2

Parts of Python Programming Language

AIM

Understand various Operators, Expression, Data Types, User input and print statements upon which multifaceted operations can be built in Python Programming Language.

LEARNING OUTCOMES

After completing this chapter, you should be able to

- Define identifiers, keywords, operators and expressions.
- Use different operators, expressions and variables available in Python.
- Build complex expressions using operators.
- Determine the data types of values.
- Use indentation and comments in writing Python programs.

In this chapter, we discuss the fundamentals of Python Programming language which you need to know before writing simple Python programs. This chapter describes how Python programs should work at the most basic level and gives details about operators, types and keywords upon which complex solutions can be built. Once you gain familiarity with these foundational elements of Python programming language, then you will appreciate how a lot of functionality can be accomplished with less verbose code.

2.1 Identifiers

An identifier is a name given to a variable, function, class or module. Identifiers may be one or more characters in the following format:

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myCountry, other_1 and good_morning, all are valid examples. A Python identifier can begin with an alphabet (A – Z and a – z and _).
- An identifier cannot start with a digit but is allowed everywhere else. 1plus is invalid, but plus1 is perfectly fine.

- Keywords cannot be used as identifiers.
- One cannot use spaces and special symbols like !, @, #, $, % etc. as identifiers.
- Identifier can be of any length.

2.2 Keywords

Keywords are a list of reserved words that have predefined meaning. Keywords are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name. Attempting to use a keyword as an identifier name will cause an error. The following TABLE 2.1 shows the Python keywords.

TABLE 2.1
List of Keywords in Python

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

2.3 Statements and Expressions

A statement is an instruction that the Python interpreter can execute. Python program consists of a sequence of statements. Statements are everything that can make up a line (or several lines) of Python code. For example, z = 1 is an assignment statement.

Expression is an arrangement of values and operators which are evaluated to make a new value. Expressions are statements as well. A value is the representation of some entity like a letter or a number that can be manipulated by a program. A single value >>> 20 or a single variable >>> z or a combination of variable, operator and value >>> z + 20 are all examples of expressions. An expression, when used in interactive mode is evaluated by the interpreter and result is displayed instantly. For example,

```
>>> 8 + 2
10
```

But the same expression when used in Python program does not show any output altogether. You need to explicitly print the result.

2.4 Variables

Variable is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution. In Python, there is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type. *To define a new variable in Python, we simply assign a value to a name.* If a need for variable arises you need to think of a variable name based on the rules mentioned in the following subsection and use it in the program.

2.4.1 Legal Variable Names

Follow the below-mentioned rules for creating legal variable names in Python.

- Variable names can consist of any number of letters, underscores and digits.
- Variable should not start with a number.
- Python Keywords are not allowed as variable names.
- Variable names are case-sensitive. For example, computer and Computer are different variables.

Also, follow these guidelines while naming a variable, as having a consistent naming convention helps in avoiding confusion and can reduce programming errors.

- Python variables use lowercase letters with words separated by underscores as necessary to improve readability, like this whats_up, how_are_you. Although this is not strictly enforced, it is considered a best practice to adhere to this convention.
- Avoid naming a variable where the first character is an underscore. While this is legal in Python, it can limit the interoperability of your code with applications built by using other programming languages.
- Ensure variable names are descriptive and clear enough. This allows other programmers to have an idea about what the variable is representing.

2.4.2 Assigning Values to Variables

The general format for assigning values to variables is as follows:

variable_name = expression

The equal sign (=) also known as simple assignment operator is used to assign values to variables. In the general format, 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 expression which can be a value or any code snippet that results in a value. That value is stored in the variable on the execution of the assignment statement. Assignment operator should not be confused with the = used in algebra to denote equality. For example, enter the code shown below in interactive mode and observe the results.

```
>>> number =100
>>> miles =1000.0
```

```
3. >>> name ="Python"
4. >>> number
   100
5. >>> miles
   1000.0
6. >>> name
   'Python'
```

In ① integer type value is assigned to a variable *number*, in ② float type value has been assigned to variable *miles* and in ③ string type value is assigned to variable *name*. ④, ⑤ and ⑥ prints the value assigned to these variables.

In Python, not only the value of a variable may change during program execution but also the type of data that is assigned. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable. A new assignment overrides any previous assignments. For example,

```
1. >>> century = 100
2. >>> century
   100
3. >>> century = "hundred"
4. >>> century
   'hundred'
```

In ① an integer value is assigned to *century* variable and then in ③ you are assigning a string value to *century* variable. Different values are printed in each case as seen in ② and ④.

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

```
1. >>> a = b = c =1
2. >>> a
   1
3. >>> b
   1
4. >>> c
   1
```

An integer value is assigned to variables a, b and c simultaneously ①. Values for each of these variables are displayed as shown in ②, ③ and ④.

2.5 Operators

Operators are symbols, such as +, −, =, >, and <, that perform certain mathematical or logical operation to manipulate data values and produce a result based on some rules. An operator manipulates the data values called operands.

Consider the expression,

```
>>> 4 + 6
```

where 4 and 6 are operands and + is the operator.

Python language supports a wide range of operators. They are

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators

2.5.1 Arithmetic Operators

Arithmetic operators are used to execute arithmetic operations such as addition, subtraction, division, multiplication etc. The following TABLE 2.2 shows all the arithmetic operators.

TABLE 2.2

List of Arithmetic Operators

Operator	Operator Name	Description	Example
+	Addition operator	Adds two operands, producing their sum.	p + q = 5
−	Subtraction operator	Subtracts the two operands, producing their difference.	p − q = −1
*	Multiplication operator	Produces the product of the operands.	p * q = 6
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	q / p = 1.5
%	Modulus operator	Divides left hand operand by right hand operand and returns a remainder.	q % p = 1
**	Exponent operator	Performs exponential (power) calculation on operators.	p**q = 8
//	Floor division operator	Returns the integral part of the quotient.	9//2 = 4 and 9.0//2.0 = 4.0

Note: The value of p is 2 and q is 3.

For example,

1.
   ```
   >>> 10+35
   45
   ```

2.
   ```
   >>> −10+35
   25
   ```

3.
   ```
   >>> 4*2
   8
   ```

4.
   ```
   >>> 4**2
   16
   ```

```
5. >>> 45/10
   4.5
6. >>> 45//10.0
   4.0
7. >>> 2025%10
   5
8. >>> 2025//10
   202
```

Above code illustrates various arithmetic operations ①–⑧.

2.5.2 Assignment Operators

Assignment operators are used for assigning the values generated after evaluating the right operand to the left operand. Assignment operation always works from right to left. Assignment operators are either simple assignment operator or compound assignment operators. Simple assignment is done with the equal sign (=) and simply assigns the value of its right operand to the variable on the left. For example,

```
1. >>> x = 5
2. >>> x = x + 1
3. >>> x
   6
```

In ① you assign an integer value of 5 to variable *x*. In ② an integer value of 1 is added to the variable *x* on the right side and the value 6 after the evaluation is assigned to the variable *x*. The latest value stored in variable *x* is displayed in ③.

Compound assignment operators support shorthand notation for avoiding the repetition of the left-side variable on the right side. Compound assignment operators combine assignment operator with another operator with = being placed at the end of the original operator.

For example, the statement

```
>>> x = x + 1
```

can be written in a compactly form as shown below.

```
>>> x += 1
```

If you try to update a variable which doesn't contain any value, you get an error.

```
1. >>> z = z + 1
   NameError: name 'z' is not defined
```

Trying to update variable *z* which doesn't contain any value results in an error because Python evaluates the right side before it assigns a value to *z* ①.

```
1. >>> z = 0
2. >>> x = z + 1
```

Before you can update a variable ②, you have to assign a value to it ①.

The following TABLE 2.3 shows all the assignment operators.

TABLE 2.3

List of Assignment Operators

Operator	Operator Name	Description	Example
=	Assignment	Assigns values from right side operands to left side operand.	z = p + q assigns value of p + q to z
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand.	z += p is equivalent to z = z + p
−=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand.	z −= p is equivalent to z = z − p
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand.	z *= p is equivalent to z = z * p
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand.	z /= p is equivalent to z = z / p
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand.	z**= p is equivalent to z = z ** p
//=	Floor Division Assignment	Produces the integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	z //= p is equivalent to z = z // p
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	z %= p is equivalent to z = z % p

For example,

```
1. >>> p = 10
2. >>> q = 12
3. >>> q += p
4. >>> q
   22
5. >>> q *= p
6. >>> q
   220
7. >>> q /= p
8. >>> q
   22.0
9. >>> q %= p
10. >>> q
   2.0
11. >>> q **= p
12. >>> q
   1024.0
13. >>> q //= p
14. >>> q
   102.0
```

Above code illustrates various assignment operations ①–⑭.

Learned readers coming from other languages should note that Python programming language doesn't support Autoincrement (++) and Autodecrement (--) operators.

2.5.3 Comparison Operators

When the values of two operands are to be compared then comparison operators are used. The output of these comparison operators is always a Boolean value, either True or False. The operands can be Numbers or Strings or Boolean values. Strings are compared letter by letter using their ASCII values, thus, "P" is less than "Q", and "Aston" is greater than "Asher". TABLE 2.4 shows all the comparison operators.

TABLE 2.4

List of Comparison Operators

Operator	Operator Name	Description	Example
==	Equal to	If the values of two operands are equal, then the condition becomes True.	(p == q) is not True.
!=	Not Equal to	If values of two operands are not equal, then the condition becomes True.	(p != q) is True
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes True.	(p > q) is not True.
<	Lesser than	If the value of left operand is less than the value of right operand, then condition becomes True.	(p < q) is True.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes True.	(p >= q) is not True.
<=	Lesser than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes True.	(p <= q) is True.

Note: The value of p is 10 and q is 20.

For example,

1. >>>10 == 12
   False
2. >>>10 != 12
   True
3. >>>10 < 12
   True
4. >>>10 > 12
   False
5. >>>10 <= 12
   True

6. >>>10 >= 12
   False
7. >>> "P" < "Q"
   True
8. >>> "Aston" > "Asher"
   True
9. >>> True == True
   True

Above code illustrates various comparison operations ①–⑨.

Don’t confuse the equality operator == with the assignment operator =. The expression x==y compares x with y and has the value True if the values are the same. The expression x=y assigns the value of y to x.

2.5.4 Logical Operators

The logical operators are used for comparing or negating the logical values of their operands and to return the resulting logical value. The values of the operands on which the logical operators operate evaluate to either True or False. The result of the logical operator is always a Boolean value, True or False. TABLE 2.5 shows all the logical operators.

TABLE 2.5

List of Logical Operators

Operator	Operator Name	Description	Example
and	Logical AND	Performs AND operation and the result is True when both operands are True	p and q results in False
or	Logical OR	Performs OR operation and the result is True when any one of both operand is True	p or q results in True
not	Logical NOT	Reverses the operand state	not p results in False

Note: The Boolean value of p is True and q is False.

The Boolean logic Truth table is shown in TABLE 2.6.

TABLE 2.6

Boolean Logic Truth Table

P	Q	P and Q	P or Q	Not P
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

For example,

```
1. >>> True and False
   False
2. >>> True or False
   True
3. >>> not(True) and False
   False
4. >>> not(True and False)
   True
5. >>> (10 < 0) and (10 > 2)
   False
6. >>> (10 < 0) or (10 > 2)
   True
7. >>> not(10 < 0) or (10 > 2)
   True
8. >>> not(10 < 0 or 10 > 2)
   False
```

Above code illustrates various comparison operations ①–⑧.

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" valuation using the following rules:

1. False and (some_expression*)* is short-circuit evaluated to False.
2. True or (some_expression*)* is short-circuit evaluated to True.

The rules of logic guarantee that these evaluations are always correct. The *some_expression* part of the above expressions is not evaluated, so any side effects of doing so do not take effect. For example,

```
1. >>> 1 > 2 and 9 > 6
   False
2. >>> 3 > 2 or 8 < 4
   True
```

In ① the expression 1 > 2 is evaluated to *False*. Since *and* operator is used in the statement the expression is evaluated to *False* and the remaining expression 9 > 6 is not evaluated. In ② the expression 3 > 2 is evaluated to *True*. As *or* operator is used in the statement the expression is evaluated to *True* while the remaining expression 8 < 4 is ignored.

2.5.5 Bitwise Operators

Bitwise operators treat their operands as a sequence of bits (zeroes and ones) and perform bit by bit operation. For example, the decimal number ten has a binary representation of 1010. Bitwise operators perform their operations on such binary representations, but they return standard Python numerical values. TABLE 2.7 shows all the bitwise operators.

TABLE 2.7
List of Bitwise Operators

Operator	Operator Name	Description	Example
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands are 1s.	p & q = 12 (means 0000 1100)
|	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.	p | q = 61 (means 0011 1101)
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.	(p ^ q) = 49 (means 0011 0001)
~	Binary Ones Complement	Inverts the bits of its operand.	(~p) = –61 (means 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	p << 2 = 240 (means 1111 0000)
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	p >> 2 = 15 (means 0000 1111)

Note: The value of p is 60 and q is 13.

The Bitwise Truth table is shown in TABLE 2.8.

TABLE 2.8
Bitwise Truth Table

P	Q	P & Q	P | Q	P ^ Q	~ P
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

A sequence consisting of ones and zeroes is known as *binary*. The smallest amount of information that you can store in a computer is known as a *bit*. A *bit* is represented as either 0 or 1.

FIGURE 2.1 shows examples for bitwise logical operations. The value of operand a is 60 and value of operand b is 13.

Bitwise and (&)					**Bitwise or (|)**				
a	=	0011 1100	→	(60)	a	=	0011 1100	→	(60)
b	=	0000 1101	→	(13)	b	=	0000 1101	→	(13)
a & b	=	0000 1100	→	(12)	a | b	=	0011 1101	→	(61)
Bitwise exclusive or (^)					**One's Complement (~)**				
a	=	0011 1100	→	(60)	a	=	0011 1100	→	(60)
b	=	0000 1101	→	(13)	~ a	=	1100 0011	→	(-61)
a ^ b	=	0011 0001	→	(49)					
Binary left shift (<<)					**Binary right shift (>>)**				
a	=	0011 1100	→	(60)	a	=	0011 1100	→	(60)
a << 2	=	1111 0000	→	(240)	a >> 2	=	0000 1111	→	(15)
left shift of 2 bits					right shift of 2 bits				

FIGURE 2.1
Examples of bitwise logical operators.

FIGURE 2.2 shows how the expression 60 << 2 would be evaluated in a byte.

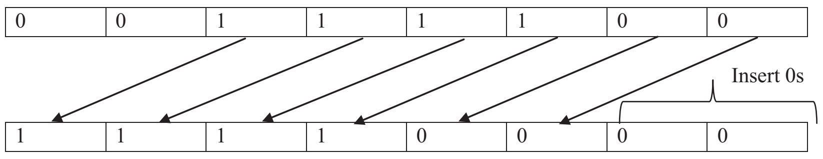

FIGURE 2.2
Example of bitwise left shift of two bits.

Due to this operation,

- Each of the bits in the operand (60) is shifted two places to the left.
- The two bit positions emptied on the right end are filled with 0s.
- The resulting value is 240.

For example,

```
1. >>> p =60
2. >>> p << 2
   240
3. >>> p = 60
4. >>> p >> 2
   15
5. >>> q = 13
6. >>> p & q
   12
7. >>> p | q
   61
```

```
8. >>> ~p
   -61
9. >>> p << 2
   240
10. >>> p >> 2
   15
```

Above code illustrates various Bitwise operations ①–⑩.

2.6 Precedence and Associativity

Operator precedence determines the way in which operators are parsed with respect to each other. Operators with higher precedence become the operands of operators with lower precedence. Associativity determines the way in which operators of the same precedence are parsed. Almost all the operators have left-to-right associativity. Operator precedence is listed in TABLE 2.9 starting with the highest precedence to lowest precedence.

TABLE 2.9

Operator Precedence in Python

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

Consider the following code,

```
1. >>> 2 + 3 * 6
   20
2. >>> (2 + 3) * 6
   30
3. >>> 6 * 4 / 2
   12.0
```

Expressions with higher-precedence operators are evaluated first. In ① multiplication * is having precedence over addition. So, 3 * 6 gets evaluated first and the result is added to 2. This behavior can be overridden using parentheses as shown in ②. Parentheses have the highest precedence and the expression inside the parentheses gets evaluated first, which in our case is 2 + 3 and the result is multiplied with 6. In ③ both multiplication and division have the same precedence hence starting from left to right, the multiplication operator is evaluated first and the result is divided by 2.

2.7 Data Types

Data types specify the type of data like numbers and characters to be stored and manipulated within a program. Basic data types of Python are

- Numbers
- Boolean
- Strings
- None

2.7.1 Numbers

Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as int, float and complex class in Python. Integers can be of any length; it is only limited by the memory available. A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is an integer, 1.0 is floating point number. Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

2.7.2 Boolean

Booleans may not seem very useful at first, but they are essential when you start using conditional statements. In fact, since a condition is really just a yes-or-no question, the answer to that question is a Boolean value, either True or False. The Boolean values, True and False are treated as reserved words.

2.7.3 Strings

A string consists of a sequence of one or more characters, which can include letters, numbers, and other types of characters. A string can also contain spaces. You can use single quotes or double quotes to represent strings and it is also called a string literal. Multiline strings can be denoted using triple quotes, ''' or """. These are fixed values, not variables that you literally provide in your script. For example,

1. >>> s = 'This is single quote string'
2. >>> s = "This is double quote string"

```
3. >>> s = '''This is
                    Multiline
                              string'''
4. >>> s = "a"
```

In ① a string is defined using single quotes, in ② a string is defined using double quotes and a multiline string is defined in ③, a single character is also treated as string ④.

You use literals to represent values in Python. These are fixed values, not variables that you literally provide in your program. A string literal is zero or more characters enclosed in double (") or single (') quotation marks, an integer literal is 12 and float literal is 3.14.

2.7.4 None

None is another special data type in Python. *None* is frequently used to represent the absence of a value. For example,

```
1. >>> money = None
```

None value is assigned to variable *money* ①.

2.8 Indentation

In Python, Programs get structured through indentation (FIGURE 2.3). Usually, we expect indentation from any program code, but in Python it is a requirement and not a matter of style. This principle makes the code look cleaner and easier to understand and read. Any statements written under another statement with the same indentation is interpreted to

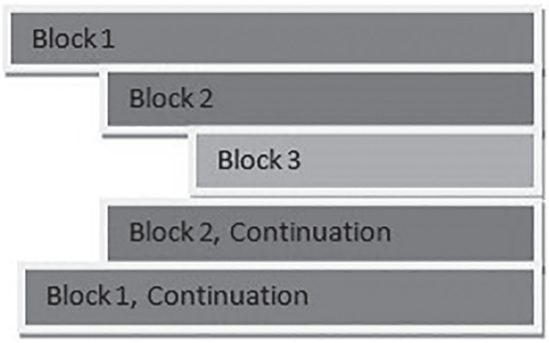

FIGURE 2.3
Code blocks and indentation in Python.

belong to the same code block. If there is a next statement with less indentation to the left, then it just means the end of the previous code block.

In other words, if a code block has to be deeply nested, then the nested statements need to be indented further to the right. In the above diagram, *Block 2* and *Block 3* are nested under *Block 1*. Usually, four whitespaces are used for indentation and are preferred over tabs. Incorrect indentation will result in *IndentationError.*

2.9 Comments

Comments are an important part of any program. A comment is a text that describes what the program or a particular part of the program is trying to do and is ignored by the Python interpreter. Comments are used to help you and other programmers understand, maintain, and debug the program. Python uses two types of comments: single-line comment and multiline comments.

2.9.1 Single Line Comment

In Python, use the hash (#) symbol to start writing a comment. Hash (#) symbol makes all text following it on the same line into a comment. For example,

```
#This is single line Python comment
```

2.9.2 Multiline Comments

If the comment extends multiple lines, then one way of commenting those lines is to use hash (#) symbol at the beginning of each line. For example,

```
#This is
#multiline comments
#in Python
```

Another way of doing this is to use triple quotes, either ''' or """. These triple quotes are generally used for multiline strings. However, they can be used as a multiline comment as well. For example,

```
'''This is
 multiline comment
 in Python using triple quotes'''
```

2.10 Reading Input

In Python, *input()* function is used to gather data from the user. The syntax for input function is,

variable_name = input([prompt])

prompt is a string written inside the parenthesis that is printed on the screen. The *prompt* statement gives an indication to the user of the value that needs to be entered through the keyboard. *When the user presses Enter key, the program resumes and input returns what the user typed as a string*. Even when the user inputs a number, it is treated as a string which should be casted or converted to number explicitly using appropriate type casting function.

```
1. >>> person = input("What is your name?")
2. What is your name? Carrey
3. >>> person
   'Carrey'
```

① the *input()* function prints the prompt statement on the screen (in this case "What is your name?") indicating the user that keyboard input is expected at that point and then it waits for a line to be typed in. User types in his response in ②. The *input()* function reads the line from the user and converts the line into a string. As can be seen in ③, the line typed by the user is assigned to the person variable.

A function is a piece of code that is called by name. It can be passed data to operate on (i.e., the arguments) and can optionally return data (the return value). You shall learn more about functions in Chapter 4.

2.11 Print Output

The *print()* function allows a program to display text onto the console. The print function will print everything as strings and anything that is not already a string is automatically converted to its string representation. For example,

```
1. >>> print("Hello World!!")
   Hello World!!
```

① prints the string Hello World!! onto the console. Notice that the string Hello World is enclosed within double quotes inside the *print()* function.

Even though there are different ways to print values in Python, we discuss two major string formats which are used inside the *print()* function to display the contents onto the console as they are less error prone and results in cleaner code. They are

1. str.format()
2. f-strings

2.11.1 *str.format()* Method

Use str.*format()* method if you need to insert the value of a variable, expression or an object into another string and display it to the user as a single string. The *format()* method returns a new string with inserted values. The *format()* method works for all releases of Python 3.x. The *format()* method uses its arguments to substitute an appropriate value for each format code in the template.

The syntax for *format()* method is,

str.format(p0, p1, ..., k0=v0, k1=v1, ...)

where p0, p1,... are called as positional arguments and, k0, k1,... are keyword arguments with their assigned values of v0, v1,... respectively.

Positional arguments are a list of arguments that can be accessed with an index of argument inside curly braces like {index}. Index value starts from zero.

Keyword arguments are a list of arguments of type *keyword* = *value*, that can be accessed with the name of the argument inside curly braces like {keyword}.

Here, *str* is a mixture of text and curly braces of indexed or keyword types. The indexed or keyword curly braces are replaced by their corresponding argument values and is displayed as a single string to the user.

The term "Method" is used almost exclusively in Object-oriented programming.

'Method' is the object-oriented word for 'function'. A Method is a piece of code that is called by a name that is associated with an object. You shall learn more about Classes and Objects in Chapter 11.

Program 2.1: Program to Demonstrate input() and print() Functions

```
country = input("Which country do you live in?")
print("I live in {0}".format(country))
```

Output

```
Which country do you live in? India
I live in India
```

The 0 inside the curly braces {0} is the index of the first (0th) argument (here in our case, it is variable country ①) whose value will be inserted at that position ②.

Program 2.2: Program to Demonstrate the Positional Change of Indexes of Arguments

```
a = 10
b = 20
print("The values of a is {0} and b is {1}".format(a, b))
print("The values of b is {1} and a is {0}".format(a, b))
```

Output

```
The values of a is 10 and b is 20
The values of b is 20 and a is 10
```

You can have as many arguments as you want, as long as the indexes in curly braces have a matching argument in the argument list ③. {0} index gets replaced with the data value of variable a ① and {1} with the data value of variable b ②. This allows for re-arranging the order of display without changing the arguments ④.

```
1. >>> print("Give me {ball} ball".format(ball = "tennis"))
   Give me tennis ball
```

The keyword argument {ball} gets replaced with its assigned value ①.

2.11.2 f-strings

Formatted strings or f-strings were introduced in Python 3.6. A *f-string* is a string literal that is prefixed with "f". These strings may contain replacement fields, which are expressions enclosed within curly braces {}. The expressions are replaced with their values. In the real world, it means that you need to specify the name of the variable inside the curly braces to display its value. An **f** at the beginning of the string tells Python to allow any currently valid variable names within the string.

We use f-strings along within *print()* function to print the contents throughout this book, as f-strings are the most practical and straightforward way of formatting strings unless some special case arises.

Program 2.3: Code to Demonstrate the Use of f-strings with print() Function

```
1. country = input("Which country do you live in?")
2. print(f"I live in {country}")
```

Output

```
Which country do you live in? India
I live in India
```

Input string is assigned to variable *country* ①. Observe the character **f** prefixed before the quotes and the variable name is specified within the curly braces ②.

Program 2.4: Given the Radius, Write Python Program to Find the Area and Circumference of a Circle

```
1. radius = int(input("Enter the radius of a circle"))
2. area_of_a_circle = 3.1415 * radius * radius
3. circumference_of_a_circle = 2 * 3.1415 * radius
4. print(f"Area={area_of_a_circle}andCircumference={circumference_of_a_circle}")
```

Output

```
Enter the radius of a circle 5
Area = 78.53750000000001 and Circumference = 31.415000000000003
```

Get input for *radius* from the user ① and ② to calculate the *area* of a circle using the formula πr^2 and ③ *circumference* of a circle is calculated using the formula $2\pi r$. Finally, ④ print the results.

Program 2.5: Write Python Program to Convert the Given Number of Days to a Measure of Time Given in Years, Weeks and Days. For Example, 375 Days Is Equal to 1 Year, 1 Week and 3 Days (Ignore Leap Year).

```
1. number_of_days = int(input("Enter number of days"))
2. number_of_years = int(number_of_days/365)
3. number_of_weeks = int(number_of_days % 365 / 7)
4. remaining_number_of_days = int(number_of_days % 365 % 7)
5. print(f"Years = {number_of_years}, Weeks = {number_of_weeks}, Days =
   {remaining_number_of_days}")
```

Output

```
Enter number of days375
Years = 1, Weeks = 1, Days = 3
```

Total number of days is specified by the user ①. Number of years ② is calculated by dividing the total number of days by 365. To calculate the number of weeks ③, deduct the number of days using % 365 and divide by 7. Now deduct the number of days using % 365 and number of weeks by % 7 to calculate the remaining number of days ④. Finally, display the results ⑤.

2.12 Type Conversions

You can explicitly cast, or convert, a variable from one type to another.

2.12.1 The *int()* Function

To explicitly convert a float number or a string to an integer, cast the number using *int()* function.

Program 2.6: Program to Demonstrate int() Casting Function

```
1. float_to_int = int(3.5)
2. string_to_int = int("1") #number treated as string
3. print(f"After Float to Integer Casting the result is {float_to_int}")
4. print(f"After String to Integer Casting the result is {string_to_int}")
```

Output

```
After Float to Integer Casting the result is 3
After String to Integer Casting the result is 1
```

Convert float and string values to integer ①–② and display the result ③–④.

```
1. >>>numerical_value = input("Enter a number")
   Enter a number 9
2. >>> numerical_value
   '9'
3. >>> numerical_value = int(input("Enter a number"))
   Enter a number 9
4. >>> numerical_value
   9
```

①–② User enters a value of 9 which gets assigned to variable *numerical_value* and the value is treated as string type. In order to assign an integer value to the variable, you have to enclose the *input()* function within the *int()* function which converts the input string type to integer type ③–④. A string to integer conversion is possible only when the string value is inherently a number (like "1") and not a character.

2.12.2 The *float()* Function

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

Program 2.7: Program to Demonstrate float() Casting Function

```
1. int_to_float = float(4)
2. string_to_float = float("1") #number treated as string
3. print(f"After Integer to Float Casting the result is {int_to_float}")
4. print(f"After String to Float Casting the result is {string_to_float}")
```

Output

```
After Integer to Float Casting the result is 4.0
After String to Float Casting the result is 1.0
```

Convert integer and string values to float ①–② and display the result ③–④.

2.12.3 The *str()* Function

The *str()* function returns a string which is fairly human readable.

Program 2.8: Program to Demonstrate str() Casting Function

```
1. int_to_string = str(8)
2. float_to_string = str(3.5)
3. print(f"After Integer to String Casting the result is {int_to_string}")
4. print(f"After Float to String Casting the result is {float_to_string}")
```

Output

```
After Integer to String Casting the result is 8
After Float to String Casting the result is 3.5
```

Here, integer and float values are converted ①–② to string using *str()* function and results are displayed ③–④.

2.12.4 The *chr()* Function

Convert an integer to a string of one character whose ASCII code is same as the integer using *chr()* function. The integer value should be in the range of 0–255.

Program 2.9: Program to Demonstrate chr() Casting Function

```
1. ascii_to_char = chr(100)
2. print(f'Equivalent Character for ASCII value of 100 is {ascii_to_char}')
```

Output

```
Equivalent Character for ASCII value of 100 is d
```

An integer value corresponding to an ASCII code is converted ① to the character and printed ②.

2.12.5 The *complex()* Function

Use *complex()* function to print a complex number with the value real + imag*j or convert a string or number to a complex number. If the first argument for the function is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the function serves as a numeric conversion function like int(), long() and float(). If both arguments are omitted, the *complex()* function returns 0j.

Program 2.10: Program to Demonstrate complex() Casting Function

```
1. complex_with_string = complex("1")
2. complex_with_number = complex(5, 8)
3. print(f"Result after using string in real part {complex_with_string}")
4. print(f"Result after using numbers in real and imaginary part {complex_with_
   number}")
```

Output

```
Result after using string in real part (1+0j)
Result after using numbers in real and imaginary part (5+8j)
```

The first argument is a string ①. Hence you are not allowed to specify the second argument. In ② the first argument is an integer type, so you can specify the second argument which is also an integer. Results are printed out in ③ and ④.

2.12.6 The *ord()* Function

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

Program 2.11: Program to Demonstrate ord() Casting Function

```
1. unicode_for_integer = ord('4')
2. unicode_for_alphabet = ord("Z")
3. unicode_for_character = ord("#")
4. print(f"Unicode code point for integer value of 4 is {unicode_for_integer}")
5. print(f"Unicode code point for alphabet 'A' is {unicode_for_alphabet}")
6. print(f"Unicode code point for character '$' is {unicode_for_character}")
```

Output

```
Unicode code point for integer value of 4 is 52
Unicode code point for alphabet 'A' is 90
Unicode code point for character '$' is 35
```

The *ord()* function converts an integer ①, alphabet ② and a character ③ to its equivalent Unicode code point integer value and prints the result ④–⑥.

2.12.7 The *hex()* Function

Convert an integer number (of any size) to a lowercase hexadecimal string prefixed with "0x" using *hex()* function.

Program 2.12: Program to Demonstrate hex() Casting Function

```
1. int_to_hex = hex(255)
2. print(f"After Integer to Hex Casting the result is {int_to_hex}")
```

Output

```
After Integer to Hex Casting the result is 0xff
```

Integer value of 255 is converted to equivalent hex string of 0xff ① and result is printed as shown in ②.

2.12.8 The *oct()* Function

Convert an integer number (of any size) to an octal string prefixed with "0o" using *oct()* function.

Program 2.13: Program to Demonstrate oct() Casting Function

```
1. int_to_oct = oct(255)
2. print(f"After Integer to Hex Casting the result is {int_to_oct}")
```

Output

```
After Integer to Hex Casting the result is 0o377
```

Integer value of 255 is converted to equivalent oct string of 0o37 ① and result is printed as shown in ②.

2.13 The *type()* Function and Is Operator

The syntax for *type()* function is,

type(object)

The *type()* function returns the data type of the given object.

```
1. >>> type(1)
   <class 'int'>
2. >>> type(6.4)
   <class 'float'>
3. >>> type("A")
   <class 'str'>
4. >>> type(True)
   <class 'bool'>
```

The *type()* function comes in handy if you forget the type of variable or an object during the course of writing programs.

The operators *is* and *is not* are identity operators. Operator *is* evaluates to *True* if the values of operands on either side of the operator point to the same object and *False* otherwise. The operator *is not* evaluates to *False* if the values of operands on either side of the operator point to the same object and *True* otherwise.

```
1. >>> x = "Seattle"
2. >>> y = "Seattle"
3. >>> x is y
   True
```

Both *x* and *y* point to same object "Seattle" ①–② and results in *True* when evaluated using *is* operator ③.

2.14 Dynamic and Strongly Typed Language

Python is a dynamic language as the type of the variable is determined during run-time by the interpreter. Python is also a strongly typed language as the interpreter keeps track of all the variables types. In a strongly typed language, you are simply not allowed to do anything that's incompatible with the type of data you are working with. For example,

```
1. >>> 5 + 10
   15
2. >>> 1 + "a"
   Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
   TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In ① values which are of integer types are added and they are compatible. However, in ② when you try to add 1, which is an integer type with "*a*" which is string type, then it results in Traceback as they are not compatible.

In Python, Traceback is printed when an error occurs. The last line tells us the kind of error that occurred which in our case is the unsupported operand type(s).

2.15 Summary

- Identifier is a name given to the variable and must begin with a letter or underscore and can include any number of letter, digits or underscore.
- Keywords have predefined meaning.
- A Variable holds a value that may change.
- Python supports the following operators:
  - Arithmetic Operators
  - Comparison
  - Assignment
  - Logical
  - Bitwise
  - Identity
- Statement is a unit of code that the Python interpreter can execute.
- An expression is a combination of variables, operators, values and reserved keywords.
- The Hash (#) and Triple Quotes (''' ''') are used for commenting.
- Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.
- The print() and input() functions handle simple text output (to the screen) and input (from the keyboard).
- The str(), int(), and float() functions will evaluate to the string, integer, or floating-point number form of the value they are passed.
- A docstring is a string enclosed by triple quotation marks and provides program documentation.