Unit I

Introduction

➢ Python Features

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation.

- **Easy to Learn and Use**
  Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.
- **Expressive Language**
  Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**. It will take only one line to execute, while Java or C takes multiple lines.
- **Interpreted Language**
  Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.
- **Cross-platform Language**
  Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.
- **Free and Open Source**
  Python is freely available for everyone. It is freely available on its official website www.python.org.
- **Object-Oriented Language**
  Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation,
- **Extensible**
  It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.
- **Large Standard Library**
  It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting.
- **Embeddable**
  The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

- **Dynamic Memory Allocation**
  In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to **x,** then we don't need to write **int x = 15.** Just write x = 15.

➢ Python Applications

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

- **Web Applications**
  We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc. One of Python web-framework named Django is used on **Instagram**.
- **Desktop GUI Applications**
  The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a **Tk GUI library** to develop a user interface.
- **Software Development**
  Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.
- **Scientific and Numeric**
  This is the era of Artificial intelligence where the machine can perform the task the same as the human. Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.
- **Business Applications**
  Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features.
- **Audio or Video-based Applications**
  Python is flexible to perform multiple tasks and can be used to create multimedia applications.

➢ Python History and Versions

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland.

- In February 1991, Guido Van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
- ABC programming language is said to be the predecessor of Python language, which was capable of Exception Handling and interfacing with the Amoeba Operating System.
- The following programming languages influence Python:
- ABC language.
- Modula-3
- There is a fact behind choosing the name Python. **Guido van Rossum** was reading the script of a popular BBC comedy series "**Monty Python's Flying Circus**". It was late on-air 1970s.
- Van Rossum wanted to select a name which unique, sort, and little-bit mysterious. So he decided to select naming Python after the **"Monty Python's Flying Circus"** for their newly created programming language.
- The comedy series was creative and well random. It talks about everything. Thus it is slow and unpredictable, which made it very interesting.
- Python is also versatile and widely used in every technical field, such as Machine Learning, Artificial Intelligence, Web Development, Mobile Application, Desktop Application, Scientific Calculation, etc.

- **Python Version**
    - Python programming language is being updated regularly with new features and supports. There are lots of update in Python versions, started from 1994 to current release.
    - There are two major Python versions- Python 2 and Python 3
    - On 16 October 2000, Python 2.0 was released with many new features.
    - On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

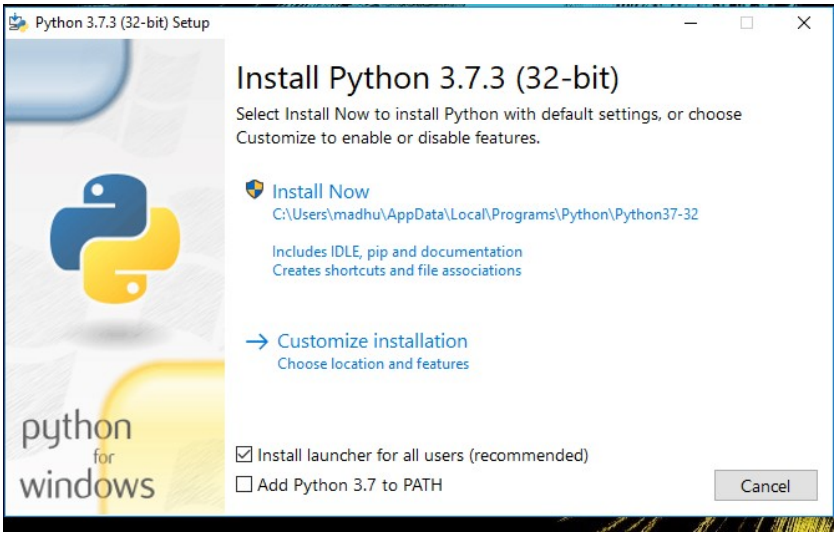
➢ Installation of Python

Step 1: Download the Python Installer binaries

- Open the official Python website in your web browser. Navigate to the Downloads tab for Windows.
- Choose the latest Python 3 release. In our example, we choose the latest Python 3.7.3 version.
- Click on the link to download **Windows x86 executable installer** if you are using a 32-bit installer. In case your Windows installation is a 64-bit system, then download **Windows x86-64 executable installer**.

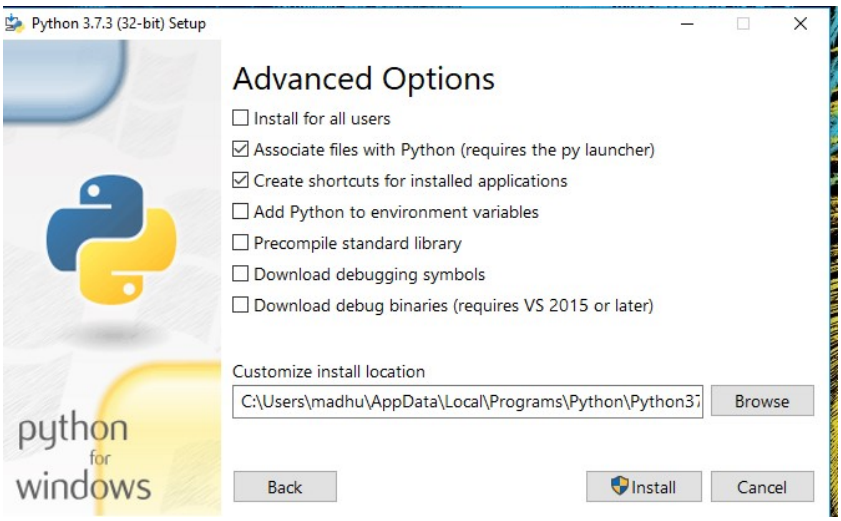
Step 2: Run the Executable Installer

- Once the installer is downloaded, run the Python installer.
- Check the **Install launcher for all users** check box. Further, you may check the **Add Python 3.7 to path** check box to include the interpreter in the execution path.



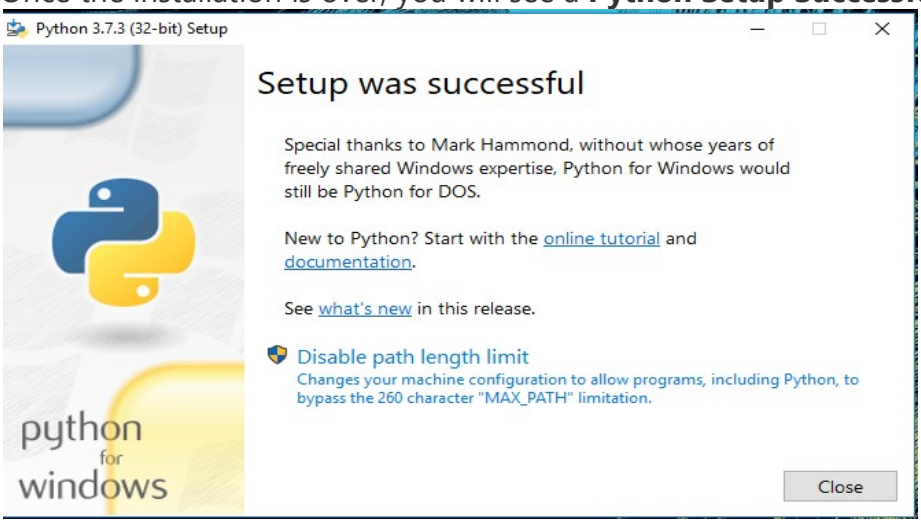
- Select **Customize installation**. Choose the optional features by checking the following check boxes:

    - Documentation
    - pip
    - tcl/tk and IDLE (to install tkinter and IDLE)
    - Python test suite (to install the standard library test suite of Python)
    - Install the global launcher for `.py` files. This makes it easier to start Python
    - Install for all users



- Click **Next**.8. This takes you to **Advanced Options** available while installing Python. Here, select the **Install for all users** and **Add Python to environment variables** check boxes. Optionally, you can select the **Associate files with Python**, **Create shortcuts for installed applications** and other advanced options. Make note of the python installation directory displayed in this step. You would need it for the next step. After selecting the Advanced options, click **Install** to start installation.

- Once the installation is over, you will see a **Python Setup Successful** window.



Python Basics

➢ Python Keywords and Identifiers

- ### Python Keywords

  Keywords are predefined, reserved words used in Python programming that have special meanings to the compiler.

  We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the Python language.

  All the keywords except True, False and None are in lowercase and they must be written as they are. The list of all the keywords is given below.

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
And        continue   for        lambda     try
As         def        from       nonlocal   while
Assert     del        global     not        with
Async      elif       if         or         yield
```

- ### Python Identifiers

Identifiers are the name given to variables, classes, methods, etc. For example,

```
language = 'Python'
```

Here, language is a variable (an identifier) which holds the value 'Python'.

We cannot use keywords as variable names as they are reserved names that are built-in to Python. For example,

```
continue = 'Python'
```

The above code is wrong because we have used continue as a variable name. To learn more about variables, visit Python Variables.

- ### Rules for Naming an Identifier
  - Identifiers cannot be a keyword.
  - Identifiers are case-sensitive.
  - It can have a sequence of letters and digits. However, it must begin with a letter or The first letter of an identifier cannot be a digit.
  - It's a convention to start an identifier with a letter rather
  - Whitespaces are not allowed.
  - We cannot use special symbols like **!**, **@**, **#**, **$**, and so on.

➢ Statements and Expressions

A combination of operands and operators is called an **expression**. The expression in Python produces some value or result after being interpreted by the Python interpreter. An expression in Python is a combination of operators and operands.
An example of expression can be: x = x + 10. In this expression, the first 10 is added to the variable x. After the addition is performed, the result is assigned to the variable x.
Example:

```
x = 25          # a statement
x = x + 10      # an expression

print(x)
```

➢ Variables

Variables are containers for storing data values.

Creating Variables

Python has no command for declaring a variable.
A variable is created the moment you first assign a value to it.
Example

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.
Example

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

➢ Operators

Operators are used to perform operations on variables and values.
Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic Operators + , -, *, /, %, //,**
Relational/Comparision Operator >, <, >=, <=, ==, !=
Logical Operator and, or , not
Bitwise Operator &, |, ^,~, << , >>
Assignment Operator =, +=, -=,*=,/=, %=, //=, **=, &=, |=,^=,<<=,>>=
Special Operator Identity Operator is, is not
Membership Operator in , not in

➢ Precedence and Association

Operators	Meaning
`()`	Parentheses
`**`	Exponent
`+x, -x, ~x`	Unary plus, Unary minus, Bitwise NOT
`*, /, //, %`	Multiplication, Division, Floor division, Modulus
`+, -`	Addition, Subtraction
`<<, >>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==, !=, >, >=, <, <=, is, is not, in, not in`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

In Python, operators have different levels of precedence, which determine the order in which they are evaluated. When multiple operators are present in an expression, the

ones with higher precedence are evaluated first. In the case of operators with the same precedence, their associativity comes into play, determining the order of evaluation.

Operator Precedence and Associativity in Python

This table lists all operators from the highest precedence to the lowest precedence.

Precedence	Operators	Description	Associativity
1	**()**	Parentheses	Left to right
2	**x[index]**, **x[index:index]**	Subscription, slicing	Left to right
3	await x	Await expression	N/A
4	**	Exponentiation	Right to left
5	**+x**, **-x**, **~x**	Positive, negative, bitwise NOT	Right to left
6	*, **@**, */*, **//**, **%**	Multiplication, matrix, division, floor division, remainder	Left to right
7	**+**, **–**	Addition and subtraction	Left to right
8	**<<**, **>>**	Shifts	Left to right
9	**&**	Bitwise AND	Left to right
10	**^**	Bitwise XOR	Left to right
11	**|**	Bitwise OR	Left to right

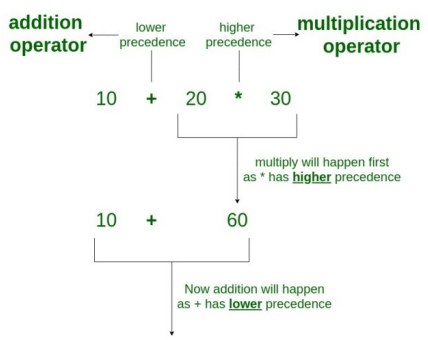
Precedence	Operators	Description	Associativity
12	**in**, **not in**, **is**, **is not**, **<**, **<=**, **>**, **>=**, **!=**, **==**	Comparisons, membership tests, identity tests	Left to Right
13	**not x**	Boolean NOT	Right to left
14	**and**	Boolean AND	Left to right
15	**or**	Boolean OR	Left to right
16	**if-else**	Conditional expression	Right to left
17	**lambda**	Lambda expression	N/A
18	**:=**	Assignment expression (walrus operator)	Right to left

Precedence of Python Operators

This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Example

10 + 20 * 30



10 + 20 * 30 is calculated as **10 + (20 * 30)** and not as **(10 + 20) * 30**

Python Code of the above Example

```
# Precedence of '+' & '*'
expr = 10 + 20 * 30

print(expr)
```

Output

610

Precedence of Logical Operators in Python

In the given code, the 'if' **block is executed even if the age is 0. Because the precedence of logical** 'and' **is greater than the logical** 'or'**.**

```
# Precedence of 'or' & 'and'
name = "Alex"
age = 0

if name == "Alex" or name == "John" and age >= 2:
    print("Hello! Welcome.")
else:
    print("Good Bye!!")
```

Output

Hello! Welcome.

Hence, To run the '**else**' block we can use parenthesis**( )** as their precedence is highest among all the operators.

```
# Precedence of 'or' & 'and'
name = "Alex"
age = 0

if (name == "Alex" or name == "John") and age >= 2:
    print("Hello! Welcome.")
else:
    print("Good Bye!!")
```

Output

Good Bye!!

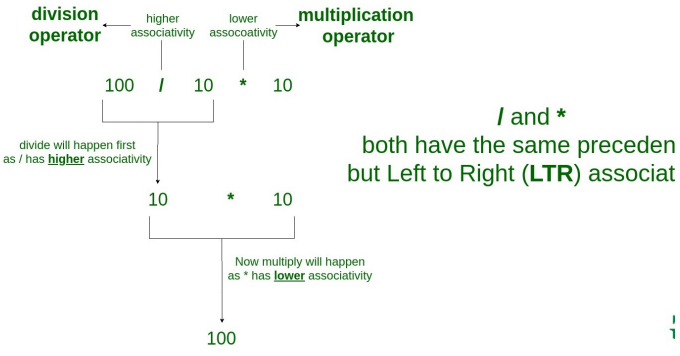
Associativity of Python Operators

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left **to** Right **or from** Right **to** Left**.**

Example

In this code, '*' and '/' have the same precedence and their associativity is Left to Right, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Operator Associativity



Code

```
# Left-right associativity
# 100 / 10 * 10 is calculated as
# (100 / 10) * 10 and not
# as 100 / (10 * 10)
print(100 / 10 * 10)

# Left-right associativity
# 5 - 2 + 3 is calculated as
# (5 - 2) + 3 and not
# as 5 - (2 + 3)
print(5 - 2 + 3)
 # left-right associativity
print(5 - (2 + 3))
 # right-left associativity
# 2 ** 3 ** 2 is calculated as
# 2 ** (3 ** 2) and not
# as (2 ** 3) ** 2
print(2 ** 3 ** 2)
```

Output

100
6
0
512

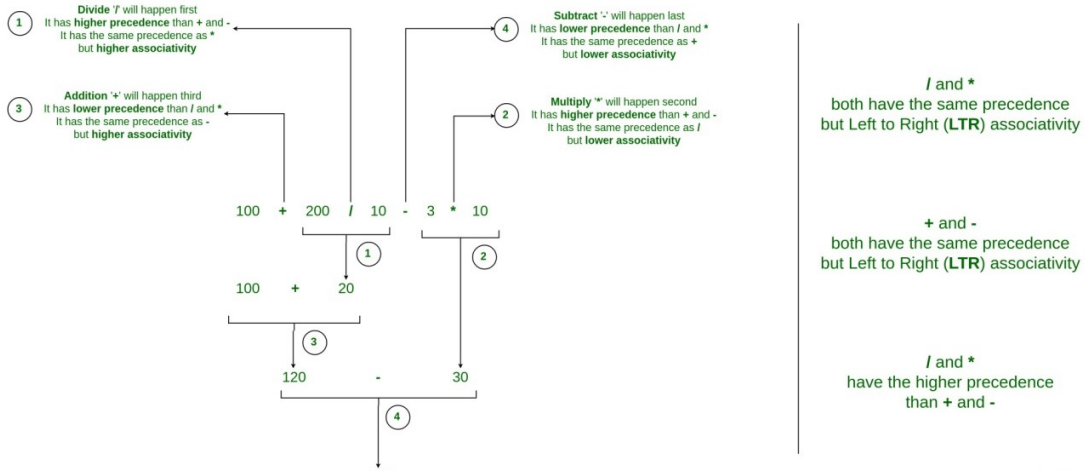
Operators Precedence and Associativity in Python

Operators Precedence and Associativity are two main characteristics of operators that determine the evaluation order of sub-expressions in the absence of brackets.

Example

100 + 200 / 10 - 3 * 10

Operator Precedence and Associativity



100 + 200 / 10 - 3 * 10 is calculated as 100 + (200 / 10) - (3 * 10) and not as (100 + 200) / (10 - 3) * 10

Python Code of the above Example

Output

90.0

➢ Data Types

Variables can hold values, and every value has a data-type. Python is a dynamically typed language.The interpreter implicitly binds the value with its type.

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.

```
a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```

output:

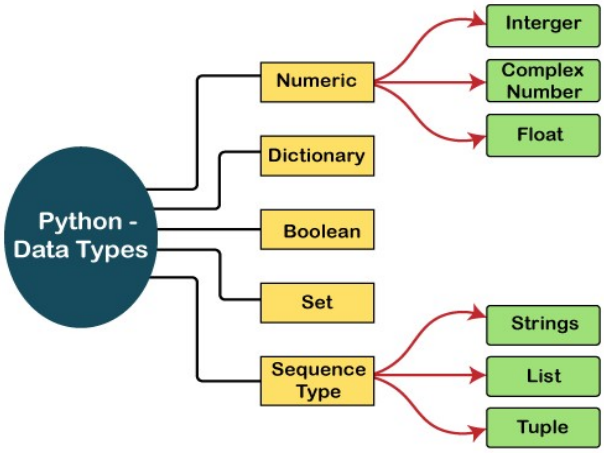
```
<type 'int'>
<type 'str'>
<type 'float'>
```

Standard data types

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

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

- Numbers
- Sequence Type
- Boolean
- Set
- Dictionary



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;

```
a = 5
print("The type of a", type(a))
 b = 40.5
print("The type of b", type(b))
  c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

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

Float - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.

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'.

Example - 1

```
str = "string using double quotes"
print(str)
s = """A multiline
string"'
print(s)
```

Example - 2

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

➢ Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Example

Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

➢ Comments

We may wish to describe the code we develop. We might wish to take notes of why a section of script functions, for instance. We leverage the remarks to accomplish this. Formulas, procedures, and sophisticated business logic are typically explained with comments. The Python interpreter overlooks the remarks and solely interprets the script when running a program. Single-line comments, multi-line comments, and documentation strings are the 3 types of comments in Python.

Types of Comments in Python

In Python, there are 3 types of comments. They are described below:

Single-Line Comments

Single-line remarks in Python have shown to be effective for providing quick descriptions for parameters, function definitions, and expressions. A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation. Consider the accompanying code snippet, which shows how to use a single line comment:

Code

```
# This code is to show an example of a single-line comment
print( 'This statement does not have a hashtag before it' )
```

Multi-Line Comments

Python does not provide the facility for multi-line comments. However, there are indeed many ways to create multi-line comments.

With Multiple Hashtags (#)

In Python, we may use hashtags (#) multiple times to construct multiple lines of comments. Every line with a (#) before it will be regarded as a single-line comment.

Code

```
# it is a
# comment
# extending to multiple lines
```

Using String Literals

Because Python overlooks string expressions that aren't allocated to a variable, we can utilize them as comments.

Code

```
'it is a comment extending to multiple lines'
```

➢ Built-in Functions- Console Input and Console Output

Python input()

input() Syntax

The syntax of input() function is:

Example 1: How input() works in Python?

```
# get input from user

inputString = input()
print('The inputted string is:', inputString)
```

Run Code

Output

```
Python is interesting.
The inputted string is: Python is interesting
```

Example 2: Get input from user with a prompt

```
# get input from user
inputString = input('Enter a string:')
print('The inputted string is:', inputString)
```

Run Code

Output

```
Enter a string: Python is interesting.
The inputted string is: Python is interesting
```

Python Output Function:
The print() function prints the specified message to the screen, or other standard output device.
The message can be a string, or any other object, the object will be converted into a string before written to the screen.
Syntax

```
print(object(s), sep=separator, end=end, file=file, flush=flush)
```

Parameter Values

Parameter	Description
object(s)	Any object, and as many as you like. Will be converted to string before printed
sep='*separator*'	Optional. Specify how to separate the objects, if there is more than one. Default is ' '
end='*end*'	Optional. Specify what to print at the end. Default is '\n' (line feed)
file	Optional. An object with a write method. Default is sys.stdout
flush	Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False

➢ Type Conversions

Data Type Conversion in Python

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	**int(x [,base])**Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])**Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)**Converts x to a floating-point number.
4	**complex(real [,imag])**Creates a complex number.
5	**str(x)**Converts object x to a string representation.

Python Type Conversion

In programming, type conversion is the process of converting data of one type to another. For example: converting int data to str.
There are two types of type conversion in Python.

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

Python Implicit Type Conversion

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

Example 1: Converting integer to float

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

```
integer_number = 123
float_number = 1.23
new_number = integer_number + float_number
# display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))
```

Output

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

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

Explicit Type Conversion

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

We use the built-in functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Example 2: Addition of string and integer Using Explicit Conversion

```
num_string = '12'
num_integer = 23
print("Data type of num_string before Type Casting:",type(num_string))
# explicit type conversion
num_string = int(num_string)
print("Data type of num_string after Type Casting:",type(num_string))
num_sum = num_integer + num_string
print("Sum:",num_sum)
print("Data type of num_sum:",type(num_sum))
```

Run Code

Output

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

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

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

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

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

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

➢ Python Libraries; Importing Libraries with Examples

Python has created several open-source libraries, each with its root source. A library is an initially merged collection of code scripts that can be used iteratively to save time. It's similar to a physical library in that it holds reusable resources, as the name implies.

A Python library is also a group of interconnected modules. It contains code bundles that can be reused in a variety of programs. It simplifies and facilitates Python programming for programmers. Because then we won't have to write the very same code for different programs. Machine learning, computer science, data visualization, and other fields rely heavily on Python libraries.

Standard Libraries of Python

Python's syntax, semantics, and tokens are all contained in the Python Standard Library. It comes with built-in modules that give the user access to basic functions like I/O and a few other essential modules. The Python libraries have been written in the C language for the most part. There are over 200 core modules in the Python standard library. Python is a powerful programming language because of all of these factors. The Python Standard Library is extremely important. Programmers won't be able to use Python's features unless they have it. Apart from that, Python has several libraries that make a programmer's life easier. Let us study some of the most popular libraries:

Math Module in Python

Python has a built-in math module. It is a standard module, so we don't need to install it separately. We only have to import it into the program we want to use. We can import the module, like any other module of Python, using import math to implement the functions to perform mathematical operations.

```
# This program will show the calculation of square root using the math module
# importing the math module
import math
print(math.sqrt( 9 ))
```

Output:

```
3.0
```

Code

```
# importing the required library
import math

# printing the value of Euler's number using the math module
print( "The value of Euler's Number is: ", math.e )
```

Output:

```
1. The value of Euler's Number is:  2.718281828459045
```

Python Control Flow

➢ Control Flow 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.

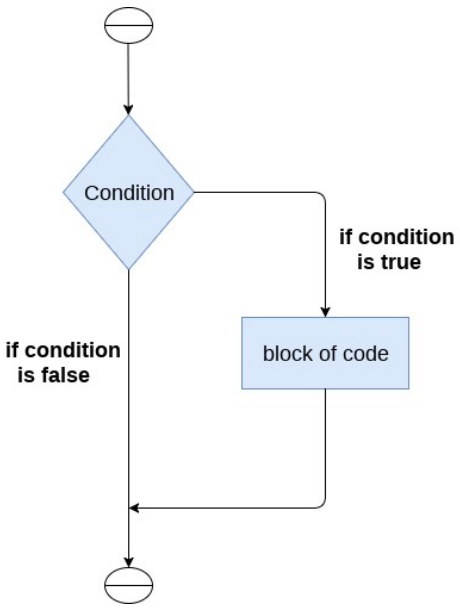
Indentation in Python

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.

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.

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.

```
if expression:
    statement
```

Example1:

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

Example 2:

```
a = int(input("Enter a? "));
b = int(input("Enter b? "));
c = int(input("Enter c? "));
if a>b and a>c:
    print("a is largest");
if b>a and b>c:
    print("b is largest");
if c>a and c>b:
    print("c is largest");
```

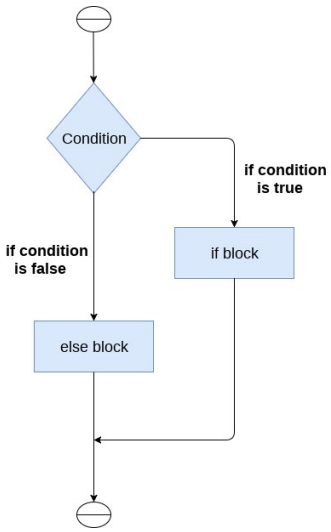
THE IF-ELSE STATEMENT

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.

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```



Example1:

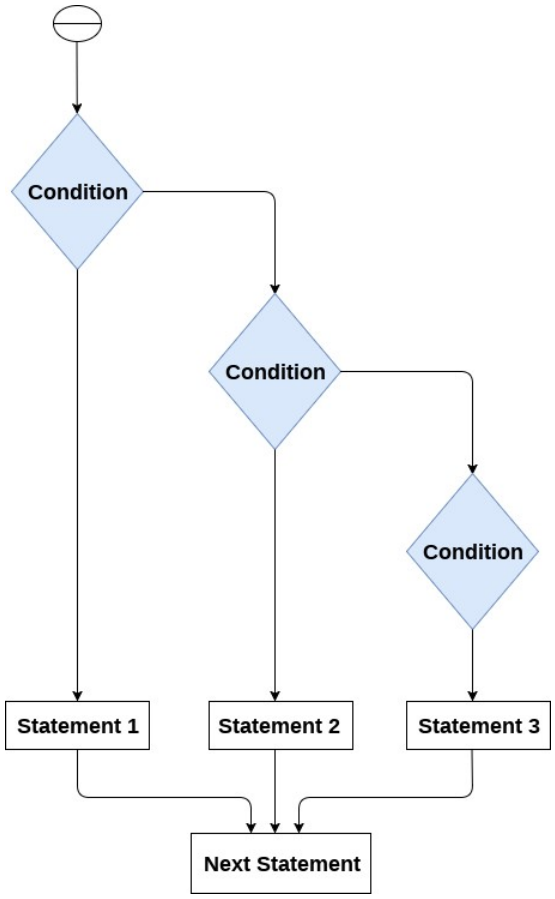
```
age = int (input("Enter your age? "))
if age>=18:
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

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.




```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```

Example1:

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

➢ 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

```
for value in sequence:
    { 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.

Example:

```
print("welcome")
for x in range(3):
        print(x)
print("Good morning college")
```

output:

```
welcome
0
1
2
Good morning college
```

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).

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

```
# Python program to show the working of range() function
print(range(15))
 print(list(range(15)))
print(list(range(4, 9)))
print(list(range(5, 25, 4)))
```

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:

```
while <condition>:
    { 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

```
# Python program to show how to use a while loop
counter = 0
# Initiating the loop
while counter < 10: # giving the condition
    counter = counter + 3
    print("Python Loops")
```

EXIT() FUNCTION

Syntax of exit() in Python

We can use the in-built **exit()** function to quit and come out of the execution loop of the program in Python.

Syntax:

```
exit()
```

exit() is defined in site module and it works only if the **site module** is imported so it should be used in the interpreter only.

PYTHON FUNCTIONS

What is a function?

A function is a piece of code that is organized and reused that performs one, similar action. Functions offer greater modularity to your application, and a higher amount of code reuse.

As you have guessed, Python gives you many built-in functions such as print (), etc. However, you can also design custom functions. These functions are referred to as user-defined functions.

Defining a Function

Functions can be defined to give the necessary functionality. These are the basic rules for defining a function in Python.

- Function blocks start with the word "def" then followed by the function name with the parentheses (). Function blocks begin with the keyword def followed by parentheses (())
- All input parameters or arguments must be enclosed in these parentheses. It is also possible to define parameters within these parentheses.

- The first sentence of a function could be an optional one - the documentation string for the function, or the docstring.
- Code block in each function begins with the letter ":" (:) and is indented.
- The expression return ends a function, returning an expression to the caller. Return statements that do not have arguments are equivalent to return None.

Syntax

```
def functionname( parameters ):

    # Function_docstring

    function_suite

    return (expression)
```

In default, parameters behave in a locational manner, and you must notify that they are in the same order that they were created.

Example

The function that follows uses a string input parameter and prints it out on a regular display.

```
def printme( str ):
    print (str)
    return
```

The function printme will take a string argument and return result to print on screen.

Calling a Function

Defining a function simply provides it with a name that defines the parameters that will be used in the function, and the structure of the code blocks.
When the structure of a program is completed, you can run it using another program and directly from the Python prompt. Here is an example of how to invoke the printme() function.

Example

```
def printme( str ):
        print (str)
        return;
printme("First Call")
printme("Second Call")
```

Output

```
First Call
Second Call
```

Return Value

Return statements are used to stop the operation of the called in a way that "returns" the result (value of the expression after "return") to the user. The following statements that follow the return statements are not executed. If the returned statement does not contain an expression, then number None will be returned. The return statement is not used in conjunction with other functions.

Example1

```
def add(a, b):
        return a + b
def is_true(a):
        return bool(a)
res = add(5, 4)
print("Result of of adding function is {}".format(res))
res = is_true(2<5)
print("Result of is_true function is {}".format(res))
```

Output

```
Result of of adding function is 9
Result of is_true function is True
```

Example2

```
def multiply(a, b):
        return a * b
res = multiply(5, 4)
print("Multiplication result is {}".format(res))
```

Output

```
Multiplication result is 20
```

Example3

```
def divide(a, b):
        return a / b
res = divide(5, 4)
print("Division result is {}".format(res))
```

Output

```
Division result is 1.25
```

Summary

- A function is a piece of code that is organized and reused that performs one, similar action.
- Function blocks start with the word "def" then followed by the function name with the parentheses ().
- Code block in each function begins with the letter ":" (:) and is indented.
- Defining a function simply provides it with a name that defines the parameters that will be used in the function, and the structure of the code blocks.
- Return statements are used to stop the operation of the called in a way that "returns" the result.
- If the returned statement does not contain an expression, then number None will be returned.

Arguments

The terms argument and parameter can refer to the same thing: data that is passed to functions.

From the perspective of a function:

- A parameter is a variable that is listed inside the parentheses within the definition of the function.
- A parameter is the number of arguments that are sent to the function whenever it is invoked.
- A function by default must be invoked with the proper number of arguments. This means that if your program requires two arguments, you must call the function using two arguments, not more and not less.

Arbitrary Arguments

Sometimes, we do not have a clear idea of the number of arguments that can be passed to the function. Python allows us to deal with this type of scenario by making functions that can be called with any quantity of arguments.

In the definition of the function, we put an underscore (*) before the parameter's name to indicate this type of argument.

Example1

```
def greet(*names):
        for name in names:
                print("Hello", name)
greet("India", "Dubai", "UK", "Canada")
```

Output

```
Hello India
Hello Dubai
Hello UK
Hello Canada
```

Example2

```
def fruits(*fnames):
        for fruit in fnames:
        print(fruit)
fruits("Orange","Banana","Apple","Grapes")
```

Output

Orange
Banana
Apple
Grapes

Example3

```
list_to_sum = [2,4,6,8,10]
def sum_function(numbers):
        total = 0
        for i in numbers:
                total += i
        return total
print(sum_function(list_to_sum))
```

Output

30

Keyword Arguments

When we invoke a function using some parameters, the values are assigned to arguments following their positions. Python lets functions are invoked using keyword arguments. When calling functions in this manner it is possible to change the sequence (position) of the arguments can be altered. The calls to these functions are valid and yield the same results.

Example1

```
def my_function(child3, child2, child1):
        print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Output

The greatest number is 30
The above program can be run in python IDE as shown in the following image.

Example2

```
def my_function(name1, name2, name3):
        print("The second name is " + name2)
my_function(name1 = om', name2='sai', name3 = 'ram')
```

Output

The second name is sai

Example3

```
def my_function(name, age):
        print(name + " have age " + age + " years")
my_function(name= "Ali", age = "34")
```

Output

Ali have age 34 years

Summary

- A parameter is a variable that is listed inside the parentheses within the definition of the function.
- A parameter is the number of arguments that are sent to the function whenever it is invoked.
- In the definition of the function, we put an underscore (*) before the parameter's name to indicate this type of argument.

Namespace and Scope in Python

In Python, we deal with variables as well as libraries, functions, modules, etc. There is a good chance that the variable's name you are planning to use is being used as the name of a different variable, or even as the name of an additional function or a different method. In this situation, it is necessary to know how these names are handled through a Python program. This is known as a namespace.

The following are the three namespace categories. The three categories of namespace are:

- **Local Namespace**: Each of the names of variables and functions that a program declares are stored within this namespace. The namespace is in existence for as long as the program is running. < /p>
- **Global Namespace**: Global Namespace contains the names of the variables and functions part of the modules utilized in the Python program. It includes all names that make up the Local namespace.
- **Built-in Namespace**: It is the top degree of the namespace that is made available by default names that are part of the Python interpreter, which can be loaded to serve as the programming environment. It includes Global Namespace which in turn includes names in the local area.

Scope in Python

The namespace can last for a long time if it is accessible. It is also referred to as the scope. The scope also depends on the region of coding where the object or variable is situated. It is evident in the program below how variables declared within an inner loop

can be accessed for the out loop, but not vice versa. Please note that the names of outer functions are a part of the global variable.

Example1

```
count = 11
def some_method():
        global count
        count = count + 1
        print(count)
some_method()
```

Output

12

Example2

```
def outer_function():
        a = 10
        def inner_function():
                a = 20
                print('a =', a)
        inner_function()
        print('a =', a)
a = 30
outer_function()
print('a =', a)
```

Output

a = 20
a = 10
a = 30

Summary

- Each of the names of variables and functions that a program declares are stored within this namespace is called local namespace.
- Global Namespace contains the names of the variables and functions part of the modules utilized in the Python program.
- Built in namespace is the top degree of the namespace that is made available by default names that are part of the Python interpreter, which can be loaded to serve as the programming environment.
- The namespace can last for a long time if it is accessible. It is also referred to as the scope.
- The scope also depends on the region of coding where the object or variable is situated.

Python Lists

In short, a list is a collection of arbitrary objects, somewhat akin to an array in many other programming languages but more flexible. Lists are defined in Python by enclosing a comma-separated sequence of objects in square brackets (`[]`), as shown below:

```
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> print(a)
['foo', 'bar', 'baz', 'qux']
>>> a
['foo', 'bar', 'baz', 'qux']
```

The important characteristics of Python lists are as follows:

- Lists are ordered.
- Lists can contain any arbitrary objects.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.

Each of these features is examined in more detail below.

Lists Are Ordered

A list is not merely a collection of objects. It is an ordered collection of objects. The order in which you specify the elements when you define a list is an innate characteristic of that list and is maintained for that list's lifetime. (You will see a Python data type that is not ordered in the next tutorial on dictionaries.)

Lists that have the same elements in a different order are not the same:

```
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> b = ['baz', 'qux', 'bar', 'foo']
>>> a == b
False
>>> a is b
False


>>> [1, 2, 3, 4] == [4, 1, 3, 2]
False
```

Lists Can Contain Arbitrary Objects

A list can contain any assortment of objects. The elements of a list can all be the same type:

```
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
```

Or the elements can be of varying types:

Python

```
>>> a = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
>>> a
[21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
```

Lists can even contain complex objects, like functions, classes, and modules, which you will learn about in upcoming tutorials:

```
>>> int
<class 'int'>
>>> len
<built-in function len>
>>> def foo():
...     pass
...
>>> foo
<function foo at 0x035B9030>
>>> import math
>>> math
<module 'math' (built-in)>

>>> a = [int, len, foo, math]
>>> a
[<class 'int'>, <built-in function len>, <function foo at 0x02CA2618>,
<module 'math' (built-in)>]
```

A list can contain any number of objects, from zero to as many as your computer's memory will allow:

```
>>> a = []
>>> a
[]

>>> a = [ 'foo' ]
>>> a
['foo']
```

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
... 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
... 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
... 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
... 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77,
78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
97, 98, 99, 100]
```

(A list with a single object is sometimes referred to as a singleton list.)

List objects needn't be unique. A given object can appear in a list multiple times:

```
>>> a = ['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
>>> a
['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
```

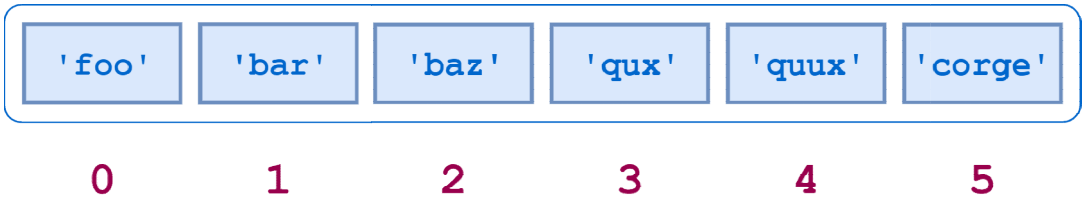
List Elements Can Be Accessed by Index

Individual elements in a list can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based as it is with strings.

Consider the following list:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

The indices for the elements in a are shown below:



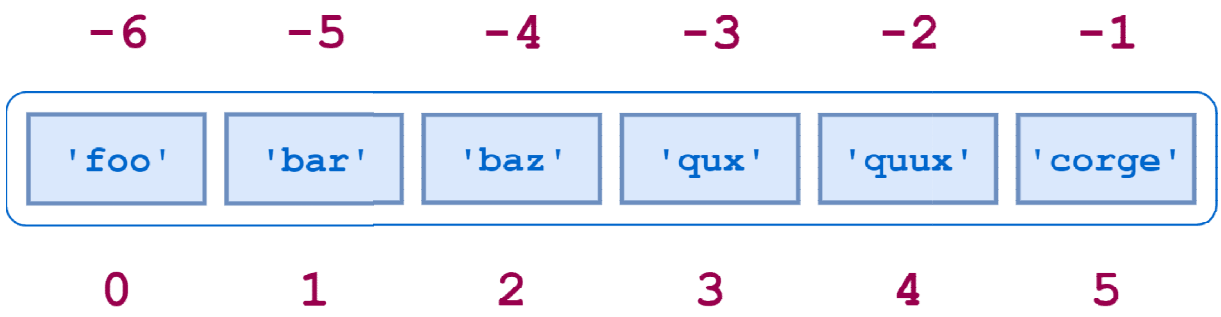
List Indices

Here is Python code to access some elements of a:

```
>>> a[0]
'foo'
>>> a[2]
```

```
'baz'
>>> a[5]
'corge'
```

Virtually everything about string indexing works similarly for lists. For example, a negative list index counts from the end of the list:



Negative List Indexing

```
>>> a[-1]
'corge'
>>> a[-2]
'quux'
>>> a[-5]
'bar'
```

Slicing also works. If `a` is a list, the expression `a[m:n]` returns the portion of `a` from index `m` to, but not including, index `n`:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:5]
['baz', 'qux', 'quux']
```

Other features of string slicing work analogously for list slicing as well:

- Both positive and negative indices can be specified:

  ```
  >>> a[-5:-2]
  ['bar', 'baz', 'qux']
  >>> a[1:4]
  ['bar', 'baz', 'qux']
  >>> a[-5:-2] == a[1:4]
  True
  ```

- Omitting the first index starts the slice at the beginning of the list, and omitting the second index extends the slice to the end of the list:

```
>>> print(a[:4], a[0:4])
['foo', 'bar', 'baz', 'qux'] ['foo', 'bar', 'baz', 'qux']
>>> print(a[2:], a[2:len(a)])
['baz', 'qux', 'quux', 'corge'] ['baz', 'qux', 'quux', 'corge']


>>> a[:4] + a[4:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:4] + a[4:] == a
True
```

- You can specify a stride—either positive or negative:

```
>>> a[0:6:2]
['foo', 'baz', 'quux']
>>> a[1:6:2]
['bar', 'qux', 'corge']
>>> a[6:0:-2]
['corge', 'qux', 'bar']
```

- The syntax for reversing a list works the same way it does for strings:

```
>>> a[::-1]
['corge', 'quux', 'qux', 'baz', 'bar', 'foo']
```

- The `[:]` syntax works for lists. However, there is an important difference between how this operation works with a list and how it works with a string.

  If `s` is a string, `s[:]` returns a reference to the same object:

```
>>> s = 'foobar'
>>> s[:]
'foobar'
>>> s[:] is s
True
```

  Conversely, if `a` is a list, `a[:]` returns a new object that is a copy of `a`:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:] is a
False
```

Several Python operators and built-in functions can also be used with lists in ways that are analogous to strings:

- The `in` and `not in` operators:

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> 'qux' in a
True
>>> 'thud' not in a
True
```

- The concatenation (+) and replication (*) operators:

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a + ['grault', 'garply']
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']
>>> a * 2
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'foo', 'bar', 'baz',
'qux', 'quux', 'corge']
```

- The `len()`, `min()`, and `max()` functions:

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> len(a)
6
>>> min(a)
'bar'
>>> max(a)
'qux'
```

It's not an accident that strings and lists behave so similarly. They are both special cases of a more general object type called an iterable, which you will encounter in more detail in the upcoming tutorial on definite iteration.

By the way, in each example above, the list is always assigned to a variable before an operation is performed on it. But you can operate on a list literal as well:

```
>>> ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][2]
'baz'

>>> ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1]
['corge', 'quux', 'qux', 'baz', 'bar', 'foo']
```

```
>>> 'quux' in ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
True

>>> ['foo', 'bar', 'baz'] + ['qux', 'quux', 'corge']
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> len(['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1])
6
```

For that matter, you can do likewise with a string literal:

```
>>> 'If Comrade Napoleon says it, it must be right.'[::-1]
'.thgir eb tsum ti ,ti syas noelopaN edarmoC fI'
```

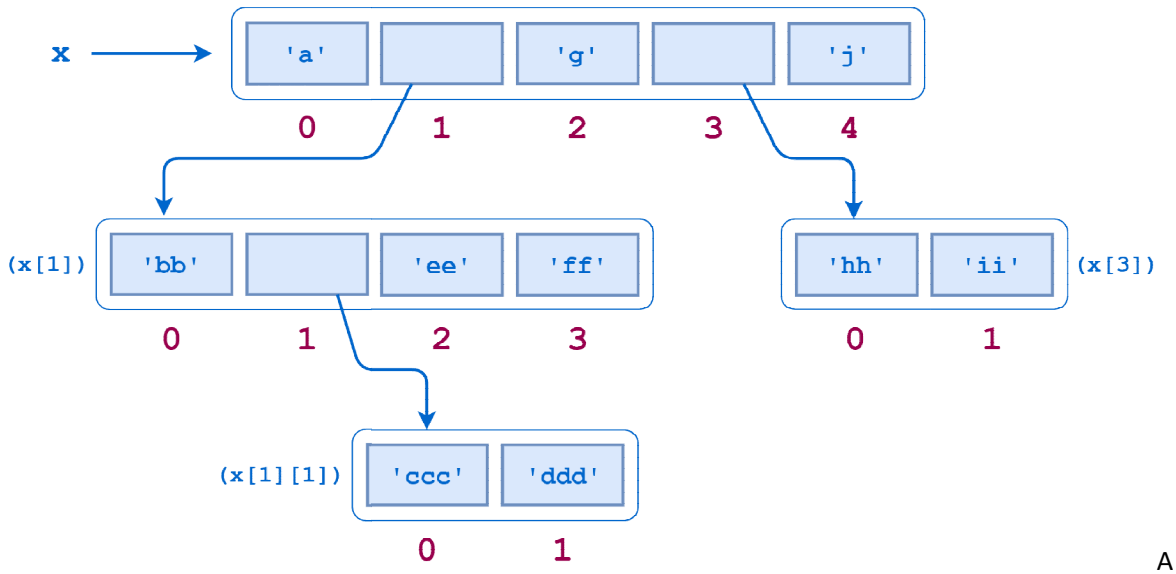
Lists Can Be Nested

You have seen that an element in a list can be any sort of object. That includes another list. A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth.

Consider this (admittedly contrived) example:

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

The object structure that x references is diagrammed below:



A Nested List

x[0], x[2], and x[4] are strings, each one character long:

```
>>> print(x[0], x[2], x[4])
```

```
a g j
```
But `x[1]` and `x[3]` are sublists:

```
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[3]
['hh', 'ii']
```
To access the items in a sublist, simply append an additional index:

```
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[1][0]
'bb'
>>> x[1][1]
['ccc', 'ddd']
>>> x[1][2]
'ee'
>>> x[1][3]
'ff'

>>> x[3]
['hh', 'ii']
>>> print(x[3][0], x[3][1])
hh ii
```
`x[1][1]` is yet another sublist, so adding one more index accesses its elements:

```
>>> x[1][1]
['ccc', 'ddd']
>>> print(x[1][1][0], x[1][1][1])
ccc ddd
```
There is no limit, short of the extent of your computer's memory, to the depth or complexity with which lists can be nested in this way.

All the usual syntax regarding indices and slicing applies to sublists as well:

```
>>> x[1][1][-1]
'ddd'
>>> x[1][1:3]
[['ccc', 'ddd'], 'ee']
```

```
>>> x[3][::-1]
['ii', 'hh']
```
However, be aware that operators and functions apply to only the list at the level you specify and are not recursive. Consider what happens when you query the length of `x` using `len()`:

```
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> len(x)
5


>>> x[0]
'a'
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']
>>> x[2]
'g'
>>> x[3]
['hh', 'ii']
>>> x[4]
'j'
```
`x` has only five elements—three strings and two sublists. The individual elements in the sublists don't count toward `x`'s length.

You'd encounter a similar situation when using the `in` operator:

```
>>> 'ddd' in x
False
>>> 'ddd' in x[1]
False
>>> 'ddd' in x[1][1]
True
```
`'ddd'` is not one of the elements in `x` or `x[1]`. It is only directly an element in the sublist `x[1][1]`. An individual element in a sublist does not count as an element of the parent list(s).

Lists Are Mutable

Most of the data types you have encountered so far have been atomic types. Integer or float objects, for example, are primitive units that can't be further broken down. These types are immutable, meaning that they can't be changed once they have been assigned. It doesn't make much sense to think of changing the value of an integer. If you want a different integer, you just assign a different one.

By contrast, the string type is a composite type. Strings are reducible to smaller parts—the component characters. It might make sense to think of changing the characters in a string. But you can't. In Python, strings are also immutable.

The list is the first mutable data type you have encountered. Once a list has been created, elements can be added, deleted, shifted, and moved around at will. Python provides a wide range of ways to modify lists.

Modifying a Single List Value

A single value in a list can be replaced by indexing and simple assignment:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']


>>> a[2] = 10
>>> a[-1] = 20
>>> a
['foo', 'bar', 10, 'qux', 'quux', 20]
```

You may recall from the tutorial Strings and Character Data in Python that you can't do this with a string:

```
>>> s = 'foobarbaz'
>>> s[2] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

A list item can be deleted with the `del` command:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']


>>> del a[3]
>>> a
['foo', 'bar', 'baz', 'quux', 'corge']
```

Modifying Multiple List Values

What if you want to change several contiguous elements in a list at one time? Python allows this with slice assignment, which has the following syntax:

```
a[m:n] = <iterable>
```

Again, for the moment, think of an iterable as a list. This assignment replaces the specified slice of `a` with `<iterable>`:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']


>>> a[1:4]
['bar', 'baz', 'qux']
>>> a[1:4] = [1.1, 2.2, 3.3, 4.4, 5.5]
>>> a
['foo', 1.1, 2.2, 3.3, 4.4, 5.5, 'quux', 'corge']
>>> a[1:6]
[1.1, 2.2, 3.3, 4.4, 5.5]
>>> a[1:6] = ['Bark!']
>>> a
['foo', 'Bark!', 'quux', 'corge']
```

The number of elements inserted need not be equal to the number replaced. Python just grows or shrinks the list as needed.

You can insert multiple elements in place of a single element—just use a slice that denotes only one element:

```
>>> a = [1, 2, 3]
>>> a[1:2] = [2.1, 2.2, 2.3]
>>> a
[1, 2.1, 2.2, 2.3, 3]
```

Note that this is not the same as replacing the single element with a list:

```
>>> a = [1, 2, 3]
>>> a[1] = [2.1, 2.2, 2.3]
>>> a
[1, [2.1, 2.2, 2.3], 3]
```

You can also insert elements into a list without removing anything. Simply specify a slice of the form `[n:n]` (a zero-length slice) at the desired index:

```
>>> a = [1, 2, 7, 8]
>>> a[2:2] = [3, 4, 5, 6]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
```

You can delete multiple elements out of the middle of a list by assigning the appropriate slice to an empty list. You can also use the `del` statement with the same slice:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[1:5] = []
>>> a
['foo', 'corge']
```

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> del a[1:5]
>>> a
['foo', 'corge']
```

Prepending or Appending Items to a List

Additional items can be added to the start or end of a list using the + concatenation operator or the += augmented assignment operator:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a += ['grault', 'garply']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a = [10, 20] + a
>>> a
[10, 20, 'foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

Note that a list must be concatenated with another list, so if you want to add only one element, you need to specify it as a singleton list:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a += 20
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    a += 20
TypeError: 'int' object is not iterable

>>> a += [20]
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 20]
.

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> a += 'corge'
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'c', 'o', 'r', 'g', 'e']
```

This result is perhaps not quite what you expected. When a string is iterated through, the result is a list of its component characters. In the above example, what gets concatenated onto list `a` is a list of the characters in the string `'corge'`.

If you really want to add just the single string `'corge'` to the end of the list, you need to specify it as a singleton list:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> a += ['corge']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

Methods That Modify a List

Finally, Python supplies several built-in methods that can be used to modify lists. Information on these methods is detailed below.

```
>>> s = 'foobar'
>>> t = s.upper()
>>> print(s, t)
foobar FOOBAR
```

List methods are different. Because lists are mutable, the list methods shown here modify the target list in place.

`a.append(<obj>)`

Appends an object to a list.

`a.append(<obj>)` appends object `<obj>` to the end of list `a`:

```
>>> a = ['a', 'b']
>>> a.append(123)
>>> a
['a', 'b', 123]
```

Remember, list methods modify the target list in place. They do not return a new list:

```
>>> a = ['a', 'b']
>>> x = a.append(123)
>>> print(x)
None
>>> a
['a', 'b', 123]
```

Remember that when the + operator is used to concatenate to a list, if the target operand is an iterable, then its elements are broken out and appended to the list individually:

```
>>> a = ['a', 'b']
>>> a + [1, 2, 3]
['a', 'b', 1, 2, 3]
```

The `.append()` method does not work that way! If an iterable is appended to a list with `.append()`, it is added as a single object:

```
>>> a = ['a', 'b']
>>> a.append([1, 2, 3])
>>> a
['a', 'b', [1, 2, 3]]
```

Thus, with `.append()`, you can append a string as a single entity:

```
>>> a = ['a', 'b']
>>> a.append('foo')
>>> a
['a', 'b', 'foo']
```

a.extend(<iterable>)

Extends a list with the objects from an iterable.

Yes, this is probably what you think it is. `.extend()` also adds to the end of a list, but the argument is expected to be an iterable. The items in `<iterable>` are added individually:

```
>>> a = ['a', 'b']
>>> a.extend([1, 2, 3])
>>> a
['a', 'b', 1, 2, 3]
```

In other words, `.extend()` behaves like the `+` operator. More precisely, since it modifies the list in place, it behaves like the `+=` operator:

```
>>> a = ['a', 'b']
>>> a += [1, 2, 3]
>>> a
['a', 'b', 1, 2, 3]
```

a.insert(<index>, <obj>)

Inserts an object into a list.

`a.insert(<index>, <obj>)` inserts object `<obj>` into list `a` at the specified `<index>`. Following the method call, `a[<index>]` is `<obj>`, and the remaining list elements are pushed to the right:

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.insert(3, 3.14159)
>>> a[3]
3.14159
>>> a
['foo', 'bar', 'baz', 3.14159, 'qux', 'quux', 'corge']
```

a.remove(<obj>)

Removes an object from a list.

`a.remove(<obj>)` removes object `<obj>` from list `a`. If `<obj>` isn't in `a`, an exception is raised:

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.remove('baz')
>>> a
['foo', 'bar', 'qux', 'quux', 'corge']

>>> a.remove('Bark!')
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    a.remove('Bark!')
ValueError: list.remove(x): x not in list
```

a.pop(index=-1)

Removes an element from a list.

This method differs from `.remove()` in two ways:

1. You specify the index of the item to remove, rather than the object itself.
2. The method returns a value: the item that was removed.

`a.pop()` simply removes the last item in the list:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop()
'corge'
>>> a
['foo', 'bar', 'baz', 'qux', 'quux']
```

```
>>> a.pop()
'quux'
>>> a
['foo', 'bar', 'baz', 'qux']
```

If the optional `<index>` parameter is specified, the item at that index is removed and returned. `<index>` may be negative, as with string and list indexing:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(1)
'bar'
>>> a
['foo', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(-3)
'qux'
>>> a
['foo', 'baz', 'quux', 'corge']
```

`<index>` defaults to `-1`, so `a.pop(-1)` is equivalent to `a.pop()`.

Lists Are Dynamic

This tutorial began with a list of six defining characteristics of Python lists. The last one is that lists are dynamic. You have seen many examples of this in the sections above. When items are added to a list, it grows as needed:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[2:2] = [1, 2, 3]
>>> a += [3.14159]
>>> a
['foo', 'bar', 1, 2, 3, 'baz', 'qux', 'quux', 'corge', 3.14159]
```

Similarly, a list shrinks to accommodate the removal of items:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[2:3] = []
>>> del a[0]
>>> a
['bar', 'qux', 'quux', 'corge']
```