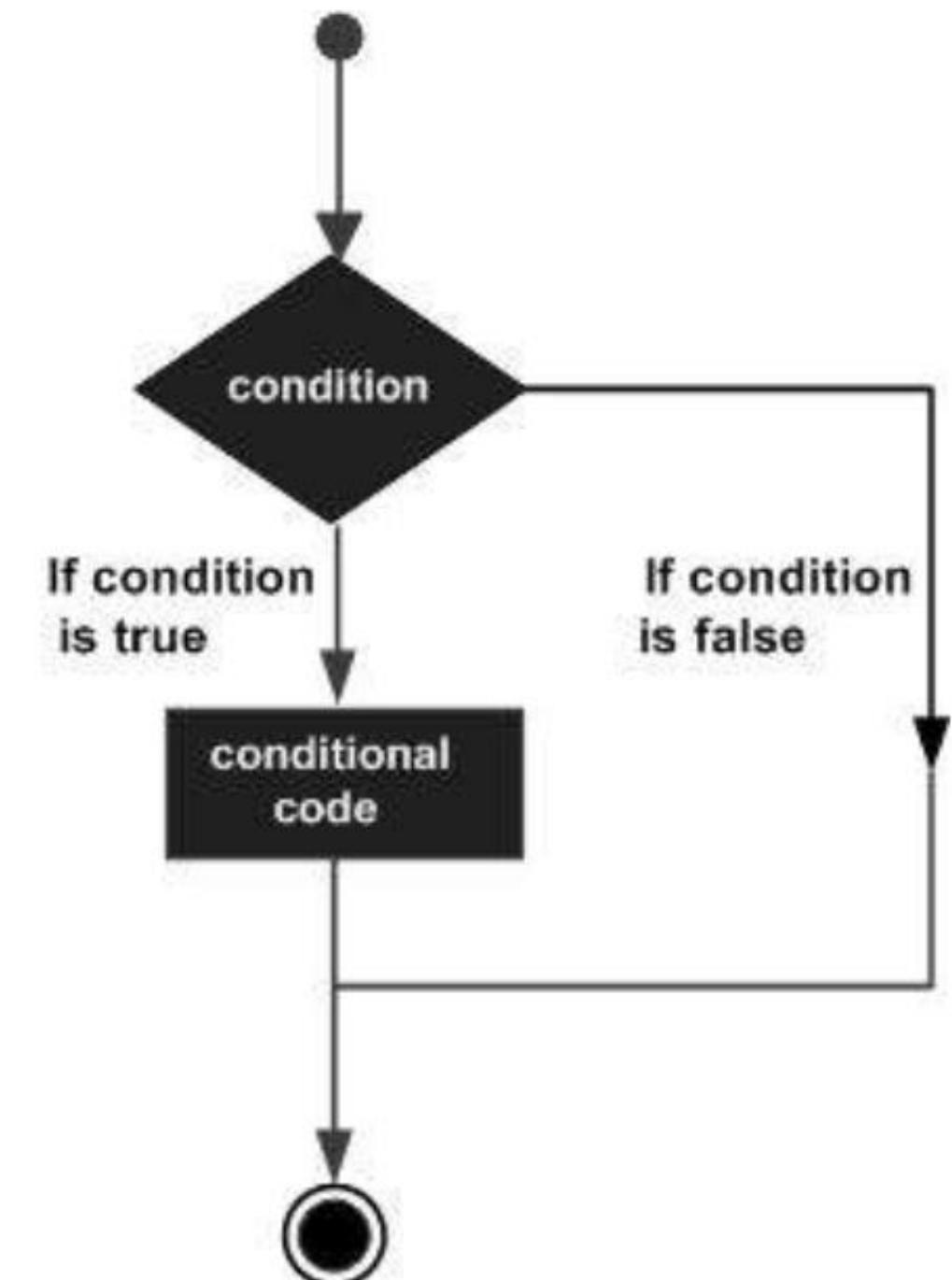
```
Hello Python
```



# Decision Making

- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
- Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

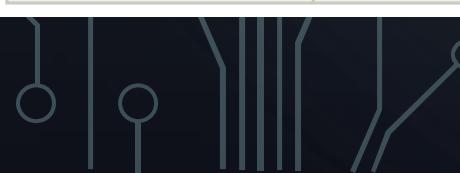
# Decision Making





# Decision Making Example

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.



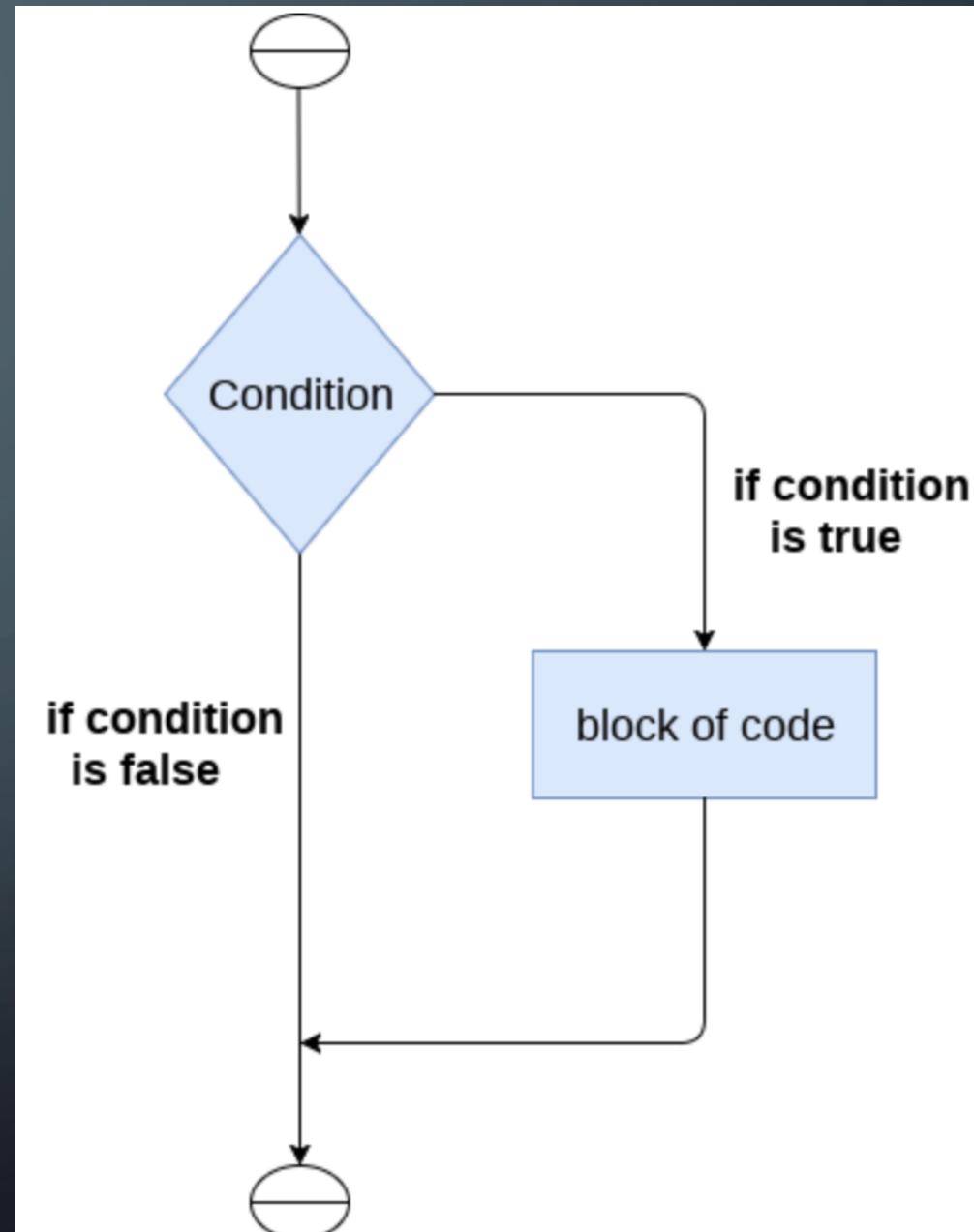


# Indentation in Python

- python doesn't allow the use of parentheses for the block level code.
- In Python, indentation is used to declare a block.
- If two statements are at the same indentation level, then they are the part of the same block.
- Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

# if Statement

- The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block.
- The condition of if statement can be any valid logical expression which can be either evaluated to true or false.





# If statement

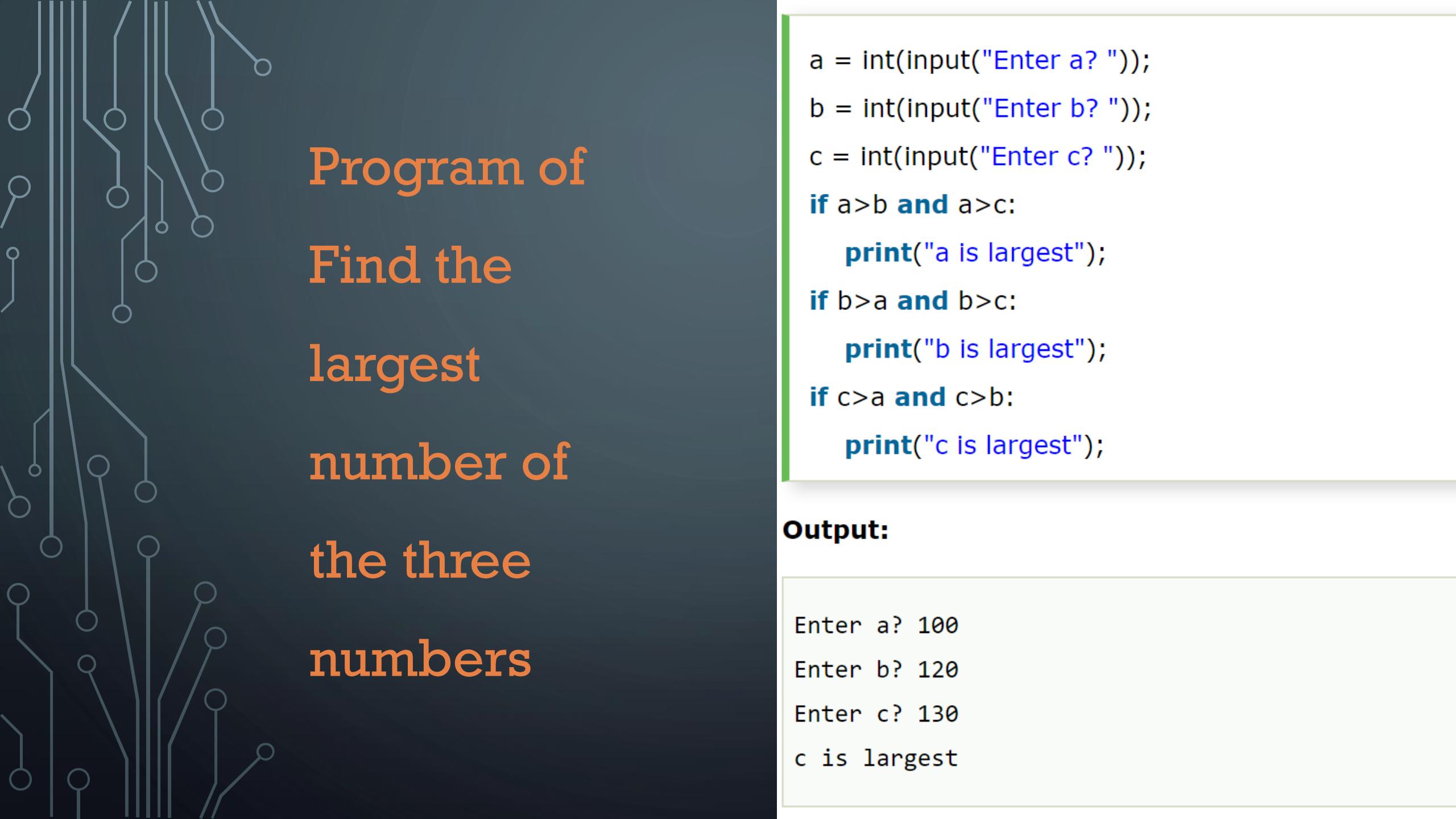
## Syntax

```
if expression:  
    statement
```

```
num = int(input("enter the number?"))  
if num%2 == 0:  
    print("Number is even")
```

### Output:

```
enter the number?10  
Number is even
```

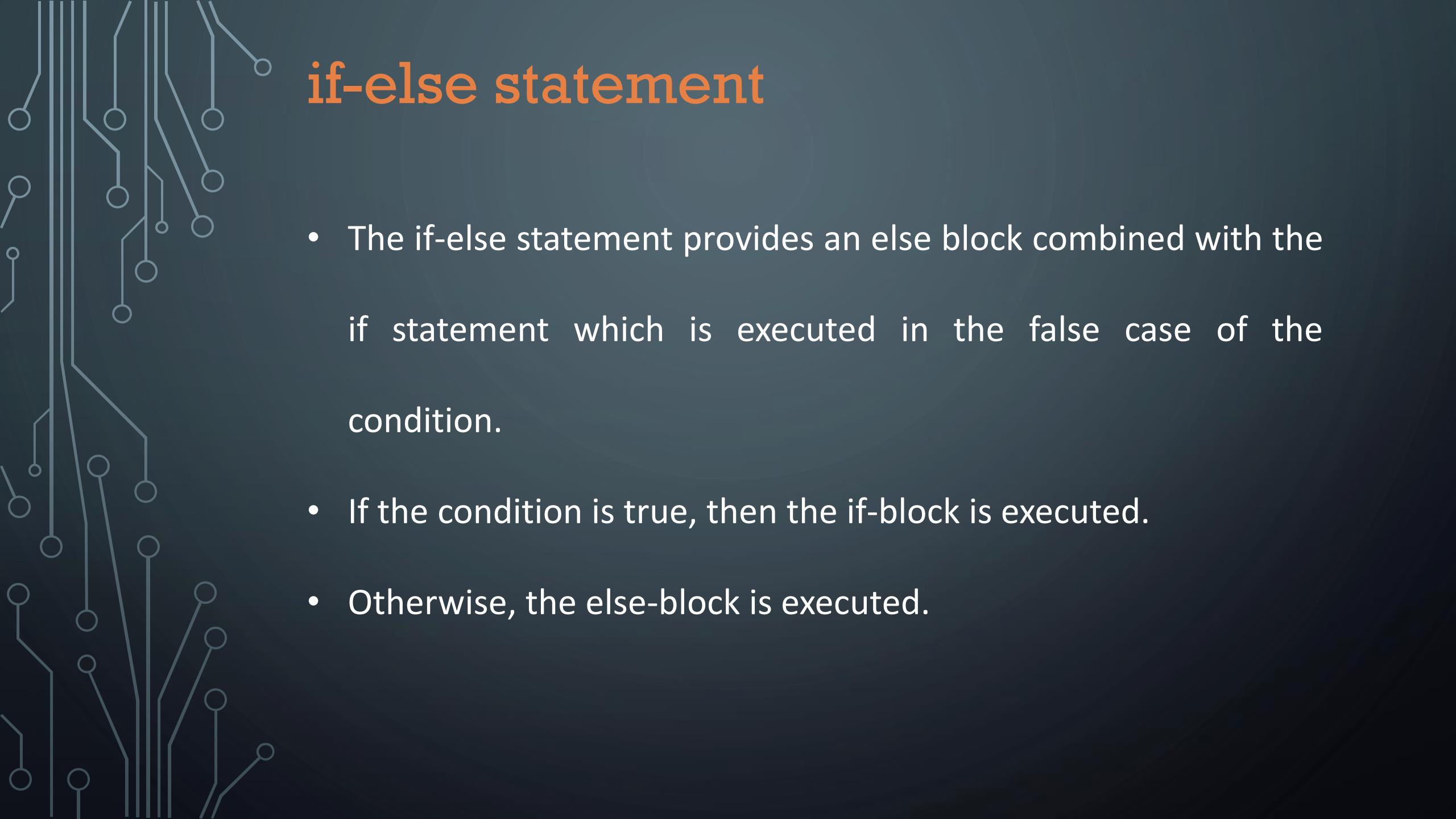


# Program of Find the largest number of the three numbers

```
a = int(input("Enter a? "));  
b = int(input("Enter b? "));  
c = int(input("Enter c? "));  
if a>b and a>c:  
    print("a is largest");  
if b>a and b>c:  
    print("b is largest");  
if c>a and c>b:  
    print("c is largest");
```

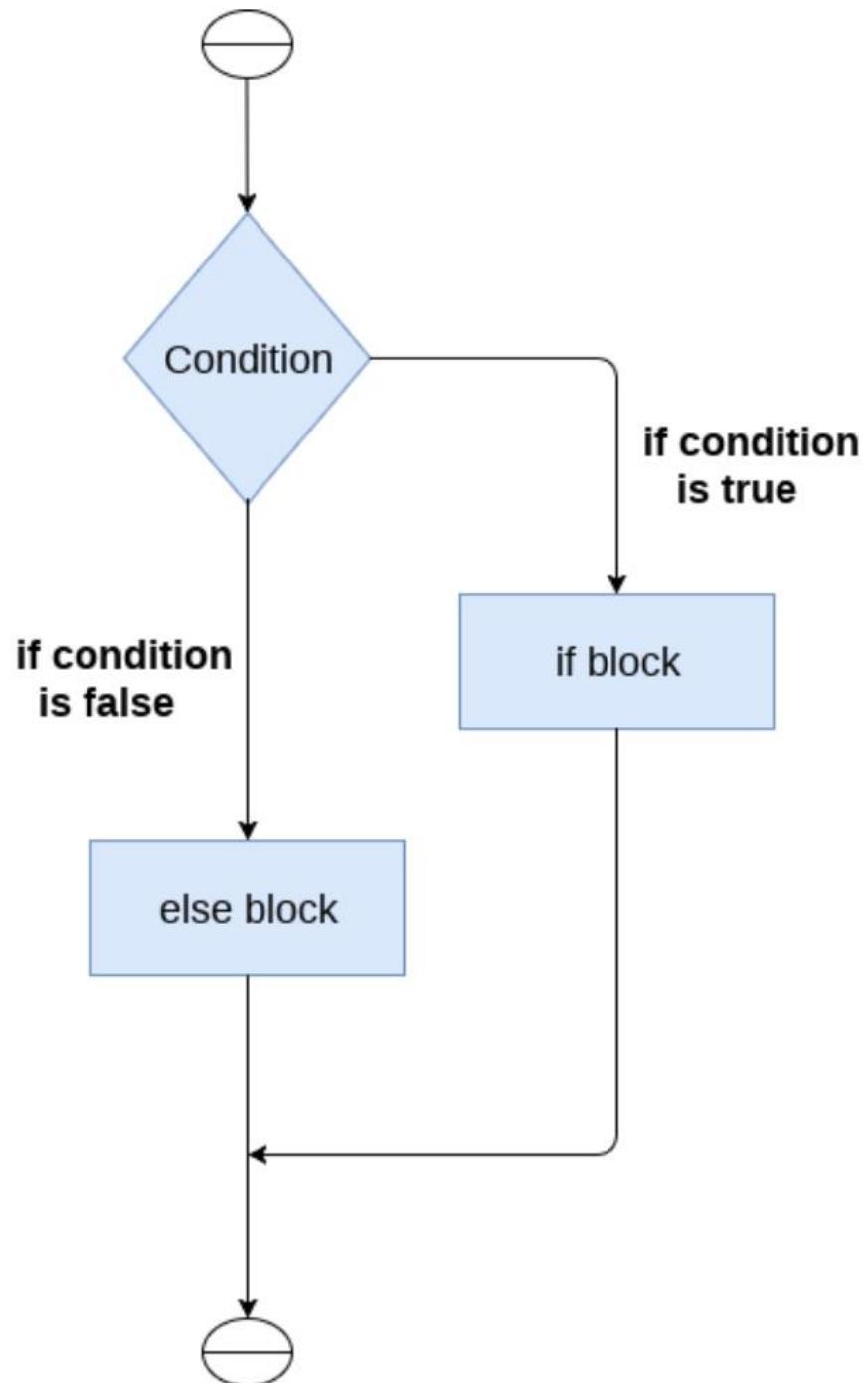
## Output:

```
Enter a? 100  
Enter b? 120  
Enter c? 130  
c is largest
```

A repeating pattern of light gray circuit board tracks and component pads forms the background of the slide.

# if-else statement

- The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.
- If the condition is true, then the if-block is executed.
- Otherwise, the else-block is executed.



**if condition:**

#block of statements

**else:**

#another block of statements (else-block)

## Example 1 : Program to check whether a person is eligible to vote or not.

```
age = int(input("Enter your age? "))
if age>=18:
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

### Output:

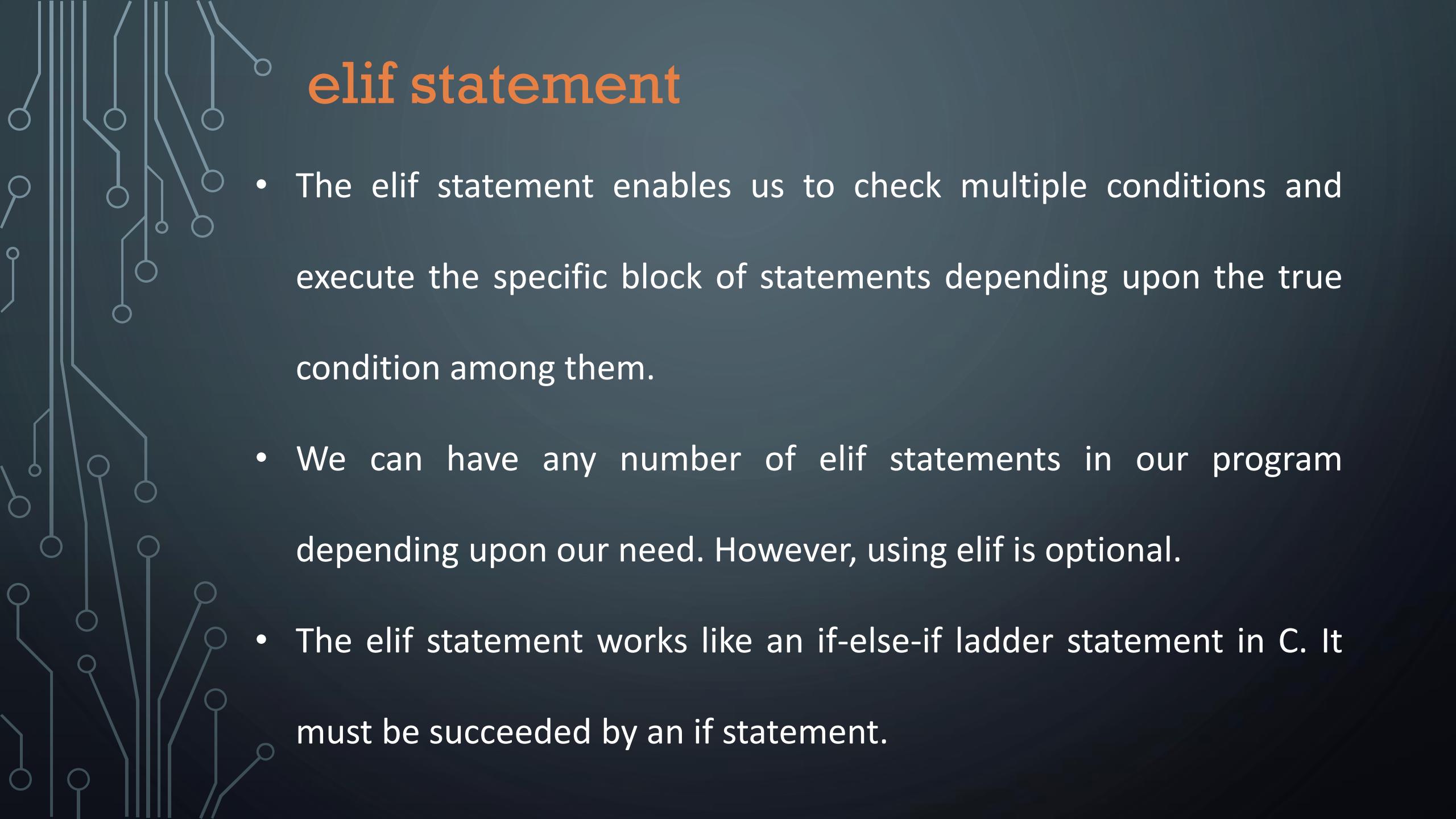
```
Enter your age? 90
You are eligible to vote !!
```

## Example 2: Program to check whether a number is even or not.

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even...")
else:
    print("Number is odd...")
```

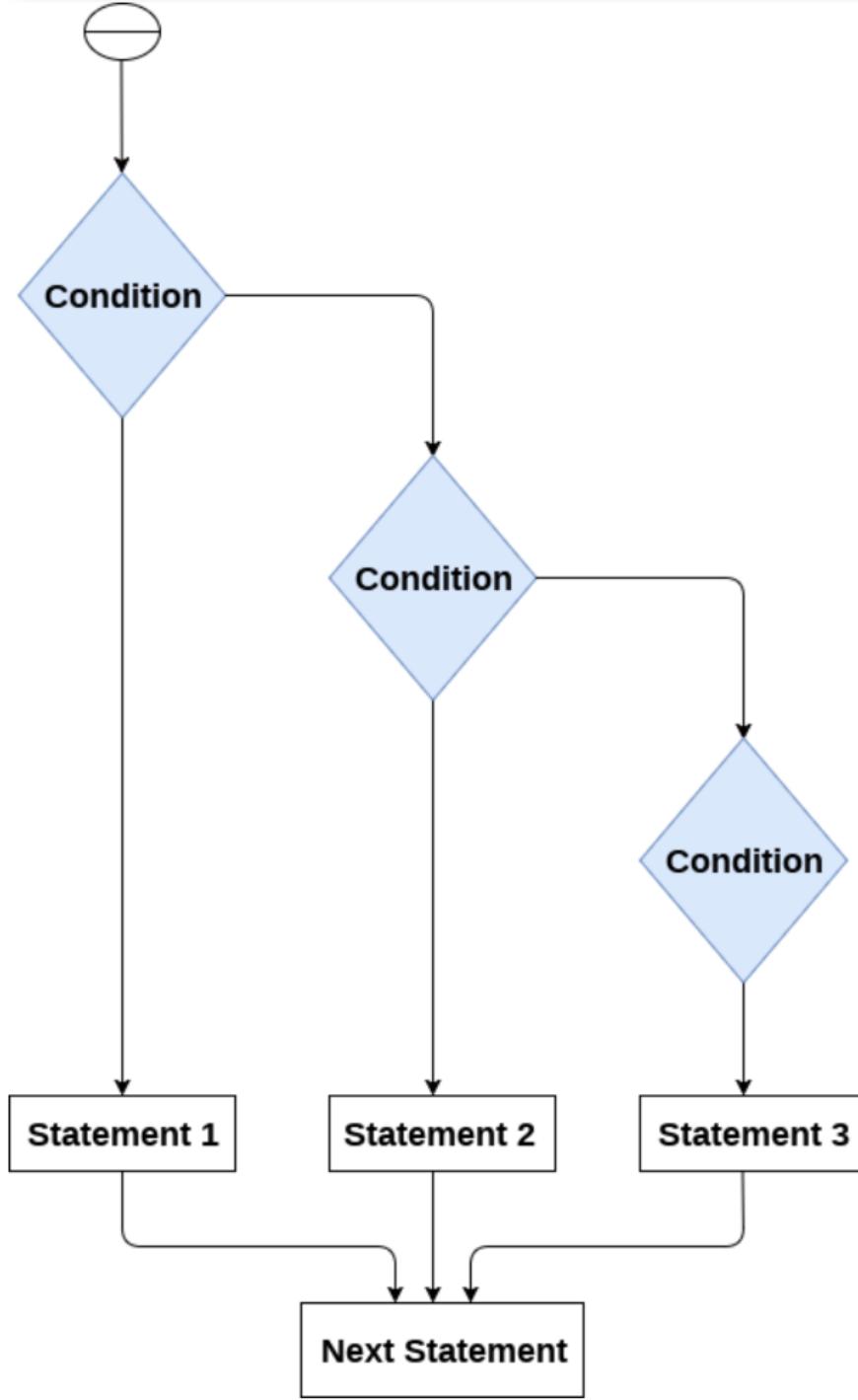
### Output:

```
enter the number?10
Number is even
```



# elif statement

- The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them.
- We can have any number of elif statements in our program depending upon our need. However, using elif is optional.
- The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.



**if expression 1:**

# block of statements

**elif expression 2:**

# block of statements

**elif expression 3:**

# block of statements

**else:**

# block of statements

## Example 1

```
number = int(input("Enter the number?"))

if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

### Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

## Example 2

```
marks = int(input("Enter the marks? "))

if marks > 85 and marks <= 100:
    print("Congrats ! you scored grade A ...")

if marks > 60 and marks <= 85:
    print("You scored grade B + ...")

if marks > 40 and marks <= 60:
    print("You scored grade B ...")

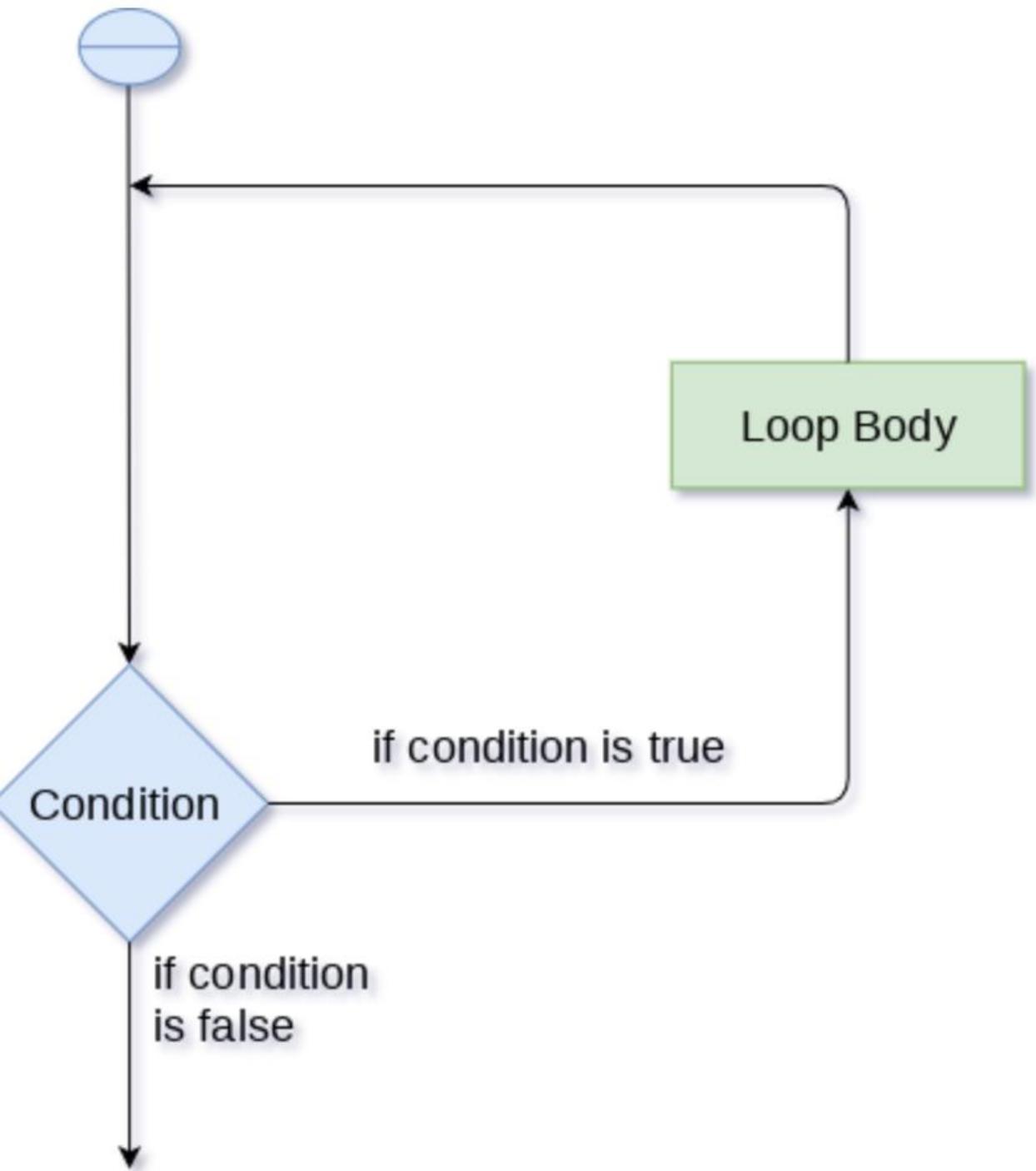
if (marks > 30 and marks <= 40):
    print("You scored grade C ...")

else:
    print("Sorry you are fail ?")
```



# Python Loops

- The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.
- For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



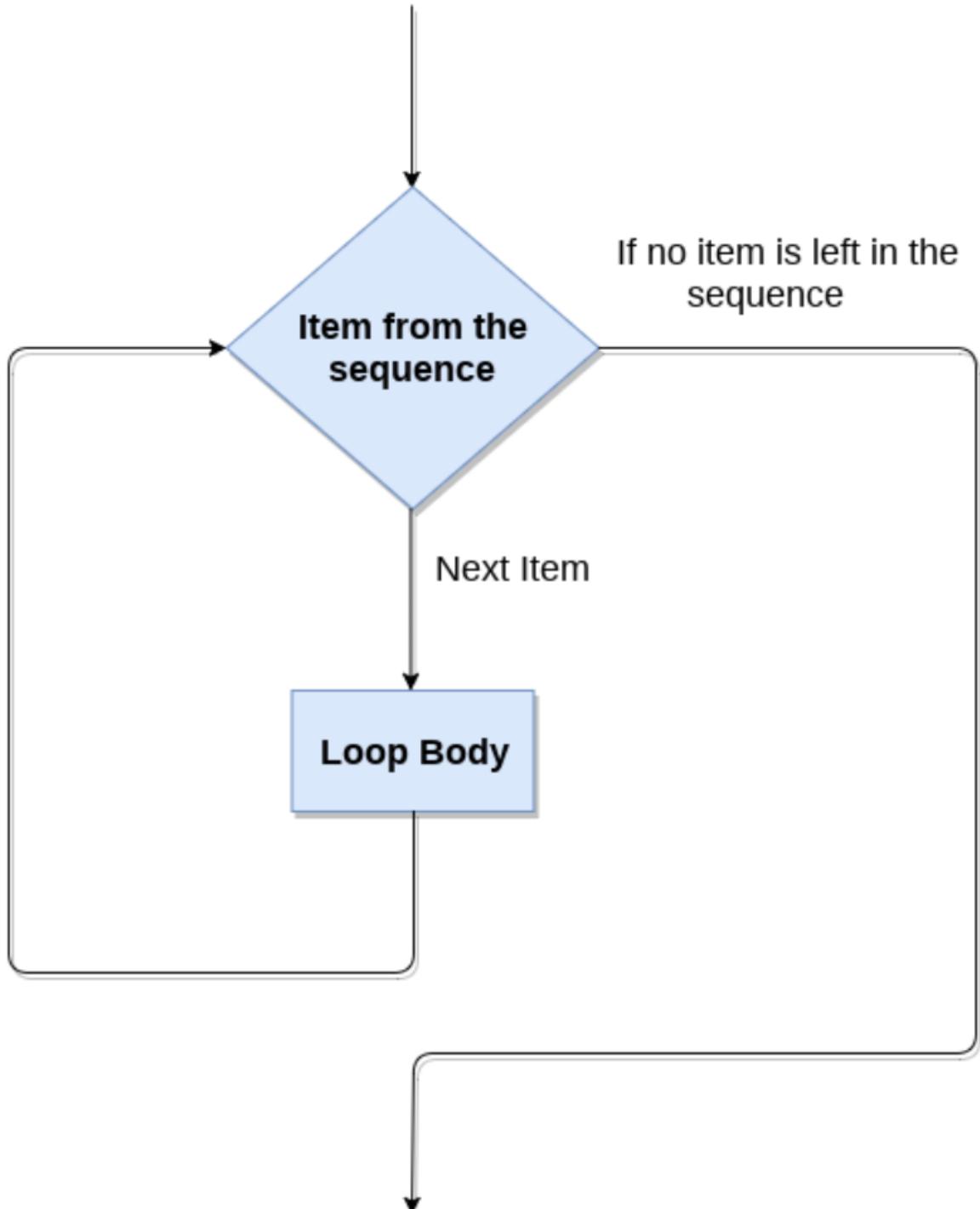
## Advantages of loops

- It provides code re-usability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of data structures (array or linked lists).



# Python for loop

- The **for loop in Python** is used to iterate the statements or a part of the program several times.
- It is frequently used to traverse the data structures like list, tuple, or dictionary.
- The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied.
- The for loop is also called as a per-tested loop.
- It is better to use for loop if the number of iteration is known in advance.



# Syntax

**for iterating\_var in sequence:  
statement(s)**

# Python for loop

Example

```
i = 1
```

```
N = int(input("Enter the number up to which you want to print the natural numbers?"))
```

```
for i in range(0 , 10):  
    print(i , end = ' ')
```

Output

0 1 2 3 4 5 6 7 8 9

## Python for loop example : printing the table of the given number

```
i=1;  
num = int(input("Enter a number:"));  
for i in range(1,11):  
    print("%d X %d = %d"% (num,i,num*i));
```

### Output:

```
Enter a number:10  
10 X 1 = 10  
10 X 2 = 20  
10 X 3 = 30  
10 X 4 = 40  
10 X 5 = 50  
10 X 6 = 60  
10 X 7 = 70  
10 X 8 = 80  
10 X 9 = 90  
10 X 10 = 100
```



# Nested for loop in python

- Python allows us to nest any number of for loops inside a for loop.
- The inner loop is executed n number of times for every iteration of the outer loop.

## Syntax

```
for iterating_var1 in sequence:  
    for iterating_var2 in sequence:  
        #block of statements  
        #Other statements
```

```
n = int(input("Enter the number of rows you want to print?"))

i,j=0,0

for i in range(0,n):
    print()
    for j in range(0,i+1):
        print("*",end="")
```

## Output:

Enter the number of rows you want to print?5

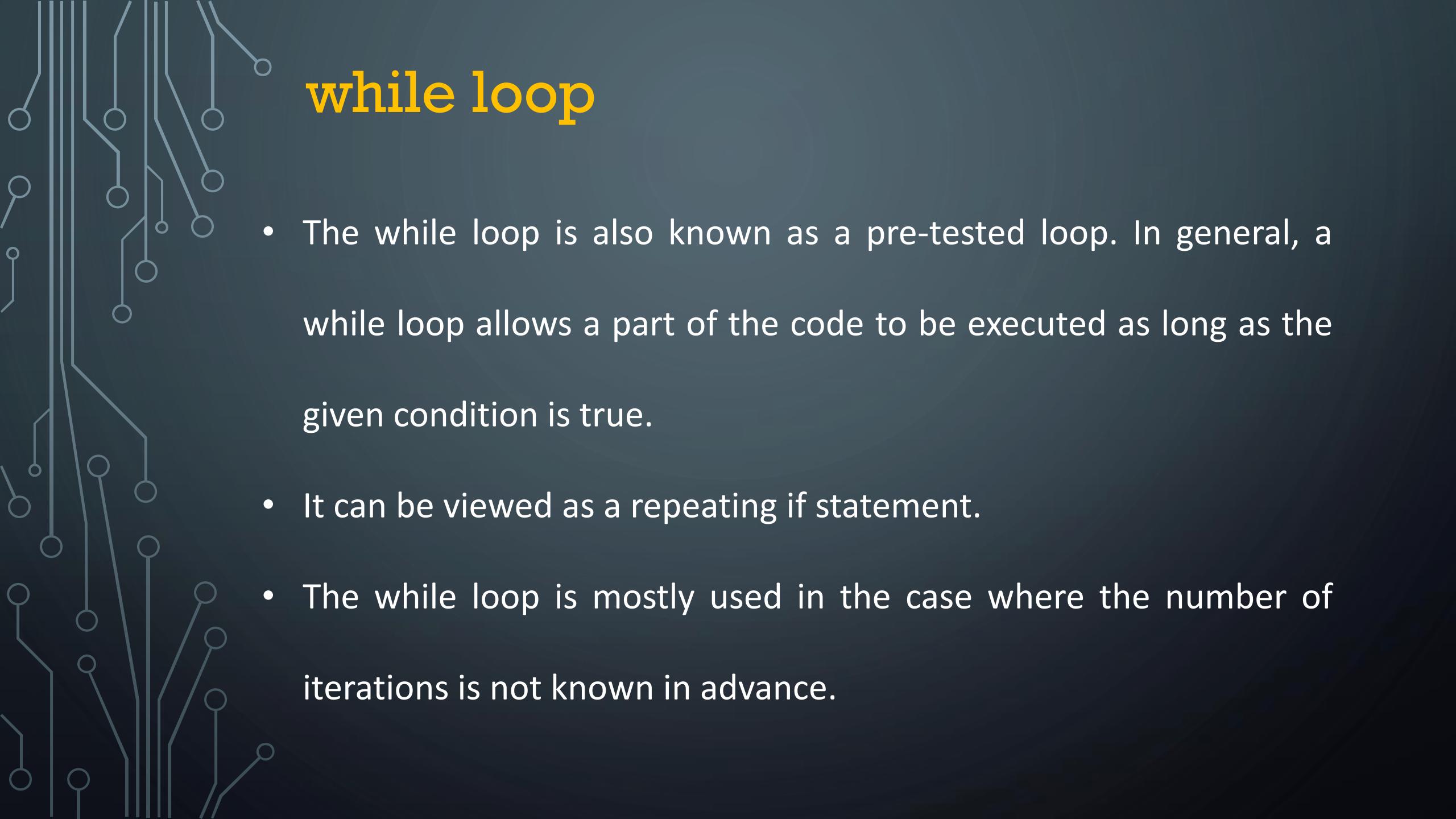
```
*
```

```
**
```

```
***
```

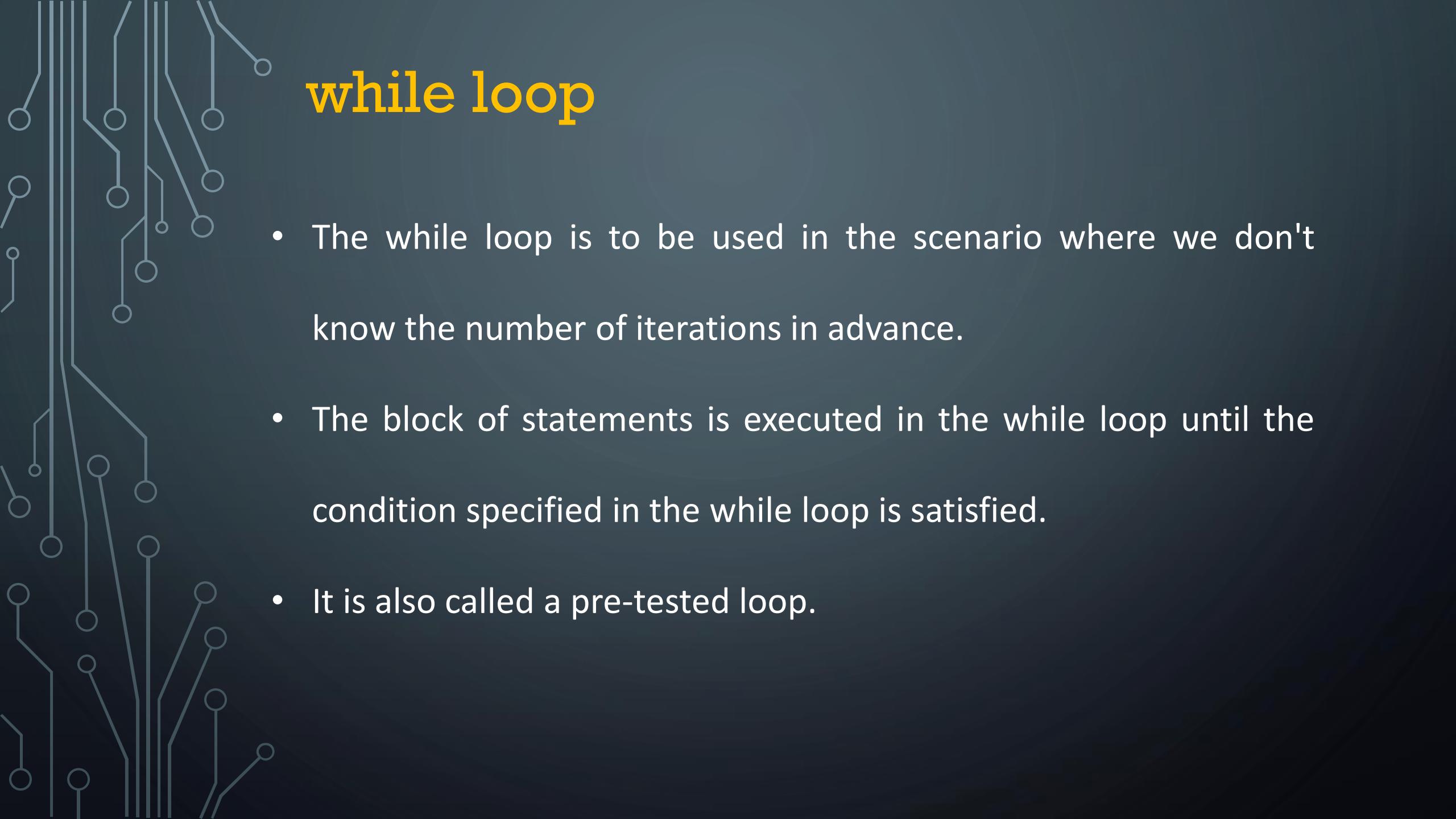
```
***
```

```
*****
```

A repeating pattern of light gray circuit board traces and component pads forms the background of the slide.

# while loop

- The while loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed as long as the given condition is true.
- It can be viewed as a repeating if statement.
- The while loop is mostly used in the case where the number of iterations is not known in advance.

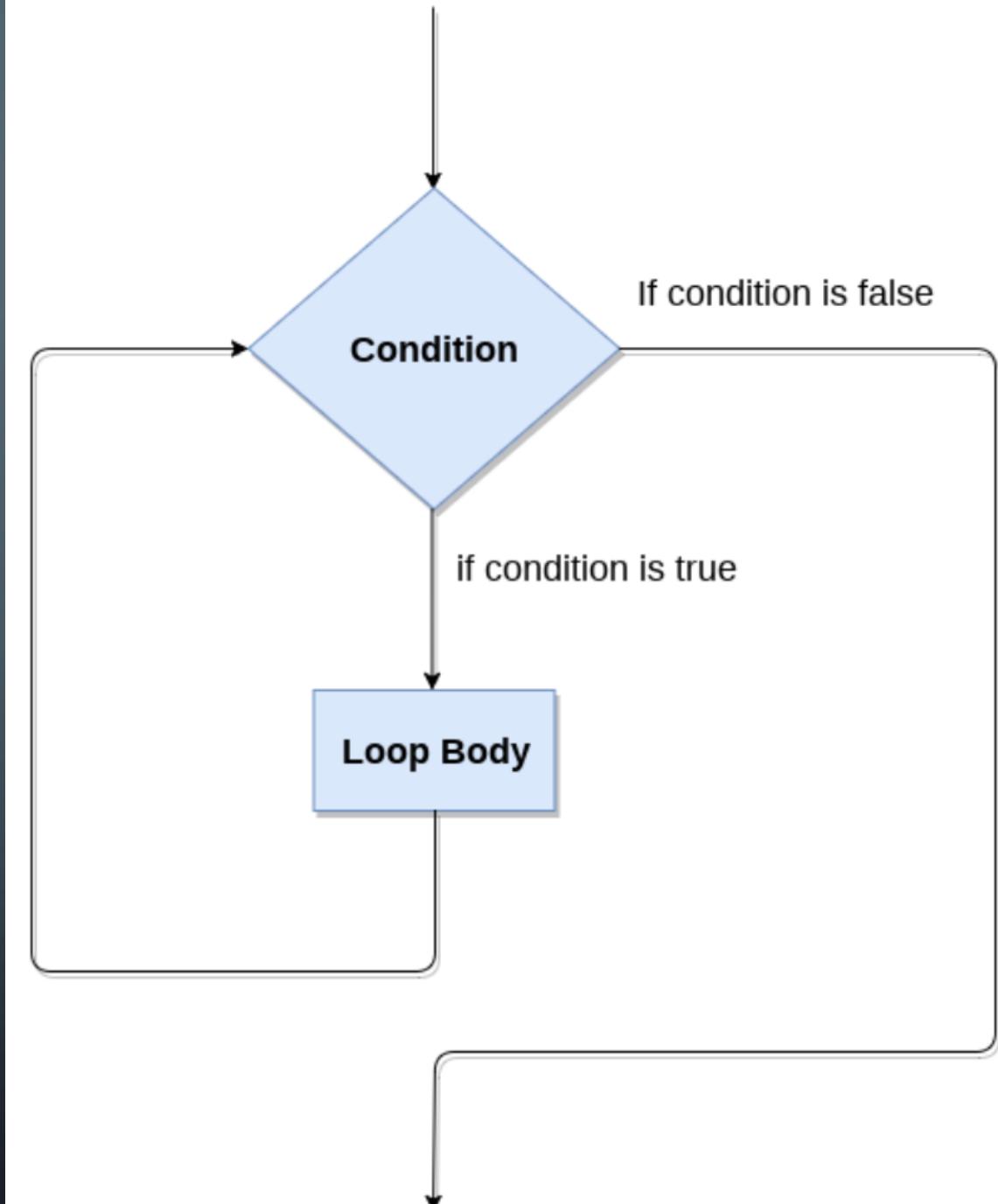


# while loop

- The while loop is to be used in the scenario where we don't know the number of iterations in advance.
- The block of statements is executed in the while loop until the condition specified in the while loop is satisfied.
- It is also called a pre-tested loop.

# while loop

while expression:  
statements





## Output:

# Example 1

```
i=1;  
while i<=10:  
    print(i);  
    i=i+1;
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

### Output:

Enter the number?10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

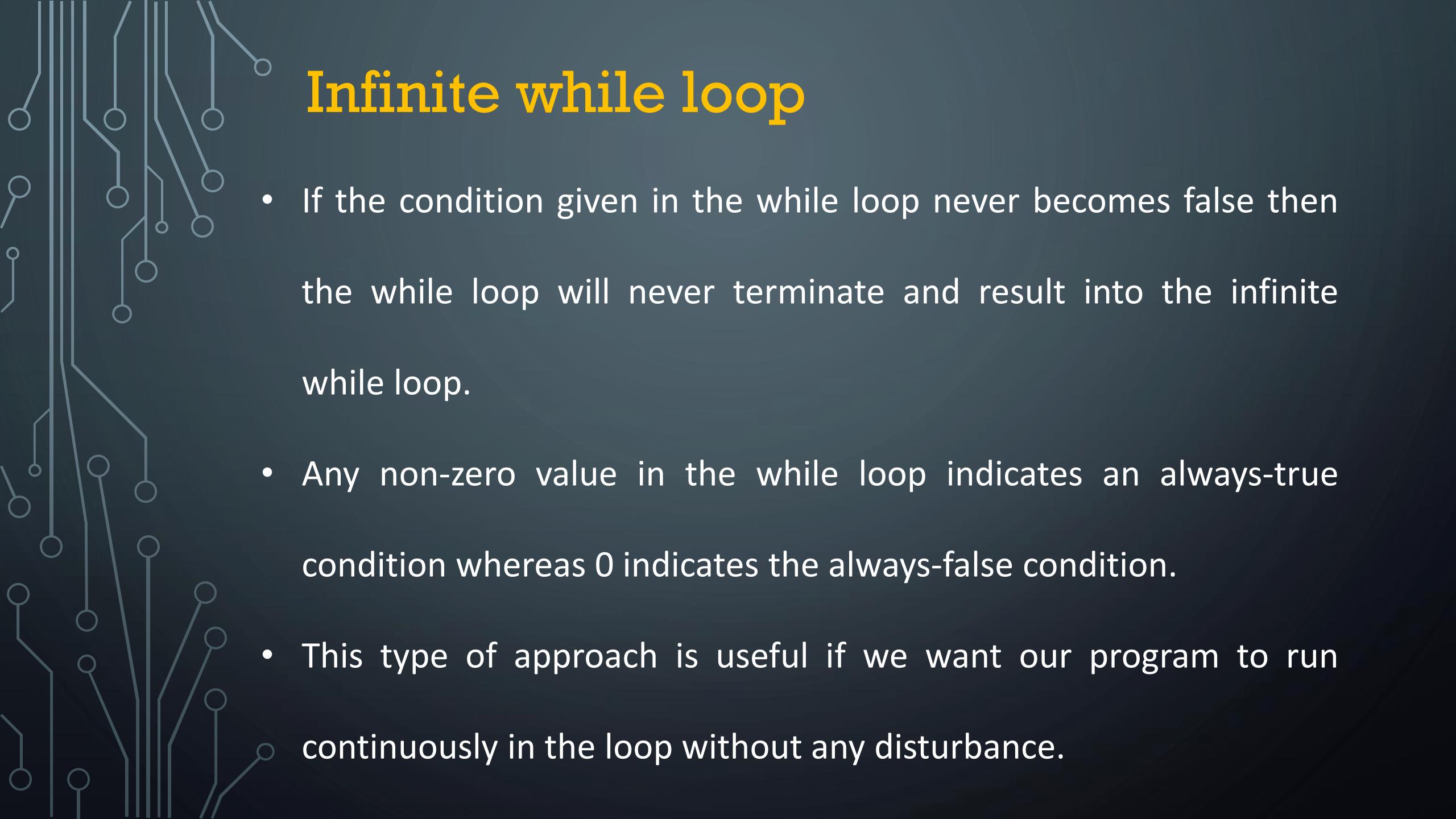
10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

## Example 2

```
i=1  
number=0  
b=9  
number = int(input("Enter the number?"))  
while i<=10:  
    print("%d X %d = %d \n"%(number,i,number*i));  
    i = i+1;
```

A faint, repeating pattern of a circuit board layout, consisting of vertical lines and small circular pads, serves as the background for the slide.

# Infinite while loop

- If the condition given in the while loop never becomes false then the while loop will never terminate and result into the infinite while loop.
- Any non-zero value in the while loop indicates an always-true condition whereas 0 indicates the always-false condition.
- This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

# Example 1

```
while (1):  
    print("Hi! we are inside the infinite while loop");
```

## Output:

```
Hi! we are inside the infinite while loop  
(infinite times)
```

Infinite  
while  
loop

```
var = 1  
while var != 2:  
    i = int(input("Enter the number?"))  
    print ("Entered value is %d"% (i))
```

### Output:

```
Enter the number?102  
Entered value is 102  
Enter the number?102  
Entered value is 102  
Enter the number?103  
Entered value is 103  
Enter the number?103  
(infinite loop)
```

## Example 2

```
str = "python"  
for i in str:  
    if i == 'o':  
        break  
    print(i);
```

### Output:

p  
y  
t  
h

```
i = 0;  
while 1:  
    print(i, " ",end=""),  
    i=i+1;  
    if i == 10:  
        break;  
print("came out of while loop");
```

## Output:

```
0 1 2 3 4 5 6 7 8 9  came out of while loop
```



# String

- strings can be created by enclosing the character or the sequence of characters in the quotes.
- Python allows us to use single quotes, double quotes, or triple quotes to create the string.

```
str = "Hi Python !"
```

```
print(type(str))
```



# String

- Strings are ordered text based data which are represented by enclosing the same in single/double/triple quotes.
- Such string manipulation patterns come up often in the context of data science work, and is one big perk of Python in this context.

# Indexing

- indexing of the python strings

starts from 0.

**str = "HELLO"**

H	E	L	L	O
0	1	2	3	4

**str[0] = 'H'**

**str[1] = 'E'**

**str[2] = 'L'**

**str[3] = 'L'**

**str[4] = 'O'**

# Slicing

- the slice operator [ ] is used to access the individual characters of the string.
- However, we can use the : (colon) operator in python to access the substring.

`str = "HELLO"`

H	E	L	L	O
0	1	2	3	4

`str[0] = 'H'`      `str[:] = 'HELLO'`

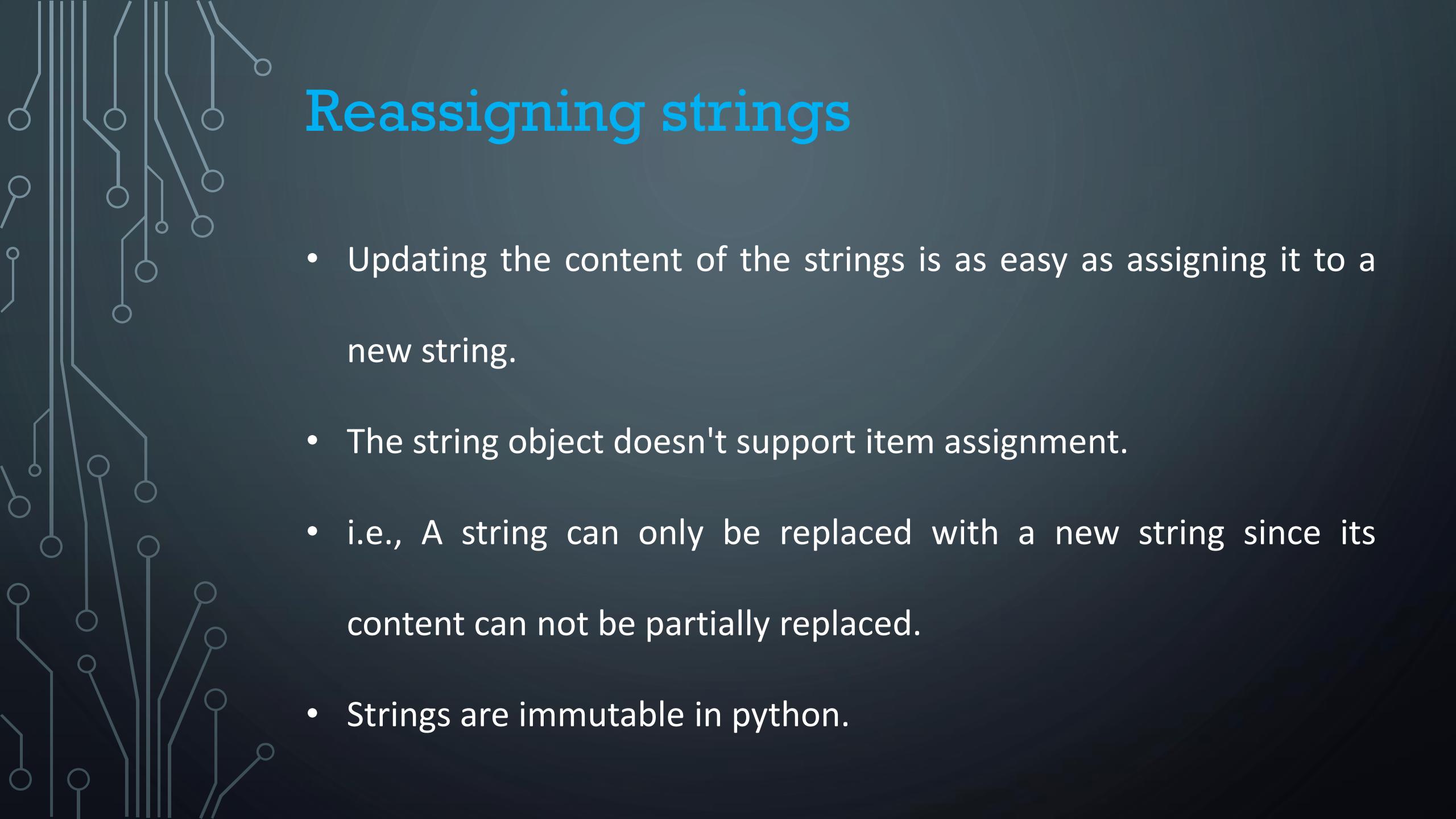
`str[1] = 'E'`      `str[0:] = 'HELLO'`

`str[2] = 'L'`      `str[:5] = 'HELLO'`

`str[3] = 'L'`      `str[:3] = 'HEL'`

`str[4] = 'O'`      `str[0:2] = 'HE'`

`str[1:4] = 'ELL'`



# Reassigning strings

- Updating the content of the strings is as easy as assigning it to a new string.
- The string object doesn't support item assignment.
- i.e., A string can only be replaced with a new string since its content can not be partially replaced.
- Strings are immutable in python.

## Example 1

```
str = "HELLO"  
str[0] = "h"  
print(str)
```

### Output:

```
Traceback (most recent call last):  
  File "12.py", line 2, in <module>  
    str[0] = "h";  
TypeError: 'str' object does not support item assignment
```

# String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

In [1]: ► str = "Hello"  
str1 = " world"

In [2]: ► print(str\*3) # prints HelloHelloHello  
HelloHelloHello

In [3]: ► print(str+str1) # prints Hello world  
Hello world

In [4]: ► print(str[4]) # prints o  
o

In [5]: ► print(str[2:4]); # prints ll  
ll

In [6]: ► `print('w' in str) # prints false as w is not present in str`  
False

In [7]: ► `print('wo' not in str1) # prints false as wo is present in str1.`  
False

In [8]: ► `print(r'C://python37') # prints C://python37 as it is written`  
C://python37

In [9]: ► `print("The string str : %s"%(str)) # prints The string str : Hello`  
The string str : Hello

In [ ]: ► |



# Python Formatting operator

- Python allows us to use the format specifiers used in C's printf statement.
- The format specifiers in python are treated in the same way as they are treated in C.
- However, Python provides an additional operator % which is used as an interface between the format specifiers and their values.
- We can say that it binds the format specifiers to the values.



# Formatting operator Example

```
Integer = 10;  
Float = 1.290  
String = "Ayush"  
print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... My value is %s"%  
(Integer,Float,String));
```

## Output:

```
Hi I am Integer ... My value is 10  
Hi I am float ... My value is 1.290000  
Hi I am string ... My value is Ayush
```



# Upper & Lower Case

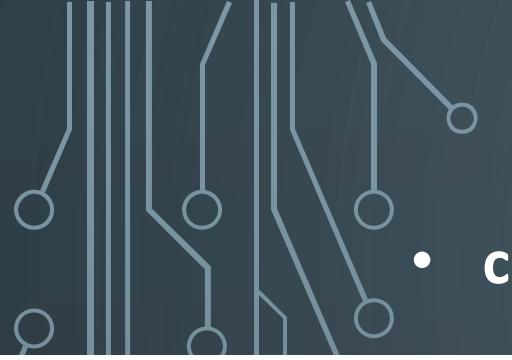
In [15]: ► `s = "MSC Integrated"  
s.lower()`

Out[15]: 'msc integrated'

In [16]: ► `s.upper()`

Out[16]: 'MSC INTEGRATED'

In [ ]: ► |



# Capitalize

- `capitalize( )` is used to capitalize the first element in the string.

In [20]: ►

```
s1 = 'marvel universe'  
s1.capitalize()  
# print(s1.capitalize())
```

Out[20]: 'Marvel universe'

In [23]: ►

```
s2 = 'universe'  
s2.capitalize()  
print(s2.capitalize())
```

Universe



# Replace

- `replace( )` function replaces the element with another element.

```
In [24]: ► S1 = 'marvel universe'  
          S1.replace('marvel', 'DC')
```

Out[24]: 'DC universe'

```
In [26]: ► S2 = 'Taj Mahal is beautiful'  
          S2.replace('Taj Mahal', 'Gujarat')
```

Out[26]: 'Gujarat is beautiful'



# Index of string

```
In [32]: # Searching for the first index of a substring  
S1 ='Taj Mahal is beautiful'  
S1.index('Mahal')
```

Out[32]: 4

```
In [33]: S2 ='Taj Mahal is beautiful'  
S2.index('is')
```

Out[33]: 10



# Startwith & Endswith

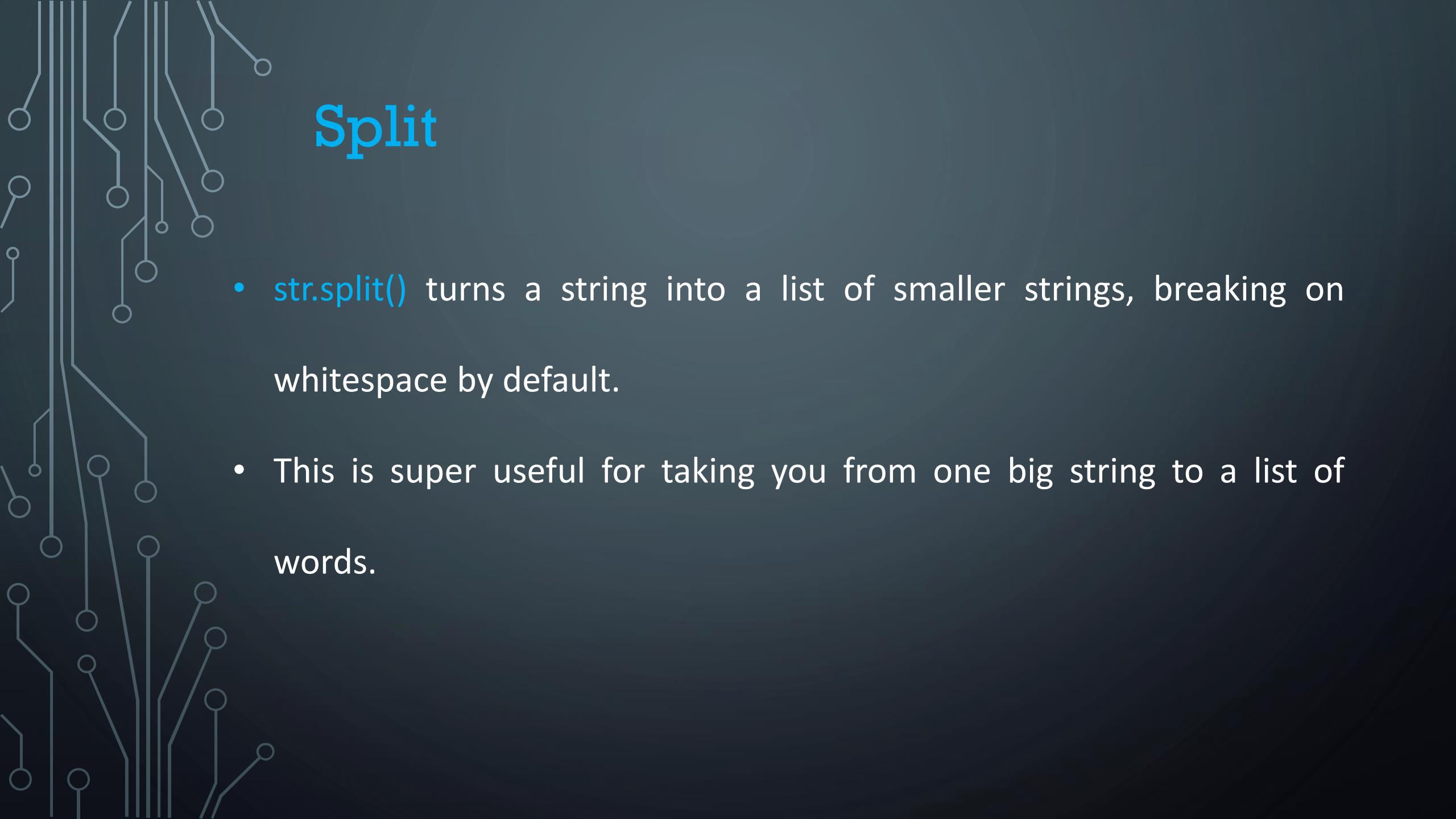
- `endswith()` & `startswith()` function is used to check if the given string starts or ends with the particular char which is given as input.

```
In [34]: ► S1 ='Taj Mahal is beautiful'  
          print(S1.startswith('Taj'))  
          print(S1.startswith('Mahal'))
```

True  
False

```
In [35]: ► S1 ='Taj Mahal is beautiful'  
          print(S1.endswith('beautiful'))  
          print(S1.endswith('Mahal'))
```

True  
False



# Split

- `str.split()` turns a string into a list of smaller strings, breaking on whitespace by default.
- This is super useful for taking you from one big string to a list of words.

# Split

```
In [36]: ┏━ S1 = 'marvel universe'  
         ┃ S1.split()
```

```
Out[36]: ['marvel', 'universe']
```

```
In [37]: ┏━ S1 ='Taj Mahal is beautiful'  
         ┃ S2.split()
```

```
Out[37]: ['Taj', 'Mahal', 'is', 'beautiful']
```

```
In [41]: ┏━ datestr = '1956-01-31'  
         ┃ year, month, day = datestr.split('-')
```

```
In [42]: ┏━ print(year)  
         ┃ print(month)  
         ┃ print(day)
```

```
1956  
01  
31
```



# Join

- **str.join()** takes us in the other direction, of the Split of the string.
- It can also join with other literals.

```
In [45]: █ '/'.join([month, day, year])
```

```
Out[45]: '01/31/1956'
```

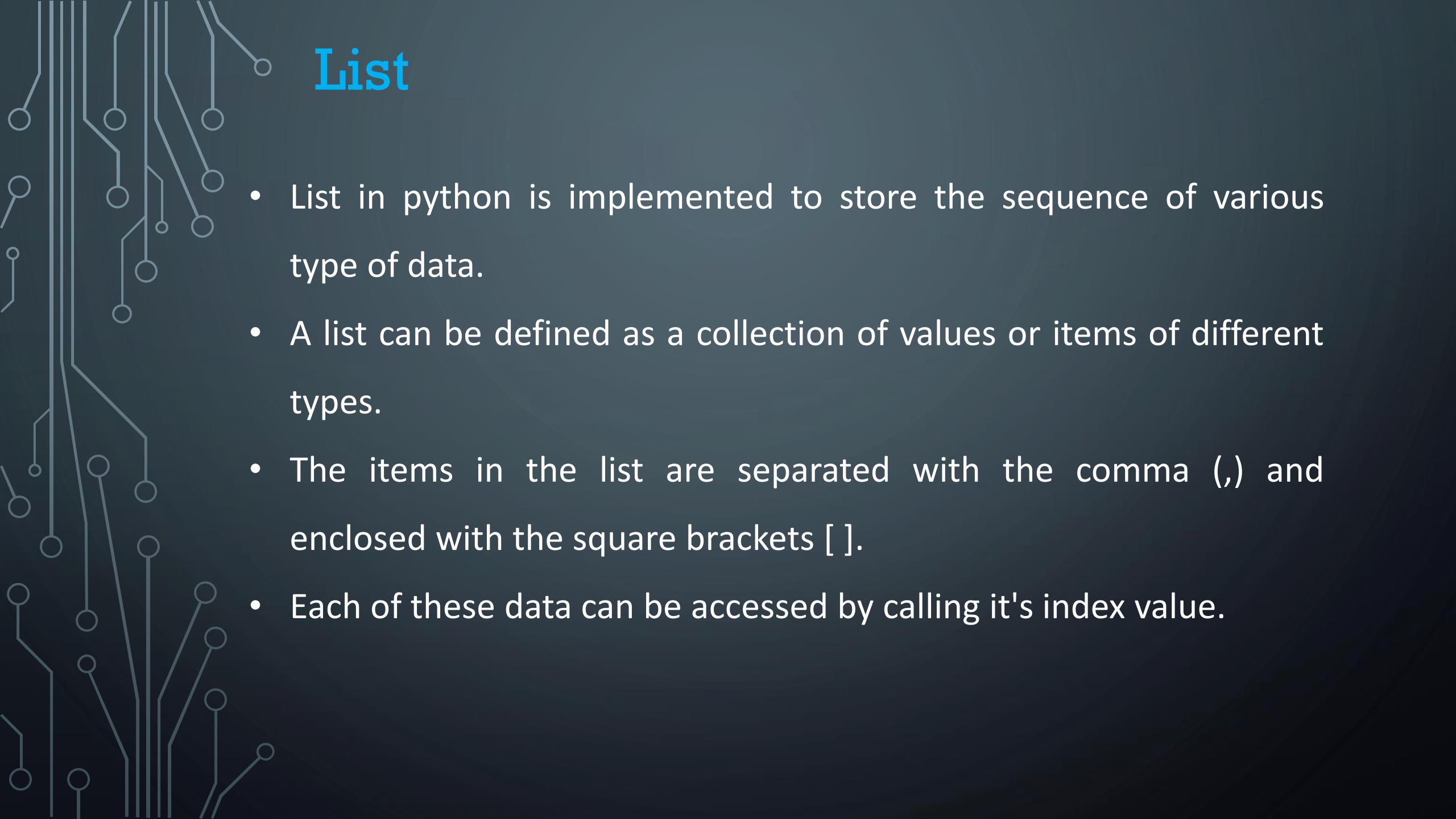
```
In [57]: █ S1 = 'marvel universe'  
S2 = S1.split()
```

```
In [59]: █ '👏 '.join(S1)
```

```
Out[59]: 'm👏 a👏 r👏 v👏 e👏 l👏 u👏 n👏 i👏 v👏 e👏 r👏 s👏 e👏 e'
```

```
In [60]: █ '👏 '.join(S2)
```

```
Out[60]: 'marvel👏 universe'
```



# List

- List in python is implemented to store the sequence of various type of data.
- A list can be defined as a collection of values or items of different types.
- The items in the list are separated with the comma (,) and enclosed with the square brackets [ ].
- Each of these data can be accessed by calling it's index value.



# Create & Type of List

```
In [1]: a = []
a
```

```
Out[1]: []
```

```
In [2]: b = list()
b
```

```
Out[2]: []
```

```
In [3]: print (type(a))
```

```
<class 'list'>
```



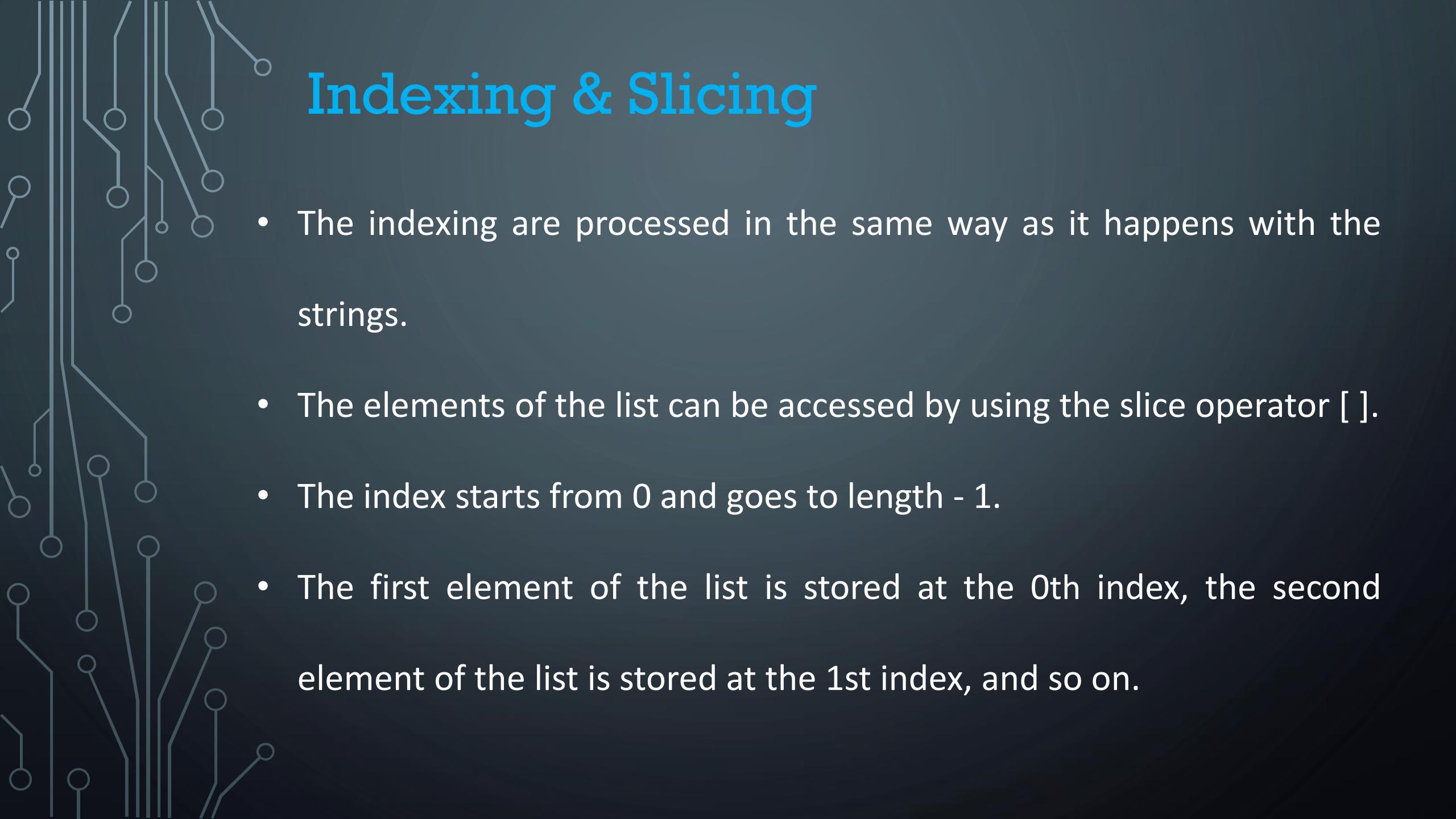
# List

- However, python contains all data types that are capable to store the sequences but the most common and reliable type is list.

```
L1 = ["RK", 102, "Gujarat", "1.6542467", "Mr. Kumar"]  
# print(L1)
```

```
L1
```

```
['RK', 102, 'Gujarat', '1.6542467', 'Mr. Kumar']
```



# Indexing & Slicing

- The indexing are processed in the same way as it happens with the strings.
- The elements of the list can be accessed by using the slice operator [ ].
- The index starts from 0 and goes to length - 1.
- The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.



# Indexing & Slicing

List = [ 0, 1, 2, 3, 4, 5]

0	1	2	3	4	5
---	---	---	---	---	---

List[0] = 0

List[0:] = [0,1,2,3,4,5]

List[1] = 1

List[:] = [0,1,2,3,4,5]

List[2] = 2

List[2:4] = [2, 3]

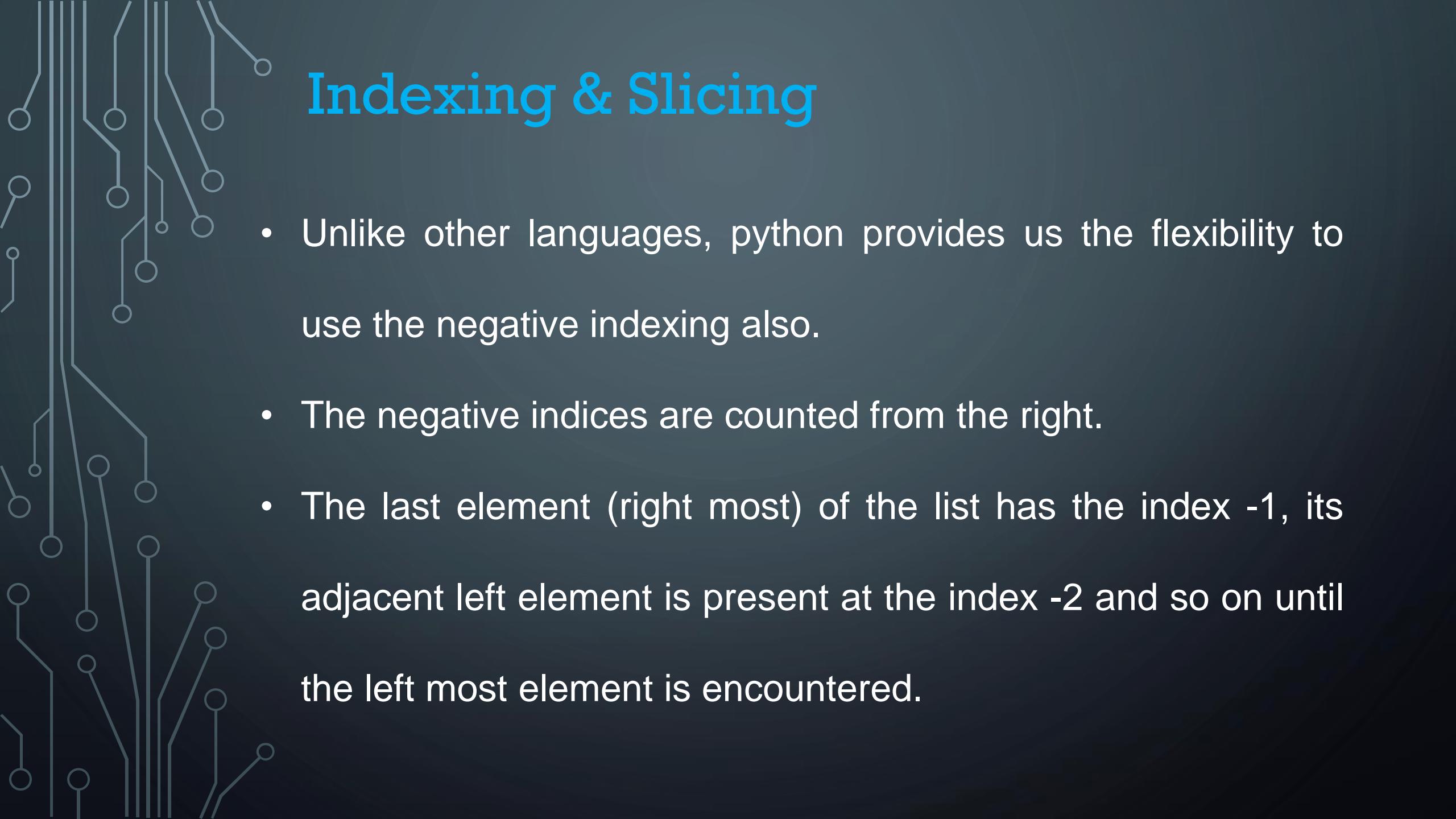
List[3] = 3

List[1:3] = [1, 2]

List[4] = 4

List[:4] = [0, 1, 2, 3]

List[5] = 5



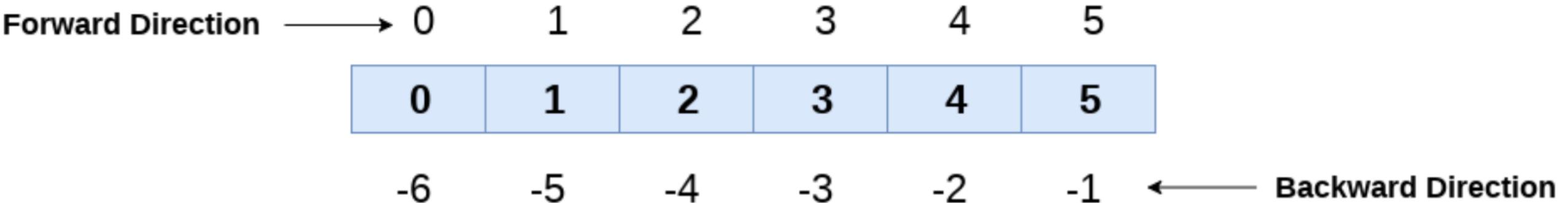
# Indexing & Slicing

- Unlike other languages, python provides us the **flexibility** to use the **negative indexing** also.
- The negative indices are counted from the right.
- The last element (right most) of the list has the index -1, its adjacent left element is present at the index -2 and so on until the left most element is encountered.



# Indexing

List = [ 0, 1, 2, 3, 4, 5]



In [12]: ► x = ['apple', 'orange', 'banana']

In [2]: ► x[-1]

Out[2]: 'banana'

In [3]: ► x[-2]

Out[3]: 'orange'

In [9]: ► x[:-1]

Out[9]: ['apple', 'orange']

In [15]: ► x[0:-1]

Out[15]: ['apple', 'orange']



# Updating List value

- Lists are the most versatile data structures in python since they are immutable and their values can be updated by using the slice and assignment operator.
- Python also provide us the `append()` method which can be used to add values to the string.

In [1]: ► List = [1, 2, 3, 4, 5, 6]  
print(List)

[1, 2, 3, 4, 5, 6]

In [2]: ► List[2] = 10;  
print(List)

[1, 2, 10, 4, 5, 6]

In [3]: ► List[1:3] = [89, 78]  
print(List)

[1, 89, 78, 4, 5, 6]

In [4]: ► List.append(100)  
print(List)

[1, 89, 78, 4, 5, 6, 100]

# Delete & Remove & Pop

- The list elements can also be deleted by using the `del` keyword.
- Python also provides us the `remove()` method if we do not know which element is to be deleted from the list.
- `pop( )` is used to remove element based on it's index value which can be assigned to a variable. One can also remove element by specifying the element itself using the `remove( )` function.

```
In [11]: ► List = [1, 2, 3, 4, 5, 6]
```

```
In [12]: ► del List[0]  
print(List)
```

```
[2, 3, 4, 5, 6]
```

```
In [13]: ► del List[3]  
print(List)
```

```
[2, 3, 4, 6]
```

```
In [14]: ► List.remove(4)  
print(List)
```

```
[2, 3, 6]
```

```
In [16]: ► List.pop(2)  
print(List)
```

```
[2, 3]
```

```
In [18]: ► del List  
print(List)
```

```
NameError
```

```
Traceback (most recent call last)
```

```
<ipython-input-18-b915620d9aa9> in <module>
```

```
----> 1 del List
```

```
    2 print(List)
```

```
NameError: name 'List' is not defined
```

# Python List Operations

- The concatenation (+) and repetition (\*) operator work in the same way as they were working with the strings.
- Consider a List **`l1 = [1, 2, 3, 4]`, and `l2 = [5, 6, 7, 8]`**

<b>Operator</b>	<b>Description</b>	<b>Example</b>
Repetition	The repetition operator enables the list elements to be repeated multiple times.	<code>L1*2 = [1, 2, 3, 4, 1, 2, 3, 4]</code>
Concatenation	It concatenates the list mentioned on either side of the operator.	<code>l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8]</code>
Membership	It returns true if a particular item exists in a particular list otherwise false.	<code>print(2 in l1) prints True.</code>
Iteration	The for loop is used to iterate over the list elements.	<pre>for i in l1:     print(i)</pre> <p><b>Output</b></p> <pre>1 2 3 4</pre>
Length	It is used to get the length of the list	<code>len(l1) = 4</code>

# Iteration of List

- A list can be iterated by using a for - in loop.
- You can get split, slice and other operations perform by using iterating list.

```
In [1]: ► List = [1, 2, 3, 4, 5, 6]
```

```
In [2]: ► #i will iterate over the elements of the List and contains each element in each iteration.  
for i in List:  
    print(i)
```

```
1  
2  
3  
4  
5  
6
```

```
In [3]: ► planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']  
for i in planets:  
    print(i)
```

```
Mercury  
Venus  
Earth  
Mars  
Jupiter  
Saturn  
Uranus  
Neptune
```



# Append & Extend

- Python provides `append()` function by using which we can add an element to the list.
- However, the `append()` method can only add the value to the end of the list.
- But if nested list is not what is desired then `extend()` function can be used.

In [7]: ► List1 = [1, 2, 3, 4, 5, 6]  
List2 = [7,8]

In [8]: ► List1.append(9)  
print(List1)

[1, 2, 3, 4, 5, 6, 9]

In [9]: ► List1.append(List2)  
print(List1)

[1, 2, 3, 4, 5, 6, 9, [7, 8]]

In [10]: ► List3 = [10,11]

In [11]: ► List1.extend(List3)  
print(List1)

[1, 2, 3, 4, 5, 6, 9, [7, 8], 10, 11]



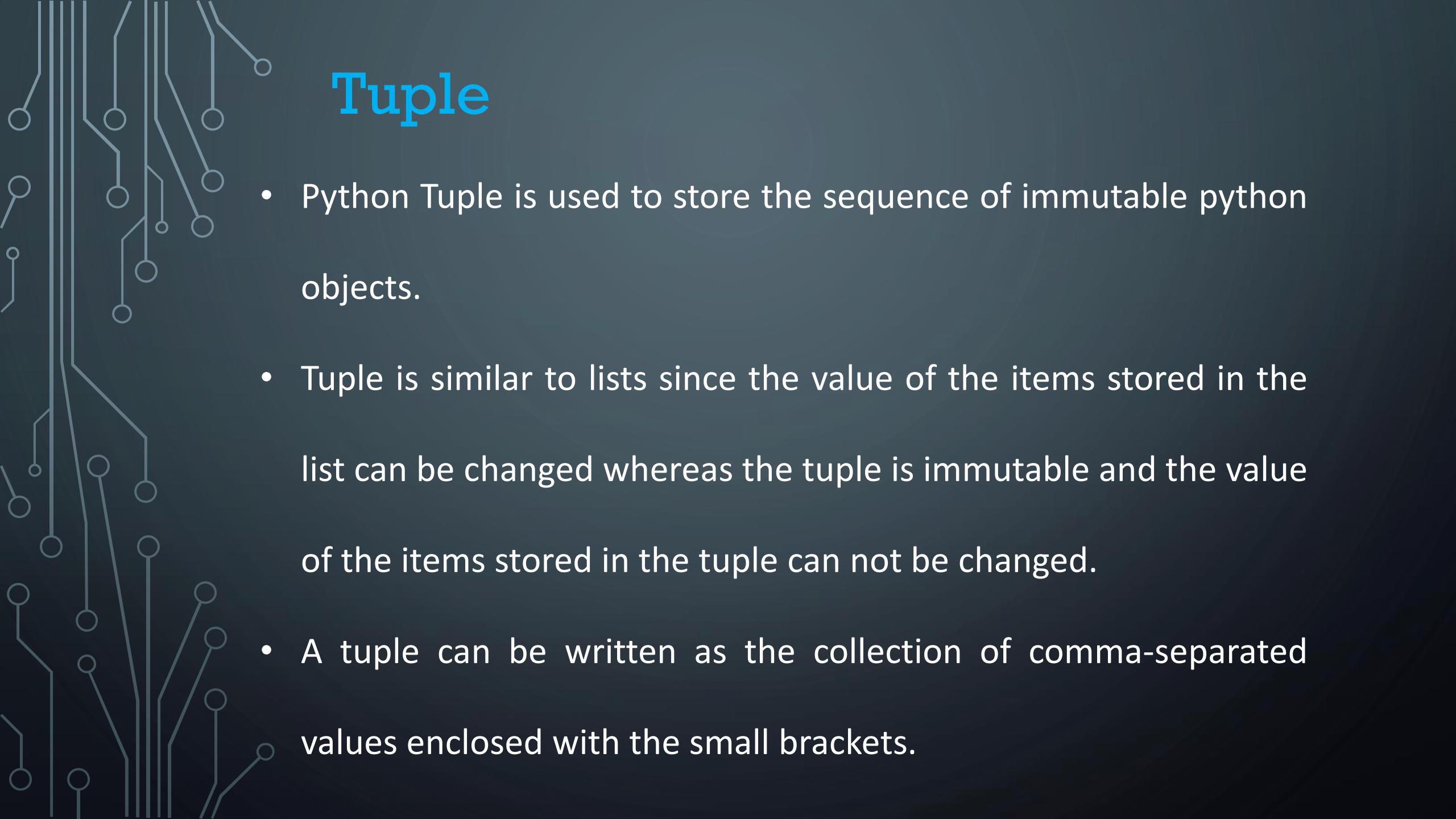
# Python in-built Functions

SN	Function	Description
1	cmp(list1, list2)	It compares the elements of both the lists.
2	len(list)	It is used to calculate the length of the list.
3	max(list)	It returns the maximum element of the list.
4	min(list)	It returns the minimum element of the list.
5	list(seq)	It converts any sequence to the list.



# Python in-built Methods

SN	Function	Description
1	<code>list.append(obj)</code>	The element represented by the object obj is added to the list.
2	<code>list.clear()</code>	It removes all the elements from the list.
3	<code>List.copy()</code>	It returns a shallow copy of the list.
4	<code>list.count(obj)</code>	It returns the number of occurrences of the specified object in the list.
5	<code>list.extend(seq)</code>	The sequence represented by the object seq is extended to the list.
6	<code>list.index(obj)</code>	It returns the lowest index in the list that object appears.
7	<code>list.insert(index, obj)</code>	The object is inserted into the list at the specified index.
8	<code>list.pop(obj=list[-1])</code>	It removes and returns the last object of the list.
9	<code>list.remove(obj)</code>	It removes the specified object from the list.
10	<code>list.reverse()</code>	It reverses the list.
11	<code>list.sort([func])</code>	It sorts the list by using the specified compare function if given.



# Tuple

- Python Tuple is used to store the sequence of immutable python objects.
- Tuple is similar to lists since the value of the items stored in the list can be changed whereas the tuple is immutable and the value of the items stored in the tuple can not be changed.
- A tuple can be written as the collection of comma-separated values enclosed with the small brackets.

- Tuples can be created two ways.

```
In [1]: ➤ t = ()  
print(t)
```

```
()
```

```
In [2]: ➤ t2 = tuple()  
print(t2)
```

```
()
```

In [3]: ► tuple1 = (10, 20, 30, 40, 50, 60)  
print(tuple1)

(10, 20, 30, 40, 50, 60)

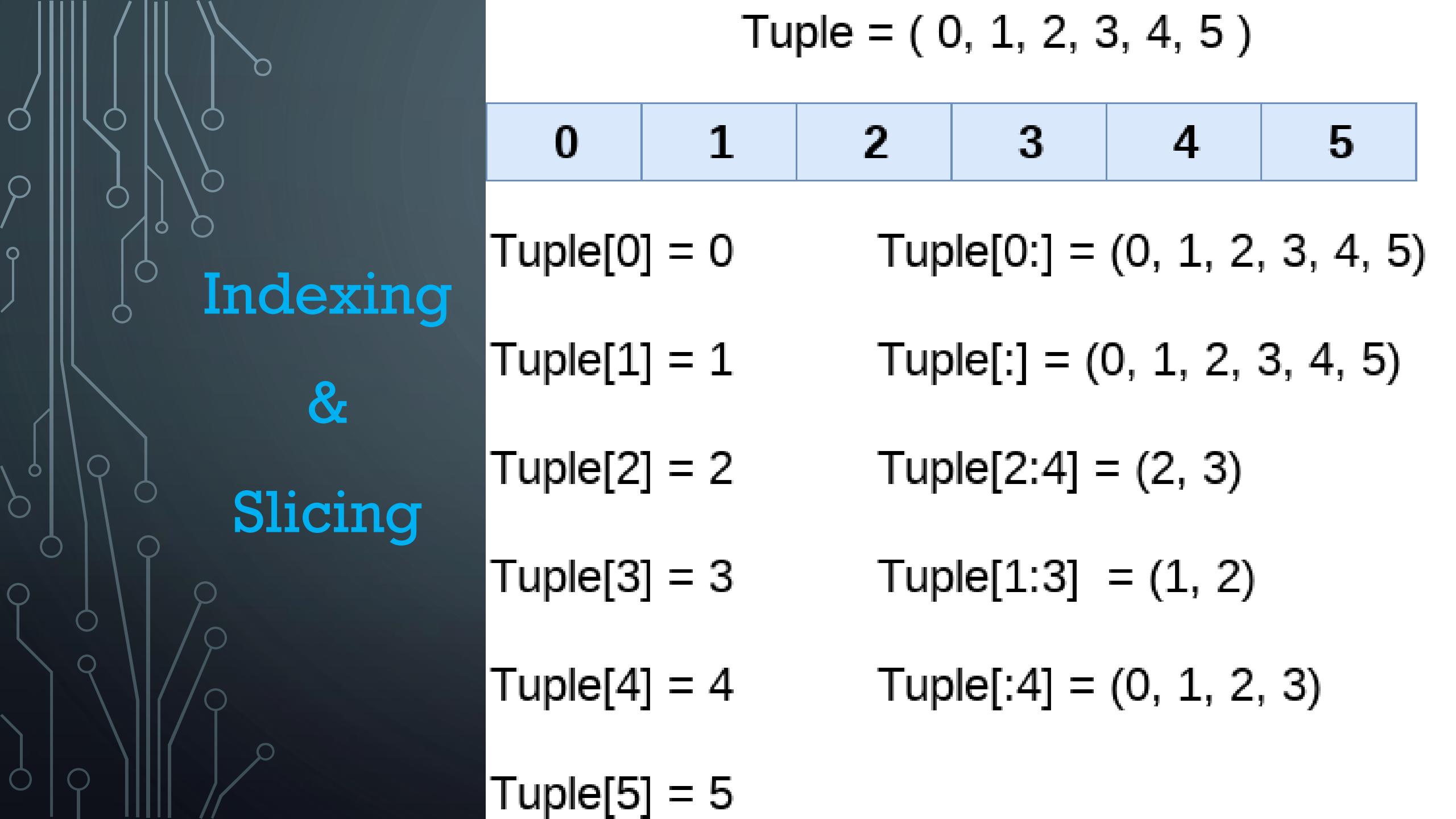
In [4]: ► count = 0  
for i in tuple1:  
 print("tuple1[%d] = %d"%(count, i));

tuple1[0] = 10  
tuple1[0] = 20  
tuple1[0] = 30  
tuple1[0] = 40  
tuple1[0] = 50  
tuple1[0] = 60



# Indexing & Slicing

- The indexing and slicing in tuple are similar to lists.
- The indexing in the tuple starts from 0 and goes to `length(tuple) - 1`.
- The items in the tuple can be accessed by using the slice operator.
- Python also allows us to use the colon operator to access multiple items in the tuple.



# Indexing & Slicing

`Tuple = ( 0, 1, 2, 3, 4, 5 )`

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
----------	----------	----------	----------	----------	----------

`Tuple[0] = 0`

`Tuple[0:] = (0, 1, 2, 3, 4, 5)`

`Tuple[1] = 1`

`Tuple[:] = (0, 1, 2, 3, 4, 5)`

`Tuple[2] = 2`

`Tuple[2:4] = (2, 3)`

`Tuple[3] = 3`

`Tuple[1:3] = (1, 2)`

`Tuple[4] = 4`

`Tuple[:4] = (0, 1, 2, 3)`

`Tuple[5] = 5`



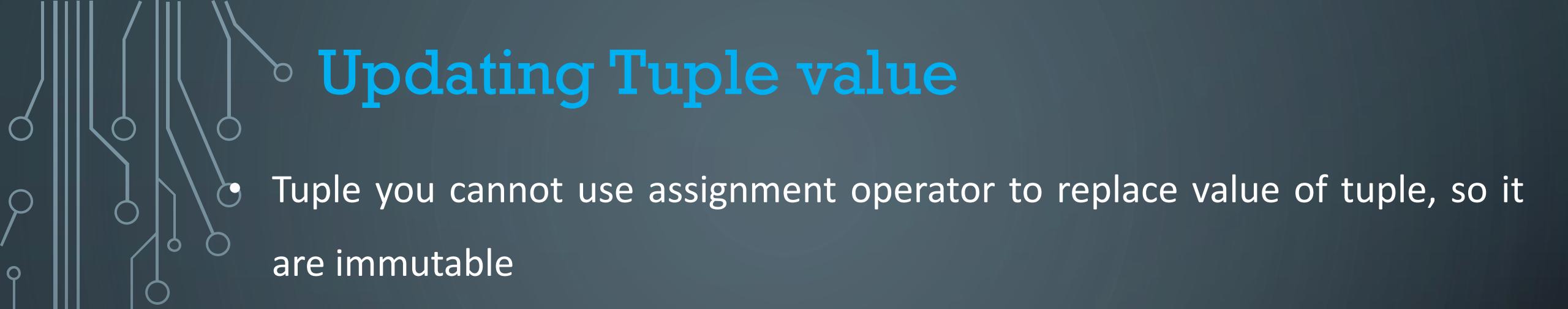
# Iteration of Tuple

```
In [6]: ➜ t = (1, 2, 3, 4, 5)  
      for i in t:  
          print(i)
```

```
1  
2  
3  
4  
5
```

```
In [7]: ➜ t = (1, 2.3, 'spam')  
      for i in t:  
          print(i)
```

```
1  
2.3  
spam
```



# Updating Tuple value

- Tuple you cannot use assignment operator to replace value of tuple, so it are immutable

```
In [8]: ┏ t = (1, 2, 3, 4, 5)  
      └ print(t)
```

```
(1, 2, 3, 4, 5)
```

```
In [9]: ┏ t[2] = 10
```

```
-----  
TypeError
```

```
<ipython-input-9-07ce4274e407> in <module>  
----> 1 t[2] = 10
```

```
Traceback (most recent call last)
```

```
TypeError: 'tuple' object does not support item assignment
```



# Mapping Tuple

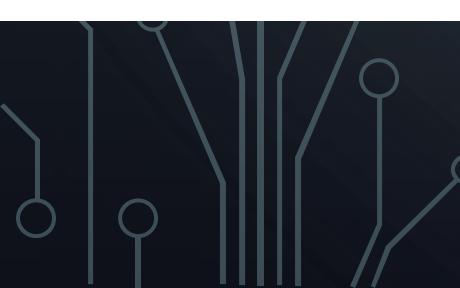
```
In [1]: ➤ (a,b,c)= ('alpha','beta','gamma')
```

```
In [2]: ➤ print (a,b,c)
```

```
alpha beta gamma
```

```
In [3]: ➤ d = tuple('Marvel Universe')  
print (d)
```

```
('M', 'a', 'r', 'v', 'e', 'l', ' ', 'U', 'n', 'i', 'v', 'e', 'r', 's', 'e')
```





# Count

- **count()** function counts the number of specified element that is present in the tuple.

In [3]: ➤ `d = tuple('Marvel Universe')  
print (d)`

```
('M', 'a', 'r', 'v', 'e', 'l', ' ', 'U', 'n', 'i', 'v', 'e', 'r', 's', 'e')
```

In [4]: ➤ `d.count('a')`

Out[4]: 1

In [5]: ➤ `d.count('e')`

Out[5]: 3



# Index

- **index()** function returns the index of the specified element.
- If the elements are more than one then the index of the first element of that specified element is returned.

In [3]: ► `d = tuple('Marvel Universe')  
print (d)`

```
('M', 'a', 'r', 'v', 'e', 'l', ' ', 'U', 'n', 'i', 'v', 'e', 'r', 's', 'e')
```

In [6]: ► `d.index('a')`

Out[6]: 1

In [7]: ► `d.index('e')`

Out[7]: 4

# Divmod

`divmod()` is a built-in function that takes two (non-complex) numbers as arguments and returns a pair of numbers consisting of their quotient and remainder when using integer division.

In [8]: ► xyz = divmod(10,3)  
print (xyz)  
print (type(xyz))

(3, 1)  
<class 'tuple'>

In [11]: ► xyz = divmod(100,4)  
print (xyz)

(25, 0)

In [12]: ► divmod(10,7)

Out[12]: (1, 3)

In [13]: ► divmod(13.5,7.8)

Out[13]: (1.0, 5.7)

# Tuple Operations

- The operators like concatenation (+), repetition (\*), Membership (in) works in the same way as they work with the list.
- Consider Tuple  $t = (1, 2, 3, 4, 5)$  and Tuple  $t1 = (6, 7, 8, 9)$  are declared.

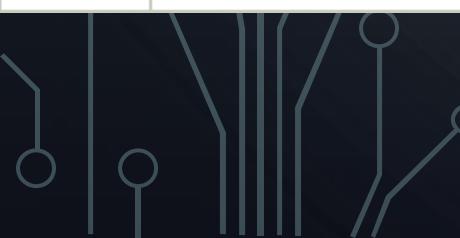
# Tuple Operations

Operator	Description	Example
Repetition	The repetition operator enables the tuple elements to be repeated multiple times.	$T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)$
Concatenation	It concatenates the tuple mentioned on either side of the operator.	$T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9)$
Membership	It returns true if a particular item exists in the tuple otherwise false.	<code>print (2 in T1) prints True.</code>
Iteration	The for loop is used to iterate over the tuple elements.	<code>for i in T1:     print(i)</code> <b>Output</b> 1 2 3 4 5
Length	It is used to get the length of the tuple.	<code>len(T1) = 5</code>



# Tuple inbuilt functions

SN	Function	Description
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.
2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple.
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.





# Nested Tuple

- You can create new tuple using another tuple.

In [15]: ►

```
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'Java')
```

In [16]: ►

```
Tuple3 = (Tuple1, Tuple2)
print(Tuple3)
```

((0, 1, 2, 3), ('python', 'Java'))

# List v/s Tuple

List	Tuple
The literal syntax of list is shown by the [ ].	The literal syntax of the tuple is shown by the ( ).
The List is mutable.	The tuple is immutable.
The List has the variable length.	The tuple has the fixed length.
The list provides more functionality than tuple.	The tuple provides less functionality than the list.

# List v/s Tuple

List	Tuple
<p>The list is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed.</p>	<p>The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary.</p>
<p>Lists consume more memory.</p>	<p>Tuples consume less memory as compared to the list.</p>

# List v/s Tuple

List	Tuple
The list is better for performing operations, such as insertion and deletion.	Tuple data type is appropriate for accessing the elements. No insertion operation.
Lists have several built-in methods	Tuple does not have many built-in methods.
Example: <code>List = [1,'RK','2.56','Marvel Universe']</code>	Example: <code>Tuple1 = (0, 1, 2.65, 'python', 'Java')</code>



# Set

- Python, **Set** is an unordered collection of data type that is iterable, mutable and has no duplicate elements.
- **Set** in Python is equivalent to sets in mathematics.
- The elements of the set can not be duplicate.
- The set can be created by enclosing the comma separated items with the curly braces.

- Sets are declared as `set()` which will initialize a empty set.
- Also `set([sequence])` can be executed to declare a set with elements.

In [2]: ► `s1 = set() # Empty Set  
print(s1)  
print (type(s1))`

`set()  
<class 'set'>`

In [3]: ► `s2 = set([1,2,2,3,3,4])  
print (s2)`

`{1, 2, 3, 4}`

In [4]: ► `l = set([1,2,3,3,4,5,5,6,900])  
l`

Out[4]: `{1, 2, 3, 4, 5, 6, 900}`



# Set operations

- we can perform various mathematical operations on python sets like union, intersection, difference, etc.



## Adding items to the set

- Python provides the `add()` method which can be used to add some particular item to the set.

In [1]: ► `S1 = set()  
print(S1)`

set()

In [4]: ► `S1.add(1)  
S1.add(2)  
S1.add(3)  
S1.add((4,5))  
print(S1)`

{(4, 5), 1, 2, 3}

In [5]: ► `S2 = set()  
print(S2)`

set()

In [7]: ► *# You can also add using for Loop*  
`for i in range(1, 6):  
 S2.add(i)  
print(S2)`

{1, 2, 3, 4, 5}



# Update

- Addition of two or more elements `Update()` method is used.
- The `update()` method accepts lists, strings, tuples as well as other sets as its arguments. In all of these cases, duplicate elements are avoided.
- Lists cannot be added to a set as elements because Lists are not hashable whereas Tuples can be added because tuples are immutable and hence `Hashable(object(member or element) value never change)`.

In [11]: ► `s1 = {1,2,3,4,5}  
print(s1)`

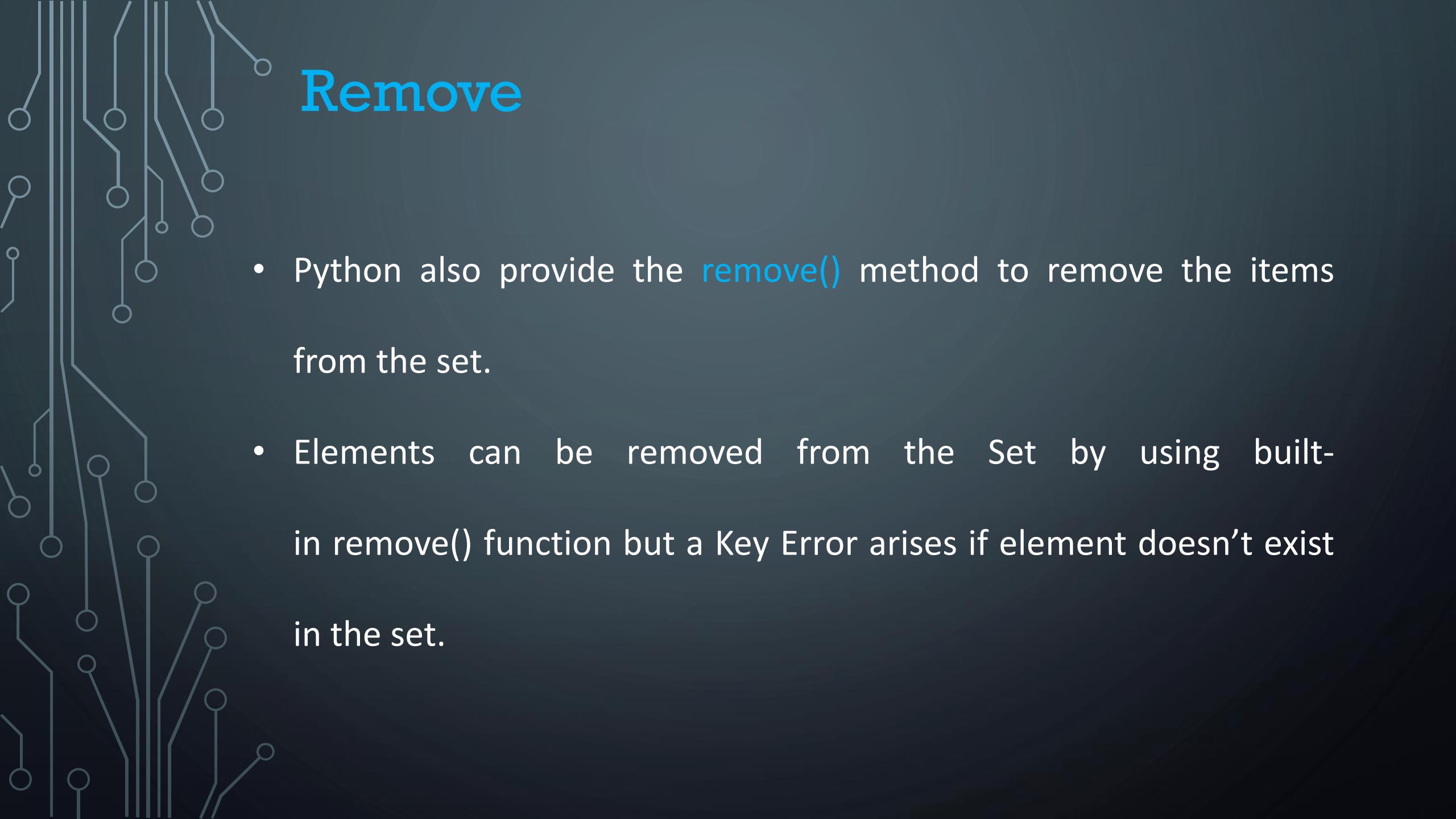
{1, 2, 3, 4, 5}

In [12]: ► `s1.update([6,7])  
print(s1)`

{1, 2, 3, 4, 5, 6, 7}

In [14]: ► `list = [8,9,10]  
s1.update(list)  
print(s1)`

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}



# Remove

- Python also provide the `remove()` method to remove the items from the set.
- Elements can be removed from the Set by using built-in `remove()` function but a Key Error arises if element doesn't exist in the set.

In [36]: ► `set1 = set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
print(set1)`

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [37]: ► `for i in range(1, 5):  
 set1.remove(i)  
print(set1)`

{5, 6, 7, 8, 9, 10, 11, 12}

In [38]: ► `set1.remove(6)  
set1.remove(7)`

In [39]: ► `print(set1)`

{5, 8, 9, 10, 11, 12}



# Discard

- To remove elements from a set without Key Error, use `discard()`, if the element doesn't exist in the set.
- Python provides `discard()` method which can be used to remove the items from the set.



## Discard

In [19]: ► `set1 = set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
print(set1)`

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [20]: ► `set1.discard(8)  
set1.discard(9)  
print(set1)`

{1, 2, 3, 4, 5, 6, 7, 10, 11, 12}



# Pop

- We can also use the `pop()` method to remove the item.
- This method will always remove the last(some times random) item.



In [1]: ► `set1 = set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
print(set1)`

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [2]: ► `set1.pop()  
print(set1)`

{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [6]: ► `Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}  
print(Days)`

{'Monday', 'Friday', 'Wednesday', 'Tuesday', 'Saturday', 'Sunday', 'Thursday'}

In [7]: ► `print(Days.pop())`

Monday



# Clear

- To remove all the elements from the set, `clear()` function is used.

```
In [11]: ► Months = set(["January", "February", "March", "April", "May", "June"])
         print(Months)
         Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
         print(Days)
```

```
{'April', 'February', 'June', 'March', 'January', 'May'}
{'Monday', 'Friday', 'Wednesday', 'Tuesday', 'Saturday', 'Sunday', 'Thursday'}
```

```
In [12]: ► Months.clear()
         print(Months)
```

```
set()
```

```
In [13]: ► Days.clear()
         print(Days)
```

```
set()
```

# Difference between discard() and remove()

- Despite the fact that discard() and remove() method both perform the same task, There is one main difference between discard() and remove().
- If the key to be deleted from the set using discard() doesn't exist in the set, the python will not give the error. The program maintains its control flow.
- On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the python will give the error.

```
In [26]: ► Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}  
print(Days)
```

```
{'Monday', 'Friday', 'Wednesday', 'Tuesday', 'Saturday', 'Sunday', 'Thursday'}
```

```
In [27]: ► Days.discard("Fri")  
print(Days)
```

```
{'Monday', 'Friday', 'Wednesday', 'Tuesday', 'Saturday', 'Sunday', 'Thursday'}
```

```
In [25]: ► Days.remove("Tue")  
print(Days)
```

```
-----  
KeyError                                                 Traceback (most recent call last)  
<ipython-input-25-07ded8260d11> in <module>  
----> 1 Days.remove("Tue")  
      2 print(Days)
```

```
KeyError: 'Tue'
```

# Union of two Sets

- The union of two sets are calculated by using the or (|) operator.
- The union of the two sets contains the all the items that are present in both the sets.
- Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
- Days2 = {"Friday","Saturday","Sunday"}

In [1]: ► Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}  
print(Days1)  
Days2 = {"Friday", "Saturday", "Sunday"}  
print(Days2)

```
{'Monday', 'Thursday', 'Wednesday', 'Tuesday'}  
'Friday', 'Saturday', 'Sunday'
```

## Method:-1 using union | operator

In [2]: ► print(Days1 | Days2)

```
{'Friday', 'Monday', 'Tuesday', 'Saturday', 'Wednesday', 'Sunday', 'Thursday'}
```

## Method:-2 using union() method

In [3]: ► print(Days1.union(Days2))

```
{'Friday', 'Monday', 'Tuesday', 'Saturday', 'Wednesday', 'Sunday', 'Thursday'}
```



# Intersection of two sets

- The & (intersection) operator is used to calculate the intersection of the two sets in python.
- The intersection of the two sets are given as the set of the elements that common in both sets.
- `set1 = {"Ayush", "John", "David", "Martin"}`
- `set2 = {"Steve", "Milan", "David", "Martin"}`

In [4]: ►

```
set1 = {"RK", "Abhay", "Riva", "Mansi"}  
print(set1)  
set2 = {"Yana", "Mayur", "RK", "Mansi"}  
print(set2)
```

```
{'Abhay', 'Riva', 'Mansi', 'RK'}  
'Mayur', 'Mansi', 'Yana', 'RK'
```

## Method:-1 using & operator

In [5]: ►

```
print(set1&set2)
```

```
{'Mansi', 'RK'}
```

## Method:-2 using intersection() method

In [6]: ►

```
print(set1.intersection(set2))
```

```
{'Mansi', 'RK'}
```



# The intersection\_update() method

- The `intersection_update()` method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
- The `Intersection_update()` method is different from `intersection()` method since it modifies the original set by removing the unwanted items, on the other hand, `intersection()` method returns a new set.<sup>3</sup>

```
In [8]: ► a = {"RK", "Abhay", "Mayur"}  
      print(a)  
      b = {"Mansi", "RK", "Yana"}  
      print(b)  
      c = {"Neev", "RK", "Riva"}  
      print(c)
```

```
{'Abhay', 'Mayur', 'RK'}  
{'Mansi', 'Yana', 'RK'}  
{'Neev', 'Riva', 'RK'}
```

```
In [9]: ► a.intersection_update(b, c)  
      print(a)
```

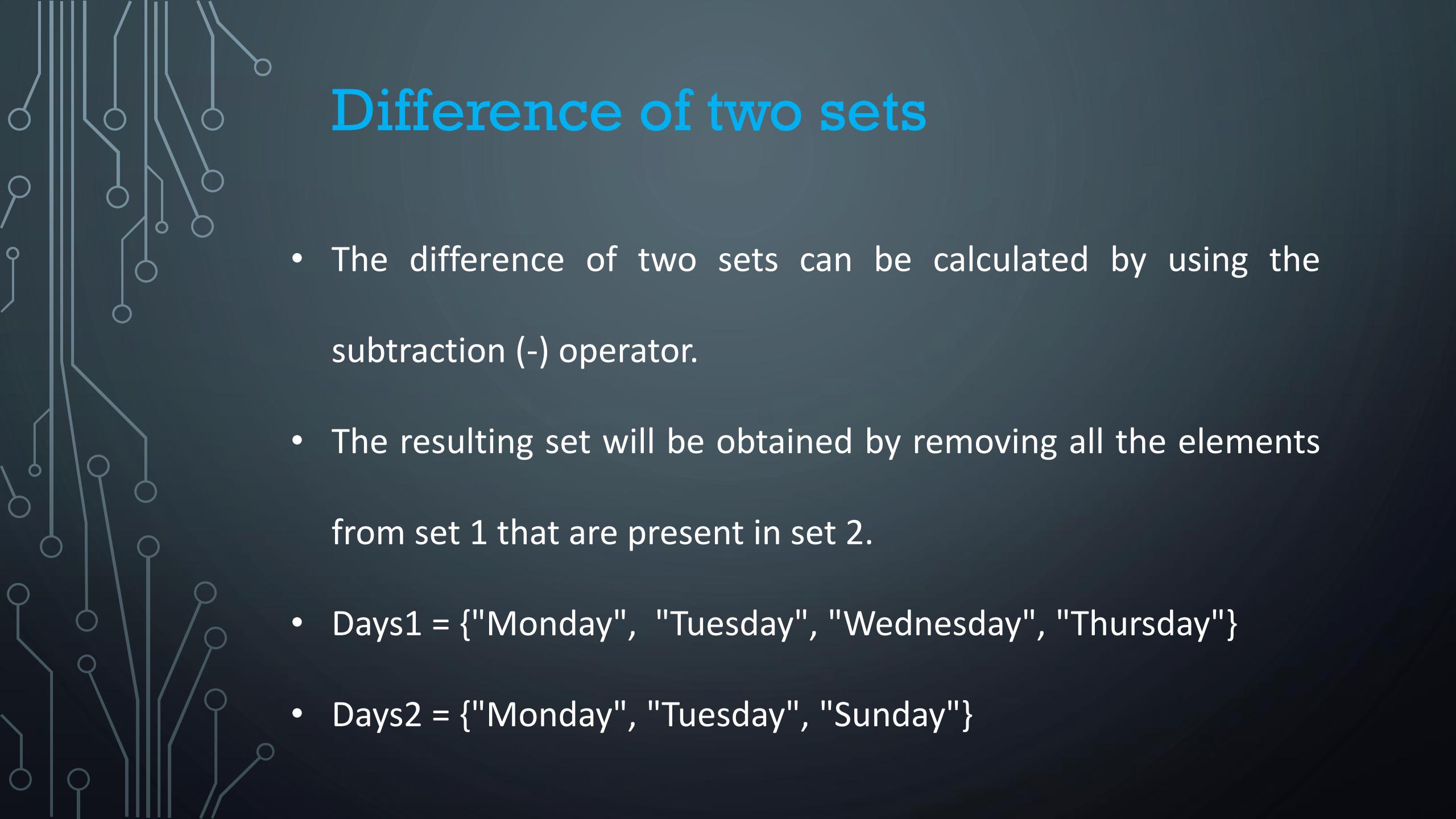
```
{'RK'}
```

```
In [11]: ► b.intersection_update(a, c)  
      print(b)
```

```
{'RK'}
```

```
In [12]: ► c.intersection_update(b, c)  
      print(c)
```

```
{'RK'}
```



# Difference of two sets

- The difference of two sets can be calculated by using the subtraction (-) operator.
- The resulting set will be obtained by removing all the elements from set 1 that are present in set 2.
- Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
- Days2 = {"Monday", "Tuesday", "Sunday"}

```
In [13]: ► Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}  
          print(Days1)  
          Days2 = {"Friday", "Saturday", "Sunday"}  
          print(Days2)
```

```
{'Monday', 'Thursday', 'Wednesday', 'Tuesday'}  
{'Friday', 'Saturday', 'Sunday'}
```

## Method:-1 using subtraction ( - ) operator

```
In [14]: ► print(Days1-Days2)
```

```
{'Monday', 'Thursday', 'Wednesday', 'Tuesday'}
```

## Method:-2 using difference() method

```
In [15]: ► print(Days1.difference(Days2))
```

```
{'Monday', 'Thursday', 'Wednesday', 'Tuesday'}
```



# Set comparisons

- Python allows us to use the comparison operators.
- i.e., `<`, `>`, `<=`, `>=`, `==` with the sets by using which we can check whether a set is subset, superset, or equivalent to other set.
- The Boolean true or false is returned depending upon the items present inside the sets.

In [16]: ► Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}  
print(Days1)  
Days2 = {"Monday", "Tuesday"}  
print(Days2)  
Days3 = {"Monday", "Tuesday", "Friday"}  
print(Days3)

```
{'Monday', 'Thursday', 'Wednesday', 'Tuesday'}  
{'Monday', 'Tuesday'}  
{'Friday', 'Monday', 'Tuesday'}
```

In [17]: ► #Days1 is the superset of Days2 hence it will print true.  
print (Days1>Days2)

True

In [18]: ► #prints false since Days1 is not the subset of Days2  
print (Days1<Days2)

False

In [19]: ► #prints false since Days2 and Days3 are not equivalent  
print (Days2 == Days3)

False



# FrozenSets

- The frozen sets are the immutable form of the normal sets.
- i.e., the items of the frozen set can not be changed and therefore it can be used as a key in dictionary.
- The elements of the frozen set can not be changed after the creation.
- We cannot change or append the content of the frozen sets by using the methods like add() or remove().
- The frozenset() method is used to create the frozenset object.
- The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.



# FrozenSets

```
In [20]: Set = frozenset([1,2,3,4,5])  
        print(type(Set))
```

```
<class 'frozenset'>
```

```
In [22]: for i in Set:  
        print(i);
```

```
1  
2  
3  
4  
5
```

```
In [24]: Set.add(6) #gives an error since we cannot change the content of Frozenset after creation
```

---

```
AttributeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-24-fb8c200a4f81> in <module>
```

```
----> 1 Set.add(6) #gives an error since we cannot change the content of Frozenset after creation
```

```
AttributeError: 'frozenset' object has no attribute 'add'
```

# Frozenset for the dictionary

- If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

```
In [25]: ➜ Dictionary = {"Name":"John", "Country":"USA", "ID":101}
          print(type(Dictionary))

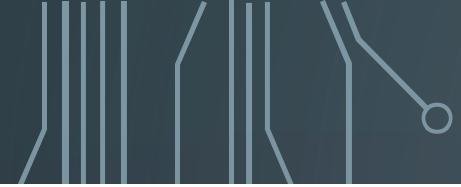
          <class 'dict'>
```

```
In [26]: ➜ Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the dictionary
          print(type(Frozenset))

          <class 'frozenset'>
```

```
In [27]: ➜ for i in Frozenset:
          print(i)
```

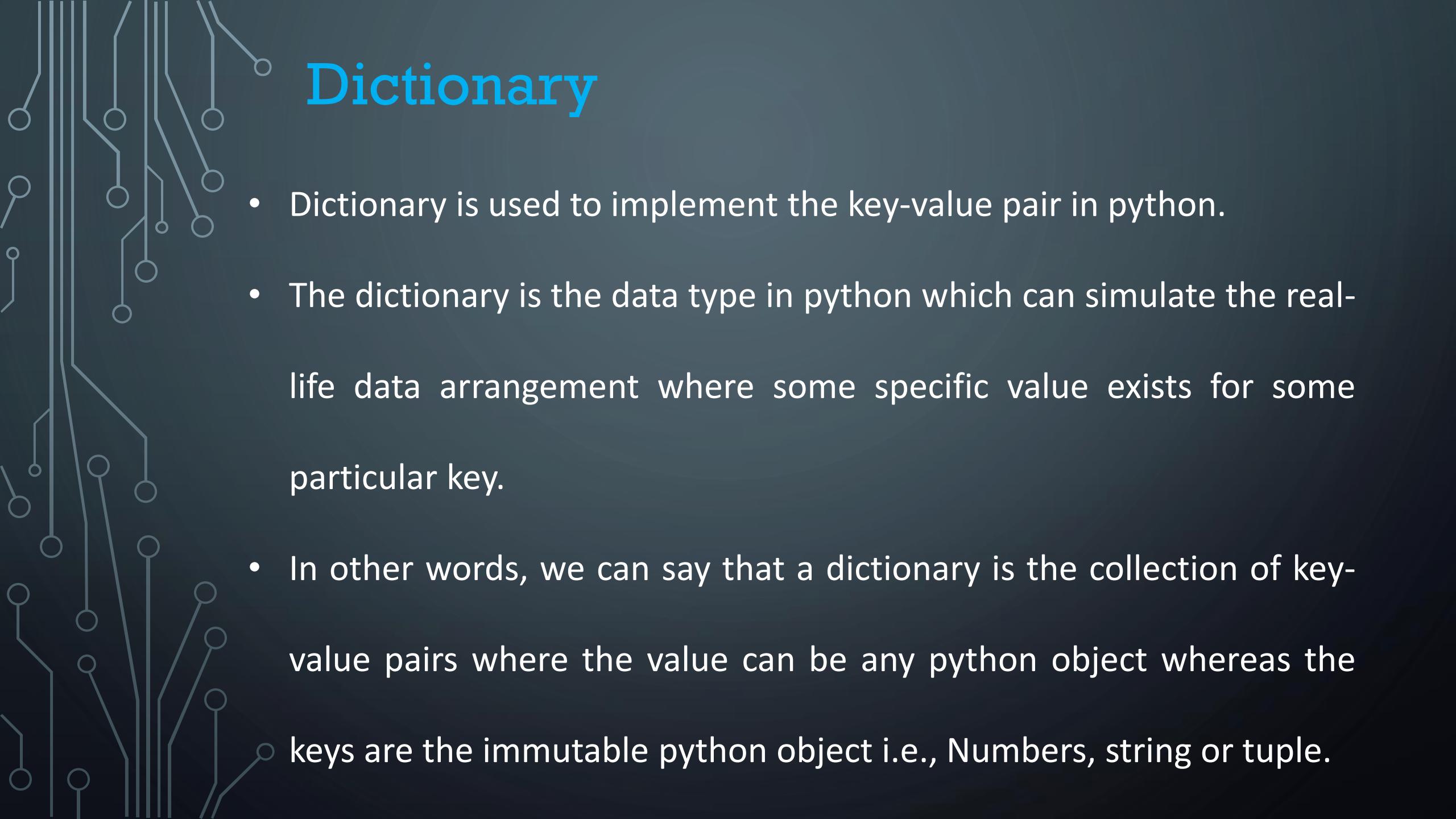
ID  
Name  
Country



# Python in-built Set methods

SN	Method	Description
1	<code>add(item)</code>	It adds an item to the set. It has no effect if the item is already present in the set.
2	<code>clear()</code>	It deletes all the items from the set.
3	<code>copy()</code>	It returns a shallow copy of the set.
4	<code>difference_update(...)</code>	It modifies this set by removing all the items that are also present in the specified sets.
5	<code>discard(item)</code>	It removes the specified item from the set.
6	<code>intersection()</code>	It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified).
7	<code>intersection_update(...)</code>	It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

8	Isdisjoint(...)	Return True if two sets have a null intersection.
9	Issubset(...)	Report whether another set contains this set.
10	Issuperset(...)	Report whether this set contains another set.
11	pop()	Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.
12	remove(item)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
13	symmetric_difference(...)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
14	symmetric_difference_update(...)	Update a set with the symmetric difference of itself and another.
15	union(...)	Return the union of sets as a new set. (i.e. all elements that are in either set.)
16	update()	Update a set with the union of itself and others.



# Dictionary

- Dictionary is used to implement the key-value pair in python.
- The dictionary is the data type in python which can simulate the real-life data arrangement where some specific value exists for some particular key.
- In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any python object whereas the keys are the immutable python object i.e., Numbers, string or tuple.



# Dictionary

- Dictionaries are defined using curly brackets { }:
- Dictionaries are a built-in Python data structure for mapping keys to values.
- Dictionaries are more used like a database because here you can index a particular sequence with your user defined string.

# Create dictionary

- The dictionary can be created by using multiple key-value pairs enclosed with the small brackets () and separated by the colon (:).
- The collections of the key-value pairs are enclosed within the curly braces {}.

```
In [1]: d0 = {}  
d1 = dict()  
print(type(d0))  
print(type(d1))  
  
<class 'dict'>  
<class 'dict'>  
  
In [2]: Dict = {"Name": "RK", "Age": 25}  
print(Dict)  
  
{'Name': 'RK', 'Age': 25}
```



# Create dictionary

- Dictionary works somewhat like a list but with an added capability of assigning it's own index style.

In [6]: ►

```
d = {}  
print(d)
```

```
{}
```

In [7]: ►

```
d['One'] = 1  
d['OneTwo'] = 12  
print (d)
```

```
{'One': 1, 'OneTwo': 12}
```



# Zip Function to Create Dictionary

- `zip( )` function is used to combine two lists.
- The two lists are combined to form a single list and each elements are clubbed with their respective elements from the other list inside a tuple.
- Tuples because that is what is assigned and the value should not change.
- Further, To convert the above into a dictionary. `dict( )` function is used.

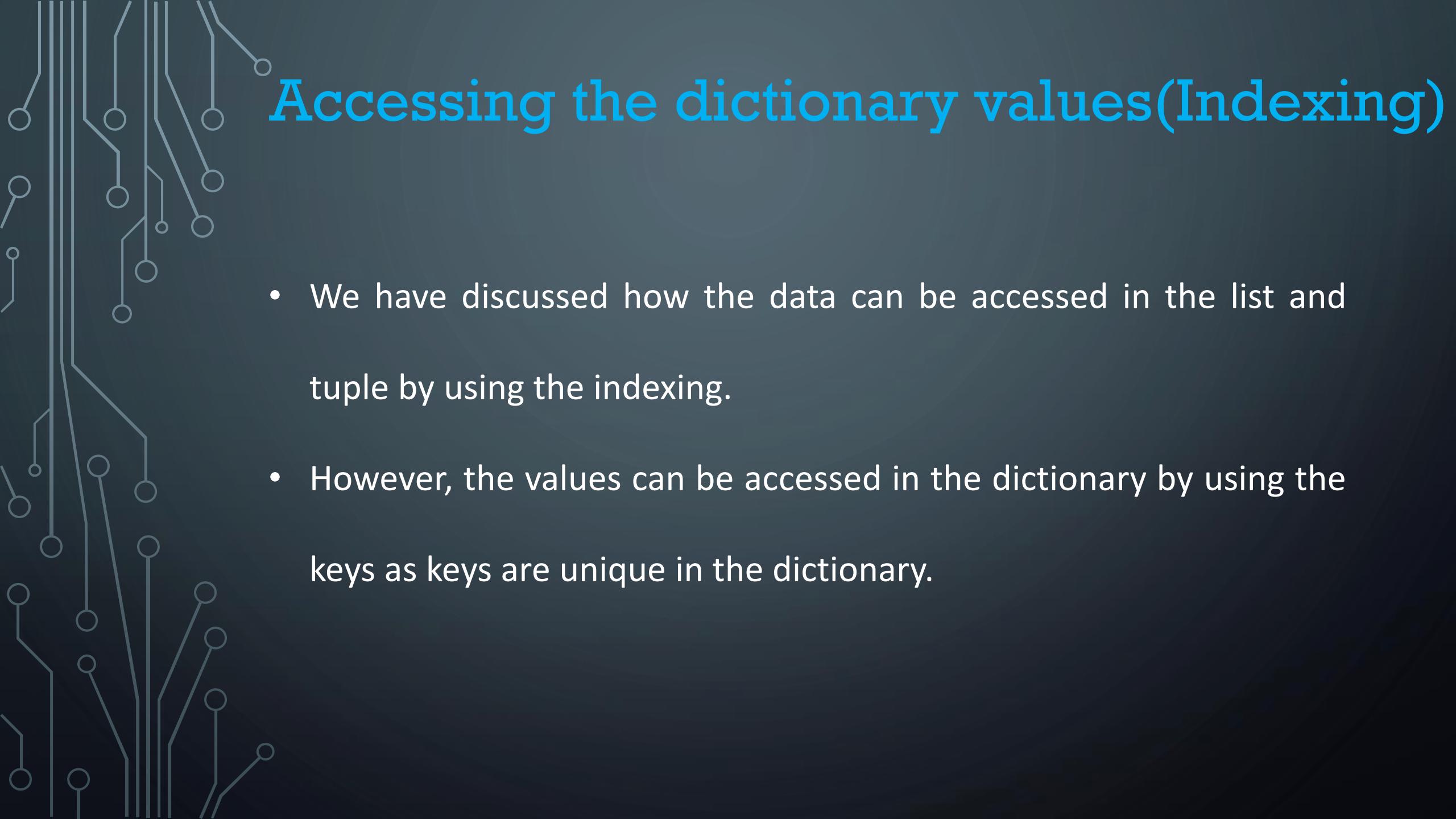
```
In [48]: ► names = ['One', 'Two', 'Three', 'Four', 'Five']
      print(names)
      numbers = [1, 2, 3, 4, 5]
      print(numbers)
```

```
['One', 'Two', 'Three', 'Four', 'Five']
[1, 2, 3, 4, 5]
```

```
In [49]: ► d = zip(names, numbers)
```

```
In [50]: ► new_d = dict(d)
      print(new_d)
```

```
{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
```



# Accessing the dictionary values(Indexing)

- We have discussed how the data can be accessed in the list and tuple by using the indexing.
- However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

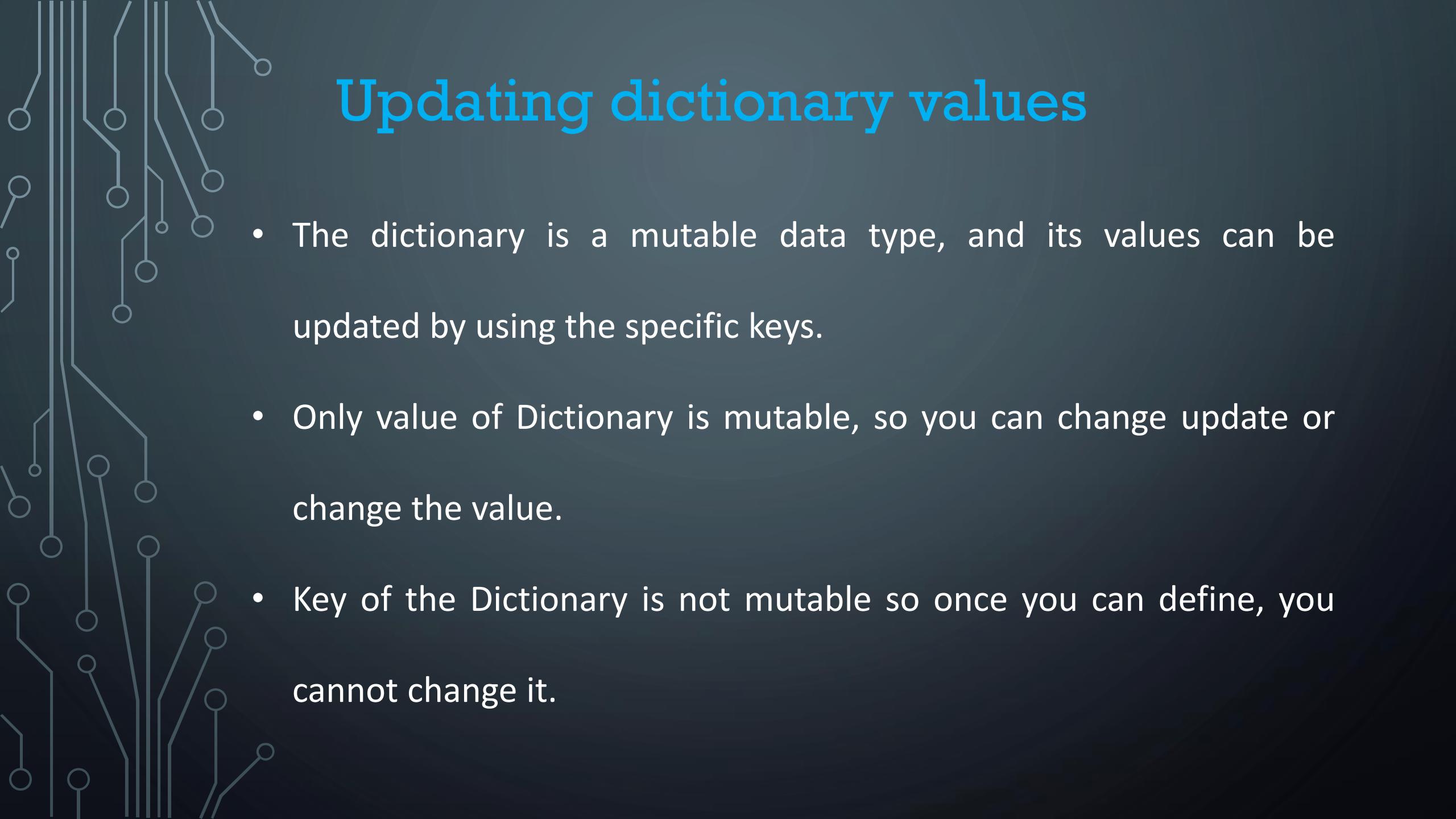
```
In [9]: ► d = {'a':1, 'b':2, 'c':3}
      print(d)
      print(type(d))
      Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
      print(Employee)
      print(type(Employee))
```

```
{'a': 1, 'b': 2, 'c': 3}
<class 'dict'>
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
<class 'dict'>
```

```
In [10]: ► print("Name : %s" %Employee["Name"])
      print("Age : %d" %Employee["Age"])
      print("Salary : %d" %Employee["salary"])
      print("Company : %s" %Employee["Company"])
```

```
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

```
In [11]: ► print(d['a'])
      print(d['c'])
```



# Updating dictionary values

- The dictionary is a mutable data type, and its values can be updated by using the specific keys.
- Only value of Dictionary is mutable, so you can change update or change the value.
- Key of the Dictionary is not mutable so once you can define, you cannot change it.

```
In [14]: ► Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}  
print(Employee)
```

```
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

```
In [16]: ► Employee["Name"] = "Abhay"  
Employee["Age"] = 22  
Employee["salary"] = 30000  
Employee["Company"] = "Greycell"  
print(Employee)
```

```
{'Name': 'Abhay', 'Age': 22, 'salary': 30000, 'Company': 'Greycell'}
```

```
In [17]: ► Employee["Name"] = input("Name: ");  
Employee["Age"] = int(input("Age: "));  
Employee["salary"] = int(input("Salary: "));  
Employee["Company"] = input("Company:");  
print(Employee)
```

Name: RK

Age: 25

Salary: 40000

Company:Social Orbit

```
{'Name': 'RK', 'Age': 25, 'salary': 40000, 'Company': 'Social Orbit'}
```

# Delete Elements of Dictionary

- The items of the dictionary can be deleted by using the `del` keyword

```
In [18]: ► Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}  
print(Employee)
```

```
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

```
In [19]: ► del Employee["Name"]  
del Employee["Company"]  
print(Employee)
```

```
{'Age': 29, 'salary': 25000}
```

```
In [20]: ► del Employee  
print(Employee)
```

```
NameError Traceback (most recent call last)  
<ipython-input-20-13fd2b93d2ac> in <module>  
      1 del Employee  
----> 2 print(Employee)
```

```
NameError: name 'Employee' is not defined
```

# Remove & Clear

- **pop( )** function is used to get the remove that particular element and this removed element can be assigned to a new variable.
- But remember only the value is stored and not the key.
- Because the is just a index value.
- **clear( )** function is used to erase the entire database that was created.

In [25]: ► d = {'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}  
print(d)

```
{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
```

In [26]: ► a2 = d.pop('Four')  
print (d)  
print (a2)

```
{'One': 1, 'Two': 2, 'Three': 3, 'Five': 5}  
4
```

In [27]: ► d.clear()  
print(d)

```
{}  
()
```



# Iterating Dictionary

- A dictionary can be iterated using the for loop.

**for loop to print the items of the dictionary by using items() method.**

In [39]: ➤ `for x in d.items():  
 print(x)`

```
('One', 1)  
('Two', 2)  
('Three', 3)  
('Four', 4)  
('Five', 5)
```



## for loop to print all the keys of a dictionary

In [30]: ➜ `for x in d:  
 print(x)`

One  
Two  
Three  
Four  
Five

## for loop to print all the values of the dictionary

In [31]: ➜ `for x in d:  
 print(d[x])`

1  
2  
3  
4  
5

**for loop to print the values of the dictionary by using keys() method.**

In [36]: ► `for x in d.keys():`

```
    print(x)
```

One  
Two  
Three  
Four  
Five

**for loop to print the values of the dictionary by using values() method.**

In [37]: ► `for x in d.values():`

```
    print(x)
```

1  
2  
3  
4  
5



# Properties of Dictionary keys

A. In the dictionary, we can not store multiple values for the same keys. If we pass more than one values for a single key, then the value which is last assigned is considered as the value of the key.

```
In [40]: Employee = {"Name": "John", "Age": 29, "Salary":25000,"Company":"GOOGLE","Name":"Johnn"}  
for x,y in Employee.items():  
    print(x,y)
```

Name Johnn  
Age 29  
Salary 25000  
Company GOOGLE





# Properties of Dictionary keys

**B.** In python, the key cannot be any mutable object. We can use numbers, strings, or tuple as the key but we can not use any mutable object like the list as the key in the dictionary.

```
In [42]: Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE", [100,201,301]:"Department ID"}  
for x,y in Employee.items():  
    print(x,y)
```

---

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-42-8bf9640c55a5> in <module>  
----> 1 Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE", [100,201,301]:"Department ID"}  
      2 for x,y in Employee.items():  
      3     print(x,y)  
  
TypeError: unhashable type: 'list'
```



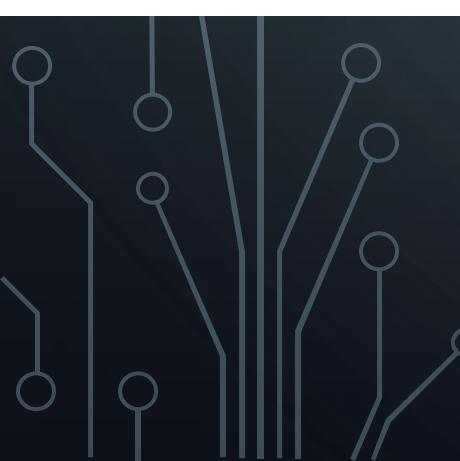
# More Example of Dictionary

```
▶ planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

```
▶ planet_to_initial = {planet: planet[0] for planet in planets}
print(planet_to_initial)
```

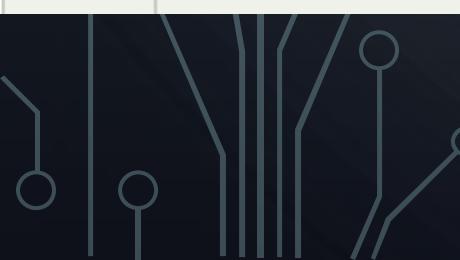
```
{'Mercury': 'M', 'Venus': 'V', 'Earth': 'E', 'Mars': 'M', 'Jupiter': 'J', 'Saturn': 'S', 'Uranus': 'U', 'Neptune': 'N'}
```





# Built-in Dictionary Functions

SN	Function	Description
1	cmp(dict1, dict2)	It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false.
2	len(dict)	It is used to calculate the length of the dictionary.
3	str(dict)	It converts the dictionary into the printable string representation.
4	type(variable)	It is used to print the type of the passed variable.



# Built-in Dictionary methods

<code>dic.clear()</code>	It is used to delete all the items of the dictionary.
<code>dict.copy()</code>	It returns a shallow copy of the dictionary.
<code>dict.fromkeys(iterable, value = None, /)</code>	Create a new dictionary from the iterable with the values equal to value.
<code>dict.get(key, default = "None")</code>	It is used to get the value specified for the passed key.
<code>dict.has_key(key)</code>	It returns true if the dictionary contains the specified key.

# Built-in Dictionary methods

dict.items()	It returns all the key-value pairs as a tuple.
dict.keys()	It returns all the keys of the dictionary.
dict.setdefault(key , default= "None")	It is used to set the key to the default value if the key is not specified in the dictionary
dict.update(dict2)	It updates the dictionary by adding the key-value pair of dict2 to this dictionary.
dict.values()	It returns all the values of the dictionary.

# Built-in Dictionary methods

<code>len()</code>	<p><code>Len()</code> method find the length of the dictionary.</p>
<code>popitem()</code>	<p><code>popitem()</code> method removes an element from the dictionary. It removes arbitrary element and return its value. If the dictionary is empty, it returns an error Key Error.</p>
<code>pop()</code>	<p><code>pop()</code> method removes an element from the dictionary. It removes the element which is associated to the specified key.</p>

# Built-in Dictionary methods

`count()`

Count() method count the total numbers of elements in dictionary.

`index()`

Index() find the position of the dictionary keys & Values

# List v/s Set v/s Dictionary v/s Tuple

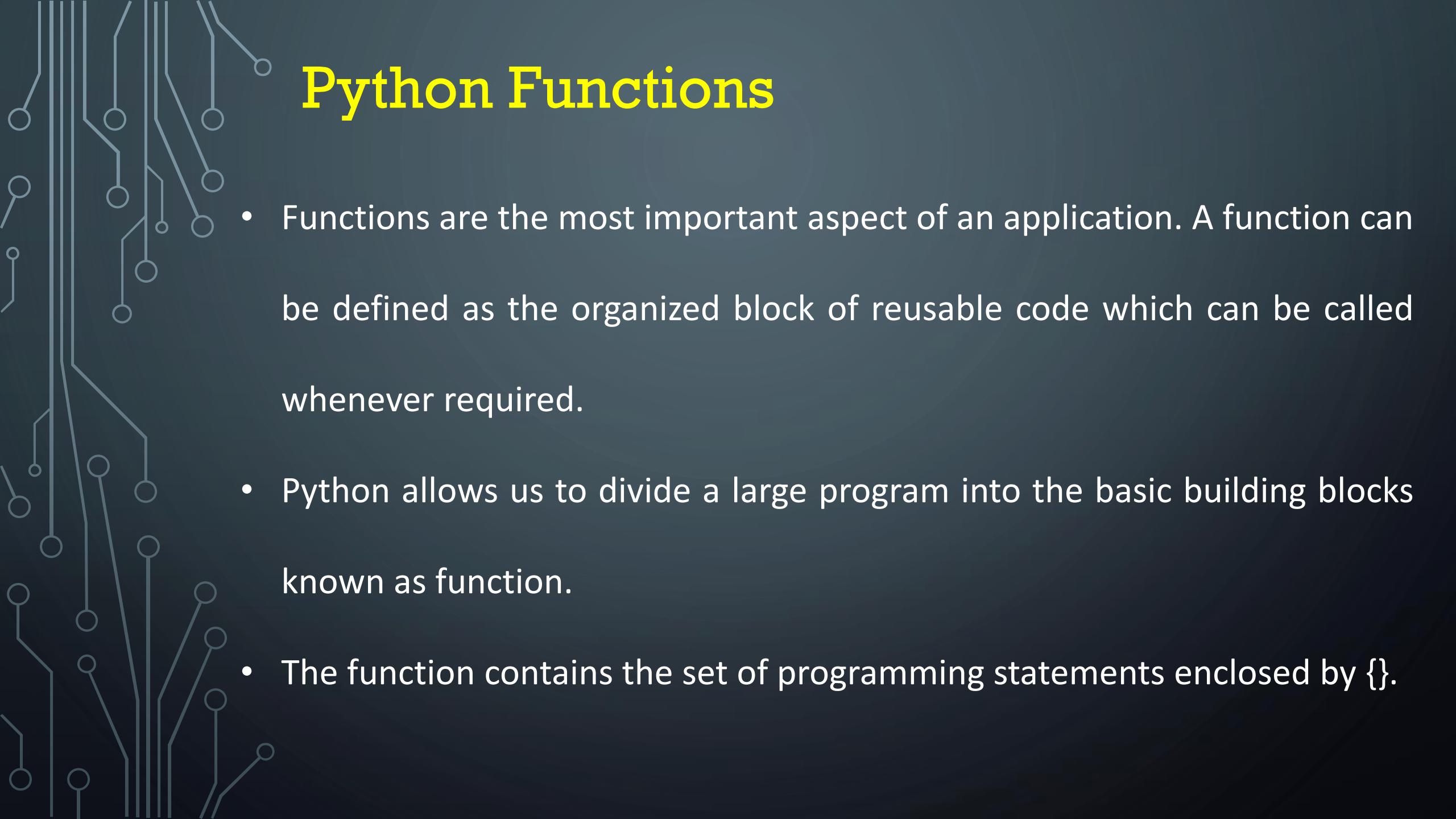
List	Set	Dictionary	Tuple
List = [10,12,15]	Set = {1,23,45,'RK',3.5}	Dict = {1: "RK", 2:"2.36"}	T = ('Spam',3.15,4,'RK')
Access: List[0], List[1]	Set cannot be indexed	Access: print(Dict[1])	Access: print(T[0], T[1])
Can contain duplicate Element.	Cannot contain duplicate Element.	Cannot contain duplicate Key but can contain value.	Can contain duplicate Element.
List[0] = 100	Set.add(100)	Dict[1] = "Yana"	Methods not available, it gives error

# List v/s Set v/s Dictionary v/s Tuple

List	Set	Dictionary	Tuple
Mutable	Mutable	Mutable	Immutable – Value cannot be changed, once assigned
<code>List = []</code> <code>L = list()</code>	<code>Set = {}</code> <code>S = set()</code>	<code>Dict = {}</code> <code>D = dict()</code>	<code>Tuple = ()</code> <code>T = tuple()</code>
Slicing can be possible <code>Print(list[1:2])</code>	Slicing not possible	Slicing not possible	Slicing possible <code>Print(Tuple[2:3])</code>

# List v/s Set v/s Dictionary v/s Tuple

List	Set	Dictionary	Tuple
<ul style="list-style-type: none"><li>• Use list if you have a collection of data that does not need random access.</li><li>• Use lists if you need a simple, iterable collection that modified frequently.</li></ul>	<ul style="list-style-type: none"><li>• Membership testing and the elimination duplicate entries.</li><li>• When you need uniqueness for the entry.</li></ul>	<ul style="list-style-type: none"><li>• When you need logical association between key &amp; values(like JSON data).</li><li>• Dictionary easy to access then JSON.</li></ul>	<ul style="list-style-type: none"><li>• Use tuple when data cannot change.</li><li>• Tuple use with key of dictionary because key of dictionary is immutable.</li></ul>



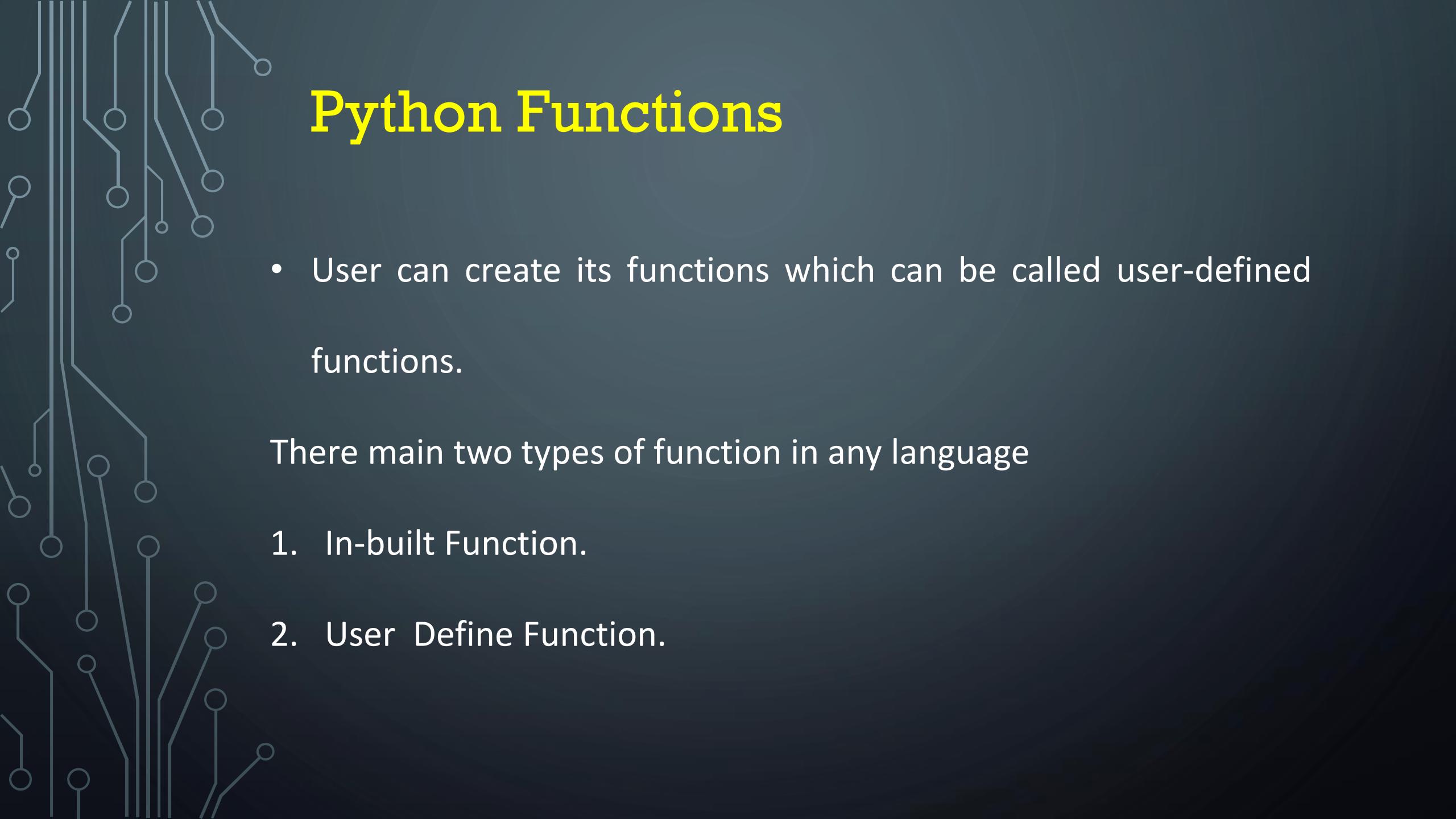
# Python Functions

- Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code which can be called whenever required.
- Python allows us to divide a large program into the basic building blocks known as function.
- The function contains the set of programming statements enclosed by {}.



# Python Functions

- In other words, we can say that the collection of functions creates a program.
- The function is also known as procedure or subroutine in other programming languages.
- A function can be called multiple times to provide reusability and modularity(use with different modules) to the python program.

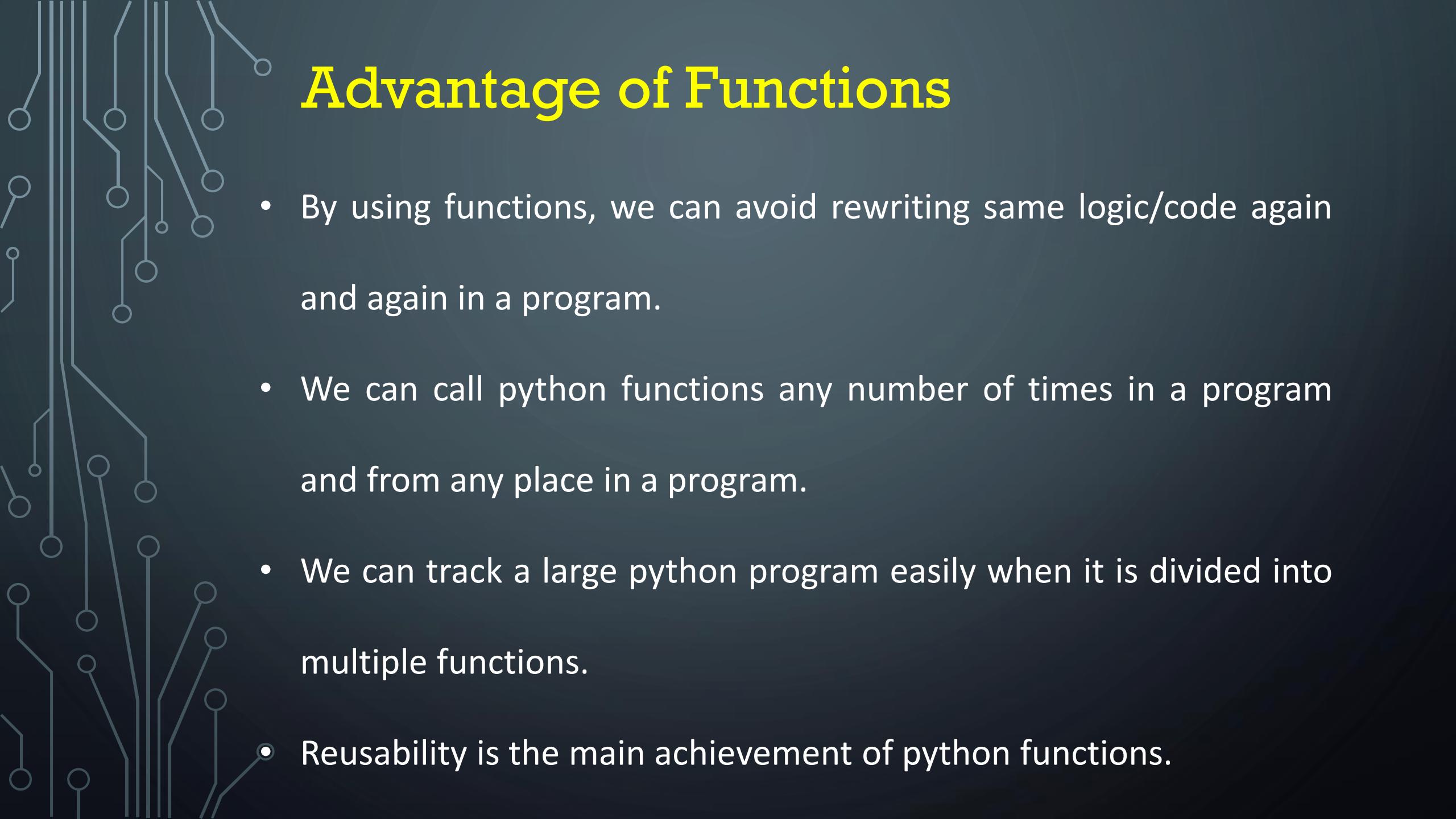


# Python Functions

- User can create its functions which can be called user-defined functions.

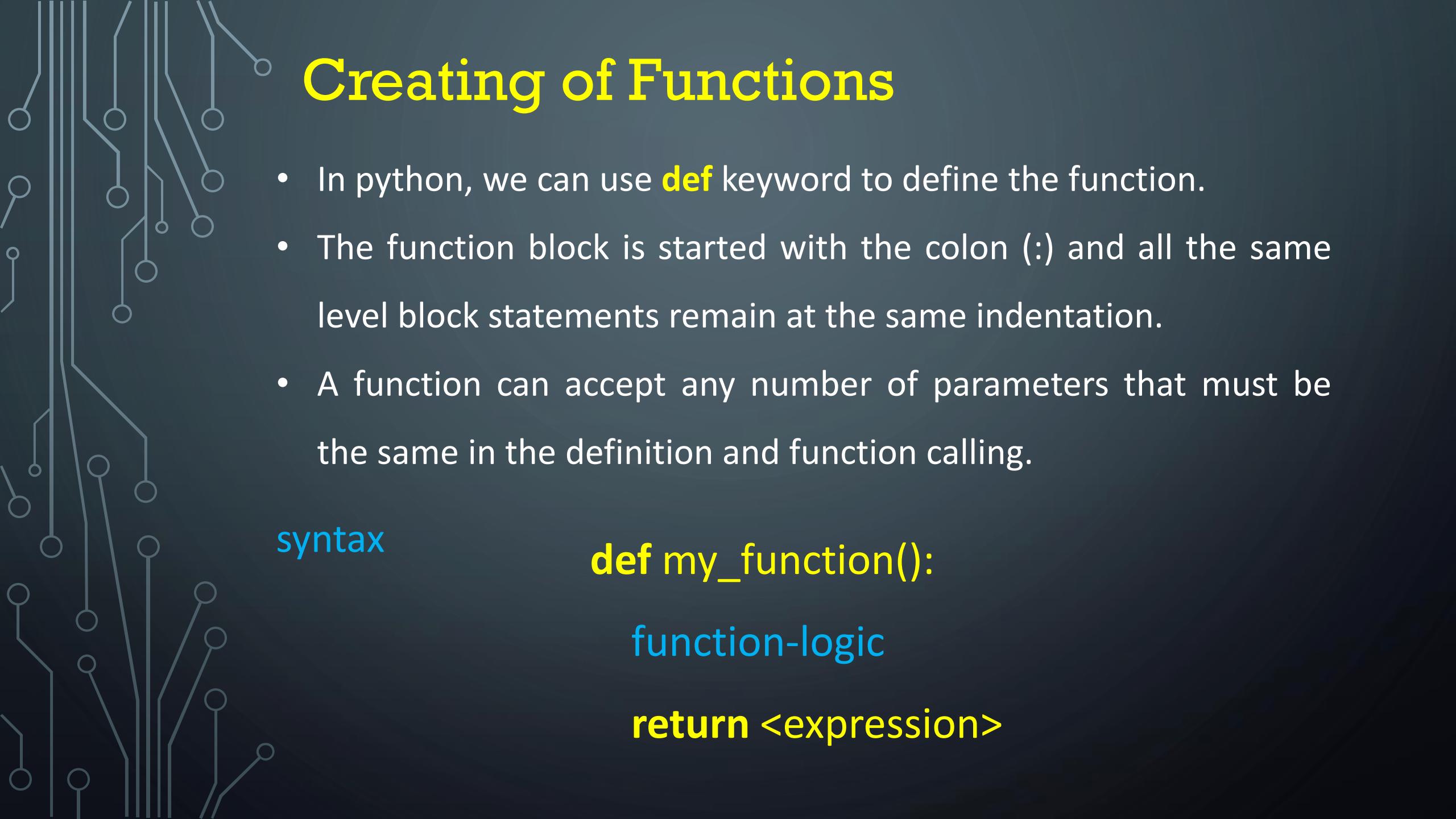
There main two types of function in any language

1. In-built Function.
2. User Define Function.



# Advantage of Functions

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call python functions any number of times in a program and from any place in a program.
- We can track a large python program easily when it is divided into multiple functions.
- Reusability is the main achievement of python functions.



# Creating of Functions

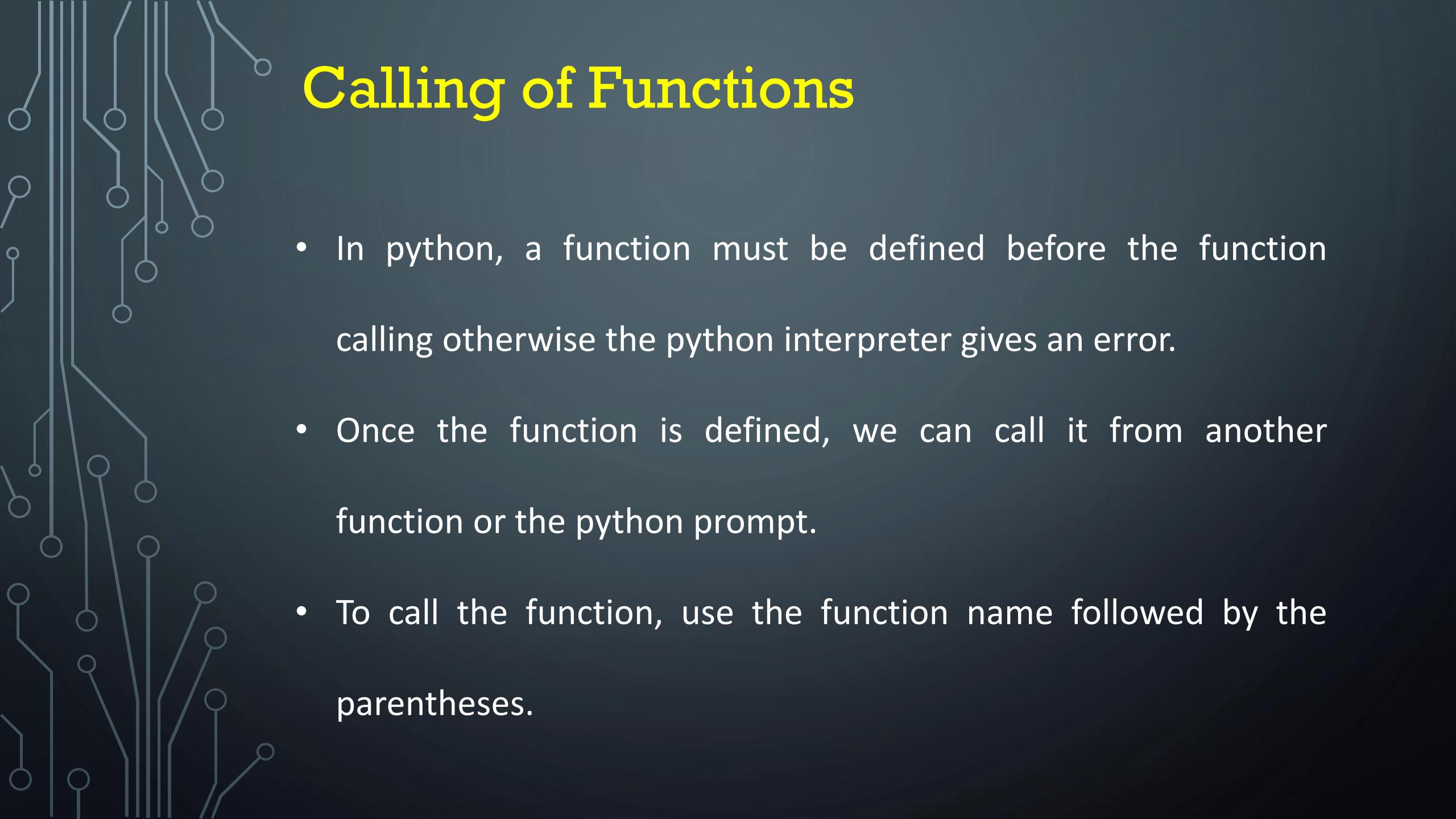
- In python, we can use **def** keyword to define the function.
- The function block is started with the colon (:) and all the same level block statements remain at the same indentation.
- A function can accept any number of parameters that must be the same in the definition and function calling.

**syntax**

```
def my_function():
```

**function-logic**

```
return <expression>
```



# Calling of Functions

- In python, a function must be defined before the function calling otherwise the python interpreter gives an error.
- Once the function is defined, we can call it from another function or the python prompt.
- To call the function, use the function name followed by the parentheses.

# Create function

In [1]: ► `def hello_world():  
 print("hello world")`

# Calling of function

In [2]: ► `hello_world()`

hello world

## Create function

In [4]: ► `def sum (a,b):  
 return a+b;`

## Calling of function

In [5]: ► `sum(2,3)`

Out[5]: 5

In [6]: ► `x = 2  
y = 3`

In [7]: ► `sum(x,y)`

Out[7]: 5



# Parameters in Functions

- The information into the functions can be passed as the parameters.
- The parameters are specified in the parentheses.
- We can give any number of parameters, but we have to separate them with a comma.

## Create function

In [10]: ► *#defining the function*  
`def func (name):  
 print("Hi",name);`

## Calling of function

In [11]: ► *#calling the function*  
`func("RK")`

Hi RK

In [12]: ► `func("Abhay")`

Hi Abhay

In [16]: ► *#python function to calculate the sum of two variables*  
*#defining the function*

```
def sum (a,b):  
    return a+b;
```

In [17]: ► *#taking values from the user*

```
a = int(input("Enter a: "))  
b = int(input("Enter b: "))
```

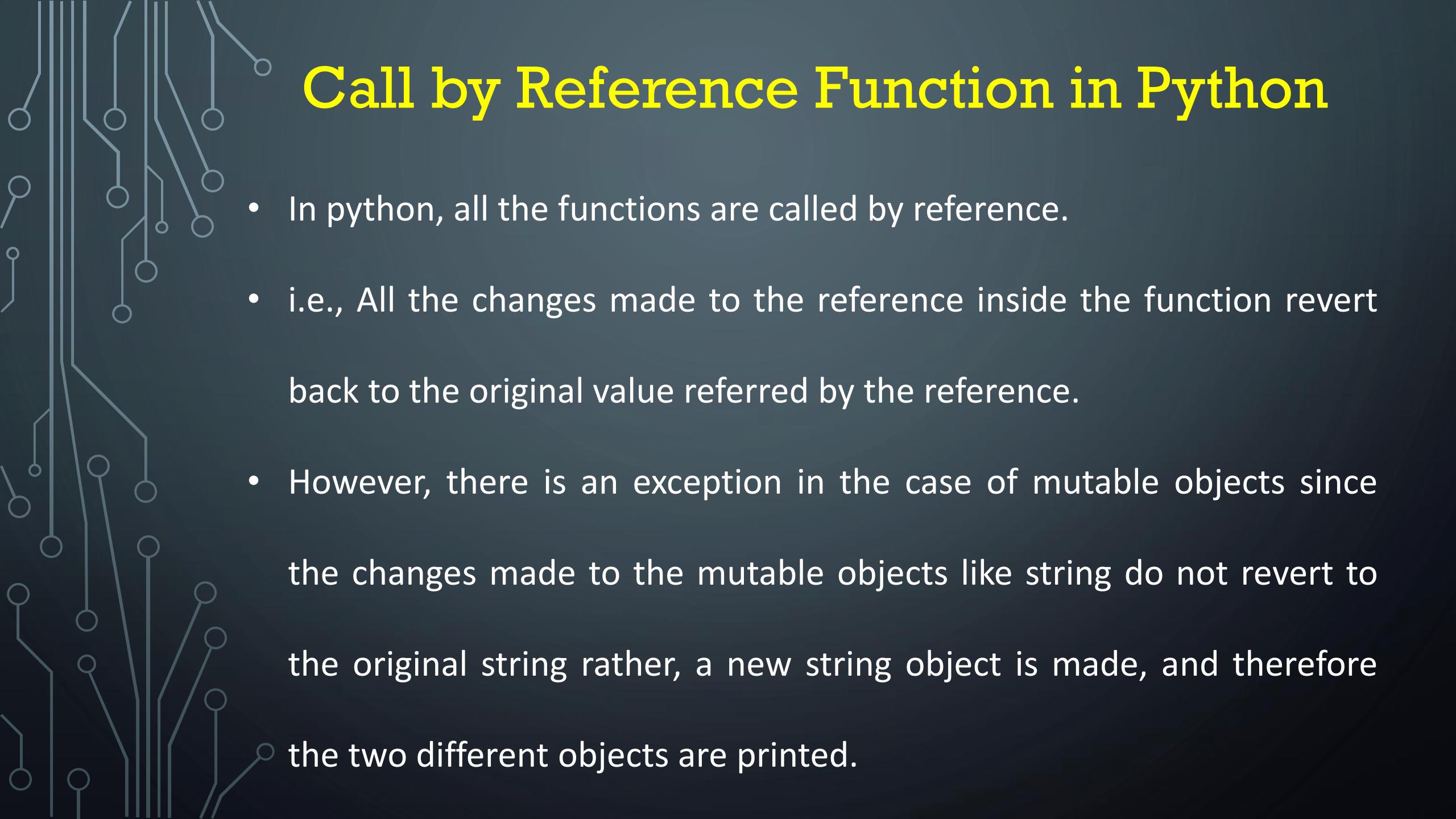
Enter a: 10

Enter b: 20

In [18]: ► *#printing the sum of a and b*  
*# calling of the function*

```
print("Sum = ",sum(a,b))
```

Sum = 30



# Call by Reference Function in Python

- In python, all the functions are called by reference.
- i.e., All the changes made to the reference inside the function revert back to the original value referred by the reference.
- However, there is an exception in the case of mutable objects since the changes made to the mutable objects like string do not revert to the original string rather, a new string object is made, and therefore the two different objects are printed.

## Example 1 Passing mutable Object (List)

In [55]: ► *#defining the function*

```
def change_list(list1):
    list1.append(20);
    list1.append(30);
    print("list inside function = ",list1)
```

In [56]: ► *#defining the List*

```
L1 = [10,30,40,50]
```

In [57]: ► *#calling the function*

```
change_list(L1);

print("list outside function = ",L1);
```

```
list inside function =  [10, 30, 40, 50, 20, 30]
list outside function =  [10, 30, 40, 50, 20, 30]
```

## Example 2 Passing Immutable Object (String)

In [46]: ► *#defining the function*  
`def change_string (str):  
 str = str + " Hows you";  
 print("printing the string inside function :",str);`

In [47]: ► *#defining the String*  
`S1 = "Hi I am there"`

In [50]: ► *#calling the function*  
`change_string(S1)  
print("printing the string outside function :",S1)`

printing the string inside function : Hi I am there Hows you  
printing the string outside function : Hi I am there

# Types of arguments

There may be several types of arguments which can be passed at the time of function calling.

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments



# Required Arguments

- we can provide the arguments at the time of function calling.
- As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition.



# Required Arguments

- If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.
- If number of argument not match it also give the error.

In [1]: ►

```
'''the argument name is the required argument to the function func'''  
def func(name):  
    message = "Hi "+name;  
    return message
```

In [2]: ►

```
name = input("Enter the name?")
```

Enter the name?RK

In [3]: ►

```
print(func(name))
```

Hi RK

```
In [4]: ► def simple_interest(p,r,n):  
         return (p*r*n)/100
```

```
In [5]: ► p = float(input("Enter the principle amount? "))  
        r = float(input("Enter the rate of interest? "))  
        n = float(input("Enter the time in years? "))
```

Enter the principle amount? 10000

Enter the rate of interest? 10

Enter the time in years? 2

```
In [6]: ► print("Simple Interest: ",simple_interest(p,r,n))
```

Simple Interest: 2000.0

```
In [9]: ► def simple_interest(p,n,r):  
         return (p*n*r)/100
```

```
In [10]: ► p = float(input("Enter the principle amount? "))  
         r = float(input("Enter the rate of interest? "))  
         n = float(input("Enter the time in years? "))
```

Enter the principle amount? 10000

Enter the rate of interest? 10

Enter the time in years? 2

```
In [11]: ► print("Simple Interest: ",simple_interest(p,r,n))
```

Simple Interest: 2000.0

In [13]: ►

```
'''the function calculate  
returns the sum of  
two arguments a and b '''  
def calculate(a,b):  
    return a+b
```

In [14]: ►

```
'''this causes an error as  
we are missing a  
required arguments b.'''  
calculate(10)
```

---

**TypeError**

Traceback (most recent call last)

<ipython-input-14-a0fff587c214> in <module>

2 we are missing a

3 required arguments b.'''

----> 4 calculate(10)

**TypeError**: calculate() missing 1 required positional argument: 'b'



# Keyword Arguments

- Python allows us to call the function with the keyword arguments.

This kind of function call will enable us to pass the arguments in the random order.

- The name of the arguments is treated as the keywords and matched in the function calling and definition.
- If the same match is found, the values of the arguments are copied in the function definition.

```
In [15]: ► '''function func is called  
with the name and message  
as the keyword arguments '''  
  
def func(name,message):  
    print("printing the message with",name,"and ",message)
```

```
In [16]: ► '''name and message is  
copied with the  
values John and  
hello respectively '''  
  
func(name = "John",message="hello")
```

printing the message with John and hello

## Example 2 providing the values in different order at the calling

```
In [17]: ► '''The function simple_interest(p, t, r)  
is called with the keyword arguments  
the order of arguments doesn't matter in this case'''  
def simple_interest(p,t,r):  
    return (p*t*r)/100
```

```
In [18]: ► print("Simple Interest:",simple_interest(t=10,r=10,p=1900))
```

Simple Interest: 1900.0

In [23]: ►

```
'''The function simple_interest(p, t, r)
is called with the keyword arguments.'''
def simple_interest(p,t,r):
    return (p*t*r)/100
```

In [24]: ►

```
print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900))
```

```
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-24-14a6568f3598> in <module>
----> 1 print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900))
```

**TypeError**: simple\_interest() got an unexpected keyword argument 'time'

In [25]: ►

```
def func(name1,message,name2):
    print("printing the message with",name1,",",message,",and",name2)
```

In [26]: ►

```
#the first argument is not the keyword argument
func("John",message="hello",name2="David")
```

printing the message with John , hello , and David



# Default Arguments

- Python allows us to initialize the arguments at the function definition.
- If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

```
In [2]: ► def printme(name,age=22):  
        print("My name is",name,"and age is",age)
```

```
In [3]: ► '''the variable age is not passed into  
        the function however the default value of  
        age is consider'''  
  
        printme(name = "john")
```

My name is john and age is 22

In [4]: ► `def printme(name,age=22):  
 print("My name is",name,"and age is",age)`

In [5]: ► `'''the variable age is not passed into  
the function however the default value  
of age is considered in the function'''  
printme(name = "john")`

My name is john and age is 22

In [6]: ► `'''the value of age is overwritten here,  
10 will be printed as age '''  
printme(age = 10,name="David")`

My name is David and age is 10



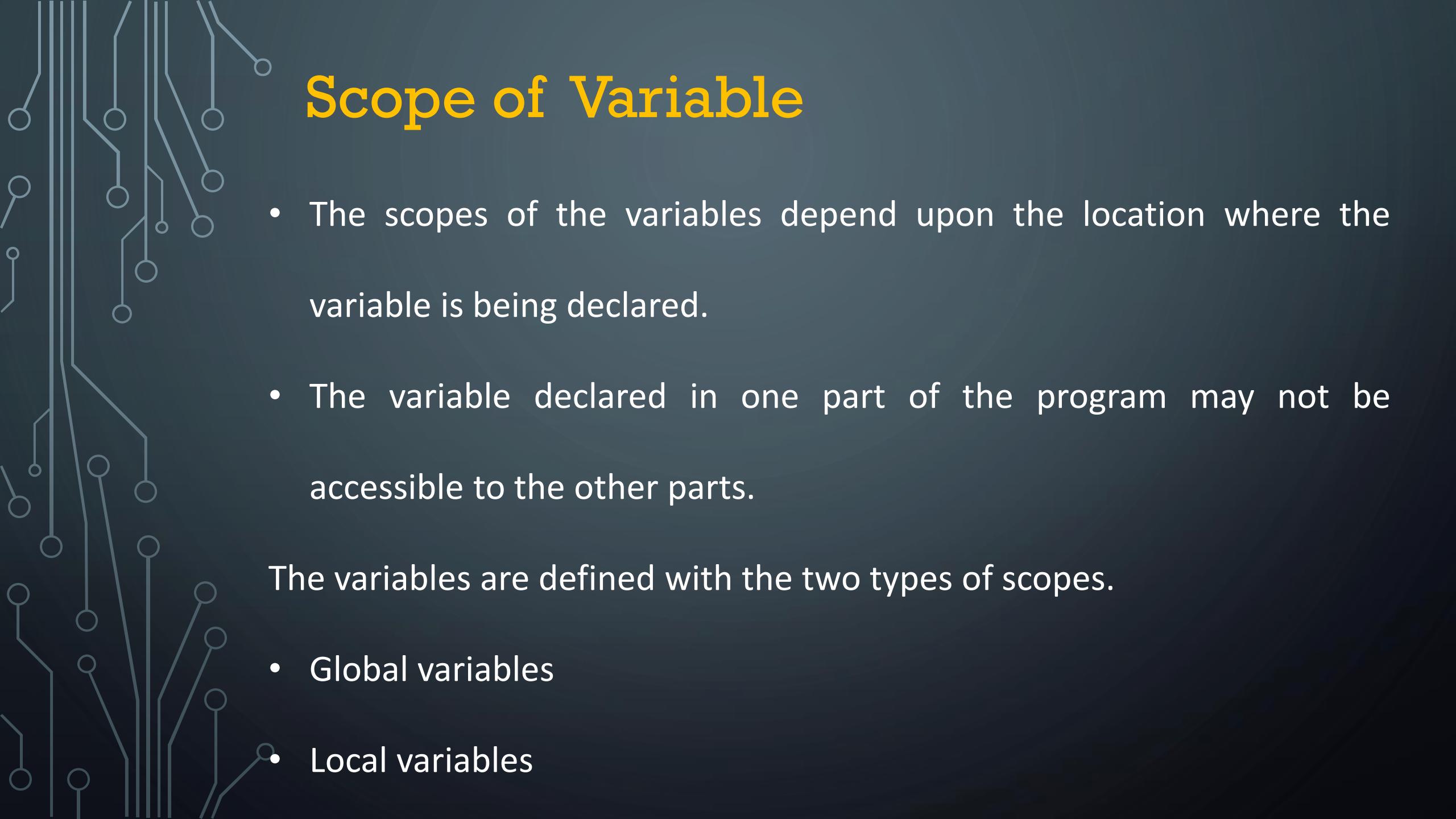
# Variable Length Arguments

- In the large projects, sometimes we may not know the number of arguments to be passed in advance.
- In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.
- We have to define the variable with \* (star) as \*<variable - name >.

```
In [7]: ► def printme(*names):
          print("type of passed argument is ",type(names))
          print("printing the passed arguments...")
          for name in names:
              print(name)
```

```
In [8]: ► printme("john","David","smith","nick")
```

```
type of passed argument is <class 'tuple'>
printing the passed arguments...
john
David
smith
nick
```

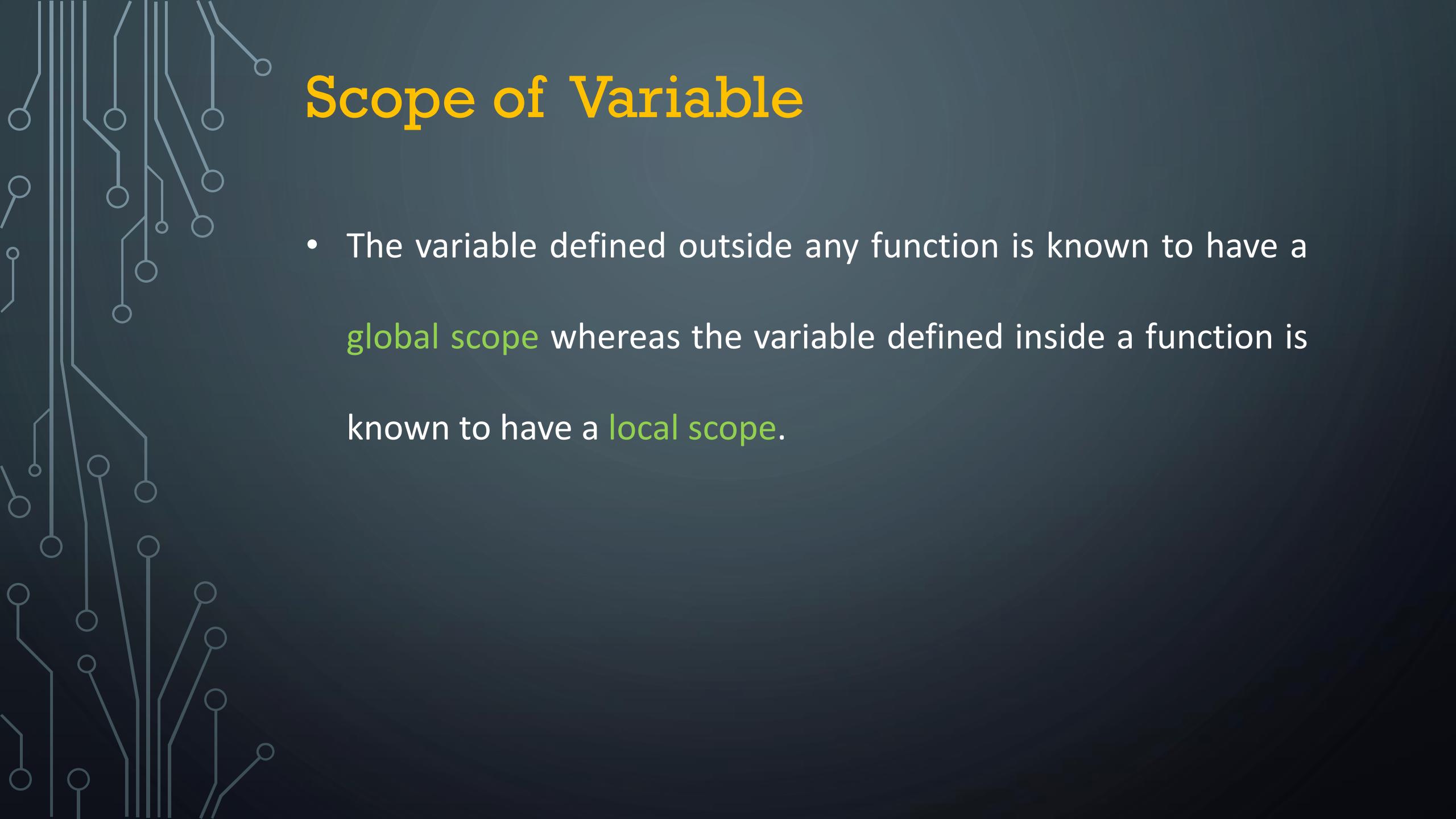


# Scope of Variable

- The scopes of the variables depend upon the location where the variable is being declared.
- The variable declared in one part of the program may not be accessible to the other parts.

The variables are defined with the two types of scopes.

- Global variables
- Local variables



# Scope of Variable

- The variable defined outside any function is known to have a **global scope** whereas the variable defined inside a function is known to have a **local scope**.

In [26]: ►

```
def sum():
    a = 10
    b = 20
    c = a+b
    print("Summation of a & b : ",c)
```

In [27]: ►

```
sum()
```

Summation of a & b : 30

In [28]: ►

```
# function defined variable not access outside
print(c)
```

---

NameError

Traceback (most recent call last)

```
<ipython-input-28-1dd5973cae19> in <module>
```

```
----> 1 print(c)
```

NameError: name 'c' is not defined

In [29]: ►

```
a = 10  
b = 20  
def sum():  
    return a+b
```

In [30]: ►

```
sum()
```

Out[30]: 30

In [31]: ►

```
# function defined variable not access outside  
print(a+b)
```

30

In [ ]:



# Lifetime of Variable

- A variable's lifetime is the period of time for which it resides in the memory.
- A variable that's declared inside python function is destroyed after the function stops executing.
- So the next time the function is called, it does not remember the previous value of that variable.

In [33]: ►

```
def func1():
    counter=0
    counter+=1
    print(counter)
```

In [34]: ►

```
func1()
```

1



# Python Lambda Functions

- Python allows us to not declare the function in the standard manner.
- i.e., by using the def keyword. Rather, the anonymous functions are declared by using lambda keyword.
- Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

# Python Lambda Functions

- The anonymous function contains a small piece of code.
- It simulates inline functions of C and C++, but it is not exactly an inline function.

## Syntax

**lambda arguments : expression**

In [4]: ► `'''a is an argument and a+10  
is an expression which got  
evaluated and returned.'''`

`x = lambda a:a+10`

In [5]: ► `print("sum = ",x(20))`

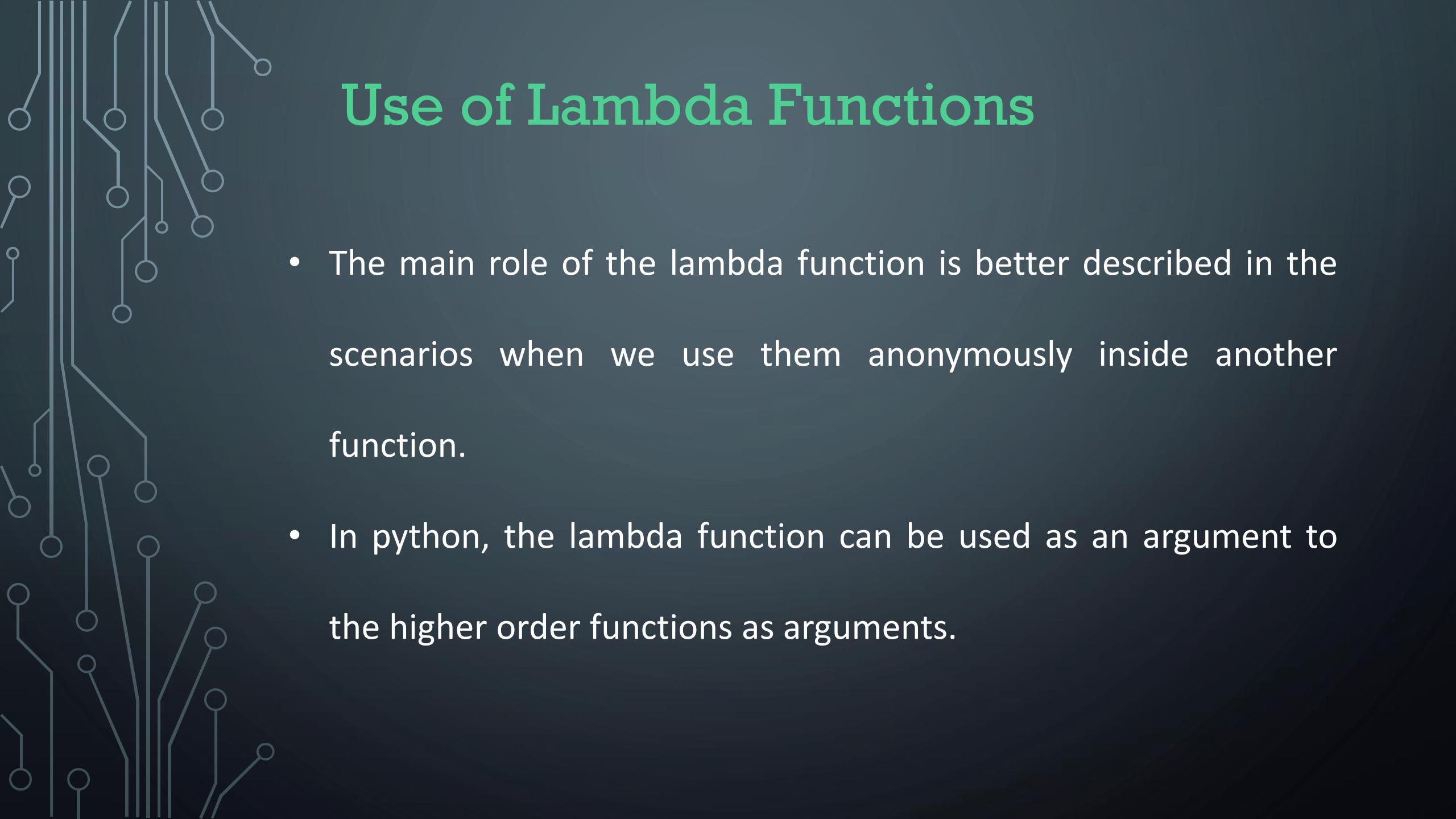
sum = 30

In [6]: ► `'''a and b are the arguments  
and a+b is the expression  
which gets evaluated and returned.'''`

`x = lambda a,b:a+b`

`print("sum = ",x(20,10))`

sum = 30



# Use of Lambda Functions

- The main role of the lambda function is better described in the scenarios when we use them anonymously inside another function.
- In python, the lambda function can be used as an argument to the higher order functions as arguments.

```
In [10]: ► '''the function table(n) prints the table of n  
a will contain the iteration variable i and a  
multiple of n is returned at each function call'''  
  
def table(n):  
    return lambda a:a**n;
```

```
In [11]: ► n = int(input("Enter the number?"))  
  
'''the entered number is passed into the function table.  
b will contain a lambda function which is called again  
and again with the iteration variable i '''  
b = table(n)  
#the Lambda function b is called with the iteration variable i,  
  
for i in range(1,11):  
    print(n, "X", i, "=", b(i));
```

Enter the number?5

5 X 1 = 5  
5 X 2 = 10  
5 X 3 = 15  
5 X 4 = 20  
5 X 5 = 25  
5 X 6 = 30  
5 X 7 = 35  
5 X 8 = 40  
5 X 9 = 45  
5 X 10 = 50

## Use of lambda function with filter

In [12]: ► `'''program to filter out the list which contains  
odd numbers '''  
  
List = {1,2,3,4,10,123,22}`

In [13]: ► `'''the list contains all the items of the list for  
which the lambda function evaluates to true '''  
  
Oddlist = list(filter(lambda x:(x%3 == 0),List))  
  
print(Oddlist)`

[3, 123]

## Use of lambda function with map

```
In [14]: '''program to triple each number  
of the list using map'''  
  
List = {1,2,3,4,10,123,22}
```

```
In [15]: '''this will return the triple of each  
item of the list and add it to new_list'''  
  
new_list = list(map(lambda x:x*3,List))  
  
print(new_list)
```

```
[3, 6, 9, 12, 30, 66, 369]
```

# Python Built-in Function

- The Python built-in functions are defined as the functions whose functionality is pre-defined in Python.



# abs()

- The python **abs()** function is used to return the absolute value of a number.
- It takes only one argument, a number whose absolute value is to be returned.
- The argument can be an integer and floating-point number.
- If the argument is a complex number, then, **abs()** returns its magnitude.

In [19]: ►

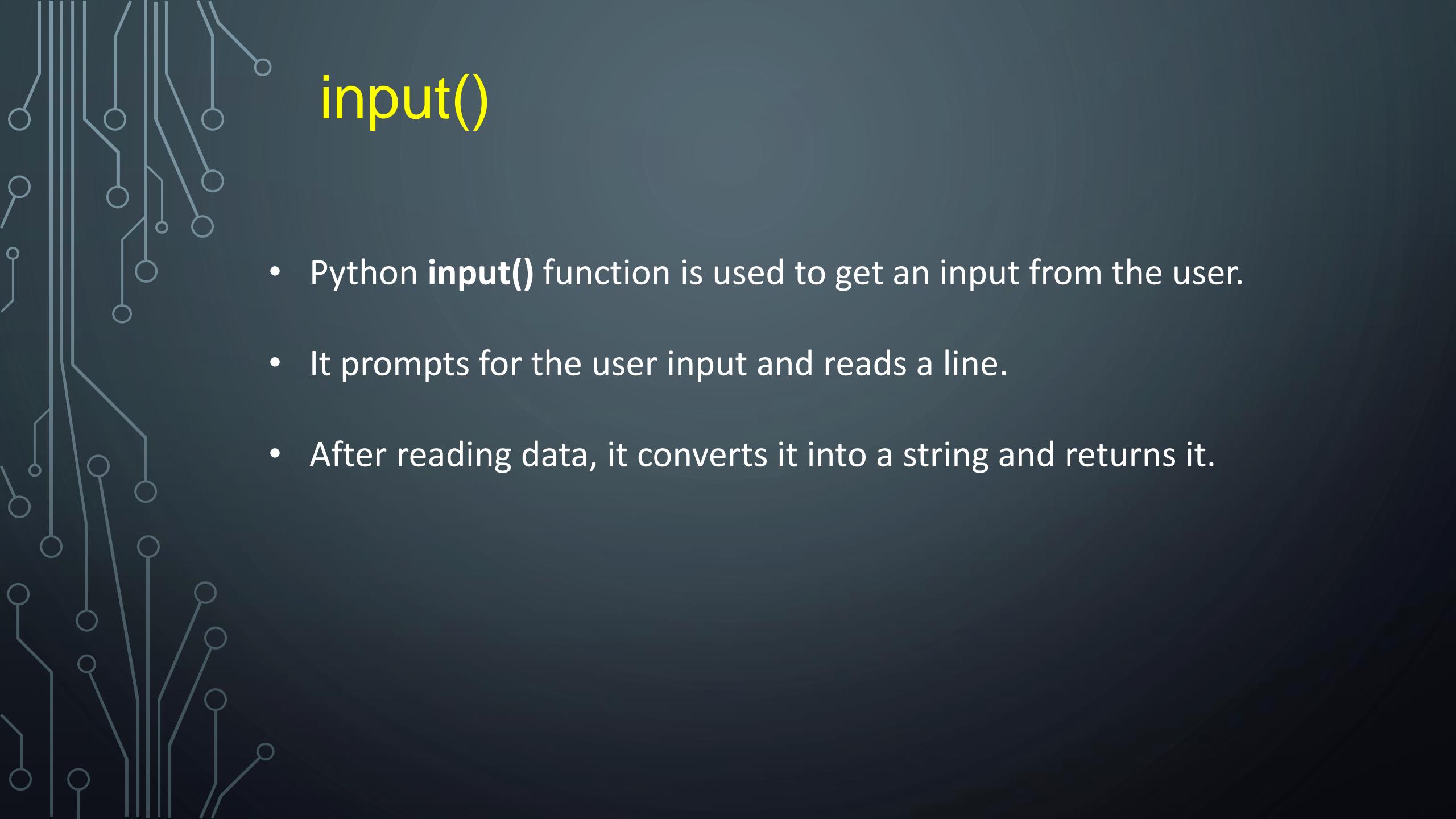
```
# integer number
integer = -20
print('Absolute value of -20 is:', abs(integer))
```

Absolute value of -20 is: 20

In [20]: ►

```
# floating number
floating = -20.83
print('Absolute value of -20.83 is:', abs(floating))
```

Absolute value of -20.83 is: 20.83



# input()

- Python **input()** function is used to get an input from the user.
- It prompts for the user input and reads a line.
- After reading data, it converts it into a string and returns it.

In [29]:

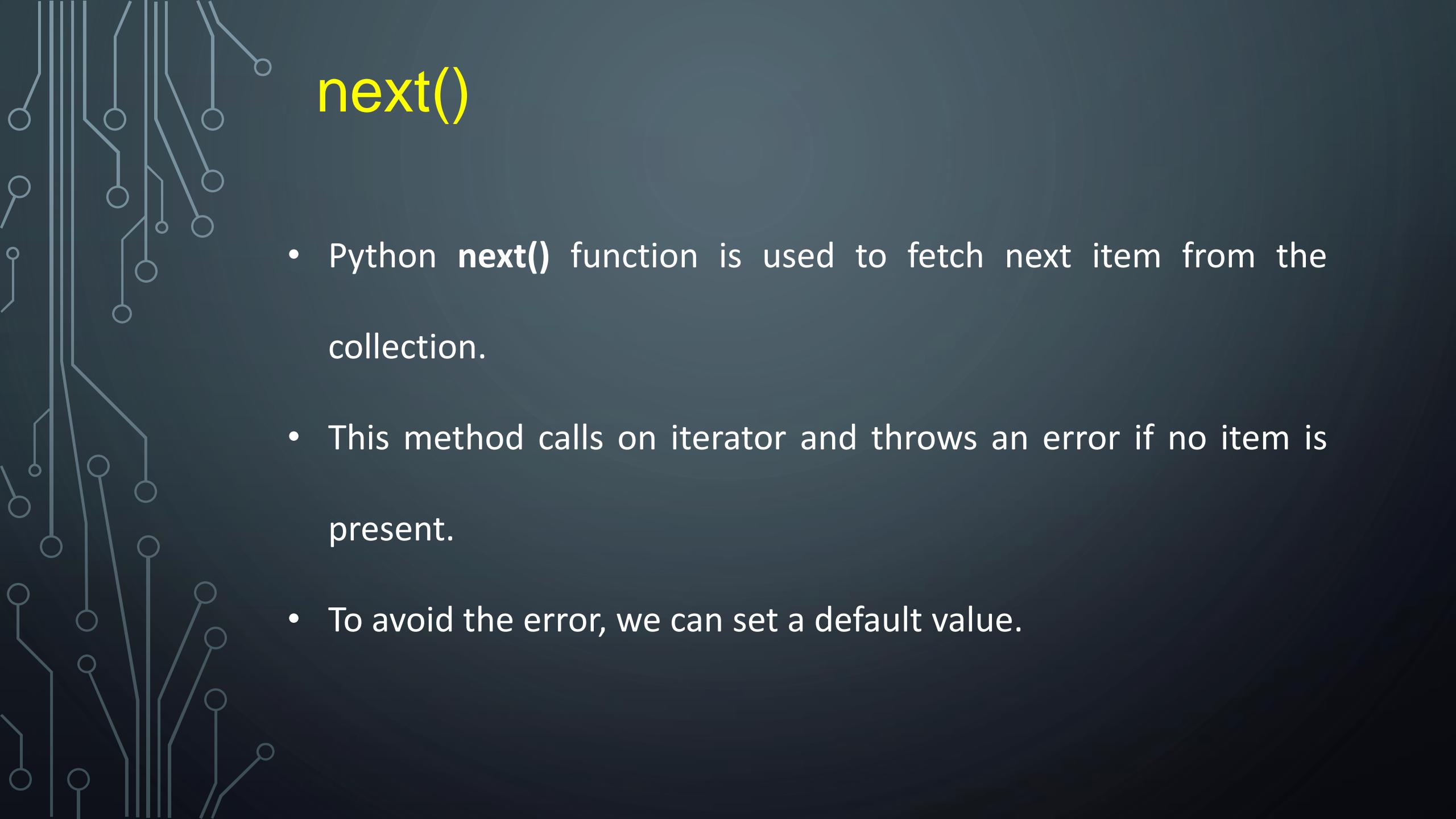
```
# Calling function
val = input("Enter a value: ")
# Displaying result
print("You entered:",val)
```

Enter a value: Marvel Universe  
You entered: Marvel Universe

In [30]:

```
# Calling function
val = input("Enter a value: ")
# Displaying result
print("You entered:",val)
```

Enter a value: 25.5  
You entered: 25.5



# next()

- Python **next()** function is used to fetch next item from the collection.
- This method calls on iterator and throws an error if no item is present.
- To avoid the error, we can set a default value.

```
In [43]: ► number = iter([256, 32, 82]) # Creating iterator
```

```
In [44]: ► item = next(number)  
print(item)
```

256

```
In [45]: ► item = next(number)  
print(item)
```

32

```
In [46]: ► item = next(number)  
print(item)
```

82

```
In [47]: ► item = next(number)  
print(item)
```

---

**StopIteration**

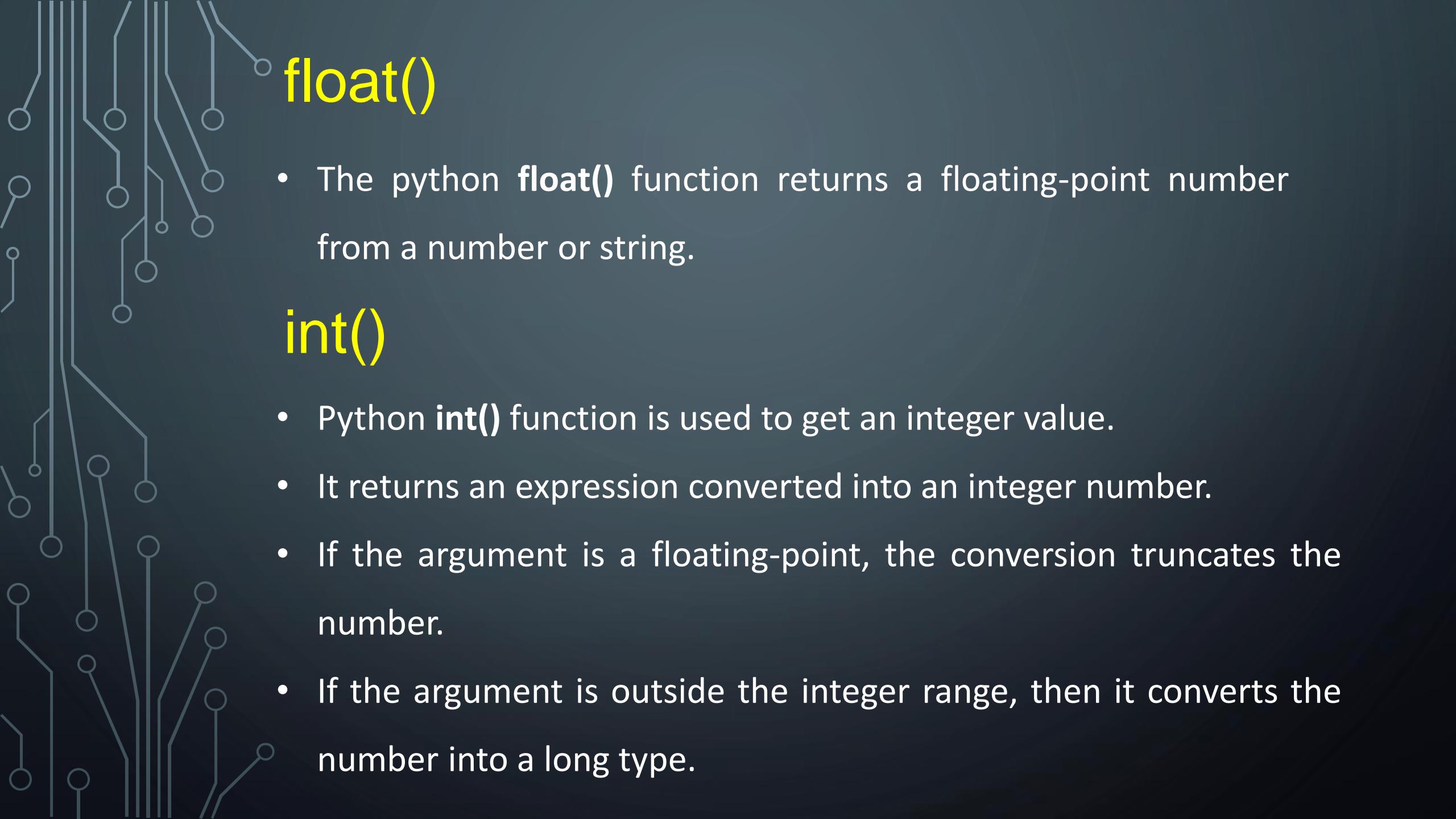
Traceback (most recent call last)

<ipython-input-47-abd3859f3f65> in <module>

----> 1 item = next(number)

2 print(item)

**StopIteration:**

A faint, light-grey circuit board pattern serves as the background for the slide.

# float()

- The python **float()** function returns a floating-point number from a number or string.

# int()

- Python **int()** function is used to get an integer value.
- It returns an expression converted into an integer number.
- If the argument is a floating-point, the conversion truncates the number.
- If the argument is outside the integer range, then it converts the number into a long type.

```
In [48]: ► # for integers  
print(float(9))
```

9.0

```
In [49]: ► # for floats  
print(float(8.19))
```

8.19

```
In [50]: ► # for string floats  
print(float("-24.27"))
```

-24.27

```
In [51]: ► # for string floats with whitespaces  
print(float(" -17.19\n"))
```

-17.19

```
In [52]: ► # string float error  
print(float("xyz"))
```

```
-----  
ValueError                                                 Traceback (most recent call last)  
<ipython-input-52-fa89eb40175d> in <module>  
      1 # string float error  
----> 2 print(float("xyz"))
```

**ValueError**: could not convert string to float: 'xyz'

In [53]: ► val1 = int(10) # integer value  
print(val1)

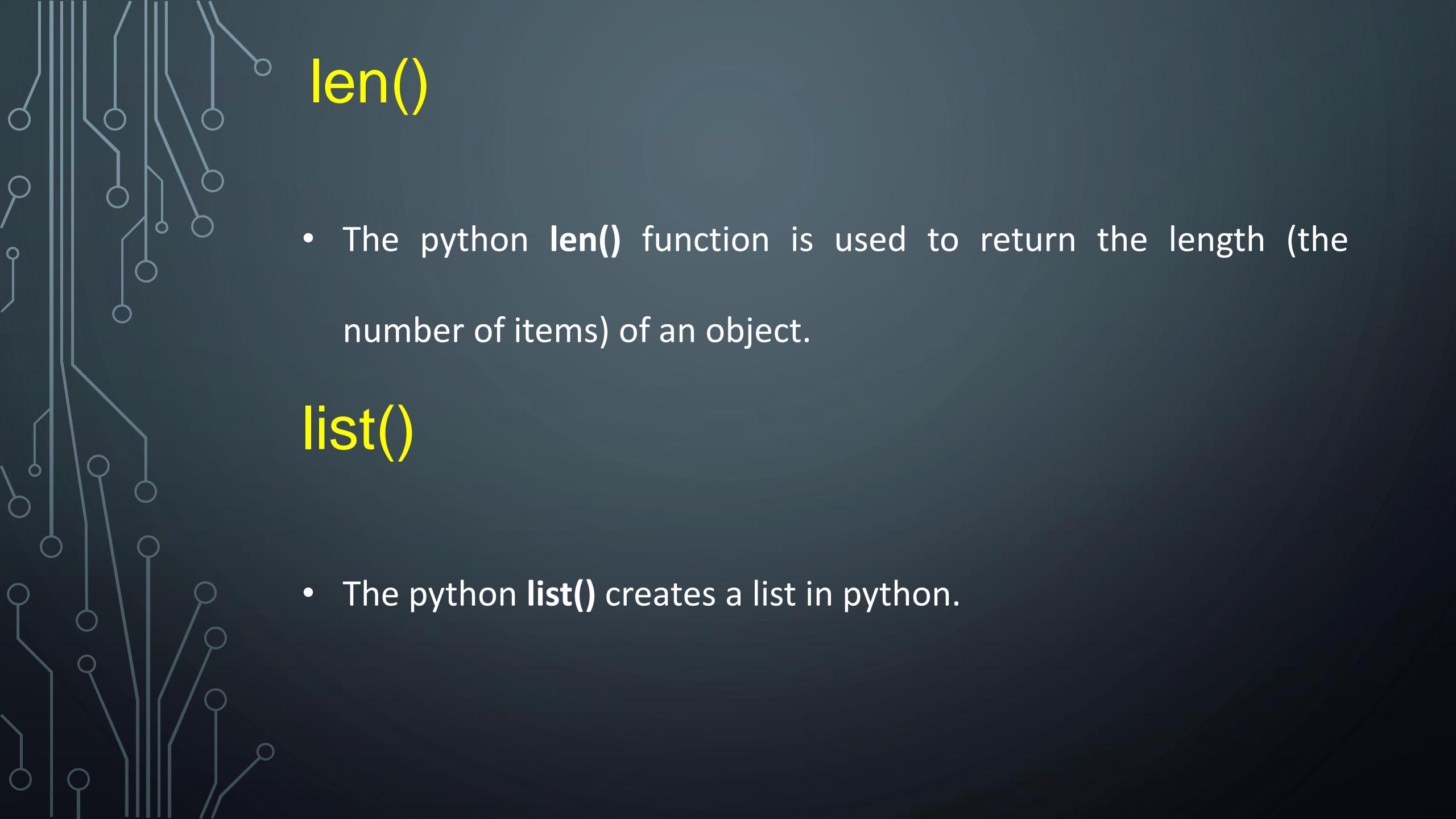
10

In [54]: ► val2 = int(10.52) # float value  
print(val2)

10

In [55]: ► val3 = int('10') # string value  
print(val3)

10

A repeating pattern of light gray circuit board tracks and circular pads forms the background of the slide.

## len()

- The python **len()** function is used to return the length (the number of items) of an object.

## list()

- The python **list()** creates a list in python.

In [61]: ► strA = 'Python'  
print(len(strA))

6

In [62]: ► List = [1,3,5,2,5]  
print(len(List))

5

In [63]: ► T = tuple([1,4,'RK',7,8.9])  
print(len(T))

5

In [64]: ► # empty list  
print(list())  
[]

In [65]: ► # string  
String = 'abcde'  
print(list(String))  
['a', 'b', 'c', 'd', 'e']

In [66]: ► # tuple  
Tuple = (1,2,3,4,5)  
print(list(Tuple))  
[1, 2, 3, 4, 5]

In [67]: ► # list  
List = [1,2,3,4,5]  
print(list(List))  
[1, 2, 3, 4, 5]



# divmod()

- Python **divmod()** function is used to get remainder and quotient of two numbers.
- This function takes two numeric arguments and returns a tuple.
- Both arguments are required and numeric

In [68]: ► A = divmod(10,2)  
print(A)

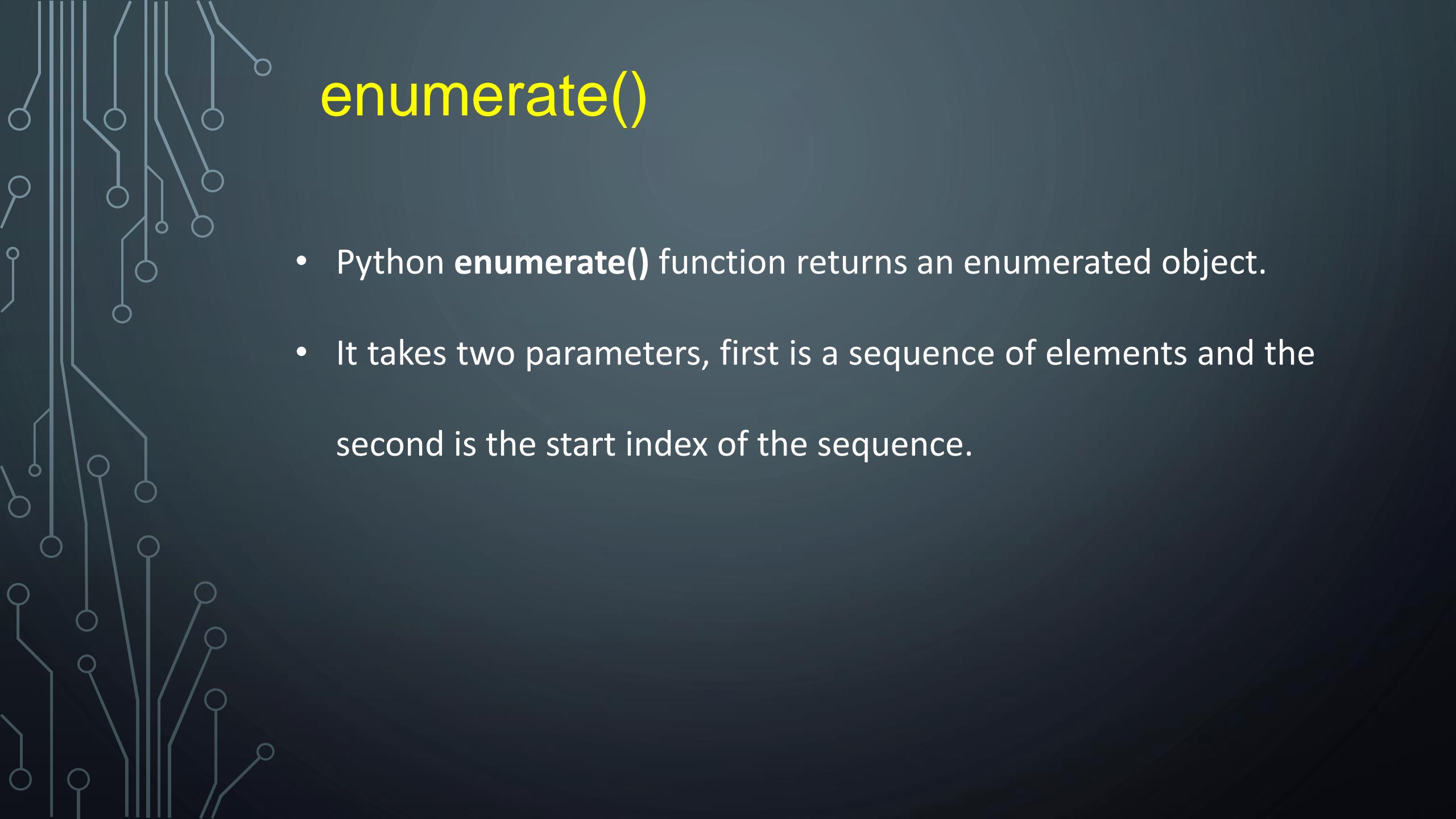
(5, 0)

In [69]: ► B = divmod(10,3)  
print(B)

(3, 1)

In [70]: ► A = divmod(33.45,5)  
print(A)

(6.0, 3.4500000000000003)

A faint, light-grey circuit board pattern serves as the background for the slide.

# enumerate()

- Python **enumerate()** function returns an enumerated object.
- It takes two parameters, first is a sequence of elements and the second is the start index of the sequence.

In [77]: ►

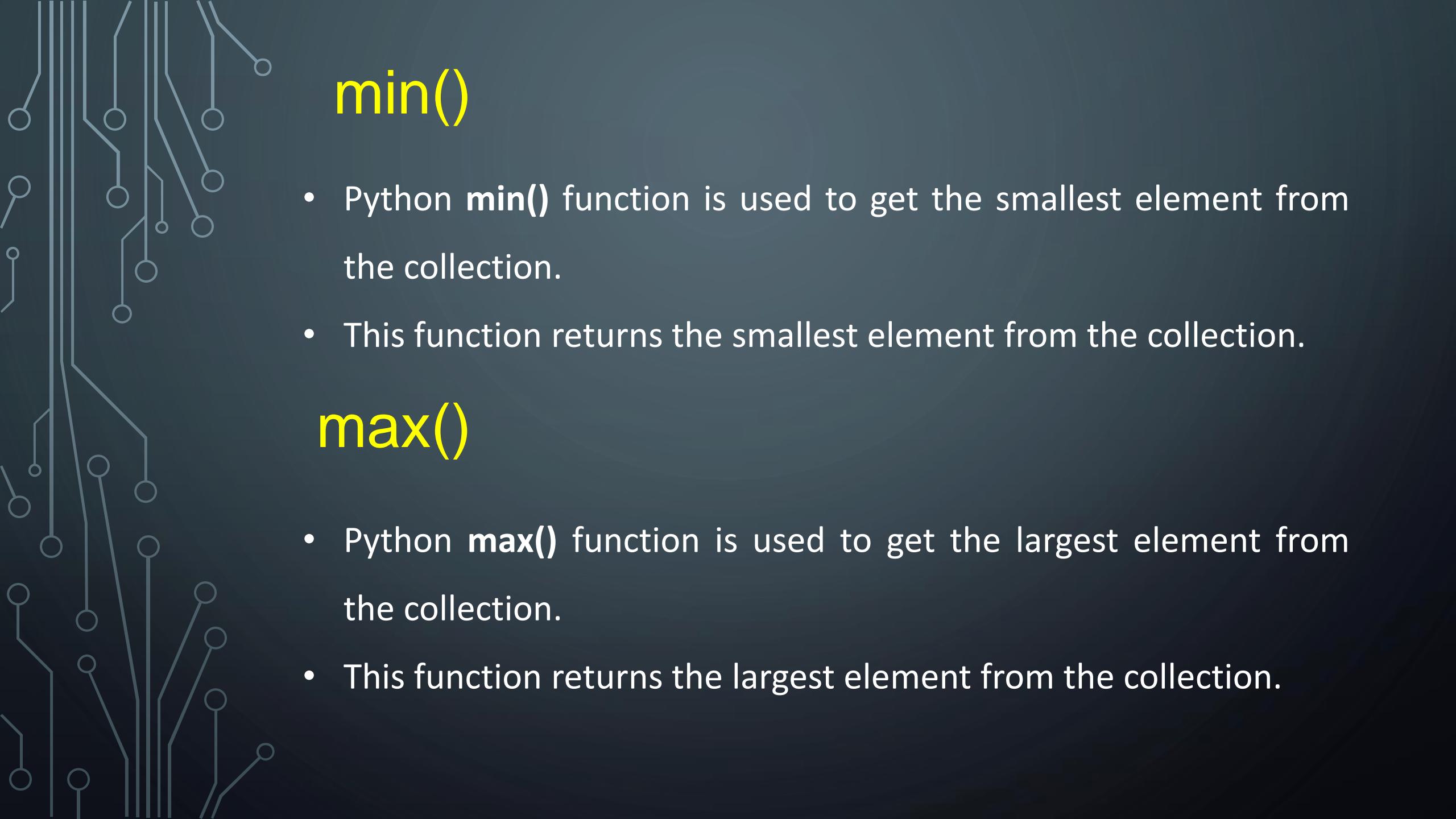
```
result = enumerate([1,2,3])
print(result)
print(list(result))
```

```
<enumerate object at 0x000002580CB0CAB0>
[(0, 1), (1, 2), (2, 3)]
```

In [78]: ►

```
result = enumerate(("A","B","C"))
print(result)
print(list(result))
```

```
<enumerate object at 0x000002580CB0CA68>
[(0, 'A'), (1, 'B'), (2, 'C')]
```

A repeating pattern of light gray circuit board tracks and circular pads forms the background of the slide.

## min()

- Python **min()** function is used to get the smallest element from the collection.
- This function returns the smallest element from the collection.

## max()

- Python **max()** function is used to get the largest element from the collection.
- This function returns the largest element from the collection.

In [80]: ► A\_1 = min(2225,325,2025)  
print(A\_1)

325

In [81]: ► A\_2 = min(1000.25,2025.35,5625.36,10052.50)  
print(A\_2)

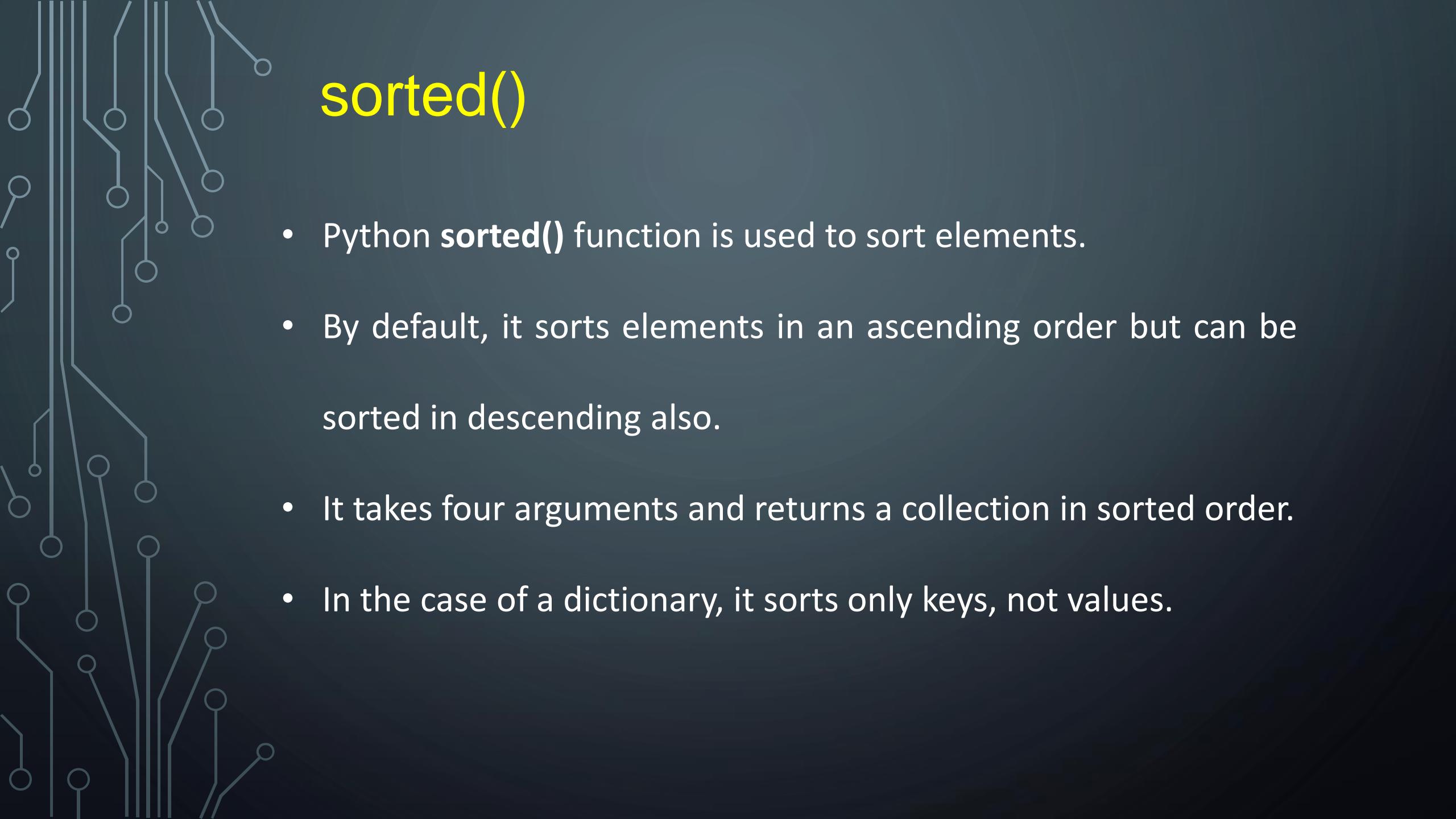
1000.25

In [82]: ► B\_1 = max(2225,325,2025)  
print(B\_1)

2225

In [83]: ► B\_2 = min(1000.25,2025.35,5625.36,10052.50)  
print(B\_2)

1000.25



# sorted()

- Python **sorted()** function is used to sort elements.
- By default, it sorts elements in an ascending order but can be sorted in descending also.
- It takes four arguments and returns a collection in sorted order.
- In the case of a dictionary, it sorts only keys, not values.

In [1]: ► str = "Marvel" # declaring string  
sorted1 = sorted(str) # sorting string  
print(sorted1)

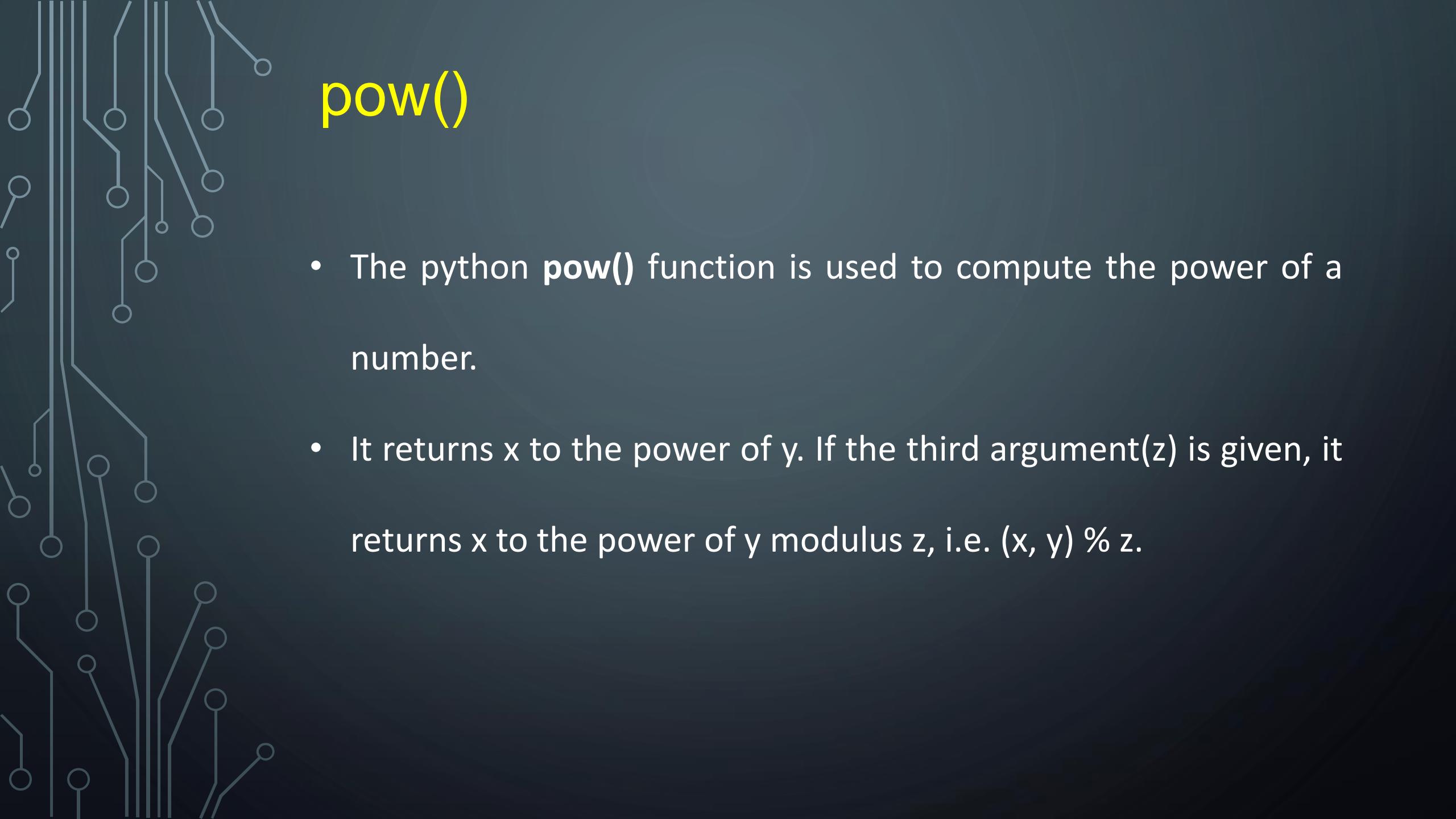
[ 'M', 'a', 'e', 'l', 'r', 'v' ]

In [2]: ► list = [1,5,8,2,5,7,4.6]  
s = sorted(list)  
print(s)

[1, 2, 4.6, 5, 5, 7, 8]

In [3]: ► list = [1,5,8,2,5,7,4.6]  
s = sorted(list, reverse=True)  
print(s)

[8, 7, 5, 5, 4.6, 2, 1]

A faint, light-grey circuit board pattern serves as the background for the slide.

# pow()

- The python **pow()** function is used to compute the power of a number.
- It returns  $x$  to the power of  $y$ . If the third argument( $z$ ) is given, it returns  $x$  to the power of  $y$  modulus  $z$ , i.e.  $(x, y) \% z$ .

```
In [4]: ► # positive x, positive y (x**y)  
print(pow(4, 2))
```

16

```
In [5]: ► # negative x, positive y  
print(pow(-4, 2))
```

16

```
In [6]: ► # positive x, negative y (x**-y)  
print(pow(4, -2))
```

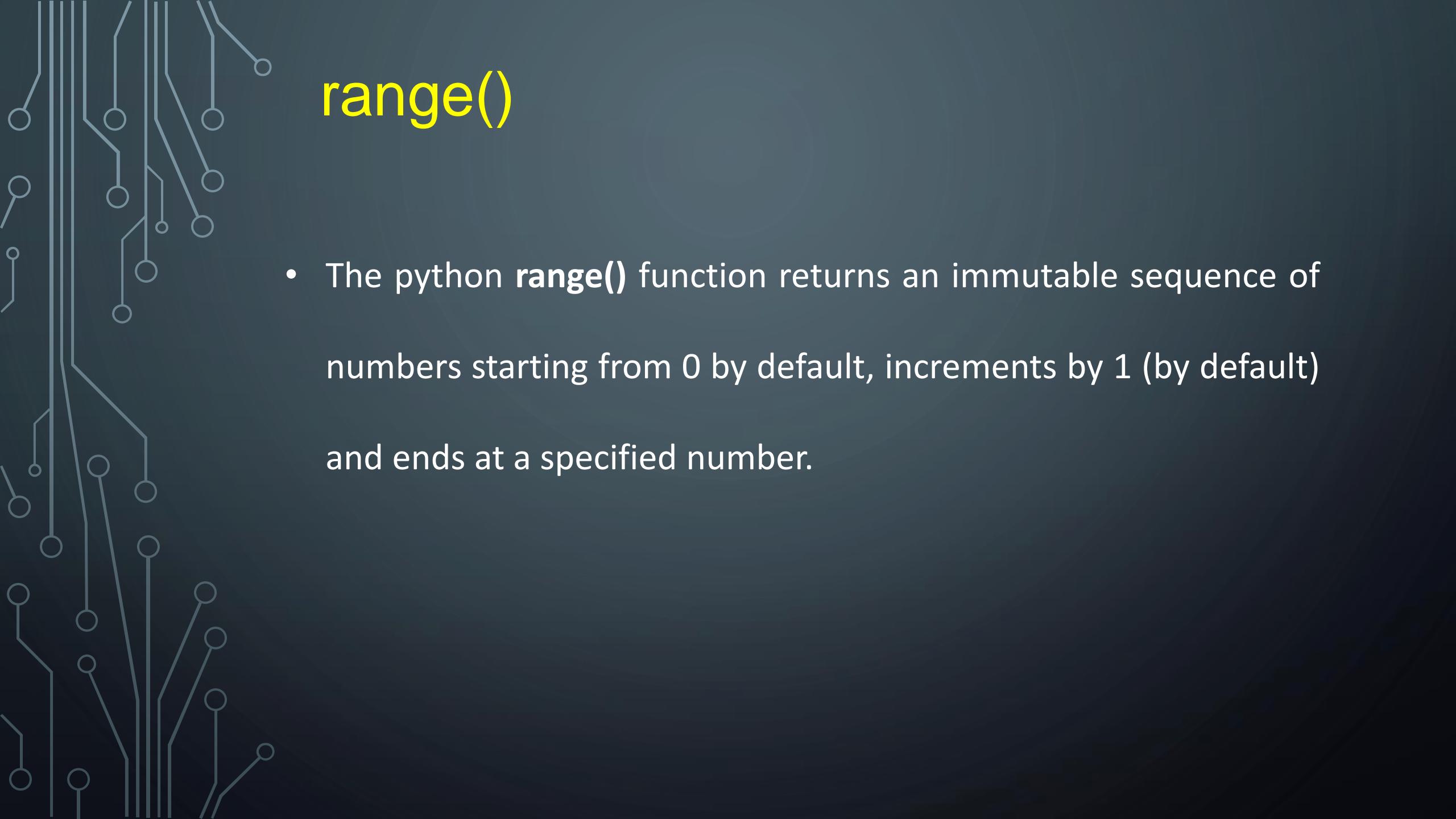
0.0625

```
In [7]: ► # negative x, negative y  
print(pow(-4, -2))
```

0.0625

```
In [10]: ► '''1st & 2nd argument using in power function  
and result of that power modulus with  
the 3rd argument'''
```

```
print(pow(4, 2, 3))
```

A faint, light-gray circuit board pattern serves as the background for the slide.

# range()

- The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

In [38]: ► L1 = range(0)  
print(L1)

```
range(0, 0)
```

In [39]: ► L2 = range(4)  
print(L2)

```
range(0, 4)
```

In [40]: ► for i in L2:  
 print(i)

```
0  
1  
2  
3
```

```
In [41]: ► # using the range(start, stop)
L3 = range(1,4)
print(L3)
```

```
range(1, 4)
```

```
In [42]: ► for i in L3:
    print(i)
```

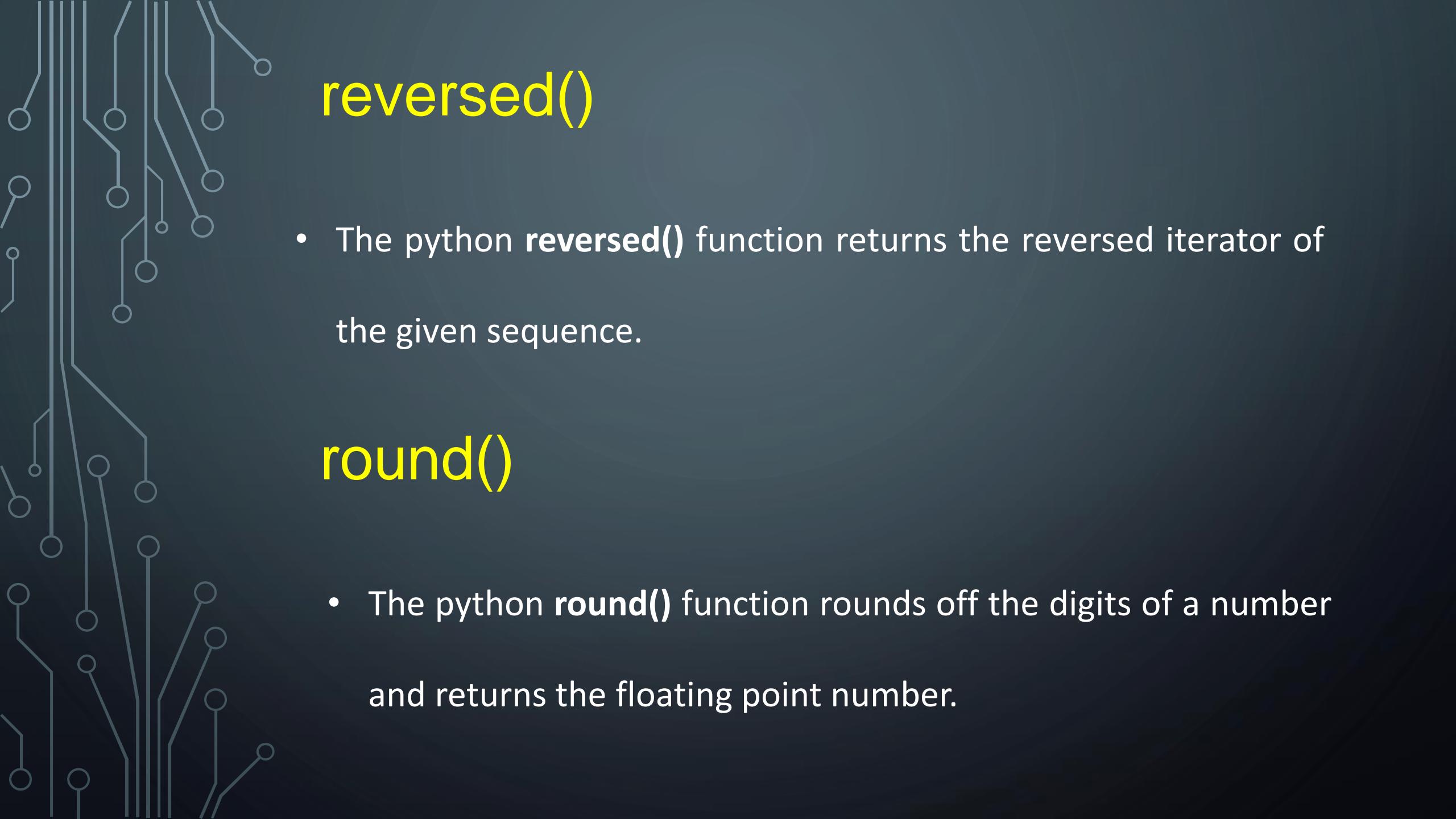
```
1
2
3
```

```
In [43]: ► # using the range(start, stop, step)
L4 = range(1,4,2)
print(L4)
```

```
range(1, 4, 2)
```

```
In [36]: ► for i in L4:
    print(i)
```

```
1
3
```

A repeating pattern of light gray circuit board tracks and component pads forms the background of the slide.

## reversed()

- The python **reversed()** function returns the reversed iterator of the given sequence.

## round()

- The python **round()** function rounds off the digits of a number and returns the floating point number.

In [6]: ► String = 'Python'

```
print(list(reversed(String)))
```

```
['n', 'o', 'h', 't', 'y', 'P']
```

In [7]: ► Tuple = ('n', 'o', 'h', 't', 'y', 'P')

```
print(list(reversed(Tuple)))
```

```
['P', 'y', 't', 'h', 'o', 'n']
```

In [8]: ► Range = range(8, 12)

```
print(list(reversed(Range)))
```

```
[11, 10, 9, 8]
```

In [9]: ► List = [1, 2, 7, 5]

```
print(list(reversed(List)))
```

```
[5, 7, 2, 1]
```

In [10]: ► print(round(10))

10

In [11]: ► print(round(10.8))

11

In [12]: ► print(round(6.6))

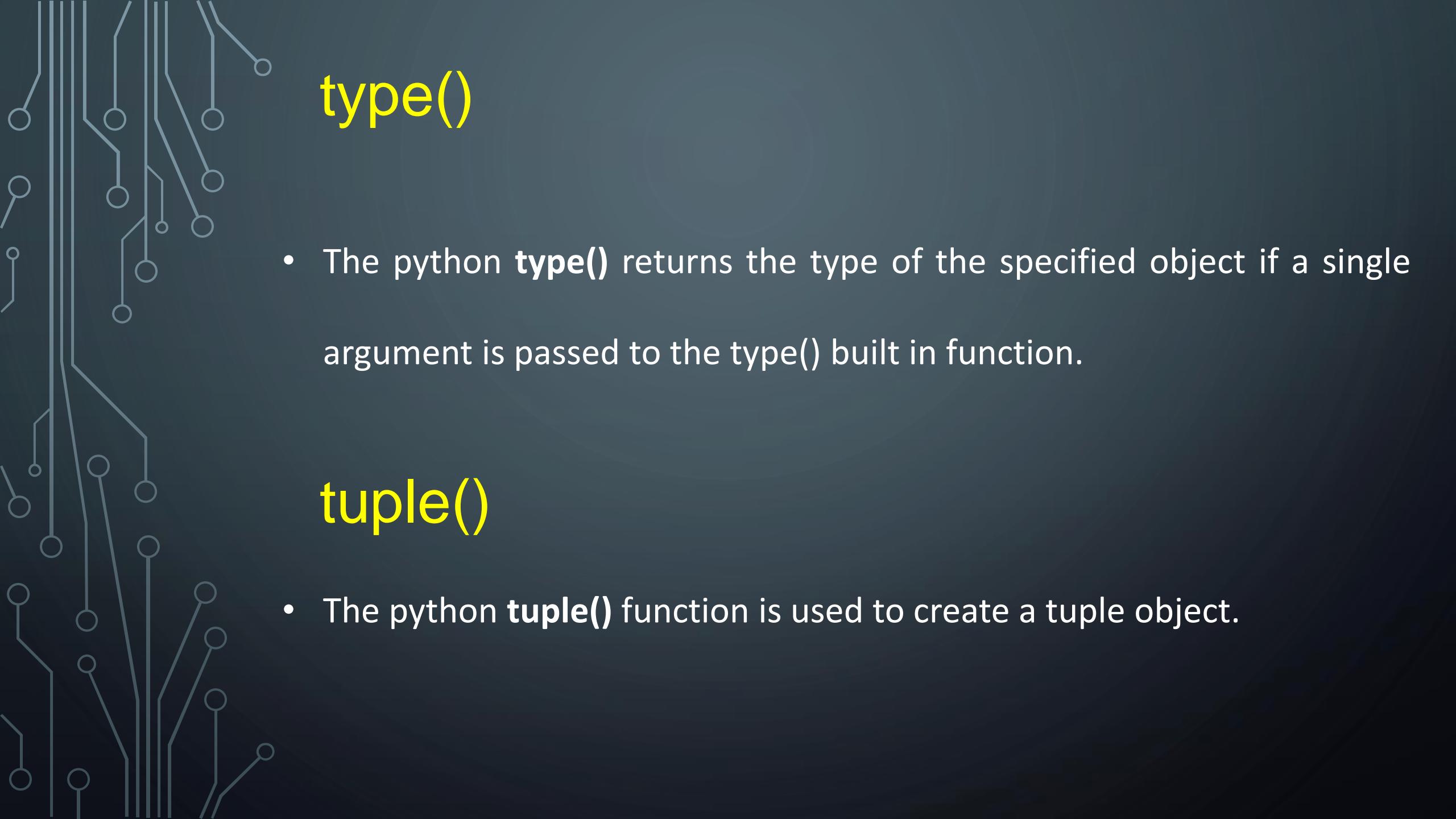
7

In [13]: ► print(round(6.5))

6

In [14]: ► print(round(6.4))

6

A dark blue background featuring a faint, repeating circuit board or network diagram pattern in light grey.

## type()

- The python **type()** returns the type of the specified object if a single argument is passed to the type() built in function.

## tuple()

- The python **tuple()** function is used to create a tuple object.

In [23]: ► List = [4, 5]  
print(type(List))

<class 'list'>

In [24]: ► Dict = {4: 'four', 5: 'five'}  
print(type(Dict))

<class 'dict'>

In [25]: ► class Python:  
 a = 0  
  
InstanceOfPython = Python()  
print(type(InstanceOfPython))

<class '\_\_main\_\_.Python'>

In [19]: ► t1 = tuple()  
print('t1=', t1)

t1= ()

In [20]: ► # creating a tuple from a List  
t2 = tuple([1, 6, 9])  
print('t2=', t2)

t2= (1, 6, 9)

In [21]: ► # creating a tuple from a string  
t1 = tuple('Java')  
print('t1=', t1)

t1= ('J', 'a', 'v', 'a')

In [22]: ► # creating a tuple from a dictionary  
t1 = tuple({4: 'four', 5: 'five'})  
print('t1=', t1)

t1= (4, 5)



## vars()

- The python **vars()** function returns the `__dict__` attribute of the given object.

## zip()

- The python **zip()** Function returns a zip object, which maps a similar index of multiple containers.

```
In [1]: ► class Python:  
        def __init__(self, x = 7, y = 9):  
            self.x = x  
            self.y = y
```

```
In [2]: ► InstanceOfPython = Python()  
        print(vars(InstanceOfPython))  
  
        {'x': 7, 'y': 9}
```

```
In [7]: ► test = zip()  
# referring a zip class  
print('The type of an empty zip : ', type(test))  
  
list1 = ['Alpha', 'Beta', 'Gamma', 'Sigma']  
list2 = ['one', 'two', 'three', 'six']  
  
test = zip(list1, list2) # zip the values  
  
print('\nPrinting the values of zip')  
for values in test:  
    print(values) # print each tuples
```

The type of an empty zip : <class 'zip'>

Printing the values of zip

('Alpha', 'one')  
('Beta', 'two')  
('Gamma', 'three')  
('Sigma', 'six')

```
In [11]: ► numList = [4,5, 6]
          strList = ['four', 'five', 'six']

          result = zip(numList, strList)
```

```
In [12]: ► for i in result:
              print(i)
```

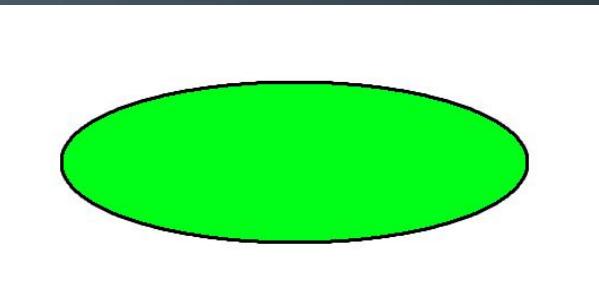
```
(4, 'four')
(5, 'five')
(6, 'six')
```

# Flow Chart

- Flowchart is a graphical representation of an algorithm.
- Programmers often use it as a program-planning tool to solve a problem.
- It makes use of symbols which are connected among them to indicate the flow of information and processing.

# Flow Chart Symbols

- **Terminal:** The oval symbol indicates Start, Stop and Halt in a program's logic flow.

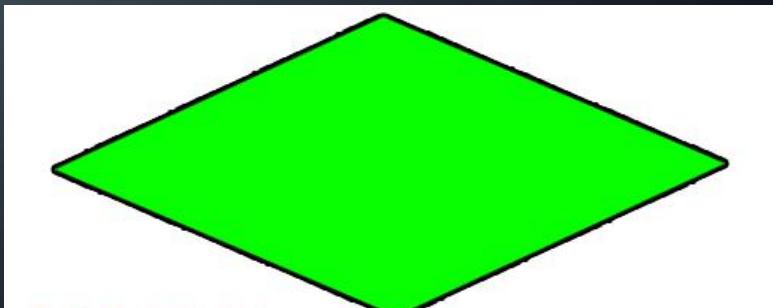
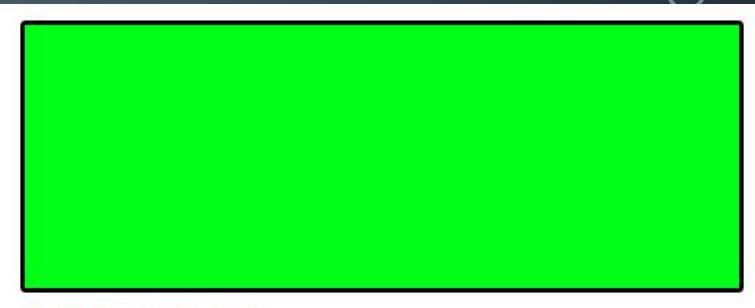


- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



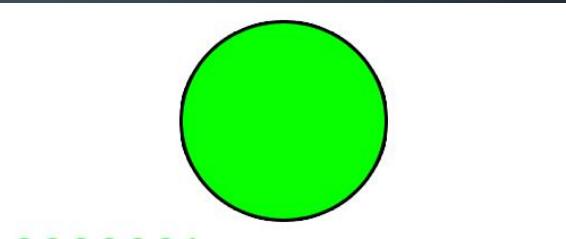
# Flow Chart Symbols

- **Processing:** A box represents arithmetic instructions.  
All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.
- **Decision** Diamond symbol represents a decision point.  
Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.

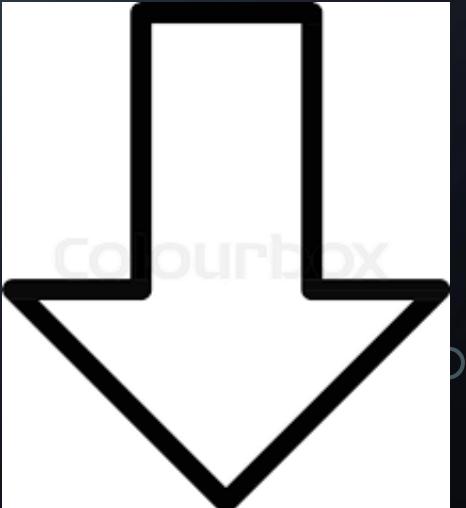


# Flow Chart Symbols

- **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.

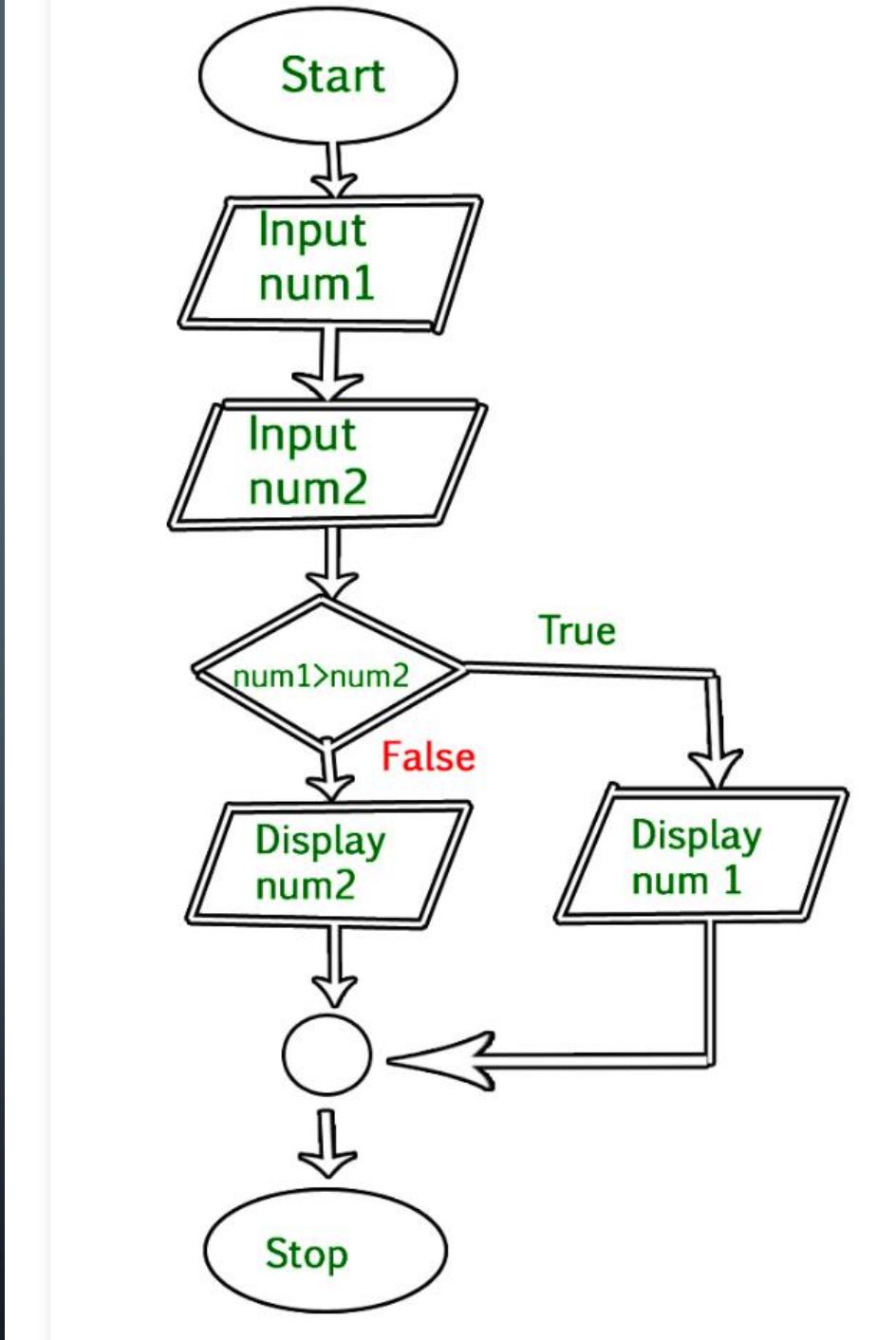


- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



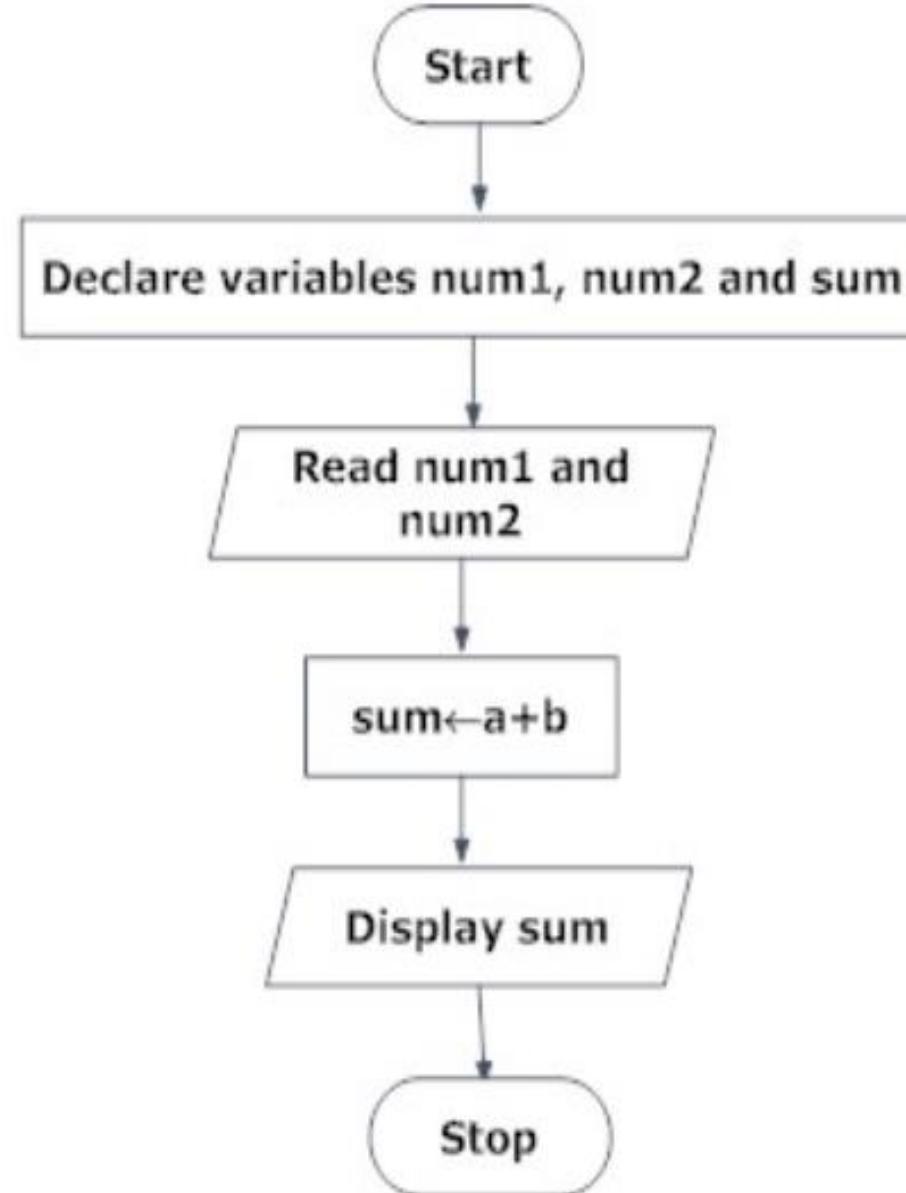
# Flow Chart Example

- Program to find greater number among given two numbers num1 & num2.



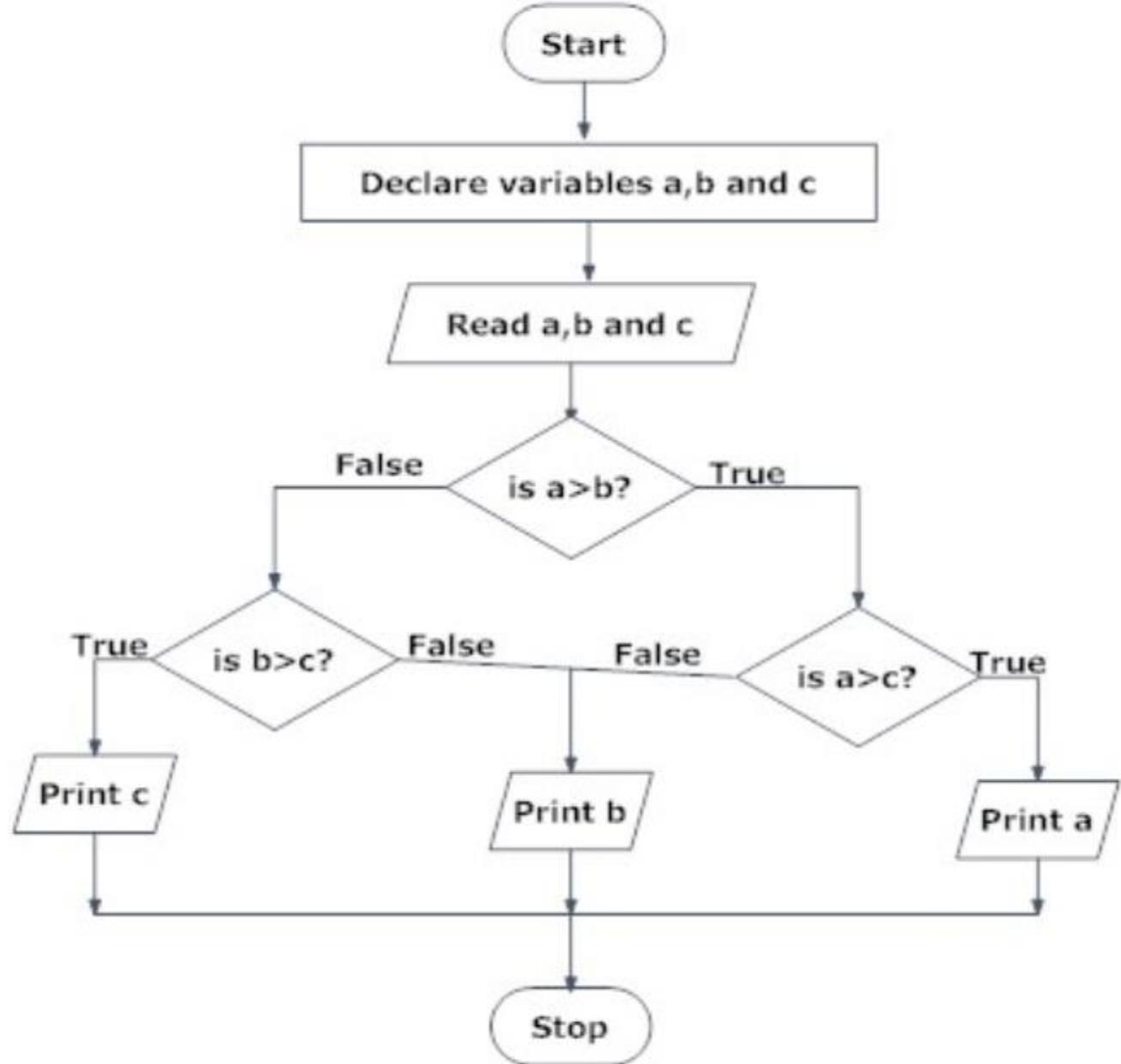
# Flow Chart Example

- Flow chart to add two number, input enter by user.



# Flow Chart Example

- Program to find the largest among three different numbers entered by user.



# Operator Precedence

$2 + 3 * 4$

- We can add 2 and 3, then multiply the result by 4. Also, we can multiply 3 and 4 first, then add 2 with it. Here we can see that the operators' precedence is important.

# Operator Precedence

Parenthesis	( ) or { } or [ ]
Exponentiation	**
Multiply, Divide, modulo	%,/,*,
Addition and Subtraction	+,-
Right and Left Shift	>>,<<
Bitwise AND	&
Bitwise OR and XOR	,^
Comparison Operators	==,!!=,>,<,>=,<=
Assignment Operator	=

In [1]: ➤ 3 + 3 / 3

Out[1]: 4.0

In [2]: ➤ (3+3)/3

Out[2]: 2.0

In [3]: ➤ (((((13+5)\*2)-4)/2)-13

Out[3]: 3.0

In [6]: ► 3\*5//4  
# 3\*5 gives us 15  
# 15//4 gives us 3

Out[6]: 3

In [5]: ► 3\*(5//4)  
# 5//4 gives us 1  
# 3\*1 gives us 3

Out[5]: 3

In [7]: ► (2\*\*3)\*\*2

Out[7]: 64

In [9]: ► 2\*\*3\*\*2  
# This gives us 2\*\*9  
# This gives us 512

Out[9]: 512

*Thank you ...*

*Your Best Lesson is your last  
mistake .....*

*Never Give Up*

*Go with the Green*

