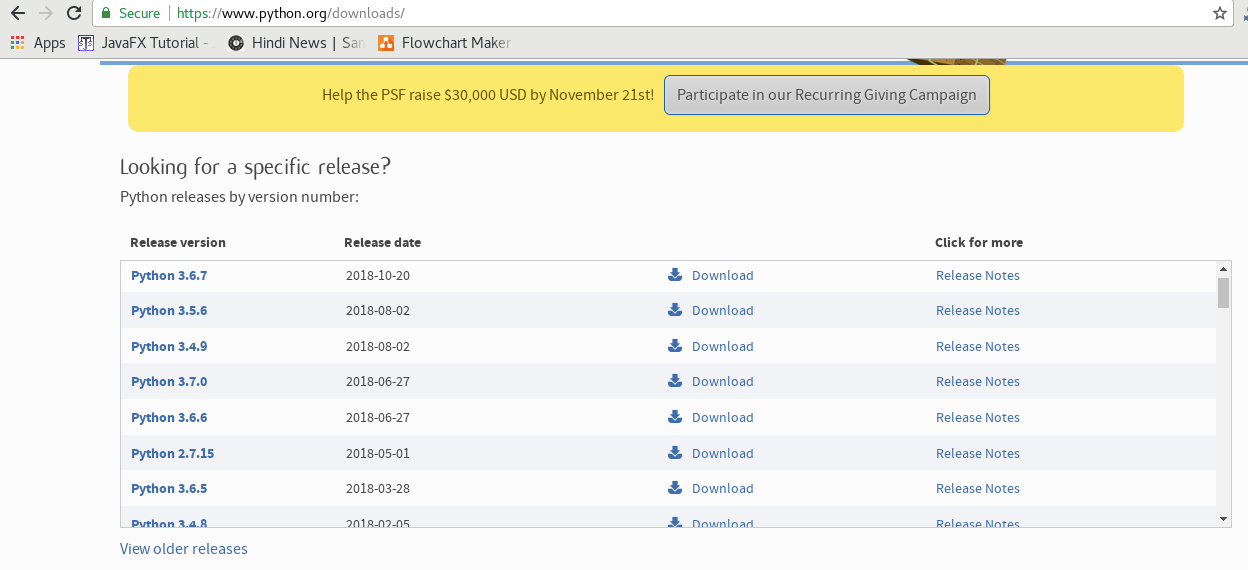
How to Install Python (Environment Set-up)

Installation on Windows

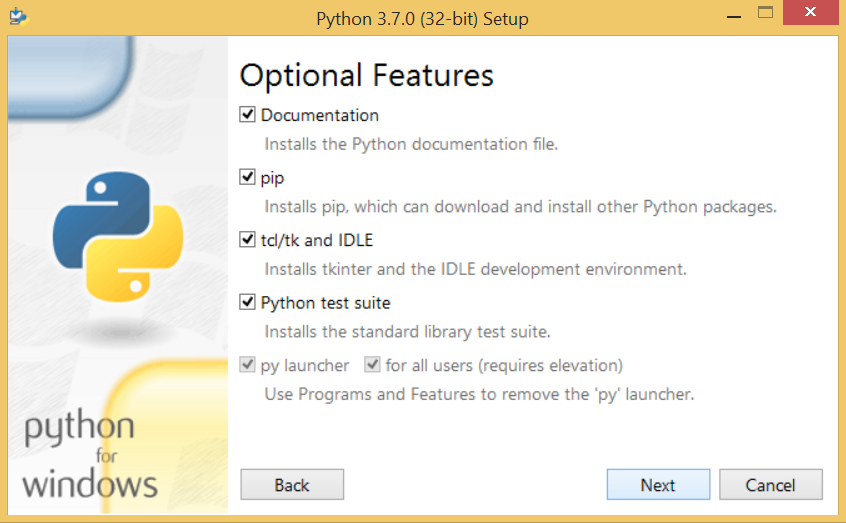
Visit the link [*https://www.python.org/downloads/*](https://www.python.org/downloads/) to download the latest release of Python. In this process, we will install Python 3.6.7 on our Windows operating system.



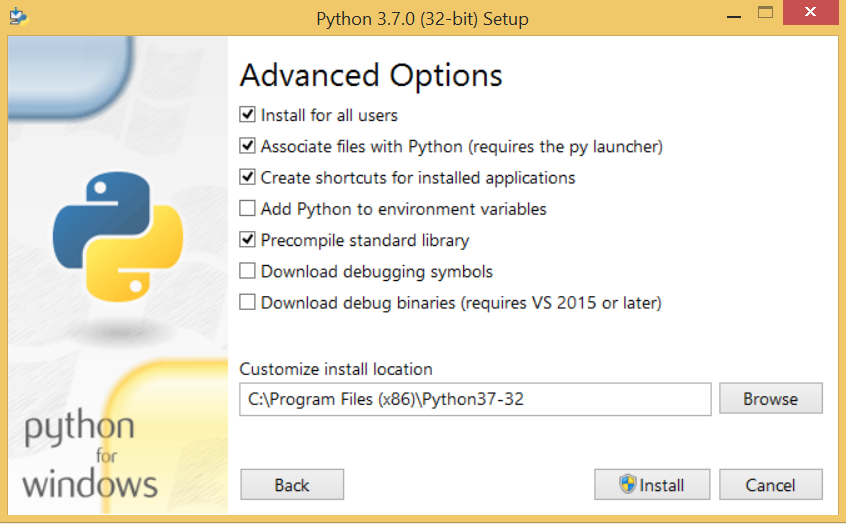
Double-click the executable file which is downloaded; the following window will open. Select Customize installation and proceed.

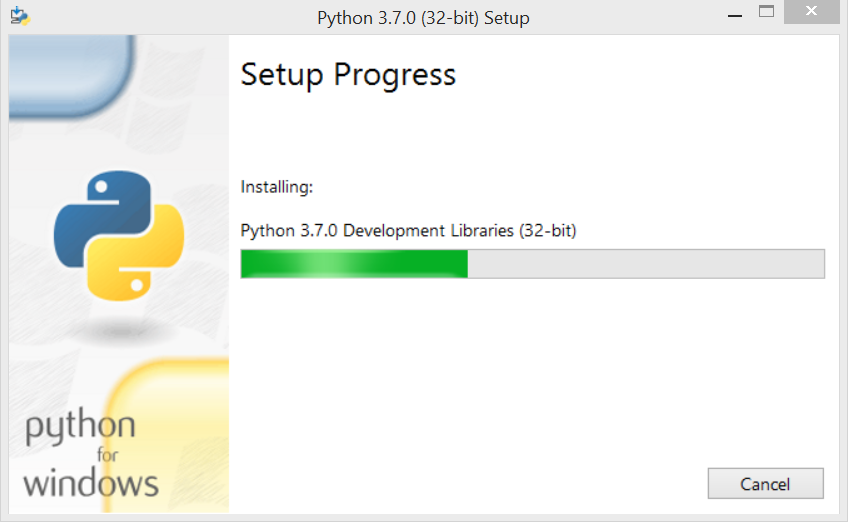
The following window shows all the optional features. All the features need to be installed and are checked by default; we need to click next to continue.

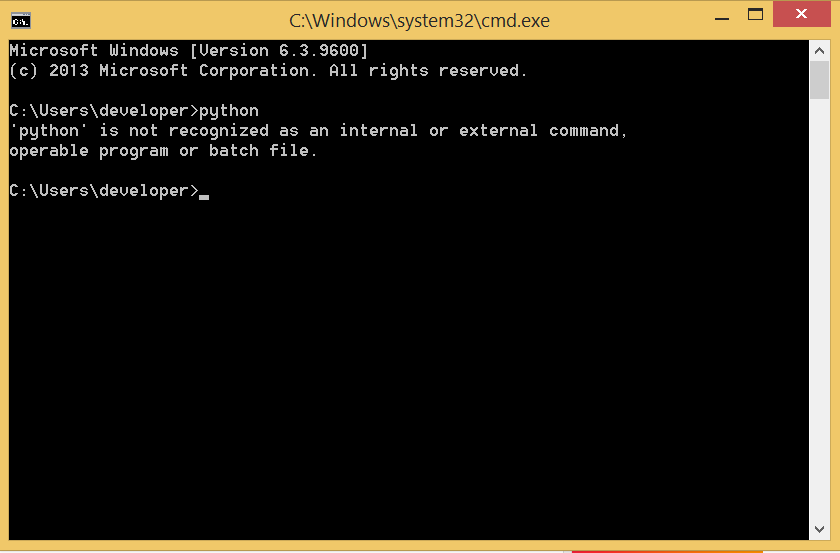
The following window shows a list of advanced options. Check all the options which you want to install and click next. Here, we must notice that the first check-box (install for all users) must be checked.

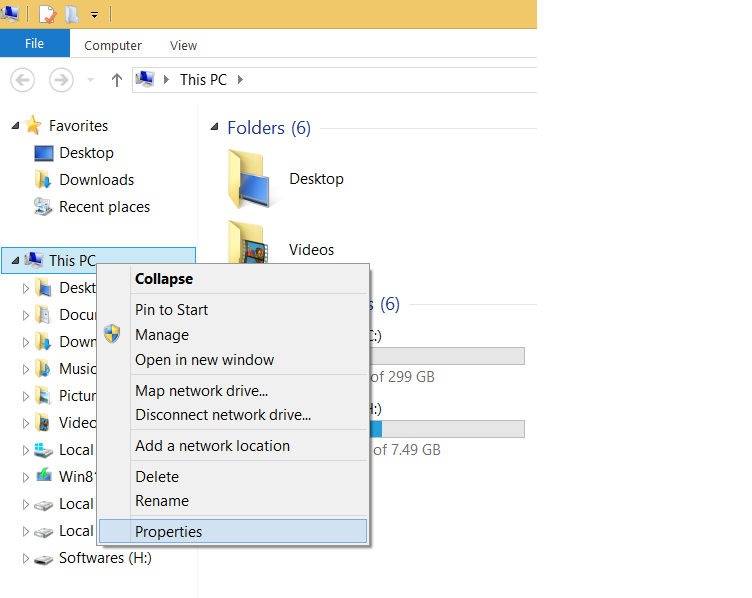
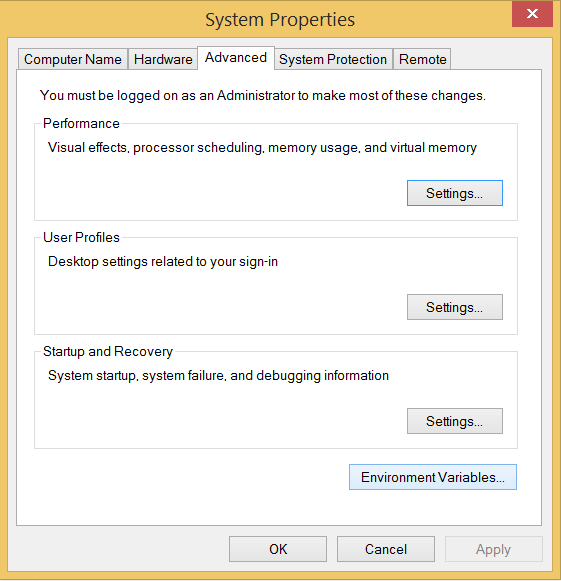
Now, we are ready to install python-3.6.7. Let's install it.

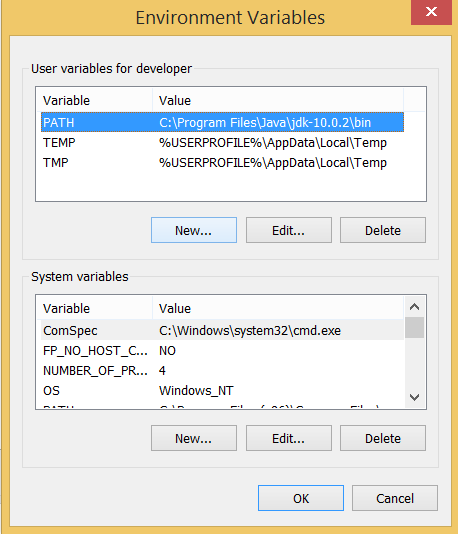
Now, try to run python on the command prompt. Type the command **python** in case of python2 or python3 in case of **python3**. It will show an error as given in the below image. It is because we haven't set the path.

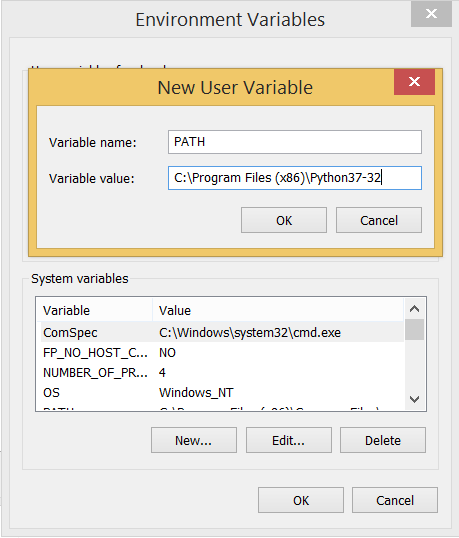
To set the path of python, we need to the right click on "my computer" and go to Properties → Advanced → Environment Variables.

Add the new path variable in the user variable section.

Type **PATH** as the variable name and set the path to the installation directory of the python shown in the below image.

Now, the path is set, we are ready to run python on our local system. Restart CMD, and type **python** again. It will open the python interpreter shell where we can execute the python statements.

# First Python Program

Python provides us the two ways to run a program:

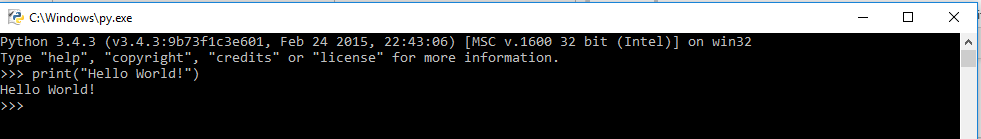
* Using Interactive interpreter prompt
* Using a script file

## Interactive interpreter prompt

Python provides us the feature to execute the python statement one by one at the interactive prompt. It is preferable in the case where we are concerned about the output of each line of our python program.

To open the interactive mode, open the terminal (or command prompt) and type python (python3 in case if you have python2 and python3 both installed on your system).

Let's run a python statement to print the traditional hello world on the console. Python3 provides print() function to print some message on the console. We can pass the message as a string into this function. Consider the following image.



Here, we get the message **"Hello World !"** printed on the console.

## Using a script file

Interpreter prompt is good to run the individual statements of the code. However, we can not write the code every-time on the terminal.

We need to write our code into a file which can be executed later. For this purpose, open an editor like notepad, create a file named first.py (python used .py extension) and write the following code in it.

Print ("hello world"); #here, we have used print() function to print the message on the console.

To run this file named as first.py, we need to run the following command on the terminal.

**$ python first.py**

## PyCharm

JetBrains provides the most popular and a widely used cross-platform IDE **PyCharm** to run the python programs.

## PyCharm installation

As we have already stated, PyCharm is a cross-platform IDE, and hence it can be installed on a variety of the operating systems.

### **Windows**

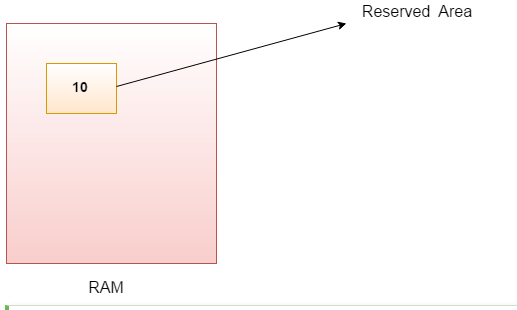
Installing PyCharm on Windows is very simple. To install PyCharm on Windows operating system, visit the link <https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows> to download the executable installer. **Double click** the installer (.exe) file and install PyCharm by clicking next at each step.

# Python Variables

Variable is a name which is used to refer memory location.

**Variable** is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

Variable also known as identifier and used to hold value.



Identifier Naming

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

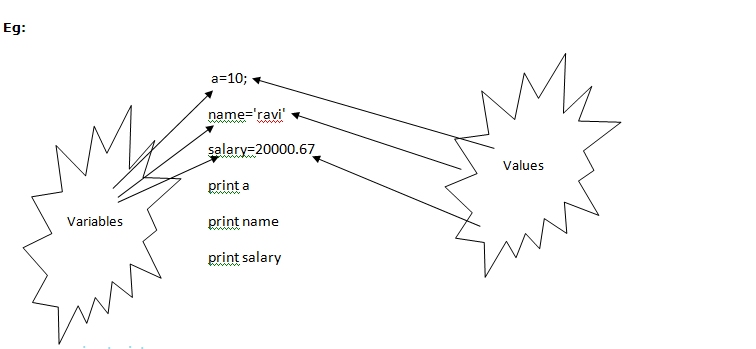
* The first character of the variable must be an alphabet or underscore ( \_ ).
* All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore or digit (0-9).
* Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, \*).
* Identifier name must not be similar to any keyword defined in the language.
* Identifier names are case sensitive for example my name, and MyName is not the same.
* Examples of valid identifiers : a123, \_n, n\_9, etc.
* Examples of invalid identifiers: 1a, n%4, n 9, etc.

## Declaring Variable and Assigning Values

Python does not bound us to declare variable before using in the application. It allows us to create variable at required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.



Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment.

We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Lets see given examples.

**1. Assigning single value to multiple variables**

**Eg:**

x=y=z=50

**print** x

**print** y

**print** z

**Output:**

>>>

50

50

50

>>>

**2.Assigning multiple values to multiple variables:**

**Eg:**

a,b,c=5,10,15

**print** a

**print** b

**print** c

**Output:**

>>>

5

10

15

>>>

The values will be assigned in the order in which variables appears.

**Tokens:**

* Tokens can be defined as a punctuator mark, reserved words and each individual word in a statement.
* Token is the smallest unit inside the given program.

There are following tokens in Python:

* Keywords.
* Identifiers.
* Literals.
* Operators.

Python Data Types

Variables can hold values of different data types. Python is a dynamically typed language hence we need not define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

a=10

b="Hi Python"

c = 10.5

**print**(type(a));

**print**(type(b));

**print**(type(c));

**Output:**

<type 'int'>

<type 'str'>

<type 'float'>

## Standard data types

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. [Numbers](https://www.javatpoint.com/python-data-types#numbers)
2. [String](https://www.javatpoint.com/python-data-types#string)
3. [List](https://www.javatpoint.com/python-data-types#list)
4. [Tuple](https://www.javatpoint.com/python-data-types#tuple)
5. [Dictionary](https://www.javatpoint.com/python-data-types#dictionary)

In this section of the tutorial, we will give a brief introduction of the above data types. We will discuss each one of them in detail later in this tutorial.

### **Numbers**

Number stores numeric values. Python creates Number objects when a number is assigned to a variable. For example;

1. a = 3 , b = 5  #a and b are number objects

Python supports 4 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)
3. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)
4. complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)

Python allows us to use a lower-case L to be used with long integers. However, we must always use an upper-case L to avoid confusion.

A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts respectively).

### **String**

The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.

String handling in python is a straightforward task since there are various inbuilt functions and operators provided.

In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+" python" returns "hello python".

The operator \* is known as repetition operator as the operation "Python " \*2 returns "Python Python ".

The following example illustrates the string handling in python.

1. str1 = 'hello javatpoint' #string str1
2. str2 = ' how are you' #string str2
3. **print** (str1[0:2]) #printing first two character using slice operator
4. **print** (str1[4]) #printing 4th character of the string
5. **print** (str1\*2) #printing the string twice
6. **print** (str1 + str2) #printing the concatenation of str1 and str2

**Output:**

he

o

hello javatpointhello javatpoint

hello javatpoint how are you

### **List**

Lists are similar to arrays in C. However; the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (\*) works with the list in the same way as they were working with the strings.

Consider the following example.

1. l  = [1, "hi", "python", 2]
2. **print** (l[3:]);
3. **print** (l[0:2]);
4. **print** (l);
5. **print** (l + l);
6. **print** (l \* 3);

**Output:**

[2]

[1, 'hi']

[1, 'hi', 'python', 2]

[1, 'hi', 'python', 2, 1, 'hi', 'python', 2]

[1, 'hi', 'python', 2, 1, 'hi', 'python', 2, 1, 'hi', 'python', 2]

### **Tuple**

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

1. t  = ("hi", "python", 2)
2. **print** (t[1:]);
3. **print** (t[0:1]);
4. **print** (t);
5. **print** (t + t);
6. **print** (t \* 3);
7. **print** (type(t))
8. t[2] = "hi";

**Output:**

('python', 2)

('hi',)

('hi', 'python', 2)

('hi', 'python', 2, 'hi', 'python', 2)

('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)

<type 'tuple'>

Traceback (most recent call last):

File "main.py", line 8, in <module>

t[2] = "hi";

TypeError: 'tuple' object does not support item assignment

### **Dictionary**

Dictionary is an ordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma and enclosed in the curly braces {}.

Consider the following example.

1. d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'};
2. **print**("1st name is "+d[1]);
3. **print**("2nd name is "+ d[4]);
4. **print** (d);
5. **print** (d.keys());
6. **print** (d.values());

**Output:**

1st name is Jimmy

2nd name is mike

{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}

[1, 2, 3, 4]

['Jimmy', 'Alex', 'john', 'mike']

# Python Keywords

Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter. Each keyword have a special meaning and a specific operation. These keywords can't be used as variable. Following is the List of Python Keywords.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| True | False | None | and | as |
| asset | def | class | continue | break |
| else | finally | elif | del | except |
| global | for | if | from | import |
| raise | try | or | return | pass |
| nonlocal | in | not | is | lambda |

Python Operators

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a particular programming language. Python provides a variety of operators described as follows.

* Arithmetic operators
* Comparison operators
* Assignment Operators
* Logical Operators
* Bitwise Operators
* Membership Operators
* Identity Operators

Arithmetic operators

Arithmetic operators are used to perform arithmetic operations between two operands. It includes +(addition), - (subtraction), \*(multiplication), /(divide), %(reminder), //(floor division), and exponent (\*\*).

Consider the following table for a detailed explanation of arithmetic operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **+ (Addition)** | It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30 |
| **- (Subtraction)** | It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if a = 20, b = 10 => a ? b = 10 |
| **/ (divide)** | It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2 |
| **\* (Multiplication)** | It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a \* b = 200 |
| **% (reminder)** | It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0 |
| **\*\* (Exponent)** | It is an exponent operator represented as it calculates the first operand power to second operand. |
| **// (Floor division)** | It gives the floor value of the quotient produced by dividing the two operands. |

Comparison operator

Comparison operators are used to comparing the value of the two operands and returns boolean true or false accordingly. The comparison operators are described in the following table.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | If the value of two operands is equal, then the condition becomes true. |
| != | If the value of two operands is not equal then the condition becomes true. |
| <= | If the first operand is less than or equal to the second operand, then the condition becomes true. |
| >= | If the first operand is greater than or equal to the second operand, then the condition becomes true. |
| <> | If the value of two operands is not equal, then the condition becomes true. |
| > | If the first operand is greater than the second operand, then the condition becomes true. |
| **<** | If the first operand is less than the second operand, then the condition becomes true. |

Python assignment operators

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | It assigns the the value of the right expression to the left operand. |
| += | It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30. |
| -= | It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10. |
| \*= | It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a\* = b will be equal to a = a\* b and therefore, a = 200. |
| %= | It divides the value of the left operand by the value of the right operand and assign the reminder back to left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0. |
| \*\*= | a\*\*=b will be equal to a=a\*\*b, for example, if a = 4, b =2, a\*\*=b will assign 4\*\*2 = 16 to a. |
| //= | A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a. |

Bitwise operator

The bitwise operators perform bit by bit operation on the values of the two operands.

**For example,**

1. **if** a = 7;
2. b = 6;
3. then, binary (a) = 0111
4. binary (b) = 0011
6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & (binary and) | If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied. |
| | (binary or) | The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1. |
| ^ (binary xor) | The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0. |
| ~ (negation) | It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa. |
| << (left shift) | The left operand value is moved left by the number of bits present in the right operand. |
| >> (right shift) | The left operand is moved right by the number of bits present in the right operand. |

Logical Operators

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| and | If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true. |
| or | If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true. |
| not | If an expression **a** is true then not (a) will be false and vice versa. |

Membership Operators

Python membership operators are used to check the membership of value inside a data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| in | It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary). |
| not in | It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary). |

Identity Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| is | It is evaluated to be true if the reference present at both sides point to the same object. |
| is not | It is evaluated to be true if the reference present at both side do not point to the same object. |

Operator Precedence

The precedence of the operators is important to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in python is given below.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | The exponent operator is given priority over all the others used in the expression. |
| ~ + - | The negation, unary plus and minus. |
| \* / % // | The multiplication, divide, modules, reminder, and floor division. |
| + - | Binary plus and minus |
| >> << | Left shift and right shift |
| & | Binary and. |
| ^ | | Binary xor and or |
| <= < > >= | Comparison operators (less then, less then equal to, greater then, greater then equal to). |
| <> == != | Equality operators. |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

Python Comments

Comments in Python can be used to explain any program code. It can also be used to hide the code as well.

Comments are the most helpful stuff of any program. It enables us to understand the way, a program works. In python, any statement written along with # symbol is known as a comment. The interpreter does not interpret the comment.

Comment is not a part of the program, but it enhances the interactivity of the program and makes the program readable.

Python supports two types of comments:

**1) Single Line Comment:**

In case user wants to specify a single line comment, then comment must start with ?#?

**Eg:**

1. # This is single line comment.
2. **print** "Hello Python"

**Output:**

Hello Python

**2) Multi Line Comment:**

Multi lined comment can be given inside triple quotes.

**eg:**

1. ''''' This
2. Is
3. Multipline comment'''

**eg:**

1. #single line comment
2. **print** "Hello Python"
3. '''''This is
4. multiline comment'''

**Output:**

Hello Python

# Decision Making in Python

# **control structures**

# A control structure directs the order of execution of the statements in a program (referred to as the program’s [control flow](https://en.wikipedia.org/wiki/Control_flow)).

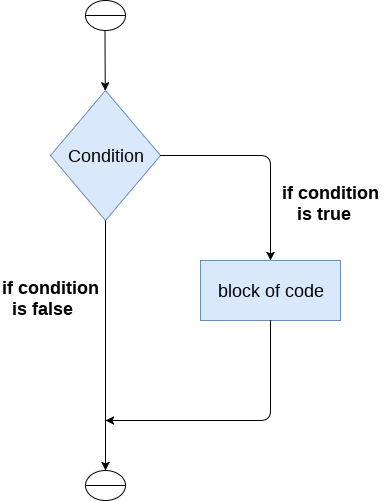
There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arises in programming also where we need to make some decisions and based on these decision we will execute the next block of code.

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

* [if statement](https://www.geeksforgeeks.org/decision-making-python-else-nested-elif/#if)
* [if..else statements](https://www.geeksforgeeks.org/decision-making-python-else-nested-elif/#if-else)
* [nested if statements](https://www.geeksforgeeks.org/decision-making-python-else-nested-elif/#nif)
* [if-elif ladder](https://www.geeksforgeeks.org/decision-making-python-else-nested-elif/#if-elif)

The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.

The syntax of the if-statement is given below.

1. **if** expression:
2. statement

### **Example 1**

1. num = int(input("enter the number?"))
2. **if** num%2 == 0:
3. **print**("Number is even")

**Output:**

enter the number?10

Number is even

### **Example 2 : Program to print the largest of the three numbers.**

1. a = int(input("Enter a? "));
2. b = int(input("Enter b? "));
3. c = int(input("Enter c? "));
4. **if** a>b **and** a>c:
5. **print**("a is largest");
6. **if** b>a **and** b>c:
7. **print**("b is largest");
8. **if** c>a **and** c>b:
9. **print**("c is largest");

**Output:**

Enter a? 100

Enter b? 120

Enter c? 130

c is largest

|  |
| --- |
| # python program to illustrate If statement    i = 10  if (i > 15):     print ("10 is less than 15")  print ("I am Not in if") |

Output:

I am Not in if

Thus, a compound if statement in Python looks like this:

1 if <expr>:

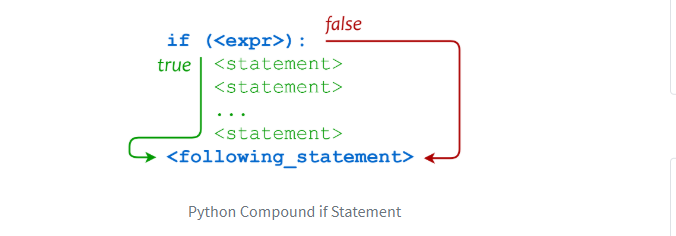
2 <statement>

3 <statement>

4 ...

5 <statement>

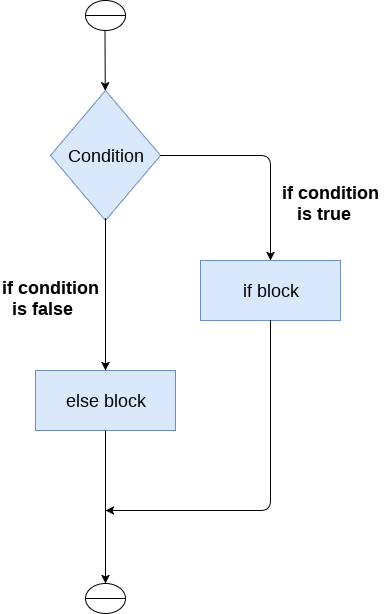
6 <following\_statement>



## The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

The syntax of the if-else statement is given below.

1. **if** condition:
2. #block of statements
3. **else**:
4. #another block of statements (else-block)

### **Program to check whether a person is eligible to vote or not.**

1. age = int (input("Enter your age? "))
2. **if** age>=18:
3. **print**("You are eligible to vote !!");
4. **else**:
5. **print**("Sorry! you have to wait !!");

**Output:**

Enter your age? 90

You are eligible to vote !!

### **Program to check whether a number is even or not.**

1. num = int(input("enter the number?"))
2. **if** num%2 == 0:
3. **print**("Number is even...")
4. **else**:
5. **print**("Number is odd...")
6. **Output:**
7. enter the number?10
8. Number is even

num = 3

# Try these two variations as well.

# num = -5

# num = 0

if num >= 0:

print("Positive or Zero")

else:

print("Negative number")

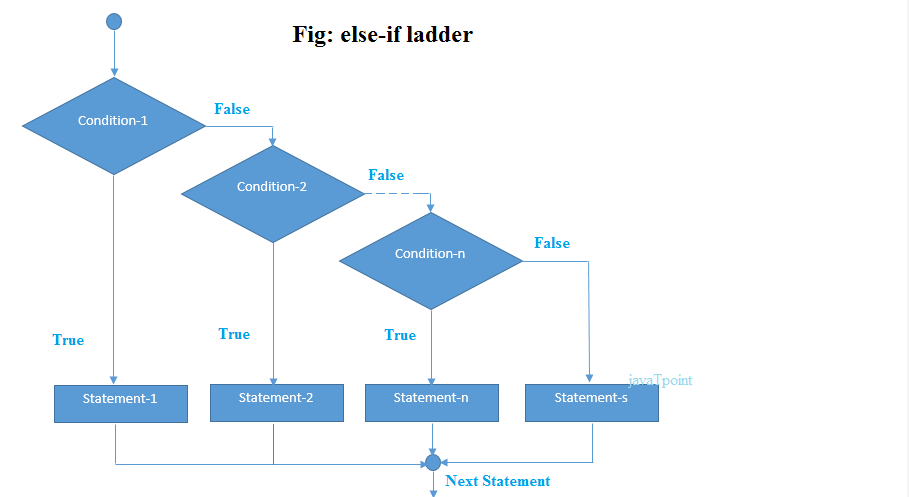
The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement is given below.

1. **if** expression 1:
2. # block of statements
4. **elif** expression 2:
5. # block of statements
7. **elif** expression 3:
8. # block of statements
10. **else**:
11. # block of statements



### **Example 1**

1. number = int(input("Enter the number?"))
2. **if** number==10:
3. **print**("number is equals to 10")
4. **elif** number==50:
5. **print**("number is equal to 50");
6. **elif** number==100:
7. **print**("number is equal to 100");
8. **else**:
9. **print**("number is not equal to 10, 50 or 100");

**Output:**

Enter the number?15

number is not equal to 10, 50 or 100

### **Example 2**

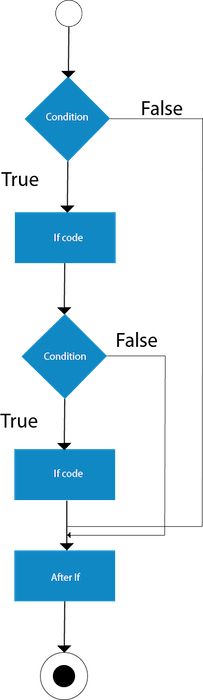
1. marks = int(input("Enter the marks? "))
2. f marks > 85 **and** marks <= 100:
3. **print**("Congrats ! you scored grade A ...")
4. lif marks > 60 **and** marks <= 85:
5. **print**("You scored grade B + ...")
6. lif marks > 40 **and** marks <= 60:
7. **print**("You scored grade B ...")
8. lif (marks > 30 **and** marks <= 40):
9. **print**("You scored grade C ...")
10. lse:
11. **print**("Sorry you are fail ?")

## Nested if statement

he nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

**Syntax:**

1. **if**(condition):
2. //code to be executed
3. **if**(condition):
4. //code to be executed



**Example:**

1. //Java Program to demonstrate the use of Nested If Statement.
2. **public** **class** JavaNestedIfExample {
3. **public** **static** **void** main(String[] args) {
4. //Creating two variables for age and weight
5. **int** age=20;
6. **int** weight=80;
7. //applying condition on age and weight
8. **if**(age>=18){
9. **if**(weight>50){
10. System.out.println("You are eligible to donate blood");
11. }
12. }
13. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaNestedIfExample)

Output:

You are eligible to donate blood

**Example 2:**

1. //Java Program to demonstrate the use of Nested If Statement.
2. **public** **class** JavaNestedIfExample2 {
3. **public** **static** **void** main(String[] args) {
4. //Creating two variables for age and weight
5. **int** age=25;
6. **int** weight=48;
7. //applying condition on age and weight
8. **if**(age>=18){
9. **if**(weight>50){
10. System.out.println("You are eligible to donate blood");
11. } **else**{
12. System.out.println("You are not eligible to donate blood");
13. }
14. } **else**{
15. System.out.println("Age must be greater than 18");
16. }
17. }  }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaNestedIfExample2)

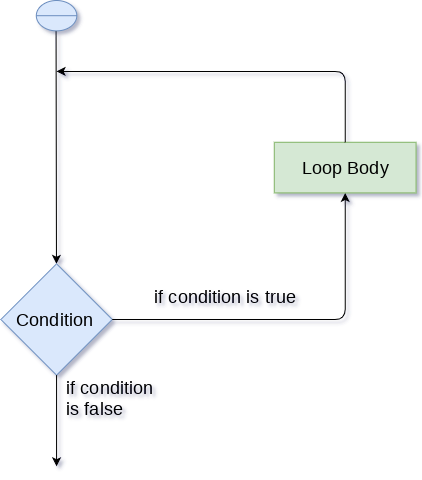
Output:

You are not eligible to donate blood

# Python Loops

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



## Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantages of loops

There are the following advantages of loops in Python.

1. It provides code re-usability.
2. Using loops, we do not need to write the same code again and again.
3. Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the following loop statements in Python.

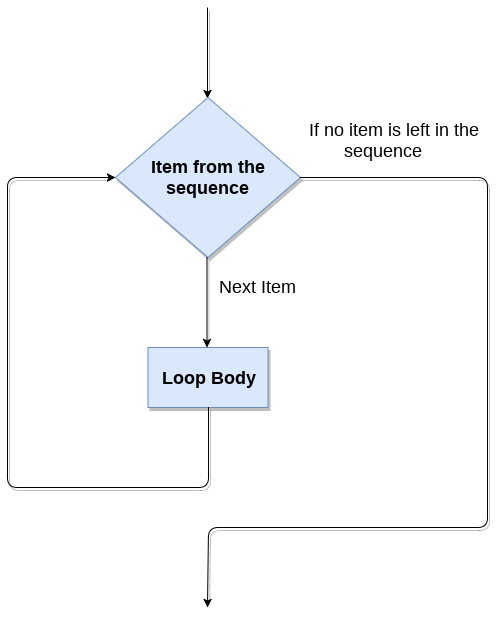
|  |  |
| --- | --- |
| **Loop Statement** | **Description** |
| for loop | The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance. |
| while loop | The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop. |
| do-while loop | The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs). |

Python for loop

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

1. **for** iterating\_var **in** sequence:
2. statement(s)



Example

1. i=1
2. n=int(input("Enter the number up to which you want to print the natural numbers?"))
3. **for** i **in** range(0,10):
4. **print**(i,end = ' ')

**Output:**

0 1 2 3 4 5 6 7 8 9

Python for loop example : printing the table of the given number

1. i=1;
2. num = int(input("Enter a number:"));
3. **for** i **in** range(1,11):
4. **print**("%d X %d = %d"%(num,i,num\*i));

**Output:**

Enter a number:10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

Nested for loop in python

Python allows us to nest any number of for loops inside a for loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax of the nested for loop in python is given below.

1. **for** iterating\_var1 **in** sequence:
2. **for** iterating\_var2 **in** sequence:
3. #block of statements
4. #Other statements

Example 1

1. n = int(input("Enter the number of rows you want to print?"))
2. i,j=0,0
3. **for** i **in** range(0,n):
4. **print**()
5. **for** j **in** range(0,i+1):
6. **print**("\*",end="")

**Output:**

Enter the number of rows you want to print?5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

Using else statement with for loop

Unlike other languages like C, C++, or Java, python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

Example 1

1. **for** i **in** range(0,5):
2. **print**(i)
3. **else**:**print**("for loop completely exhausted, since there is no break.");

In the above example, for loop is executed completely since there is no break statement in the loop. The control comes out of the loop and hence the else block is executed.

**Output:**

0

1

2

3

4

for loop completely exhausted, since there is no break.

Example 2

1. **for** i **in** range(0,5):
2. **print**(i)
3. **break**;
4. **else**:**print**("for loop is exhausted");
5. **print**("The loop is broken due to break statement...came out of loop")

In the above example, the loop is broken due to break statement therefore the else statement will not be executed. The statement present immediate next to else block will be executed.

**Output:**

0

The loop is broken due to break statement...came out of loop

Python while loop

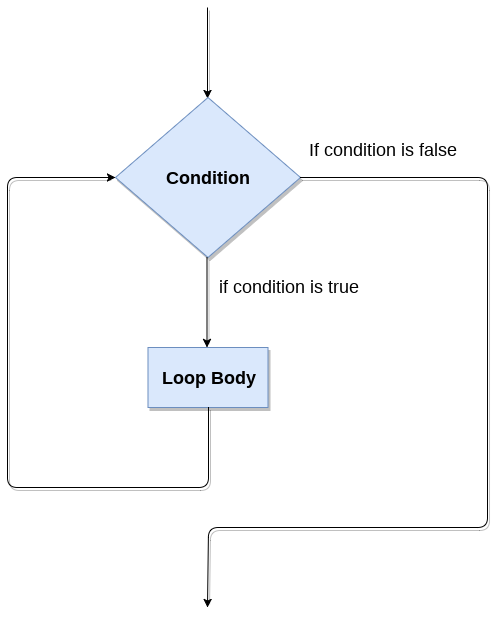
The while loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed as long as the given condition is true.

It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax is given below.

1. **while** expression:
2. statements

Here, the statements can be a single statement or the group of statements. The expression should be any valid python expression resulting into true or false. The true is any non-zero value.



Example 1

1. i=1;
2. **while** i<=10:
3. **print**(i);
4. i=i+1;

**Output:**

1

2

3

4

5

6

7

8

9

10

Example 2

1. i=1
2. number=0
3. b=9
4. number = int(input("Enter the number?"))
5. **while** i<=10:
6. **print**("%d X %d = %d \n"%(number,i,number\*i));
7. i = i+1;

**Output:**

Enter the number?10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

Infinite while loop

If the condition given in the while loop never becomes false then the while loop will never terminate and result into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition whereas 0 indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Example 1

1. **while** (1):
2. **print**("Hi! we are inside the infinite while loop");

**Output:**

Hi! we are inside the infinite while loop

(infinite times)

Example 2

1. var = 1
2. **while** var != 2:
3. i = int(input("Enter the number?"))
4. **print** ("Entered value is %d"%(i))

**Output:**

Enter the number?102

Entered value is 102

Enter the number?102

Entered value is 102

Enter the number?103

Entered value is 103

Enter the number?103

(infinite loop)

Using else with Python while loop

Python enables us to use the while loop with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed and the statement present after else block will be executed.

Consider the following example.

1. i=1;
2. **while** i<=5:
3. **print**(i)
4. i=i+1;
5. **else**:**print**("The while loop exhausted");

**Output:**

1

2

3

4

5

The while loop exhausted

Example 2

1. i=1;
2. **while** i<=5:
3. **print**(i)
4. i=i+1;
5. **if**(i==3):
6. **break**;
7. **else**:**print**("The while loop exhausted");

**Output:**

1

2

Python break statement

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

The syntax of the break is given below.

1. #loop statements
2. **break**;

Example 1

1. list =[1,2,3,4]
2. count = 1;
3. **for** i **in** list:
4. **if** i == 4:
5. **print**("item matched")
6. count = count + 1;
7. **break**
8. **print**("found at",count,"location");

**Output:**

item matched

found at 2 location

Example 2

1. str = "python"
2. **for** i **in** str:
3. **if** i == 'o':
4. **break**
5. **print**(i);

**Output:**

p

y

t

h

Example 3: break statement with while loop

1. i = 0;
2. **while** 1:
3. **print**(i," ",end=""),
4. i=i+1;
5. **if** i == 10:
6. **break**;
7. **print**("came out of while loop");

**Output:**

0 1 2 3 4 5 6 7 8 9 came out of while loop

Example 3

1. n=2
2. **while** 1:
3. i=1;
4. **while** i<=10:
5. **print**("%d X %d = %d\n"%(n,i,n\*i));
6. i = i+1;
7. choice = int(input("Do you want to continue printing the table, press 0 for no?"))
8. **if** choice == 0:
9. **break**;
10. n=n+1

**Output:**

2 X 1 = 2

2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

Do you want to continue printing the table, press 0 for no?1

3 X 1 = 3

3 X 2 = 6

3 X 3 = 9

3 X 4 = 12

3 X 5 = 15

3 X 6 = 18

3 X 7 = 21

3 X 8 = 24

3 X 9 = 27

3 X 10 = 30

Do you want to continue printing the table, press 0 for no?0

Python continue Statement

The continue statement in python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

The syntax of Python continue statement is given below.

1. #loop statements
2. **continue**;
3. #the code to be skipped

Example 1

1. i = 0;
2. **while** i!=10:
3. **print**("%d"%i);
4. **continue**;
5. i=i+1;

**Output:**

infinite loop

Example 2

1. i=1; #initializing a local variable
2. #starting a loop from 1 to 10
3. **for** i **in** range(1,11):
4. **if** i==5:
5. **continue**;
6. **print**("%d"%i);

**Output:**

1

2

3

4

6

7

8

9

10

Pass Statement

The pass statement is a null operation since nothing happens when it is executed. It is used in the cases where a statement is syntactically needed but we don't want to use any executable statement at its place.

For example, it can be used while overriding a parent class method in the subclass but don't want to give its specific implementation in the subclass.

Pass is also used where the code will be written somewhere but not yet written in the program file.

The syntax of the pass statement is given below.

Example

1. list = [1,2,3,4,5]
2. flag = 0
3. **for** i **in** list:
4. **print**("Current element:",i,end=" ");
5. **if** i==3:
6. **pass**;
7. **print**("\nWe are inside pass block\n");
8. flag = 1;
9. **if** flag==1:
10. **print**("\nCame out of pass\n");
11. flag=0;

**Output:**

Current element: 1 Current element: 2 Current element: 3

We are inside pass block

Came out of pass

Current element: 4 Current element: 5

Python Pass

In Python, pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty. It is same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.

**Python Pass Syntax**

1. **pass**

**Python Pass Example**

1. **for** i **in** [1,2,3,4,5]:
2. **if** i==3:
3. **pass**
4. **print** "Pass when value is",i
5. **print** i,

**Output:**

1. >>>
2. 1 2 Pass when value **is** 3
3. 3 4 5
4. >>>

Python String

Till now, we have discussed numbers as the standard data types in python. In this section of the tutorial, we will discuss the most popular data type in python i.e., string.

In python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Consider the following example in python to create a string.

1. str = "Hi Python !"

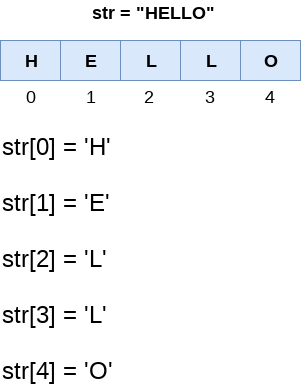
Here, if we check the type of the variable str using a python script

1. **print**(type(str)), then it will **print** string (str).

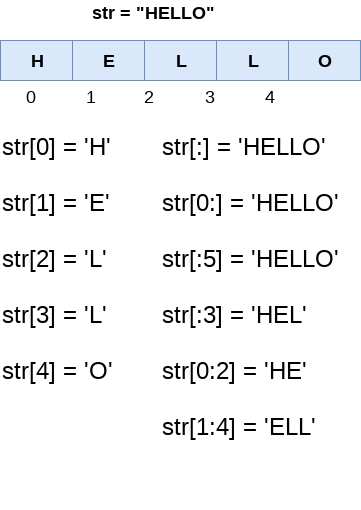
In python, strings are treated as the sequence of strings which means that python doesn't support the character data type instead a single character written as 'p' is treated as the string of length 1.

Strings indexing and splitting

Like other languages, the indexing of the python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.



As shown in python, the slice operator [] is used to access the individual characters of the string. However, we can use the : (colon) operator in python to access the substring. Consider the following example.



Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'python' is given, then str[1:3] will always include str[1] = 'p', str[2] = 'y', str[3] = 't' and nothing else.

Reassigning strings

Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with a new string since its content can not be partially replaced. Strings are immutable in python.

Consider the following example.

Example 1

1. str = "HELLO"
2. str[0] = "h"
3. **print**(str)

**Output:**

Traceback (most recent call last):

File "12.py", line 2, in <module>

str[0] = "h";

TypeError: 'str' object does not support item assignment

However, in example 1, the string str can be completely assigned to a new content as specified in the following example.

Example 2

1. str = "HELLO"
2. **print**(str)
3. str = "hello"
4. **print**(str)

**Output:**

HELLO

hello

String Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | It is known as concatenation operator used to join the strings given either side of the operator. |
| \* | It is known as repetition operator. It concatenates the multiple copies of the same string. |
| [] | It is known as slice operator. It is used to access the sub-strings of a particular string. |
| [:] | It is known as range slice operator. It is used to access the characters from the specified range. |
| in | It is known as membership operator. It returns if a particular sub-string is present in the specified string. |
| not in | It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string. |
| r/R | It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string. |
| % | It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python. |

Example

Consider the following example to understand the real use of Python operators.

1. str = "Hello"
2. str1 = " world"
3. **print**(str\*3) # prints HelloHelloHello
4. **print**(str+str1)# prints Hello world
5. **print**(str[4]) # prints o
6. **print**(str[2:4]); # prints ll
7. **print**('w' **in** str) # prints false as w is not present in str
8. **print**('wo' **not** **in** str1) # prints false as wo is present in str1.
9. **print**(r'C://python37') # prints C://python37 as it is written
10. **print**("The string str : %s"%(str)) # prints The string str : Hello

**Output:**

HelloHelloHello

Hello world

o

ll

False

False

C://python37

The string str : Hello

Python Formatting operator

Python allows us to use the format specifiers used in C's printf statement. The format specifiers in python are treated in the same way as they are treated in C. However, Python provides an additional operator % which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.

Consider the following example.

1. Integer = 10;
2. Float = 1.290
3. String = "Ayush"
4. **print**("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... My value is %s"%(Integer,Float,String));

**Output:**

Hi I am Integer ... My value is 10

Hi I am float ... My value is 1.290000

Hi I am string ... My value is Ayush

Built-in String functions

Python provides various in-built functions that are used for string handling. Many String fun

|  |  |
| --- | --- |
| **Method** | **Description** |
| [capitalize()](https://www.javatpoint.com/python-string-capitalize-method) | It capitalizes the first character of the String. This function is deprecated in python3 |
| [casefold()](https://www.javatpoint.com/python-string-casefold-method) | It returns a version of s suitable for case-less comparisons. |
| [center(width ,fillchar)](https://www.javatpoint.com/python-string-center-method) | It returns a space padded string with the original string centred with equal number of left and right spaces. |
| [count(string,begin,end)](https://www.javatpoint.com/python-string-count-method) | It counts the number of occurrences of a substring in a String between begin and end index. |
| decode(encoding = 'UTF8', errors = 'strict') | Decodes the string using codec registered for encoding. |
| [encode()](https://www.javatpoint.com/python-string-encode-method) | Encode S using the codec registered for encoding. Default encoding is 'utf-8'. |
| [endswith(suffix ,begin=0,end=len(string))](https://www.javatpoint.com/python-string-endswith-method) | It returns a Boolean value if the string terminates with given suffix between begin and end. |
| [expandtabs(tabsize = 8)](https://www.javatpoint.com/python-string-expandtabs-method) | It defines tabs in string to multiple spaces. The default space value is 8. |
| [find(substring ,beginIndex, endIndex)](https://www.javatpoint.com/python-string-find-method) | It returns the index value of the string where substring is found between begin index and end index. |
| [format(value)](https://www.javatpoint.com/python-string-format-method) | It returns a formatted version of S, using the passed value. |
| [index(subsring, beginIndex, endIndex)](https://www.javatpoint.com/python-string-index-method) | It throws an exception if string is not found. It works same as find() method. |
| [isalnum()](https://www.javatpoint.com/python-string-isalnum-method) | It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false. |
| [isalpha()](https://www.javatpoint.com/python-string-isalpha-method) | It returns true if all the characters are alphabets and there is at least one character, otherwise False. |
| [isdecimal()](https://www.javatpoint.com/python-string-isdecimal-method) | It returns true if all the characters of the string are decimals. |
| [isdigit()](https://www.javatpoint.com/python-string-isdigit-method) | It returns true if all the characters are digits and there is at least one character, otherwise False. |
| [isidentifier()](https://www.javatpoint.com/python-string-isidentifier-method) | It returns true if the string is the valid identifier. |
| [islower()](https://www.javatpoint.com/python-string-islower-method) | It returns true if the characters of a string are in lower case, otherwise false. |
| [isnumeric()](https://www.javatpoint.com/python-string-isnumeric-method) | It returns true if the string contains only numeric characters. |
| [isprintable()](https://www.javatpoint.com/python-string-isprintable-method) | It returns true if all the characters of s are printable or s is empty, false otherwise. |
| [isupper()](https://www.javatpoint.com/python-string-isupper-method) | It returns false if characters of a string are in Upper case, otherwise False. |
| [isspace()](https://www.javatpoint.com/python-string-isspace-method) | It returns true if the characters of a string are white-space, otherwise false. |
| [istitle()](https://www.javatpoint.com/python-string-istitle-method) | It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case. |
| [isupper()](https://www.javatpoint.com/python-string-isupper-method) | It returns true if all the characters of the string(if exists) is true otherwise it returns false. |
| [join(seq)](https://www.javatpoint.com/python-string-join-method) | It merges the strings representation of the given sequence. |
| len(string) | It returns the length of a string. |
| [ljust(width[,fillchar])](https://www.javatpoint.com/python-string-ljust-method) | It returns the space padded strings with the original string left justified to the given width. |
| [lower()](https://www.javatpoint.com/python-string-lower-method) | It converts all the characters of a string to Lower case. |
| [lstrip()](https://www.javatpoint.com/python-string-lstrip-method) | It removes all leading whitespaces of a string and can also be used to remove particular character from leading. |
| [partition()](https://www.javatpoint.com/python-string-partition-method) | It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings. |
| maketrans() | It returns a translation table to be used in translate function. |
| [replace(old,new[,count])](https://www.javatpoint.com/python-string-replace-method) | It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given. |
| [rfind(str,beg=0,end=len(str))](https://www.javatpoint.com/python-string-rfind-method) | It is similar to find but it traverses the string in backward direction. |
| [rindex(str,beg=0,end=len(str))](https://www.javatpoint.com/python-string-rindex-method) | It is same as index but it traverses the string in backward direction. |
| [rjust(width,[,fillchar])](https://www.javatpoint.com/python-string-rjust-method) | Returns a space padded string having original string right justified to the number of characters specified. |
| [rstrip()](https://www.javatpoint.com/python-string-rstrip-method) | It removes all trailing whitespace of a string and can also be used to remove particular character from trailing. |
| [rsplit(sep=None, maxsplit = -1)](https://www.javatpoint.com/python-string-rsplit-method) | It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space. |
| [split(str,num=string.count(str))](https://www.javatpoint.com/python-string-split-method) | Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter. |
| [splitlines(num=string.count('\n'))](https://www.javatpoint.com/python-string-splitlines-method) | It returns the list of strings at each line with newline removed. |
| [startswith(str,beg=0,end=len(str))](https://www.javatpoint.com/python-string-startswith-method) | It returns a Boolean value if the string starts with given str between begin and end. |
| strip([chars]) | It is used to perform lstrip() and rstrip() on the string. |
| [swapcase()](https://www.javatpoint.com/python-string-swapcase-method) | It inverts case of all characters in a string. |
| title() | It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut. |
| [translate(table,deletechars = '')](https://www.javatpoint.com/python-string-translate-method) | It translates the string according to the translation table passed in the function . |
| [upper()](https://www.javatpoint.com/python-string-upper-method) | It converts all the characters of a string to Upper Case. |
| [zfill(width)](https://www.javatpoint.com/python-string-zfill-method) | Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). |
| [rpartition()](https://www.javatpoint.com/python-string-rpartition-method) |  |

Python List

List in python is implemented to store the sequence of various type of data. However, python contains six data types that are capable to store the sequences but the most common and reliable type is list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be defined as follows.

1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]
3. L3 = [1, "Ryan"]

If we try to print the type of L1, L2, and L3 then it will come out to be a list.

Lets consider a proper example to define a list and printing its values.

1. emp = ["John", 102, "USA"]
2. Dep1 = ["CS",10];
3. Dep2 = ["IT",11];
4. HOD\_CS = [10,"Mr. Holding"]
5. HOD\_IT = [11, "Mr. Bewon"]
6. **print**("printing employee data...");
7. **print**("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
8. **print**("printing departments...");
9. **print**("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]));
10. **print**("HOD Details ....");
11. **print**("CS HOD Name: %s, Id: %d"%(HOD\_CS[1],HOD\_CS[0]));
12. **print**("IT HOD Name: %s, Id: %d"%(HOD\_IT[1],HOD\_IT[0]));
13. **print**(type(emp),type(Dep1),type(Dep2),type(HOD\_CS),type(HOD\_IT));

**Output:**

printing employee data...

Name : John, ID: 102, Country: USA

printing departments...

Department 1:

Name: CS, ID: 11

Department 2:

Name: IT, ID: 11

HOD Details ....

CS HOD Name: Mr. Holding, Id: 10

IT HOD Name: Mr. Bewon, Id: 11

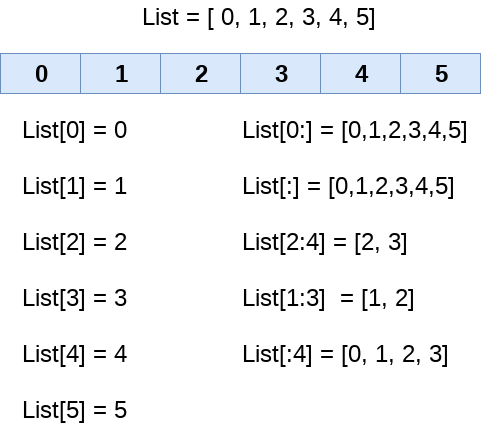
<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>

List indexing and splitting

The indexing are processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

Consider the following example.



Unlike other languages, python provides us the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (right most) of the list has the index -1, its adjacent left element is present at the index -2 and so on until the left most element is encountered.

Python Lists

Updating List values

Lists are the most versatile data structures in python since they are immutable and their values can be updated by using the slice and assignment operator.

Python also provide us the append() method which can be used to add values to the string.

Consider the following example to update the values inside the list.

1. List = [1, 2, 3, 4, 5, 6]
2. **print**(List)
3. List[2] = 10;
4. **print**(List)
5. List[1:3] = [89, 78]
6. **print**(List)

**Output:**

[1, 2, 3, 4, 5, 6]

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

The list elements can also be deleted by using the **del** keyword. Python also provides us the remove() method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

1. List = [0,1,2,3,4]
2. **print**(List)
3. **del** List[0]
4. **print**(List)
5. **del** List[3]
6. **print**(List)

**Output:**

[0, 1, 2, 3, 4]

[1, 2, 3, 4]

[1, 2, 3]

Python List Operations

The concatenation (+) and repetition (\*) operator work in the same way as they were working with the strings.

Lets see how the list responds to various operators.

1. Consider a List l1 = [1, 2, 3, 4], **and** l2 = [5, 6, 7, 8]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| Repetition | The repetition operator enables the list elements to be repeated multiple times. | L1\*2 = [1, 2, 3, 4, 1, 2, 3, 4] |
| Concatenation | It concatenates the list mentioned on either side of the operator. | l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8] |
| Membership | It returns true if a particular item exists in a particular list otherwise false. | print(2 in l1) prints True. |
| Iteration | The for loop is used to iterate over the list elements. | for i in l1:  print(i)  **Output**  1  2  3  4 |
| Length | It is used to get the length of the list | len(l1) = 4 |

Iterating a List

A list can be iterated by using a for - in loop. A simple list containing four strings can be iterated as follows.

1. List = ["John", "David", "James", "Jonathan"]
2. **for** i **in** List: #i will iterate over the elements of the List and contains each element in each iteration.
3. **print**(i);

**Output:**

John

David

James

Jonathan

Adding elements to the list

Python provides append() function by using which we can add an element to the list. However, the append() method can only add the value to the end of the list.

Consider the following example in which, we are taking the elements of the list from the user and printing the list on the console.

1. l =[];
2. n = int(input("Enter the number of elements in the list")); #Number of elements will be entered by the user
3. **for** i **in** range(0,n): # for loop to take the input
4. l.append(input("Enter the item?")); # The input is taken from the user and added to the list as the item
5. **print**("printing the list items....");
6. **for** i **in** l: # traversal loop to print the list items
7. **print**(i, end = "  ");

**Output:**

Enter the number of elements in the list 5

Enter the item?1

Enter the item?2

Enter the item?3

Enter the item?4

Enter the item?5

printing the list items....

1 2 3 4 5

Removing elements from the list

1. List = [0,1,2,3,4]
2. **print**("printing original list: ");
3. **for** i **in** List:
4. **print**(i,end=" ")
5. List.remove(0)
6. **print**("\nprinting the list after the removal of first element...")
7. **for** i **in** List:
8. **print**(i,end=" ")

**Output:**

printing original list:

0 1 2 3 4

printing the list after the removal of first element...

1 2 3 4

Python List Built-in functions

Python provides the following built-in functions which can be used with the lists.

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | cmp(list1, list2) | It compares the elements of both the lists. |
| 2 | len(list) | It is used to calculate the length of the list. |
| 3 | max(list) | It returns the maximum element of the list. |
| 4 | min(list) | It returns the minimum element of the list. |
| 5 | list(seq) | It converts any sequence to the list. |

Python List built-in methods

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | [list.append(obj)](https://www.javatpoint.com/python-list-append-method) | The element represented by the object obj is added to the list. |
| 2 | [list.clear()](https://www.javatpoint.com/python-list-clear-method) | It removes all the elements from the list. |
| 3 | [List.copy()](https://www.javatpoint.com/python-list-copy-method) | It returns a shallow copy of the list. |
| 4 | [list.count(obj)](https://www.javatpoint.com/python-list-count-method) | It returns the number of occurrences of the specified object in the list. |
| 5 | [list.extend(seq)](https://www.javatpoint.com/python-list-extend-method) | The sequence represented by the object seq is extended to the list. |
| 6 | [list.index(obj)](https://www.javatpoint.com/python-list-index-method) | It returns the lowest index in the list that object appears. |
| 7 | [list.insert(index, obj)](https://www.javatpoint.com/python-list-insert-method) | The object is inserted into the list at the specified index. |
| 8 | [list.pop(obj=list[-1])](https://www.javatpoint.com/python-list-pop-method) | It removes and returns the last object of the list. |
| 9 | [list.remove(obj)](https://www.javatpoint.com/python-list-remove-method) | It removes the specified object from the list. |
| 10 | [list.reverse()](https://www.javatpoint.com/python-list-reverse-method) | It reverses the list. |
| 11 | [list.sort([func])](https://www.javatpoint.com/python-list-sort-method) | It sorts the list by using the specified compare function if given. |

Python Tuple

Python Tuple is used to store the sequence of immutable python objects. Tuple is similar to lists since the value of the items stored in the list can be changed whereas the tuple is immutable and the value of the items stored in the tuple can not be changed.

A tuple can be written as the collection of comma-separated values enclosed with the small brackets. A tuple can be defined as follows.

1. T1 = (101, "Ayush", 22)
2. T2 = ("Apple", "Banana", "Orange")

Example

1. tuple1 = (10, 20, 30, 40, 50, 60)
2. **print**(tuple1)
3. count = 0
4. **for** i **in** tuple1:
5. **print**("tuple1[%d] = %d"%(count, i));

**Output:**

(10, 20, 30, 40, 50, 60)

tuple1[0] = 10

tuple1[0] = 20

tuple1[0] = 30

tuple1[0] = 40

tuple1[0] = 50

tuple1[0] = 60

Example 2

1. tuple1 = tuple(input("Enter the tuple elements ..."))
2. **print**(tuple1)
3. count = 0
4. **for** i **in** tuple1:
5. **print**("tuple1[%d] = %s"%(count, i));

**Output:**

Enter the tuple elements ...12345

('1', '2', '3', '4', '5')

tuple1[0] = 1

tuple1[0] = 2

tuple1[0] = 3

tuple1[0] = 4

tuple1[0] = 5

However, if we try to reassign the items of a tuple, we would get an error as the tuple object doesn't support the item assignment.

An empty tuple can be written as follows.

1. T3 = ()

The tuple having a single value must include a comma as given below.

1. T4 = (90,)

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

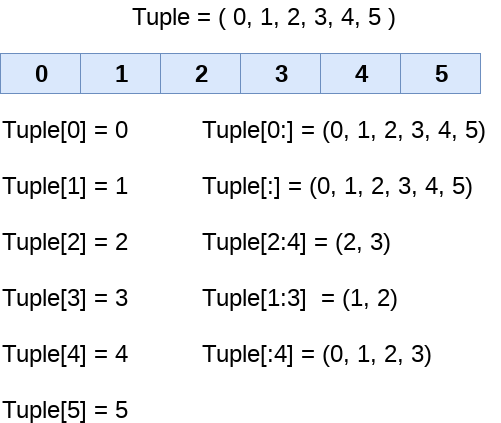
We will see all these aspects of tuple in this section of the tutorial.

Tuple indexing and splitting

The indexing and slicing in tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the slice operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.



Unlike lists, the tuple items can not be deleted by using the del keyword as tuples are immutable. To delete an entire tuple, we can use the del keyword with the tuple name.

Consider the following example.

1. tuple1 = (1, 2, 3, 4, 5, 6)
2. **print**(tuple1)
3. **del** tuple1[0]
4. **print**(tuple1)
5. **del** tuple1
6. **print**(tuple1)

**Output:**

(1, 2, 3, 4, 5, 6)

Traceback (most recent call last):

File "tuple.py", line 4, in <module>

print(tuple1)

NameError: name 'tuple1' is not defined

Like lists, the tuple elements can be accessed in both the directions. The right most element (last) of the tuple can be accessed by using the index -1. The elements from left to right are traversed using the negative indexing.

Consider the following example.

1. tuple1 = (1, 2, 3, 4, 5)
2. **print**(tuple1[-1])
3. **print**(tuple1[-4])

**Output:**

5

2

Basic Tuple operations

The operators like concatenation (+), repetition (\*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| Repetition | The repetition operator enables the tuple elements to be repeated multiple times. | T1\*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5) |
| Concatenation | It concatenates the tuple mentioned on either side of the operator. | T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9) |
| Membership | It returns true if a particular item exists in the tuple otherwise false. | print (2 in T1) prints True. |
| Iteration | The for loop is used to iterate over the tuple elements. | for i in T1:  print(i)  **Output**  1  2  3  4  5 |
| Length | It is used to get the length of the tuple. | len(T1) = 5 |

Python Tuple inbuilt functions

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | cmp(tuple1, tuple2) | It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false. |
| 2 | len(tuple) | It calculates the length of the tuple. |
| 3 | max(tuple) | It returns the maximum element of the tuple. |
| 4 | min(tuple) | It returns the minimum element of the tuple. |
| 5 | tuple(seq) | It converts the specified sequence to the tuple. |

Where use tuple

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.

2. Tuple can simulate dictionary without keys. Consider the following nested structure which can be used as a dictionary.

1. [(101, "John", 22), (102, "Mike", 28),  (103, "Dustin", 30)]

3. Tuple can be used as the key inside dictionary due to its immutable nature.

List VS Tuple

|  |  |  |
| --- | --- | --- |
| **SN** | **List** | **Tuple** |
| 1 | The literal syntax of list is shown by the []. | The literal syntax of the tuple is shown by the (). |
| 2 | The List is mutable. | The tuple is immutable. |
| 3 | The List has the variable length. | The tuple has the fixed length. |
| 4 | The list provides more functionality than tuple. | The tuple provides less functionality than the list. |
| 5 | The list Is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed. | The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary. |

Nesting List and tuple

We can store list inside tuple or tuple inside the list up to any number of level.

Lets see an example of how can we store the tuple inside the list.

1. Employees = [(101, "Ayush", 22), (102, "john", 29), (103, "james", 45), (104, "Ben", 34)]
2. **print**("----Printing list----");
3. **for** i **in** Employees:
4. **print**(i)
5. Employees[0] = (110, "David",22)
6. **print**();
7. **print**("----Printing list after modification----");
8. **for** i **in** Employees:
9. **print**(i)

**Output:**

----Printing list----

(101, 'Ayush', 22)

(102, 'john', 29)

(103, 'james', 45)

(104, 'Ben', 34)

----Printing list after modification----

(110, 'David', 22)

(102, 'john', 29)

(103, 'james', 45)

(104, 'Ben', 34)

Python Set

The set in python can be defined as the unordered collection of various items enclosed within the curly braces. The elements of the set can not be duplicate. The elements of the python set must be immutable.

Unlike other collections in python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together or we can get the list of elements by looping through the set.

Creating a set

The set can be created by enclosing the comma separated items with the curly braces. Python also provides the set method which can be used to create the set by the passed sequence.

Example 1: using curly braces

1. Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ... ")
5. **for** i **in** Days:
6. **print**(i)

**Output:**

{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}

<class 'set'>

looping through the set elements ...

Friday

Tuesday

Monday

Saturday

Thursday

Sunday

Wednesday

Example 2: using set() method

1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ... ")
5. **for** i **in** Days:
6. **print**(i)

**Output:**

{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}

<class 'set'>

looping through the set elements ...

Friday

Wednesday

Thursday

Saturday

Monday

Tuesday

Sunday

Python Set operations

In the previous example, we have discussed about how the set is created in python. However, we can perform various mathematical operations on python sets like union, intersection, difference, etc.

Adding items to the set

Python provides the add() method which can be used to add some particular item to the set. Consider the following example.

Example:

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nAdding other months to the set...");
5. Months.add("July");
6. Months.add("August");
7. **print**("\nPrinting the modified set...");
8. **print**(Months)
9. **print**("\nlooping through the set elements ... ")
10. **for** i **in** Months:
11. **print**(i)

**Output:**

printing the original set ...

{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...

Printing the modified set...

{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...

February

July

May

April

March

August

June

January

To add more than one item in the set, Python provides the **update()** method.

Consider the following example.

Example

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nupdating the original set ... ")
5. Months.update(["July","August","September","October"]);
6. **print**("\nprinting the modified set ... ")
7. **print**(Months);

**Output:**

printing the original set ...

{'January', 'February', 'April', 'May', 'June', 'March'}

updating the original set ...

printing the modified set ...

{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}

Removing items from the set

Python provides **discard()** method which can be used to remove the items from the set.

Consider the following example.

Example

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving some months from the set...");
5. Months.discard("January");
6. Months.discard("May");
7. **print**("\nPrinting the modified set...");
8. **print**(Months)
9. **print**("\nlooping through the set elements ... ")
10. **for** i **in** Months:
11. **print**(i)

**Output:**

printing the original set ...

{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...

Printing the modified set...

{'February', 'March', 'April', 'June'}

looping through the set elements ...

February

March

April

June

Python also provide the remove() method to remove the items from the set. Consider the following example to remove the items using remove() method.

Example

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving some months from the set...");
5. Months.remove("January");
6. Months.remove("May");
7. **print**("\nPrinting the modified set...");
8. **print**(Months)

**Output:**

printing the original set ...

{'February', 'June', 'April', 'May', 'January', 'March'}

Removing some months from the set...

Printing the modified set...

{'February', 'June', 'April', 'March'}

We can also use the pop() method to remove the item. However, this method will always remove the last item.

Consider the following example to remove the last item from the set.

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving some months from the set...");
5. Months.pop();
6. Months.pop();
7. **print**("\nPrinting the modified set...");
8. **print**(Months)

**Output:**

printing the original set ...

{'June', 'January', 'May', 'April', 'February', 'March'}

Removing some months from the set...

Printing the modified set...

{'May', 'April', 'February', 'March'}

Python provides the clear() method to remove all the items from the set.

Consider the following example.

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving all the items from the set...");
5. Months.clear()
6. **print**("\nPrinting the modified set...")
7. **print**(Months)

**Output:**

printing the original set ...

{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...

set()

Difference between discard() and remove()

Despite the fact that discard() and remove() method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the python will give the error.

Consider the following example.

Example

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving items through discard() method...");
5. Months.discard("Feb"); #will not give an error although the key feb is not available in the set
6. **print**("\nprinting the modified set...")
7. **print**(Months)
8. **print**("\nRemoving items through remove() method...");
9. Months.remove("Jan") #will give an error as the key jan is not available in the set.
10. **print**("\nPrinting the modified set...")
11. **print**(Months)

**Output:**

printing the original set ...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through remove() method...

Traceback (most recent call last):

File "set.py", line 9, in

Months.remove("Jan")

KeyError: 'Jan'

Union of two Sets

The union of two sets are calculated by using the or (|) operator. The union of the two sets contains the all the items that are present in both the sets.

Consider the following example to calculate the union of two sets.

Example 1 : using union | operator

1. Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. **print**(Days1|Days2) #printing the union of the sets

**Output:**

{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 'Thursday'}

Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.

Example 2: using union() method

1. Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. **print**(Days1.union(Days2)) #printing the union of the sets

**Output:**

{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 'Saturday'}

Intersection of two sets

The & (intersection) operator is used to calculate the intersection of the two sets in python. The intersection of the two sets are given as the set of the elements that common in both sets.

Consider the following example.

Example 1: using & operator

1. set1 = {"Ayush","John", "David", "Martin"}
2. set2 = {"Steve","Milan","David", "Martin"}
3. **print**(set1&set2) #prints the intersection of the two sets

**Output:**

{'Martin', 'David'}

Example 2: using intersection() method

1. set1 = {"Ayush","John", "David", "Martin"}
2. set2 = {"Steave","Milan","David", "Martin"}
3. **print**(set1.intersection(set2)) #prints the intersection of the two sets

**Output:**

{'Martin', 'David'}

The intersection\_update() method

The intersection\_update() method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

The Intersection\_update() method is different from intersection() method since it modifies the original set by removing the unwanted items, on the other hand, intersection() method returns a new set.

Consider the following example.

1. a = {"ayush", "bob", "castle"}
2. b = {"castle", "dude", "emyway"}
3. c = {"fuson", "gaurav", "castle"}
5. a.intersection\_update(b, c)
7. **print**(a)

**Output:**

{'castle'}

Difference of two sets

The difference of two sets can be calculated by using the subtraction (-) operator. The resulting set will be obtained by removing all the elements from set 1 that are present in set 2.

Consider the following example.

Example 1 : using subtraction ( - ) operator

1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. **print**(Days1-Days2) #{"Wednesday", "Thursday" will be printed}

**Output:**

{'Thursday', 'Wednesday'}

Example 2 : using difference() method

1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. **print**(Days1.difference(Days2)) # prints the difference of the two sets Days1 and Days2

**Output:**

{'Thursday', 'Wednesday'}

Set comparisons

Python allows us to use the comparison operators i.e., <, >, <=, >= , == with the sets by using which we can check whether a set is subset, superset, or equivalent to other set. The boolean true or false is returned depending upon the items present inside the sets.

Consider the following example.

1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday"}
3. Days3 = {"Monday", "Tuesday", "Friday"}
5. #Days1 is the superset of Days2 hence it will print true.
6. **print** (Days1>Days2)
8. #prints false since Days1 is not the subset of Days2
9. **print** (Days1<Days2)
11. #prints false since Days2 and Days3 are not equivalent
12. **print** (Days2 == Days3)

**Output:**

True

False

False

FrozenSets

The frozen sets are the immutable form of the normal sets, i.e., the items of the frozen set can not be changed and therefore it can be used as a key in dictionary.

The elements of the frozen set can not be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().

The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

Consider the following example to create the frozen set.

1. Frozenset = frozenset([1,2,3,4,5])
2. **print**(type(Frozenset))
3. **print**("\nprinting the content of frozen set...")
4. **for** i **in** Frozenset:
5. **print**(i);
6. Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation

**Output:**

<class 'frozenset'>

printing the content of frozen set...

1

2

3

4

5

Traceback (most recent call last):

File "set.py", line 6, in <module>

Frozenset.add(6) #gives an error since we can change the content of Frozenset after creation

AttributeError: 'frozenset' object has no attribute 'add'

Frozenset for the dictionary

If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

Consider the following example.

1. Dictionary = {"Name":"John", "Country":"USA", "ID":101}
2. **print**(type(Dictionary))
3. Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the dictionary
4. **print**(type(Frozenset))
5. **for** i **in** Frozenset:
6. **print**(i)

**Output:**

<class 'dict'>

<class 'frozenset'>

Name

Country

ID

Python Built-in set methods

Python contains the following methods to be used with the sets.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [add(item)](https://www.javatpoint.com/python-set-add-method) | It adds an item to the set. It has no effect if the item is already present in the set. |
| 2 | clear() | It deletes all the items from the set. |
| 3 | copy() | It returns a shallow copy of the set. |
| 4 | difference\_update(....) | It modifies this set by removing all the items that are also present in the specified sets. |
| 5 | [discard(item)](https://www.javatpoint.com/python-set-discard-method) | It removes the specified item from the set. |
| 6 | intersection() | It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified). |
| 7 | intersection\_update(....) | It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified). |
| 8 | Isdisjoint(....) | Return True if two sets have a null intersection. |
| 9 | Issubset(....) | Report whether another set contains this set. |
| 10 | Issuperset(....) | Report whether this set contains another set. |
| 11 | [pop()](https://www.javatpoint.com/python-set-pop-method) | Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty. |
| 12 | [remove(item)](https://www.javatpoint.com/python-set-remove-method) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 13 | symmetric\_difference(....) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 14 | symmetric\_difference\_update(....) | Update a set with the symmetric difference of itself and another. |
| 15 | union(....) | Return the union of sets as a new set. (i.e. all elements that are in either set.) |
| 16 | update() | Update a set with the union of itself and others. |

Python Dictionary

Dictionary is used to implement the key-value pair in python. The dictionary is the data type in python which can simulate the real-life data arrangement where some specific value exists for some particular key.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any python object whereas the keys are the immutable python object, i.e., Numbers, string or tuple.

Dictionary simulates Java hash-map in python.

Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the small brackets () and separated by the colon (:). The collections of the key-value pairs are enclosed within the curly braces {}.

The syntax to define the dictionary is given below.

1. Dict = {"Name": "Ayush","Age": 22}

In the above dictionary **Dict**, The keys **Name**, and **Age** are the string that is an immutable object.

Let's see an example to create a dictionary and printing its content.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)

**Output**

<class 'dict'>

printing Employee data ....

{'Age': 29, 'salary': 25000, 'Name': 'John', 'Company': 'GOOGLE'}

Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**("Name : %s" %Employee["Name"])
5. **print**("Age : %d" %Employee["Age"])
6. **print**("Salary : %d" %Employee["salary"])
7. **print**("Company : %s" %Employee["Company"])

**Output:**

<class 'dict'>

printing Employee data ....

Name : John

Age : 29

Salary : 25000

Company : GOOGLE

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

Updating dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys.

Let's see an example to update the dictionary values.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. **print**("printing the new data");
11. **print**(Employee)

**Output:**

<class 'dict'>

printing Employee data ....

{'Name': 'John', 'salary': 25000, 'Company': 'GOOGLE', 'Age': 29}

Enter the details of the new employee....

Name: David

Age: 19

Salary: 8900

Company:JTP

printing the new data

{'Name': 'David', 'salary': 8900, 'Company': 'JTP', 'Age': 19}

Deleting elements using del keyword

The items of the dictionary can be deleted by using the del keyword as given below.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Deleting some of the employee data")
6. **del** Employee["Name"]
7. **del** Employee["Company"]
8. **print**("printing the modified information ")
9. **print**(Employee)
10. **print**("Deleting the dictionary: Employee");
11. **del** Employee
12. **print**("Lets try to print it again ");
13. **print**(Employee)

**Output:**

<class 'dict'>

printing Employee data ....

{'Age': 29, 'Company': 'GOOGLE', 'Name': 'John', 'salary': 25000}

Deleting some of the employee data

printing the modified information

{'Age': 29, 'salary': 25000}

Deleting the dictionary: Employee

Lets try to print it again

Traceback (most recent call last):

File "list.py", line 13, in <module>

print(Employee)

NameError: name 'Employee' is not defined

Iterating Dictionary

A dictionary can be iterated using the for loop as given below.

Example 1

**# for loop to print all the keys of a dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3. **print**(x);

**Output:**

Name

Company

salary

Age

Example 2

**#for loop to print all the values of the dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3. **print**(Employee[x]);

**Output:**

29

GOOGLE

John

25000

Example 3

**#for loop to print the values of the dictionary by using values() method.**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.values():
3. **print**(x);

**Output:**

GOOGLE

25000

John

29

Example 4

#for loop to print the items of the dictionary by using items() method.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.items():
3. **print**(x);

**Output:**

('Name', 'John')

('Age', 29)

('salary', 25000)

('Company', 'GOOGLE')

Properties of Dictionary keys

1. In the dictionary, we can not store multiple values for the same keys. If we pass more than one values for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

1. Employee = {"Name": "John", "Age": 29, "Salary":25000,"Company":"GOOGLE","Name":"Johnn"}
2. **for** x,y **in** Employee.items():
3. **print**(x,y)

**Output:**

Salary 25000

Company GOOGLE

Name Johnn

Age 29

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuple as the key but we can not use any mutable object like the list as the key in the dictionary.

Consider the following example.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
2. **for** x,y **in** Employee.items():
3. **print**(x,y)

**Output:**

Traceback (most recent call last):

File "list.py", line 1, in

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}

TypeError: unhashable type: 'list'

Built-in Dictionary functions

The built-in python dictionary methods along with the description are given below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | cmp(dict1, dict2) | It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false. |
| 2 | len(dict) | It is used to calculate the length of the dictionary. |
| 3 | str(dict) | It converts the dictionary into the printable string representation. |
| 4 | type(variable) | It is used to print the type of the passed variable. |

Built-in Dictionary methods

The built-in python dictionary methods along with the description are given below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [dic.clear()](https://www.javatpoint.com/python-dictionary-clear-method) | It is used to delete all the items of the dictionary. |
| 2 | [dict.copy()](https://www.javatpoint.com/python-dictionary-copy-method) | It returns a shallow copy of the dictionary. |
| 3 | [dict.fromkeys(iterable, value = None, /)](https://www.javatpoint.com/python-dictionary-fromkeys-method) | Create a new dictionary from the iterable with the values equal to value. |
| 4 | [dict.get(key, default = "None")](https://www.javatpoint.com/python-dictionary-get-method) | It is used to get the value specified for the passed key. |
| 5 | dict.has\_key(key) | It returns true if the dictionary contains the specified key. |
| 6 | [dict.items()](https://www.javatpoint.com/python-dictionary-items-method) | It returns all the key-value pairs as a tuple. |
| 7 | [dict.keys()](https://www.javatpoint.com/python-dictionary-keys-method) | It returns all the keys of the dictionary. |
| 8 | [dict.setdefault(key,default= "None")](https://www.javatpoint.com/python-dictionary-setdefault-method) | It is used to set the key to the default value if the key is not specified in the dictionary |
| 9 | [dict.update(dict2)](https://www.javatpoint.com/python-dictionary-update-method) | It updates the dictionary by adding the key-value pair of dict2 to this dictionary. |
| 10 | [dict.values()](https://www.javatpoint.com/python-dictionary-values-method) | It returns all the values of the dictionary. |
| 11 | [len()](https://www.javatpoint.com/python-dictionary-len-method) |  |
| 12 | [popItem()](https://www.javatpoint.com/python-dictionary-popitem-method) |  |
| 13 | [pop()](https://www.javatpoint.com/python-dictionary-pop-method) |  |
| 14 | [count()](https://www.javatpoint.com/python-dictionary-count-method) |  |
| 15 | [index()](https://www.javatpoint.com/python-dictionary-index-method) |  |

Python Functions

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the python program.

In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

Python provide us various inbuilt functions like range() or print(). Although, the user can create its functions which can be called user-defined functions.

Advantage of functions in python

There are the following advantages of C functions.

* By using functions, we can avoid rewriting same logic/code again and again in a program.
* We can call python functions any number of times in a program and from any place in a program.
* We can track a large python program easily when it is divided into multiple functions.
* Reusability is the main achievement of python functions.
* However, Function calling is always overhead in a python program.

Creating a function

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

1. **def** my\_function():
2. function-suite
3. **return** <expression>

The function block is started with the colon (:) and all the same level block statements remain at the same indentation.

A function can accept any number of parameters that must be the same in the definition and function calling.

Function calling

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

A simple function that prints the message "Hello Word" is given below.

1. **def** hello\_world():
2. **print**("hello world")
4. hello\_world()

**Output:**

hello world

Parameters in function

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses. We can give any number of parameters, but we have to separate them with a comma.

Consider the following example which contains a function that accepts a string as the parameter and prints it.

Example 1

1. #defining the function
2. **def** func (name):
3. **print**("Hi ",name);
5. #calling the function
6. func("Ayush")

Example 2

1. #python function to calculate the sum of two variables
2. #defining the function
3. **def** sum (a,b):
4. **return** a+b;
6. #taking values from the user
7. a = int(input("Enter a: "))
8. b = int(input("Enter b: "))
10. #printing the sum of a and b
11. **print**("Sum = ",sum(a,b))

**Output:**

Enter a: 10

Enter b: 20

Sum = 30

Call by reference in Python

In python, all the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

However, there is an exception in the case of mutable objects since the changes made to the mutable objects like string do not revert to the original string rather, a new string object is made, and therefore the two different objects are printed.

Example 1 Passing Immutable Object (List)

1. #defining the function
2. **def** change\_list(list1):
3. list1.append(20);
4. list1.append(30);
5. **print**("list inside function = ",list1)
7. #defining the list
8. list1 = [10,30,40,50]
10. #calling the function
11. change\_list(list1);
12. **print**("list outside function = ",list1);

**Output:**

list inside function = [10, 30, 40, 50, 20, 30]

list outside function = [10, 30, 40, 50, 20, 30]

Example 2 Passing Mutable Object (String)

1. #defining the function
2. **def** change\_string (str):
3. str = str + " Hows you";
4. **print**("printing the string inside function :",str);
6. string1 = "Hi I am there"
8. #calling the function
9. change\_string(string1)
11. **print**("printing the string outside function :",string1)

**Output:**

printing the string inside function : Hi I am there Hows you

printing the string outside function : Hi I am there

Types of arguments

There may be several types of arguments which can be passed at the time of function calling.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Required Arguments

Till now, we have learned about function calling in python. However, we can provide the arguments at the time of function calling. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.

Consider the following example.

Example 1

1. #the argument name is the required argument to the function func
2. **def** func(name):
3. message = "Hi "+name;
4. **return** message;
5. name = input("Enter the name?")
6. **print**(func(name))

**Output:**

Enter the name?John

Hi John

Example 2

1. #the function simple\_interest accepts three arguments and returns the simple interest accordingly
2. **def** simple\_interest(p,t,r):
3. **return** (p\*t\*r)/100
4. p = float(input("Enter the principle amount? "))
5. r = float(input("Enter the rate of interest? "))
6. t = float(input("Enter the time in years? "))
7. **print**("Simple Interest: ",simple\_interest(p,r,t))

**Output:**

Enter the principle amount? 10000

Enter the rate of interest? 5

Enter the time in years? 2

Simple Interest: 1000.0

Example 3

1. #the function calculate returns the sum of two arguments a and b
2. **def** calculate(a,b):
3. **return** a+b
4. calculate(10) # this causes an error as we are missing a required arguments b.

**Output:**

TypeError: calculate() missing 1 required positional argument: 'b'

Keyword arguments

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Consider the following example.

Example 1

1. #function func is called with the name and message as the keyword arguments
2. **def** func(name,message):
3. **print**("printing the message with",name,"and ",message)
4. func(name = "John",message="hello") #name and message is copied with the values John and hello respectively

**Output:**

printing the message with John and hello

Example 2 providing the values in different order at the calling

1. #The function simple\_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case
2. **def** simple\_interest(p,t,r):
3. **return** (p\*t\*r)/100
4. **print**("Simple Interest: ",simple\_interest(t=10,r=10,p=1900))

**Output:**

Simple Interest: 1900.0

If we provide the different name of arguments at the time of function call, an error will be thrown.

Consider the following example.

Example 3

1. #The function simple\_interest(p, t, r) is called with the keyword arguments.
2. **def** simple\_interest(p,t,r):
3. **return** (p\*t\*r)/100
5. **print**("Simple Interest: ",simple\_interest(time=10,rate=10,principle=1900)) # doesn't find the exact match of the name of the arguments (keywords)

**Output:**

TypeError: simple\_interest() got an unexpected keyword argument 'time'

The python allows us to provide the mix of the required arguments and keyword arguments at the time of function call. However, the required argument must not be given after the keyword argument, i.e., once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.

Consider the following example.

Example 4

1. **def** func(name1,message,name2):
2. **print**("printing the message with",name1,",",message,",and",name2)
3. func("John",message="hello",name2="David") #the first argument is not the keyword argument

**Output:**

printing the message with John , hello ,and David

The following example will cause an error due to an in-proper mix of keyword and required arguments being passed in the function call.

Example 5

1. **def** func(name1,message,name2):
2. **print**("printing the message with",name1,",",message,",and",name2)
3. func("John",message="hello","David")

**Output:**

SyntaxError: positional argument follows keyword argument

Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

Example 1

1. **def** printme(name,age=22):
2. **print**("My name is",name,"and age is",age)
3. printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function

**Output:**

My name is john and age is 22

Example 2

1. **def** printme(name,age=22):
2. **print**("My name is",name,"and age is",age)
3. printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function
4. printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age

**Output:**

My name is john and age is 22

My name is David and age is 10

Variable length Arguments

In the large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.

However, at the function definition, we have to define the variable with \* (star) as \*<variable - name >.

Consider the following example.

Example

1. **def** printme(\*names):
2. **print**("type of passed argument is ",type(names))
3. **print**("printing the passed arguments...")
4. **for** name **in** names:
5. **print**(name)
6. printme("john","David","smith","nick")

**Output:**

type of passed argument is <class 'tuple'>

printing the passed arguments...

john

David

smith

nick

Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope whereas the variable defined inside a function is known to have a local scope.

Consider the following example.

Example 1

1. **def** print\_message():
2. message = "hello !! I am going to print a message." # the variable message is local to the function itself
3. **print**(message)
4. print\_message()
5. **print**(message) # this will cause an error since a local variable cannot be accessible here.

**Output:**

hello !! I am going to print a message.

File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in

print(message)

NameError: name 'message' is not defined

Example 2

1. **def** calculate(\*args):
2. sum=0
3. **for** arg **in** args:
4. sum = sum +arg
5. **print**("The sum is",sum)
6. sum=0
7. calculate(10,20,30) #60 will be printed as the sum
8. **print**("Value of sum outside the function:",sum) # 0 will be printed

**Output:**

The sum is 60

Value of sum outside the function: 0

Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

**Python abs() Function Example**

1. #  integer number
2. integer = -20
3. **print**('Absolute value of -40 is:', abs(integer))
5. #  floating number
6. floating = -20.83
7. **print**('Absolute value of -40.83 is:', abs(floating))

**Output:**

Absolute value of -20 is: 20

Absolute value of -20.83 is: 20.83

Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

**Python all() Function Example**

1. # all values true
2. k = [1, 3, 4, 6]
3. **print**(all(k))
5. # all values false
6. k = [0, False]
7. **print**(all(k))
9. # one false value
10. k = [1, 3, 7, 0]
11. **print**(all(k))
13. # one true value
14. k = [0, False, 5]
15. **print**(all(k))
17. # empty iterable
18. k = []
19. **print**(all(k))

**Output:**

True

False

False

False

True

Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

**Python bin() Function Example**

1. x =  10
2. y =  bin(x)
3. **print** (y)

**Output:**

0b1010

Python bool()

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

**Python bool() Example**

1. test1 = []
2. **print**(test1,'is',bool(test1))
3. test1 = [0]
4. **print**(test1,'is',bool(test1))
5. test1 = 0.0
6. **print**(test1,'is',bool(test1))
7. test1 = None
8. **print**(test1,'is',bool(test1))
9. test1 = True
10. **print**(test1,'is',bool(test1))
11. test1 = 'Easy string'
12. **print**(test1,'is',bool(test1))

**Output:**

[] is False

[0] is True

0.0 is False

None is False

True is True

Easy string is True

Python bytes()

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

**Python bytes() Example**

1. string = "Hello World."
2. array = bytes(string, 'utf-8')
3. **print**(array)

**Output:**

b ' Hello World.'

Python callable() Function

A python **callable()** function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

**Python callable() Function Example**

1. x = 8
2. **print**(callable(x))

**Output:**

False

Python compile() Function

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

**Python compile() Function Example**

1. # compile string source to code
2. code\_str = 'x=5\ny=10\nprint("sum =",x+y)'
3. code = compile(code\_str, 'sum.py', 'exec')
4. **print**(type(code))
5. **exec**(code)
6. **exec**(x)

**Output:**

<class 'code'>

sum = 15

Python exec() Function

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

**Python exec() Function Example**

1. x = 8
2. **exec**('print(x==8)')
3. **exec**('print(x+4)')

**Output:**

True

12

Python sum() Function

As the name says, python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

**Python sum() Function Example**

1. s = sum([1, 2,4 ])
2. **print**(s)
4. s = sum([1, 2, 4], 10)
5. **print**(s)

**Output:**

7

17

Python any() Function

The python **any()** function returns true if any item in an iterable is true. Otherwise, it returns False.

**Python any() Function Example**

1. l = [4, 3, 2, 0]
2. **print**(any(l))
4. l = [0, False]
5. **print**(any(l))
7. l = [0, False, 5]
8. **print**(any(l))
10. l = []
11. **print**(any(l))

**Output:**

True

False

True

False

Python ascii() Function

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

**Python ascii() Function Example**

1. normalText = 'Python is interesting'
2. **print**(ascii(normalText))
4. otherText = 'Pythön is interesting'
5. **print**(ascii(otherText))
7. **print**('Pyth\xf6n is interesting')

**Output:**

'Python is interesting'

'Pyth\xf6n is interesting'

Pythön is interesting

Python bytearray()

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

**Python bytearray() Example**

1. string = "Python is a programming language."
3. # string with encoding 'utf-8'
4. arr = bytearray(string, 'utf-8')
5. **print**(arr)

**Output:**

bytearray(b'Python is a programming language.')

Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

**Python eval() Function Example**

1. x = 8
2. **print**(eval('x + 1'))

**Output:**

9

Python float()

The python **float()** function returns a floating-point number from a number or string.

**Python float() Example**

1. # for integers
2. **print**(float(9))
4. # for floats
5. **print**(float(8.19))
7. # for string floats
8. **print**(float("-24.27"))
10. # for string floats with whitespaces
11. **print**(float("     -17.19\n"))
13. # string float error
14. **print**(float("xyz"))

**Output:**

9.0

8.19

-24.27

-17.19

ValueError: could not convert string to float: 'xyz'

Python format() Function

The python **format()** function returns a formatted representation of the given value.

**Python format() Function Example**

1. # d, f and b are a type
3. # integer
4. **print**(format(123, "d"))
6. # float arguments
7. **print**(format(123.4567898, "f"))
9. # binary format
10. **print**(format(12, "b"))

**Output:**

123

123.456790

1100

Python frozenset()

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

**Python frozenset() Example**

1. # tuple of letters
2. letters = ('m', 'r', 'o', 't', 's')
4. fSet = frozenset(letters)
5. **print**('Frozen set is:', fSet)
6. **print**('Empty frozen set is:', frozenset())

**Output:**

Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})

Empty frozen set is: frozenset()

Python getattr() Function

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

**Python getattr() Function Example**

1. **class** Details:
2. age = 22
3. name = "Phill"
5. details = Details()
6. **print**('The age is:', getattr(details, "age"))
7. **print**('The age is:', details.age)

**Output:**

The age is: 22

The age is: 22

Python globals() Function

The python **globals()** function returns the dictionary of the current global symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

**Python globals() Function Example**

1. age = 22
3. globals()['age'] = 22
4. **print**('The age is:', age)

**Output:**

The age is: 22

Python hasattr() Function

The python **any()** function returns true if any item in an iterable is true, otherwise it returns False.

**Python hasattr() Function Example**

1. l = [4, 3, 2, 0]
2. **print**(any(l))
4. l = [0, False]
5. **print**(any(l))
7. l = [0, False, 5]
8. **print**(any(l))
10. l = []
11. **print**(any(l))

**Output:**

True

False

True

False

Python iter() Function

The python **iter()** function is used to return an iterator object. It creates an object which can be iterated one element at a time.

**Python iter() Function Example**

1. # list of numbers
2. list = [1,2,3,4,5]
4. listIter = iter(list)
6. # prints '1'
7. **print**(next(listIter))
9. # prints '2'
10. **print**(next(listIter))
12. # prints '3'
13. **print**(next(listIter))
15. # prints '4'
16. **print**(next(listIter))
18. # prints '5'
19. **print**(next(listIter))

**Output:**

1

2

3

4

5

Python len() Function

The python **len()** function is used to return the length (the number of items) of an object.

**Python len() Function Example**

1. strA = 'Python'
2. **print**(len(strA))

**Output:**

6

Python list()

The python **list()** creates a list in python.

**Python list() Example**

1. # empty list
2. **print**(list())
4. # string
5. String = 'abcde'
6. **print**(list(String))
8. # tuple
9. Tuple = (1,2,3,4,5)
10. **print**(list(Tuple))
11. # list
12. List = [1,2,3,4,5]
13. **print**(list(List))

**Output:**

[]

['a', 'b', 'c', 'd', 'e']

[1,2,3,4,5]

[1,2,3,4,5]

Python locals() Function

The python **locals()** method updates and returns the dictionary of the current local symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

**Python locals() Function Example**

1. **def** localsAbsent():
2. **return** locals()
4. **def** localsPresent():
5. present = True
6. **return** locals()
8. **print**('localsNotPresent:', localsAbsent())
9. **print**('localsPresent:', localsPresent())

**Output:**

localsAbsent: {}

localsPresent: {'present': True}

Python map() Function

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.).

**Python map() Function Example**

1. **def** calculateAddition(n):
2. **return** n+n
4. numbers = (1, 2, 3, 4)
5. result = map(calculateAddition, numbers)
6. **print**(result)
8. # converting map object to set
9. numbersAddition = set(result)
10. **print**(numbersAddition)

**Output:**

<map object at 0x7fb04a6bec18>

{8, 2, 4, 6}

Python memoryview() Function

The python **memoryview()** function returns a memoryview object of the given argument.

**Python memoryview () Function Example**

1. #A random bytearray
2. randomByteArray = bytearray('ABC', 'utf-8')
4. mv = memoryview(randomByteArray)
6. # access the memory view's zeroth index
7. **print**(mv[0])
9. # It create byte from memory view
10. **print**(bytes(mv[0:2]))
12. # It create list from memory view
13. **print**(list(mv[0:3]))

**Output:**

65

b'AB'

[65, 66, 67]

Python object()

The python **object()** returns an empty object. It is a base for all the classes and holds the built-in properties and methods which are default for all the classes.

**Python object() Example**

1. python = object()
3. **print**(type(python))
4. **print**(dir(python))

**Output:**

<class 'object'>

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_',

'\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_',

'\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_',

'\_\_str\_\_', '\_\_subclasshook\_\_']

Python open() Function

The python **open()** function opens the file and returns a corresponding file object.

**Python open() Function Example**

1. # opens python.text file of the current directory
2. f = open("python.txt")
3. # specifying full path
4. f = open("C:/Python33/README.txt")

**Output:**

Since the mode is omitted, the file is opened in 'r' mode; opens for reading.

Python chr() Function

Python **chr()** function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

**Python chr() Function Example**

1. # Calling function
2. result = chr(102) # It returns string representation of a char
3. result2 = chr(112)
4. # Displaying result
5. **print**(result)
6. **print**(result2)
7. # Verify, is it string type?
8. **print**("is it string type:", type(result) **is** str)

**Output:**

ValueError: chr() arg not in range(0x110000)

Python complex()

Python **complex()** function is used to convert numbers or string into a complex number. This method takes two optional parameters and returns a complex number. The first parameter is called a real and second as imaginary parts.

**Python complex() Example**

1. # Python complex() function example
2. # Calling function
3. a = complex(1) # Passing single parameter
4. b = complex(1,2) # Passing both parameters
5. # Displaying result
6. **print**(a)
7. **print**(b)

**Output:**

(1.5+0j)

(1.5+2.2j)

Python delattr() Function

Python **delattr()** function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

**Python delattr() Function Example**

1. **class** Student:
2. id = 101
3. name = "Pranshu"
4. email = "pranshu@abc.com"
5. # Declaring function
6. **def** getinfo(self):
7. **print**(self.id, self.name, self.email)
8. s = Student()
9. s.getinfo()
10. delattr(Student,'course') # Removing attribute which is not available
11. s.getinfo() # error: throws an error

**Output:**

101 Pranshu pranshu@abc.com

AttributeError: course

Python dir() Function

Python **dir()** function returns the list of names in the current local scope. If the object on which method is called has a method named \_\_dir\_\_(), this method will be called and must return the list of attributes. It takes a single object type argument.

**Python dir() Function Example**

1. # Calling function
2. att = dir()
3. # Displaying result
4. **print**(att)

**Output:**

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_',

'\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_']

Python divmod() Function

Python **divmod()** function is used to get remainder and quotient of two numbers. This function takes two numeric arguments and returns a tuple. Both arguments are required and numeric

**Python divmod() Function Example**

1. # Python divmod() function example
2. # Calling function
3. result = divmod(10,2)
4. # Displaying result
5. **print**(result)

**Output:**

(5, 0)

Python enumerate() Function

Python **enumerate()** function returns an enumerated object. It takes two parameters, first is a sequence of elements and the second is the start index of the sequence. We can get the elements in sequence either through a loop or next() method.

**Python enumerate() Function Example**

1. # Calling function
2. result = enumerate([1,2,3])
3. # Displaying result
4. **print**(result)
5. **print**(list(result))

**Output:**

<enumerate object at 0x7ff641093d80>

[(0, 1), (1, 2), (2, 3)]

Python dict()

Python **dict()** function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

* If no argument is passed, it creates an empty dictionary.
* If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.
* If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

**Python dict() Example**

1. # Calling function
2. result = dict() # returns an empty dictionary
3. result2 = dict(a=1,b=2)
4. # Displaying result
5. **print**(result)
6. **print**(result2)

**Output:**

{}

{'a': 1, 'b': 2}

Python filter() Function

Python **filter()** function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns **true value**.

The first argument can be **none**, if the function is not available and returns only elements that are **true**.

**Python filter() Function Example**

1. # Python filter() function example
2. **def** filterdata(x):
3. **if** x>5:
4. **return** x
5. # Calling function
6. result = filter(filterdata,(1,2,6))
7. # Displaying result
8. **print**(list(result))

**Output:**

[6]

Python hash() Function

Python **hash()** function is used to get the hash value of an object. Python calculates the hash value by using the hash algorithm. The hash values are integers and used to compare dictionary keys during a dictionary lookup. We can hash only the types which are given below:

**Hashable types:** \* bool \* int \* long \* float \* string \* Unicode \* tuple \* code object.

**Python hash() Function Example**

1. # Calling function
2. result = hash(21) # integer value
3. result2 = hash(22.2) # decimal value
4. # Displaying result
5. **print**(result)
6. **print**(result2)

**Output:**

21

461168601842737174

Python help() Function

Python **help()** function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

**Python help() Function Example**

1. # Calling function
2. info = help() # No argument
3. # Displaying result
4. **print**(info)

**Output:**

Welcome to Python 3.5's help utility!

Python min() Function

Python **min()** function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

**Python min() Function Example**

1. # Calling function
2. small = min(2225,325,2025) # returns smallest element
3. small2 = min(1000.25,2025.35,5625.36,10052.50)
4. # Displaying result
5. **print**(small)
6. **print**(small2)

**Output:**

325

1000.25

Python set() Function

In python, a set is a built-in class, and this function is a constructor of this class. It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

**Python set() Function Example**

1. # Calling function
2. result = set() # empty set
3. result2 = set('12')
4. result3 = set('javatpoint')
5. # Displaying result
6. **print**(result)
7. **print**(result2)
8. **print**(result3)

**Output:**

set()

{'1', '2'}

{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}

Python hex() Function

Python **hex()** function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

**Python hex() Function Example**

1. # Calling function
2. result = hex(1)
3. # integer value
4. result2 = hex(342)
5. # Displaying result
6. **print**(result)
7. **print**(result2)

**Output:**

0x1

0x156

Python id() Function

Python **id()** function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

**Python id() Function Example**

1. # Calling function
2. val = id("Javatpoint") # string object
3. val2 = id(1200) # integer object
4. val3 = id([25,336,95,236,92,3225]) # List object
5. # Displaying result
6. **print**(val)
7. **print**(val2)
8. **print**(val3)

**Output:**

139963782059696

139963805666864

139963781994504

Python setattr() Function

Python **setattr()** function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

**Python setattr() Function Example**

1. **class** Student:
2. id = 0
3. name = ""
5. **def** \_\_init\_\_(self, id, name):
6. self.id = id
7. self.name = name
9. student = Student(102,"Sohan")
10. **print**(student.id)
11. **print**(student.name)
12. #print(student.email) product error
13. setattr(student, 'email','sohan@abc.com') # adding new attribute
14. **print**(student.email)

**Output:**

102

Sohan

sohan@abc.com

Python slice() Function

Python **slice()** function is used to get a slice of elements from the collection of elements. Python provides two overloaded slice functions. The first function takes a single argument while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection.

**Python slice() Function Example**

1. # Calling function
2. result = slice(5) # returns slice object
3. result2 = slice(0,5,3) # returns slice object
4. # Displaying result
5. **print**(result)
6. **print**(result2)

**Output:**

slice(None, 5, None)

slice(0, 5, 3)

Python sorted() Function

Python **sorted()** function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In the case of a dictionary, it sorts only keys, not values.

**Python sorted() Function Example**

1. str = "javatpoint" # declaring string
2. # Calling function
3. sorted1 = sorted(str) # sorting string
4. # Displaying result
5. **print**(sorted1)

**Output:**

['a', 'a', 'i', 'j', 'n', 'o', 'p', 't', 't', 'v']

Python next() Function

Python **next()** function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

**Python next() Function Example**

1. number = iter([256, 32, 82]) # Creating iterator
2. # Calling function
3. item = next(number)
4. # Displaying result
5. **print**(item)
6. # second item
7. item = next(number)
8. **print**(item)
9. # third item
10. item = next(number)
11. **print**(item)

**Output:**

256

32

82

Python input() Function

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

**Python input() Function Example**

1. # Calling function
2. val = input("Enter a value: ")
3. # Displaying result
4. **print**("You entered:",val)

**Output:**

Enter a value: 45

You entered: 45

Python int() Function

Python **int()** function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.

If the number is not a number or if a base is given, the number must be a string.

**Python int() Function Example**

1. # Calling function
2. val = int(10) # integer value
3. val2 = int(10.52) # float value
4. val3 = int('10') # string value
5. # Displaying result
6. **print**("integer values :",val, val2, val3)

**Output:**

integer values : 10 10 10

Python isinstance() Function

Python **isinstance()** function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The **isinstance()** function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

**Python isinstance() function Example**

1. **class** Student:
2. id = 101
3. name = "John"
4. **def** \_\_init\_\_(self, id, name):
5. self.id=id
6. self.name=name
8. student = Student(1010,"John")
9. lst = [12,34,5,6,767]
10. # Calling function
11. **print**(isinstance(student, Student)) # isinstance of Student class
12. **print**(isinstance(lst, Student))

**Output:**

True

False

Python oct() Function

Python **oct()** function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error **TypeError**, if argument type is other than an integer.

**Python oct() function Example**

1. # Calling function
2. val = oct(10)
3. # Displaying result
4. **print**("Octal value of 10:",val)

**Output:**

Octal value of 10: 0o12

Python ord() Function

The python **ord()** function returns an integer representing Unicode code point for the given Unicode character.

**Python ord() function Example**

1. # Code point of an integer
2. **print**(ord('8'))
4. # Code point of an alphabet
5. **print**(ord('R'))
7. # Code point of a character
8. **print**(ord('&'))

**Output:**

56

82

38

Python pow() Function

The python **pow()** function is used to compute the power of a number. It returns x to the power of y. If the third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

**Python pow() function Example**

1. # positive x, positive y (x\*\*y)
2. **print**(pow(4, 2))
4. # negative x, positive y
5. **print**(pow(-4, 2))
7. # positive x, negative y (x\*\*-y)
8. **print**(pow(4, -2))
10. # negative x, negative y
11. **print**(pow(-4, -2))

**Output:**

16

16

0.0625

0.0625

Python print() Function

The python **print()** function prints the given object to the screen or other standard output devices.

**Python print() function Example**

1. **print**("Python is programming language.")
3. x = 7
4. # Two objects passed
5. **print**("x =", x)
7. y = x
8. # Three objects passed
9. **print**('x =', x, '= y')

**Output:**

Python is programming language.

x = 7

x = 7 = y

Python range() Function

The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

**Python range() function Example**

1. # empty range
2. **print**(list(range(0)))
4. # using the range(stop)
5. **print**(list(range(4)))
7. # using the range(start, stop)
8. **print**(list(range(1,7 )))

**Output:**

[]

[0, 1, 2, 3]

[1, 2, 3, 4, 5, 6]

Python reversed() Function

The python **reversed()** function returns the reversed iterator of the given sequence.

**Python reversed() function Example**

1. # for string
2. String = 'Java'
3. **print**(list(reversed(String)))
5. # for tuple
6. Tuple = ('J', 'a', 'v', 'a')
7. **print**(list(reversed(Tuple)))
9. # for range
10. Range = range(8, 12)
11. **print**(list(reversed(Range)))
13. # for list
14. List = [1, 2, 7, 5]
15. **print**(list(reversed(List)))

**Output:**

['a', 'v', 'a', 'J']

['a', 'v', 'a', 'J']

[11, 10, 9, 8]

[5, 7, 2, 1]

Python round() Function

The python **round()** function rounds off the digits of a number and returns the floating point number.

**Python round() Function Example**

1. #  for integers
2. **print**(round(10))
4. #  for floating point
5. **print**(round(10.8))
7. #  even choice
8. **print**(round(6.6))

**Output:**

10

11

7

Python issubclass() Function

The python **issubclass()** function returns true if object argument(first argument) is a subclass of second class(second argument).

**Python issubclass() Function Example**

1. **class** Rectangle:
2. **def** \_\_init\_\_(rectangleType):
3. **print**('Rectangle is a ', rectangleType)
5. **class** Square(Rectangle):
6. **def** \_\_init\_\_(self):
7. Rectangle.\_\_init\_\_('square')
9. **print**(issubclass(Square, Rectangle))
10. **print**(issubclass(Square, list))
11. **print**(issubclass(Square, (list, Rectangle)))
12. **print**(issubclass(Rectangle, (list, Rectangle)))

**Output:**

True

False

True

True

Python str

The python **str()** converts a specified value into a string.

**Python str() Function Example**

1. str('4')

**Output:**

'4'

Python tuple() Function

The python **tuple()** function is used to create a tuple object.

**Python tuple() Function Example**

1. t1 = tuple()
2. **print**('t1=', t1)
4. # creating a tuple from a list
5. t2 = tuple([1, 6, 9])
6. **print**('t2=', t2)
8. # creating a tuple from a string
9. t1 = tuple('Java')
10. **print**('t1=',t1)
12. # creating a tuple from a dictionary
13. t1 = tuple({4: 'four', 5: 'five'})
14. **print**('t1=',t1)

**Output:**

t1= ()

t2= (1, 6, 9)

t1= ('J', 'a', 'v', 'a')

t1= (4, 5)

Python type()

The python **type()** returns the type of the specified object if a single argument is passed to the type() built in function. If three arguments are passed, then it returns a new type object.

**Python type() Function Example**

1. List = [4, 5]
2. **print**(type(List))
4. Dict = {4: 'four', 5: 'five'}
5. **print**(type(Dict))
7. **class** Python:
8. a = 0
10. InstanceOfPython = Python()
11. **print**(type(InstanceOfPython))

**Output:**

<class 'list'>

<class 'dict'>

<class '\_\_main\_\_.Python'>

Python vars() function

The python **vars()** function returns the \_\_dict\_\_ attribute of the given object.

**Python vars() Function Example**

1. **class** Python:
2. **def** \_\_init\_\_(self, x = 7, y = 9):
3. self.x = x
4. self.y = y
6. InstanceOfPython = Python()
7. **print**(vars(InstanceOfPython))

**Output:**

{'y': 9, 'x': 7}

Python zip() Function

The python **zip()** Function returns a zip object, which maps a similar index of multiple containers. It takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.

**Python zip() Function Example**

1. numList = [4,5, 6]
2. strList = ['four', 'five', 'six']
4. # No iterables are passed
5. result = zip()
7. # Converting itertor to list
8. resultList = list(result)
9. **print**(resultList)
11. # Two iterables are passed
12. result = zip(numList, strList)
14. # Converting itertor to set
15. resultSet = set(result)
16. **print**(resultSet)

**Output:**

[]

{(5, 'five'), (4, 'four'), (6, 'six')}

Python Lambda Functions

Python allows us to not declare the function in the standard manner, i.e., by using the def keyword. Rather, the anonymous functions are declared by using lambda keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

The syntax to define an Anonymous function is given below.

1. **lambda** arguments : expression

Example 1

1. x = **lambda** a:a+10 # a is an argument and a+10 is an expression which got evaluated and returned.
2. **print**("sum = ",x(20))

**Output:**

sum = 30

Example 2

Multiple arguments to Lambda function

1. x = **lambda** a,b:a+b # a and b are the arguments and a+b is the expression which gets evaluated and returned.
2. **print**("sum = ",x(20,10))

**Output:**

sum = 30

Why use lambda functions?

The main role of the lambda function is better described in the scenarios when we use them anonymously inside another function. In python, the lambda function can be used as an argument to the higher order functions as arguments. Lambda functions are also used in the scenario where we need a Consider the following example.

Example 1

1. #the function table(n) prints the table of n
2. **def** table(n):
3. **return** **lambda** a:a\*n; # a will contain the iteration variable i and a multiple of n is returned at each function call
4. n = int(input("Enter the number?"))
5. b = table(n) #the entered number is passed into the function table. b will contain a lambda function which is called again and again with the iteration variable i
6. **for** i **in** range(1,11):
7. **print**(n,"X",i,"=",b(i)); #the lambda function b is called with the iteration variable i,

**Output:**

Enter the number?10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

Example 2

Use of lambda function with filter

1. #program to filter out the list which contains odd numbers
2. List = {1,2,3,4,10,123,22}
3. Oddlist = list(filter(**lambda** x:(x%3 == 0),List)) # the list contains all the items of the list for which the lambda function evaluates to true
4. **print**(Oddlist)

**Output:**

[3, 123]

Example 3

Use of lambda function with map

1. #program to triple each number of the list using map
2. List = {1,2,3,4,10,123,22}
3. new\_list = list(map(**lambda** x:x\*3,List)) # this will return the triple of each item of the list and add it to new\_list
4. **print**(new\_list)

**Output:**

[3, 6, 9, 12, 30, 66, 369]

(factorial 4)

(4 \* (factorial 3))

(4 \* (3 \* (factorial 3)))

(4 \* (3 \* (2 \* (factorial 1))))

(4 \* (3 \* (2 \* 1)))

(4 \* (3 \* 2))

(4 \* 6)

(24)

def factorial(n):

if n == 0:

return 1

else:

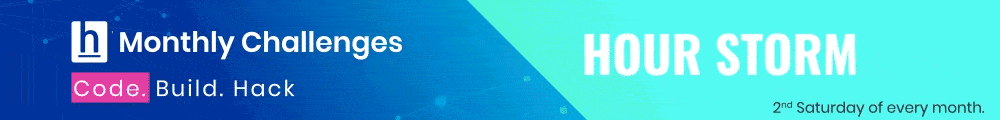
return n \* factorial(n - 1)

## What is the game of Tower of Hanoi?

Tower of Hanoi consists of three pegs or towers with n disks placed one over the other.

The objective of the puzzle is to move the stack to another peg following these simple rules.

1. Only one disk can be moved at a time.
2. No disk can be placed on top of the smaller disk.

[](https://www.hackerearth.com/challenges/?utm_source=blog&utm_medium=strip)

Before we proceed, let’s understand Recursion –

### **What is Recursion?**

When a function calls itself, it’s called Recursion.

It will be easier for those who have seen the movie Inception.

Leonardo had a dream, in that dream he had another dream, in that dream he had yet another dream, and that goes on.

So it’s like there is a function called dream()dream(), and we are just calling it in itself.



|  |  |
| --- | --- |
| 1  2  3 | function dream()      print "Dreaming"      dream() |

Recursion is useful in solving problems which can be broken down into smaller problems of the **same kind**.

But when it comes to solving problems using Recursion there are several things to be taken care of.

Let’s take a simple example and try to understand those.

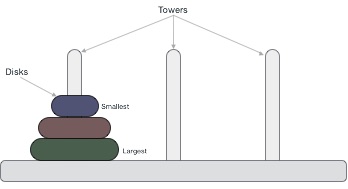
Following is the pseudo code of finding the factorial of a given number XX using recursion.



|  |  |
| --- | --- |
| 1  2  3  4 | function factorial(x)      if x is 0                    // base case          return 1      return x\*factorial(x-1)       // break into smaller problem(s) |

Detailed explanation to Recursion can be found

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



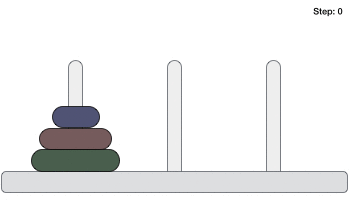
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

* Only one disk can be moved among the towers at any given time.
* Only the "top" disk can be removed.
* No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



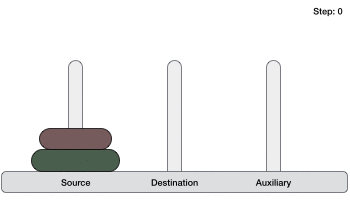
Tower of Hanoi puzzle with n disks can be solved in minimum **2n−1** steps. This presentation shows that a puzzle with 3 disks has taken **23 - 1 = 7** steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination**and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

* First, we move the smaller (top) disk to aux peg.
* Then, we move the larger (bottom) disk to destination peg.
* And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

**Step 1** − Move n-1 disks from **source** to **aux**

**Step 2** − Move nth disk from **source** to **dest**

**Step 3** − Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows −

START

Procedure Hanoi(disk, source, dest, aux)

IF disk == 1, THEN

move disk from source to dest

ELSE

Hanoi(disk - 1, source, aux, dest) // Step 1

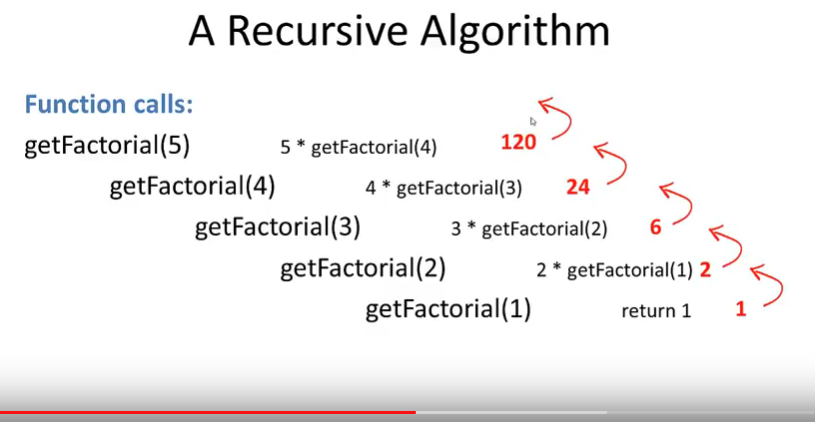
move disk from source to dest // Step 2

Hanoi(disk - 1, aux, dest, source) // Step 3

END IF

END Procedure

STOP



We can track how the function works by adding two print() functions to the previous function definition:

def factorial(n):

print("factorial has been called with n = " + str(n))

if n == 1:

return 1

else:

res = n \* factorial(n-1)

print("intermediate result for ", n, " \* factorial(" ,n-1, "): ",res)

return res

print(factorial(5))

This Python script outputs the following results:

factorial has been called with n = 5

factorial has been called with n = 4

factorial has been called with n = 3

factorial has been called with n = 2

factorial has been called with n = 1

intermediate result for 2 \* factorial( 1 ): 2

intermediate result for 3 \* factorial( 2 ): 6

intermediate result for 4 \* factorial( 3 ): 24

intermediate result for 5 \* factorial( 4 ): 120

120

The Fibonacci numbers are the numbers of the following sequence of integer values:   
  
0,1,1,2,3,5,8,13,21,34,55,89, ...   
  
The Fibonacci numbers are defined by:   
Fn = Fn-1 + Fn-2   
with F0 = 0 and F1 = 1

he Fibonacci numbers are easy to write as a Python function. It's more or less a one to one mapping from the mathematical definition:

def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return fib(n-1) + fib(n-2)

1. Write a recursive Python function that returns the sum of the first n integers.   
   (Hint: The function will be similiar to the factorial function!)
2. def sum\_n(n):
3. if n== 0:
4. return 0
5. else:
6. return n + sum\_n(n-1)
7. Now try to execute the above function like this
8. 

|  |  |
| --- | --- |
| 1 | print(fact(2000)) |

1. You will get
2. 

|  |  |
| --- | --- |
| 1 | RuntimeError: maximum recursion depth exceeded in comparison |

1. This happens because python stop calling recursive function after 1000 calls by default. To change this behavior you need to amend the code as follows.
2. 

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import sys  sys.setrecursionlimit(3000)    def fact(n):      if n == 0:          return 1      else:          return n \* fact(n-1)      print(fact(2000)) |

# Recursion in Python

A function that calls itself is a recursive function. This method is used when a certain problem is defined in terms of itself.

In English there are many examples of recursion:

* "To understand recursion, you must first understand recursion",
* "A human is someone whose mother is human".

In Python, **a function is recursive if it calls itself and has a termination condition**. Why a termination condition? To stop the function from calling itself ad infinity.

### What is Recursion?

As stated in the introduction, [recursion](https://en.wikipedia.org/wiki/Recursion) involves a process calling itself in the definition. A recursive function generally has two components:

* The **base case** which is a condition that determines when the recursive function should stop
* The call to itself

Let's take a look at a small example to demonstrate both components:

Do you see the recursion in this definition itself? They used the word ‘recursive’ to define ‘recursion’. We sense an Easter egg here. Anyway, so as we talk about recursion, we’ll take the coolest example first. Take a look at the logo for PyPy, an implementation of Python with a Just-In-Time Compiler.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2018/01/pypy-logo.png)

*Python Recursion – pypy*

The snake biting its own tail, feeding itself, is an example of recursion we’d like to give to you.

To take a more general example, when our anxiety creates more anxiety for us, it is recursion.

In programming, recursion is when a function calls itself. We’ll see this in detail in the following sections of recursion in Python Example.

First, let’s do it without Python recursion function.

1. >>> def **factorial**(n):
2. f=1
3. while n>0:
4. f\*=n
5. n-=1
6. **print**(f)
7. >>> **factorial**(4)

24

1. >>> **factorial**(5)

120

Now, let’s implement this with recursion. We mean to make factorial() call factorial().

1. >>> def **factorial**(n):
2. if n==1:
3. return 1
4. return n\***factorial**(n-1)
5. >>> **factorial**(5)

120

1. >>> **factorial**(4)

24

### **a. Python Recursion Function Advantages**

With Python recursion, there are some benefits we observe:

1. A recursive code has a cleaner-looking code.
2. Recursion makes it easier to code, as it breaks a task into smaller ones.
3. It is easier to generate a sequence using recursion than by using nested iteration.

### **b. Python Recursion Function Disadvantages**

The flip side of the coin is easy to quote:

1. Although it makes code look cleaner, it may sometimes be hard to follow.
2. They may be simpler, but recursive calls are expensive. They take up a lot of memory and time.
3. Finally, it isn’t as easy to debug a recursive function.

First, let’s define a function to calculate the sum of the first n natural numbers.

1. >>> def **sumofn**(n):
2. if n==1:
3. return 1
4. return n+**sumofn**(n-1)
5. >>> **sumofn**(16)
6. When a function calls itself, its called Recursion. It will be easier for those who have seen the movie Inception. Leonardo had a dream, in that dream he had another dream, in that dream he had yet another dream, and that goes on. So it's like there is a function called dream(), and we are just calling it in itself.
7. function dream()
8. print "Dreaming"
9. dream()

Recursion is useful in solving problems which can be broken down into smaller problems of the **same kind**. But when it comes to solving problems using Recursion there are several things to be taken care of. Let's take a simple example and try to understand those. Following is the pseudo code of finding factorial of a given number X using recursion.

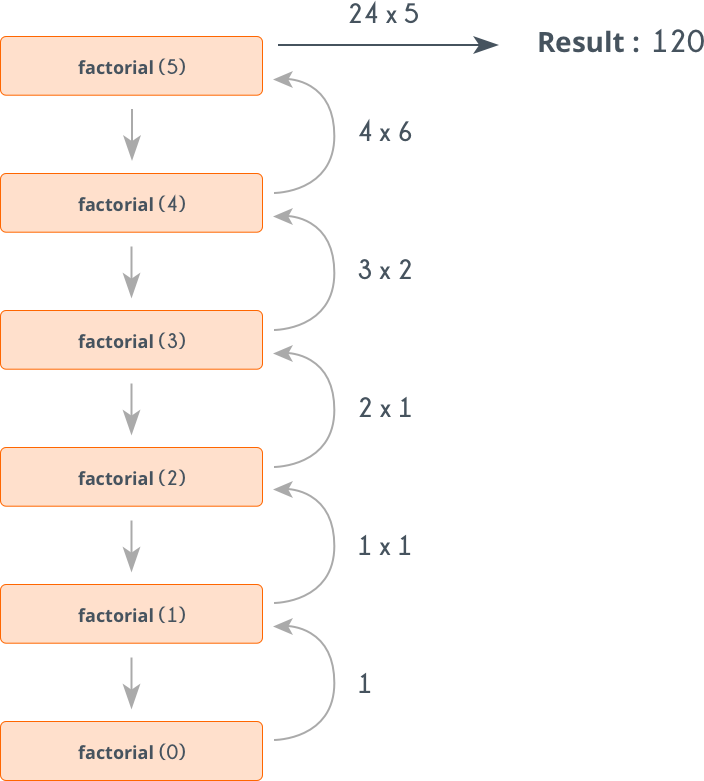
function factorial(x)

if x is 0 // base case

return 1

return x\*factorial(x-1) // break into smaller problem(s)

The following image shows how it works for factorial(5).



**Base Case:** Any recursive method must have a terminating condition. Terminating condition is one for which the answer is already known and we just need to return that. For example for the factorial problem, we know that factorial(0)=1, so when x is 0 we simply return 1, otherwise we break into smaller problem i.e. find factorial of x−1. If we don't include a Base Case, the function will keep calling itself, and ultimately will result in stack overflow. For example, the dream() function given above has no base case. If you write a code for it in any language, it will give a runtime error.

**Number of Recursive calls**: There is an upper limit to the number of recursive calls that can be made. To prevent this make sure that your base case is reached before stack size limit exceeds.

So, if we want to solve a problem using recursion, then we need to make sure that:

* The problem can broken down into smaller problems of **same type**.
* Problem has some base case(s).
* Base case is reached before the stack size limit exceeds.
* **def** recur\_factorial(n):
* **if** n == 1:
* **return** n
* **else**:
* **return** n\*recur\_factorial(n-1)
* # take input from the user
* num = int(input("Enter a number: "))
* # check is the number is negative
* **if** num < 0:
* **print**("Sorry, factorial does not exist for negative numbers")
* **elif** num == 0:
* **print**("The factorial of 0 is 1")
* **else**:
* **print**("The factorial of",num,"is",recur\_factorial(num))

# Introduction to Debugging

*“The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do.”  — Andrew Singer*

Programming is a complex process. Since it is done by human beings, errors may often occur. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**

One of the most important skills you need to acquire to complete this book successfully is the ability to debug your programs. Debugging might be the most under-appreciated, and under-taught, skill in introductory computer science. For that reason we are introducing a series of “debugging interludes.” Debugging is a skill that you need to master over time, and some of the tips and tricks are specific to different aspects of Python programming. So look for additional Way of the Programmer interludes throughout the rest of this book.

Programming is an odd thing in a way. Here is why. As programmers we spend 99% of our time trying to get our program to work. We struggle, we stress, we spend hours deep in frustration trying to get our program to execute correctly. Then when we do get it going we celebrate, hand it in, and move on to the next homework assignment or programming task. But here is the secret, when you are successful, you are happy, your brain releases a bit of chemical that makes you feel good. You need to organize your programming so that you have lots of little successess. It turns out your brain doesn’t care all that much if you have successfully written hello world, or a fast fourier transform (trust me its hard) you still get that little release that makes you happy. When you are happy you want to go on and solve the next little problem. Essentially I’m telling you once again, start small, get something small working, and then add to it.

## 3.2.1. How to Avoid Debugging

Perhaps the most important lesson in debugging is that it is **largely avoidable** – if you work carefully.

1. **Start Small** This is probably the single biggest piece of advice for programmers at every level. Of course it’s tempting to sit down and crank out an entire program at once. But, when the program – inevitably – does not work then you have a myriad of options for things that might be wrong. Where to start? Where to look first? How to figure out what went wrong? I’ll get to that in the next section. So, start with something really small. Maybe just two lines and then make sure that runs ok. Hitting the run button is quick and easy, and gives you immediate feedback about whether what you have just done is ok or not. Another immediate benefit of having something small working is that you have something to turn in. Turning in a small, incomplete program, is almost always better than nothing.

2. **Keep it working** Once you have a small part of your program working the next step is to figure out something small to add to it. If you keep adding small pieces of the program one at a time, it is much easier to figure out what went wrong, as it is most likely that the problem is going to be in the new code you have just added. Less new code means its easier to figure out where the problem is.

This notion of **Get something working and keep it working** is a mantra that you can repeat throughout your career as a programmer. It’s a great way to avoid the frustrations mentioned above. Think of it this way. Every time you have a little success, your brain releases a tiny bit of chemical that makes you happy. So, you can keep yourself happy and make programming more enjoyable by creating lots of small victories for yourself.

Ok, lets look at an example. Lets solve the problem posed in question 3 at the end of the Simple Python Data chapter. Ask the user for the time now (in hours 0 – 23), and ask for the number of hours to wait. Your program should output what the time will be on the clock when the alarm goes off.

So, where to start? The problem requires two pieces of input from the user, so lets start there and make sure we can get the data we need.

current\_time = input("what is the current time (in hours)?")

wait\_time = input("How many hours do you want to wait")

print(current\_time)

print(wait\_time)

So far so good. Now lets take the next step. We need to figure out what the time will be after waiting wait\_time number of hours. A good first approximation to that is to simply add wait\_time to current\_time and print out the result. So lets try that.

current\_time = input("what is the current time (in hours 0--23)?")

wait\_time = input("How many hours do you want to wait")

print(current\_time)

print(wait\_time)

final\_time = current\_time + wait\_time

print(final\_time)

current\_time\_str = input("what is the current time (in hours 0-23)?")

wait\_time\_str = input("How many hours do you want to wait")

current\_time\_int = int(current\_time\_str)

wait\_time\_int = int(wait\_time\_str)

final\_time\_int = current\_time\_int + wait\_time\_int

print(final\_time\_int)

Now, that’s a lot better, and in fact depending on the hours you chose, it may be exactly right. If you entered 8 for the current time and 5 for the wait time then 13 is correct. But if you entered 17 (5pm) for the hours and 9 for the wait time then the result of 26 is not correct. This illustrates an important aspect of **testing**, which is that it is important to test your code on a range of inputs. It is especially important to test your code on **boundary conditions**. In this case you would want to test your program for hours including 0, 23, and some in between. You would want to test your wait times for 0, and some really large numbers. What about negative numbers? Negative numbers don’t make sense, but since we don’t really have the tools to deal with telling the user when something is wrong we will not worry about that just yet.

So finally we need to account for those numbers that are bigger than 23. For this we will need one final step, using the modulo operator.

current\_time\_str = input("what is the current time (in hours 0-23)?")

wait\_time\_str = input("How many hours do you want to wait")

current\_time\_int = int(current\_time\_str)

wait\_time\_int = int(wait\_time\_str)

final\_time\_int = current\_time\_int + wait\_time\_int

final\_answer = final\_time\_int % 24

print("The time after waiting is: ", final\_answer)

✔️ Programming errors are called bugs and the process of finding and removing them from a program is called debugging.

# Beginning tips for Debugging

# 3.3. 👩‍💻 Beginning tips for Debugging

Debugging a program is a different way of thinking than writing a program. The process of debugging is much more like being a detective. Here are a few rules to get you thinking about debugging.

1. Everyone is a suspect (Except Python)! It’s common for beginner programmers to blame Python, but that should be your last resort. Remember that Python has been used to solve CS1 level problems millions of times by millions of other programmers. So, Python is probably not the problem.
2. Check your assumptions. At this point in your career you are still developing your mental model of how Python does its work. Its natural to think that your code is correct, but with debugging you need to make your code the primary suspect. Even ifyou think it is right, you should verify that it really is by liberally using print statements to verify that the values of variables really are what you think they should be. You’ll be surprised how often they are not.
3. Find clues. This is the biggest job of the detective and right now there are two important kinds of clues for you to understand.
   * Error Messages
   * Print Statements

Three kinds of errors can occur in a program: [syntax errors](http://en.wikipedia.org/wiki/Syntax_error), [runtime errors](http://en.wikipedia.org/wiki/Runtime_error), and [semantic errors](http://en.wikipedia.org/wiki/Logic_error). It is useful to distinguish between them in order to track them down more quickly.

# 3.4. Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit. You will not be able to complete the execution of your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. However, as you gain experience, you will make fewer errors and you will also be able to find your errors faster.

Can you spot the syntax error in the code below?

print("Hello World!"

**Check your understanding**

debug-4-1: Which of the following is a syntax error?

Top of Form

A. Attempting to divide by 0.  
B. Forgetting a colon at the end of a statement where one is required.  
C. Forgetting to divide by 100 when printing a percentage amount.  
Check MeCompare me

Bottom of Form

✔️ This is a problem with the formal structure of the program. Python knows where colons are required and can detect when one is missing simply by looking at the code without running it.

debug-4-2: Who or what typically finds syntax errors?

Top of Form

A. The programmer.  
B. The compiler / interpreter.  
C. The computer.  
D. The teacher / instructor.  
Check MeCompare me

Bottom of Form

✔️ The compiler and / or interpreter is a computer program that determines if your program is written in a way that can be translated into machine language for execution.

# 3.5. Runtime Errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

**Check your understanding**

debug-51: Which of the following is a run-time error?

Top of Form

A. Attempting to divide by 0.  
B. Forgetting a colon at the end of a statement where one is required.  
C. Forgetting to divide by 100 when printing a percentage amount.  
Check MeCompare me

Bottom of Form

✔️ Python cannot reliably tell if you are trying to divide by 0 until it is executing your program (e.g., you might be asking the user for a value and then dividing by that value—you cannot know what value the user will enter before you run the program).

# 3.6. Semantic Errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

**Check your understanding**

debug-6-1: Which of the following is a semantic error?

Top of Form

A. Attempting to divide by 0.  
B. Forgetting a semi-colon at the end of a statement where one is required.  
C. Forgetting to divide by 100 when printing a percentage amount.  
Check MeCompare me

Bottom of Form

✔️ This will produce the wrong answer because the programmer implemented the solution incorrectly. This is a semantic error.

# 3.7. 👩‍💻 Know Your Error Messages[¶](https://fopp.umsi.education/runestone/static/fopp/Debugging/KnowyourerrorMessages.html#know-your-error-messages)

current\_time\_str = input("What is the current time (in hours 0-23)?")

wait\_time\_str = input("How many hours do you want to wait")

current\_time\_int = int(current\_time\_str)

wait\_time\_int = int(wait\_time\_int)

final\_time\_int = current\_time\_int + wait\_time\_int

print(final\_time\_int)

### **Error**

NameError: name 'wait\_time\_int' is not defined on line 5

### **Description**

A name error almost always means that you have used a variable before it has a value. Often this may be a simple typo, so check the spelling carefully.

### **To Fix**

Check the right hand side of assignment statements and your function calls, this is the most likely place for a NameError to be found.

current\_time\_str = input("What is the current time (in hours 0-23)?")

wait\_time\_str = input("How many hours do you want to wait"

current\_time\_int = int(current\_time\_str)

wait\_time\_int = int(wait\_time\_str)

final\_time\_int = current\_time\_int + wait\_time\_int

print(final\_time\_int)

### **Error**

SyntaxError: bad input on line 4

### **Description**

This message indicates that Python can't figure out the syntax of a particular statement. Some examples are assigning to a literal, or a function call

### **To Fix**

Check your assignment statments and make sure that the left hand side of the assignment is a variable, not a literal or a function.

**Finding Clues** How can you help yourself find these problems? One trick that can be very valuable in this situation is to simply start by commenting out the line number that is flagged as having the error. If you comment out line four, the error message now changes to point to line 5. Now you ask yourself, am I really that bad that I have two lines in a row that have errors on them? Maybe, so taken to the extreme, you could comment out all of the remaining lines in the program. Now the error message changes to TokenError: EOF in multi-line statement This is a very technical way of saying that Python got to the end of file (EOF) while it was still looking for something. In this case a right parenthesis.

current\_time\_str = input("What is the "current time" (in hours 0-23)?")

wait\_time\_str = input("How many hours do you want to wait")

current\_time\_int = int(current\_time\_str)

wait\_time\_int = int(wait\_time\_str)

final\_time\_int = current\_time\_int + wait\_time\_int

print(final\_time\_int)

### **Error**

SyntaxError: bad input on line 1

### **Description**

This message indicates that Python can't figure out the syntax of a particular statement. Some examples are assigning to a literal, or a function call

### **To Fix**

Check your assignment statments and make sure that the left hand side of the assignment is a variable, not a literal or a function.

**Finding Clues** If you follow the same advice as for the last problem, comment out line one, you will immediately get a different error message. Here’s where you need to be very careful and not panic. The error message you get now is: NameError: name 'current\_time\_str' is not defined on line 4. You might be very tempted to think that this is somehow related to the earlier problem and immediately conclude that there is something wrong with the variable name current\_time\_str but if you reflect for a minute you will see that by commenting out line one you have caused a new and unrelated error. That is you have commented out the creation of the name current\_time\_str. So of course when you want to convert it to an int you will get the NameError. Yes, this can be confusing, but it will become much easier with experience. It’s also important to keep calm, and evaluate each new clue carefully so you don’t waste time chasing problems that are not really there.

Uncomment line 1 and you are back to the ParseError. Another track is to eliminate a possible source of error. Rather than commenting out the entire line you might just try to assign current\_time\_str to a constant value. For example you might make line one look like this:current\_time\_str = "10"  #input("What is the "current time" (in hours 0-23)?"). Now you have assigned current\_time\_str to the string 10, and commented out the input statement. And now the program works! So you conclude that the problem must have something to do with the input function.

## TypeError[¶](https://fopp.umsi.education/runestone/static/fopp/Debugging/KnowyourerrorMessages.html#typeerror)

a = input('wpisz godzine')

x = input('wpisz liczbe godzin')

int(x)

int(a)

h = x // 24

s = x % 24

print (h, s)

a = a + s

print ('godzina teraz', a)

### **Error**

TypeError: unsupported operand type(s) for FloorDiv: 'str' and 'int' on line 5

### **Description**

Type errors most often occur when an expression tries to combine two objects with types that should not be combined. Like raising a string to a power

### **To Fix**

To fix a type error you will most likely need to trace through your code and make sure the variables have the types you expect them to have. It may be helpful to print out each variable along the way to be sure its value is what you think it should be.

**Finding Clues** One thing that can help you in this situation is to print out the values and the types of the variables involved in the statement that is causing the error. You might try adding a print statement after line 4 print(x, type(x)) You will see that at least we have confirmed that x is of type string. Now you need to start to work backward through the program. You need to ask yourself, where is x used in the program? x is used on lines 2, 3, and of course 5 and 6 (where we are getting an error). So maybe you move the print statement to be after line 2 and again after 3. Line 3 is where you expect the value of x to be changed to an integer. Could line 4 be mysteriously changing x back to a string? Not very likely. So the value and type of x is just what you would expect it to be after line 2, but not after line 3. This helps you isolate the problem to line 3. In fact if you employ one of our earlier techniques of commenting out line 3 you will see that this has no impact on the error, and is a big clue that line 3 as it is currently written is useless.

## NameError

Name errors almost always mean that you have used a variable before it has a value. Often NameErrors are simply caused by typos in your code. They can be hard to spot if you don’t have a good eye for catching spelling mistakes. Other times you may simply mis-remember the name of a variable or even a function you want to call. You have seen one example of a NameError at the beginning of this section. Here is another one. See if you can get this program to run successfully:

Save & RunDownloadLoad HistoryShow CodeLens

str\_time = input("What time is it now?")

str\_wait\_time = input("What is the number of nours to wait?")

time = int(str\_time)

wai\_time = int(str\_wait\_time)

time\_when\_alarm\_go\_off = time + wait\_time

print(time\_when\_alarm\_go\_off)

### **Error**

NameError: name 'wait\_time' is not defined on line 6

### **Description**

A name error almost always means that you have used a variable before it has a value. Often this may be a simple typo, so check the spelling carefully.

### **To Fix**

Check the right hand side of assignment statements and your function calls, this is the most likely place for a NameError to be found.

**Finding Clues** With name errors one of the best things you can do is use the editor, or browser search function. Quite often if you search for the exact word in the error message one of two things will happen:

1. The word you are searching for will appear only once in your code, it’s also likely that it will be on the right hand side of an assignment statement, or as a parameter to a function. That should confirm for you that you have a typo somewhere. If the name in question **is** what you thought it should be then you probably have a typo on the left hand side of an assignment statement on a line before your error message occurs. Start looking backward at your assignment statements. In some cases it’s really nice to leave all the highlighted strings from the search function visible as they will help you very quickly find a line where you might have expected your variable to be highlighted.

2. The second thing that may happen is that you will be looking directly at a line where you expected the search to find the string in question, but it will not be highlighted. Most often that will be the typo right there.

Here is another one for you to try:

n = input("What time is it now (in hours)?")

n = imt(n)

m = input("How many hours do you want to wait?")

m = int(m)

q = m % 12

print("The time is now", q)

### **Error**

NameError: name 'imt' is not defined on line 2

### **Description**

A name error almost always means that you have used a variable before it has a value. Often this may be a simple typo, so check the spelling carefully.

### **To Fix**

Check the right hand side of assignment statements and your function calls, this is the most likely place for a NameError to be found.

## ValueError[¶](https://fopp.umsi.education/runestone/static/fopp/Debugging/KnowyourerrorMessages.html#valueerror)

Value errors occur when you pass a parameter to a function and the function is expecting a certain limitations on the values, and the value passed is not compatible. We can illustrate that with this particular program in two different ways.

current\_time\_str = input("What is the current time (in hours 0-23)?")

current\_time\_int = int(current\_time\_str)

wait\_time\_str = input("How many hours do you want to wait")

wait\_time\_int = int(wait\_time\_int)

final\_time\_int = current\_time\_int + wait\_time\_int

print(final\_time\_int)

### **Error**

ValueError: invalid literal for int() with base 10: '' on line 2

### **Description**

A ValueError most often occurs when you pass a parameter to a function and the function is expecting one type and you pass another.

### **To Fix**

The error message gives you a pretty good hint about the name of the function as well as the value that is incorrect. Look at the error message closely and then trace back to the variable containing the problematic value.

(ac4\_7\_8)

Run the program but instead of typing in anything to the dialog box just click OK. You should see the following error message: ValueError: invalid literal for int() with base 10: '' on line: 4 This error is not because you have made a mistake in your program. Although sometimes we do want to check the user input to make sure its valid, but we don’t have all the tools we need for that yet. The error happens because the user did not give us something we can convert to an integer, instead we gave it an empty string. Try running the program again. Now this time enter “ten” instead of the number 10. You will get a similar error message.

ValueErrors are not always caused by user input error, but in this program that is the case. We’ll look again at ValueErrors again when we get to more complicated programs. For now it is worth repeating that you need to keep track of the restrictions needed for your variables, and understand what your function is expecting. You can do this by writing comments in your code, or by naming your variables in a way that reminds you of their proper form.

# Modules

A **module** is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the **standard library**. Providing additional functionality through modules allows you to only use the functionality you need when you need it, and it keeps your code cleaner.

Functions imported as part of a module live in their own **namespace**. A namespace is simply a space within which all names are distinct from each other. The same name can be reused in different namespaces but two objects can’t have the same name within a single namespace. One example of a namespace is the set of street names within a single city. Many cities have a street called “Main Street”, but it’s very confusing if two streets in the same city have that name! Another example example is the folder organization of file systems. You can have a file called todo in your work folder as well as your personal folder, but you know which is which because of the folder it’s in; each folder has its own namespace for files. Note that human names are not part of a namespace that enforces uniqueness; that’s why governments have invented unique identifiers to assign to people, like passport numbers.

The [Python Documentation](https://docs.python.org/3.6/) site for Python version 3.6 is an extremely useful reference for all aspects of Python. The site contains a listing of all the standard modules that are available with Python (see [Global Module Index](https://docs.python.org/3.6/py-modindex.html)). You will also see that there is a [Standard Library Reference](https://docs.python.org/3.6/library/index.html) and a [Tutorial](https://docs.python.org/3.6/tutorial/index.html) as well as installation instructions, how-tos, and frequently asked questions. We encourage you to become familiar with this site and to use it often.

If you have not done so already, take a look at the Global Module Index. Here you will see an alphabetical listing of all the modules that are available as part of the standard library. Find the turtle module.

## 4.2.1. Importing Modules

In order to use Python modules, you have to **import** them into a Python program. That happens with an import statement: the word import, and then the name of the module. The name is case-sensitive. Roughly translated to English, an import statement says “there’s some code in another file; please make its functions and variables available in this file.” More technically, an import statement causes all the code in another file to be executed. Any variables that are bound during that execution (including functions that are defined) may then be referred in some way (to be discussed) in the current file.

By convention, all import commands are put at the very top of your file. They can be put elsewhere, but that can lead to some confusions, so it’s best to follow the convention.

Where do these other files that you can import come from? It could be a code file that you wrote yourself, or it could be code that someone else wrote and you copied on to your computer.

For example, if you have a file myprog.py in directory ~/Desktop/mycode/, and myprog.py contains a line of code import morecode, then the python interpreter will look for a file called morecode.py, excecute its code, and make its object bindings available for reference in the rest of the code in myprog.py.

Note that it is import morecode, not import morecode.py, but the other file has to be called morecode.py.

The tests you see in your problem sets are also using a Python module that’s in the standard library, calledunittest. Right now, you can’t see the code that causes those tests to run, because we have hidden it from you, but later in the course, you will learn how to write your own Unit Tests for code, and to do so, you will need to write an import statement at the beginning of your programs. Even before you learn how to write your own tests, you will see code for Unit Tests in your problem set files.

**Don’t overwrite standard library modules!**

Given the order of search for external Python modules that is described in the list above, it is possible to overwrite a standard library. For example, if you create a file random.py in the same directory where myprog.py lives, and then myprog.py invokes import random, it will import yourfile rather than the standard library module. That’s not usually what you want, so be careful about how you name your python files!

## 4.2.2. Syntax for Importing Modules and Functionality

When you see imported modules in a Python program, there are a few variations that have slightly different consequences.

1. The most common is import morecode. That imports everything in morecode.py. To invoke a function f1 that is defined in morecode.py, you would write morecode.f1(). Note that you have to explicitly mention morecode again, to specify that you want the f1 function from the morecode namespace. If you just write f1(), python will look for an f1 that was defined in the current file, rather than in morecode.py.
2. You can also give the imported module an alias (a different name, just for when you use it in your program). For example, after executing import morecode as mc, you would invoke f1 as mc.f1(). You have now given the morecode module the alias mc. Programmers often do this to make code easier to type.
3. A third possibility for importing occurs when you only want to import SOME of the functionality from a module, and you want to make those objects be part of the current module’s namespace. For example, you could write from morecode import f1. Then you could invoke f1 without referencing morecode again: f1().

**Note: Python modules and limitations with activecode**

Throughout the chapters of this book, activecode windows allow you to practice the Python that you are learning. We mentioned in the first chapter that programming is normally done using some type of development environment and that the activecode used here was strictly to help us learn. It is not the way we write production programs.

To that end, it is necessary to mention that many of the modules available in standard Python will **not** work in the activecode environment. In fact, only turtle, math, random, and a couple others have been ported at this point. If you wish to explore any additional modules, you will need to run from the native python interpreter on your computer.

**Check your understanding**

modules-1-1: In Python a module is:

Top of Form

A. A file containing Python definitions and statements intended for use in other Python programs.  
B. A separate block of code within a program.  
C. One line of code in a program.  
D. A file that contains documentation about functions in Python.  
Check MeCompare me

Bottom of Form

✔️ A module can be reused in different programs.

modules-1-2: To find out information on the standard modules available with Python you should:

Top of Form

A. Go to the Python Documentation site.  
B. Look at the import statements of the program you are working with or writing.  
C. Ask the professor.  
D. Look in this textbook.  
Check MeCompare me

Bottom of Form

✔️ The site contains a listing of all the standard modules that are available with Python.

We often want to use **random numbers** in programs. Here are a few typical uses:

* To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,
* To shuffle a deck of playing cards randomly,
* To randomly allow a new enemy spaceship to appear and shoot at you,
* To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
* For encrypting your banking session on the Internet.

Python provides a module random that helps with tasks like this. You can take a look at it in the documentation. Here are the key things we can do with it.

import random

prob = random.random()

print(prob)

diceThrow = random.randrange(1,7) # return an int, one of 1,2,3,4,5,6

print(diceThrow)

def reverse(s):

    if len(s) == 0:

        return s

    else:

        return reverse(s[1:]) + s[0]

s = "Geeksforgeeks"

print ("The original string  is : ",end="")

print (s)

print ("The reversed string(using recursion) is : ",end="")

print (reverse(s))