



Programming lab



Reference: Algorithmic Thinking with python

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	**

'=' 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
```


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.

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

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

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

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.

This example illustrates the bitwise shift operators as shown below:

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

Let us now see the operations in detail.

$$120 = \underline{0111\ 1000}$$

Right shifting once – $\underline{0011\ 1100}$

Right shifting twice – $\underline{0001\ 1110}$

$$120>>2 = 30$$

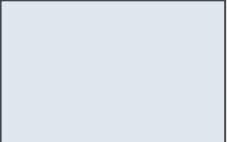
$$10 = \underline{0000\ 1010}$$

Left shifting once – $\underline{0001\ 0100}$

Left shifting twice – $\underline{0010\ 1000}$

Left shifting thrice – $\underline{0101\ 0000}$

$$10<<3 = 80$$



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
```




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.

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)
```

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
```



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.

functions

A function should be “defined” before its first use. To define a function, you specify its name and then write the logic for implementing its functionality. A function definition looks like this:

```
def function-name(parameter-list):  
    statement-1  
    statement-2  
  
    .  
    .  
    statement-n  
    return-statement
```

- Function definition begins with the keyword `def` followed by the function name. Then you have the parameter list – a comma-separated list of arguments enclosed in a pair of parentheses. This line of the function definition is called **header**.
- The block of statements that constitute the function logic (`statement-1` to `return-statement`) is called **function body**.
- The final statement `return` exits a function.
- The header has to end with a colon, and the body has to be indented.
- The parameters and the return statement are optional, i.e., you could write a function without these.

Defining a function is not the end of the game. To use its functionality, you need to *call* it. The function will not get executed unless it is called. You call a function by its name, passing (supplying) the value for each parameter separated by commas and the entire list enclosed in parentheses. If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function. The following example illustrates how a Python function is defined and called:

```
>>> def printName(): #This is how you define a function
...     name=input("Enter your name")
...     print("Hi",name,"! Welcome to the world of functions!")
...
>>> printName() #This is how you call a function
Enter your nameSam
Hi Sam ! Welcome to the world of functions!
```


how does a function-based program work?

- ❑ The way the statements in a program get executed is called flow of execution or control flow.
- ❑ Execution always begins at the first statement of the program.
- ❑ Statements are executed one at a time, in order from top to bottom.
- ❑ Function definitions do not alter the flow of execution – statements inside the function body are not executed until the function is called.
- ❑ A function call changes the flow of execution.
- ❑ When a function call is encountered, instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.
- ❑ When the end of the program is reached, the execution terminates.

8.4.1 Function with no arguments and no return value

```
>>> def sum():  
...     a=int(input("Enter an integer"))  
...     b=int(input("Enter another integer"))  
...     s=a+b  
...     print("The sum of the entered integers is",s)  
...  
>>> sum()  
Enter an integer6  
Enter another integer8  
The sum of the entered integers is 14
```

8.4.2 Function with arguments but no return value

```
>>> def sum(c,d):  
...     s=c+d  
...     print("The sum of the entered integers is",s)  
...  
>>> a=int(input("Enter an integer"))  
Enter an integer3  
>>> b=int(input("Enter another integer"))  
Enter another integer5  
>>> sum(a,b)  
The sum of the entered integers is 8
```

8.4.3 Function with return value but no arguments

```
>>> def sum():  
  
...     a=int(input("Enter an integer"))  
...     b=int(input("Enter another integer"))  
...     s=a+b  
...     return s  
...  
>>> print("The sum of the entered integers is",sum())  
Enter an integer4  
Enter another integer7  
The sum of the entered integers is 11
```

8.4.4 Function with arguments and return value

```
>>> def sum(c,d):  
...     s=c+d  
...     return s  
...  
>>> a=int(input("Enter an integer"))  
Enter an integer10  
>>> b=int(input("Enter another integer"))  
Enter another integer15  
>>> print("The sum of the entered integers is",sum(a,b))  
The sum of the entered integers is 25
```


8.5 Algorithm and flowchart for modules

How do you write pseudocode for a problem solution with multiple modules? For this, you just list out the pseudocode for all the modules. See Figure 8.1 for an example that inputs two numbers, adds them, and prints the sum:

ADDINTEGERS

```
1  READ( $a, b$ )  
2  SUM( $a, b$ )
```

SUM(a, b)

```
1   $s = a + b$   
2  PRINT( $s$ )
```

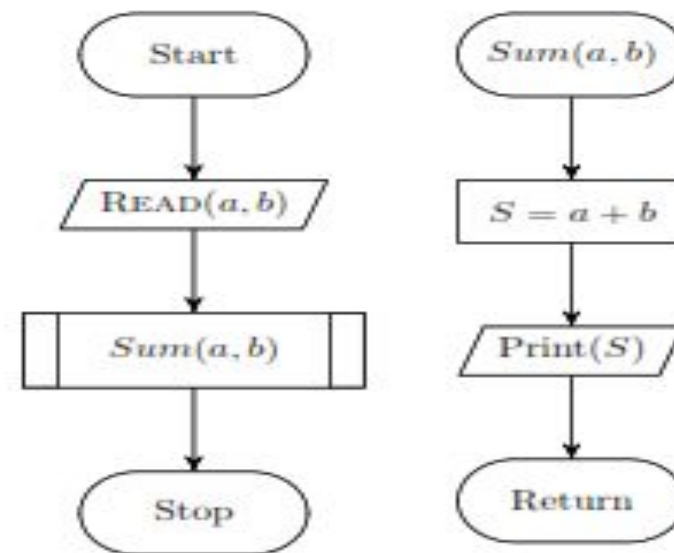


Figure 8.1: Adding two numbers using functions

Variable scope and parameter passing

- The region of the program text where a variable's value can be accessed is
- called its scope.
- Thus, the scope of a variable defines the active regions of the variable.
- The variables defined inside a function are said to have local scope.
- This means if you try to access the value of a variable outside the function where it is defined, you are bound to get an error:

```
>>> def defineVar():  
...     var=7  
...  
>>> defineVar()  
>>> print(var)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'var' is not defined.
```

- ❖ Function calls obey scope rules.
- ❖ When a function is called, the values of the actual parameters are copied to the formal parameters.
- ❖ The changes made, if any, to the formal parameters, are not reflected in the calling function.
- ❖ The changes are made only to the local copy of the formal parameters.

```
>>> def updateVar(var):  
...     var+=9  
...  
>>> var=7  
>>> print("The original variable value is ",var)  
The original variable value is 7  
>>> updateVar(var)  
>>> print("The modified variable value is ",var)  
The modified variable value is 7
```

Keyword arguments and default arguments

- The arguments for a function can be specified in two ways:
- 1. Positional arguments
- 2. Named arguments (aka keyword arguments)
- Positional arguments are the default way in which arguments are supplied to
- a function.
- Here the values are passed to arguments in the same order in which
- they occur in the function definition. When the positions of the arguments are
- changed, the values assigned also change.
- See an example:


```
>>> def myfun(a,b):  
...     print("The values of a and b are respectively",a,b)  
...  
>>> myfun(3,4)  
The values of a and b are respectively 3 4  
>>> myfun(4,3)  
The values of a and b are respectively 4 3
```

-
- In order not to bother with the order of the parameters, Python has the mechanism of named arguments or keyword arguments.
 - This allows you to specify the argument names in the function call along with their values.
 - Since the argument names are specified, the order of the arguments doesn't matter.
 - This is illustrated below:

```
>>> def myfun(a,b):  
...     print("The values of a and b are respectively",a,b)  
...  
>>> myfun(a=3,b=4)  
The values of a and b are respectively 3 4  
>>> myfun(b=4,a=3)  
The values of a and b are respectively 3 4
```

Default arguments or optional arguments

- allow you to assign default values to named arguments.
- The default values for the arguments are included while defining the function.
- When the function is called without passing values for the default arguments, their default values will be taken.
- On the other hand, if the function call includes values for default arguments as well, the passed values will override the default values.

```
>>> def sample(a,b=7):  
...     print("The values of a and b are respectively",a,b)  
...  
>>> sample(3)  
The values of a and b are respectively 3 7  
>>> sample(3,4)  
The values of a and b are respectively 3 4
```

You can pass values to default arguments in two ways:

- ❑ 1. by position: Here the values are passed in the order in which the arguments occur in the function header.
- ❑ 2. by name: Here the values are passed using the argument names in the
- ❑ function call.

```
>>> def sample(a,b=7,c=9):  
...     print("The values of a, b and c are respectively",a,b,c)  
...  
>>> sample(3)  
The values of a, b and c are respectively 3 7 9  
>>> sample(3,4) # default values taken by position  
The values of a, b and c are respectively 3 4 9  
>>> sample(3,c=5) # default values taken by name  
The values of a, b and c are respectively 3 7 5
```


A function can have both default and mandatory (non-default) arguments.

- In such cases, the parameter list should start with the mandatory arguments,
- then followed by the default arguments.
- Otherwise, you will get an error as shown below:

```
>>> def sample(a=7,b):  
    File "<stdin>", line 1  
        def sample(a=7,b):  
            ^  
SyntaxError: non-default argument follows default argument
```

Strings

- A string is a sequence of zero or more characters.
- Each character in the string occupies one byte of memory, and the last character is always '\0' which is called the null character.
- The terminating null character is important because it is the way by which, the Python interpreter knows where the string ends.

- Figure 9.1 shows how a string “Hello World” is stored in memory.
- Notice the null character at the end. The figure also shows how the character positions are numbered starting with 0 from the left end or starting with -1 from the right end.
- Each character position is known as an index.
- The null character position is not numbered.
- It is not considered a part of the string; its sole purpose is to know the string boundary.

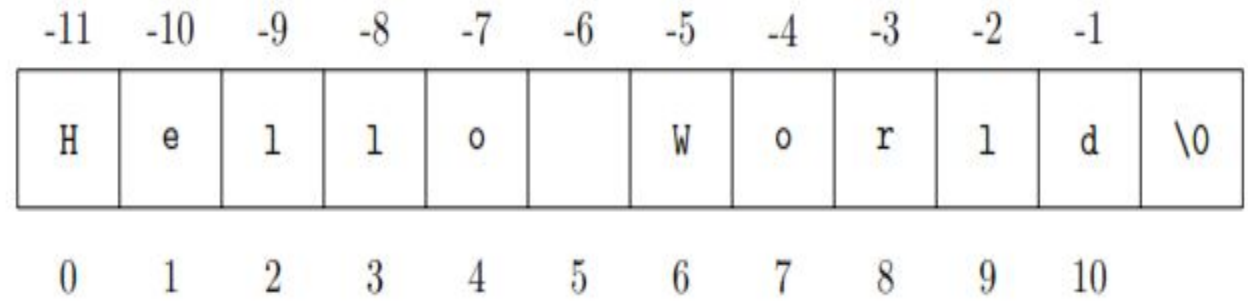


Figure 9.1: A string stored in memory

To know the length of a string, you use the `len()` method as shown below:

```
>>> str="Hello World"
>>> print(len(str))
11
```

As expected, the null character is not counted while finding the string length.

9.3 The subscript operator

To print a single character out of the string, you should use the **subscript operator**. The subscript operator (also called bracket operator) is denoted by `[]`. You just want to mention the index whose character you want to print. Here are some illustrations:

```
>>> str="Algorithmic thinking"
>>> print(str[0])
A
>>> print(str[5])
i
>>> print(str[11])

>>> print(str[18])
n
>>> print(str[20])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> print(str[19])
g
```

In the example above, the string length is 20; this means that the last index is 19. Thus, if you try to print the character in the 20th position, you get an error.

Python also allows negative subscript values. In this case, you need to count backward from -1 to access characters from the right end of the string. See below:

-
- Python also allows negative subscript values.
 - In this case, you need to count backward from -1 to access characters from the right end of the string.

```
>>> str="Try out!"
>>> print(str[-1])
!
>>> print(str[-3])
u
>>> print(str[-9])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

It is invalid to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. See the example below:

```
>>> pet="cat"
>>> pet[0]='r'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

The “object” mentioned in the error is the string and the “item” is the character we tried to assign. The reason for the error is that strings are **immutable**, which means you can’t change an existing string.

9.4 Traversing a string

Accessing the string characters one at a time, starting from the beginning and moving towards the end, is called **traversal**. The **for** loop comes in handy for this:

```
>>> str="Hi there"
>>> for c in str:
...     print(c)
...
H
i

t
h
e
r
e
```

9.5 Concatenation and repetition operators

The `+` operator performs **concatenation**, which means joining the strings by linking them end-to-end. See the example:

```
>>> str1="Python"  
>>> str2="Program"  
>>> print(str1+str2)  
PythonProgram
```

Note that the concatenation operator does not insert any white space between the joined strings.

The `*` operator performs repetition. It repeats a given string some fixed number of times. The left operand for `*` is the string to be repeated, and the right operand is an integer denoting the number of times the string is to be repeated. See the code:

```
>>> str="Order!"  
>>> print(str*3)  
Order!Order!Order!
```


9.6 String slicing

To extract a substring out of a given string, you should use subscript operator `[:]`. The operator `[n:m]` returns the part of the string starting at position `n` and ending at position `m-1`. See the example below:

```
>>> languages="Python, Java and C++"  
>>> print(languages[0:6])  
Python  
>>> print(languages[8:12])  
Java  
>>> print(languages[17:20])  
C++
```

If $n \geq m$, the operator `[n:m]` results in an empty string. See the code:

```
>>> languages="Python, Java and C++"  
>>> print(languages[4:4])  
  
>>>
```

The indices `n` and `m` in the operator `[n:m]` are optional. If you omit the first index `n`, the slice starts at the beginning of the string. If you omit the second index `m`, the slice goes to the end of the string. An example follows:

```
>>> languages="Python, Java and C++"  
>>> print(languages[:6])  
Python  
>>> print(languages[17:])  
C++
```

When `m` is omitted, supplying a negative value to the index `n` prints the last few characters of the string. To be specific, `str[-k:]` prints the last `k` characters of `str`. See the example below:

```
>>> languages="Python, Java and C++"  
>>> print(languages[-3:])  
C++
```

Omitting both indices `n` and `m` prints the full string. See below:

```
>>> languages="Python, Java and C++"  
>>> print(languages[:])  
Python, Java and C++
```

In addition to mentioning start and end indices in the slicing operator, you can also specify the step that denotes the distance between the characters you want to include in the resultant string. Suppose you want to print every alternate character in a string `str`, you just use the operator `::2`. See an example:

```
>>> languages="Python, Java and C++"  
>>> print(languages[::-2])  
Pto,Jv n +
```

Since the start and end indices are omitted, the slice starts from the beginning of the string and goes to its end. Since the step is mentioned as 2, every alternate character will be selected.

9.7 Comparison operators on strings

The comparison operators work on strings as well. To see if two strings are equal, we simply use the equality operator:

```
>>> str1="python"  
>>> str2="pythen"  
>>> print(str1==str2)  
False  
>>> str2="python"  
>>> print(str1==str2)
```

```
True
```

Other comparison operations are useful for determining the lexicographic (alphabetical) order of strings.

- The comparison operator when used with strings compares the ASCII values of the corresponding characters in the strings.
- Since the ASCII values are assigned in increasing order, a character that has a lower ASCII value precedes (comes before alphabetically) another character with a higher ASCII value.

```
>>> str1="C"
>>> str2="Java"
>>> if str1 < str2:
...     print(str1,"comes before",str2)
... elif str2 < str1:
...     print(str2,"comes before",str1)
... else:
...     print("The two strings are the same")
...
C comes before Java
```

9.8 The in operator

`in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second and `False` otherwise. See the example below:

```
>>> str1="C"
>>> str2="C++"
>>> str3="Python"
>>> print(str1 in str2)
True
>>> print(str1 in str3)
False
>>> print(str2 in str1)
False
```

The `in` operator can be used to print the characters that are present in (common to) two given strings as follows:

```
>>> str1="algorithms"
>>> str2="problem solving"
>>> for c in str1:
...     if c in str2:
...         print(c,end=" ")

...
l g o r i m s
```

9.9 The `string` module

The `string` module contains **string methods** that can be used to manipulate strings. Table 9.1 lists some of the string methods. To use them, you have to first import the module:

```
import string
```

Table 9.1: Some useful string methods, **s** represents to a string

String method	Purpose
<code>s.count(sub, start, end)</code>	Returns the number of non-overlapping occurrences of substring sub in the portion of s starting from start and ending at end-1 .
<code>s.find(sub, start, end)</code>	Returns the index of the first occurrence of the substring sub in the portion of s starting from start and ending at end-1 .
<code>s.replace(old, new, count)</code>	Returns a copy of s with all occurrences of substring old replaced by new . If the optional argument count is given, only the first count occurrences are replaced.
<code>s.isalpha()</code>	Returns True if s contains only letters or False otherwise.
<code>s.isdigit()</code>	Returns True if s contains only digits or False otherwise.
<code>s.lower()</code>	Returns a copy of s converted to lower-case.
<code>s.upper()</code>	Returns a copy of s converted to upper-case.

☞ The parameters `start` and `end` are optional in the methods `find()` and `count()`. If not present, the entire string is considered.

☞ `find()` returns `-1` if the substring to be searched is not present in the given string.

The following example illustrates the use of some of the string methods.

```
>>> import string
>>> s="malayalam"
>>> str1="string"
>>> str2="STR2"
>>> print(s.find("al"))
1
>>> print(s.find("al",2))
5
>>> print(s.find("al",2,5))
-1
>>> print(s.count("al"))
2
>>> print(s.count("al",2))
1
>>> print(s.count("al",2,5))
0
>>> print(s.replace("la","pq"))
mapqyapqm
>>> print(s.replace("la","pq",1))
mapqyalam
>>> print(str1.isalpha())
True
>>> print(str2.isalpha())
False
>>> print(str1.upper())
STRING
>>> print(str2.lower())
str2
```


In addition to string methods, the `string` module also provides several string constants; some of which are illustrated below:

```
>>> import string
>>> print(string.ascii_lowercase)
abcdefghijklmnopqrstuvwxyz
>>> print(string.ascii_uppercase)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
>>> print(string.ascii_letters)
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
>>> print(string.digits)
0123456789
```

You can use these constants to test whether a character is upper-case or lower-case, or whether it is a digit. See below:

```
>>> import string
>>> ch='p'
>>> print(ch in string.ascii_lowercase)
True
```

Program 9.46. To count the occurrences of a character in a string without using the count method.

```
str=input("Enter a string")
ch=input("Enter the character to search for")
count=0
for char in str:
    if char==ch:
        count+=1
print(ch,"occurs",count,"times in",str)
```

Program 9.52. To convert a binary number to decimal.

```
bstring = input("Enter a binary number")
decimal = 0
exponent = len(bstring) - 1
for digit in bstring:
    decimal = decimal + int(digit)*2**exponent
    exponent = exponent - 1
print("The decimal equivalent of",bstring,"is", decimal)
```

Lists

- ★ A list is an ordered set of data values.
- ★ The values that make up a list are called its elements or items.
- ★ The logical structure of a list is similar to that of a string.
- ★ Each item in the list has a unique index that specifies its position and the items are ordered by their positions.
- ★ Unlike strings, where each element is a character, the items in a list can be of different data types.



creating lists

To create a new list; you simply enclose the elements in a pair of square brackets ([]). Following are some examples:

```
>>> newlist=[10,"hi",3.14]
>>> print(newlist)
[10, 'hi', 3.14]
>>> mynest=[2,4,[1,3]]
>>> print(mynest)
[2, 4, [1, 3]]
```

As in `mynest` above, a list can have another list as its element. Such a list is called **nested list**. The list `[1,3]` which is an element of `mynest` is called a *member list*. A list that contains no elements is called an *empty list*, denoted as `[]`.

Lists of integers can also be built using the `range()` and `list()` functions.

```
>>> digits=list(range(10))
>>> print(digits)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> naturals=list(range(1,11))
>>> print(naturals)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> mul5=list(range(10,50,5))
>>> print(mul5)
[10, 15, 20, 25, 30, 35, 40, 45]
```

len() :determine its length (number of elements in the list).

- ★ In nested list, the member list will be counted as a single element.

```
>>> languages=["C","Java","Python","C++"]
>>> print(len(languages))
4
>>> breakfast=["dosa","chapathi",["bread","butter","jam"]]
>>> print(len(breakfast))
3
```

Accessing list elements and traversal :subscript operator

```
>>> newnest=[10, 20, [25, 30], 40]
>>> print(newnest[0])
10
>>> print(newnest[2])
[25, 30]
>>> print(newnest[-1])
40
>>> print(newnest[4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print(newnest[5-4])
20
>>> print(newnest[1.0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
>>> print(newnest[-5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```
