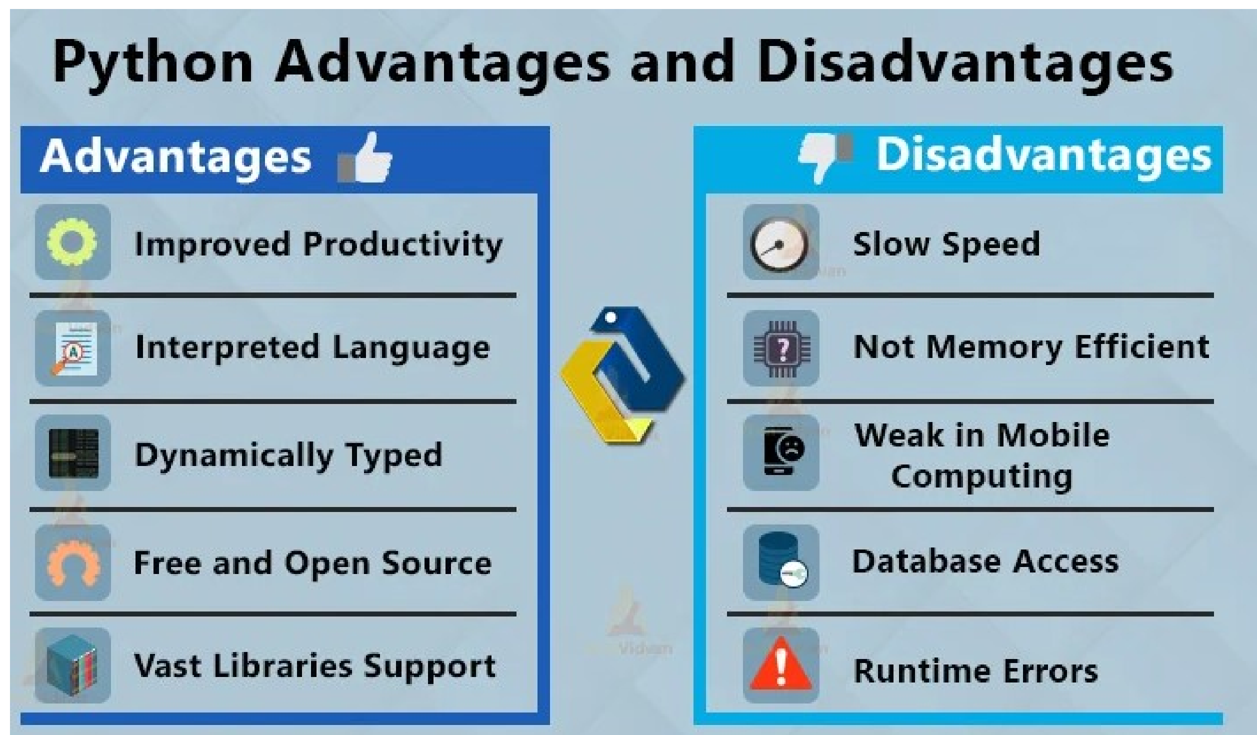
- Strengths and Weaknesses



- IDLE

Every Python installation comes with an Integrated Development and Learning Environment, which you'll see shortened to IDLE or even IDE. These are a class of applications that help you write code more efficiently. While there are many IDEs for you to choose from, Python IDLE is very bare-bones, which makes it the perfect tool for a beginning programmer.

Python IDLE comes included in Python installations on Windows and Mac. If you're a Linux user, then you should be able to find and download Python IDLE using your package manager. Once you've installed it, you can then use Python IDLE as an interactive interpreter or as a file editor

- # Dynamic Typing

In Dynamic Typing, type checking is performed at runtime. For example, Python is a dynamically typed language. It means that the type of a variable is allowed to change over its lifetime. Other dynamically typed languages are -Perl, Ruby, PHP, Javascriptetc. Let's take a Python code example to see if a variable can change type

```
## variable a is assigned to a string
a ="hello"
print(type(a))

## variable a is assigned to an integer
a = 5
print(type(a))
```

This confirms that the type of variable "a" is allowed to change and Python correctly infers the type as it changes.

Let's take another example of Dynamic Typing in Python

```
## simple function
def add(a, b):
    return a + b

## calling the function with string
print(add('hello', 'world'))

## calling the function with integer
print(add(2, 4))
```

In Python, we don't really have a good idea of what are the types that this function deals with and also what the type of the return value is going to be

```
a = 12.0
print(type(a))
b = 24
print(type(b))
c = 'data'
print(type(c))
print (a * 3)
print (b * 3)
print (c * 3)
```

Output:

```
<class 'float'>
<class 'int'>
<class 'str'>
36.0
72
```

```
datadatadata
```

- Relationship Between Objects, Variables and References

```
a = 12.0
print (type(a))
a = 24
print(type(a))
a = 'data'
print (type(a))
a = 2+3j
print (type(a))
```

Output:

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

Python Naming Conventions

1. General

- Avoid using names that are too general or too wordy. Strike a good balance between the two.
- Bad: data_structure, my_list, info_map, dictionary_for_the_purpose_of_storing_data_representing_word_definitions
- Good: user_profile, menu_options, word_definitions
- Don’t be a jackass and name things “O”, “l”, or “I”
- When using CamelCase names, capitalize all letters of an abbreviation (e.g. HTTPServer)

2. Packages

- Package names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

3. Modules

- Module names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

4. Classes

- Class names should follow the UpperCaseCamelCase convention
- Python’s built-in classes, however are typically lowercase words
- Exception classes should end in “Error”

5. Global (module-level) Variables

- Global variables should be all lowercase
- Words in a global variable name should be separated by an underscore

6. Instance Variables

- Instance variable names should be all lower case
- Words in an instance variable name should be separated by an underscore
- Non-public instance variables should begin with a single underscore
- If an instance name needs to be mangled, two underscores may begin its name

7. Methods

- Method names should be all lower case
- Words in an method name should be separated by an underscore
- Non-public method should begin with a single underscore
- If a method name needs to be mangled, two underscores may begin its name

8. Method Arguments

- Instance methods should have their first argument named ‘self’.
- Class methods should have their first argument named ‘cls’

9. Functions

- Function names should be all lower case
- Words in a function name should be separated by an underscore

10. Constants

- Constant names must be fully capitalized
- Words in a constant name should be separated by an underscore

Case Type	Description	Example
Snake Case	All words lower case separated by underscores	`my_function`, `user_input`
Pascal Case	The first letter of each word is capitalized except the first word	`MyClass`, `AnotherClass`
Camel Case	First letter of each word is capitalized except the first word	`myVariable`, `isInstance`
Upper Case with Underscores	All letters are upper case and words are separated by underscores	`MAX_OVERFLOW`, `TOTAL`

Python Global Variable Naming Conventions

In Python, a variable declared outside of a function or a block of code is referred to as a global variable. These global variables can be accessed by any function in the program.

When it comes to naming global variables in Python, the PEP8 style guide provides clear guidelines.

As a convention, global variables should be named following the snake_case style. This means that if your variable name consists of multiple words, they should be in lowercase and separated by underscores.

Here's an example:

```
# Correct way
global_variable = "I'm a global variable"
```

That said, it's important to note that using global variables should generally be avoided as they can lead to confusing code and may be prone to errors.

In cases where you need constants (i.e., variables whose values won't change), Python's convention is to use uppercase letters and separate words with underscores. These are technically global variables as well, but they have different use cases.

Here’s an example:

```
# Correct way
MAX_SIZE = 100
PI_VALUE = 3.14159
```

As we see above, constants such as MAX_SIZE and PI_VALUE are in uppercase, emphasizing that they should not be changed.

Python Class Naming Conventions

In Python, the naming convention for classes is PascalCase, which means that the first letter of each word is capitalized and there are no underscores between words.

Here’s an example:

```
# Correct way
class MyNewClass:
   pass

# Incorrect way
class my_new_class:
   pass
```

Python Class Method Naming Convention

Methods in Python classes should be named using snake_case. This means that all words should be in lowercase and separated by underscores.

Here’s an example:

```
class MyClass:
   def my_method(self):
      pass
```

Python Class Attribute Naming Convention

Similar to methods, class attributes (variables defined in the class) should also be named using snake_case in Python.

Here's an example:

```
class MyClass:
   my_attribute = 10
```

Python Class File Naming Convention

When naming Python files that contain classes, you should follow the snake_case convention as well. This helps make it clear and readable, especially when importing modules.

Here's an example: `my_python_file.py`

Python Class Instance Naming Convention

When creating instances of a class (also known as objects) in Python, it's recommended to use snake_case as well. This keeps it consistent with the rest of your variables.

Here's an example:

```
class MyClass:
   pass

my_class_instance = MyClass()
```

Python Object Naming Conventions

Objects in Python are instances of a class that you've defined. The naming convention for Python objects follows the same rules as that for regular variables.

When you create an object, the name should be in snake_case, which means all words are in lowercase, and words are separated by underscores. The name should be descriptive, making it clear what the object represents. Avoid using Python keywords and function names as object names.

Let's consider a class named `Car`:

```
class Car:
   def __init__(self, color, make, model):
      self.color = color
      self.make = make
```

```
        self.model = model
```

When creating an object (or instance) of the `Car` class, the name should adhere to the snake_case convention. Here's an example:

```
# Correct way
my_car = Car('red', 'Toyota', 'Corolla')

# Incorrect way
MyCar = Car('red', 'Toyota', 'Corolla')
myCar = Car('red', 'Toyota', 'Corolla')
```

The example above, `my_car` clearly communicates that this is an instance of a car and follows the snake_case naming convention.

Python Variable Naming Conventions

In Python, variable names follow the snake_case naming convention as per the PEP8 style guide. This means that all words in the name are in lowercase, and each word is separated by an underscore.

Here's an example:

```
# Correct way
my_variable = 10

# Incorrect way
myVariable = 10
MyVariable = 10
```

Here are a few more guidelines when naming variables in Python:

- Variable names should be descriptive and meaningful to make the code more readable and understandable.
-

For instance, if you have a variable holding the number of users, `num_users` would be a clear and understandable choice.

```
num_users = 500
```

- Avoid using single-character variable names, except for common ones like `i` or `j` in loops.

```
for i in range(10):
   print(i)
```

- Do not start variable names with a number or special character.

```
# Incorrect
123abc = "Hello"
@name = "Hello"

# Correct
```

```
abc123 = "Hello"
name = "Hello"
```

- Avoid using Python keywords and function names as variable names. For instance, don't name your variable list, dict, str, print, etc.

```
# Incorrect
list = [1, 2, 3]

# Correct
my_list = [1, 2, 3]
```

Python Function Naming Convention

In Python, function names follow the snake_case naming convention as per the PEP8 style guide. This means all words should be in lowercase, and each word is separated by an underscore.

Here's an example:

```
# Correct way
def my_function():
    pass

# Incorrect way
def MyFunction():
    pass

def myFunction():
    pass
```

Here are a few more guidelines when naming functions in Python:

- **Descriptive Names**: Function names should be descriptive and indicate the function's purpose. This improves readability and understanding of the code.

```
# Correct way
def calculate_average():
    pass

# Incorrect way
def func1():
    pass
```

In the example above, calculate_average gives a clear indication of what the function is supposed to do, while func1 doesn't provide any meaningful context.

- **Avoid Using Python Keywords and Function Names**: Avoid naming your function the same as Python keywords or existing function names. This can lead to unwanted behavior and confusion.

```
# Incorrect
def print():
    pass

# Correct
def print_custom_message():
    pass
```

- **Use Action Words**: Since functions usually perform an action, it's a good practice to start the function name with a verb. This immediately gives an idea of what action the function performs.

```
# Correct
def get_total():
    pass

# Correct
def print_report():
    pass
```

Python Strings

In computer programming, a string is a sequence of characters. For example, `"hello"` is a string containing a sequence of characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`.

We use single quotes or double quotes to represent a string in Python. For example,

```
# create a string using double quotes
string1 = "Python programming"

# create a string using single quotes
string1 = 'Python programming'
```

Here, we have created a string variable named `string1`. The variable is initialized with the string `Python Programming`.

Example: Python String

```
# create string type variables

name = "Python"
print(name)

message = "I love Python."
print(message)
```
Run Code

Output

```
Python
I love Python.
```

In the above example, we have created string-type variables: `name` and `message` with values `"Python"` and `"I love Python"` respectively.

Here, we have used double quotes to represent strings but we can use single quotes too.

Access String Characters in Python

We can access the characters in a string in three ways.

- **Indexing:** One way is to treat strings as a list and use index values. For example,

```
greet = 'hello'

# access 1st index element
print(greet[1]) # "e"
```
Run Code

- **Negative Indexing:** Similar to a list, Python allows negative indexing for its strings. For example,

```
greet = 'hello'

# access 4th last element
print(greet[-4]) # "e"
```
Run Code

- **Slicing:** Access a range of characters in a string by using the slicing operator colon `:`. For example,

```
greet = 'Hello'

# access character from 1st index to 3rd index
print(greet[1:4])  # "ell"
```
Run Code

Note: If we try to access an index out of the range or use numbers other than an integer, we will get errors.

Python Strings are immutable

In Python, strings are immutable. That means the characters of a string cannot be changed. For example,

```
message = 'Hola Amigos'
message[0] = 'H'
print(message)
```

Run Code

Output

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

However, we can assign the variable name to a new string. For example,

```
message = 'Hola Amigos'

# assign new string to message variable
message = 'Hello Friends'

prints(message); # prints "Hello Friends"
```

Run Code

Python Multiline String

We can also create a multiline string in Python. For this, we use triple double quotes """ or triple single quotes `'''`. For example,

```
# multiline string
message = """
Never gonna give you up
Never gonna let you down
"""

print(message)
```

Run Code

Output

```
Never gonna give you up
```

```
Never gonna let you down
```

In the above example, anything inside the enclosing triple-quotes is one multiline string.

Python String Operations

There are many operations that can be performed with strings which makes it one of the most used data types in Python.

1. Compare Two Strings

We use the `==` operator to compare two strings. If two strings are equal, the operator returns `True`. Otherwise, it returns `False`. For example,

```
str1 = "Hello, world!"
str2 = "I love Python."
str3 = "Hello, world!"

# compare str1 and str2
print(str1 == str2)

# compare str1 and str3
print(str1 == str3)
```
Run Code

Output

```
False
True
```

In the above example,

- `str1` and `str2` are not equal. Hence, the result is `False`.
- `str1` and `str3` are equal. Hence, the result is `True`.

2. Join Two or More Strings

In Python, we can join (concatenate) two or more strings using the `+` operator.

```
greet = "Hello, "
name = "Jack"

# using + operator
result = greet + name
print(result)
```

```
# Output: Hello, Jack
```
Run Code

In the above example, we have used the `+` operator to join two strings: `greet` and `name`.

Iterate Through a Python String

We can iterate through a string using a for loop. For example,

```
greet = 'Hello'

# iterating through greet string
for letter in greet:
    print(letter)
```
Run Code

Output

```
H
e
l
l
o
```

Python String Length

In Python, we use the `len()` method to find the length of a string. For example,

```
greet = 'Hello'

# count length of greet string
print(len(greet))

# Output: 5
```
Run Code

String Membership Test

We can test if a substring exists within a string or not, using the keyword `in`.

```
print('a' in 'program') # True
print('at' not in 'battle') False
```
Run Code

Methods of Python String

Besides those mentioned above, there are various string methods present in Python. Here are some of those methods:

Methods	Description
upper()	converts the string to uppercase
lower()	converts the string to lowercase
partition()	returns a tuple
replace()	replaces substring inside
find()	returns the index of first occurrence of substring
rstrip()	removes trailing characters
split()	splits string from left
startswith()	checks if string starts with the specified string
isnumeric()	checks numeric characters
index()	returns index of substring

Escape Sequences in Python

The escape sequence is used to escape some of the characters present inside a string.

Suppose we need to include both double quote and single quote inside a string,

```
example = "He said, "What's there?""

print(example) # throws error
```
Run Code

Since strings are represented by single or double quotes, the compiler will treat `"He said, "` as the string. Hence, the above code will cause an error.

To solve this issue, we use the escape character `\` in Python.

```
# escape double quotes
example = "He said, \"What's there?\""

# escape single quotes
example = 'He said, "What\'s there?"'

print(example)

# Output: He said, "What's there?"
```
Run Code

Here is a list of all the escape sequences supported by Python.

Escape Sequence	Description
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\a`	ASCII Bell
`\b`	ASCII Backspace
`\f`	ASCII Formfeed
`\n`	ASCII Linefeed

`\r`	ASCII Carriage Return
`\t`	ASCII Horizontal Tab
`\v`	ASCII Vertical Tab
`\ooo`	Character with octal value ooo
`\xHH`	Character with hexadecimal value HH

Python String Formatting (f-Strings)

Python **f-Strings** make it really easy to print values and variables. For example,

```
name = 'Cathy'
country = 'UK'

print(f'{name} is from {country}')
```

Run Code

Output

```
Cathy is from UK
```

Here, `f'{name} is from {country}'` is an **f-string**.

This new formatting syntax is powerful and easy to use. From now on, we will use f-Strings to print strings and variables.

How String slicing in Python works

For **understanding slicing** we will use different methods, here we will cover 2 methods of string slicing, one using the in-build slice() method and another using the **[:] array slice**. String slicing in Python is about obtaining a sub-string from the given string by slicing it respectively from start to end.

Python slicing can be done in two ways:

- Using a slice() method
- Using the array slicing [::] method
-

Index tracker for positive and negative index: String indexing and slicing in python. Here, the Negative comes into consideration when tracking the string in reverse.

Method 1: Using the slice() method

The slice() constructor creates a slice object representing the set of indices specified by range(start, stop, step).
Syntax:

- *slice(stop)*
- *slice(start, stop, step)*

Parameters: start: *Starting index where the slicing of object starts.* ***stop:*** *Ending index where the slicing of object stops.* ***step:*** *It is an optional argument that determines the increment between each index for slicing.* ***Return Type:*** *Returns a sliced object containing elements in the given range only.*
Example:

- Python3

```
# Python program to demonstrate

# string slicing



# String slicing

String = 'ASTRING'



# Using slice constructor

s1 = slice(3)

s2 = slice(1, 5, 2)
```

```
s3 = slice(-1, -12, -2)



print("String slicing")

print(String[s1])

print(String[s2])

print(String[s3])
```

Output:

```
String slicing

AST

SR

GITA
```

Method 2: Using the List/array slicing [::] method

In Python, indexing syntax can be used as a substitute for the slice object. This is an easy and convenient way to slice a string using list slicing and Array slicing both syntax-wise and execution-wise. A start, end, and step have the same mechanism as the slice() constructor.
Below we will see **string slicing in Python with examples.**
Syntax

```
arr[start:stop]         # items start through stop-1

arr[start:]             # items start through the rest of the array

arr[:stop]              # items from the beginning through stop-1

arr[:]                  # a copy of the whole array

arr[start:stop:step]    # start through not past stop, by step
```

Example 1:

In this example, we will see **slicing in python list** the index start from 0 indexes and ending with a 2 index(stops at 3-1=2).

- Python3

```
# Python program to demonstrate

# string slicing
```

```
# String slicing

String = 'hellohowareyou'



# Using indexing sequence

print(String[:3])
```

Output:

Hel

- ```
  b = "Hello, World!"
  print(b[2:5])
  ```

output:

llo

- ```
  b = "Hello, World!"
  print(b[:5])
  ```

output:

Hello

- `>>> s = 'mybacon'`
- `>>> s[0:7:2]`
- `'mbcn'`
- `>>> s[1:7:2]`
- `'yao'`
- `>>> s = '12345' * 5`
- `>>> s`
- `'1234512345123451234512345'`
- `>>> s[::5]`
- `'11111'`
- `>>> s[4::5]`
- `'55555'`

Storing Operators

he operator module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, operator.add(x, y) is equivalent to the expression x+y. The function names are those used for special class methods; variants without leading and trailing __ are also provided for convenience.

```
import operator
operations = [operator.add, operator.sub]
# add two numbers
s = operations[0](1, 2)
```

Types of Python Operators

Here's a list of different types of Python operators that we will learn in this tutorial.

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

1. Python Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc. For example,

```
sub = 10 - 5 # 5
```

Here, `-` is an arithmetic operator that subtracts two values or variables.

Operator	Operation	Example
`+`	Addition	`5 + 2 = 7`
`-`	Subtraction	`4 - 2 = 2`

`*`	Multiplication	`2 * 3 = 6`
`/`	Division	`4 / 2 = 2`
`//`	Floor Division	`10 // 3 = 3`
`%`	Modulo	`5 % 2 = 1`
`**`	Power	`4 ** 2 = 16`

Example 1: Arithmetic Operators in Python

```
a = 7
b = 2

# addition
print ('Sum: ', a + b)

# subtraction
print ('Subtraction: ', a - b)

# multiplication
print ('Multiplication: ', a * b)

# division
print ('Division: ', a / b)

# floor division
print ('Floor Division: ', a // b)

# modulo
print ('Modulo: ', a % b)

# a to the power b
print ('Power: ', a ** b)
```

Run Code

Output

```
Sum: 9
Subtraction: 5
Multiplication: 14
Division: 3.5
```

```
Floor Division: 3
Modulo: 1
Power: 49
```

In the above example, we have used multiple arithmetic operators,

- `+` to add `a` and `b`
- `-` to subtract `b` from `a`
- `*` to multiply `a` and `b`
- `/` to divide `a` by `b`
- `//` to floor divide `a` by `b`
- `%` to get the remainder
- `**` to get `a` to the power `b`

2. Python Assignment Operators

Assignment operators are used to assign values to variables. For example,

```
# assign 5 to x
var x = 5
```

Here, `=` is an assignment operator that assigns `5` to `x`.

Here's a list of different assignment operators available in Python.

Operator	Name	Example
`=`	Assignment Operator	`a = 7`
`+=`	Addition Assignment	`a += 1 # a = a + 1`
`-=`	Subtraction Assignment	`a -= 3 # a = a - 3`
`*=`	Multiplication Assignment	`a *= 4 # a = a * 4`
`/=`	Division Assignment	`a /= 3 # a = a / 3`

`%=`	Remainder Assignment	`a %= 10 # a = a % 10`
`**=`	Exponent Assignment	`a **= 10 # a = a ** 10`

Example 2: Assignment Operators

```
# assign 10 to a
a = 10

# assign 5 to b
b = 5

# assign the sum of a and b to a
a += b      # a = a + b

print(a)

# Output: 15
```

Run Code

Here, we have used the `+=` operator to assign the sum of `a` and `b` to `a`.

Similarly, we can use any other assignment operators according to the need.

3. Python Comparison Operators

Comparison operators compare two values/variables and return a boolean result: `True` or `False`. For example,

```
a = 5
b =2

print (a > b)    # True
```

Run Code

Here, the `>` comparison operator is used to compare whether `a` is greater than `b` or not.

Operator	Meaning	Example
`==`	Is Equal To	`3 == 5` gives us **False**

`!=`	Not Equal To	`3 != 5` gives us **True**
`>`	Greater Than	`3 > 5` gives us **False**
`<`	Less Than	`3 < 5` gives us **True**
`>=`	Greater Than or Equal To	`3 >= 5` give us **False**
`<=`	Less Than or Equal To	`3 <= 5` gives us **True**

Example 3: Comparison Operators

```
a = 5

b = 2

# equal to operator
print('a == b =', a == b)

# not equal to operator
print('a != b =', a != b)

# greater than operator
print('a > b =', a > b)

# less than operator
print('a < b =', a < b)

# greater than or equal to operator
print('a >= b =', a >= b)

# less than or equal to operator
print('a <= b =', a <= b)
```

Run Code

Output

```
a == b = False
a != b = True
a > b = True
a < b = False
a >= b = True
a <= b = False
```

Note: Comparison operators are used in decision-making and loops. We'll discuss more of the comparison operator and decision-making in later tutorials.

4. Python Logical Operators

Logical operators are used to check whether an expression is `True` or `False`. They are used in decision-making. For example,

```
a = 5
b = 6

print((a > 2) and (b >= 6))    # True
```

Run Code

Here, `and` is the logical operator **AND**. Since both `a > 2` and `b >= 6` are `True`, the result is `True`.

Operator	Example	Meaning
`and`	a **and** b	**Logical AND**: `True` only if both the operands are `True`
`or`	a **or** b	**Logical OR**: `True` if at least one of the operands is `True`
`not`	**not** a	**Logical NOT**: `True` if the operand is `False` and vice-versa.

Example 4: Logical Operators

```
# logical AND
print(True and True)     # True
print(True and False)    # False

# logical OR
print(True or False)     # True

# logical NOT
print(not True)          # False
```

Run Code

Note: Here is the truth table for these logical operators.

5. Python Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, **2** is `10` in binary and **7** is `111`.

In the table below: Let `x` = 10 (`0000 1010` in binary) and `y` = 4 (`0000 0100` in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (`0000 0000`)
|	Bitwise OR	x | y = 14 (`0000 1110`)
~	Bitwise NOT	~x = -11 (`1111 0101`)
^	Bitwise XOR	x ^ y = 14 (`0000 1110`)
>>	Bitwise right shift	x >> 2 = 2 (`0000 0010`)
<<	Bitwise left shift	x << 2 = 40 (`0010 1000`)

6. Python Special operators

Python language offers some special types of operators like the **identity** operator and the **membership** operator. They are described below with examples.

Identity operators

In Python, `is` and `is not` are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example

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

Example 4: Identity operators in Python

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

print(x1 is not y1)  # prints False

print(x2 is y2)  # prints True

print(x3 is y3)  # prints False
```

Run Code

Here, we see that `x1` and `y1` are integers of the same values, so they are equal as well as identical. Same is the case with `x2` and `y2` (strings).

But `x3` and `y3` are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

Membership operators

In Python, `in` and `not in` are the membership operators. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
`in`	`True` if value/variable is **found** in the sequence	`5 in x`
`not in`	`True` if value/variable is **not found** in the sequence	`5 not in x`

Example 5: Membership operators in Python

```
x = 'Hello world'
y = {1:'a', 2:'b'}

# check if 'H' is present in x string
print('H' in x)  # prints True

# check if 'hello' is present in x string
print('hello' not in x)  # prints True

# check if '1' key is present in y
print(1 in y)  # prints True

# check if 'a' key is present in y
print('a' in y)  # prints False
```

Run Code

Output

```
True
True
True
False
```

Here, `'H'` is in `x` but `'hello'` is not present in `x` (remember, Python is case sensitive).

Similarly, `1` is key and `'a'` is the value in dictionary `y`. Hence, `'a' in y` returns `False`.

Python Data Types

In this tutorial, you will learn about different data types we can use in Python with the help of examples.

In computer programming, data types specify the type of data that can be stored inside a variable. For example,

```
num = 24
```

Here, **24** (an integer) is assigned to the `num` variable. So the data type of `num` is of the `int` class.

Python Data Types

Data Types	Classes	Description
Numeric	int, float, complex	holds numeric values
String	str	holds sequence of characters
Sequence	list, tuple, range	holds collection of items
Mapping	dict	holds data in key-value pair form
Boolean	bool	holds either `True` or `False`

Set	set, frozenset	hold collection of unique items

Since everything is an object in Python programming, data types are actually classes and variables are instances(object) of these classes.

Python Numeric Data type

In Python, numeric data type is used to hold numeric values.

Integers, floating-point numbers and complex numbers fall under Python numbers category. They are defined as `int`, `float` and `complex` classes in Python.

- `int` - holds signed integers of non-limited length.
- `float` - holds floating decimal points and it's accurate up to **15** decimal places.
- `complex` - holds complex numbers.

We can use the `type()` function to know which class a variable or a value belongs to.

Let's see an example,

```
num1 = 5
print(num1, 'is of type', type(num1))

num2 = 2.0
print(num2, 'is of type', type(num2))

num3 = 1+2j
print(num3, 'is of type', type(num3))
```

Run Code

Output

```
5 is of type <class 'int'>
```

```
2.0 is of type <class 'float'>
(1+2j) is of type <class 'complex'>
```

In the above example, we have created three variables named `num1`, `num2` and `num3` with values **5**, **5.0**, and `1+2j` respectively.
We have also used the `type()` function to know which class a certain variable belongs to.
Since,

- **5** is an integer value, `type()` returns `int` as the class of `num1` i.e <class 'int'>
- **2.0** is a floating value, `type()` returns `float` as the class of `num2` i.e <class 'float'>
- `1 + 2j` is a complex number, `type()` returns `complex` as the class of `num3` i.e <class 'complex'>

Python List Data Type

List is an ordered collection of similar or different types of items separated by commas and enclosed within brackets `[ ]`. For example,

```
languages = ["Swift", "Java", "Python"]
```

Here, we have created a list named `languages` with **3** string values inside it.

Access List Items

To access items from a list, we use the index number **(0, 1, 2 ...)**. For example,

```
languages = ["Swift", "Java", "Python"]

# access element at index 0
print(languages[0])   # Swift
```

```
# access element at index 2
print(languages[2])   # Python
```
Run Code

In the above example, we have used the index values to access items from the languages list.

- `languages[0]` - access first item from `languages` i.e. `Swift`
- `languages[2]` - access third item from `languages` i.e. `Python`

Python Tuple Data Type

Tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

In Python, we use the parentheses `()` to store items of a tuple. For example,

```
product = ('Xbox', 499.99)
```

Here, `product` is a tuple with a string value `Xbox` and integer value **499.99**.

Access Tuple Items

Similar to lists, we use the index number to access tuple items in Python . For example,

```
# create a tuple
product = ('Microsoft', 'Xbox', 499.99)

# access element at index 0
print(product[0])   # Microsoft

# access element at index 1
print(product[1])   # Xbox
```
Run Code

Python String Data Type

String is a sequence of characters represented by either single or double quotes. For example,

```
name = 'Python'
print(name)

message = 'Python for beginners'
print(message)
```

Run Code

Output

```
Python
Python for beginners
```

In the above example, we have created string-type variables: `name` and `message` with values `'Python'` and `'Python for beginners'` respectively.

Python Set Data Type

Set is an unordered collection of unique items. Set is defined by values separated by commas inside braces `{ }`. For example,

```
# create a set named student_id
student_id = {112, 114, 116, 118, 115}

# display student_id elements
print(student_id)

# display type of student_id
```

```
print(type(student_id))
```
Run Code

Output

```
{112, 114, 115, 116, 118}
<class 'set'>
```

Here, we have created a set named `student_info` with **5** integer values.

Since sets are unordered collections, indexing has no meaning. Hence, the slicing operator `[]` does not work.

Python Dictionary Data Type

Python dictionary is an ordered collection of items. It stores elements in key/value pairs.

Here, keys are unique identifiers that are associated with each value.

Let's see an example,

```
# create a dictionary named capital_city
capital_city = {'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

print(capital_city)
```
Run Code

Output

```
{'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}
```

In the above example, we have created a dictionary named `capital_city`.

Here,

1. **Keys** are `'Nepal'`, `'Italy'`, `'England'`

2. **Values** are `'Kathmandu'`, `'Rome'`, `'London'`

Access Dictionary Values Using Keys

We use `keys` to retrieve the respective `value`. But not the other way around.

For example,

```
# create a dictionary named capital_city
capital_city = {'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

print(capital_city['Nepal'])  # prints Kathmandu

print(capital_city['Kathmandu'])  # throws error message
```

Run Code

Here, we have accessed values using keys from the `capital_city` dictionary.

Since `'Nepal'` is key, `capital_city['Nepal']` accesses its respective value i.e. `Kathmandu`

However, `'Kathmandu'` is the value for the `'Nepal'` key, so `capital_city['Kathmandu']` throws an error message.

Python Type Conversion

In programming, type conversion is the process of converting data of one type to another. For example: converting `int` data to `str`.

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion
- Explicit Conversion - manual type conversion

Python Implicit Type Conversion

In certain situations, Python automatically converts one data type to another. This is known as implicit type conversion.

Example 1: Converting integer to float

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

```
integer_number = 123
float_number = 1.23

new_number = integer_number + float_number

# display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))
```
Run Code

Output

```
Value: 124.23
Data Type: <class 'float'>
```

In the above example, we have created two variables: `integer_number` and `float_number` of `int` and `float` type respectively.
Then we added these two variables and stored the result in `new_number`.
As we can see `new_number` has value **124.23** and is of the `float` data type.
It is because Python always converts smaller data types to larger data types to avoid the loss of data.

Note:

- We get `TypeError`, if we try to add `str` and `int`. For example, `'12' + 23`. Python is not able to use Implicit Conversion in such conditions.
- Python has a solution for these types of situations which is known as Explicit Conversion.

Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type.

We use the built-in functions like `int()`, `float()`, `str()`, etc to perform explicit type conversion.
This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Example 2: Addition of string and integer Using Explicit Conversion

```
num_string = '12'
num_integer = 23

print("Data type of num_string before Type Casting:",type(num_string))

# explicit type conversion
num_string = int(num_string)

print("Data type of num_string after Type Casting:",type(num_string))

num_sum = num_integer + num_string

print("Sum:",num_sum)
print("Data type of num_sum:",type(num_sum))
```

Run Code

Output

```
Data type of num_string before Type Casting: <class 'str'>
Data type of num_string after Type Casting: <class 'int'>
Sum: 35
Data type of num_sum: <class 'int'>
```

In the above example, we have created two variables: `num_string` and `num_integer` with `str` and `int` type values respectively. Notice the code,

```
num_string = int(num_string)
```

Here, we have used `int()` to perform explicit type conversion of `num_string` to integer type.

After converting `num_string` to an integer value, Python is able to add these two variables.

Finally, we got the `num_sum` value i.e **35** and data type to be `int`.

Key Points to Remember

1. Type Conversion is the conversion of an object from one data type to another data type.
2. Implicit Type Conversion is automatically performed by the Python interpreter.
3. Python avoids the loss of data in Implicit Type Conversion.
4. Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.
5. In Type Casting, loss of data may occur as we enforce the object to a specific data type.

Python abs()

returns absolute value of a number

Python all()

returns true when all elements in iterable is true

Python any()

Checks if any Element of an Iterable is True

Python ascii()

Returns String Containing Printable Representation

Python bin()

converts integer to binary string

Python bool()

Converts a Value to Boolean

Python bytearray()

returns array of given byte size

Python bytes()

returns immutable bytes object

Python callable()

Checks if the Object is Callable

Python chr()

Returns a Character (a string) from an Integer

Python classmethod()

returns class method for given function

Python compile()

Returns a Python code object

Python complex()

Creates a Complex Number

Python delattr()

Deletes Attribute From the Object

Python dict()

Creates a Dictionary

Python dir()

Tries to Return Attributes of Object

Python divmod()

Returns a Tuple of Quotient and Remainder

Python enumerate()

Returns an Enumerate Object

Python eval()

Runs Python Code Within Program

Python exec()

Executes Dynamically Created Program

Python filter()

constructs iterator from elements which are true

Python float()

returns floating point number from number, string

Python format()

returns formatted representation of a value

Python frozenset()

returns immutable frozenset object

Python getattr()

returns value of named attribute of an object

Python globals()

returns dictionary of current global symbol table

Python hasattr()

returns whether object has named attribute

Python hash()

returns hash value of an object

Python help()

Invokes the built-in Help System

Python hex()

Converts to Integer to Hexadecimal

Python id()

Returns Identify of an Object

Python input()

reads and returns a line of string

Python int()

returns integer from a number or string

Python isinstance()

Checks if a Object is an Instance of Class

Python issubclass()

Checks if a Class is Subclass of another Class

Python iter()

returns an iterator

Python len()

Returns Length of an Object

Python list()

creates a list in Python

Python locals()

Returns dictionary of a current local symbol table

Python map()

Applies Function and Returns a List

Python max()

returns the largest item

Python memoryview()

returns memory view of an argument

Python min()

returns the smallest value

Python next()

Retrieves next item from the iterator

Python object()

creates a featureless object

Python oct()

returns the octal representation of an integer

Python open()

Returns a file object

Python ord()

returns an integer of the Unicode character

Python pow()

returns the power of a number

Python print()

Prints the Given Object

Python property()

returns the property attribute

Python range()

returns a sequence of integers

Python repr()

returns a printable representation of the object

Python reversed()

returns the reversed iterator of a sequence

Python round()

rounds a number to specified decimals

Python set()

constructs and returns a set

Python setattr()

sets the value of an attribute of an object

Python slice()

returns a slice object

Python sorted()

returns a sorted list from the given iterable

Python staticmethod()

transforms a method into a static method

Python str()

returns the string version of the object

Python sum()

Adds items of an Iterable

Python super()

Returns a proxy object of the base class

Python tuple()

Returns a tuple

[Python type()]

Returns the type of the object

[Python vars()]

Returns the __dict__ attribute

[Python zip()]

Returns an iterator of tuples

[Python __import__()]

Function called by the import statement