**UNIT II**

**DATA, EXPRESSIONS, and STATEMENTS**

**What is Python??**

Python is a powerful high-level, object-oriented,interpretedprogramming language created by Guido van Rossum.

**Why choose Python as your first language?**

* Simple Elegant Syntax
* Not overly strict
* Expressiveness of the language
* Great Community and Support

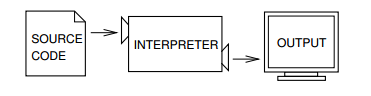
**Features of Python**

* A simple language which is easier to learn
* Free and open-source
* Portability
* Extensible and Embeddable
* A high-level, interpreted language
* Large standard libraries to solve common tasks
* Object-oriented

**Interpreter**

Interpreter is a program that can analyze and execute a program line by line.

***Python Interpreter***



Python Interpreter`s Mode of operation:

* Python has two basic modes: normal and interactive.
* The normal mode is the mode where the scripted and finished .py files are run in the Python interpreter.
* Interactive mode is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory.
* As new lines are fed into the interpreter, the fed program is evaluated both in part and in whole.

**In Normal Mode:**

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script



Produces the output

1

2

The assignment statement produces no output.

**In Interactive Mode:**

You can test bits of code in interactive mode before you put them in a script. For example, if you are using Python as a calculator, you might type,

>>> miles = 26.2

>>> miles \* 1.61

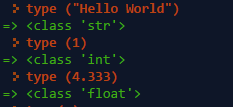
42.182

* The first line assigns a value to miles, but it has no visible effect.
* The second line is an expression, so the interpreter evaluates it and displays the result.
* So we learn that the outcome of the expression is about 42 kilometers.

**VALUES AND TYPES**

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!’ .

These values belong to different types.For instance,2 is a integer, and 'Hello, World!' is a string. The interpreter can tell you what type a value has and it is shown below



**Variables**

A variable is a name that refers to a value. An assignment statement creates new variables and gives them values.

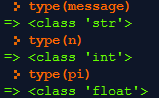
**>>>** Message = 'And now for something completely different'

**>>>** n = 17

**>>>** Pi = 3.1415926535897932

This example makes three assignments. The first assigns a string to a new variable named message; the second gives the integer 17 to n; the third assigns the (approximate) value of π to pi.

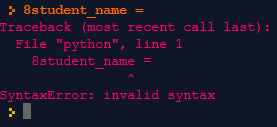
The type of a variable is the type of the value it refers to.



**Variable names and keywords**

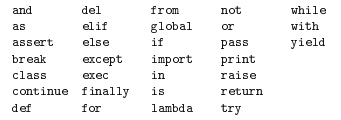
Programmers generally choose names for their variables that are meaningful, they document what the variable is used for. Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter. The underscore character, \_, can appear in a name. It is often used in names with multiple words, such as **my\_name** or **airspeed\_of\_unladen\_swallow**.

If you give a variable an illegal name, you`ll get a syntax error as shown below,

****

**Python has 31 keywords:**

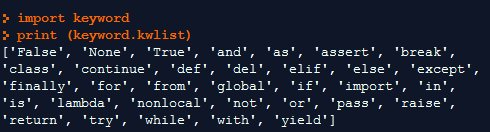
Keywords are the reserved words in Python. We cannot use a keyword as variable name, function name or any other identifier. The interpreter uses **keywords** to recognize the structure of the program.

****

The above keywords may get altered in different versions of Python. Some extra might get added or some might be removed. You can always get the list of keywords in your current version by typing the following in the prompt.

>>>Import Keyword

>>>Print (keyword.kwlist)

****

**PYTHON OPERATORS**

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand. Python has a number of operators which are classified below.

* + Arithmetic operators
  + Comparison (Relational) operators
  + Logical (Boolean) operators
  + Bitwise operators
  + Assignment operators
  + Special operators

**Arithmetic operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| + | Add two operands or unary plus | x + y+2 |
| - | Subtract right operand from the left or unary minus | x - y-2 |
| \* | Multiply two operands | x \* y |
| / | Divide left operand by the right one (always results into float) | x / y |
| % | Modulus - remainder of the division of left operand by the right | x % y (remainder of x/y) |
| // | Floor division - division that results into whole number adjusted to the left in the number line | x // y |
| \*\* | Exponent - left operand raised to the power of right | x\*\*y (x to the power y) |

## Comparison operators

Comparison operators are used to compare values. It either returns True or False according to the condition.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| > | Greater that - True if left operand is greater than the right | x > y |
| < | Less that - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

## Logical operators

Logical operators are the **and, or, not** operators.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| and | True if both the operands are true | x and y |
| or | True if either of the operands is true | x or y |
| not | True if operand is false (complements the operand) | not x |

## Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

**In the table below:** Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| & | Bitwise AND | x& y = 0 (0000 0000) |
| | | Bitwise OR | x | y = 14 (0000 1110) |
| ~ | Bitwise NOT | ~x = -11 (1111 0101) |
| ^ | Bitwise XOR | x ^ y = 14 (0000 1110) |
| >> | Bitwise right shift | x>> 2 = 2 (0000 0010) |
| << | Bitwise left shift | x<< 2 = 40 (0010 1000) |

## Assignment operators

Assignment operators are used in Python to assign values to variables. a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left. There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example** | **Equivatent to** |
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| \*= | x \*= 5 | x = x \* 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| //= | x //= 5 | x = x // 5 |
| \*\*= | x \*\*= 5 | x = x \*\* 5 |
| &= | x &= 5 | x = x & 5 |
| |= | x |= 5 | x = x | 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| >>= | x >>= 5 | x = x >> 5 |
| <<= | x <<= 5 | x = x << 5 |

## Special operators

Python language offers some special type of operators like the identity operator or the membership operator. They are described below with examples.

### Identity operators

**is** and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| is | True if the operands are identical (refer to the same object) | x is True |
| is not | True if the operands are not identical (do not refer to the same object) | x is not True |

### Membership operators

**in** and **not in** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence ([string](https://www.programiz.com/python-programming/string), [list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [set](https://www.programiz.com/python-programming/set) and [dictionary](https://www.programiz.com/python-programming/dictionary)).In a dictionary we can only test for presence of key, not the value.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| in | True if value/variable is found in the sequence | 5 in x |
| not in | True if value/variable is not found in the sequence | 5 not in x |

**PRECEDENCE OF OPERATORS**

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

* **Parentheses** have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60, even if it doesn’t change the result.
* **Exponentiation** has the next highest precedence, so 2\*\*1+1 is 3, not 4, and 3\*1\*\*3 is 3, not 27.
* **Multiplication** and **Division** have the same precedence, which is higher than **Addition** and **Subtraction**, which also have the same precedence. So 2\*3-1 is 5, not 4, and 6+4/2 is 8, not 5.
* Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression **degrees / 2 \* pi**, the division happens first and the result is multiplied by **pi**. To divide by **2π**, you can use parentheses or write **degrees / 2 / pi**.

**COMMENT**

In Python there are basically two ways to comment: **Single line** and **Multiple line**. Single line commenting is good for a short, quick comment (or for debugging), while the block comment is often used to describe something much more in detail or to block out an entire chunk of code.

***One Line Comments***

Typically, you just use the # (pound) sign to comment out everything that follows it on that line.

***Example***

print("Not a comment")  
#print("Am a comment")Result

***output :***

Not a comment

### Multiple Line Comments

Multiple line comments are slightly different. Simply use three single quotes before and after the part you want commented.

Example

'''  
print("We are in a comment")  
print ("We are still in a comment")  
'''  
print("We are out of the comment")

***Output***

We are out of the comment

**EXPRESSIONS**

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions

5+4

Area=L\*B

x + 17

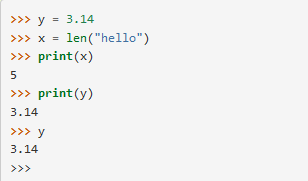
Expressions need to be evaluated. If you ask Python to print an expression, the interpreter **evaluates** the expression and displays the result.

The **evaluation of an expression** produces a value, which is why expressions can appear on the right hand side of assignment statements.

**STATEMENTS**

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statement: print and assignment. Technically an expression is also a statement, but it is probably simpler to think of them as different things. The important difference is that an expression has a value; a statement does not.

Consider the example below,



When we enter the assignment statement, y = 3.14, only the prompt is returned. There is no value. This is due to the fact that statements, such as the assignment statement, do not return a value. They are simply executed. On the other hand, the result of executing the assignment statement is the creation of a reference from a variable, y, to a value, 3.14 When we execute the print function working on y, we see the value that y is referring to. In fact, evaluating y by itself results in the same response.

**DATA TYPES**

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types

* Numbers
* String
* List
* Tuple
* Dictionary

## Python Numbers

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

For example,

var1 = 1

var2 = 10

Python supports different numerical types,

* **int (signed integers)** − They are often called just integers or **ints**. They are positive or negative whole numbers with no decimal point. Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no '**long integer**' in Python 3 anymore.
* **float (floating point real values)** − Also called floats, they represent real numbers and are written with a decimal point dividing the integer and the fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).
* **complex (complex numbers)** − are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

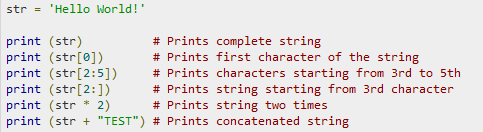
|  |  |  |
| --- | --- | --- |
| **int** | **float** | **complex** |
| 10 | 0.0 | 3.14j |
| 100 | 15.20 | 45.j |
| -786 | -21.9 | 9.322e-36j |
| 080 | 32.3+e18 | .876j |
| -0490 | -90. | -.6545+0J |
| -0×260 | -32.54e100 | 3e+26J |
| 0×69 | 70.2-E12 | 4.53e-7j |
|  |  |  |

**Value and its types**

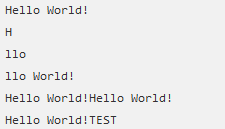
## PYTHON STRINGS

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator.



This will produce the following result,

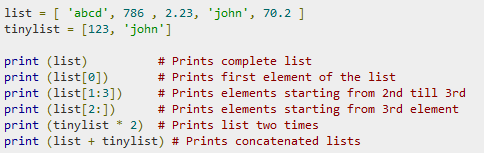


## PYTHON LISTS

A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type.

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.

For example



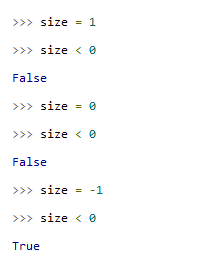
## This produces the following result

## 

**BOOLEAN**

Booleans are either true or false. Python has two constants, cleverly named True and False, which can be used to assign booleanvalues directly. Expressions can also evaluate to a boolean value.

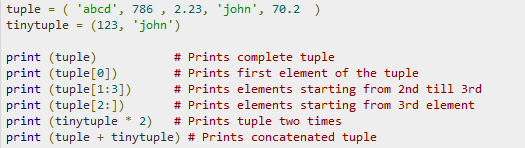
In certain places (like if statements), Python expects an expression to evaluate to a boolean value. These places are called *boolean contexts*. You can use virtually any expression in a boolean context, and Python will try to determine its truth value.



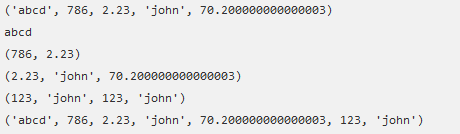
**TUPLES**

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The main difference between lists and tuples are, Lists are enclosed in square 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.

For example



This produces the following result

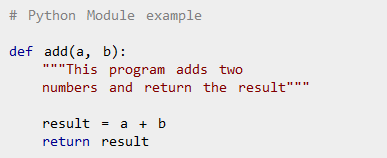


**MODULES**

Modules refer to a file containing Python statements and definitions. A file containing Python code, for e.g.: **example.py,** is called a module and its module name would be example. We use modules to break down large programs into small manageable and organized files.

Furthermore, modules provide reusability of code. We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.



Here, we have defined a function add () inside a module named example. The function takes in two numbers and returns their sum.

## IMPORT MODULES

We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.



This does not enter the names of the functions defined in example directly in the current symbol table. It only enters the module name example there.

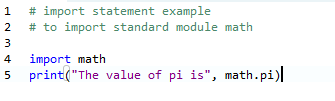
Using the module name we can access the function using dot (.) operation. For example:



There are various ways to import modules. They are listed as follows.

### *Python import statement*

We can import a module using import statement and access the definitions inside it using the dot operator as described above. Here is an example.

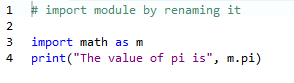


When you run the program, the output will be :



### *Import with renaming*

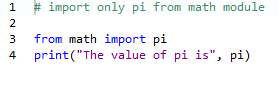
We can import a module by renaming it as follows.



We have renamed the math module as m. This can save us typing time in some cases. Note that the name math is not recognized in our scope. Hence, **math.pi** is invalid**, m.pi** is the correct implementation.

### Python *from...import statement*

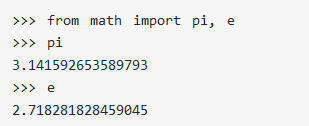
We can import specific names from a module without importing the module as a whole. Here is an example.



***Output:***

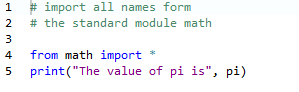


We imported only the attribute pi form the module. In such case we don't use the dot operator. We could have imported multiple attributes as follows.



### *Import all names*

We can import all names (definitions) from a module using the following construct.

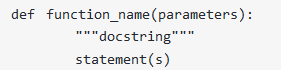


We imported all the definitions from the math module. This makes all names except those beginning with an underscore, visible in our scope. Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

# PYTHON FUNCTIONS

Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.

## Syntax of Function



Above shown is a function definition which consists of following components.

1. Keyword def marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

## Example of a function

## *Function Call*

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.



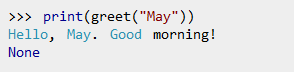
## *The return statement*

The return statement is used to exit a function and go back to the place from where it was called.

### *Syntax of return:*

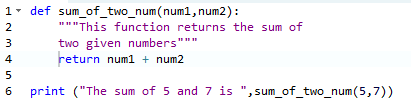


This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.



Here, None is the returned value.

### *Example of return*



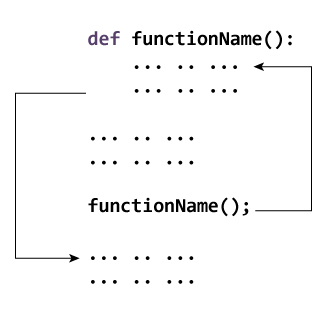
***Output:***



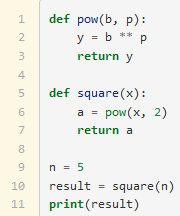
# FLOW OF EXECUTION

When we are working with functions it is really important to know the order in which statements are executed. This is called the **flow of execution**

Execution always begins at the first statement of the program. Statements are executed one at a time, in order, from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off. It is shown pictorially below.



Consider the following Python code.



1. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
2. 1, 2, 3, 5, 6, 7, 9, 10, 11
3. 9, 10, 11, 1, 2, 3, 5, 6, 7
4. 9, 10, 5, 6, 7, 1, 2, 3, 11
5. 1, 5, 9, 10, 6, 2, 3, 7, 11

Python starts at line 1, notices that it is a function definition and skips over all of the lines in the function definition until it finds a line that it no longer included in the function (line 5).

It then notices line 5 is also a function definition and again skips over the function body to line 9.

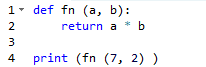
On line 10 it notices it has a function to execute, so it goes back and executes the body of that function. Notice that that function includes another function call. Finally, it will return to line 11 after the function square is complete.

**PARAMETERS AND ARGUMENTS**

***Arguments***

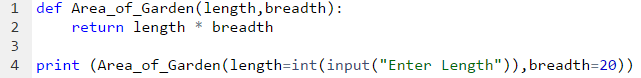
Arguments are part of a function call. There are four different kinds of arguments:

**Positional:** Arguments without a name or is not followed by an equal sign (=) and default value.



Here 7,2 are the positional arguments.

**Keyword:** A keyword **argument** is followed by an equal sign and an expression that gives its default value.

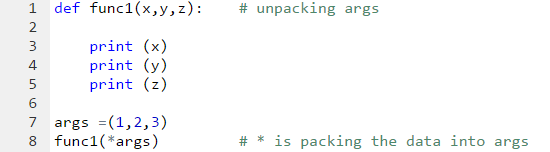


In the above code the first argument in the function call is equal to an expression and the second argument contains a default value.

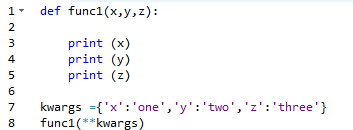
**Packed positional:** These arguments by \*. Then the argument is unpacked, and the values are treated as separate positional arguments. That \* and the \*\* operators both perform two different, but complementary operations depending on where they're used.

They perform an operation called 'packing'. True to it's name, what this does is *pack* all the arguments that this method call receives into one single variable, a tuple called args. You can use any variable name you want, of course, but args seems to be the most common and Pythonic way of doing things.

The \* operator can be used in the context of a method call. What it does now is explode the **args** array and call the method as if you'd typed in each variable separately.



**Packed keyword:** An argument which is treated as a mapping will be preceded by \*\*

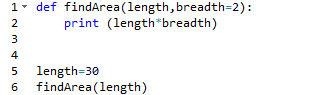


In the above example the dictionary data kwargs is packed by \*\* during function call. In the function definition this kwargs is unpacked and each parameter is considered as a keyword and it is been mapped with the arguments passed and the corresponding value is taken for that parameter.

***Parameters***

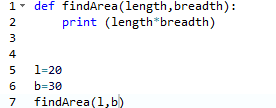
Parameters are part of a function definition. There are four different kinds of parameters:

**Positional-or-keyword:** Normal parameters in a function definition, with or without default values.

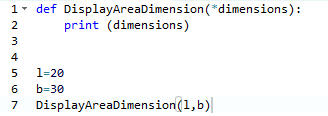
****

In the example above the function definition contains a default parameter for the breadth, so even though the caller has not supplied the necessary argument the default parameter value is taken in to consideration.

**Positional-only:** This Positional-only parameter Accepts argument without default value.

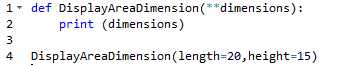


**Var-positional:** This kind of parameter accepts variable number of arguments without any default value and the parameter is usually preceded with \*.



Here in the above program the various positional argument from the function call is been packed into a tuple in the function definition.

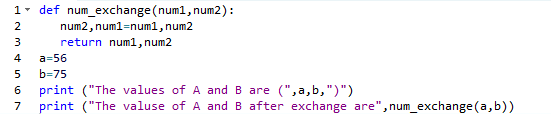
**Var-keyword:** This type of parameter is used to take or accept variable length argument from a function call with default value. This kind of argument is preceded with \*\*.



In the above example the default valued arguments from the function call is been packed into a dictionary in the function definition. It can be simply displayed in the print statement or it can be iterated.

**ILLUSTRATIVE PROGRAMS**

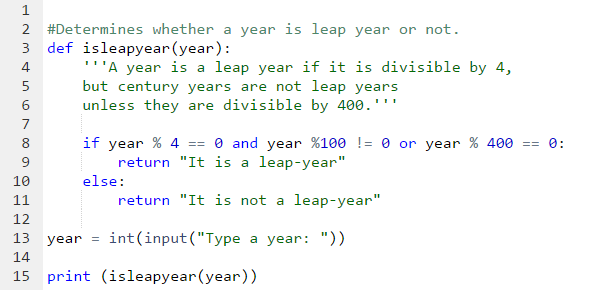
1. Exchange the values of two variables

****

Output

****

B) Test for leap year

****

**Output:**

****

C) Create a Python program to circulate the values of n variables. Try it using all

the concepts learnt as of now.