TRIBHUVAN UNIVERSITY

INSTITUTE OF SCIENCE AND TECHNOLOGY

Central Department of Computer Science and Information Technology

Kirtipur, Kathmandu

A Case Study Report

on

Python Programming Language

Submitted by:

Name: Brihat Ratna Bajracharya

Roll No.: 19/075

Submitted to:

Mrs. Lalita Sthapit

Central Department of Computer Science and Information Technology

Date of submission: *2077 Chaitra 3*

TABLE OF CONTENTS

Introduction

Python is an interpreted, high-level and general-purpose programming language. Python emphasizes on code readability with its notable use of significant indentation. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library. Python is ranked as one of the most popular programming languages.

History

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to ABC programming language, which was inspired by SETL, capable of exception handling and interfacing with the Amoeba operating system. Its implementation began in December 1989. Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018. In January 2019, active Python core developers elected Brett Cannon, Nick Coghlan, Barry Warsaw, Carol Willing and Van Rossum to a five-member "Steering Council" to lead the project.

The language's core philosophy is given as:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

Python Versions

Python 2.0 was released on 16 October 2000 with many major new features, including a cycle-detecting garbage collector and support for Unicode.

Python 3.0 was released on 3 December 2008. It was a major revision of the language that is not completely backward-compatible. Many of its major features were backported to Python 2.6.x and 2.7.x version series. Releases of Python 3 include the 2to3 utility, which automates (partially) the translation of Python 2 code to Python 3.

Python 2.7's end-of-life date was initially set at 2015 then postponed to 2020 out of concern that a large body of existing code could not easily be forward-ported to Python 3. No more security patches or other improvements will be released for it. With Python 2's end-of-life, only Python 3.6.x and later are supported. Python 3.9.2 is the latest stable release of python released on 19 February 2021..

Applications

Large organizations like Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram, Spotify and some smaller entities like ILM and ITA uses python. The social news networking site Reddit was written mostly in Python. Python also serves as a scripting language for web applications, e.g., via mod_wsgi for the Apache web server. With Web Server Gateway Interface, a standard API has evolved to facilitate these applications. Web frameworks like Django, Pylons, Pyramid, TurboGears, web2py, Tornado, Flask, Bottle and Zope support developers in the design and maintenance of complex applications. Pyjs and IronPython can be used to develop the client-side of Ajax-based applications. SQLAlchemy can be used as a data mapper to a relational database. Twisted is a framework to program communications between computers, and is used (for example) by Dropbox.

Strong libraries such as NumPy, SciPy and Matplotlib allow the effective use of Python in scientific computing, with specialized libraries such as Biopython and Astropy providing domain-specific functionality. SageMath is a mathematical software with a notebook interface programmable in Python: its library covers many aspects of mathematics, including algebra, combinatorics, numerical mathematics, number theory, and calculus. OpenCV has python bindings with a rich set of features for computer vision and image processing.

Python is the de-facto language for artificial intelligence projects and machine learning projects with the help of libraries like TensorFlow, Keras, Pytorch and Scikit-learn. As a scripting language with modular architecture, simple syntax and rich text processing tools, Python is often used for natural language processing as well.

PIP (PIP Installs Packages)

pip is a package-management system written in Python used to install and manage software packages. It connects to an online repository of public and paid-for private packages, called the Python Package Index (pypi).

Most distributions of Python come with pip preinstalled. Python 2.7.9 and later (on the python2 series), and Python 3.4 and later include pip (pip3 for Python 3) by default. Python 2.7 (and 3.5) support was dropped with the next release, pip 21, in January 2021.

Lexical analysis

A Python program is read by a parser. Input to the parser is a stream of tokens, generated by the lexical analyzer. Python reads program text as Unicode code points; the encoding of a source file can be given by an encoding declaration and defaults to UTF-8. If the source file cannot be decoded, a SyntaxError is raised.

Line structure

Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more physical lines by following the explicit or implicit line joining rules.

Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files and strings, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform. The end of input also serves as an implicit terminator for the final physical line.

Comments

A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax.

Encoding declarations

If a comment in the first or second line of the Python script matches the regular expression coding[=:]\s*([-\w.]+), this comment is processed as an encoding declaration; the first group of this

expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line. The recommended forms of an encoding expression are

```
# -*- coding: <encoding-name> -*-
```

which is recognized also by GNU Emacs, and

```
# vim:fileencoding=<encoding-name>
```

which is recognized by Bram Moolenaar's VIM.

The default encoding is UTF-8. If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, including string literals, comments and identifiers.

Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character.

Example:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes.

Example:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard interactive interpreter, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a TabError is raised in that case.

Here is an example of a correctly indented piece of Python code:

```
def perm(l):
    # Compute the list of all permutations of l
```

```
    if len(l) <= 1:
        return [l]
    r = []
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        p = perm(s)
        for x in p:
            r.append(l[i:i+1] + x)
    return r
```

Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).

Identifiers and keywords

The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be spelled exactly (case-sensitive) as written here:

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**

Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

_*

Not imported by from module import *. The special identifier _ is used in the interactive interpreter to store the result of the last evaluation; it is stored in the builtins module. When not in interactive mode, _ has no special meaning and is not defined.

__*__

System-defined names, informally known as “dunder” names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the Special method names section and elsewhere.

__*

Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between “private” attributes of base and derived classes.

Literals

Literals are notations for constant values of some built-in types.

String and Bytes literals

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "R" | "U" | "f" | "F" | "fr" | "Fr" |
                     "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring     ::=   "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''" | '"""' longstringitem*
'"""'
shortstringitem ::=  shortstringchar | stringescapeseq
longstringitem  ::=  longstringchar | stringescapeseq
shortstringchar ::=  <any source character except "\" or newline or the
quote>
longstringchar  ::=  <any source character except "\">
stringescapeseq ::=  "\" <any source character>

bytesliteral   ::=  bytesprefix(shortbytes | longbytes)
bytesprefix    ::=  "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" |
"Rb" | "RB"
shortbytes     ::=  "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
longbytes       ::=   "'''" longbytesitem* "'''" | '"""' longbytesitem*
'"""'
shortbytesitem ::=  shortbyteschar | bytesescapeseq
longbytesitem  ::=  longbyteschar | bytesescapeseq
```

```
shortbyteschar ::=  <any ASCII character except "\" or newline or the quote>
longbyteschar  ::=  <any ASCII character except "\">
bytesescapeseq ::=  "\" <any ASCII character>
```

Escape Sequence

\newline	Backslash and newline ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Integer literals

```
integer      ::=  decinteger | bininteger | octinteger | hexinteger
decinteger   ::=  nonzerodigit (["_"] digit)* | "0"+ (["_"] "0")*
bininteger   ::=  "0" ("b" | "B") (["_"] bindigit)+
octinteger   ::=  "0" ("o" | "O") (["_"] octdigit)+
hexinteger   ::=  "0" ("x" | "X") (["_"] hexdigit)+
nonzerodigit ::=  "1"..."9"
digit        ::=  "0"..."9"
bindigit     ::=  "0" | "1"
octdigit     ::=  "0"..."7"
hexdigit     ::=  digit | "a"..."f" | "A"..."F"
```

Operators

```
+       -       *       **      /       //      %      @
<<      >>      &       |       ^       ~       :=
<       >       <=      >=      ==      !=
```

Delimiters

```
(       )       [       ]       {       }
,       :       .       ;       @       =       ->
+=      -=      *=      /=      //=     %=      @=
&=      |=      ^=      >>=     <<=     **=
```

Data Types

Defining variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable

Python allows to assign a single value to several variables simultaneously. For example −

```
a = b = c = 1
```

Also assign multiple objects to multiple variables like

```
a,b,c = 1,2,"john"
```

Standard Data Types

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. Example

```
var1 = 1
var2 = 10
```

Python supports four different numerical types: **int** (signed integers), **long** (long integers, they can also be represented in octal and hexadecimal), **float** (floating point real values), **complex** (complex numbers)

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 and -1 at the end. The plus (+) sign is used for string concatenation the asterisk (*) is used for repetition.

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

Data Type Conversion

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.

```
int(x [,base])         Converts x to an integer. In base base.
long(x [,base] )       Converts x to a long integer with base.
float(x)               Converts x to a floating-point number.
complex(real [,imag]) Creates a complex number.
str(x)                 Converts object x to a string representation.
repr(x)                Converts object x to an expression string.
eval(str)              Evaluates a string and returns an object.
tuple(s)               Converts s to a tuple.
list(s)                Converts s to a list.
```

`set(s)`	`Converts s to a set.`
`dict(d)`	`Creates a dictionary. d must be (key,value) tuples.`
`frozenset(s)`	`Converts s to a frozen set.`
`chr(x)`	`Converts an integer to a character.`
`unichr(x)`	`Converts an integer to a Unicode character.`
`ord(x)`	`Converts a single character to its integer value.`
`hex(x)`	`Converts an integer to a hexadecimal string.`
`oct(x)`	`Converts an integer to an octal string.`

Operators

Python language supports the following types of operators.

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

Python Arithmetic Operators

Assume a = 10 and variable b = 20, then

Operator	**Description**
+ Addition	Adds values on either side of the operator. a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand. a – b = -10
* Multiplication	Multiplies values on either side of the operator a * b = 200
/ Division	Divides left hand operand by right hand operand b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder b % a = 0
** Exponent	Performs exponential (power) calculation on operators a**b =10 to the power 20

// Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

They are also called Relational operators.

Assume variable a = 10 and variable b = 20, then

Operator	Description
==	If the values of two operands are equal, then the condition becomes true. (a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true. (a != b) is true.
<>	If values of two operands are not equal, then condition becomes true. (a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true. (a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true. (a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. (a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true. (a <= b) is true.

Python Assignment Operators

Assume variable a = 10 and variable b = 20, then

Operator	Description
=	Assigns values from right side operands to left side operand c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand c -= a is equivalent to c = c - a

*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand, c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands −

```
a   = 0011 1100
b   = 0000 1101
---------------
a&b = 0000 1100  (AND)
a|b = 0011 1101  (OR)
a^b = 0011 0001  (XOR)
~a  = 1100 0011  (NOT)
```

Operator	Description
& Binary AND	Operator copies a bit to the result if it exists in both operands
| Binary OR	It copies a bit if it exists in either operand.
^ Binary XOR	It copies the bit if it is set in one operand but not both.
~ Complement	It is unary and has the effect of 'flipping' bits.
<< Left Shift	Value is moved left by the number of bits specified by the right operand.
>> Right Shift	Value is moved right by the number of bits specified by the right operand

Python Logical Operators

Assume variable a = 10 and variable b = 20, then

Operator	Description
and	If both the operands are true then condition becomes true. (a and b) is true.
or	If any of the two operands are non-zero then condition becomes true. (a or b) is true.
not	Used to reverse the logical state of its operand. not(a and b) is false.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description
in	Evaluates to true if it finds a variable in the specified sequence else false
not in	Evaluates to true if it does not finds a variable in the specified sequence else false

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators.

Operator	Description
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object else true. x is not y, here is not results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

Highest to lowest operator precedence is given below

```
**                          (Exponentiation)
~                           (Complement)
*, /, %, //                 (Multiply, divide, modulo and floor division)
+, -                        (Addition and subtraction)
>>, <<                      (Right and left bitwise shift)
&                           (Bitwise 'AND')
^, |                        (Bitwise exclusive `OR' and regular `OR')
<=, <, >, >=                (Comparison operators)
<>, ==, !=                  (Equality operators)
=, %=, /=, //=, -=, +=, *=, **=  (Assignment operators)
is, is not                  (Identity operators)
in, not in                  (Membership operators)
not or and                  (Logical operators)
```

Python Loop Types

Python programming language provides following types of loops to handle looping requirements.

while loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

for loop

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops

Loops can have nested loop inside any another ‘while’, ‘for’ or ‘do..while’ loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. C++ supports the following control statements.

break statement

Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.

continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

pass statement

The pass statement in Python is used when a statement is required syntactically but do not want any command or code to execute.

Statement Level Control Structures in Python

Selection Statements

Two-Way Selection Statements (If-clause)

Syntax

```
if expression:
    statement(s)
else:
    statement(s)
```

Also support nested if clause

Single If-Clause

If the suite of an if clause consists only of a single line, it may go on the same line as the header statement.

Syntax:

```
if expression: statement(s)
```

Multiple Selection Statements

Multiple Selection can also be implemented using if-clause as follows:

Syntax:

```
if control_expression_1:
    statement_1
elif control_expression_2:
    statement_2
. . .
elif control_expression_n:
    statement_n
else:
    statement
```

Iterative Statements

For-loop, and while-loop are the iterative statements in Python

for-loop

Python for-loop has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var in sequence:
    statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating_var. Next, the statements block is executed. Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

An alternative way of iterating through each item is by index offset into the sequence itself.

Syntax:

```
for index in range(offset_value):
    loop body statements
```

Python supports to have an else statement associated with a loop statement. The else statement is executed when the loop has exhausted iterating the list.

while loop

Syntax:

```
while expression:
        statement(s)
```

Nested-loop

Python programming language allows to use one loop inside another loop.

Nested for-loop syntax

```
for iterating_var in sequence:
    for iterating_var in sequence:
        statements(s)
    statements(s)
```

Nested while loop syntax

```
while expression:
    while expression:
        statement(s)
    statement(s)
```

Sequence Subprogram control (Functions in Python)

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. User defined funtions have simple rules to define in Python.

Function blocks begin with the keyword def followed by the function name and parentheses (( )).

Any input parameters or arguments should be placed within these parentheses.

The first statement of a function can be an optional statement - docstring.

The code block within every function starts with a colon (:) and is indented.

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   “”” function_docstring “””
   function_suite
   return [expression]
```

Return Statement − The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, values are passed as parameters. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does.

Function Arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

Keyword arguments

Keyword arguments are related to the function calls. When keyword arguments are used in a function call, the caller identifies the arguments by the parameter name. This allows to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Parameter Passing in Python

Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing". If passed arguments are immutable arguments like integers, strings or tuples to a function, the passing acts like ***call-by-value***. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place within the function. If we pass a list to a function, we have to consider two cases: Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.

Anonymous Functions

Anonymous Functions are not declared in the standard manner by using the def keyword. The lambda keyword to create small anonymous functions.

Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions. An anonymous function cannot be a direct call to print because lambda requires an expression. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the

global namespace. Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Python Modules

A module allows to logically organize the Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference. Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

import Statement

Any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax −

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

from . . . import Statement

Python's from statement imports specific attributes from a module into the current namespace. The from...import has the following syntax

```
from modname import name1[, name2[, ... nameN]]
```

from . . . import * Statement

It is also possible to import all names from a module into the current namespace by using the following import statement

```
from modname import *
```

Locating Modules

When importing a module, the Python interpreter searches for the module in the following sequences

1. The current directory.
2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
3. If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

In windows,

```
set PYTHONPATH = c:\python20\lib;
```

And for UNIX system

```
set PYTHONPATH = /usr/local/lib/python
```

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on. These packages can be imported using the method defined above in this section.

Python Files I/O

Printing to the Screen

The simplest way to produce output is using the print statement passing zero or more expressions separated by commas. This function converts the expressions into a string and writes the result to standard output.

Example:

```
print("Hello, World!!!")
```

Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are **raw_input** and **input**

raw_input Function

The raw_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline).

```
#!/usr/bin/python2
str = raw_input("Enter your input: ")
print "Received input is : ", str
```

input Function

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
#!/usr/bin/python
str = input("Enter your input: ")
print "Received input is : ", str
```

Opening and Closing Files

Python provides basic functions and methods necessary to manipulate files by default. Most of the file manipulation is done using a file object.

open Function

This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

file_name − string value that contains the name of the file

access_mode − determines the mode in which the file has to be opened, i.e., read, write, append, etc. The default file access mode is read (r).

buffering − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file.

The different modes of opening a file are **r** (readonly), **rb** (readonly in binary), **r+** (read and write), **rb+** (readwrite in binary), **w** (write only, overwrites existing file), **wb** (write in binary), **w+** (write and read), **wb+** (write and read binary), **a** (append), **ab** (append in binary), **a+** (append and read), **ab+** (append read in binary)

file Object Attributes

```
file.closed     Returns true if file is closed, false otherwise.
file.mode       Returns access mode with which file was opened.
file.name       Returns name of the file.
file.softspace  Returns false if space explicitly required with print,
                true otherwise.
```

close() Method

The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done. Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close()
```

Reading and Writing Files

The file object provides a set of access methods to make our lives easier. We would see how to use read() and write() methods to read and write files.

write() Method

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text. The write() method does not add a newline character ('\n') to the end of the string

Syntax

```
fileObject.write(string)
```

Here, passed parameter is the content to be written into the opened file.

read() Method

The read() method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

```
fileObject.read([count])
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

File Positions

The **tell() method** tells the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The **seek(offset[, from])** method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved. If from is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Renaming and Deleting Files

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

rename() Method

The rename() method takes two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_file_name, new_file_name)
```

remove() Method

The remove() method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

```
os.remove(file_name)
```

Object Oriented in Python

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy.

Class

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows −

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

The class has a documentation string, which can be accessed via ClassName.__doc__. The class_suite consists of all the component statements defining class members, data attributes and functions.

Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts. Attributes of the object can be accessed using the dot operator with object. Class variable would be accessed using class name. However, instead of using the normal statements to access attributes, following functions can be used.

```
getattr(obj, name[, default]) - to access the attribute of object.
hasattr(obj,name) - to check if an attribute exists or not.
setattr(obj,name,value) - to set an attribute. Created if not exist.
delattr(obj, name) - to delete an attribute.
```

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

```
__dict__    -    Dictionary containing the class's namespace.
__doc__     -    Class documentation string or none, if undefined.
__name__    -    Class name.
__module__  -    Module name in which the class is defined. This
                 attribute is "__main__" in interactive mode.
```

__bases__ - **A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.**

Other object oriented features (abstraction, encapsulation, inheritance, and polymorphism) are similar to generic object oriented programming

Exception Handling in Python

Python provides two very important features to handle any unexpected error in Python programs and to add debugging capabilities in them

Python Standard Exceptions

Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution (example: Ctrl+c press)
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.

IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Python Assertions

An assertion is a sanity-check that can be turn on or turn off when done with testing of the program. The easiest way to think of an assertion is to liken it to a raise-if statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised. Assertions are carried out by the assert statement.

assert Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception.

Syntax:

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses **ArgumentExpression** as the argument for the **AssertionError**. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

try-except-finally statement block

The try block tests a block of code for errors. The except block handles the error. And, the finally block executes code, regardless of the result of the try- and except blocks.

Syntax:

```
try:
  block of code that may contain errors
except:
  block of code to handle raised exception
finally:
  block of code after try-except
```

Bibliography

[1] "2. Lexical analysis," *2. Lexical analysis - Python 3.9.2 documentation*. [Online]. Available: https://docs.python.org/3/reference/lexical_analysis.html. [Accessed: 10-Mar-2021].

[2] "Assertions in Python," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/assertions_in_python.htm. [Accessed: 15-Mar-2021].

[3] "Python (programming language)," *Wikipedia*, 9-Mar-2021. [Online]. Available: https://en.wikipedia.org/wiki/Python_(programming_language). [Accessed: 9-Mar-2021].

[4] "Python - Exceptions Handling," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_exceptions.htm. [Accessed: 15-Mar-2021].

[5] "Python - Files I/O," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_files_io.htm. [Accessed: 13-Mar-2021].

[6] "Python - Functions," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_functions.htm. [Accessed: 12-Mar-2021].

[7] "Python - Loops," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_loops.htm. [Accessed: 12-Mar-2021].

[8] "Python - Modules," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_modules.htm. [Accessed: 14-Mar-2021].

[9] "Python - Object Oriented," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_classes_objects.htm. [Accessed: 12-Mar-2021].

[10] "Python - Variable Types," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_variable_types.htm. [Accessed: 10-Mar-2021].

[11] "Python IF Statement," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_if_statement.htm. [Accessed: 11-Mar-2021].

[12] "Python Standard Exceptions," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/standard_exceptions.htm. [Accessed: 15-Mar-2021].

[13] "Python Tutorial: Passing Arguments," *Python Course*. [Online]. Available: https://www.python-course.eu/python3_passing_arguments.php. [Accessed: 14-Mar-2021].

[14] "Python while Loop Statements," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/python/python_while_loop.htm. [Accessed: 12-Mar-2021].