History of Python Programming Language

Introduction

Python is a high-level, interpreted, general-purpose programming language known for its simplicity and readability.

History

- 1989: Python was created by Guido van Rossum at Centrum Wiskunde & Informatica (CWI), Netherlands.
- February 1991: First version (Python 0.9.0) was released.
- Python was named after the British comedy show "Monty Python's Flying Circus", not after the snake.
- 2000: Python 2.0 was released with features such as garbage collection and list comprehensions.
- 2008: Python 3.0 was released with significant improvements and incompatibility with Python 2.
- January 1, 2020: Official support for Python 2 ended.
- Today, Python is maintained by the Python Software Foundation (PSF) and is one of the most popular programming languages in the world.

Key Milestones

Year	Event
1989	Development of Python started
1991	Python 0.9.0 released
1994	Python 1.0 released
2000	Python 2.0 released
2008	Python 3.0 released
2020	Python 2 support ended

Features of Python

- Simple and easy to learn
- Open-source
- Platform independent
- Interpreted language

- Object-oriented
- Large standard library
- Supports GUI applications, web development, and AI

Thrust Areas of Python

Thrust Areas are the major application domains where Python is widely used.

1. Web Development

Python is used to develop dynamic websites and web applications.

Frameworks

- Django
- Flask
- FastAPI

Examples

- E-commerce websites
- Content Management Systems
- Web portals

---

2. Data Science and Analytics

Python provides powerful libraries for data analysis and visualization.

Libraries

- NumPy
- Pandas
- Matplotlib
- Seaborn

Applications

- Data analysis
- Business intelligence
- Statistical computing

---

3. Artificial Intelligence (AI) and Machine Learning (ML)

Python is the most popular language for AI and ML development.

Libraries

- Scikit-learn
- TensorFlow
- PyTorch
- Keras

Applications

- Image recognition
- Speech recognition
- Recommendation systems
- Predictive analytics

---

4. Deep Learning

Used for building neural networks and intelligent systems.

Frameworks

- TensorFlow
- PyTorch
- Keras

Applications

- Medical image analysis
- Self-driving cars
- Natural Language Processing

---

5. Scientific and Numerical Computing

Python is widely used in research and scientific applications.

Libraries

- SciPy
- NumPy
- SymPy

Applications

- Mathematical modeling

- Simulations
- Engineering calculations

---

6. Automation and Scripting

Python automates repetitive tasks efficiently.

Applications

- File handling
- Report generation
- System administration
- Task scheduling

---

7. Desktop GUI Applications

Python can be used to create graphical user interfaces.

Frameworks

- Tkinter
- PyQt
- Kivy

Examples

- Calculator applications
- Management systems
- Educational software

---

8. Database Applications

Python interacts with various databases.

Databases

- MySQL
- PostgreSQL
- SQLite
- MongoDB

Applications

- Student management systems

- Banking systems
- Inventory management

---

9. Internet of Things (IoT)

Python is used in embedded systems and IoT devices.

Platforms

- Raspberry Pi
- MicroPython

Applications

- Smart homes
- Smart agriculture
- Sensor monitoring

---

10. Cybersecurity and Ethical Hacking

Python helps in security testing and network analysis.

Tools/Libraries

- Scapy
- Requests
- Nmap integrations

Applications

- Network monitoring
- Vulnerability assessment
- Security automation

---

11. Game Development

Python is used for developing simple and educational games.

Libraries

- **Pygame**

Applications

- 2D games
- Learning games

- Simulations

---

12. Cloud Computing and DevOps

Python is extensively used in cloud automation.

Applications

- AWS automation
- Azure scripting
- Infrastructure management
- CI/CD pipelines

Short Note:

History of Python: Python was developed by Guido van Rossum in 1989 and first released in 1991. It is an interpreted, high-level, open-source programming language known for simplicity and readability. Python 3.0 was released in 2008 and is the current major version.

Thrust Areas of Python: Web Development, Data Science, Artificial Intelligence, Machine Learning, Deep Learning, Scientific Computing, Automation, GUI Applications, Database Management, IoT, Cybersecurity, Game Development, and Cloud Computing.

Introduction To Python Programming Language

Python language, first released in 1991 by Guido van Rossum, is a high-level, general-purpose programming language that evolved from the ABC language. It is renowned for its simplicity and readability, making it an ideal choice for both beginners and experienced developers. Python's design emphasizes code readability and simplicity, which helps in writing clean and maintainable code.

- Python supports multiple programming paradigms, including Object-Oriented Programming (OOP), Functional Programming, and Structured Programming.
- Its extensive standard library provides a wide range of modules and functions, which facilitates various programming tasks and reduces the need for additional extensive libraries.
- Python includes automatic memory management and garbage collection, which helps manage memory usage efficiently. It is widely used in various fields such as software development, data science, web development, automation, and education due to its ease of use and versatility.

Features Of Python Programming Language

- Python is a very simple language editor that is easy to use, read, and open.
- It is a translated language and not a compound language. It has an English-language syntax.

- This language also supports the object-oriented language model and has libraries with functions and adequate performance. Hence it is easy to use data structures with built-in insert, and append functions.
- It is an easily extensible and embeddable stand-alone platform. It contains an extensive standard library with numerous modules and packages. Python has one of the largest tech communities for developers on platforms like StackOverflow and Meetup.
- Python is an advanced language in terms of performance and ease of use because of its vast and highly adaptable rich library.

Advantages And Disadvantages Of Python

Advantages Of Python	Disadvantages Of Python
Python is portable (a computer programming language capable of developing more than one computer program) and interactive, making it versatile across different platforms and suitable for various applications.	Python has speed limits (as translated) and is generally slower compared to integrated languages such as C and C ++.
Python allows for quick development and prototyping with its concise syntax and extensive standard library.	Multithreading creates problems in Python due to Global Interpreter Lock (GIL). GIL is nothing but a mutex that allows only one cable to run at a time because CPU-bound systems with multiple cables are not as fast as those with single wires.
Python offers robust libraries and frameworks, such as NumPy for numerical computations and Pandas for data analysis, which are valuable for data science and machine learning.	Python is not natural in the mobile environment and therefore, can be seen as the weak language of a portable computer. Android and iOS do not support Python as an official programming language.
Python has a large and active community that provides extensive resources, including libraries, frameworks, tutorials, and forums. This strong community support helps developers find solutions to problems, learn best practices, and stay updated with the latest advancements in the language.	Python also has its limitations with database access. The Python website access layer is in its infancy and has not improved compared to popular technologies such as JDBC (Java DataBase Connectivity) and ODBC (Open Database Connectivity).

Python Programming

1. Python Programming Language

Python is a **high-level, interpreted, object-oriented, and general-purpose programming language**. It is easy to learn and widely used in web development, data science, artificial intelligence, automation, and software development.

Features of Python

- Simple and easy syntax
- Interpreted language
- Platform independent
- Open source
- Supports Object-Oriented Programming (OOP)
- Large standard library

---

2. Identifiers

Identifiers are names used to identify variables, functions, classes, modules, etc.

Rules for Identifiers

- Must begin with a letter (A-Z, a-z) or underscore (_)
- Can contain letters, digits, and underscores
- Cannot start with a digit
- Cannot use keywords as identifiers
- Case-sensitive

Valid Identifiers

name

_age

student1

total_marks

Invalid Identifiers

1name

class

first-name

---

3. Keywords

Keywords are reserved words with predefined meanings.

Python Keywords

False await else import pass

None break except in raise

True class finally is return

and continue for lambda try

as def from nonlocal while

assert del global not with

async elif if or yield

Example

```
if x > 10:
  print("Greater")
```

---

4. Statements and Expressions

Expression

An expression is a combination of values, variables, and operators that produces a result.

Example

```
a = 10
b = 20
c = a + b
```

Here:

a + b

is an expression.

Statement

A statement is an instruction executed by Python.

Example

```
a = 10
print(a)
```

---

5. Variables

Variables are used to store data values.

Syntax

```
variable_name = value
```

Example

```
x = 10
name = "Ravi"
pi = 3.14
```

Multiple Assignment

```
a, b, c = 10, 20, 30
```

Assign Same Value

```
x = y = z = 100
```

---

6. Operators

Operators perform operations on operands.

6.1 Arithmetic Operators

Operator	Description	Example
+	Addition	a+b
-	Subtraction	a-b
*	Multiplication	a*b
/	Division	a/b
//	Floor Division	a//b
%	Modulus	a%b
**	Exponentiation	a**b

Example

```
a = 10
b = 3

print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(a//b)
print(a%b)
```

```
print(a**b)
```

6.2 Comparison Operators

Operator	Meaning
==	Equal
!=	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal

Example

```
print(10 > 5)
print(10 == 5)
```

6.3 Logical Operators

Operator	Description
and	Logical AND
or	Logical OR
not	Logical NOT

Example

```
a = True
b = False

print(a and b)
print(a or b)
print(not a)
```

6.4 Assignment Operators

```
x = 10
x += 5
```

x -= 2

x *= 3

x /= 2

6.5 Membership Operators

Operator	Meaning
in	Present
not in	Not Present

Example

```
nums = [1,2,3,4]


print(2 in nums)
print(5 not in nums)
```

6.6 Identity Operators

Operator	Meaning
is	Same Object
is not	Different Object

Example

```
a = [1,2]
b = a


print(a is b)
```

---

7. Precedence and Associativity

Operator precedence determines which operation is performed first.

Precedence Order (High to Low)

Operator
()
**

Operator

+x, -x

*, /, //, %

+, -

==, !=, >, <

not

and

or

Example

result = 10 + 5 * 2

print(result)

Output:

20

Associativity

Most operators follow **Left to Right**.

Example:

20 / 5 * 2

Result:

8

Exponentiation follows **Right to Left**.

2 ** 3 ** 2

Result:

512

---

8. Data Types

Python supports various built-in data types.

Numeric Types

x = 10 # int

y = 3.14 # float

z = 2+3j # complex

String

name = "Python"

Boolean

flag = True

List

nums = [1,2,3]

Tuple

data = (1,2,3)

Set

s = {1,2,3}

Dictionary

student = {"name":"Ravi", "age":20}

---

9. Indentation

Python uses indentation to define code blocks.

Example

```
if 10 > 5:
    print("True")
```

Incorrect

```
if 10 > 5:
print("True")
```

This causes an **IndentationError**.

Standard Practice

Use **4 spaces** per indentation level.

---

10. Comments

Comments are used to explain code.

Single-Line Comment

```
# This is a comment
print("Hello")
```

Multi-Line Comment

"""

This is

a multi-line

comment

"""

---

11. Reading Input

Python provides the input() function.

Syntax

variable = input("Enter value: ")

Example

name = input("Enter your name: ")

print(name)

Integer Input

age = int(input("Enter age: "))

---

12. Print Output

Python uses print() to display output.

Example

print("Hello Python")

Multiple Values

name = "Ravi"

age = 20

print(name, age)

Separator

print("A", "B", "C", sep="-")

Output:

A-B-C

---

13. Type Conversions

Type conversion changes one data type into another.

Implicit Conversion

```
a = 10
b = 2.5

c = a + b
print(c)
```

Output:

12.5

Explicit Conversion

```
x = "100"
y = int(x)

print(y)
```

Common Functions

int()

float()

str()

bool()

list()

tuple()

set()

---

14. type() Function

The type() function returns the data type of an object.

Syntax

```
type(object)
```

Example

```
x = 10
print(type(x))
```

Output:

<class 'int'>

More Examples:

print(type(3.14))

print(type("Hello"))

print(type(True))

---

15. is Operator

The is operator checks whether two variables refer to the same object.

Example

a = [1,2,3]

b = a

print(a is b)

Output:

True

Difference Between == and is

a = [1,2,3]

b = [1,2,3]

print(a == b)

print(a is b)

Output:

True

False

- == → compares values
- is → compares memory locations

---

16. Dynamic and Strongly Typed Language

Dynamic Typing

Python is dynamically typed because variable types are determined during execution.

Example

```
x = 10
print(type(x))

x = "Python"
print(type(x))
```

Output:

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

The same variable can hold different data types.

Strong Typing

Python is strongly typed because it does not automatically convert incompatible data types.

Example

```
a = "10"
b = 5

# Error
# print(a + b)
```

Correct:

```
print(int(a) + b)
```

Output:

```
15
```

Summary

- **Identifiers**: Names given to variables, functions, classes.
- **Keywords**: Reserved words in Python.
- **Statements**: Executable instructions.
- **Expressions**: Produce values.
- **Variables**: Store data.
- **Operators**: Perform operations.
- **Precedence & Associativity**: Determine order of evaluation.

- **Data Types**: int, float, str, list, tuple, set, dict, etc.
- **Indentation**: Defines code blocks.
- **Comments**: Explain code.
- **input()**: Reads user input.
- **print()**: Displays output.
- **Type Conversion**: Converts one type to another.
- **type()**: Returns object type.
- **is**: Checks object identity.
- **Python** is both **Dynamically Typed** and **Strongly Typed**.

Python Notes: Control Flow Statements

Introduction

Control flow statements determine the order in which program statements are executed. They help in decision-making, looping, and exception handling.

---

1. if Statement

The if statement is used to execute a block of code only when a condition is true.

Syntax

```
if condition:
    statement(s)
```

Flow Diagram

Condition

|

True

|

Statements

Example

```
age = 18

if age >= 18:
    print("Eligible to vote")
```

Output

Eligible to vote

---

2. if-else Statement

The if-else statement executes one block if the condition is true and another block if it is false.

Syntax

```
if condition:
    statements1
else:
    statements2
```

Example

```
num = 7

if num % 2 == 0:
    print("Even Number")
else:
    print("Odd Number")
```

Output

```
Odd Number
```

---

3. if...elif...else Statement

Used when multiple conditions need to be checked.

Syntax

```
if condition1:
    statements1
elif condition2:
    statements2
elif condition3:
    statements3
else:
    default_statements
```

Example

```
marks = 85

if marks >= 90:
    print("Grade A+")
elif marks >= 75:
```

```
    print("Grade A")
elif marks >= 60:
    print("Grade B")
else:
    print("Grade C")
```

Output

Grade A

---

4. Nested if Statement

An if statement inside another if statement is called a nested if statement.

Syntax

```
if condition1:
    if condition2:
        statements
```

Example

```
age = 20
citizen = True

if age >= 18:
    if citizen:
        print("Eligible to Vote")
```

Output

Eligible to Vote

---

5. while Loop

A while loop repeatedly executes a block of code as long as the condition is true.

Syntax

```
while condition:
    statements
```

Example

```
i = 1
```

```
while i <= 5:
    print(i)
    i += 1
```

Output

```
1
2
3
4
5
```

Infinite Loop Example

```
while True:
    print("Hello")
```

---

6. for Loop

The for loop is used to iterate over a sequence such as a list, tuple, string, or range.

Syntax

```
for variable in sequence:
    statements
```

Example 1: Using range()

```
for i in range(1, 6):
    print(i)
```

Output

```
1
2
3
4
5
```

Example 2: Iterating through a String

```
for ch in "PYTHON":
    print(ch)
```

Output

P

Y

T

H

O

N

Example 3: Iterating through a List

```
colors = ["Red", "Green", "Blue"]


for color in colors:
   print(color)
```

---

7. break Statement

The break statement terminates the loop immediately.

Syntax

```
break
```

Example

```
for i in range(1, 10):
   if i == 5:
      break
   print(i)
```

Output

1

2

3

4

Use

- Exit a loop when a condition is met.
- Improve efficiency by avoiding unnecessary iterations.

---

8. continue Statement

The continue statement skips the current iteration and moves to the next iteration of the loop.

Syntax

continue

Example

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output

1

2

4

5

Use

- Skip specific values.
- Continue loop execution without terminating it.

Difference Between break and continue

break	**continue**
Terminates the loop completely	Skips current iteration
Control moves outside the loop	Control moves to next iteration
Used to stop looping	Used to ignore certain cases

Example

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

Output:

1

2

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output:

1

2

4

5

---

9. Exception Handling

An exception is an error that occurs during program execution.

Common Exceptions

Exception	Description
ZeroDivisionError	Division by zero
ValueError	Invalid value
TypeError	Invalid data type
IndexError	Invalid index
NameError	Undefined variable

---

10. try and except Statement

The try-except block is used to handle runtime errors and prevent program crashes.

Syntax

```
try:
    risky_code
except:
    error_handling_code
```

Example 1: ZeroDivisionError

```
try:
    num = 10 / 0
except ZeroDivisionError:
```

```
    print("Cannot divide by zero")
```

Output

```
Cannot divide by zero
```

Example 2: ValueError

```
try:
    age = int(input("Enter age: "))
except ValueError:
    print("Please enter a valid number")
```

Example 3: Multiple Exceptions

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print(a / b)

except ZeroDivisionError:
    print("Division by zero is not allowed")

except ValueError:
    print("Invalid input")
```

Example 4: Generic Exception

```
try:
    x = 10 / 0
except Exception as e:
    print("Error:", e)
```

Output

```
Error: division by zero
```

Advantages of Exception Handling

- Prevents abnormal program termination.
- Improves program reliability.
- Makes debugging easier.
- Allows graceful handling of errors.

Summary

Statement	Purpose
if	Execute code when condition is true
if-else	Choose between two alternatives
if-elif-else	Handle multiple conditions
Nested if	if inside another if
while	Repeat while condition is true
for	Iterate over sequences
break	Exit loop immediately
continue	Skip current iteration
try-except	Handle runtime errors

Quick Example

```
try:
    for i in range(1, 6):
        if i == 3:
            continue
        print(i)
except Exception:
    print("Error occurred")
```

Output:

```
1
2
4
5
```