History and Application Areas of Python

A simple language with a powerful impact

Simple
Readable
Powerful

History of Python

1989 — Concept & Creation
Python was created by Guido van Rossum in December 1989 at the Centrum Wiskunde & Informatica (CWI) in the Netherlands.

1991 — First Release
Python 1.0 was released in January 1991, with a focus on simplicity and readability.

2000 — Python 2.0
Released in 2000 with many new features and improvements. It became widely popular in the early 2000s.

2008 — Python 3.0
Released in 2008. It introduced major changes and is the future of Python. (Python 2 is now deprecated.)

2010s – Present — Growing Popularity
Python became one of the most popular programming languages in the world due to its simplicity, versatility and strong community support.

Today — Active Development
Python continues to evolve with regular updates (e.g., Python 3.12+), new libraries and growing usage in AI, data science, web development and more.

"From a simple idea in 1989 to a global phenomenon today — Python's journey is a story of simplicity, community and innovation."

Application Areas of Python

Python is used in a wide range of fields. Here are the major application areas:

1 Web Development
Used in frameworks like Django and Flask to build websites and web applications.

2 Data Science & Analytics
Used for data analysis, visualization and machine learning with libraries like Pandas, NumPy, Matplotlib, Seaborn.

3 Artificial Intelligence & Machine Learning
Used in AI and ML with libraries like TensorFlow, PyTorch, scikit-learn and Keras.

4 Automation & Scripting
Helps in automating repetitive tasks, system administration and writing scripts.

5 Scientific & Numerical Computing
Used for research, simulations and complex calculations with libraries like SciPy.

6 Game Development
Used to create games and interactive applications (e.g., Pygame).

7 Desktop GUI Applications
Used to build graphical user interfaces with libraries like Tkinter, PyQt.

8 Cybersecurity
Used for security analysis, penetration testing and network automation.

9 Internet of Things (IoT)
Used in smart devices, sensors and IoT systems (e.g., Raspberry Pi with Python).

Why Python?

✓ Easy to learn ✓ Readable syntax ✓ Versatile ✓ Large community ✓ High demand in the job market

Python Programming – Visual Learning

Write • Learn • Build

Simple Concepts Big Ideas!

Structure of a Python Program

A Python program is made up of different parts, each having a specific role. Here is the basic structure:

1. **Shebang Line (Optional)**
2. **Comments**
3. **Import Statement (Optional)**
4. **Variable Declaration (Assignment)**
5. **Input (Optional)**
6. **Main Logic**
7. **End of Program** (No special keyword needed)

```
1  #!/usr/bin/env python3      # Tells the system
                               # which Python to use
                               (usually not needed)

2  # This is a simple Python program

3  import math  # importing a module


5  name = "Kartik"
6  age = 20


8  # name = input("Enter your name: ")
9  # age = int(input("Enter your age: "))


11 print("Hello, " + name + "!")
12 print("You are", age, "years old.")

14 # Using a function
15 print("Square of 5 is:", math.sqrt(25))


17 # End of program
```

Key Points

- Python uses indentation instead of curly braces {}.
- Comments start with # and are ignored by the interpreter.
- Code is written in a top-to-bottom order.
- There is no required *main()* function (unlike some other languages).
- Each line of code is a statement and ends with a new line.
- Python is case-sensitive (e.g., *name* and *Name* are different).

Simple Structure ⟶ Powerful Results

Identifiers in Python

Identifiers are the names given to variables, functions, classes, modules, and other user-defined objects in a Python program. They help us identify and refer to these objects.

Rules for Identifiers

1. Must start with a letter (A–Z or a–z) or underscore (_).
2. Can contain letters, digits (0–9) and underscorses (_).
3. Cannot start with a digit (e.g., *1name* is invalid).
4. Are case-sensitive (e.g., *name* and *Name* are different).
5. Cannot be a Python keyword (e.g., *for*, *if*, *while*, etc.).

Examples

Valid Identifiers

name	✓
age	✓
total_marks	✓
_private	✓
student1	✓
user_name	✓
marks2	✓

Invalid Identifiers

1name	✗	(starts with digit)
name@	✗	(contains @)
total marks	✗	(contains space)
class	✗	(Python keyword)
for	✗	(Python keyword)
user-name	✗	(contains -)
def	✗	(Python keyword)

Keywords in Python

Keywords are reserved words in Python. They have a special meaning and cannot be used as identifiers (variable names, function names, etc.).

List of Python Keywords (33)

and	finally	or
as	for	pass
assert	from	raise
break	global	return
class	if	try
continue	import	while
def	in	with
del	is	yield
elif	lambda	
else	nonlocal	
except	not	

Remember:
Keywords are case-sensitive and cannot be used as identifiers!

Right Names + Right Words = Better Code

Operators and Precedence in Python

Perform operations • Combine values • Get the right result

Small Symbols Big Power!

Operators in Python

Operators are symbols that perform operations on values and variables. Python supports several types of operators.

1 Arithmetic Operators

Used for basic mathematical operations.

Operator	Name	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 2	2.5
%	Modulus (Remainder)	5 % 2	1
**	Exponentiation	2 ** 3	8
//	Floor Division	5 // 2	2

Example

```
a = 10
b = 3

print(a + b)    # 13
print(a - b)    # 7
print(a * b)    # 30
print(a / b)    # 3.333...
print(a % b)    # 1
print(a ** b)   # 1000
print(a // b)   # 3
```

2 Comparison Operators

Used to compare values. Returns True or False.

Operator	Meaning	Example
==	Equal to	5 == 5 → True
!=	Not equal to	5 != 3 → True
>	Greater than	5 > 3 → True
<	Less than	5 < 3 → False
>=	Greater than or equal to	5 >= 3 → True
<=	Less than or equal to	5 <= 3 → False

Example

```
x = 5
y = 3

print(x == y)   # False
print(x != y)   # True
print(x > y)    # True
print(x <= y)   # False
```

3 Logical Operators

Used to combine conditional statements.

Operator	Meaning	Example
and	Logical AND	True and False → False
or	Logical OR	True or False → True
not	Logical NOT	not True → False

Example

```
a = True
b = False

print(a and b)   # False
print(a or b)   # True
print(not a)    # False
```

4 Assignment Operators

Used to assign values to variables.

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2
%=	x %= 3	x = x % 3
**=	x **= 2	x = x ** 2
//=	x //= 2	x = x // 2

Example

```
x = 10
x += 5      # x = 15
x *= 2      # x = 30
x //= 3     # x = 10
print(x)    # 10
```

Operator Precedence in Python

Precedence decides the order in which operators are evaluated. Operators with higher precedence are evaluated first.

Precedence (Level)	Operators	Description
1	**	Exponentiation (right to left)
2	+x, -x, ~x	Unary plus, unary minus, bitwise NOT (right to left)
3	*, /, //, %	Multiplication, division, floor division, modulus (left to right)
4	+, -	Addition, subtraction (left to right)
5	<<, >>	Bitwise shift (left to right)
6	&	Bitwise AND (left to right)
7	^	Bitwise XOR (left to right)
8	|	Bitwise OR (left to right)
9	<, <=, >, >=, !=, ==	Comparison operators (left to right)
10	not	Logical NOT (right to left)
11	and	Logical AND (left to right)
12	or	Logical OR (left to right)
13	=, +=, -=, *=, /=, %=, **=, //=	Assignment operators (right to left)

Examples Showing Precedence

1. `2 + 3 * 4` → `14` (* has higher precedence)
2. `2 ** 3** 2` → `512` (right to left)
3. `(2 + 3) * 4` → `20` (parentheses have highest precedence)
4. `not True and False` → `False` (not > and)

Remember:

- Parentheses ( ) can be used to change the order.
- When operators have the same precedence, the left-to-right (or right-to-left) rule is used as shown in the table.

Understand Operators → Control the Flow → Write Better Python Code!

Basic Data Types and Type Conversion in Python

Understand data • Work with different types • Convert when needed

Same Data Different Types More Possibilities!

1 Basic Data Types in Python

Python has several built-in data types. Here are the most commonly used:

Integer (int)
Used for whole numbers (positive, negative or zero).

```
a = 10
b = -5
```

Output:
```
10   -5
```

Float (float)
Used for decimal numbers (contains a fractional part).

```
pi = 3.14
neg = -2.5
```

Output:
```
3.14   -2.5
```

Complex (complex)
Used for complex numbers (with real and imaginary parts).

```
z = 1 + 2j
```

Output:
```
(1+2j)
```

String (str)
Used for text (sequence of characters).

```
name = "Python"
msg = 'Hello'
```

Output:
```
Python
Hello
```

Boolean (bool)
Used for logical values (True or False).

```
is_valid = True
is_done = False
```

Output:
```
True
False
```

List (list)
Used to store a collection of items (ordered & changeable).

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

Output:
```
['apple', 'banana', 'cherry']
```

Tuple (tuple)
Used to store a collection of items (ordered & unchangeable).

```
point = (10, 20)
```

Output:
```
(10, 20)
```

Set (set)
Used to store unique items (unordered).

```
colors = {"red", "green", "blue"}
```

Output:
```
{'green', 'red', 'blue'}
```

Dictionary (dict)
Used to store key-value pairs (unordered, changeable).

```
student = {"name": "Kartik", "age": 20}
```

Output:
```
{'name': 'Kartik', 'age': 20}
```

Remember

- Python is dynamically typed (you don't need to declare the type).
- You can check the type using **type()**.
- Use the right data type for the right job!

Different Data Types = Better Control!

2 Type Conversion in Python

Type conversion means changing one data type to another.
Python provides built-in functions for this.

Common Type Conversion Functions

Function	Converts To	Example	Result
int()	Integer	int(3.7)	3
float()	Float	float(5)	5.0
str()	String	str(123)	'123'
bool()	Boolean	bool(0)	False
list()	List	list((1, 2, 3))	[1, 2, 3]
tuple()	Tuple	tuple([1, 2, 3])	(1, 2, 3)
set()	Set	set([1, 2, 2, 3])	{1, 2, 3}
dict()	Dictionary	dict([('a',1), ('b', 2)])	{'a': 1, 'b': 2}

Examples of Type Conversion

```
# Integer to float
x = int(5)
y = float(x)
print(y)     # 5.0

# String to integer
s = "123"
n = int(s)
print(n)     # 123

# Float to integer (removes decimal)
f = 7.8
i = int(f)
print(i)     # 7

# Boolean conversion
b = bool(0)
print(b)     # False
```

Output

```
5.0
123
7
False
```

Quick Tips

- Use **int()** for whole numbers, **float()** for decimals.
- **str()** is useful when you need to display values.
- **bool(0)**, **bool("")** and **bool([])** are **False** (empty values).
- Type conversion helps in user input, calculations and data handling.

Python + Right Data Type + Type Conversion = Powerful Programs

Statements and Expressions in Python

Statements do the work • Expressions produce values

Write Code Think Logically!

What is a Statement?

A statement is a complete instruction that performs an action (such as assigning a value, printing output, or controlling the flow of a program). It **does not return a value.**

Types of Statements in Python

1 Assignment Statement
Assigns a value to a variable.

```
name =  "Kartik"
age = 20
x = 5 + 3
```

2 Print Statement
Displays output on the screen.

```
print("Hello, Python!")
print(age)
```

3 Input Statement
Takes input from the user.

```
name = input("Enter your name: ")
print("Hello, " + name)
```

4 Conditional Statement
Executes code based on a condition.

```
if age >= 18:
    print("You are an adult")
else:
    print("You are a minor")
```

5 Loop Statement
Repeats a block of code.

```
for i in range(5):
    print(i)
```

6 Function Statement
Calls a function to perform a task.

```
def greet():
    print("Hello!")
greet()
```

7 Control Flow Statement
Changes the flow of execution.

```
break    # exit loop
continue    # skip current iteration
pass        # do nothing
```

Key Points (Statements)

- ✔ Statements end with a newline (you don't need a semicolon in Python).
- ✔ They are used to perform actions.
- ✔ Examples: assignment, print, input, if, for, while, def, break, continue, pass.

What is an Expression?

An expression is a combination of values, variables, operators, and/or function calls that **evaluates to a single value.**

Types of Expressions in Python

1 Arithmetic Expressions
Perform mathematical operations.

```
5 + 3       # 8
10 - 4      # 6
2 * 6       # 12
10 / 2      # 5.0
10 // 3     # 3
10 % 3      # 1
2 ** 3      # 8
```

Operators:
`+ - * /`
`// % **`

2 Relational Expressions
Compare values and return True or False.

```
5 > 3          # True
10 == 10       # True
7 != 5         # True
4 <= 8         # True
9 >= 12        # False
```

Operators:
`== != > <`
`>= <=`

3 Logical Expressions
Combine conditions using logical operators.

```
(5 > 3) and (2 < 8)    # True
(5 > 3) or (2 > 8)     # True
not (5 == 3)           # True
```

Operators:
`and or not`

4 Membership Expressions
Check if a value is in a sequence.

```
'a' in 'apple'         # True
5 not in [1, 2, 3]     # True
```

Operators:
`in not in`

5 Identity Expressions
Check if two variables refer to the same object.

```
x = [1, 2, 3]
y = x
x is y          # True
```

Operators:
`is is not`

Quick Summary

Statements:
- ✔ Perform actions
- ✔ Do not return a value
- ✔ Examples: print(), if, for, =, def

Expressions:
- ✔ Evaluate to a value
- ✔ Can be used in statements
- ✔ Examples: 5 + 3, x > 0, name in list

Statements give instructions → Expressions give results → Together they make Python work!

Input/Output Statements in Python

Input and output statements are used to get data from the user and display results.

1. Input Statement

- The `input()` function is used to take input from the user.
- It always takes the input as a **string**, even if the user enters a number. (You can convert it to int, float, etc.)

Syntax

```
variable = input(prompt)
```

- `variable` — Variable to store the input value
- `prompt` — Message shown to the user (optional)

Example 1: Taking a name as input

```
1 name = input("Enter your name: ")
2 print("Hello, " + name + "!")
```

Output

```
Enter your name:  Rahul
Hello, Rahul!
```

Example 2: Taking a number and converting to integer

```
1 num = input("Enter a number: ")
2 num = int(num)     # convert string to integer
3 print("Square of the number is:", num * num)
```

Output

```
Enter a number: 5
Square of the number is: 25
```

2. Output Statement

- The `print()` **function** is used to display output on the screen.
- It can display text, variables, expressions, or the result of calculations.

Syntax

```
print(value1, value2, ..., sep=' ', end='\n')
```

- `value1, value2, ...` — Values to be displayed
- `sep=' '` — Separator between values (default is space)
- `end='\n'` — End character (default is newline)

Example 1: Displaying a simple message

```
1 print("Welcome to Pythom!")
```

Output

```
Welcome to Python!
```

Example 2: Displaying multiple values

```
1 a = 10
2 b = 20
3 print("a =", a, "b =", b)
```

Output

```
a = 10 b = 20
```

Example 3: Using separator and end

```
1 print("Python", "is", "fun", sep=" | ")
2 print("End", end="!!!")
```

Output

```
Python | is | fun!!!
```

Key Points to Remember

- `input()` always returns a **string**.
- Use **type conversion** (`int()`, `float()`, etc.) if you need a number.
- `print()` can display multiple values, use `sep` and `end` to format output.
- By default, `print()` adds a newline at the end.

In Short

User Input → `input()` (get data)

`print()` → Output to Screen (show result)

Strings in Python

Creating and Storing Strings in Python

Strings are used to store and manipulate text data.

1. What is a String?

- A string is a sequence of characters.
- In Python, strings are enclosed in single quotes (`' '`), double quotes (`" "`), or triple quotes (`''' '''` / `""" """`).
- Strings are **immutable** (cannot be changed after creation).
- They are used to represent text, such as names, messages, or any sequence of characters.

Example of a string:

`"Hello, Python!"` (a string of 13 characters)

2. Creating Strings

Strings can be created using:

1 Single quotes

```
name = 'Alice'
```

2 Double quotes

```
message = "Welcome"
```

3 Triple quotes (for multi-line strings)

```
info = '''This is a
multi-line string.'''
```

Example: Different Ways to Create Strings

```
1  s1 = 'Hello'
2  s2 = "Python"
3  s3 = '''This is a
        multi-line string'''
4  s4 = """Another
        multi-line string"""
```

Output (using print):

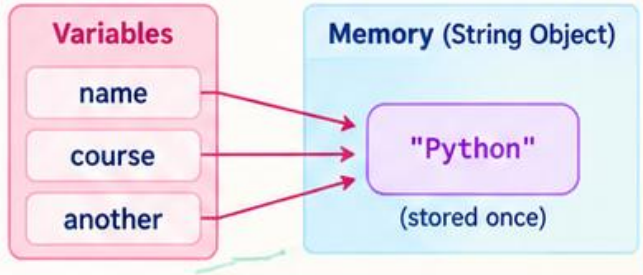
```
1  print(s1)    # Hello
2  print(s2)    # Python
3  print(s3)    # This is a
                # multi-line string
4  print(s4)    # Another
                # multi-line string
```

3. Storing Strings in Memory

- When you assign a **string to a variable**, Python stores the string in memory and **the variable** holds a reference to it.
- Strings are stored as a sequence of characters in memory.
- Multiple variables can refer to the same string (because strings are immutable).

Example

```
name = "Python"
course = "Python"
another = name
```



All three variables refer to the same string object in memory (because strings are immutable).

4. Common String Operations

Strings support many **useful** operations, such as:

Operation	Example	Result
Concatenation (+)	`"Hello" + " World"`	Hello World
Repetition (*)	`"ha" * 3`	hahaha
Length (len())	`len("Python")`	6
Indexing	`s = "Python"; s[0]`	P
Slicing	`s[1:4]`	yth
Membership (in)	`"Py" in s`	True

Key Points to Remember

- Strings are created using single quotes, double quotes or triple quotes.
- They are a sequence of characters.
- Strings are **immutable** (cannot be changed).
- When stored in memory, a variable holds a reference to the string object.
- Multiple variables can reference the same string.

In Short

Create string (using quotes) → **Store in variable** (reference in memory) → **Use string operations** (e.g., print, len, +, *)

Strings help us work with text data in Python!

Built-in Functions & String Operators in Python

Strings have many useful built-in functions and operators to work with text data.

Strings make text processing easy in Python!

1. Built-in Functions for Strings

Python provides many built-in functions (methods) to manipulate strings.

Function	Description	Example
`len()`	Returns the length of the string.	`len("Python")  # 6`
`str()`	Converts a value to string.	`str(123)  # '123'`
`upper()`	Converts all characters to uppercase.	`"hello".upper()  # 'HELLO'`
`lower()`	Converts all characters to lowercase.	`"HELLO".lower()  # 'hello'`
`strip()`	Removes leading and trailing whitespace (or specified characters).	`"  hi  ".strip()  # 'hi'`
`lstrip()`	Removes leading whitespace.	`"  hi".lstrip()  # 'hi'`
`rstrip()`	Removes trailing whitespace.	`"hi  ".rstrip()  # 'hi'`
`replace()`	Replaces a substring with another substring.	`"hello".replace("l", "L")  # 'heLLo'`
`find()`	Returns the index of the first occurrence of a substring.	`"hello".find("l")  # 2`
`count()`	Counts the occurrences of a substring.	`"banana".count("a")  # 3`
`split()`	Splits the string into a list of substrings.	`"a,b,c".split(",")  # ['a','b','c']`
`join()`	Joins elements of a list into a string.	`"-".join(["a","b","c"])  # 'a-b-c'`
`startswith()`	Checks if string starts with a substring.	`"python".startswith("py")  # True`
`endswith()`	Checks if string ends with a substring.	`"python".endswith("on")  # True`

2. String Operators

Python supports several operators to work with strings.

Operator	Name	Example	Result
`+`	Concatenation (combines strings)	`"Hello" + " World"`	`'Hello World'`
`*`	Repetition (repeats string)	`"ha" * 3`	`'hahaha'`
`in`	Membership test (checks if substring exists)	`"py" in "python"`	`True`
`not in`	Membership test (checks if substring is not present)	`"java" not in "python"`	`True`
`==`	Equality (checks if strings are equal)	`"abc" == "abc"`	`True`
`!=`	Inequality (checks if strings are not equal)	`"abc" != "xyz"`	`True`
`<`	Less than (lexicographical order)	`"apple" < "banana"`	`True`
`>`	Greater than (lexicographical order)	`"dog" > "cat"`	`True`
`<=`	Less than or equal to	`"abc" <= "abc"`	`True`
`>=`	Greater than or equal to	`"xyz" >= "abc"`	`True`

Note

- `+` and `*` are used for creating **new strings** (they don't modify the original string).
- `in`, `not in`, `==`, `!=` and **comparison operators** work with string content (lexicographical order).
- Strings are **case-sensitive** (`"Python"` and `"python"` are different).

Example: Using Built-in Functions

```
text = "  Python Programming  "
print(len(text))          # 19
print(text.strip())       # Python Programming
print(text.upper())       # PYTHON PROGRAMMING
print(text.replace("Python", "Java"))  # Java Programming
print(text.split())     # ['', 'Python', 'Programming', '']
```

Example: Using String Operators

```
s1 = "Hello"
s2 = "Python"
print(s1 + " " + s2)     # Hello Python
print(s1 * 3)            # HelloHelloHello
print("Py" in s2)        # True
print("java" not in s2)  # True
```

Key Takeaways

- **Built-in functions** help in manipulating and analyzing strings.
- String operators allow you to combine, repeat, and compare strings.
- Strings are **immutable** (cannot be changed).
- Many functions return new strings and do not modify the original.

Work with strings — turn text into useful data!

String Slicing and Joining, Formatting Strings in Python

Work with parts, combine them, and make them look great!

Slicing helps you extract parts. Joining helps you combine. Formatting makes your text look better!

1. String Slicing

- Slicing is used to get a part (substring) of a string.
- It uses the format **string[start : end : step]**.
- **start** is inclusive, **end** is exclusive.
- If **start** is omitted, it starts from the beginning.
- If **end** is omitted, it goes to the end.
- If **step** is omitted, it takes 1 character at a time.

Example:

```
s = "Python"
print(s[0:6])   # from index 0 to 5
print(s[1:4])   # characters at index 1,2,3
print(s[:3])    # from start to index 2
print(s[2:])    # from index 2 to end
print(s[::2])   # every 2nd character
print(s[::-1])  # reverse the string
```

Output:

```
Python
yth
Pyt
thon
Pto
nohtyP
```

String: Python

P	y	t	h	o	n
0	1	2	3	4	5

Slice Examples:

s[1:4]	→	yth
s[:3]	→	Pyt
s[2:]	→	thon
s[::2]	→	Pto
s[::-1]	→	nohtyP

2. String Joining

- Joining is used to combine elements of an iterable (like a list or tuple) into a single string.
- The **str.join()** method is used for joining.
- It inserts the specified separator between elements.

Example: Joining a list of strings

```
words = ["Python", "is", "fun"]
result = "-".join(words)
print(result)
```

Output:

```
Python-is-fun
```

Example: Joining characters from a string

```
chars = ["P", "y", "t", "h", "o", "n"]
result = "".join(chars)   # empty string as separator
print(result)
```

Output:

```
Python
```

Example: Joining with a space

```
words = ["Hello", "World"]
result = " ".join(words)
print(result)
```

Output:

```
Hello World
```

3. Formatting Strings

- Formatting is used to insert values into a string in a specific format.
- Python provides several ways to format strings.

A. Using f-strings (Recommended)

Direct and easy to read.

```
name = "Alice"
age = 21
print(f"Name: {name}, Age: {age}")
```

Output:

```
Name: Alice, Age: 21
```

B. Using str.format()

Flexible and widely used.

```
name = "Bob"
age = 25
print("Name: {}, Age: {}".format(name, age))
```

Output:

```
Name: Bob, Age: 25
```

C. Using % (Old Style)

Still supported, but less preferred.

```
name = "Carol"
age = 28
print("Name: %s, Age: %d" % (name, age))
```

Output:

```
Name: Carol, Age: 28
```

Formatting Options

%s	→ string	"%s" % name
%d	→ integer	"%d" % age
%f	→ float	"%f" % value
%.2f	→ float (2 decimal places)	"%.2f" % value
%10s	→ width 10 (right aligned)	"%10s" % name
%-10s	→ width 10 (left aligned)	"%-10s" % name

Key Takeaways

- **Slicing:** Use `start:end:step` to get parts of a string.
- **Joining:** Use `separator.join(iterable)` to combine strings.
- **Formatting:** Use `f-strings`, `.format()` or `%` operator to insert values into strings.

Slice it → Join it → Format it

– Strings make your data speak! –

Python Strings = Flexible + Powerful + Easy to Use

Conditional Flow Statements in Python

Decision making in Python is done using conditional flow statements.
They help the program to choose a path based on a condition.

Check a condition, then decide what to do!

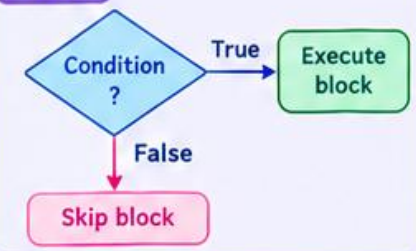
1. if Statement

The if statement executes a block of code only if the condition is true.

Syntax

```
if condition:
    # block of code (executed if condition is true)
```

Flow



Example

```
1  age = 18
2  if age >= 18:
3      print("You are eligible to vote.")
4
```

Output

```
You are eligible to vote.
```

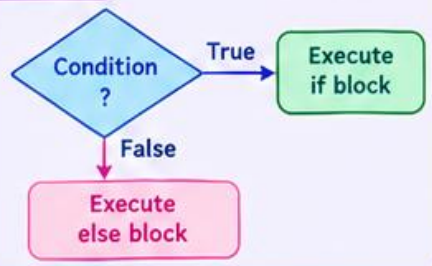
2. if...else Statement

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

Syntax

```
if condition:
    # block if true
else:
    # block if false
```

Flow



Example

```
1  num = 5
2  if num > 0:
3      print("Positive number")
4  else:
5      print("Negative number")
```

Output

```
Positive number
```

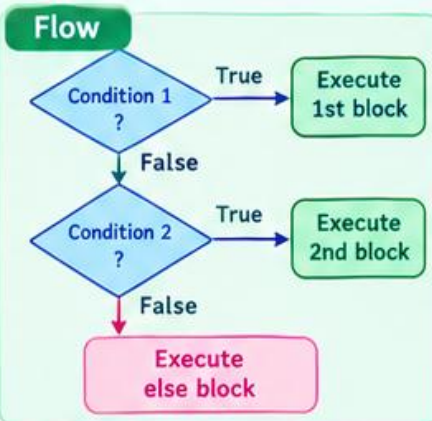
3. if...elif...else Statement

The if...elif...else statement checks multiple conditions in sequence. The first condition that is true will be executed.

Syntax

```
if condition1:
    # block if condition1 is true
elif condition2:
    # block if condition2 is true
else:
    # block if none are true
```

Flow



Example

```
1  marks = 85
2  if marks >= 90:
3      print("Grade: A")
4  elif marks >= 75:
5      print("Grade: B")
6  else:
6      print("Grade: C")
```

Output

```
Grade: B
```

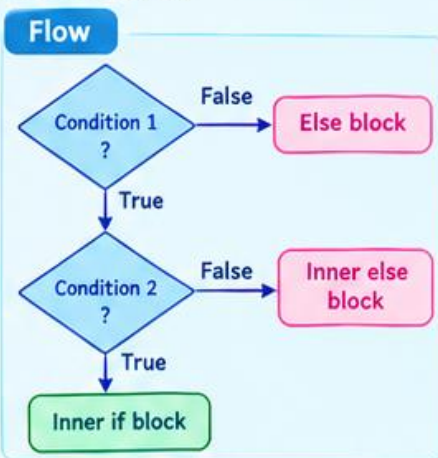
4. Nested if Statement

An if statement can be placed inside another if or else block.

Syntax

```
if condition1:
  if condition2:
      # block if both conditions are true
  else:
      # block if condition2 is false
else:
   # block if condition1 is false
```

Flow



Example

```
1  age = 20
2  if age >= 18:
3      if age >= 21:
4          print("You can drink alcohol.")
5      else:
6          print("You are an adult, but not old enough.")
7  else:
8      print("You are a minor.")
```

Output

```
You are an adult,
but not old enough.
```

Key Points to Remember

Conditions are expressions that evaluate to True or False.

- Use comparison operators (==, !=, >, <, >=, <=) and logical operators (and, or, not).
- Indentation is very important in Python (defines the block of code).
- The first true condition is executed in elif chain.
- Use nested if only when necessary (keep code simple and readable).

Common Comparison & Logical Operators

Type	Operators
Comparison	== != > < >= <=
Logical	and or not

When to Use

Use if for a single condition.

- Use if...else when you need an alternative action.
- Use if...elif...else for multiple conditions.
- Use nested if when conditions depend on each other.

Right condition → Right decision → Better program!

Loop Control Statements in Python

More control in your loops... write smarter code!

Loop control statements help us control the flow of loops.
They allow us to break, skip or control the execution of a loop based on certain conditions.

1. For Loop

- Used to iterate over a sequence (like a list, tuple, string, range, etc.).
- It runs for a fixed number of times or until the sequence ends.

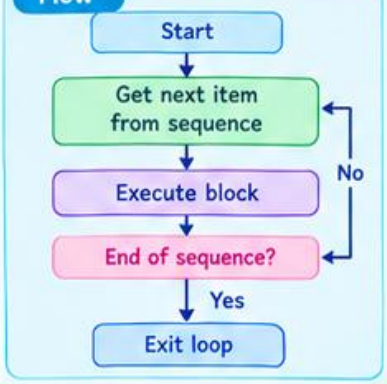
Syntax

```
for variable in sequence:
    # code to be executed
```

Example

```
for i in range(5):
    print(i)
                    # Output: 0 1 2 3 4
```

Flow



2. While Loop

- Used when the number of iterations is not known.
- It continues to execute as long as the condition is true.

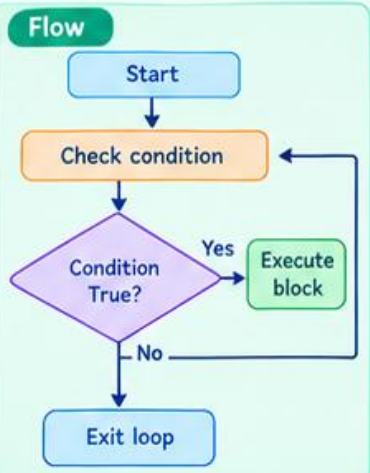
Syntax

```
while condition:
    # code to be executed
```

Example

```
n = 1
while n <= 5:
    print(n)          # Output: 1 2 3 4 5
    n += 1
```

Flow



3. break Statement

- Used to terminate the loop completely.
- Control comes out of the loop, even if the condition is not false.

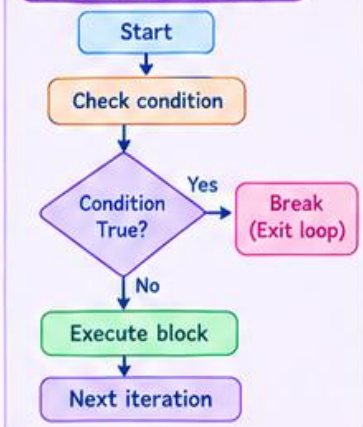
Syntax

```
break
```

Example

```
for i in range(5):
    if i == 3:
        break       # loop stops when i = 3
    print(i)
                    # Output: 0 1 2
```

Flow (with break)



4. continue Statement

- Skips the current iteration and moves to the next iteration.
- The rest of the code inside the loop (for that iteration) is not executed.

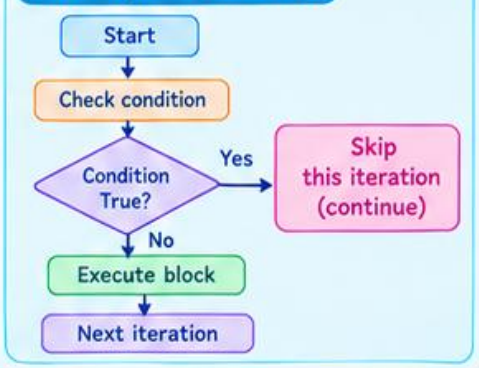
Syntax

```
continue
```

Example

```
for i in range(5):
    if i == 2:
        continue    # skip 2
    print(i)
                    # Output: 0 1 3 4
```

Flow (with continue)



5. pass Statement

- A null operation; it does nothing.
- Used when a statement is required syntactically, but you don't want to execute any code (e.g., in empty loops or placeholder).

Syntax

```
pass
```

Example

```
for i in range(5):
    if i == 3:
        pass    # does nothing
    else:
        print(i)      # Output: 0 1 2 3 4
```

Use cases of pass:

- Empty loops
- Placeholders for future code
- Defining empty functions/classes

Key Points to Remember

1. `for` loop → used for a fixed number of iterations or iterating over a sequence.
2. `while` loop → used when the number of iterations is unknown.
3. `break` → exits the loop completely.
4. `continue` → skips the current iteration and continues with the next.
5. `pass` → does nothing; used as a placeholder.

In Short

for / while (loop execution) → break (exit loop) → continue (skip iteration) → pass (do nothing)

Control the loop, control your program!