History and Applications of Python

History of Python

- Guido van Rossum was the original designer of Python in 1991.
- Development overseen by the Python Software Foundation.
- Significant versions: Python 2 and Python 3.

Python 2 and Python 3

- Python 2.0 introduced on October 16, 2000, with numerous new features.
- Python 3.0 released on December 3, 2008.

Python Features

- Generate .exe files.
- Not all programming languages are scripting languages.
- High maintenance cost.

Applications of Python

- Development of Web Applications
- Creation of Desktop GUI Applications
- Console-Based Applications

Development of Web Applications

- Python is capable for crafting web applications.
- Offers libraries for managing internet protocols like HTML, XML, JSON.
- Instagram uses Django.
- Frameworks: Django and Pyramid (robust applications), Flask and Bottle (micro-frameworks).

Creation of Desktop GUI Applications

- Python supports Graphical User Interfaces (GUIs).
- Includes Tk GUI library for interactive user interfaces.

Console-Based Applications

- Operate from the command-line or shell.
- Effective for developing text-based interactions and command execution.

Software Development

- Python is valuable in software development.
- Aids in creating control systems, management tools, and testing frameworks.

Scientific and Numeric Computing

- Suited for AI and ML due to rich scientific and mathematical libraries.
- Simplifies intricate calculations and data analysis.

Business Applications

- Ideal for e-commerce and ERP systems.
- Delivers functionality, scalability, and readability.

Multimedia Applications

- Versatile for handling audio and video tasks.
- Examples: TimPlayer and cplay.

3D CAD Applications

- Design 3D CAD applications for engineering and architectural endeavors.
- Enables creation of detailed 3D representations.

Enterprise Solutions

- Develop applications for enterprises.
- Examples: OpenERP, Tryton, and Picalo.

Image Processing

- Libraries for manipulating images.
- Examples: OpenCV and Pillow.

Conclusion

- Python's versatility and capabilities make it suitable for a wide range of applications.

Introduction to Python

An Overview

Presented by Professor Maulik Davda

What is Python?

- Python is a widely employed high-level, general-purpose programming language.
- It operates as an interpreted, object-oriented language.

Accessibility of Python

- Python is freely available and open-source.
- Ensures easy accessibility for users.

Focus of Python

- Primary focus on code reusability, readability, and the use of whitespace.

Python Characteristics

- High-level built-in data structures.
- Dynamic typing and binding.
- Appealing for swift application development.

Code Readability

- Originally conceived with an emphasis on code readability.
- Syntax enables programmers to express concepts succinctly.
- Results in fewer lines of code.

Python's Versatility

- Facilitates rapid work and enhances system integration efficiency.

Conclusion

- Python is a powerful, accessible, and versatile programming language.
- Ideal for both beginners and experienced programmers.

Keywords

- Reserved Words
- Can’t use as constant, variables, identifiers name
- Lowercase letters only
-

False	None	True	and
as	assert	break	class
continue	def	del	elif
else	except	finally	for
from	global	if	impor
in	is	lambda	nonlo
not	or	pass	raise
return	try	while	with
yield			

Identifiers

- identifiers : name used to identify a
- Starts with
  - variable
  - function
  - class
  - module
  - other objects
- Starts with letters A to Z, a to z, underscore (_)
- followed by zero or more letters, underscore (_), digit o to 9

Variables

- value can be changed during program execution
- variable is only a name given to memory location
- operations (+-*/) on variable s effects memory location
- Eg. x,y, age, total_students etc…

Rules for Python Variables

- Must starts with a letter / underscore (_)
- can’t start with number
- only contains alpha-numeric characters * underscore (_)
- eg. (A-Z, a-z, 0-9, _)
- case sensitive (age, Age, AGE all are different)

Valid v/s invalid variables

valid

myCountry

my_country

_my_country

MyCountry

MYCOUNTRY

myCountry2

invalid

2mycountry

my-country

my country

Example of Variables

code

```
#declaring variable
age = 41
salary = 123.5
name = “india”
print(age)
print(salary)
print(name)
```

Output

```
41
123.5
india
```

Variables re-declaration

Code

```
#declaring a variable
Number = 100
print(“Before :”, Number)

#re-declaring a variable
Number = 100.1
print(“After :”, Number)
```

Output

```
Before : 100
After : 100.1
```

Single value to multiple variable

Code

```
a = b = c = 10

print(a)
print(b)
print(c)
```

Output

```
10
10
10
```

different value to multiple variable

Code

```
a, b, c = 1, 10.1, “India”


print(a)
print(b)
print(c)
```

Output

```
1
10.1
India
```

Data Types & Variables

- Classification of data items
- Data types are classes
- Variables are instance (object) of the class
- Standard Built-in data types
  - Numbers - String
  - List - Tuple - Dictionary

Python Data types

- Numeric
  - integer
  - long
  - float
  - complex number
- Dictionary
- Boolean
  - subtype of plain integers
- Sequence Type
  - String
  - List
  - Tuple

Numbers

4 types of numeric types

- integers -> positive / negative
- Boolean -> Subtype of plain integers (0 / 1)
- float-> Real Numbers (integer.fractional part)
- long -> Unlimited Precesion (ચોકસાઇ)
  - integer with 20 digits (+ / -)
- complex-> 99+5j
  - 99 is Real part
  - 5j is imaginary part

Numbers Programs

```
# Integer
int_num = 42
print("Integer:", int_num)
# Float
float_num = 3.14159
print("Float:", float_num)
# Boolean
bool_true = True
bool_false = False
print("Boolean (True):", bool_true)
print("Boolean (False):", bool_false)
# Complex Number
complex_num = 2 + 3j
print("Complex Number:", complex_num)
```

```
# Demonstrating long integers (note: in Python 3, int can handle long values automatically)
long_num = 12345678901234567890
print("Long Integer:", long_num)

# Perform arithmetic operations
sum_result = int_num + float_num
print("Sum of int and float:", sum_result)

# Complex number operations
complex_sum = complex_num + (1 + 2j)
print("Sum of complex numbers:", complex_sum)
```

Numbers Programs

Output:
Integer: 42
Float: 3.14159
Boolean (True): True
Boolean (False): False
Complex Number: (2+3j)
Long Integer: 12345678901234567 89
Sum of int and float: 45.14159
Sum of complex numbers: (3+5j)

String

- Array of characters
- formed by a list of characters
- in python, string is a sequence of Unicode character.
- Quotes (both single & double)
- Python treats single quotes the same as double quotes

String Example

Code

```
str = "Hi-Double quote"
str1 = 'Hello-Single quote'

print(str)
print(str1)
```

Output

```
Hi-Double quote
Hello-Single quote
```

String

Python string are immutable (અપરિવર્તનશીલ)

- means can’t changed after created
- characters can be accessed using standard [] and zero based indexing

String Example

Code

```
str = "Hi-Double quote"
str1 = 'Hello-Single quote'


print(str[0])
print(str[6:11])
print(str + "!!")
print(len(str))
```

Output

```
H
ble q
Hi-Double quote!!
15
```

Unicode Character

- Unicode provides a unique number for every character, no matter the platform, program, or language.
- in short, A Unicode code point is a number that uniquely identifies a character.
- written in the form U+ followed by a hexadecimal number (e.g., U+0041 for the character 'A'

List

- most frequently used
- very versatile (બહુમુખી) datatype
- works similarly to string
- eg. len(), square bracket [] to access data
- first element at index 0

List Example

Code

```
weekdays = ['Mon', 'tue','wed','thur', 'fri','sat', 'sun']
weekdays2 = ["Mon", "tue","wed","thur",
"fri","sat","sun"]

print(weekdays[0])
print(weekdays2[0])

print(len(weekdays))
```

Output

```
Mon
Mon
7
```

Touple

- It is a container,
- holds a series of comma-separated values between parentheses ()
- similar to list
- both list & touple used in similar situations

Difference List & Touple

- list [square bracket]
- mutable objects

- touple (parentheses)
- immutable objects

Tuple Example

Code

```
my_touple_1 = (1,2,"Hello", 3.14, "World")
print(my_touple_1)
print(my_touple_1[3])
my_touple_2 = (5, "Six")
print(my_touple_1 + my_touple_2)
```

Output

```
(1, 2, 'Hello', 3.14, 'World')
3.14
(1, 2, 'Hello', 3.14, 'World', 5, 'Six')
```

Dictionary

- store & retrieve related information
- it contains Key-Value combination
- each key is unique
- values can be string, int, float or list
- don’t support sequence operation like string, touples & list

Dictionary Example

Code

```
student_info = {"name":"Alice","age": 10,"grade": "5th"}
print("Student Details are:", student_info)
print("Type of Student Info:", type(student_info))
```

Output

```
Student Details are: {'name': 'Alice', 'age': 10, 'grade':
'5th'}
Type of Student Info: <class 'dict'>
```

Dictionary Example(2)

Code

```
Dict = {1: 'I' ,2:'for',3:'India'}
print(Dict)
```

Output

```
Dict = {1: 'I' ,2:'for',3:'India'}
print(Dict)
```

type()

- To check the data type, built-in type() function is used
-

type() example

Code

```
x=42
print(type(x))

y="Hi"
print(type(y))

z=3.14
print(type(z))

a=False
print(type(a))
```

Output

```
x=42
print(type(x))

y="Hi"
print(type(y))

z=3.14
print(type(z))

a=False
print(type(a))
```

is operator

- type() is used to check which type of variable
- ‘is’ operator used to compare identity of two objects
- eg. 1 = 1.0
- true, but both have different data type
- first is int and second is float

is operator

Code

```
>>> x=1
>>> y=1.0
>>> x==y
True
>>> x is y
False
```

Example

```
x=1
y=1.0
print(x==y)
print(x is y)
```

Output :

```
True
False
```

is operator V/S “==”

- “==” operator checks the values of two objects are the same
-
- Basically
- is Operator: Checks whether two variables refer to the same object in memory.
- == Operator: Checks whether the values of two variables are equal.

Understanding “`is`” Operator

```
a = [1, 2, 3]
b = a
```

`a` and `b` are pointing to the same list in memory.

Therefore, `a is b` will return `True` because both `a` and `b` refer to the same object.

However, if we create `b` as a separate list:

```
a = [1, 2, 3]
b = [1, 2, 3]
```

- `a` and `b` have the same values, but they are two different objects in memory.
- `a is b` will return `False`, while `a == b` will return `True` because their contents are the same.

“is” for large number

Small Number

x = 10

y = 10

print(x is y)

Output:

True

Large Number

x = 1000

y = 1000

print(x is y)

Output

False

for larger numbers or more complex data types, Python may create separate objects:

Type Casting

Type Conversion : Process of converting value of one data type (int, float, string) to another data type

Type Casting

Implicit Type Conversion

- Automatically performed by python
- Python avoids loss of data
- Don’t need any user involvement

Explicit Type Conversion

- called as type casting
- loss of data **may** occur for specific data type

Implicit Type Conversion -

Automatically performed by python

Don’t need any user involvement

Code

```
>>> x=10
>>> print("x is of type",type(x))
x is of type <class 'int'>
>>> y=10.6
>>> print("y is of type",type(y))
y is of type <class 'float'>
>>> x=x+y
>>> print("x is of type",type(x))
x is of type <class 'float'>
```

Remarks

- variable x changed from int to float automatically
-

Explicit Type Conversion

- can be done by assigning the required data type function
- data type manually changed by the user as per requirement
- Syntax

  <required_datatype>(expression)

Explicit Type Conversion - int Example

```
x=int(1)
print("Value of x",x, type(x))

y=int(2.8)
print("Value of y",y, type(y))

z=int("3")
print("Value of z",z, type(z))

z=int("a") #error
print("Value of z",z, type(z))
```

Output

Value of x 1 <class 'int'>

Value of y 2 <class 'int'>

Value of z 3 <class 'int'>

Explicit Type Conversion - float Examp

```
x=float(1)
print("Value of x",x, type(x))

y=float(2.8)
print("Value of y",y, type(y))

z=float("3")
print("Value of z",z, type(z))

w = float("4.2")
print("Value of w",w, type(w))
```

Value of x 1.0 <class 'float'>

Value of y 2.8 <class 'float'>

Value of z 3.0 <class 'float'>

Value of w 4.2 <class 'float'>

Explicit Type Conversion - String Examp

```
x=str("s1")
print("Value of x",x, type(x))

y=str(2)
print("Value of y",y, type(y))

z=str(3.0)
print("Value of z",z, type(z))
```

Value of x s1 <class 'str'>

Value of y 2 <class 'str'>

Value of z 3.0 <class 'str'>

Input Functions

Input()

- we can ask user (ot give instruction) to enter some data
- returns a reference to the data in form of string
- it takes single parameter that is **string**
- we need to to convert input data according to our need ???

Input Functions

```
Student_name= input("Enter name:")
```

- whatever user writes
- it will store in the variable Student_name as String

Input Functions

```
Student_name= input("Enter name:")
print("Student Name in Capital :", Student_name.upper(), "Length is :", len(Student_name))
```

Output:

Enter name:Maulik

Student Name in Capital : MAULIK Length is : 6

Input Functions

If you want input in other type (integer, float, etc…) need to do type conversion explicitly.

```
radius = input("Enter radius of circle :")
radius = float (radius)
diameter = 2 * radius
print("radius is :", radius)
print("diameter is :", diameter)
```

Output

```
Enter radius of
circle :2.5
radius is : 2.5
diameter is : 5.0
```

Output Functions

Print()

- simple way to take output from python program
- it takes zero or more parameters
- default separator - >display using single blank between two words
- sep argument -> change separator character
- each print ends with a new line character by default

- by default separator & end can be change

```
print("Hello World")
print("Hello","World")
print("Hello","World", sep="***")
print("Hello","World", end="$$$")
```

Output

```
Hello World
Hello World
Hello***World
Hello World$$$
```

Formatted String

- it is a template in which words / spaces will remain constant & combined with placeholder for variable
- print with some variable and constant
- % is Format Operator

Formatted String Example

aName= ”Maulik”

age = 35

print(aName, “is”, age, “Years Old”)

Output

Maulik is 35 Years Old

Formatted String Example

aName= ”Maulik”

age = 35

print(“%s is %d Years Old” % (aName, age))

Output

Maulik is 35 Years Old

Formatted String - New Approach

```
student_name = "John"
marks = 85

# Using formatted string to include the variables in the output
print(f"Student Name: {student_name}, Marks: {marks}")
```

Output:

```
Student Name: John, Marks: 85
```

Operators

Python has 7 types of operators

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

Arithmetic Operators

+-*/ -> Skip in explanation

Modulus (%): Returns the remainder when one number is divided by another.

Example: 7 % 2 = 1

Arithmetic Operators

Floor Division (//): Divides and returns the largest integer less than or equal to the result.

Example: 7 // 2 = 3

Still Confused ??

```
a=7
b=2
print(7 % 2)#output 1
print(7 // 2)#output 3
```

Arithmetic Operators

Exponentiation (**): Raises one number to the power of another.

Example: 2 ** 3 = 8

Example

```
# Arithmetic Operators
a = 10
b = 3

print("Addition:", a + b)         # 10 + 3 = 13
print("Subtraction:", a - b)   # 10 - 3 = 7
print("Multiplication:", a * b) # 10 * 3 = 30
print("Division:", a / b)         # 10 / 3 = 3.333...
print("Modulus:", a % b)        # 10 % 3 = 1
print("Exponentiation:", a ** b) # 10 ** 3 = 1000
print("Floor Division:", a // b) # 10 // 3 = 3
```

Comparison Operators

compare two values and return either `True` or `False`.

A	B	C
Equal (==):	Checks if two values are equal.	3 == 3 (returns True)
Not equal (!=):	Checks if two values are not equal.	3 != 2 (returns True)
Greater than (>):	Checks if the left value is greater than the right.	5 > 3 (returns True)
Less than (<):	Checks if the left value is less than the right.	3 < 5 (returns True)
Greater than or equal to (>=):	Checks if the left value is greater than or equal to the right.	3 >= 2 (returns True)
Less than or equal to (<=):	Checks if the left value is less than or equal to the right.	2 <= 2 (returns True)

Example

```
# Comparison Operators
x = 5
y = 8

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

Logical Operators

used to combine conditional statements.

`and`: Returns `True` if both statements are true.

- Example: `(5 > 3) and (2 < 4)` (returns `True`)

`or`: Returns `True` if at least one statement is true.

- Example: `(5 < 3) or (2 < 4)` (returns `True`)

`not`: Reverses the result.

- Example: `not(5 > 3)` (returns `False`)

Example

```
# Logical Operators
a = True
b = False

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

Bitwise Operators

perform operations on the binary level (bit by bit)

AND (`&`): Performs binary AND.

- Example: `5 & 3` (returns `1`) because `5` is `101` and `3` is `011`; bitwise AND results in `001`.

OR (`|`): Performs binary OR.

- Example: `5 | 3` (returns `7`) because `101 | 011 = 111`.

XOR (`^`): Performs binary XOR.

- Example: `5 ^ 3` (returns `6`) because `101 ^ 011 = 110`.

NOT (`~`): Flips the bits.

- Example: `~5` (returns `-6`).

Left Shift (`<<`): Shifts bits to the left.

- Example: `5 << 1` (returns `10`).

Right Shift (`>>`): Shifts bits to the right.

- Example: `5 >> 1` (returns `2`).

Example

```
# Bitwise Operators
x = 5  # Binary: 101
y = 3  # Binary: 011

print("x & y:", x & y)   # 101 & 011 = 001 (1 in decimal)
print("x | y:", x | y)   # 101 | 011 = 111 (7 in decimal)
print("x ^ y:", x ^ y)   # 101 ^ 011 = 110 (6 in decimal)
print("~x:", ~x)         # ~101 = -110 (-6 in decimal)
print("x << 1:", x << 1) # 101 << 1 = 1010 (10 in decimal)
print("x >> 1:", x >> 1) # 101 >> 1 = 10 (2 in decimal)
```

Assignment Operators

used to assign values to variables.

=: Assigns the right-hand side value to the left-hand side variable.

- Example: `x = 5`

+=: Adds and assigns.

- Example: `x += 3` (same as `x = x + 3`)

-=: Subtracts and assigns.

- Example: `x -= 2` (same as `x = x - 2`)

***=**: Multiplies and assigns.

- Example: `x *= 4` (same as `x = x * 4`)

/=: Divides and assigns.

- Example: `x /= 2` (same as `x = x / 2`)

%=: Modulus and assigns.

- Example: `x %= 3` (same as `x = x % 3`)

Example

```
# Assignment Operators
x = 10
x += 5  # Equivalent to: x = x + 5
print("x after += 5:", x)  # 15

x -= 3  # Equivalent to: x = x - 3
print("x after -= 3:", x)  # 12

x *= 2  # Equivalent to: x = x * 2
print("x after *= 2:", x)  # 24

x /= 4  # Equivalent to: x = x / 4
print("x after /= 4:", x)  # 6.0
```

Identity Operators

check whether two objects are the same.

`is`: Returns `True` if both variables point to the same object.

- Example: `x is y`

`is not`: Returns `True` if both variables do not point to the same object.

- Example: `x is not y`

Example

```
# Identity Operators
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print("a is b:", a is b)      # False (a and b are different objects)
print("a is c:", a is c)      # True (a and c refer to the same object)
print("a is not b:", a is not b)  # True
```

Membership Operators

used to test whether a sequence contains a certain value.

`in`: Returns `True` if a value is found in a sequence (like a list, string, etc.).

- Example: `'a' in 'apple'` (returns `True`)

`not in`: Returns `True` if a value is not found in a sequence.

- Example: `'b' not in 'apple'` (returns `True`)

Example

```
# Membership Operators
lst = [1, 2, 3, 4, 5]
string = "hello"

print("2 in lst:", 2 in lst)       # True
print("6 in lst:", 6 in lst)       # False
print("'h' in string:", 'h' in string) # True
print("'x' not in string:", 'x' not in string) # True
```

Operator Precedence

determines the order in which operations are evaluated in an expression. Operators with higher precedence are evaluated first. When operators have the same precedence, they are evaluated according to their associativity (either left-to-right or right-to-left).

Operator Precedence

Precedence Level	Operator Type	Operators	Associativity
1	Parentheses	()	N/A
2	Exponentiation	**	Right-to-left
3	Unary operators	+, -, ~ (positive, negative, bitwise NOT)	Right-to-left
4	Multiplication group	*, /, //, %	Left-to-right
5	Addition group	+, -	Left-to-right
6	Bitwise shifts	<<, >>	Left-to-right
7	Bitwise AND	&	Left-to-right

Operator Precedence

Precedence Level	Operator Type	Operators	Associativity
8	Bitwise XOR	^	Left-to-right
9	Bitwise OR	`	`
10	Comparison	==, !=, >, <, >=, <=, is, is not, in, not in	Left-to-right
11	Logical NOT	not	Right-to-left
12	Logical AND	and	Left-to-right
13	Logical OR	or	Left-to-right
14	Conditional	if ... else	Right-to-left
15	Assignment	=, +=, -=, *=, /=, //=, %=	Right-to-left