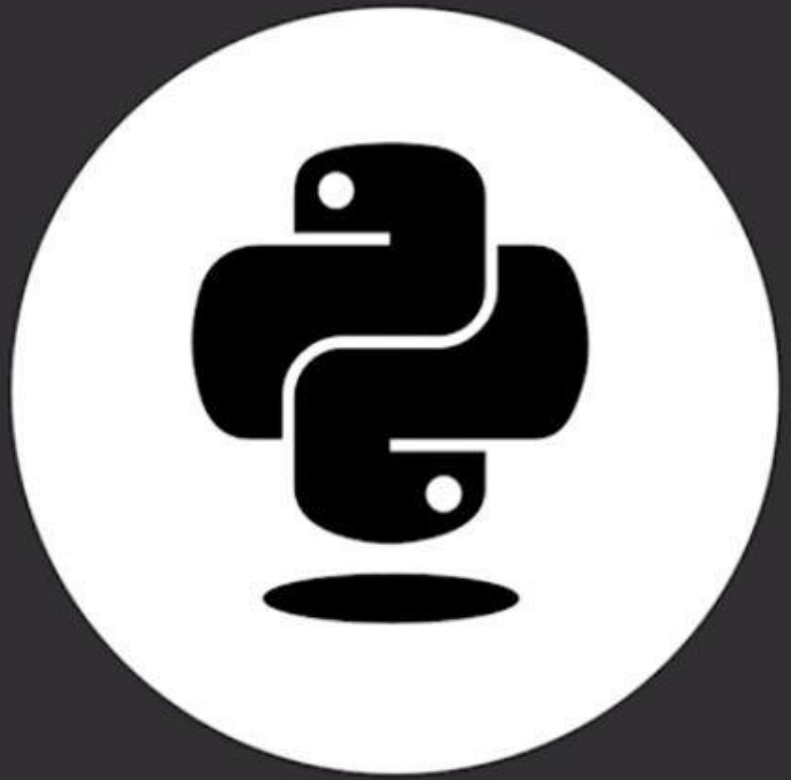
TOPIC 1

Introduction To Python

Static.Int Educare

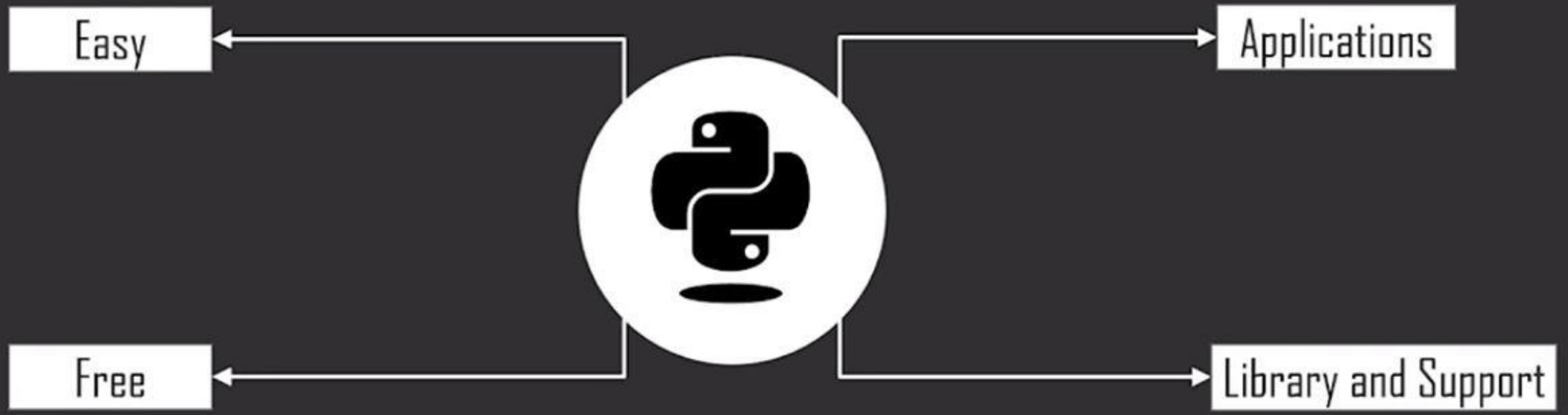
What is Python?



- Python is a **general** purpose, **high level, interpreted language** with **easy** syntax and **dynamic** semantics
- Created by **Guido Van Rossum** in 1989

Static.Int Educare

Why is Python popular?



Python has a great **community** who constantly make libraries and **help** those in need

Static.Int Educare

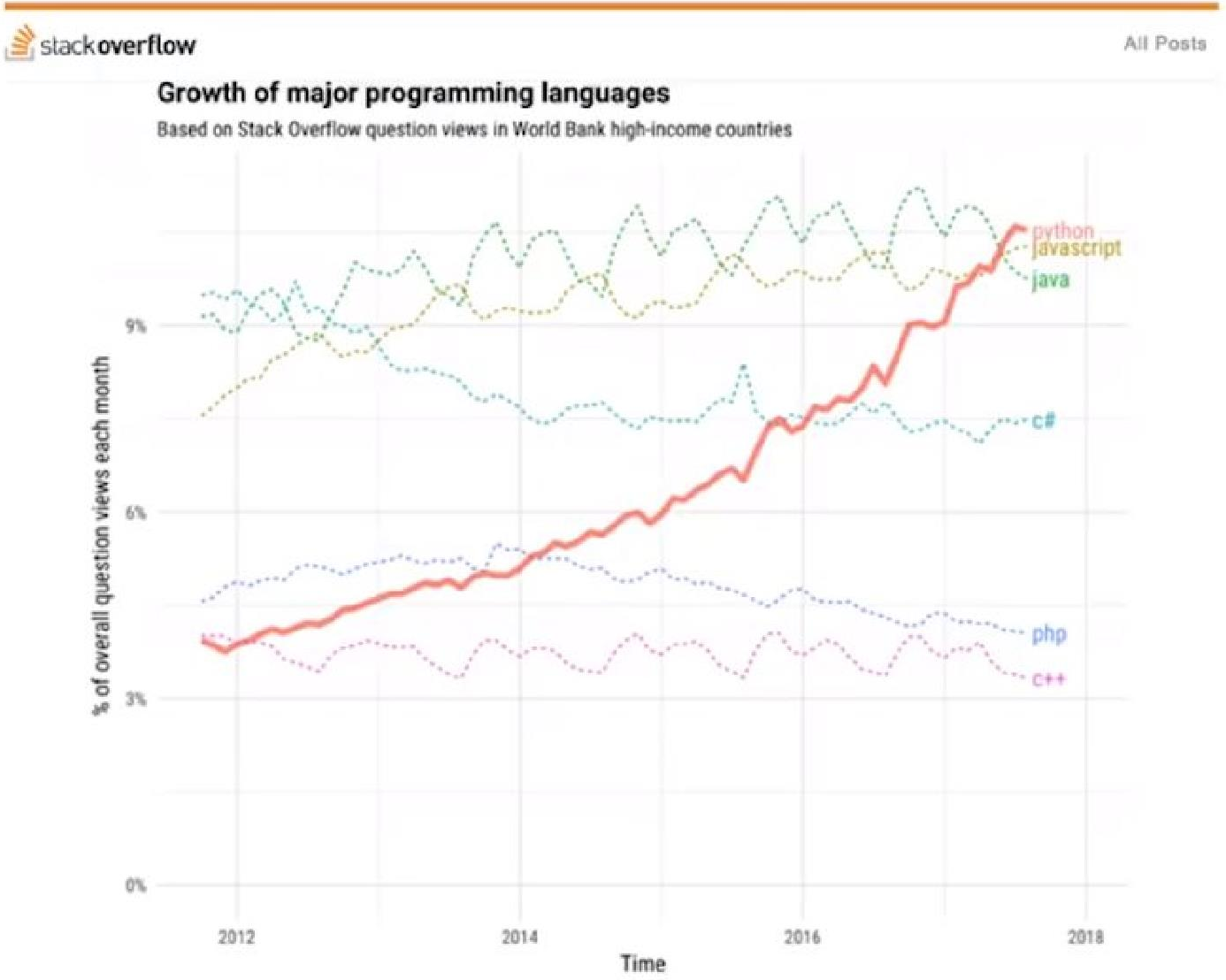
Features of Python

Static.Int Educare

Career Opportunities

Static.Int Educare

Popularity

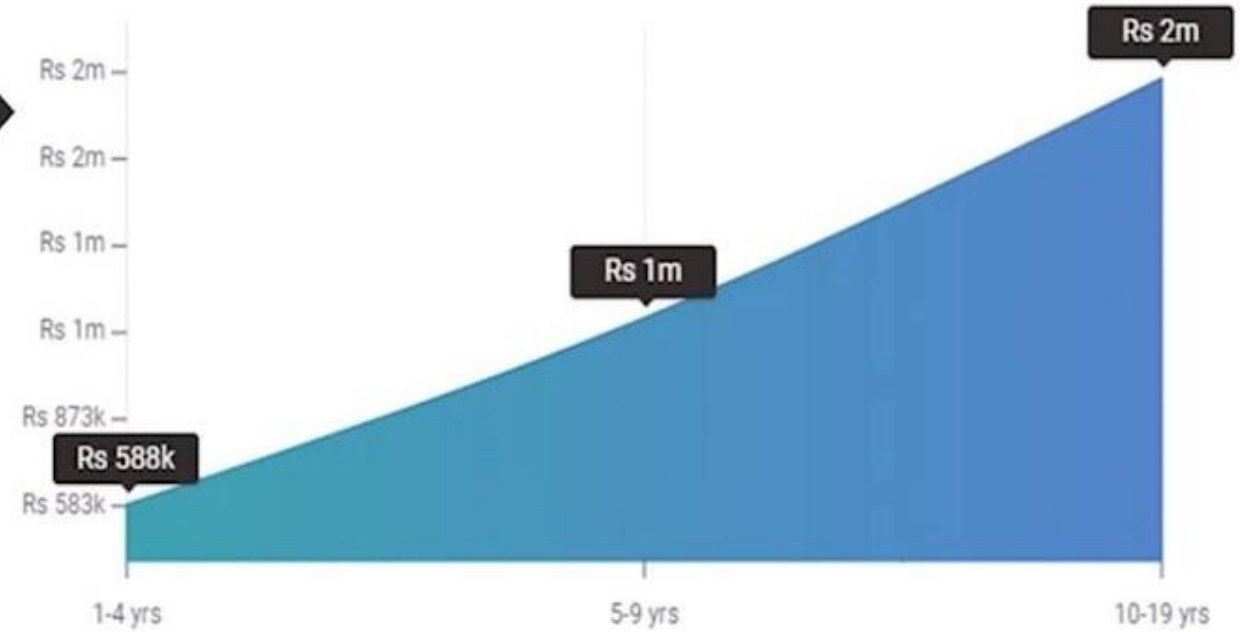


A few companies that use Python.

Static.Int Educare

Experience

Senior Python Developer earns about 800K+



Eg: To print Hello world:

Java:

```
public class HelloWorld
 {
public static void main(String[] args)
 {
 System.out.println("Hello world");
}}
```

In C:

```
 #include<stdio.h>
void main() {
printf("Hello world");
}
```

In python

```
print("Hello World")
```

Static.Int Educare

Static.Int Educare

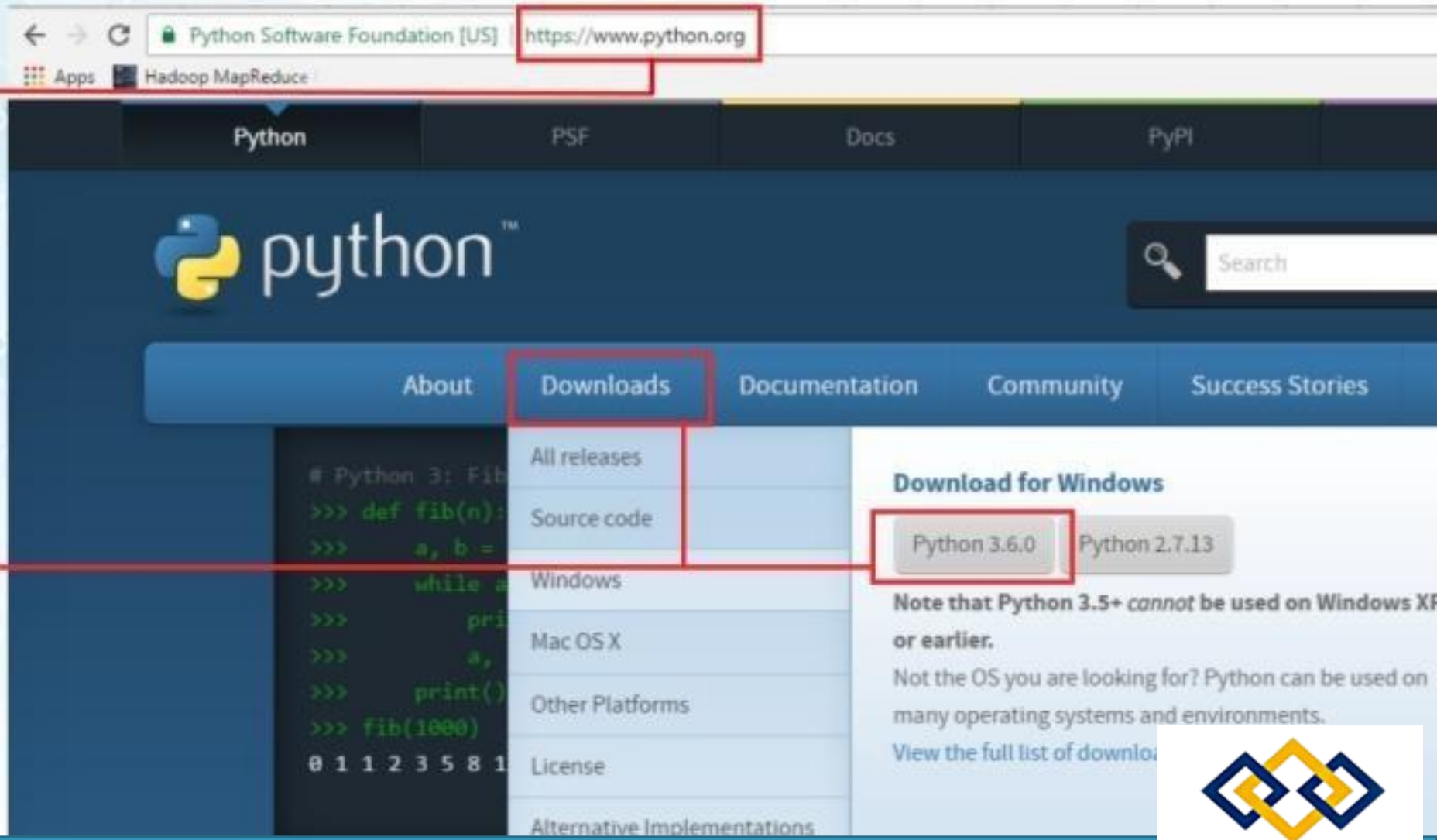
TOPIC 2

Installation

Python Installation

1. Go to www.python.org

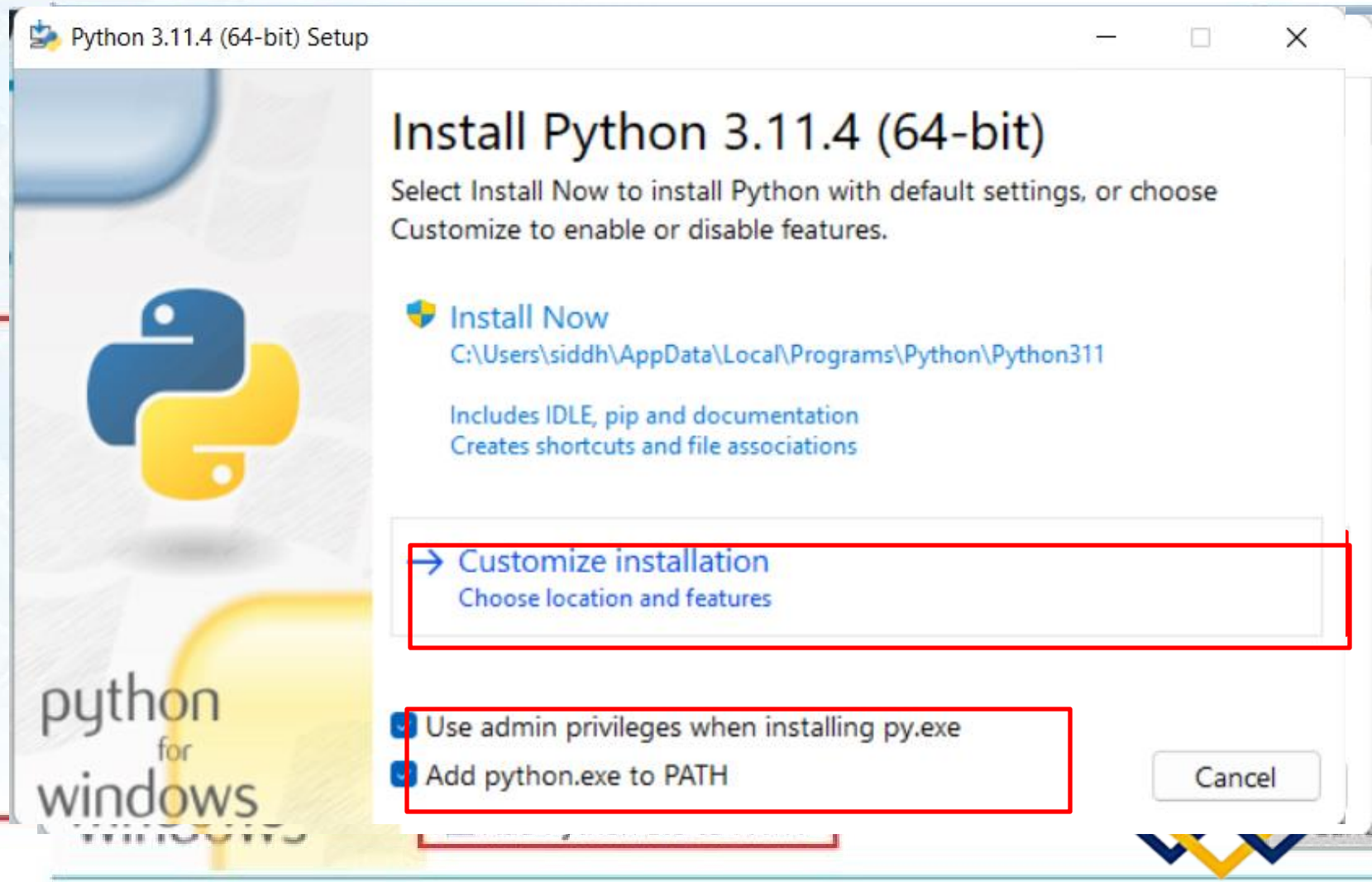
2. Under Downloads tab, select the latest Python version for Windows



Static.Int Educare

Python Installation

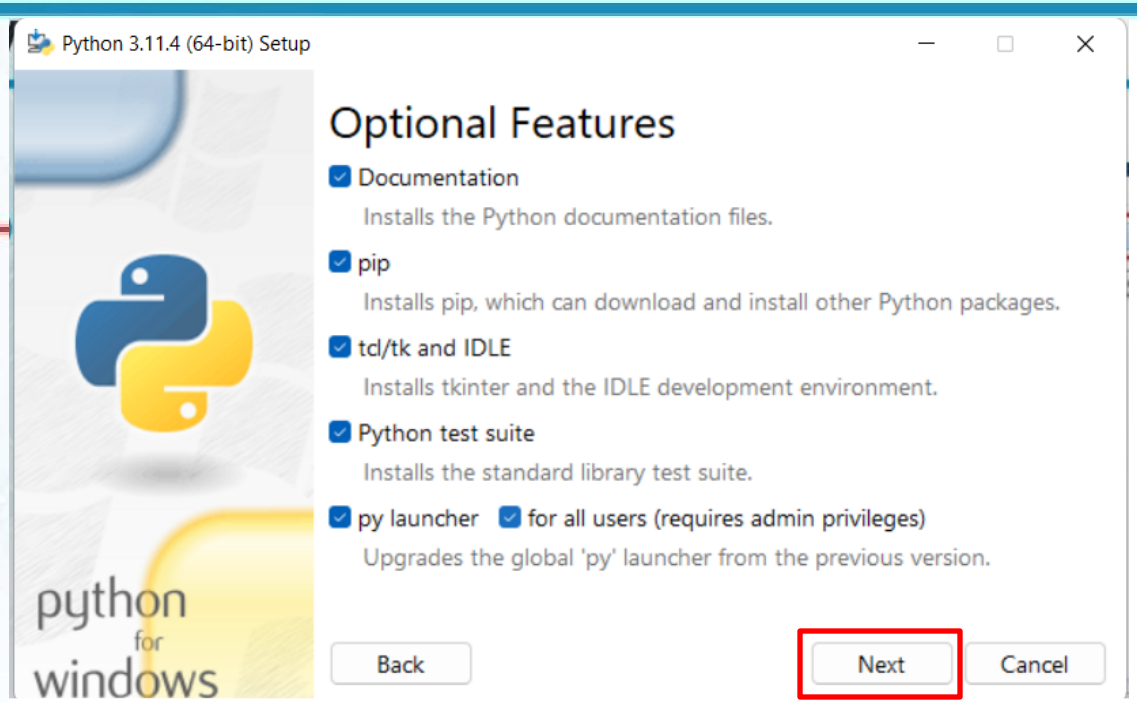
5 Click on customize installation

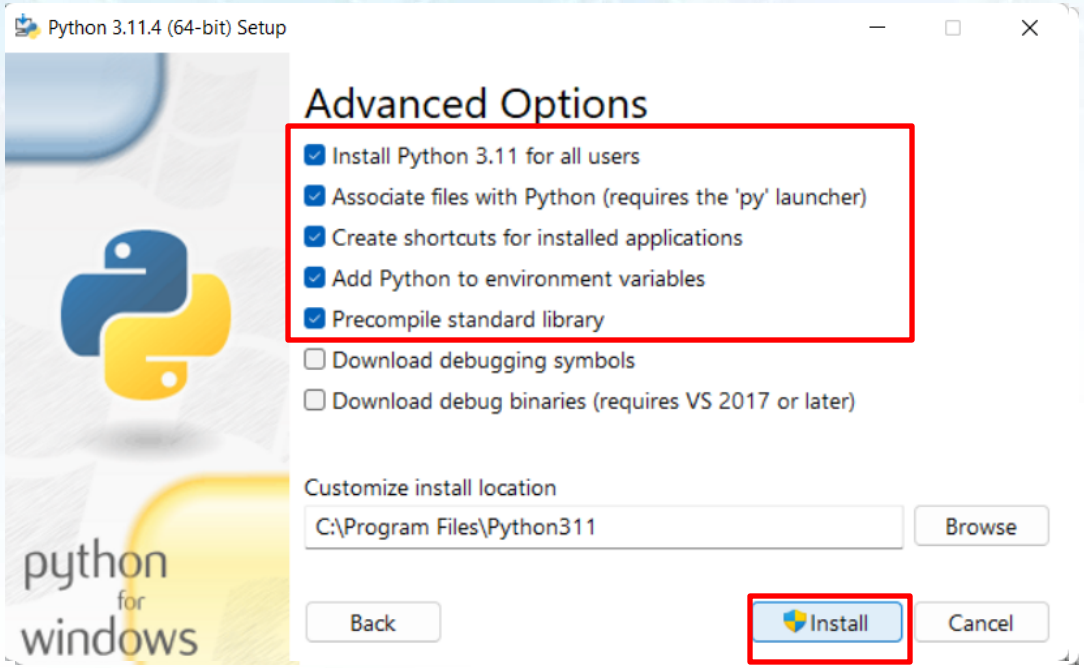


4 Select 'Add Python 3.6 to PATH'

Python Installation

6 Click on Next, then select install python for all users





Download the latest version for macOS

Download Python 3.12.3

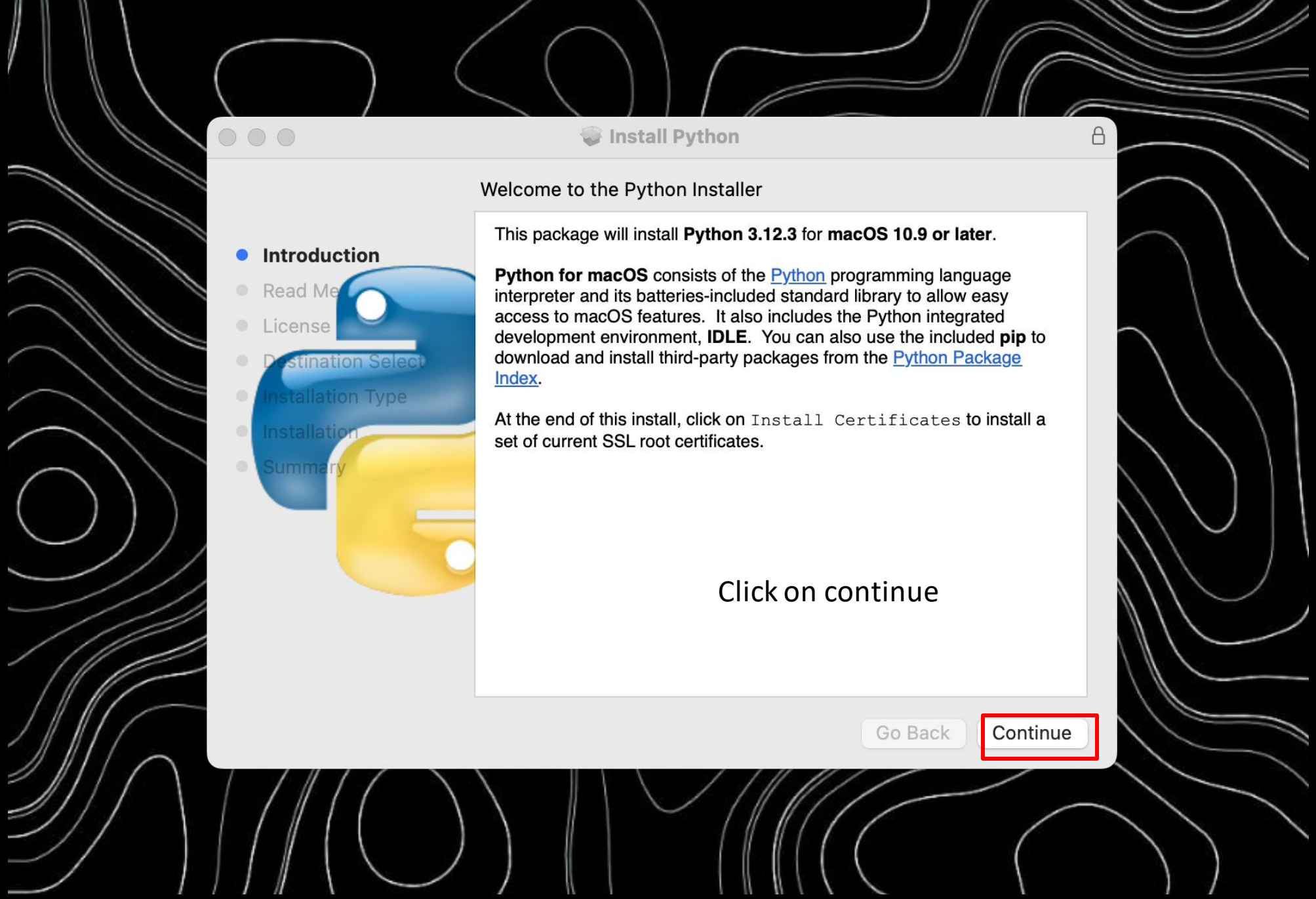
Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other

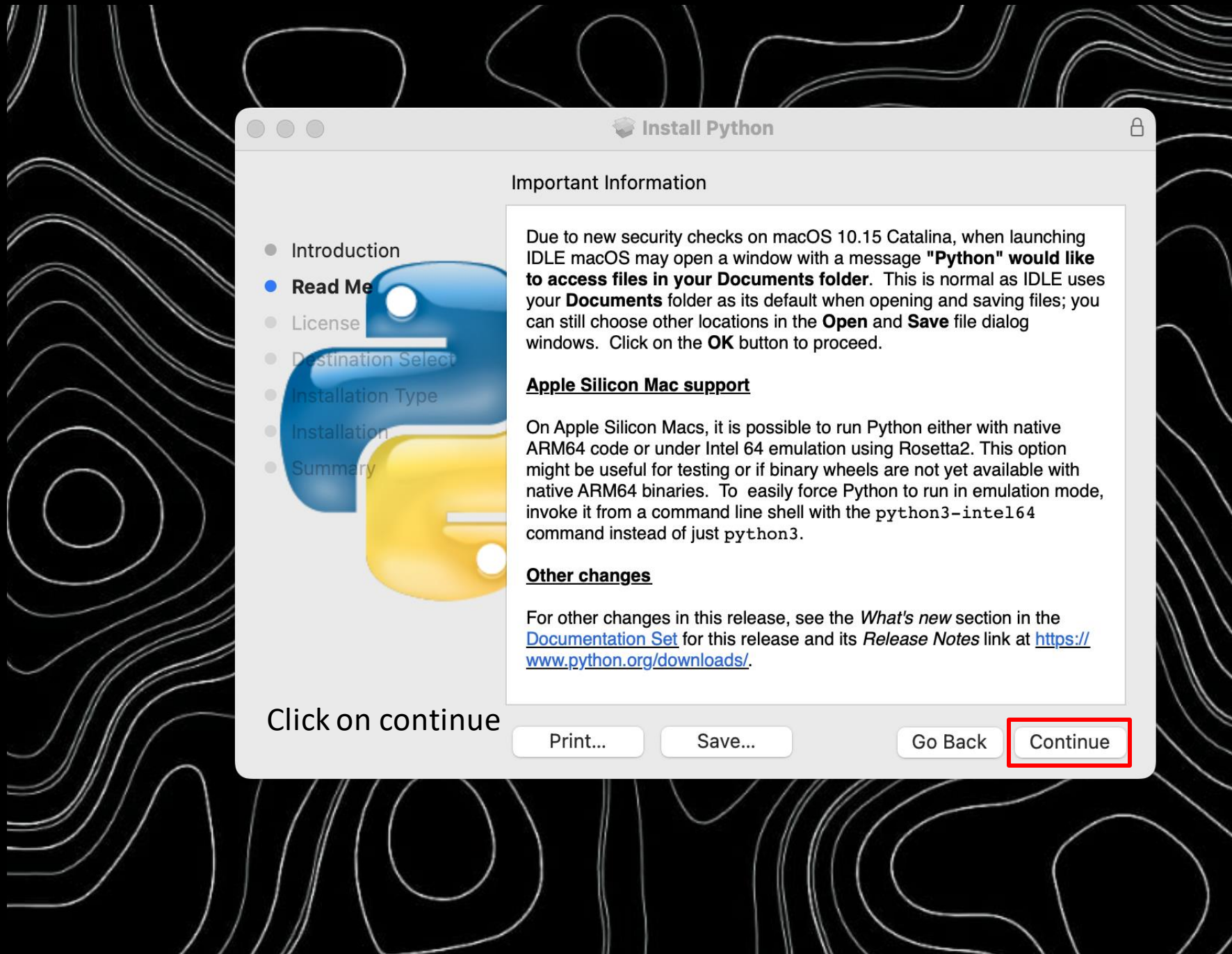
Want to help test development versions of Python 3.13? Prereleases, Docker images

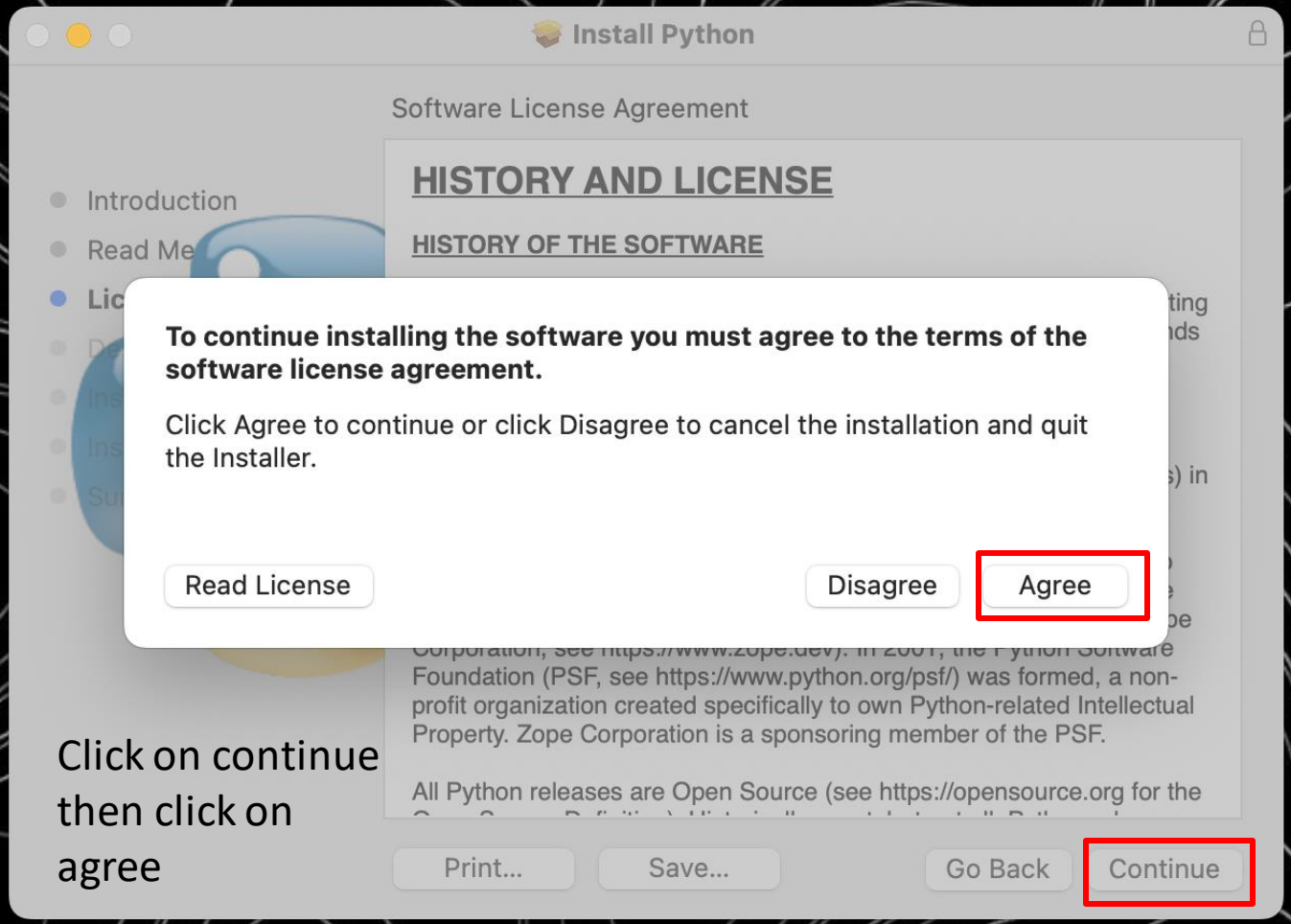
Active Python Releases

For more information visit the Python Developer's Guide.

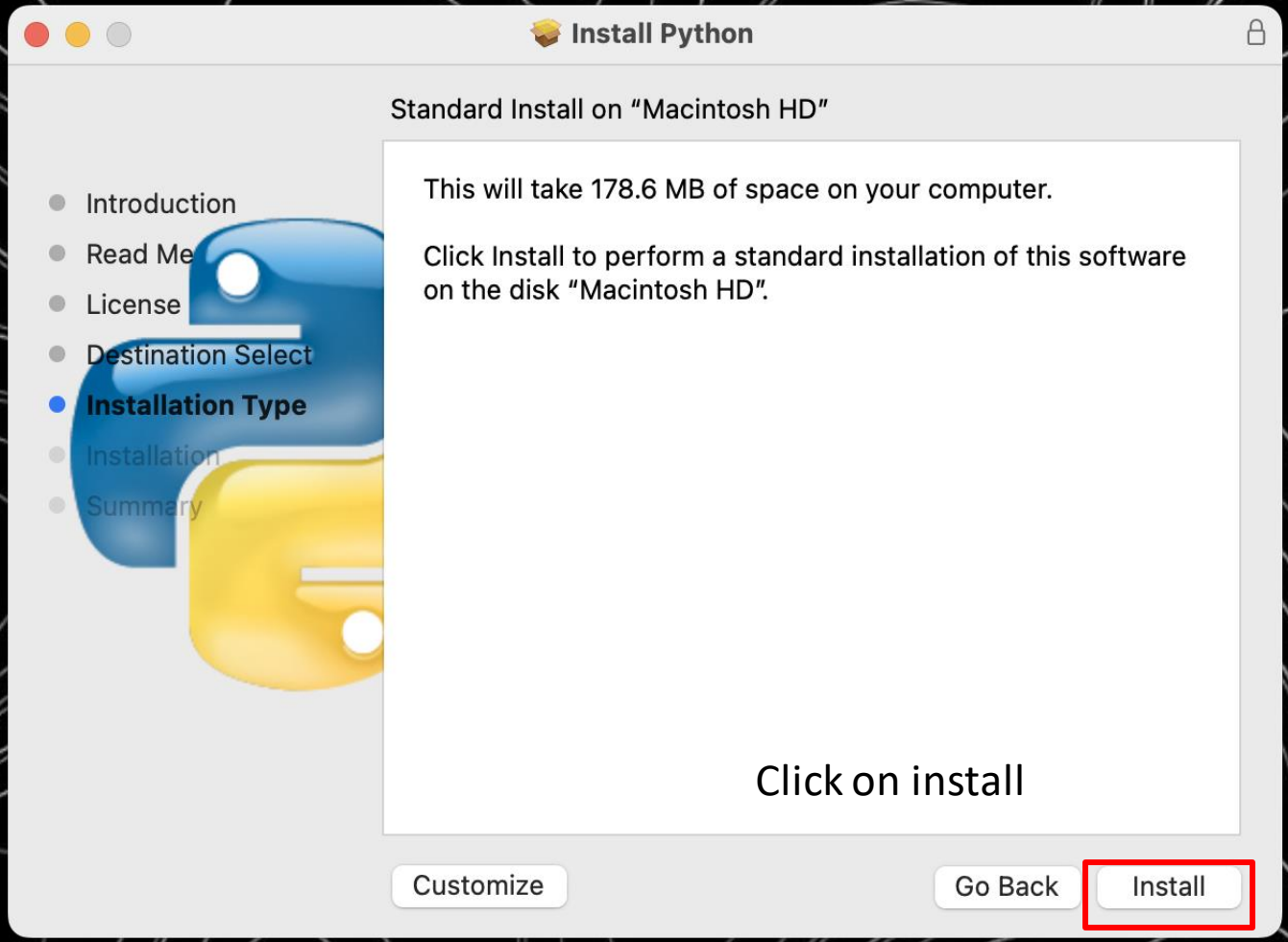
Python version	Maintenance status	First released	End of support	Release schedule
3.13	prerelease	2024-10-01 (planned)	2029-10	PEP 719
3.12	bugfix	2023-10-02	2028-10	PEP 693
3.11	bugfix	2022-10-24	2027-10	PEP 664
3.10	security	2021-10-04	2026-10	PEP 619


Install Python
Welcome to the Python Installer
Introduction
Read Me
License
Destination Select
Installation Type
Installation
Summary
This package will install Python 3.12.3 for macOS 10.9 or later.
Python for macOS consists of the Python programming language interpreter and its batteries-included standard library to allow easy access to macOS features. It also includes the Python integrated development environment, IDLE. You can also use the included pip to download and install third-party packages from the Python Package Index.
At the end of this install, click on Install Certificates to install a set of current SSL root certificates.
Click on continue
Go Back
Continue


Install Python
Important Information
Introduction
Read Me
License
Destination Select
Installation Type
Installation
Summary
Due to new security checks on macOS 10.15 Catalina, when launching IDLE macOS may open a window with a message "Python" would like to access files in your Documents folder. This is normal as IDLE uses your Documents folder as its default when opening and saving files; you can still choose other locations in the Open and Save file dialog windows. Click on the OK button to proceed.
Apple Silicon Mac support
On Apple Silicon Macs, it is possible to run Python either with native ARM64 code or under Intel 64 emulation using Rosetta2. This option might be useful for testing or if binary wheels are not yet available with native ARM64 binaries. To easily force Python to run in emulation mode, invoke it from a command line shell with the python3-intel64 command instead of just python3.
Other changes
For other changes in this release, see the What's new section in the Documentation Set for this release and its Release Notes link at https://www.python.org/downloads/.
Click on continue
Print...
Save...
Go Back
Continue



Click on continue then click on agree


Install Python
Standard Install on “Macintosh HD”
Introduction
Read Me
License
Destination Select
Installation Type
Installation
Summary
This will take 178.6 MB of space on your computer.
Click Install to perform a standard installation of this software on the disk “Macintosh HD”.
Click on install
Customize
Go Back
Install


myapp
EXTENSIONS: MARKETPLA...
python
Python
53ms
Python language support wi...
Microsoft
Python
226K
5
Extensions for Python
shiro
Install
Python Debugger
Python Debugger extension...
Microsoft
Python Indent
9.5M
4.5
Correct Python indentation
Kevin Rose
Install
Python Exte...
8.5M
4.5
Popular Visual Studio Code ...
Don Jayamanne
Install
Python for VS...
5.5M
2
Python language extension ...
Thomas Haako...
Install
Python Envir...
7.9M
3.5
View and manage Python e...
Don Jayamanne
Install
Extension: Python
Python
v2024.6.0
Microsoft
microsoft.com
124,395,658
Python language support with extension access points for I...
Disable
Uninstall
Switch to Pre-Release Version
This extension is enabled globally.
DETAILS
FEATURES
CHANGELOG
EXTENSION PACK
Python extension for Visual Studio Code
A Visual Studio Code extension with rich support for the Python language (for all actively supported Python versions), providing access points for extensions to seamlessly integrate and offer support for IntelliSense (Pylance), debugging (Python Debugger), formatting, linting, code navigation, refactoring, variable explorer, test explorer, and more!
Support for vscode.dev
Categories
Programming Languages
Debuggers
Other
Data Science
Machine Learning
Resources
Marketplace
Issues
Repository
License
0 0
0
Go Live

myapp

EXTENSIONS: MARKETPLA...

ju

Jupyter
Jupyter notebook support, i...
Microsoft

Jupyter Slide Show
Jupyter Slide Show support ...
Microsoft

Jupyter Keymap
Jupyter keymaps for notebo...
Microsoft

Jupyter Notebook Render...
Renderers for Jupyter Note...
Microsoft

Jupyter Cell Tags
Jupyter Cell Tags support f...
Microsoft

Julia 704K ★ 4.5
Julia Language Support
julialang Install

jumpy 532K ★ 4.5
Jumpy provides fast cursor ...
wmaurer Install

Extension: Jupyter

Jupyter v2024.4.0

Microsoft microsoft.com | 78,716,510 | ★★★½

Jupyter notebook support, interactive programming and co...

Disable | Uninstall | Switch to Pre-Release Version

This extension is enabled globally.

DETAILS FEATURES CHANGELOG

Extension Pack (4)

Jupyter Keymap
Jupyter keymaps for notebooks
Microsoft

Jupyter Notebook Renderers
Renderers for Jupyter Notebooks (with pl...
Microsoft

Jupyter Extension for Visual Studio Code

Categories

Extension Packs
Data Science
Machine Learning
Notebooks
Visualization

Resources

Marketplace
Issues
Repository
License

0 0 0 Go Live

TOPIC 3

BASIC BUILDING BLOCKS

What are Comments in Python?

Programmer-coherent statements

Describe what a block of code means

Static.Int Educare

How to write Comments in Python?

Comments in Python start with a # character

```
#Comments in Python start like this
print("Comments in Python start with a #")
```

OUTPUT -

```
Comments in Python start with a #
```

Static.Int Educare

How does Python interpret Comments?

Static.Int Educare

Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase **(a to z)** or uppercase **(A to Z)** or digits **(0 to 9)** or an underscore _. Names
   like myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is a valid name.
3. Keywords cannot be used as identifiers.

Static.Int Educare

Python Keywords

Keywords are the reserved words in Python.
We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
In Python, keywords are case sensitive.
There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.
All the keywords except True, False and None are in lowercase and they must be written as they are. The list of all the keywords is given below.

PYTHON VARIABLES

Creating Variables

Variables are containers for storing data values.

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
```

PYTHON INDENTATION

Most of the programming languages like C, C++, and Java use braces { } to define a block of code. Python, however, uses indentation.

A code block (body of a function, loop, etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally, four whitespaces are used for indentation and are preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

TOPIC 4

Operators

Operators in Python

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

Arithmetic Operators

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

+	Add two operands or unary plus	>> 2 + 3 5 >> +2
-	Subtract two operands or unary subtract	>> 3 – 1 2 >> -2
*	Multiply two operands	>> 2 * 3 6
/	Divide left operand with the right and result is in float	>> 6 / 3 2.0

Assignment operators

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5

Comparison Operators

Static.Int Educare

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

>	True if left operand is greater than the right	>> 2 > 3 False
<	True if left operand is less than the right	>> 2 < 3 True
==	True if left operand is equal to right	>>2 == 2 True
!=	True if left operand is not equal to the right	>> x >> = 2 >>print(x)

Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example :

```
x = True
y = False

print('x and y is', x and y)

print('x or y is', x or y)

print('not x is', not x)
```

Output

```
x and y is False
x or y is True
not x is False
```

Bitwise Operators

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

a | b

Perform OR operation on each bit of the no

111 → 7
101 → 5
111 → 7

a & b

Perform AND operation on each bit of the number

111 → 7
101 → 5
101 → 5

a ^ b

Perform XOR operation on each bit of the number

111 → 7
101 → 5
010 → 2

Static.Int Educare

Bitwise Operators

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

a >> b

Shift a right by b bits

```
3   >>   2 = 0
0011       0000
```

a << b

Shift a left by b bits

```
3   <<   2 = 12
0011       1100
```

Static.Int Educare

Identity Operators

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

is

True if the operands are identical (refer to the same object)

```
>> x = 5
>> x is 5
True
```

is not

True if the operands are not identical (do not refer to the same object)

```
>> x = 5
>> x is not 5
False
```

Static.Int Educare

Membership Operators

Static.Int Educare

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

Membership Operators are the operators, which are used to check whether a value/variable exists in the sequence like string, list, tuples, sets, dictionary or not.
These operator returns either True or False, if a value/variable found in the list, its returns True otherwise it returns False.

TOPIC 5

Control Flow

Control Flow Statements

A program’s control flow is the order in which the program’s code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls. This section covers the if statement and for and while loops; functions are covered later in this chapter. Raising and handling exceptions also affects control flow.

What is if...else statement in Python?

Decision making is required when we want to execute a code only if a certain condition is satisfied.

The if…elif…else statement is used in Python for decision making.

Python if Statement Syntax

```
if test_expression:
        statement(s)
```

Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True.

If the text expression is False, the statement(s) is not executed.

Python if...else Statement

Syntax of if...else

```
if test_expression:
    Body of if
else:
    Body of else
```

The if..else statement evaluates test expression and will execute the body of if only when the test condition is True.

If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

Python if...elif...else Statement

Syntax of if...elif...else

```
if test_expression:
    Body of if
elif test_expression:
    Body of elif
else:
    Body of else
```

The elif is short for else if. It allows us to check for multiple expressions.

If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, the body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

Python Nested if statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

Python Nested if Example

```
num = 5
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Output:

Positive number

Static.Int Educare

What is for loop in Python?

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

```
for var_name in iterable_object:
    Body of for
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

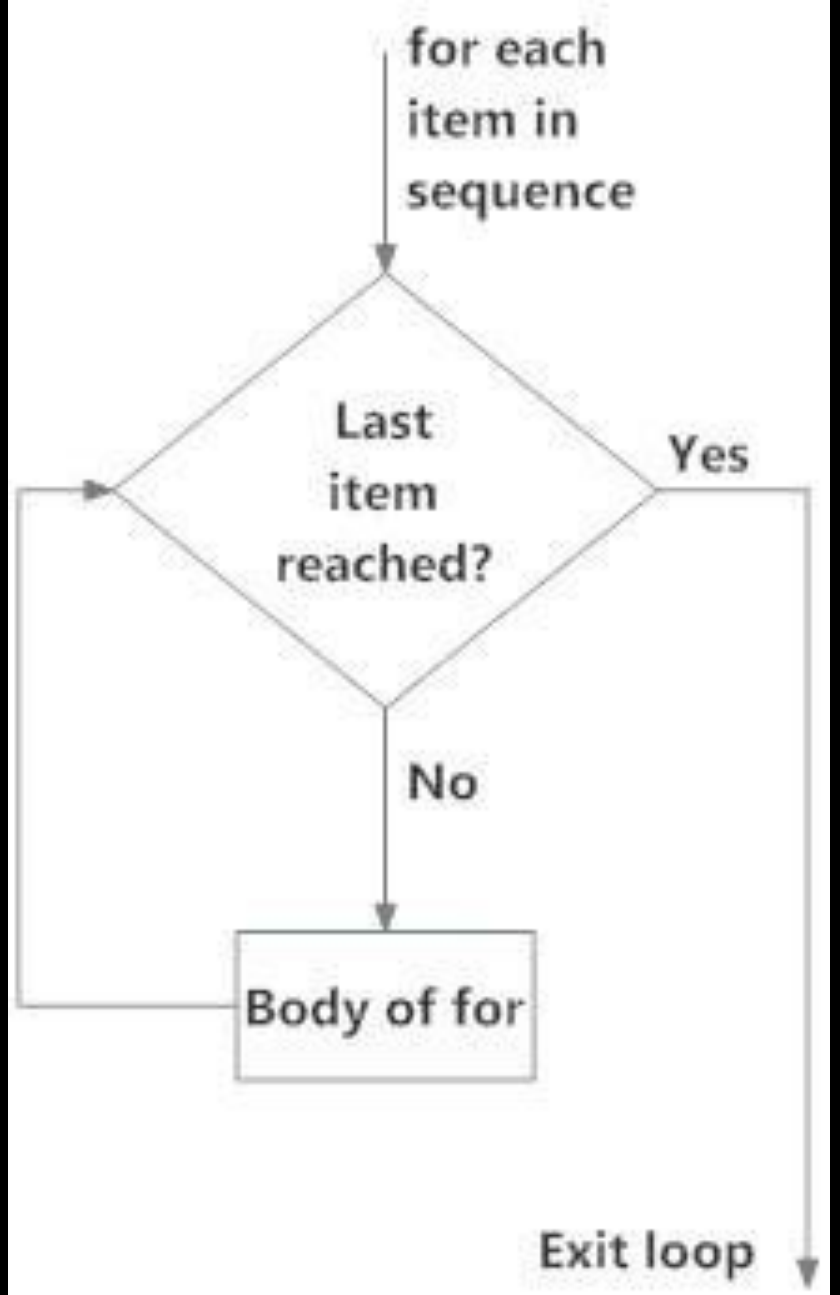


Fig: operation of for loop

What is while loop in Python?

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax of while Loop in Python

```
while test_expression:
    Body of while
```

In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation.

Static.Int Educare

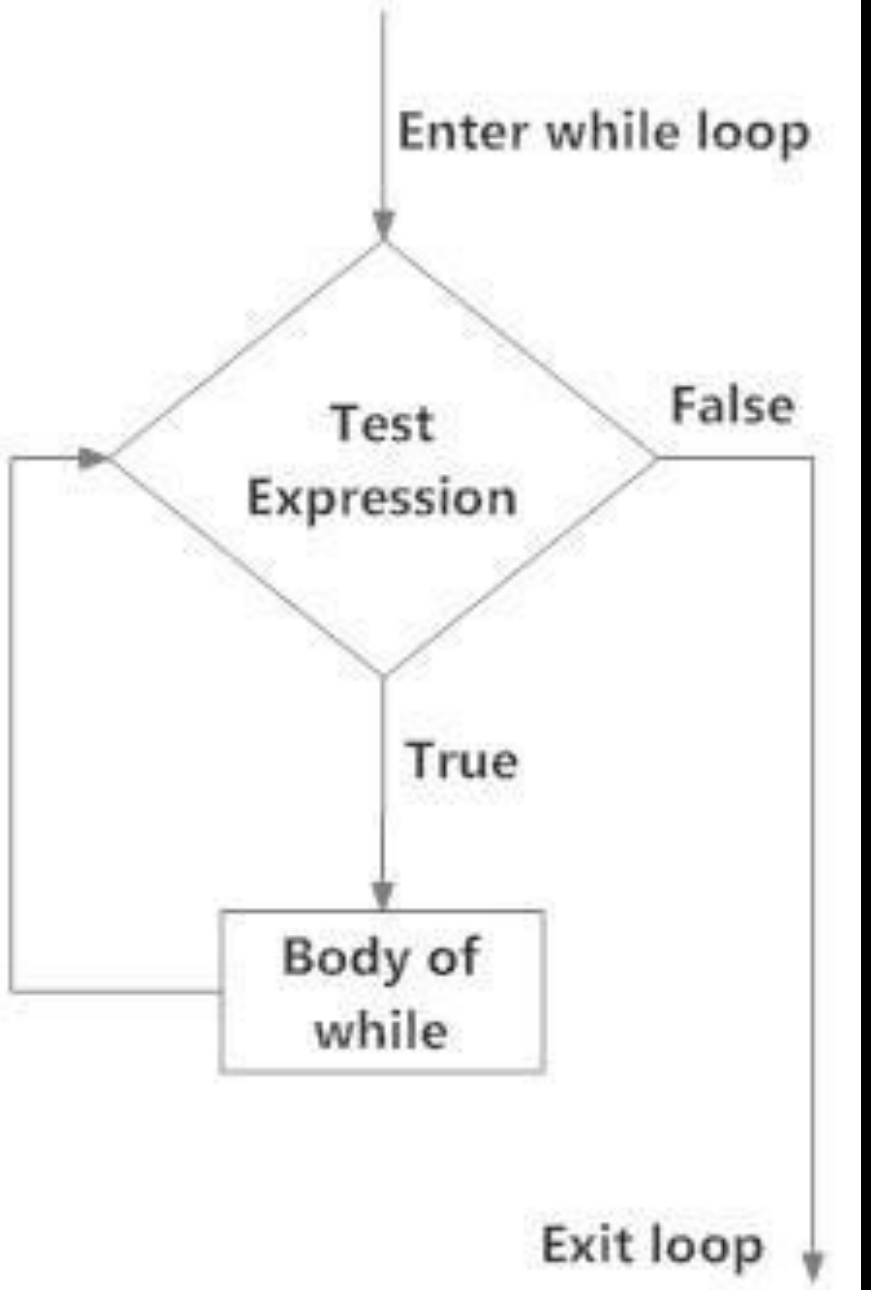


Fig: operation of while loop

Static.Int Educare

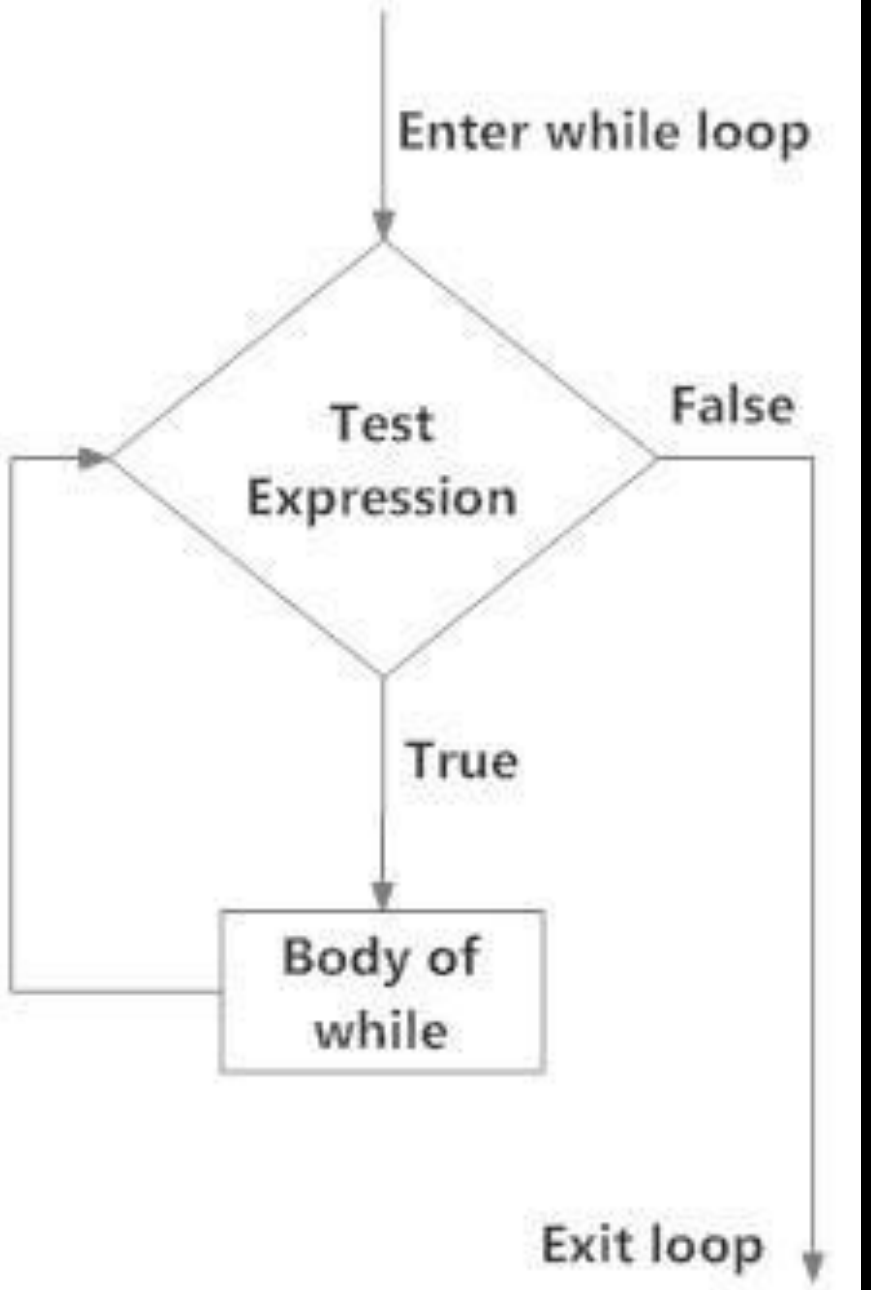


Fig: operation of while loop

TOPIC 6

Data Structures

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [ ], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = [ ]

# list of integers
my_list = [1, 2, 3]

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

A list can also have another list as an item. This is called a nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

How to access elements from a list?

There are various ways in which we can access the elements of a list.

List Index

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.

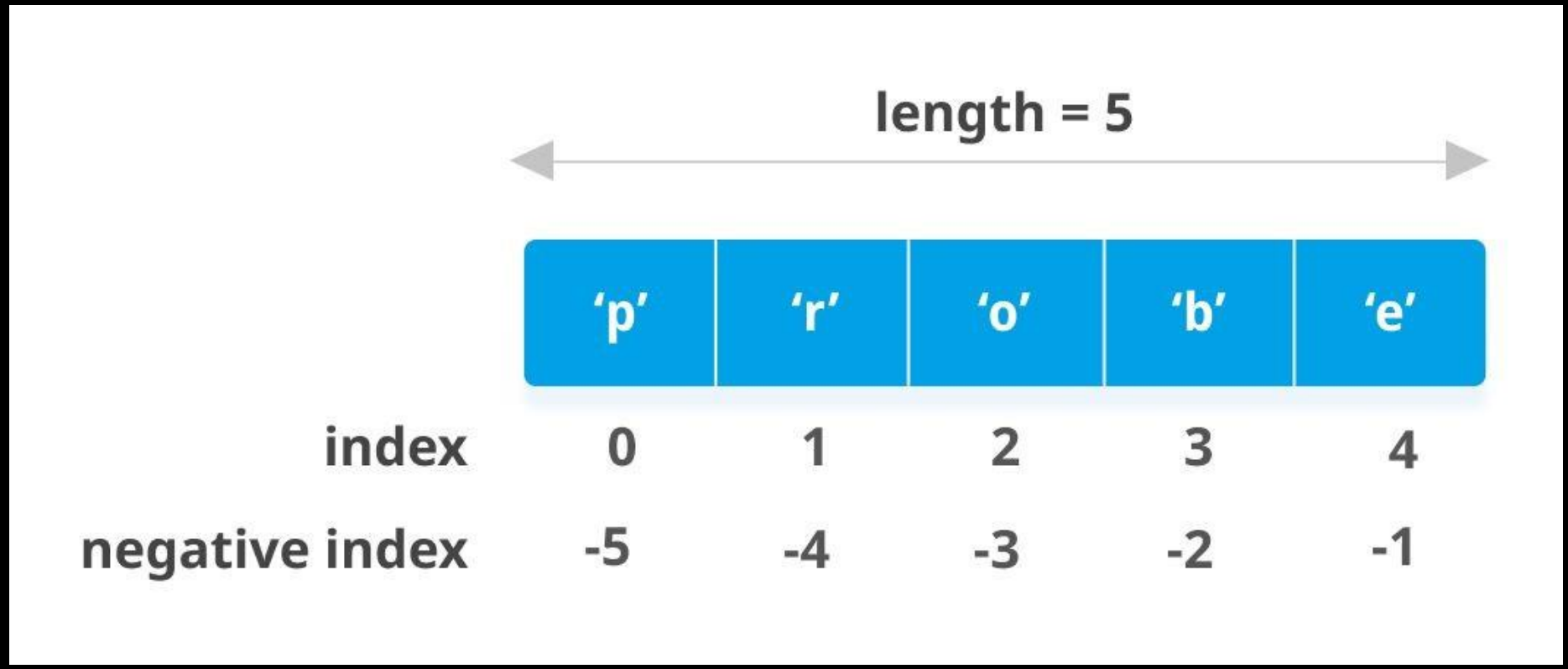
Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

```
# List indexing
my_list = ['p', 'r', 'o', 'b', 'e']
# Output: p
print(my_list[0])
# Output: o
print(my_list[2])
# Output: e
print(my_list[4])
# Nested List
n_list = ["Happy", [2, 0, 1, 5]]
# Nested indexing
print(n_list[0][1])
print(n_list[1][3])
# Error! Only integer can be used for indexing
print(my_list[4.0])
```

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing in lists
my_list = ['p','r','o','b','e']
print(my_list[-1])
print(my_list[-5])
```



Static.Int Educare

Python List Methods

append() - Add an element to the end of the list

extend() - Add all elements of a list to the another list

insert() - Insert an item at the defined index

remove() - Removes an item from the list

pop() - Removes and returns an element at the given index

clear() - Removes all items from the list

index() - Returns the index of the first matched item

count() - Returns the count of the number of items passed as an argument

sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list

copy() - Returns a shallow copy of the list

Static.Int Educare

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

 # tuple with mixed  datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

SETS

- A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
- However, a set itself is mutable. We can add or remove items from it.
- Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Python Sets
A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# Different types of sets in Python
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has key/valuepairs.

Dictionaries are optimized to retrieve values when the key is

known. Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma ,.

An item has a key and a corresponding value that is expressed as a pair (key: value).

While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty
dictionary
my_dict = { }
# dictionary with integer
keys my_dict =
{1:'apple',2:'ball'}
# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}
# using dict()
my_dict = dict({1:'apple', 2:'ball'})
```

Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (key: value) followed by a for statement inside curly braces {}.

Here is an example to make a dictionary with each item being a pair of a number and

its square. # Dictionary Comprehension

```
sqr = { x : x*x for x in range(6) }

print(sqr)
```

Output:

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

TOPIC 7 | Functions

- A function is a block of organized, reusable code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Syntax of Function :

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

Example of a function:


```
def greet(name):
        """ This function greets to the person passed in as a parameter """
                print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
Hello, Paul. Good morning!
```

: OR in our very program as :

```
def greet(name):
    """ This function greets to the person passed in as a parameter """
    print("Hello, " + name + ". Good morning!")

greet('Paul')
```

Static.Int Educare

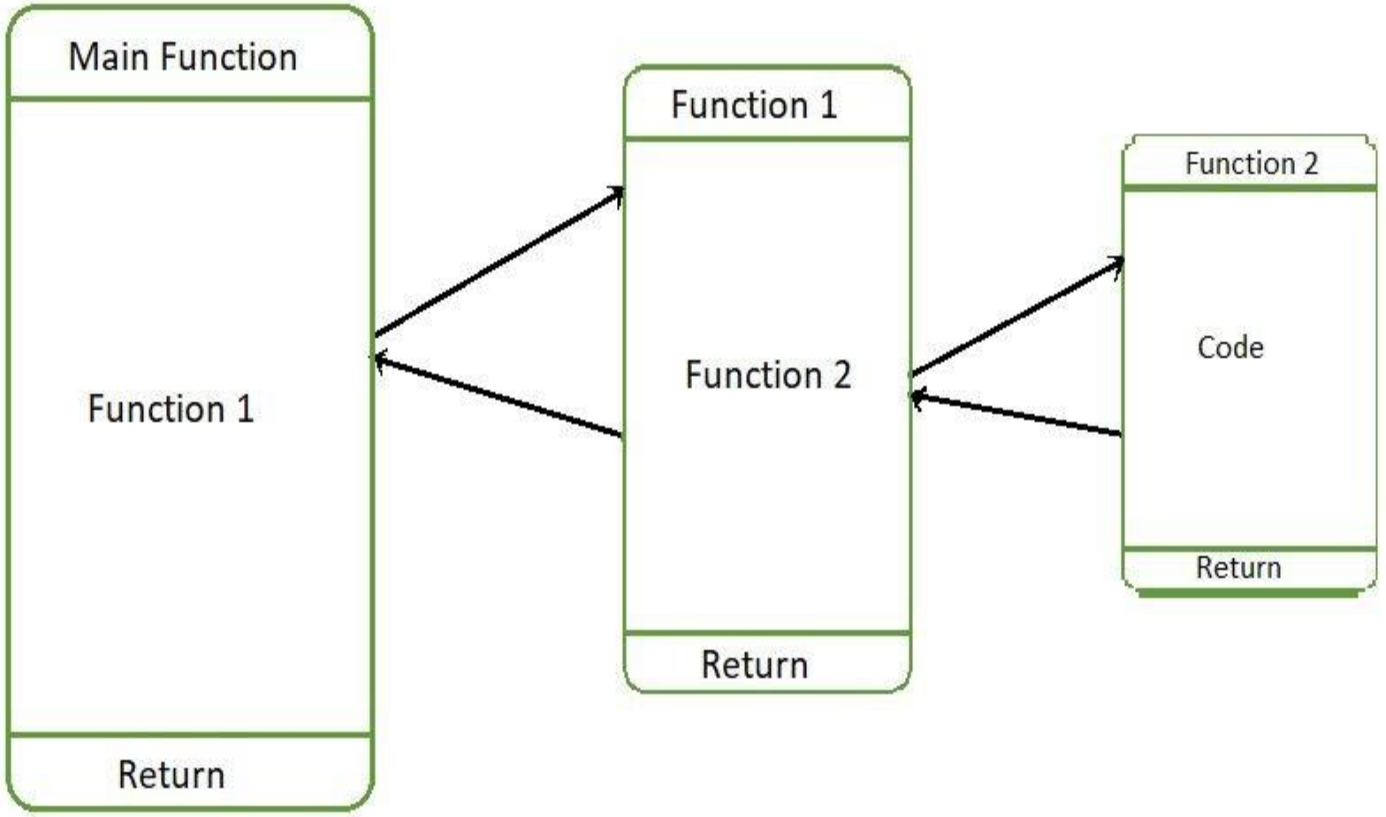
How Function works in Python?

```
def functionName():
    ... .. ...
    ... .. ...

... .. ...
... .. ...

functionName();

... .. ...
... .. ...
```



The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

return [expression_list]

Example of return

```
def absolute_value(num):
        """This function returns the absolute value of the entered number"""“
        if num >= 0:
                return num
        else:
                return -num

print(absolute_value(2))
print(absolute_value(-4))
```

Output

```
2
4
```

Scope and Lifetime of variables

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function.
- Here is an example to illustrate the scope of a variable inside a function.

```
x = 20
def  my_func():
      x = 10
      print("Value inside function : ", x)

my_func()
print("Value outside function : ", x)
```

Output

Value inside function: 10

Python Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
def greet(name, msg="Good morning!"):
    """ This function
    greets to the person
    with the provided message.

    If the message is not provided,
    it defaults to "Good morning!" """

    print("Hello", name + ', ' + msg)

greet("Kate")
greet("Bruce", "How do you do?")
```

Output

Hello Kate, Good morning!
Hello Bruce, How do you do?

```
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")


#1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

Static.Int Educare

Arbitrary Arguments, *args :

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

Example
If the number of arguments is unknown, add a * before the parameter name:

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

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

Output :
The youngest child is Linus

Arbitrary Keyword Arguments, **kwargs

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

This way the function will receive a dictionary of arguments, and can access the items accordingly:

Example
If the number of keyword arguments is unknown, add a double ** before the parameter name:

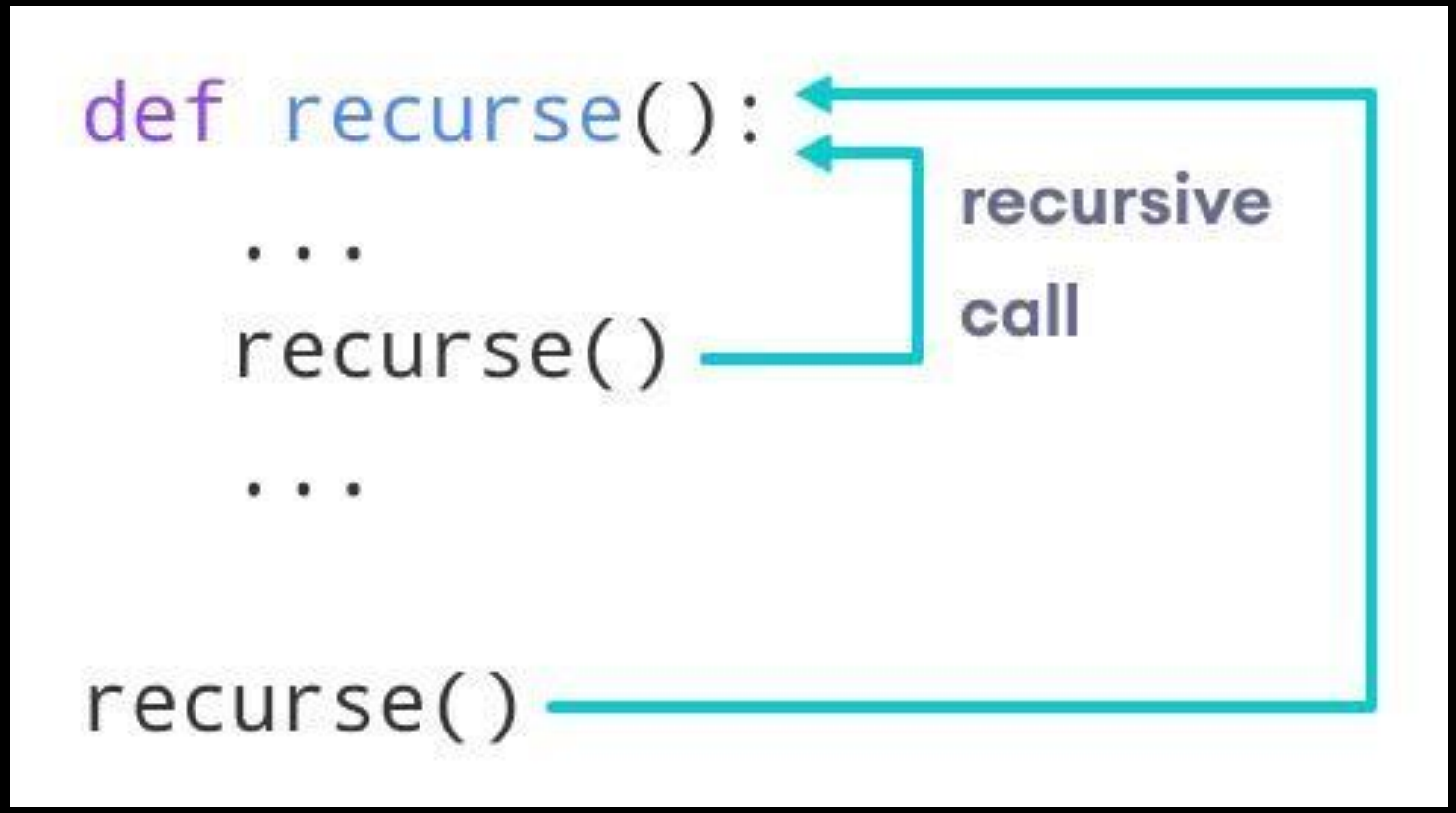
```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Aryan"    , lname = “Kanse”)
```

Python Recursive Function

- In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

- The following image shows the working of a recursive function called recurse.



Static.Int Educare

Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example of a recursive function

```
def factorial(x):
    """This is a recursive function to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

Output :

```
The factorial of 3 is 6
```

In the above example, factorial() is a recursive function as it calls itself.

Python Anonymous/Lambda Function

- In Python, an anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called lambda functions.

Syntax of Lambda Function in python

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions
double = lambda x: x * 2
print(double(5))
```

Output

10

It is nearly the same as:

```
def double(x):
        return x * 2
```

TOPIC 8

File Handling

Opening Files in Python

Python has a built-in open() function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python38/README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read r, write w or append a to the file. We can also specify if we want to open the file in text mode or binary mode.

```
>>> f = open("test.txt", ‘r’)
```

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

There are four different methods (modes) for opening a file:

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist
- "a" - Append - Opens a file for appending, creates the file if it does not exist
- "w" - Write - Opens a file for writing, creates the file if it does not exist
- "x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

- "t" - Text - Default value. Text mode
- "b" - Binary - Binary mode (e.g. images)

TOPIC 9

Exception Handling

Python Syntax Errors

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

Let's look at one example:

```
>>> if a < 3
  File "<interactive input>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

As shown in the example, an arrow indicates where the parser ran into the syntax error.

We can notice here that a colon : is missing in the if statement.

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.

For instance, they occur when we try to open a file(for reading) that does not exist (FileNotFoundError), try to divide a number by zero (ZeroDivisionError), or try to import a module that does not exist (ImportError).

Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Let's look at how Python treats these errors:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ZeroDivisionError: division by zero

>>> open("imaginary.txt")
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'
```

Sr.No.	Exception Name & Description
ArithmeticError	Base class for all errors that occur for numeric calculation.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
IndentationError	Raised when indentation is not specified properly.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.

Handling an exception

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of try....except...finally blocks −

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
finally:
   print "Written content in the file successfully"
   fh.close()
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to RuntimeError. Here, a class is created that is subclassed from RuntimeError. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

```
class Networkerror(RuntimeError):
   def__init__(self, arg):
      self.args = arg
```

So once you defined above class, you can raise the exception as follows −

```
try:
   raise Networkerror("Bad hostname")
except Networkerror as e:
   print e.args
```

TOPIC 10

Object Oriented Programming with Python

Introduction to OOPs in Python

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

1. attributes
2. behavior

Let's take an example:

Parrot is an object,
name, age, color are attributes
singing, dancing are behavior
The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Inheritance A process of using details from a new class without modifying existing class.
Encapsulation Hiding the private details of a class from other objects.
Polymorphism A concept of using common operation in different ways for different data input.

Class

A class is a blueprint for the object.

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, obj is object of class Parrot.

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

```
class Parrot:

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

# instantiate the object
blu = Parrot( )
print(blu.sing(""Happy""))
```

Output:
Blu sings 'Happy'

Inheritance

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

```
class Bird:

    def__init__(self):
        print("Bird is ready")

# child class
class Penguin(Bird):

    def__init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

peggy = Penguin()
```

Output:

Bird is ready

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ _ “ or double “__“.

Example 4: Data Encapsulation in Python

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()
c.__maxprice = 1000
c.sell()
c.setMaxPrice(1000)
c.sell()
```

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).
Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

```
class Parrot:
    def fly(self):
        print("Parrot can fly")

class Penguin:
    def fly(self):
        print("Penguin can't fly")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

Key Points to Remember:

- The programming gets easy and efficient.
- The class is sharable, so codes can be reused.
- The productivity of programmers increases
- Data is safe and secure with data abstraction.