

The data type of an item defines the operations that can be performed on it, and how the values are stored in memory. Python supports the following data types:

- Number
- String
- List
- Tuple
- Set
- Dictionary

4.5.1 Numbers

The number or numeric data type is used to store numeric values. There are three distinct numeric types:

<u>Type</u>	<u>Description</u>	<u>Examples</u>
int	integers	700,198005
float	numbers with decimal point	3.14,6.023
complex	complex numbers	3+4j, 10j

The integers include numbers that do not have decimal point. The `int` data type supports integers ranging from -2^{31} to $2^{31} - 1$.

Python uses `float` type to represent real numbers (with decimal points). The values of `float` type range approximately from -10^{308} to 10^{308} and have 16 digits of precision (number of digits after the decimal point). A floating-point number can be written using either ordinary decimal notation or scientific notation. Scientific notation is often useful for denoting numbers with very large or very small magnitudes. See the examples below:

<u>Decimal notation</u>	<u>Scientific notation</u>	<u>Meaning</u>
3.146	3.146e0	3.146×10^0
314.6	3.146e2	3.146×10^2
0.3146	3.146e-1	3.146×10^{-1}
0.003146	3.146e-3	3.146×10^{-3}

`complex` numbers are written in the form `x+yj`, where `x` is the real part and `y` is the imaginary part.

`int` type has a subtype `bool`. Any variable of type `bool` can take one of the two possible boolean values, `True` and `False` (internally represented as 1 and 0, respectively).

4.5.2 Strings

A string literal or a string is a sequence of characters enclosed in a pair of single quotes or double quotes. "Hi", "8.5", `hello` are all strings. Multi-line strings are written within a pair of triple quotes, ``` or """. The following is an example of a multi-line string:

```
""" this is a multiline  
string """
```

The strings ' ' and " " are called *empty strings*.

Statements

- A statement is an instruction that the Python interpreter can execute.
- A statement can be an expression statement or a control statement.
- An expression statement contains an arithmetic expression that the interpreter evaluates.
- A control statement is used to represent advanced features of a language like decision-making and looping.

Operators in Python

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

ARITHMETIC OPERATORS

- The remainder operator (%) requires both operands to be integers, and the second operand is non-zero.
- The division operator (/) requires that the second operand be non-zero.
- The floor division operator (//) returns the floor value of the quotient of a division operation.

```
>>> 7.0//2
3.0
>>> 7//2
3
```


 Floor division is also called integer division.

Table 5.1: Arithmetic operators

Operation	Operator
Negation	-
Addition	+
Subtraction	--
Multiplication	*
Division	/
Floor division	//
Remainder	%
Exponentiation	**

ASSIGNMENT OPERATOR

'=' is the assignment operator. The **assignment statement** creates new variables and gives them values that can be used in subsequent arithmetic expressions. See the example:

```
>>> a=10  
>>> b=5  
>>> a+b  
15
```

Python allows you to assign a single value to several variables simultaneously. An example follows:

```
>>> a=b=5
>>> a
5
>>> b
5
```

You can also assign different values to multiple variables. See below

```
>>> a,b,c=1,2.5,"ram"
>>> a
1
>>> b
2.5
>>> c
'ram'
```

Python supports the following six additional assignment operators (called *compound assignment* operators): `+=`, `-=`, `*=`, `/=`, `//=` and `%=`. These are described below:

Expression	Equivalent to
<code>a+=b</code>	<code>a=a+b</code>
<code>a-=b</code>	<code>a=a-b</code>
<code>a*=b</code>	<code>a=a*b</code>
<code>a/=b</code>	<code>a=a/b</code>
<code>a//=b</code>	<code>a=a//b</code>
<code>a%=b</code>	<code>a=a%b</code>

COMPARISON OPERATORS OR RELATIONAL OPERATORS

Table 5.2: Comparison operators

Operation	Operator
Equal to	<code>==</code>
Not equal to	<code>!=</code>
Greater than	<code>></code>
Greater than or equal to	<code>>=</code>
Less than	<code><</code>
Less than or equal to	<code><=</code>

```
>>> i,j,k=3,4,7
>>> i>j
False
>>> (j+k)>(i+5)
True
```

Comparison operators support chaining. For example, `x < y <= z` is equivalent to `x < y` and `y <= z`.

Table 5.3: Truth tables for logical OR and logical AND operators

a	b	a or b	a and b
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Table 5.4: Truth table for logical NOT

a	not a
False	True
True	False

In the context of logical operators, Python interprets all non-zero values as `True` and zero as `False`. See examples below:

```
>>> 7 and 1
1
>>> -2 or 0
-2
>>> -100 and 0
0
```

5.2.5.2 Logical bitwise operators

There are three logical bitwise operators: bitwise and ($\&$), bitwise exclusive or (\wedge), and bitwise or (\mid). Each of these operators require two integer-type operands. The operations are performed on each pair of corresponding bits of the operands based on the following rules:

- A **bitwise and** expression will return 1 if both the operand bits are 1. Otherwise, it will return 0.
- A **bitwise or** expression will return 1 if at least one of the operand bits is 1. Otherwise, it will return 0.
- A **bitwise exclusive or** expression will return 1 if the bits are not alike (one bit is 0 and the other is 1). Otherwise, it will return 0.

These results are summarized in Table 5.5. In this table, b_1 and b_2 represent the corresponding bits within the first and second operands, respectively.

Table 5.5: Logical bitwise operators

b_1	b_2	$b_1 \& b_2$	$b_1 \mid b_2$	$b_1 \wedge b_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```

>>> a=20
>>> b=108
>>> a&b
4
>>> a|b
124
>>> a^b
120

```

The following justifies these results.

$$\begin{array}{rcl}
 a & = & 0001 \ 0100 \\
 b & = & 0110 \ 1100 \\
 \hline
 a \ \& \ b & = & 0000 \ 0100 \\
 & = & 4
 \end{array}$$

$$\begin{array}{rcl}
 a & = & 0001 \ 0100 \\
 b & = & 0110 \ 1100 \\
 \hline
 a \ | \ b & = & 0111 \ 1100 \\
 & = & 124
 \end{array}$$

$$\begin{array}{rcl}
 a & = & 0001 \ 0100 \\
 b & = & 0110 \ 1100 \\
 \hline
 a \ ^ \ b & = & 0111 \ 1000 \\
 & = & 120
 \end{array}$$

5.2.5.3 Bitwise shift operators

The two bitwise shift operators are shift left (\ll) and shift right (\gg). The expression $x \ll n$ shifts each bit of the binary representation of x to the left, n times. Each time we shift the bits left, the vacant bit position at the right end is filled with a zero.

The expression $x \gg n$ shifts each bit of the binary representation of x to the right, n times. Each time we shift the bits right, the vacant bit position at the left end is filled with a zero.

Example 5.3. This example illustrates the bitwise shift operators as shown below:

```
>>> 120>>2
30
>>> 10<<3
80
```

Let us now see the operations in detail.

```
120 = 0111 1000
      _____
Right shifting once – 0011 1100
Right shifting twice – 0001 1110
      _____
120>>2 = 30
```

```
10 = 0000 1010
      _____
Left shifting once – 0001 0100
Left shifting twice – 0010 1000
Left shifting thrice – 0101 0000
      _____
10<<3 = 80
```

5.2.6 Membership Operators

These operators test for the membership of a data item in a sequence, such as a string. Two membership operators are used in Python.

- **in** – Evaluates to **True** if it finds the item in the specified sequence and **False** otherwise.
- **not in** – Evaluates to **True** if it does not find the item in the specified sequence and **False** otherwise.

See the examples below:

```
>>> 'A' in 'ASCII'
True
>>> 'a' in 'ASCII'
False
>>> 'a' not in 'ASCII'
True
```

5.2.7 Identity Operators

`is` and `is not` are the identity operators in Python. They are used to check if two values (or variables) are located in the same part of the memory. `x is y` evaluates to `true` if and only if `x` and `y` are the same object. `x is not y` yields the inverse truth value.

5.2.9 Mixed-Mode Arithmetic

Table 5.9: Type coercion rules

Operands type	Result type
int and int	int
float and float	float
int and float	float

Performing calculations involving operands of different data types is called *mixed-mode arithmetic*. Consider computing the area of a circle having 3 unit radius:

```
>>> 3.14 * 3 ** 2
28.26
```

Table 6.2: Escape sequences in Python

Escape Sequence	Meaning
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\\</code>	The <code>\</code> character
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark

Math module- - sqrt(), sin(), exp() etc

1. import the module
2. access the function or the constant by prefixing its name with "math."
(math followed by a dot)

See the examples below:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
```

```
2.718281828459045
>>> math.sqrt(3)
1.7320508075688772
```

Program 6.2. To input the user's name and print a greeting message.

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

Program 6.3. To input a number and display it.

```
num=int(input("Enter a number"))
print("The number you entered is",num)
```

Program 6.4. To add and subtract two input numbers.

```
a=int(input("Enter the first number"))
b=int(input("Enter the second number"))
sum=a+b
print("The sum of the two numbers is",sum)
difference=a-b
print("The difference between the two numbers is",difference)
```

Program 6.5. To input the sides of a rectangle and find its perimeter.

```
length=int(input("Enter the length of the rectangle"))
breadth=int(input("Enter the breadth of the rectangle"))
perimeter=2*(length+breadth)
print("Perimeter of the rectangle is",perimeter)
```

Program 6.6. To input the side of a square and find its area.

```
side=int(input("Enter the side of the square"))
area=side**2
print("Area of the square is",area)
```

Program 6.7. To input the radius of a circle and find its circumference.

```
import math
radius=int(input("Enter the radius"))
c=2*math.pi*radius
print("Circumference of the circle is",c)
```

Program 6.8. To input two values a and b and then find a^b .

```
import math
a=int(input("Enter the base"))
b=int(input("Enter the exponent"))
c=math.pow(a,b)
print(a,"to the power",b,"is",c)
```

Program 6.2. To input the user's name and print a greeting message.

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

Program 6.3. To input a number and display it.

```
num=int(input("Enter a number"))  
print("The number you entered is",num)
```

Program 6.4. To add and subtract two input numbers.

```
a=int(input("Enter the first number"))  
b=int(input("Enter the second number"))  
sum=a+b  
print("The sum of the two numbers is",sum)  
difference=a-b  
print("The difference between the two numbers is",difference)
```

Program 6.5. To input the sides of a rectangle and find its perimeter.

```
length=int(input("Enter the length of the rectangle"))  
breadth=int(input("Enter the breadth of the rectangle"))  
perimeter=2*(length+breadth)  
print("Perimeter of the rectangle is",perimeter)
```

Program 6.6. To input the side of a square and find its area.

```
side=int(input("Enter the side of the square"))  
area=side**2  
print("Area of the square is",area)
```


7.3 Selection statements

This section explores several types of selection statements that allow the computer to make choices.

7.3.1 One-way selection statement

First, we discuss the `if` statement, also known as *conditional execution*. The following example tests whether a variable `x` is positive or not.

```
>>> x=10
>>> if x>0:
...     print(x,"is positive")
...
10 is positive
```

It is possible to combine multiple conditions using logical operators. The following code snippet tests whether a number `x` is a single-digit number or not.

```
>>> x=int(input("Enter a number"))
Enter a number7
>>> if x > 0 and x < 10:
...     print(x,"is a positive single digit.")
...
7 is a positive single digit.
```

Python provides an alternative syntax for writing the condition in the above code that is similar to mathematical notation:

```
>>> x=int(input("Enter a number"))
Enter a number8
>>> if 0<x<10:
...     print(x,"is a positive single digit.")
...
8 is a positive single digit.
```

7.3.2 Two-way selection statement

The `if-else` statement, also known as *alternative execution*, is the most common type of selection statement in Python. See the following example to check if a person is major or minor.

```
>>> age=12
>>> if age>=18:
...     print("Major")
... else:
...     print("Minor")
...
Minor
```

7.3.3 Multi-way selection statement

Multi-way selection is achieved through the `if-elif-else` statement. `elif` is the abbreviation of “else if”. The following code compares two variables and prints the relation between them.

```
>>> x=10
>>> y=5
>>> if x < y:
...     print(x, "is less than", y)
... elif x > y:
...     print(x, "is greater than", y)
... else:
...     print(x, "and", y, "are equal")
...
10 is greater than 5
```

The multi-way `if` statement is called chained conditional execution.

7.4 Repetition statements or loops

Python supports two types of loops – those that repeat an action a fixed number of times (**definite iteration**) and those that act until a condition becomes false (**conditional iteration**).

7.4.1 Definite iteration: The **for** loop

We now examine Python's **for** loop, the control statement supporting definite iteration. We use **for** in association with the **range()** function that dictates the number of iterations. To be precise, **range(k)** when used with **for** causes the loop to iterate k times. See the example below that prints “Hello” 5 times.


```
>>> for i in range(5):  
...     print("Hello")  
...  
Hello  
Hello  
Hello  
Hello  
Hello
```

`range(k)` starts counting from 0, incrementing after each iteration, and stops on reaching $k - 1$. Thus to print numbers from 0 to 4, you write:

```
>>> for i in range(5):  
...     print(i)  
...  
0  
1  
2  
3  
4
```

You are reminded that `range(5)` counts from 0 to 4, not 5. To get the numbers on the same line you need to write `print(count,end=" ")`. By using `end = " "`, the Python interpreter will append whitespace following the count value, instead of the default newline character (`'\n'`) See below:

```
>>> for i in range(5):  
...     print(i,end=" ")  
...  
0 1 2 3 4
```

 Loops that count through a range of numbers are called count-controlled loops.

By default, the **for** loop starts counting from 0. To count from an explicit lower bound, you need to include it as part of the **range** function. See the example below that prints the first 10 natural numbers:

```
>>> for count in range(1,11):  
...     print(count,end=" ")  
...  
1 2 3 4 5 6 7 8 9 10
```

The **for** loops we have seen till now count through consecutive numbers in each iteration. Python provides a third variant of the **for** loop that allows to count in a non-consecutive fashion. For this, you need to explicitly mention the *step size* in the **range** function. The following code prints the even numbers between 4 and 20 (both inclusive).

```
>>> for i in range(4,21,2):  
...     print(i,end=" ")  
...  
4 6 8 10 12 14 16 18 20
```

When the step size is negative, the loop variable is decremented by that amount in each iteration. The following code displays numbers from 10 down to 1.


```
>>> for count in range(10,0,-1):  
...     print(count,end=" ")  
...  
10 9 8 7 6 5 4 3 2 1
```

Notice above that, to count from 10 down to 1, we write `for count in range(10,0,-1)` and not `for count in range(10,1,-1)`. Thus, to count from `a` down to `b`, we write `for count in range(a,b-1,s)` with $a > b$ and $s < 0$.

7.4.2 Conditional Iteration: The `while` loop

Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue. Python's `while` loop is tailor-made for this type of control logic. Here is the code to print the first 10 natural numbers, but this time with `while`.

```
>>> i=1
>>> while i<=10:
...     print(i,end=" ")
...     i=i+1
...
1 2 3 4 5 6 7 8 9 10
```

 There is no *exit-controlled loop* in Python, but you can modify the `while` loop to achieve the same functionality.

7.4.3 Nested loops

It is possible to have a loop inside another loop. We can have a **for** loop inside a **while** loop or inside another **for** loop. The same is possible for **while** loops too. The enclosing loop is called the **outer loop**, and the other loop is called the **inner loop**. The inner loop will be executed once for each iteration of the outer loop: Consider the following code:

```
>>> for i in range(1,5):    #This is the outer loop
...     for j in range(1,5): #This is the inner loop
...         print(j, end=" ")
...     print()
...
1 2 3 4
1 2 3 4
```

```
1 2 3 4
1 2 3 4
```

The loop variable `i` varies from 1 to 4 (not 5). For each value of `i`, the variable `j` varies from 1 to 4. The statement `print(j, end=" ")` prints the `j` values on a single line, and the statement `print()` takes the control to the next line after every iteration of the outer loop.

7.4.4 Loop control statements

Python provides three loop control statements that control the flow of execution in a loop. These are discussed below:

7.4.4.1 break statement

The **break** statement is used to terminate loops. Any statements inside the loop following the **break** will be neglected, and the control goes out of the loop. **break** stops the current iteration and skips the succeeding iterations (if any) and passes the control to the first statement following (outside) the loop. This is illustrated in the following code:

```
for num in range(1,5):  
    if num%2==0:  
        print(num,"is even")  
        break  
    print("The number is",num)  
print("Outside the loop")
```

This code produces the output:

```
The number is 1  
2 is even  
Outside the loop
```

7.4.4.2 continue statement

The `continue` statement is used to bypass the remainder of the current iteration through a loop. The loop does not terminate when a `continue` statement is encountered. Rather, the remaining loop statements are skipped and the control proceeds directly to the next pass through the loop. See the code below:

```
for num in range(1,5):  
    if num%2==0:  
        print(num,"is even")  
        continue  
    print("The number is",num)
```

```
print("Outside the loop")
```

The code above produces the output:

```
The number is 1  
2 is even  
The number is 3  
4 is even  
Outside the loop
```

7.4.4.3 pass statement

The `pass` statement is equivalent to the statement “Do nothing”. It can be used when a statement is required syntactically but the program requires no action. Nothing happens when `pass` is executed. It results in a NOP (No OPeration). After the `pass` is executed, control proceeds as usual to the next statement in the loop. The code below illustrates this:

```
for num in range(1,5):  
    if num%2==0:  
        print(num,"is even")  
        pass  
    print("The number is",num)  
print("Outside the loop")
```

This code produces the output:

```
The number is 1  
2 is even  
The number is 2  
The number is 3  
4 is even  
The number is 4  
Outside the loop
```

`pass` statement is useful when you want to insert empty code (empty lines) in a program, where real code statements can be added later. Empty code is not allowed in loops; if included, Python will raise errors. In such situations, `pass` statement acts as a temporary placeholder.

