Find and read 20 interview questions for Data Types, Operators, Conditional Statements, Looping Statements, Functions.

# Python Data Types

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

1. a = 5

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer 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.

1. a=10
2. b="Hi Python"
3. c = 10.5
4. **print**(type(a))
5. **print**(type(b))
6. **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. [Sequence Type](https://www.javatpoint.com/python-data-types#SequenceType)
3. [Boolean](https://www.javatpoint.com/python-data-types#Boolean)
4. [Set](https://www.javatpoint.com/python-data-types#Set)
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. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

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

1. a = 5
2. **print**("The type of a", type(a))
4. b = 40.5
5. **print**("The type of b", type(b))
7. c = 1+3j
8. **print**("The type of c", type(c))
9. **print**(" c is a complex number", isinstance(1+3j,complex))

**Output:**

The type of a <class 'int'>

The type of b <class 'float'>

The type of c <class 'complex'>

c is complex number: True

Python supports three types of numeric data.

1. **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
3. **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

### **Sequence Type**

### **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 Python provides built-in functions and operators to perform operations in the string.

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 a repetition operator as the operation "Python" \*2 returns 'Python Python'.

The following example illustrates the string in Python.

AD

**Example - 1**

1. str = "string using double quotes"
2. **print**(str)
3. s = '''''A multiline
4. string'''
5. **print**(s)

**Output:**

string using double quotes

A multiline

string

Consider the following example of string handling.

**Example - 2**

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:**

AD

he

o

hello javatpointhello javatpoint

hello javatpoint how are you

### **List**

Python 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. list1  = [1, "hi", "Python", 2]
2. #Checking type of given list
3. **print**(type(list1))
5. #Printing the list1
6. **print** (list1)
8. # List slicing
9. **print** (list1[3:])
11. # List slicing
12. **print** (list1[0:2])
14. # List Concatenation using + operator
15. **print** (list1 + list1)
17. # List repetation using \* operator
18. **print** (list1 \* 3)

**Output:**

[1, 'hi', 'Python', 2]

[2]

[1, 'hi']

[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. tup  = ("hi", "Python", 2)
2. # Checking type of tup
3. **print** (type(tup))
5. #Printing the tuple
6. **print** (tup)
8. # Tuple slicing
9. **print** (tup[1:])
10. **print** (tup[0:1])
12. # Tuple concatenation using + operator
13. **print** (tup + tup)
15. # Tuple repatation using \* operator
16. **print** (tup \* 3)
18. # Adding value to tup. It will throw an error.
19. t[2] = "hi"

**Output:**

<class 'tuple'>

('hi', 'Python', 2)

('Python', 2)

('hi',)

('hi', 'Python', 2, 'hi', 'Python', 2)

('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):

File "main.py", line 14, in <module>

t[2] = "hi";

TypeError: 'tuple' object does not support item assignment

### **Dictionary**

Dictionary is an unordered 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'}
3. # Printing dictionary
4. **print** (d)
6. # Accesing value using keys
7. **print**("1st name is "+d[1])
8. **print**("2nd name is "+ d[4])
10. **print** (d.keys())
11. **print** (d.values())

**Output:**

1st name is Jimmy

2nd name is mike

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

dict\_keys([1, 2, 3, 4])

dict\_values(['Jimmy', 'Alex', 'john', 'mike'])

### **Boolean**

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'. Consider the following example.

1. # Python program to check the boolean type
2. **print**(type(True))
3. **print**(type(False))
4. **print**(false)

**Output:**

<class 'bool'>

<class 'bool'>

NameError: name 'false' is not defined

### **Set**

Python Set is the unordered collection of the data type. It is iterable, mutable(can modify after creation), and has unique elements. In set, the order of the elements is undefined; it may return the changed sequence of the element. The set is created by using a built-in function **set(),** or a sequence of elements is passed in the curly braces and separated by the comma. It can contain various types of values. Consider the following example.

1. # Creating Empty set
2. set1 = set()
4. set2 = {'James', 2, 3,'Python'}
6. #Printing Set value
7. **print**(set2)
9. # Adding element to the set
11. set2.add(10)
12. **print**(set2)
14. #Removing element from the set
15. set2.remove(2)
16. **print**(set2)

**Output:**

{3, 'Python', 'James', 2}

{'Python', 'James', 3, 2, 10}

{'Python', 'James'

Questions for data types:

|  |
| --- |
|  1. What is the type of the following: 1 A) float B) int C) str  Answer: int  Explanation: As there is no decimal, the number is of type int |
| 2. What is the type of the following "7.1" A) float B) int C) str  Answer: str  Explanation: The type is string |
| 3. What is the result of the following code segment: int(12.3) A) 12.3 B) 12 C) 13  Answer: 12  Explanation: In Python, if you cast a float to an integer, the conversion truncates towards zero. |
| 4. What is the result of the following code segment: int(True) A) 1 B) 0 C) error  Answer: 1  Explanation: When you cast a boolean True to an integer you get a 1 |
| 5. What do you call a value that doesn’t have decimal values? A) A number B) An integer C) A string  Answer: An integer |
| 6. What do you call a value that does have decimal values? A) A float B) A number C) An integer  Answer: A float |
| 7. What data type can only have either a value of True or False? A) A string B) A boolean C) An integer  Answer: A boolean |
| 8. What code would turn the string “1” into an integer? A)str(1) B) int("1") C) float("1")  Answer: int("1") |
| 9. What character begins a single line comment? A)''' B) // C) #  Answer:# |
| 10. What do we call it when we convert from one data type to another? A)casting B) converting C) changing  Answer:casting |
| 11. What is the datatype of **np.nan**? A) int B) float C) str D) None  Answer: float |
| 12. Which of the following numbers is NOT a float? A) 1.5 B) 2.333333 C) 0.0 D) 0  Answer: 0  Explanation:0 on its own is an int. 0.0 however, is a float. |
| 13. What values can the Boolean data type hold? A) Integers, fractions, complex numbers B) Unicode characters C) True or False values D) Any other data type  Answer: True or False values |
| 14. What does it mean that Python is a dynamically-typed language? A) Variables in python can implicitly change to other types when comparing. For examples you can compare a string "2" and the number 2 using ==. B) Python variables can be assigned to different types and changes types at will. C) Python is a more efficient language than C++ D) All of the above  Answer: Python variables can be assigned to different types and changes types at will.  Explanation: Dynamic-typing just refers to the ability for variables to flexibly learn their types during assignment. |

**Python Interview Questions on Data Types**

## What are Data Types?

A data type is a set of values and a set of operations defined on data. An implementation of a data type is an expression of data and operations in terms of a specific programming language such as Java, C ++, or Python.

Now before moving to the Python interview questions based on data types let’s have a quick look at the inbuilt data types that we get in the Python programming language:

| **Data Types** | **Examples** |
| --- | --- |
| Numbers | 1234, 3.1415, 3+4j |
| Strings | ‘spam’, “Bob’s”, b’a\x01c’, |
| Lists | [1, [2, ‘three’], 4.5] |
| Dictionaries | {‘food’: ‘spam’, ‘taste’: ‘yum’} |
| Tuples | (1, ‘spam’, 4, ‘U’) |
| Sets | set(‘abc’), {‘a’, ‘b’, ‘c’} |

## Python Interview Questions on Data Types

#### Name four of the main data types in Python

Numbers, strings, lists, dictionaries, tuples, files, and sets are generally considered the main types of data. Types, None, and Booleans are sometimes also classified this way. The integer, floating-point, complex, fraction and decimal are numerical data types and simple strings and Unicode strings in Python 2 and text strings and byte strings in Python 3 are the types of string data types.

#### Why are these data types known as Python’s core data types?

They are known as the core data types because they are part of the Python language itself and are always available to create other objects, you usually need to call functions in imported modules.

Most of the data types have a specific syntax for generating objects: “spam”, for example, is an expression that creates a string and determines the set of operations that can be applied to it. For this reason, main types are built into Python syntax. Instead, you must call the built-in open function to create a file object.

#### What does immutable mean and what three types of Python core data types are considered immutable?

An immutable data type is a type of object which cannot be modified after its creation. Numbers, strings, and tuples in Python fall into this category. Although you cannot modify an immutable object in place, you can always create a new one by running an expression.

#### What does sequence mean and which three types of data fall into this category?

A sequence data type is a collection of objects ordered by a specific position. In Python, Strings, lists, and tuples are the data types based on sequences. The Sequences share common sequence operations, such as indexing, concatenation, and slicing, but also have type-specific method calls.

#### What does mapping mean and what kind of data type is based on mapping?

The term mapping refers to an object that maps keys to associated values. The Python dictionary is the only type of mapping in the base typeset. Mappings do not maintain any left-to-right position order; they support access to stored data by key, as well as type-specific method calls.

#### What is polymorphism and why should you care?

Polymorphism means that the meaning of an operation (like a+) depends on the objects being operated. This turns out to be a key idea behind good use of Python, not coercing code to specific types makes that code automatically applied to many types.

2. Operators question In python

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Python Operators The operator is a symbol that performs a certain operation between two operands, according to one definition. In a particular programming language, operators serve as the foundation upon which logic is constructed in a programme. The different operators that Python offers are listed here.   * Arithmetic operators * Comparison operators * Assignment Operators * Logical Operators * Bitwise Operators * Membership Operators * Identity Operators * Arithmetic Operators  **Arithmetic Operators** Arithmetic operations between two operands are carried out using arithmetic operators. It includes the exponent (\*\*) operator as well as the + (addition), - (subtraction), \* (multiplication), / (divide), % (reminder), and // (floor division) operators.  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 = 10, b = 10 => a+b = 20 | | **- (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 results negative. For example, if a = 20, b = 5 => a - b = 15 | | **/ (divide)** | It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0 | | **\* (Multiplication)** | It is used to multiply one operand with the other. For example, if a = 20, b = 4 => a \* b = 80 | | **% (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)** | As it calculates the first operand's power to the second operand, it is an exponent operator. | | **// (Floor division)** | It provides the quotient's floor value, which is obtained by dividing the two operands. |  **Comparison operator** Comparison operators compare the values of the two operands and return a true or false Boolean value in accordance. The following table lists the comparison operators.   |  |  | | --- | --- | | **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. | | <= | The condition is met if the first operand is smaller than or equal to the second operand. | | >= | The condition is met if the first operand is greater than or equal to the second operand. | | > | 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. |  **Assignment Operators** The right expression's value is assigned to the left operand using the assignment operators. The following table provides a description of the assignment operators.   |  |  | | --- | --- | | **Operator** | **Description** | | = | It assigns the value of the right expression to the left operand. | | += | By multiplying the value of the right operand by the value of the left operand, the left operand receives a changed value. 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 assigns 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 assigns the modified value back to then the 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 assigns the reminder back to the 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. |   AD **Bitwise Operators** The two operands' values are processed bit by bit by the bitwise operators. Consider the case below.  **For example,**   1. **if** a = 7 2. b = 6 3. then, binary (a) = 0111 4. binary (b) = 0110 6. hence, a & b = 0011 7. a | b = 0111 8. a ^ b = 0100 9. ~ a = 1000  |  |  | | --- | --- | | **Operator** | **Description** | | & (binary and) | A 1 is copied to the result if both bits in two operands at the same location are 1. If not, 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) | If the two bits are different, the outcome bit will be 1, else it will be 0. | | ~ (negation) | The operand's bits are calculated as their negations, so if one bit is 0, the next bit will be 1, and vice versa. | | << (left shift) | The number of bits in the right operand is multiplied by the leftward shift of the value of the left operand. | | >> (right shift) | The left operand is moved right by the number of bits present in the right operand. |  **Logical Operators** The assessment of expressions to make decisions typically makes use of the logical operators. The following logical operators are supported by Python.   |  |  | | --- | --- | | **Operator** | **Description** | | and | The condition will also be true if the expression is true. If the two expressions a and b are the same, then a and b must both be true. | | or | The condition will be true if one of the phrases is true. If a and b are the two expressions, then an or b must be true if and is true and b is false. | | not | If an expression **a** is true, then not (a) will be false and vice versa. |  **Membership Operators** The membership of a value inside a Python data structure can be verified using Python membership operators. The result is true if the value is in the data structure; otherwise, it returns false.   |  |  | | --- | --- | | **Operator** | **Description** | | in | If the first operand cannot be found in the second operand, it is evaluated to be true (list, tuple, or dictionary). | | not in | If the first operand is not present in the second operand, the evaluation is true (list, tuple, or dictionary). |  **Identity Operators**  |  |  | | --- | --- | | **Operator** | **Description** | | is | If the references on both sides point to the same object, it is determined to be true. | | is not | If the references on both sides do not point at the same object, it is determined to be true. |  **Operator Precedence** The order in which the operators are examined is crucial to understand since it tells us which operator needs to be considered first. Below is a list of the Python operators' precedence tables.   |  |  | | --- | --- | | **Operator** | **Description** | | \*\* | Overall other operators employed in the expression, the exponent operator is given precedence. | | ~ + - | the minus, unary plus, and negation. | | \* / % // | the division of the floor, the modules, the division, and the multiplication. | | + - | Binary plus, and minus | | >> << | Left shift. and right shift | | & | Binary and. | | ^ | | Binary xor, and or | | <= < > >= | Comparison operators (less than, less than equal to, greater than, greater then equal to). | | <> == != | Equality operators. | | = %= /= //= -= += \*= \*\*= | Assignment operators | | is is not | Identity operators | | in not in | Membership operators | | not or and | Logical operators |   AD |

3 Conditional Statements

# Python If-else statements

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

|  |  |
| --- | --- |
| **Statement** | **Description** |
|  |  |
| If Statement | The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed. |
| If - else Statement | The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed. |
| Nested if Statement | Nested if statements enable us to use if ? else statement inside an outer if statement. |

## **Indentation in Python**

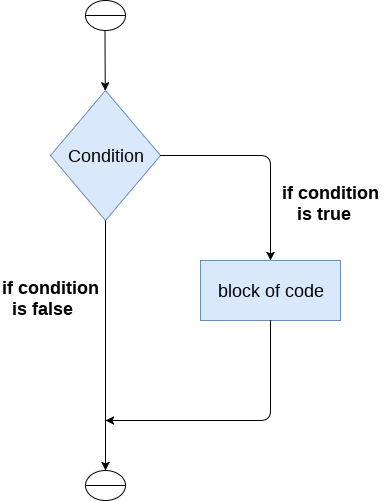
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

## **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

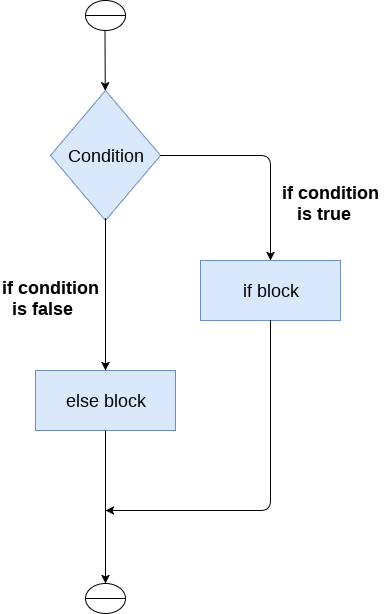
Enter c? 130

c is largest

## **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)

### **Example 1 : 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 !!

### **Example 2: 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...")

**Output:**

enter the number?10

Number is even

AD

## **The elif statement**

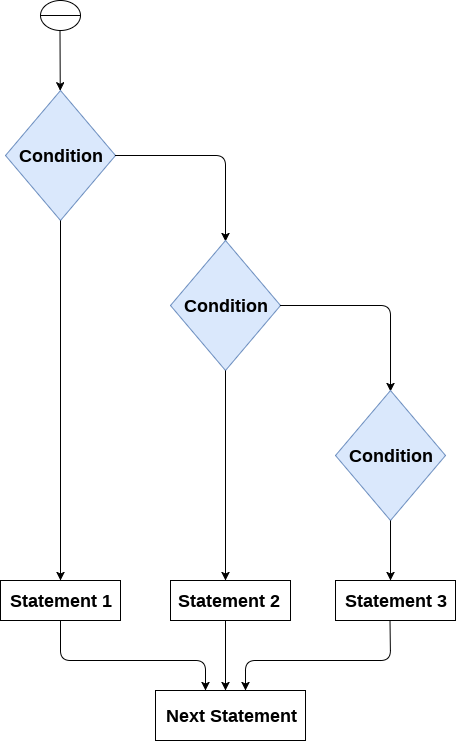
The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

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.

AD

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 ?")

# Python Loops

The following loops are available in Python to fulfil the looping needs. Python offers 3 choices for running the loops. The basic functionality of all the techniques is the same, although the syntax and the amount of time required for checking the condition differ.

We can run a single statement or set of statements repeatedly using a loop command.

The following sorts of loops are available in the Python programming language.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name of the loop** | **Loop Type & Description** |
| 1 | **While loop** | Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | **For loop** | This type of loop executes a code block multiple times and abbreviates the code that manages the loop variable. |
| 3 | **Nested loops** | We can iterate a loop inside another loop. |

## **Loop Control Statements**

Statements used to control loops and change the course of iteration are called control statements. All the objects produced within the local scope of the loop are deleted when execution is completed.

Python provides the following control statements. We will discuss them later in detail.

Let us quickly go over the definitions of these loop control statements.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name of the control statement** | **Description** |
| 1 | **Break statement** | This command terminates the loop's execution and transfers the program's control to the statement next to the loop. |
| 2 | **Continue statement** | This command skips the current iteration of the loop. The statements following the continue statement are not executed once the Python interpreter reaches the continue statement. |
| 3 | **Pass statement** | The pass statement is used when a statement is syntactically necessary, but no code is to be executed. |

## **The for Loop**

Python's for loop is designed to repeatedly execute a code block while iterating through a list, tuple, dictionary, or other iterable objects of Python. The process of traversing a sequence is known as iteration.

**Syntax of the for Loop**

1. **for** value **in** sequence:
2. { code block }

In this case, the variable value is used to hold the value of every item present in the sequence before the iteration begins until this particular iteration is completed.

Loop iterates until the final item of the sequence are reached.

**Code**

1. # Python program to show how the for loop works
3. # Creating a sequence which is a tuple of numbers
4. numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]
6. # variable to store the square of the number
7. square = 0
9. # Creating an empty list
10. squares = []
12. # Creating a for loop
13. **for** value **in** numbers:
14. square = value \*\* 2
15. squares.append(square)
16. **print**("The list of squares is", squares)

**Output:**

The list of squares is [16, 4, 36, 49, 9, 25, 64, 100, 36, 1, 81, 4]

### **Using else Statement with for Loop**

As already said, a for loop executes the code block until the sequence element is reached. The statement is written right after the for loop is executed after the execution of the for loop is complete.

Only if the execution is complete does the else statement comes into play. It won't be executed if we exit the loop or if an error is thrown.

Here is a code to better understand if-else statements.

**Code**

1. # Python program to show how if-else statements work
3. string = "Python Loop"
5. # Initiating a loop
6. **for** s **in** a string:
7. # giving a condition in if block
8. **if** s == "o":
9. **print**("If block")
10. # if condition is not satisfied then else block will be executed
11. **else**:
12. **print**(s)

**Output:**

AD

P

y

t

h

If block

n

L

If block

If block

p

Now similarly, using else with for loop.

**Syntax:**

1. **for** value **in** sequence:
2. # executes the statements until sequences are exhausted
3. **else**:
4. # executes these statements when for loop is completed

**Code**

1. # Python program to show how to use else statement with for loop
3. # Creating a sequence
4. tuple\_ = (3, 4, 6, 8, 9, 2, 3, 8, 9, 7)
6. # Initiating the loop
7. **for** value **in** tuple\_:
8. **if** value % 2 != 0:
9. **print**(value)
10. # giving an else statement
11. **else**:
12. **print**("These are the odd numbers present in the tuple")

**Output:**

3

9

3

9

7

These are the odd numbers present in the tuple

### **The range() Function**

With the help of the range() function, we may produce a series of numbers. range(10) will produce values between 0 and 9. (10 numbers).

AD

We can give specific start, stop, and step size values in the manner range(start, stop, step size). If the step size is not specified, it defaults to 1.

Since it doesn't create every value it "contains" after we construct it, the range object can be characterized as being "slow." It does provide in, len, and \_\_getitem\_\_ actions, but it is not an iterator.

The example that follows will make this clear.

**Code**

1. # Python program to show the working of range() function
3. **print**(range(15))
5. **print**(list(range(15)))
7. **print**(list(range(4, 9)))
9. **print**(list(range(5, 25, 4)))

**Output:**

range(0, 15)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

[4, 5, 6, 7, 8]

[5, 9, 13, 17, 21]

To iterate through a sequence of items, we can apply the range() method in for loops. We can use indexing to iterate through the given sequence by combining it with an iterable's len() function. Here's an illustration.

**Code**

1. # Python program to iterate over a sequence with the help of indexing
3. tuple\_ = ("Python", "Loops", "Sequence", "Condition", "Range")
5. # iterating over tuple\_ using range() function
6. **for** iterator **in** range(len(tuple\_)):
7. **print**(tuple\_[iterator].upper())

**Output:**

PYTHON

LOOPS

SEQUENCE

CONDITION

RANGE

## **While Loop**

While loops are used in Python to iterate until a specified condition is met. However, the statement in the program that follows the while loop is executed once the condition changes to false.

**Syntax of the while loop is:**

1. **while** <condition>:
2. { code block }

All the coding statements that follow a structural command define a code block. These statements are intended with the same number of spaces. Python groups statements together with indentation.  
**Code**

1. # Python program to show how to use a while loop
2. counter = 0
3. # Initiating the loop
4. **while** counter < 10: # giving the condition
5. counter = counter + 3
6. **print**("Python Loops")

**Output:**

Python Loops

Python Loops

Python Loops

Python Loops

### **Using else Statement with while Loops**

As discussed earlier in the for loop section, we can use the else statement with the while loop also. It has the same syntax.

**Code**

1. #Python program to show how to use else statement with the while loop
2. counter = 0
4. # Iterating through the while loop
5. **while** (counter < 10):
6. counter = counter + 3
7. **print**("Python Loops") # Executed untile condition is met
8. # Once the condition of while loop gives False this statement will be executed
9. **else**:
10. **print**("Code block inside the else statement")

**Output:**

Python Loops

Python Loops

Python Loops

Python Loops

Code block inside the else statement

### **Single statement while Block**

The loop can be declared in a single statement, as seen below. This is similar to the if-else block, where we can write the code block in a single line.

**Code**

1. # Python program to show how to write a single statement while loop
2. counter = 0
3. **while** (count < 3): **print**("Python Loops")

AD

## **Loop Control Statements**

Now we will discuss the loop control statements in detail. We will see an example of each control statement.

### **Continue Statement**

It returns the control to the beginning of the loop.

**Code**

1. # Python program to show how the continue statement works
3. # Initiating the loop
4. **for** string **in** "Python Loops":
5. **if** string == "o" **or** string == "p" **or** string == "t":
6. **continue**
7. **print**('Current Letter:', string)

**Output:**

Current Letter: P

Current Letter: y

Current Letter: h

Current Letter: n

Current Letter:

Current Letter: L

Current Letter: s

### **Break Statement**

It stops the execution of the loop when the break statement is reached.

**Code**

1. # Python program to show how the break statement works
3. # Initiating the loop
4. **for** string **in** "Python Loops":
5. **if** string == 'L':
6. **break**
7. **print**('Current Letter: ', string)

**Output:**

Current Letter: P

Current Letter: y

Current Letter: t

Current Letter: h

Current Letter: o

Current Letter: n

Current Letter:

### **Pass Statement**

Pass statements are used to create empty loops. Pass statement is also employed for classes, functions, and empty control statements.

**Code**

1. # Python program to show how the pass statement works
2. **for** a string **in** "Python Loops":
3. **pass**
4. **print**( 'Last Letter:', string)

**Output:**

Last Letter: s

# Python While Loops

In coding, loops are designed to execute a specified code block repeatedly. We'll learn how to construct a while loop in Python, the syntax of a while loop, loop controls like break and continue, and other exercises in this tutorial.

## **Introduction of Python While Loop**

The Python while loop iteration of a code block is executed as long as the given condition, i.e., conditional\_expression, is true.

If we don't know how many times we'll execute the iteration ahead of time, we can write an indefinite loop.

**Syntax of Python While Loop**

1. **while** conditional\_expression:
2. Code block of **while**

The given condition, i.e., conditional\_expression, is evaluated initially in the Python while loop. Then, if the conditional expression gives a boolean value True, the while loop statements are executed. The conditional expression is verified again when the complete code block is executed. This procedure repeatedly occurs until the conditional expression returns the boolean value False.

* The statements of the Python while loop are dictated by indentation.
* The code block begins when a statement is indented & ends with the very first unindented statement.
* Any non-zero number in Python is interpreted as boolean True. False is interpreted as None and 0.

## **Python While Loop Example**

Here we will sum of squares of the first 15 natural numbers using a while loop.

**Code**

1. # Python program example to show the use of **while** loop
3. num = 15
5. # initializing summation and a counter **for** iteration
6. summation = 0
7. c = 1
9. **while** c <= num: # specifying the condition of the loop
10. # begining the code block
11. summation = c\*\*2 + summation
12. c = c + 1    # incrementing the counter
14. # print the **final** sum
15. print("The sum of squares is", summation)

**Output:**

The sum of squares is 1240

Provided that our counter parameter i gives boolean true for the condition, i less than or equal to num, the loop repeatedly executes the code block i number of times.

Next is a crucial point (which is mostly forgotten). We have to increment the counter parameter's value in the loop's statements. If we don't, our while loop will execute itself indefinitely (a never-ending loop).

Finally, we print the result using the print statement.

## **Exercises of Python While Loop**

### **Prime Numbers and Python While Loop**

Using a while loop, we will construct a Python program to verify if the given integer is a prime number or not.

**Code**

1. num = [34, 12, 54, 23, 75, 34, 11]
3. def prime\_number(number):
4. condition = 0
5. iteration = 2
6. **while** iteration <= number / 2:
7. **if** number % iteration == 0:
8. condition = 1
9. **break**
10. iteration = iteration + 1
12. **if** condition == 0:
13. print(f"{number} is a PRIME number")
14. **else**:
15. print(f"{number} is not a PRIME number")
16. **for** i in num:
17. prime\_number(i)

**Output:**

34 is not a PRIME number

12 is not a PRIME number

54 is not a PRIME number

23 is a PRIME number

75 is not a PRIME number

34 is not a PRIME number

11 is a PRIME number

### **Multiplication Table using While Loop**

In this example, we will use the while loop for printing the multiplication table of a given number.

**Code**

1. num = 21
2. counter = 1
3. # we will use a **while** loop **for** iterating 10 times **for** the multiplication table
4. print("The Multiplication Table of: ", num)
5. **while** counter <= 10: # specifying the condition
6. ans = num \* counter
7. print (num, 'x', counter, '=', ans)
8. counter += 1 # expression to increment the counter

**Output:**

AD

The Multiplication Table of: 21

21 x 1 = 21

21 x 2 = 42

21 x 3 = 63

21 x 4 = 84

21 x 5 = 105

21 x 6 = 126

21 x 7 = 147

21 x 8 = 168

21 x 9 = 189

21 x 10 = 210

### **Python While Loop with List**

We will use a Python while loop to square every number of a list

**Code**

1. # Python program to square every number of a list
2. # initializing a list
3. list\_ = [3, 5, 1, 4, 6]
4. squares = []
5. # programing a **while** loop
6. **while** list\_: # until list is not empty **this** expression will give **boolean** True after that False
7. squares.append( (list\_.pop())\*\*2)
8. # print the squares
9. print( squares )

[36, 16, 1, 25, 9]

In the preceding example, we execute a while loop over a given list of integers that will repeatedly run as long as an element in the list is found.

AD

## **Python While Loop Multiple Conditions**

We'll need to recruit logical operators to combine two or more expressions specifying conditions into a single while loop. This instructs Python on collectively analyzing all of the given expressions of conditions.

We can construct a while loop with multiple conditions in this example. We have given two conditions and a and keyword, meaning until both conditions give boolean True, the loop will execute the statements.

AD

**Code**

1. num1 = 17
2. num2 = -12
4. **while** num1 > 5 and num2 < -5 : # multiple conditions in a single **while** loop
5. num1 -= 2
6. num2 += 3
7. print( (num1, num2) )

**Output:**

(15, -9)

(13, -6)

(11, -3)

Let's look at another example of multiple conditions with an OR operator.

**Code**

1. num1 = 17
2. num2 = -12
4. **while** num1 > 5 or num2 < -5 :
5. num1 -= 2
6. num2 += 3
7. print( (num1, num2) )

**Output:**

(15, -9)

(13, -6)

(11, -3)

(9, 0)

(7, 3)

(5, 6)

We can also group multiple logical expressions in the while loop, as shown in this example.

**Code**

1. num1 = 9
2. num = 14
3. maximum\_value = 4
4. counter = 0
5. **while** (counter < num1 or counter < num2) and not counter >= maximum\_value: # grouping multiple conditions
6. print(f"Number of iterations: {counter}")
7. counter += 1

**Output:**

Number of iterations: 0

Number of iterations: 1

Number of iterations: 2

Number of iterations: 3

## **Single Statement While Loop**

Similar to the if statement syntax, if our while clause consists of one statement, it may be written on the same line as the while keyword.

Here is the syntax and example of a one-line while clause -

1. # Python program to show how to create a single statement **while** loop
2. counter = 1
3. **while** counter: print('Python While Loops')

## **Loop Control Statements**

Now we will discuss the loop control statements in detail. We will see an example of each control statement.

### **Continue Statement**

It returns the control of the Python interpreter to the beginning of the loop.

**Code**

1. # Python program to show how to use **continue** loop control
3. # Initiating the loop
4. **for** string in "While Loops":
5. **if** string == "o" or string == "i" or string == "e":
6. **continue**
7. print('Current Letter:', string)

**Output:**

Current Letter: W

Current Letter: h

Current Letter: l

Current Letter:

Current Letter: L

Current Letter: p

Current Letter: s

### **Break Statement**

It stops the execution of the loop when the break statement is reached.

**Code**

1. # Python program to show how to use the **break** statement
3. # Initiating the loop
4. **for** string in "Python Loops":
5. **if** string == 'n':
6. **break**
7. print('Current Letter: ', string)

**Output:**

Current Letter: P

Current Letter: y

Current Letter: t

Current Letter: h

Current Letter: o

### **Pass Statement**

Pass statements are used to create empty loops. Pass statement is also employed for classes, functions, and empty control statements.

**Code**

1. # Python program to show how to use the pass statement
2. **for** a string in "Python Loops":
3. pass
4. print( 'Last Letter:', string)

**Output:**

Last Letter: s

# 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

AD

## **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

In this tutorial, we'll look at how to use Python continue keyword to skip the remaining statements of the current loop and go to the next iteration. Also, the difference between continue and pass keywords.

## **Application of the Continue Statement**

In Python, loops repeat processes on their own in an efficient way. However, there might be occasions when we wish to leave the current loop entirely, skip iteration, or dismiss the condition controlling the loop. We use Loop control statements in such cases. The continue keyword is a loop control statement that allows us to change the loop's control.

## **The continue Keyword**

In Python, the continue keyword return control of the iteration to the beginning of the Python for loop or Python while loop. All remaining lines in the prevailing iteration of the loop are skipped by the continue keyword, which returns execution to the beginning of the next iteration of the loop.

Both Python while and Python for loops can leverage the continue statements.

### **Example of Python Continue Statements in For Loop**

Assume the following scenario: we want to develop a program that returns numbers from 10 to 20 but not 15. It is mentioned that we must perform this with a **'for'** loop. Here's when the continue keyword comes into play. We will execute a loop from 10 to 20 and test the condition that the iterator is equal to 15. If it equals 15, we'll employ the continue statement to skip to the following iteration displaying any output; otherwise, the loop will print the result.

The following code is an example of the above scenario:

**Code**

1. # Python code to show example of continue statement
3. # looping from 10 to 20
4. **for** iterator **in** range(10, 21):
6. # If iterator is equals to 15, loop will continue to the next iteration
7. **if** iterator == 15:
8. **continue**
9. # otherwise printing the value of iterator
10. **print**( iterator )

**Output:**

10

11

12

13

14

16

17

18

19

20

Now will repeat the above code, but this time with a string. We will take a string "Javatpoint" and print each letter of the string except "a". This time we will use Python while loop to do so. Until the value of the iterator is less than the string's length, the while loop will keep executing.

**Code**

1. # Creating a string
2. string = "JavaTpoint"
3. # initializing an iterator
4. iterator = 0
6. # starting a while loop
7. **while** iterator < len(string):
8. # if loop is at letter a it will skip the remaining code and go to next iteration
9. **if** string[iterator] == 'a':
10. **continue**
11. # otherwise it will print the letter
12. **print**(string[ iterator ])
13. iterator += 1

**Output:**

J

v

T

p

o

i

n

t

## **Python Continue vs. Pass**

Usually, there is some confusion in the pass and continue keywords. So here are the differences between these two.

|  |  |  |
| --- | --- | --- |
| **Headings** | **continue** | **pass** |
| **Definition** | The continue statement is utilized to skip the current loop's remaining statements, go to the following iteration, and return control to the beginning. | The pass keyword is used when a phrase is necessary syntactically to be placed but not to be executed. |
| **Action** | It takes the control back to the start of the loop. | Nothing happens if the Python interpreter encounters the pass statement. |
| **Application** | It works with both the Python while and Python for loops. | It performs nothing; hence it is a null operation. |
| **Syntax** | It has the following syntax: -: continue | Its syntax is as follows:- pass |
| **Interpretation** | It's mostly utilized within a loop's condition. | During the byte-compile stage, the pass keyword is removed. |

# Python Pass Statement

We will discover more about pass statements in this tutorial. It is interpreted as a placeholder for the future execution of functions, classes, loops, etc.

## **What is Pass Statement in Python?**

The null statement is another name for the pass statement. A Comment is not ignored by the Python interpreter, whereas a pass statement is not. Hence, they two are different Python keywords.

We can use the pass statement as a placeholder when unsure what code to provide. So, we only have to place the pass on that line. Pass may be used when we don't wish any code to be executed. We can simply insert a pass in places where empty code is prohibited, such as loops, functions, class definitions, or if-else statements.

**Syntax of the Pass Keyword**

1. Keyword:
2. **pass**

Typically, we utilise it as a reference for the future.

Let's say we have a loop or an if-else statement that isn't to be filled now but that we wish to in the future. The pass keyword cannot have an empty body as it will be syntactically wrong. An error would be displayed by the Python interpreter suggesting to fill the space. Therefore, we create a code block that performs nothing using the pass statement.

### **Example of the Pass Statement**

**Code**

1. # Python program to show how to use a pass statement in a for loop
2. '''''pass acts as a placeholder. We can fill this place later on'''
3. sequence = {"Python", "Pass", "Statement", "Placeholder"}
4. **for** value **in** sequence:
5. **if** value == "Pass":
6. **pass** # leaving an empty if block using the pass keyword
7. **else**:
8. **print**("Not reached pass keyword: ", value)

**Output:**

Not reached pass keyword: Python

Not reached pass keyword: Placeholder

Not reached pass keyword: Statement

The same thing is also possible to create an empty function or a class.

**Code**

1. # Python program to show how to create an empty function and an empty class
3. # Empty function:
4. **def** empty():
5. **pass**
7. # Empty class
8. **class** Empty:
9. **pass**

# Python Functions

The fundamentals of Python functions, including what they are, their syntax, their primary parts, return keywords, and major types, will be covered in this tutorial. Additionally, we'll examine several instances of Python function definitions.

## **What are Python Functions?**

A function is a collection of related assertions that performs a mathematical, analytical, or evaluative operation. A collection of statements called Python Functions returns the particular task. Python functions are simple to define and essential to intermediate-level programming. The exact criteria hold to function names as they do to variable names. The goal is to group up certain often performed actions and define a function. We may call the function and reuse the code contained within it with different variables rather than repeatedly creating the same code block for different input variables.

User-defined and built-in functions are the two main categories of functions in Python. It helps maintain the programme concise, unique, and well-structured.

## **Advantages of Functions in Python**

Python functions have the following Perks.

* By including functions, we can prevent repeating the same code block repeatedly in a program.
* Python functions, once defined, can be called many times and from anywhere in a program.
* If our Python program is large, it can be separated into numerous functions which is simple to track.
* The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

However, calling functions has always been overhead in a Python program.

**Syntax of Python Function**

1. #  An example Python Function
2. **def** function\_name( parameters ):
3. # code block

The following elements make up to define a function, as seen above.

* The beginning of a function header is indicated by a keyword called def.
* function\_name is the function's name that we can use to separate it from others. We will use this name to call the function later in the program. In Python, name functions must follow the same rules as naming variables.
* We pass arguments to the defined function using parameters. However, they are optional.
* The function header is terminated by a colon (:).
* We can use a documentation string called docstring in the short form to explain the purpose of the function.
* The body of the function is made up of several valid Python statements. The indentation depth of the whole code block must be the same (usually 4 spaces).
* We can use a return expression to return a value from a defined function.

AD

### **Example of a User-Defined Function**

We will define a function that when called will return the square of the number passed to it as an argument.

**Code**

1. # Example Python Code for User-Defined function
2. **def** square( num ):
3. """
4. This function computes the square of the number.
5. """
6. **return** num\*\*2
7. object\_ = square(6)
8. **print**( "The square of the given number is: ", object\_ )

**Output:**

The square of the given number is: 36

## **Calling a Function**

A function is defined by using the def keyword and giving it a name, specifying the arguments that must be passed to the function, and structuring the code block.

After a function's fundamental framework is complete, we can call it from anywhere in the program. The following is an example of how to use the a\_function function.

**Code**

1. # Example Python Code for calling a function
2. # Defining a function
3. **def** a\_function( string ):
4. "This prints the value of length of string"
5. **return** len(string)
7. # Calling the function we defined
8. **print**( "Length of the string Functions is: ", a\_function( "Functions" ) )
9. **print**( "Length of the string Python is: ", a\_function( "Python" ) )

**Output:**

Length of the string Functions is: 9

Length of the string Python is: 6

AD

## **Pass by Reference vs. Pass by Value**

All parameters in the Python programming language are provided by reference. It indicates that if we alter the value of an argument inside of a function, the calling function will likewise reflect the change. For example,

**Code**

1. # Example Python Code for Pass by Reference vs. Value
2. # defining the function
3. **def** square( item\_list ):
4. '''''''This function will find the square of items in the list'''
5. squares = [ ]
6. **for** l **in** item\_list:
7. squares.append( l\*\*2 )
8. **return** squares
10. # calling the defined function
11. my\_list = [17, 52, 8];
12. my\_result = square( my\_list )
13. **print**( "Squares of the list are: ", my\_result )

**Output:**

AD

Squares of the list are: [289, 2704, 64]

## **Function Arguments**

The following are the types of arguments that we can use to call a function:

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

### **1) Default Arguments**

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called. Default arguments are demonstrated in the following instance.

**Code**

1. # Python code to demonstrate the use of default arguments
2. # defining a function
3. **def** function( n1, n2 = 20 ):
4. **print**("number 1 is: ", n1)
5. **print**("number 2 is: ", n2)

8. # Calling the function and passing only one argument
9. **print**( "Passing only one argument" )
10. function(30)
12. # Now giving two arguments to the function
13. **print**( "Passing two arguments" )
14. function(50,30)

**Output:**

Passing only one argument

number 1 is: 30

number 2 is: 20

Passing two arguments

number 1 is: 50

number 2 is: 30

### **2) Keyword Arguments**

A function called's arguments are linked to keyword arguments. When invoking a function with keyword arguments, the user may tell whose parameter value it is by looking at the parameter label.

AD

We can remove certain arguments or arrange them in a different order since the Python interpreter will connect the provided keywords to link the values with its parameters. Another way to use keywords to invoke the function() method is as follows:

**Code**

1. # Python code to demonstrate the use of keyword arguments
2. # Defining a function
3. **def** function( n1, n2 ):
4. **print**("number 1 is: ", n1)
5. **print**("number 2 is: ", n2)
7. # Calling function and passing arguments without using keyword
8. **print**( "Without using keyword" )
9. function( 50, 30)
11. # Calling function and passing arguments using keyword
12. **print**( "With using keyword" )
13. function( n2 = 50, n1 = 30)

**Output:**

Without using keyword

number 1 is: 50

number 2 is: 30

With using keyword

number 1 is: 30

number 2 is: 50

### **3) Required Arguments**

The arguments given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.

We must send two arguments to the function function() in the correct order, or it will return a syntax error, as seen below.

**Code**

1. # Python code to demonstrate the use of default arguments
2. # Defining a function
3. **def** function( n1, n2 ):
4. **print**("number 1 is: ", n1)
5. **print**("number 2 is: ", n2)
7. # Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30
8. **print**( "Passing out of order arguments" )
9. function( 30, 20 )
11. # Calling function and passing only one argument
12. **print**( "Passing only one argument" )
13. **try**:
14. function( 30 )
15. **except**:
16. **print**( "Function needs two positional arguments" )

**Output:**

Passing out of order arguments

number 1 is: 30

number 2 is: 20

Passing only one argument

Function needs two positional arguments

### **4) Variable-Length Arguments**

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

1. **\*args -**These are Non-Keyword Arguments
2. **\*\*kwargs -**These are Keyword Arguments.

Here is an example to clarify Variable length arguments

**Code**

1. # Python code to demonstrate the use of variable-length arguments
2. # Defining a function
3. **def** function( \*args\_list ):
4. ans = []
5. **for** l **in** args\_list:
6. ans.append( l.upper() )
7. **return** ans
8. # Passing args arguments
9. object = function('Python', 'Functions', 'tutorial')
10. **print**( object )
12. # defining a function
13. **def** function( \*\*kargs\_list ):
14. ans = []
15. **for** key, value **in** kargs\_list.items():
16. ans.append([key, value])
17. **return** ans
18. # Paasing kwargs arguments
19. object = function(First = "Python", Second = "Functions", Third = "Tutorial")
20. **print**(object)

**Output:**

['PYTHON', 'FUNCTIONS', 'TUTORIAL']

[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]

## **return Statement**

We write a return statement in a function to leave a function and give the calculated value when a defined function is called.

**Syntax:**

1. **return** < expression to be returned as output >

The return statement, which is supplied as output when a particular job or function is finished, might take the form of an argument, a statement, or a value. A declared function will return a None object if no return statement is written.

Here is an example of a return statement in Python functions.

**Code**

1. # Python code to demonstrate the use of return statements
2. # Defining a function with return statement
3. **def** square( num ):
4. **return** num\*\*2
6. # Calling function and passing arguments.
7. **print**( "With return statement" )
8. **print**( square( 52 ) )
10. # Defining a function without return statement
11. **def** square( num ):
12. num\*\*2
14. # Calling function and passing arguments.
15. **print**( "Without return statement" )
16. **print**( square( 52 ) )

**Output:**

With return statement

2704

Without return statement

None

## **The Anonymous Functions**

These types of Python functions are anonymous since we do not declare them, as we declare usual functions, using the def keyword. We can use the lambda keyword to define the short, single output, anonymous functions.

Lambda expressions can accept an unlimited number of arguments; however, they only return one value as the result of the function. They can't have numerous expressions or instructions in them. Since lambda needs an expression, an anonymous function cannot be directly called to print.

Lambda functions contain their unique local domain, meaning they can only reference variables in their argument list and the global domain name.

Although lambda expressions seem to be a one-line representation of a function, they are not like inline expressions in C and C++, which pass function stack allocations at execution for efficiency concerns.

**Syntax**

Lambda functions have exactly one line in their syntax:

1. **lambda** [argument1 [,argument2... .argumentn]] : expression

Below is an illustration of how to use the lambda function:

**Code**

1. # Python code to demonstrate ananymous functions
2. # Defining a function
3. lambda\_ = **lambda** argument1, argument2: argument1 + argument2;
5. # Calling the function and passing values
6. **print**( "Value of the function is : ", lambda\_( 20, 30 ) )
7. **print**( "Value of the function is : ", lambda\_( 40, 50 ) )

**Output:**

Value of the function is : 50

Value of the function is : 90

## **Scope and Lifetime of Variables**

The scope of a variable refers to the domain of a program wherever it is declared. A function's arguments and variables are not accessible outside the defined function. As a result, they only have a local domain.

The lifespan of a variable in RAM is how long it stays there. A function's lifespan is the same as that of its internal variables. They are taken away after we exit the function. Consequently, a function does not keep the value of a variable from previous executions.

Here's a simple example of a variable's scope within a function.

**Code**

1. # Python code to demonstrate scope and lifetime of variables
2. #defining a function to print a number.
3. **def** number( ):
4. num = 50
5. **print**( "Value of num inside the function: ", num)
7. num = 10
8. number()
9. **print**( "Value of num outside the function:", num)

**Output:**

Value of num inside the function: 50

Value of num outside the function: 10

Here, we can see that num starts out with a value of 10. The value of num outside of the function remained intact, even though the function number() changed the value of num to 50.

This is due to the fact that the function's internal variable num is different from the external variable (local to the function). Despite having the same variable name, they are two separate variables with separate scopes.

Variables beyond the function, on the contrary, are accessible within the function. These variables have a global reach. We can retrieve their values inside the function but cannot alter or change them. If we declare a variable global using the keyword global, we can also change the variable's value outside the function.

## **Python Function within Another Function**

Functions are considered first-class objects in Python. In a programming language, first-class objects are treated the same wherever they are used. They can be used in conditional expressions, as arguments, and saved in built-in data structures. A programming language is considered to implement first-class functions if it treats functions as first-class objects. The concept of First Class functions is supported by Python.

Inner or nested function refers to a function defined within another defined function. Inner functions can access the parameters of the outer scope. Inner functions are constructed to cover them from the changes that happen outside the function. Many developers regard this process as encapsulation.

**Code**

1. # Python code to show how to access variables of a nested functions
2. # defining a nested function
3. **def** word():
4. string = 'Python functions tutorial'
5. x = 5
6. **def** number():
7. **print**( string )
8. **print**( x )
10. number()
11. word()

**Output:**

Python functions tutorial

5

# Python Interview Questions on Python Conditionals

Question 1. What constitutes “True” in Python?  
**Answer:**  
A true expression is any expression that does not evaluate to 0, the empty list [ ], tuple ( ), dictionary { } or the objects None or False.

Question 2: What are the three main conditional statements in Python?  
**Answer:**  
if, elif, and else

Question 3: What are the comparison operators in Python?  
**Answer:**  
< Less than, > Greater than, <= Less than or equal to, >= Greater than or equal to, = Equal to, != not equal, o alternative not equal. Note a single = is NOT a Python comparison operator, it is an assignment operator only.

Question 4: Illustrate a basic if, elif, else structure.  
**Answer:**  
if <condition>:  
. . .  
elif<another condition>:  
. . .  
else:  
. . .

* [Python Program to Find nth Prime Number](https://btechgeeks.com/python-program-to-find-nth-prime-number/)
* [Python Data Persistence – Decision Control](https://btechgeeks.com/python-data-persistence-decision-control/)
* [Python Program to Calculate BMI](https://btechgeeks.com/python-program-to-calculate-bmi/)

### **Python Viva Questions or Python Viva Questions And Answers**

Question 5: In Python 2.5+, the equivalent of a tertiary operator has been added to the language. Provide an example of its use.  
**Answer:**  
myValue = ‘Positive’ if myNumber > 0 else ‘Negative or Zero’

Question 6: What does elif mean?  
**Answer:**  
It means else if. It is used after an if statement, to do another comparison.

Question 7. What would the output be from the following code? a =4 If a = 5:  
Print “True”  
Else:  
Print “False”  
**Answer:**  
This is a trick question. The a = 5 is not a comparison operator, but an assignment. It will yield “True”. The correct coding would be a == 5.

Question 8: How are if, elif, and else blocks defined?  
**Answer:**  
All blocks in Python are defined by indenting. All lines of a particular code block must have the same level of indenting.

Question 9: Illustrate a switch-case equivalent using if-elif-else.  
**Answer:**  
if item=valueA:  
. . .  
elif item == valueB:  
. . .  
elif item = =  valueC:  
. . .  
elifitem = valueN:  
. . .  
else:  
… #default code

Question 10: How is the Python switch statement used?  
**Answer:**  
This is a trick question, there is no built-in switch statement in Python, which is unusual. A switch statement can be easily created using if-elif using lambda or with Python dictionaries.

Question 11: Using a dictionary, create an equivalent to a switch case statement.  
**Answer:**  
deffunc1( ):  
. . .  
deffunc2( ):  
. . .  
switch = {  
‘Aardvark’: fund1,  
‘Armadillo’: fund2,  
}  
mySwi tchKey= “Armadillo ”

switch[mySwitchKey]( ) #callsJunc2( )  
switch[‘Aardvark’]( ) #calls func1( )

Question 12: Illustrate comparing two strings for equality in a case insensitive manner.  
**Answer:**  
if stringl. lower ( ) = string2.lower ( ):  
#Note: .upper( ) is equally valid.

Question 13: Illustrate comparing two strings, printing if the first string is longer, equal, or shorter than the second string.  
**Answer:**  
if len(stringl) > len(string2):  
print “Stringl is longer than string2.”  
elif len(stringl) < len(string2):  
print “String1 is shorter than string2.”  
else:  
print “String1 is the same length as string2.”

Question 14: When comparing two d^tes, what method is used?  
**Answer:**  
Date.toordinal( ) Otherwise, Python would compare the dates by their object address.

Question 15: In comparing dates and DateTime objects, what happens when one comparand is naive and the other aware?  
**Answer:**  
A TypeError is raised.

Question 16: What happens when you try to compare a DateTime object with other classes of objects?  
**Answer:**  
A TypeError is raised.

Question 17: When are dictionaries considered equal?  
**Answer:**  
If and only if their sorted lists compare equally.

Question 18: How is collection membership determined?  
**Answer:**  
Using the in and not in operators.

Question 19: Illustrate how collection membership determination would be written.  
**Answer:**  
if x in collection:  
print “It is in the collection”  
else:  
print “Not in the collection.”

Question 20: How is object identity tested? Illustrate with an example.  
**Answer:**  
Using the is and is not operators.  
if x is objecttype:  
print “x is the type you thought it was.  
else:  
print “x isn’t an objecttype.”

 Python - Conditions and Branching

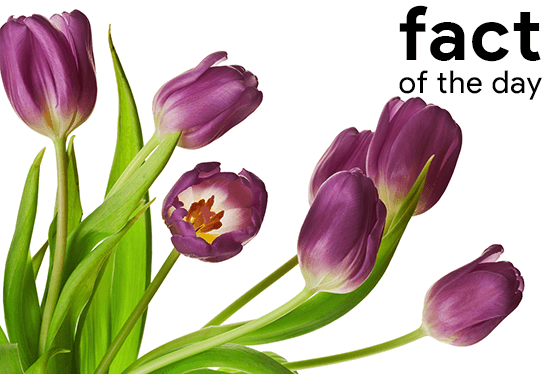
|  |
| --- |
| 1. List the 3 keywords used for conditional execution? A) = = != <> B) if else elif C) and or not  Answer: if else elif |
| 2. List the 6 conditional operators? A) and or not xor neg pos B) if else elif and or not C) > < >= <= == !=  Answer: > < >= <= == != |
| 3. What is the name of the operator used to assign 1 of 2 values depending on a condition? A) logical operator B) conditional operator C) ternary operator  Answer: ternary operator |
| 4. List the 3 logical operators? A) and or not B) = = != <> C) and xor not  Answer: and xor not |
| 5. Write code that adds num\_1 to num\_2 and saves the result to sum? A) sum = num\_1 + num\_2 B) sum = add(num\_1, num\_2) C) sum <= num\_1 + num\_2  Answer: sum = num\_1 + num\_2 |
| 6. What operator returns the remainder of a division? A) % B) / C) div  Answer: % |
| 7. Use modulus in a condition to test for an even value? A) if (i % 2) == 0 B) if (i / 2) == 0 C) if (i div 2) == 0  Answer: if (i % 2) == 0 |
| 8. How do you cast num\_1 from a string to an int? A) num\_1 = str(num\_1) B) num\_1 = integer(num\_1) C) num\_1 = int(num\_1)  Answer: num\_1 = int(num\_1) |
| 9. Show the part between curly brackets you would use to format a float to show only 2 decimal values? A) {.2f} B) {:.2f} C) {:.2d}  Answer: {:.2f} |
| 10. Write the print function using format to output this for example 1 + 2 = 3? A) format("1 + 2 = 3") B) print(format("1 + 2 = 3")) C) print("{} + {} = {}".format(1,2,3))  Answer: print("{} + {} = {}".format(1,2,3)) |

4.looping questions

 thon - Loops

|  |
| --- |
| 1. What keyword is used for looping? A) while B) for C) loop  Answer: for |
| 2. What function can generate a list of numbers? A) for B) list C) range  Answer: range |
| 3. Name the 2 keywords used for looping? A) while loop B) for loop C) for while  Answer: for while |
| 4. What module is used for generating random values? A) math B) random C) randrange  Answer: random |
| 5. What keyword is used to skip back to the beginning of a loop? A) break B) while C) continue  Answer: continue |
| 6. What keyword is used to end looping completely? A) end B) break C) continue  Answer: break |
| 7. Use range to generate a list from 1 through 5? A) range(1, 5) B) range(5) C) range(1,6)  Answer: range(1,6) |
| 8. Select the code needed to generate a random number between 1 through 50 and assign it to rand\_num? A) random.randrange(1, 51) B) random.randrange(1, 50) C) random(1, 50)  Answer: random.randrange(1, 51) |
| 9. What numbers does the following Range generate: range(5,10)? A) 5,6,7,8,9,10 B) 6,7,8,9,10 C) 5,6,7,8,9  Answer: 5,6,7,8,9  Explanation: Range starts at 5 and goes up to (but not including) 10 |
| 10. What numbers does the following range generate range(3)? A) 3 B) 1,2,3 C) 0,1,2,3 D) 0,1,2  Answer: 0,1,2  Explanation: Range will start at 0 and go up to (but not including) 3 |
| 11. What is printed out after the following code nums = range(1,5) print(nums)? A) 1,2,3,4 B) range(1,5)  Answer: range(1,5)  Explanation: Printing range does not print out all the numbers in the range. We have to iterate over the range with a loop to print the numbers. |
| 12. What numbers does the following range generate range(8,0,-2): A) 0,2,4,6 B) 8,6,4,2 C) 8,6,4,2,0  Answer: 8,6,4,2  Explanation: The 3rd argument(step) indicates that range should work backwards from 8, moving 2 numbers at a time. |
| 13. What does the following loop do?  i = 1 while i < 5: i + i print(i)  A) It prints 1,2,3,4,5 and then exits B) It prints 1,2,4 then exits C) It prints 1 forever  Answer: It prints 1 forever  Explanation: Don't forget to add the + before the = sign when incrementing! |
| 14. What does the following loop do?  # print 1 to 5 i = 0 while i <= 5: i =+ 1 print(i)  A) It prints 1 through 5 because it increments i each time B) It prints 0 through 5 because i starts at zero C) It prints 1 forever because there is a typo with the increment which should be += instead of =+. D) It prints zero through four because i starts at zero and goes up until 5.  Answer: It prints 1 forever because there is a typo with the increment which should be += instead of =+. |
| 15. What can we do to get out of the infinite loop below?  #this code runs forever... x = 0 while x != 11: x += 2 print(x)  A) Change the condition to x ! = 10 B) Change the condition to x < 11 C) Add logic that says if x == 10: break D) Press control + C while your program is running to kill it E) All of the above  Answer: All of the above |
| 16. For loops are used to loop over: A) Loopy objects B) Numbers C) Iterable objects D) Lines of code  Answer: Iterable objects |
| 17. What is the tradeoff when using while loops for looping? A) While loops are more flexible since you explicitly set the start and end conditions, but they require more setup than for loops B) While loops are faster than for loops but require more setup C) While loops can loop infinitely and give you more control, but they cannot be used to iterate over iterable objects  Answer: While loops are more flexible since you explicitly set the start and end conditions, but they require more setup than for loops |
| 18. What does the break keyword do? A) Exits the loop at the beginning of the next iteration B) Breaks your code C) Exits the loop at the end of this iteration D) Exits the loop immediately  Answer: Exits the loop immediately  Explanation: break is the fastest way to get out of a loop -- it won't run any code after the break |
| 19. What does the following code print?  for x in range(5): print(x)  A) 1 2 3 4 5 B) 1 2 3 4 C) 0 1 2 3 4  Answer: 0 1 2 3 4  Explanation: Python ranges start at zero by default and count up to, but not including, the end number. |

[](https://gknxt.com/quoteoftheday/)

[](https://gknxt.com/factoftheday/)

 thon - Loops

|  |
| --- |
| 1. What keyword is used for looping? A) while B) for C) loop  Answer: for |
| 2. What function can generate a list of numbers? A) for B) list C) range  Answer: range |
| 3. Name the 2 keywords used for looping? A) while loop B) for loop C) for while  Answer: for while |
| 4. What module is used for generating random values? A) math B) random C) randrange  Answer: random |
| 5. What keyword is used to skip back to the beginning of a loop? A) break B) while C) continue  Answer: continue |
| 6. What keyword is used to end looping completely? A) end B) break C) continue  Answer: break |
| 7. Use range to generate a list from 1 through 5? A) range(1, 5) B) range(5) C) range(1,6)  Answer: range(1,6) |
| 8. Select the code needed to generate a random number between 1 through 50 and assign it to rand\_num? A) random.randrange(1, 51) B) random.randrange(1, 50) C) random(1, 50)  Answer: random.randrange(1, 51) |
| 9. What numbers does the following Range generate: range(5,10)? A) 5,6,7,8,9,10 B) 6,7,8,9,10 C) 5,6,7,8,9  Answer: 5,6,7,8,9  Explanation: Range starts at 5 and goes up to (but not including) 10 |
| 10. What numbers does the following range generate range(3)? A) 3 B) 1,2,3 C) 0,1,2,3 D) 0,1,2  Answer: 0,1,2  Explanation: Range will start at 0 and go up to (but not including) 3 |
| 11. What is printed out after the following code nums = range(1,5) print(nums)? A) 1,2,3,4 B) range(1,5)  Answer: range(1,5)  Explanation: Printing range does not print out all the numbers in the range. We have to iterate over the range with a loop to print the numbers. |
| 12. What numbers does the following range generate range(8,0,-2): A) 0,2,4,6 B) 8,6,4,2 C) 8,6,4,2,0  Answer: 8,6,4,2  Explanation: The 3rd argument(step) indicates that range should work backwards from 8, moving 2 numbers at a time. |
| 13. What does the following loop do?  i = 1 while i < 5: i + i print(i)  A) It prints 1,2,3,4,5 and then exits B) It prints 1,2,4 then exits C) It prints 1 forever  Answer: It prints 1 forever  Explanation: Don't forget to add the + before the = sign when incrementing! |
| 14. What does the following loop do?  # print 1 to 5 i = 0 while i <= 5: i =+ 1 print(i)  A) It prints 1 through 5 because it increments i each time B) It prints 0 through 5 because i starts at zero C) It prints 1 forever because there is a typo with the increment which should be += instead of =+. D) It prints zero through four because i starts at zero and goes up until 5.  Answer: It prints 1 forever because there is a typo with the increment which should be += instead of =+. |
| 15. What can we do to get out of the infinite loop below?  #this code runs forever... x = 0 while x != 11: x += 2 print(x)  A) Change the condition to x ! = 10 B) Change the condition to x < 11 C) Add logic that says if x == 10: break D) Press control + C while your program is running to kill it E) All of the above  Answer: All of the above |
| 16. For loops are used to loop over: A) Loopy objects B) Numbers C) Iterable objects D) Lines of code  Answer: Iterable objects |
| 17. What is the tradeoff when using while loops for looping? A) While loops are more flexible since you explicitly set the start and end conditions, but they require more setup than for loops B) While loops are faster than for loops but require more setup C) While loops can loop infinitely and give you more control, but they cannot be used to iterate over iterable objects  Answer: While loops are more flexible since you explicitly set the start and end conditions, but they require more setup than for loops |
| 18. What does the break keyword do? A) Exits the loop at the beginning of the next iteration B) Breaks your code C) Exits the loop at the end of this iteration D) Exits the loop immediately  Answer: Exits the loop immediately  Explanation: break is the fastest way to get out of a loop -- it won't run any code after the break |
| 19. What does the following code print?  for x in range(5): print(x)  A) 1 2 3 4 5 B) 1 2 3 4 C) 0 1 2 3 4  Answer: 0 1 2 3 4  Explanation: Python ranges start at zero by default and count up to, but not including, the end number. |

# Python For Loop: Question and Answers

The for loop is a simple programming construct that repeats a statement or group of statements. In these questions, we look at the syntax and structure of the for loop in Python.

#### **1. What is the difference between a for loop and a while loop?**

A for loop is typically used when you know exactly how many times the loop needs to be repeated. A while loop is typically used when you don't know how many times the loop needs to be repeated.

A while loop repeats as long as its condition is true. For example, if a while loop says "while x == 5", then the line will execute as long as x equals five.



#### **2. What is the Python syntax for a for loop?**

The for loop is a programming construct that allows you to iterate over an arbitrary range of values, mapping them to the required actions.

It can be thought of as an extension of the mathematical notion of a for loop, which is defined as "a control structure that enables one to iterate (repeat) a process (such as counting or summing) while varying the process's start value (or its end point), step size, and/or direction."

- The syntax for Python for loops is:

for x in range(y): do something

#### **3. What are the advantages of using a for loop in Python?**

A for loop is a type of loop that executes a set of instructions repeatedly. It uses the following syntax: **for x in range(y): do something**

This type of loop has the following properties:

* For loops are good because they can make your code more readable and easier to follow.
* They also make sure that certain instructions are executed at least one time.

#### **4. What are the disadvantages of using a for loop in Python?**

The for-loop is one of the most basic constructs in programming. However, this construct has its own disadvantages.

1. If you don't know the upper-bound of your loop, you have to keep track of it by hand or break out of your loop early.
2. You can't use a break statement to escape from a nested loop.

#### **5. How can I use a break statement in my Python for loops?**

A break statement can be used to terminate a loop. It is often used when the programmer needs to stop iterating through the data and instead go back to checking for other conditions.

An example of such a situation is when we need to break out of a for loop if an exception occurs. This will allow us to take different paths depending on whether or not we want to handle that exception, or ignore it and continue with whatever else follows after the for-loop.

Code Example for Break Statement is as follows:

 animals = ["dog","cat","sheep","tiger","lion"]

 for x in animals:

 print (x)

 if x == "tiger":

 break;

# 20 Python Functions Interview Questions and Answers

#### **1. What are the two types of functions in Python?**

There are two types of functions in Python: built-in functions and user-defined functions. Built-in functions are functions that are already defined in the Python language, such as the print() function. User-defined functions are functions that are created by the user, and they can be created to do anything that the user wants them to do.

#### **2. Can you explain what a call graph is? How do you create one?**

A call graph is a visual representation of the relationships between the various functions in a Python program. It can be used to help debug code, optimize code, and understand code flow. To create a call graph, you can use the pycallgraph library.

#### **3. When should you use anonymous functions and when should you use regular ones?**

There is no definitive answer to this question, as it depends on the specific situation and what you are trying to accomplish. However, a general rule of thumb is that anonymous functions are best used for simple tasks that can be easily expressed in a single line of code. Regular functions, on the other hand, are better suited for more complex tasks that require multiple lines of code.

#### **4. Do Python functions have return values? If yes, then how many can they have?**

Yes, Python functions can have return values. They can have a single return value, or they can have multiple return values.

#### **5. Why does Python support both positional and keyword arguments to its functions?**

Python supports both positional and keyword arguments in order to give developers more flexibility when designing their functions. Positional arguments are those that are passed in by position, without explicitly specifying the parameter name. Keyword arguments are those that are passed in by explicitly specifying the parameter name. Python allows for both types of arguments so that developers can choose the approach that makes the most sense for their particular function.

#### **6. Is it possible for a function’s code to read from or write to variables defined outside that function? If yes, then how?**

Yes, it is possible for a function’s code to read from or write to variables defined outside that function. This is known as “accessing global variables.” To do this, the function must first use the “global” keyword to declare which variables it is accessing. For example:

global var1

var1 = 5

def func():

print(var1)

func() # Prints 5

#### **7. Can you explain what a closure is in Python? How do you implement closures?**

A closure is a function that remembers the values from the enclosing scope even when the program flow is no longer in that scope. Closures are implemented by creating a function that takes in one or more values from the enclosing scope and then returning a new function that uses those values.

#### **8. Can you explain what decorators are in Python?**

Decorators are a way to dynamically alter the behavior of a function. They are usually used as a way to add functionality to an existing function without having to modify the code of the function itself. Decorators are typically written in the form of a wrapper function.

#### **9. Is there any way to define static methods in Python? If yes, then how?**

Static methods are defined in Python by using the @staticmethod decorator. This decorator can be applied to any method, and will cause the method to be treated as a static method, even if it is not defined as such.

#### **10. Is it possible to pass a variable number of arguments to a function in Python? If yes, then how?**

Yes, it is possible to pass a variable number of arguments to a function in Python. This can be done using the \*args and \*\*kwargs parameters. \*args allows for a variable number of non-keyworded arguments to be passed to a function, while \*\*kwargs allows for a variable number of keyworded arguments to be passed.

#### **11. How would you convert JSON to Python objects?**

The json library in Python can help you convert JSON data into Python objects. The process is known as decoding. You can use the json.loads() function to decode JSON data. This function takes a JSON string and returns a Python object.

#### **12. Can you explain what recursion is and why it’s useful?**

Recursion is a function that calls itself. It’s useful because it allows you to break down a problem into smaller, more manageable pieces.

#### **13. What is tail-recursion?**

Tail-recursion is a type of recursion where the last statement in the function is a recursive call. This is important because it allows the interpreter to optimize the function by not having to keep track of the current state of the function, since it can simply jump to the beginning of the function and start again.

#### **14. Can you explain what memoization is and why it’s useful?**

Memoization is a technique for optimizing code by caching and reusing previously-computed results. It can be used to speed up calculations by avoiding redundant work, and it can also be used to make code more memory-efficient by storing the results of expensive function calls and avoiding the need to recalculate them each time the function is called.

#### **15. What is currying?**

Currying is the process of taking a function with multiple arguments and turning it into a function that takes a single argument. The single argument is then used to generate a new function that takes the next argument, and so on, until all arguments have been used. Currying is often used in functional programming to make code more concise.

#### **16. Can you give me some examples where I might want to use an assertion function?**

Assertions are often used in debugging to check for conditions that should never occur in normal execution. For example, if you are working with a list, you might want to assert that the list is never empty, or that every element in the list is greater than 0.

#### **17. Can you explain the difference between named parameters and keyword parameters in Python?**

In Python, named parameters are those that are assigned a name in the function definition, while keyword parameters are those that are assigned a value when the function is called. For example, in the following function definition, the parameter x is a named parameter, while the parameter y is a keyword parameter:

def func(x, y=5):  
pass

In this example, the parameter y will always have the value 5 when the function is called, unless a different value is explicitly provided.

#### **18. What’s the difference between positional parameters and default parameters in Python?**

Positional parameters are parameters that are required to be provided in order for the function to run. Default parameters are parameters that are not required to be provided, but have a default value that will be used if no other value is provided.

#### **19. How do you access the value returned by a function invoked in another function?**

You can access the value returned by a function invoked in another function by using the return statement. For example, if you have a function that calculates the sum of two numbers, you can access the value returned by the function by using the return statement.

#### **20. Can you give me some examples of monkeypatching in Python?**

Monkeypatching is the process of dynamically altering the behavior of a function or class at runtime. This can be useful for patching bugs, adding new features, or simply overriding the behavior of a function or class for testing purposes. Some examples of monkeypatching in Python include:

– Altering the behavior of a built-in function:

“`python  
>>> import builtins  
>>> def my\_print(\*args, \*\*kwargs):  
… print(‘Hello, world!’)  
…  
>>> builtins.print = my\_print  
>>> print(‘This will print “Hello, world!”‘)  
Hello, world!  
“`

– Altering the behavior of a method in a class:

“`python  
>>> class MyClass:  
… def my\_method(self):  
… print(‘Hello, world!’)  
…  
>>> def new\_method(self):  
… print(‘Goodbye, world!’)  
…  
>>> MyClass.my\_method = new\_method  
>>> MyClass().my\_method()  
Goodbye, world!  
“`