Introduction to Python Programming (BPLCK205B)

Module 1. Python Basics

The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features.

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- Web development (server-side).
- Software development.
- Mathematics.
- System scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Good to know:

- The most recent major version of Python is Python 3.
- Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Entering Expressions into the Interactive Shell

We can run the interactive shell by launching IDLE, which can be installed with Python.

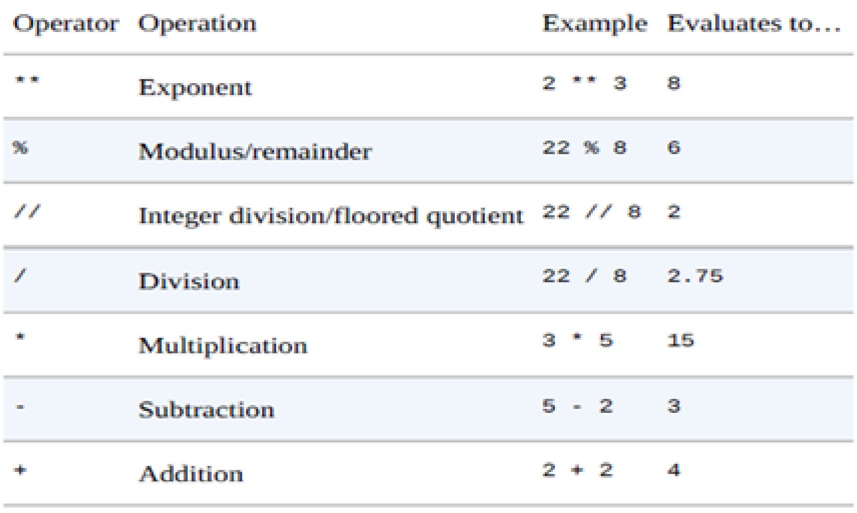
- ❖ On Windows, open the Start menu, select **All Programs** ▸ **Python 3.3**, and then select **IDLE (Python GUI)**.
- ❖ On OS X, select **Applications** ▸ **MacPython 3.3** ▸ **IDLE**.
- ❖ On Ubuntu, open a new Terminal window and enter **idle3**.

Q) List out the mathematical operators used to Python programming and give example.

OR

What is the need for role of precedence? Illustrate the rules of precedence in Python with an example.

The precedence of operators in Python program dictates the order in which the operators will be evolved in an expression. Associativity, on the other hand, defines the order in which the operators of the same precedence will be evaluated in an expression. Also, associativity can occur from either right to left or left to right.
The following is the list of operators we can use in Python expressions.



Operator	Operation	Example	Evaluates to...
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

The order of operations (also called *precedence*) of Python math operators is similar to that of mathematics. The ** operator is evaluated first; the *, /, //, and % operators are evaluated next, from left to right; and the + and - operators are evaluated last (also from left to right).

Example:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
```

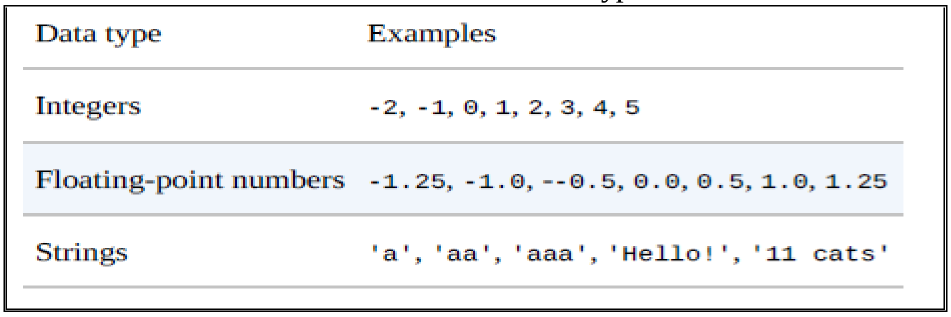
```
>>> 23 % 7
2
>>> 2    +           2
4
>>> ( 5  -   1) * ((7 + 1) / ( 3 - 1))
16.0
```

The Integer, Floating-Point, and String Data Types:

Q) Define expression? List and explain the different data types used in Python programming.

The expressions are just values combined with operators, and they always evaluate down to a single value. A data type is a category for values, and every value belongs to exactly one data type. The most common data types in Python are listed in the table below.

Table. Common Data Types



Data type	Examples
Integers	`-2, -1, 0, 1, 2, 3, 4, 5`
Floating-point numbers	`-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25`
Strings	`'a', 'aa', 'aaa', 'Hello!', '11 cats'`

Example:
The values -2 and 30, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers.

Numbers with a decimal point, such as 3.14, are called *floating-point numbers* (or *floats*). Note that even though the value 42 is an integer, the value 42.0 would be a floating-point number.

Python programs can also have text values called *strings*, or *strs* (pronounced “stirs”). Always surround string in single quote (') characters (as in 'Hello' or 'Goodbye cruel world!') so Python knows where the string begins and ends. We can even have a string with no characters in it, '', called a *blank string*.

```
>>> 'Hello world!
SyntaxError: EOL while scanning string literal
```

Q) Briefly explain String Concatenation and Replication. Give Example.

The meaning of an operator may change based on the data types of the values next to it. For example, + is the addition operator when it operates on two integers or floating-point values. However, when + is used on two string values, it joins the strings as the *string concatenation* operator.

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if we try to use the + operator on a string and an integer value, Python will display an error message.

Example:

```
>>> 'Alice' + 42
Traceback (most recent call last):
File "<pyshell#26>", line 1, in <module>
'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

The * operator is used for multiplication when it operates on two integer or floating-point values. But when the * operator is used on one string value and one integer value, it becomes the *string replication* operator.

Example:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The * operator can be used with only two numeric values (for multiplication) or one string value and one integer value (for string replication). Otherwise, Python will just display an error message.

Example:

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
File "<pyshell#33>", line 1, in <module>
'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Q) Explain the concept of Storing Values in Variables in Python.

A *variable* is like a box in the computer's memory where we can store a single value. If we want to use the result of an evaluated expression later in our program, we can save it inside a variable.

Assignment Statements

An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If we enter the assignment statement spam = 42, then a variable named spam will have the integer value 42 stored in it.

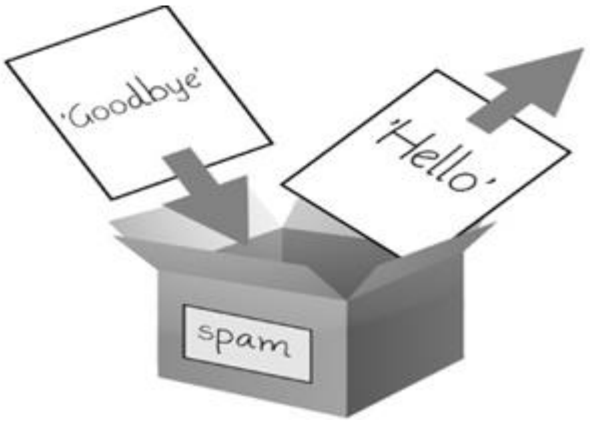
Example:	

`>>> spam = 40`
`>>> spam`
`40`
`>>> eggs = 2`
`>>> spam + eggs`
`42`
`>>> spam + eggs + spam`
`82`
`>>> spam = spam + 2`
`>>> spam`
`42`
	Figure: spam=42 is like telling the program, "The variable spam now has the integer value 42 in it.

Example:

Q) Explain the concept of overwriting the variable and give example.

A variable is *initialized* (or created) the first time a value is stored in it After that, we can use it in expressions with other variables and values. When a variable is assigned a new value, the old value is replaced with new value. This is called *overwriting* the variable.

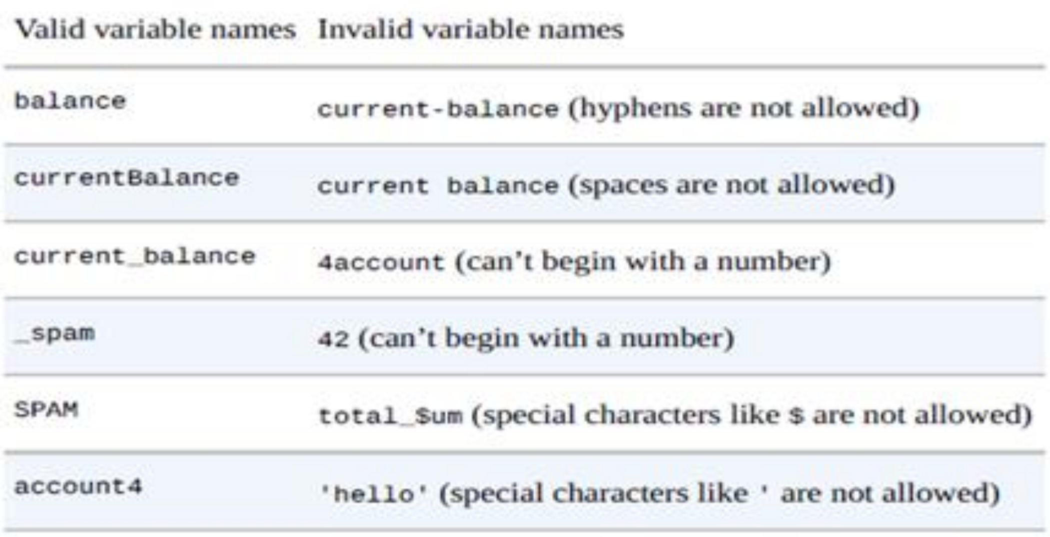
Example:
>>> **spam = 'Hello'**
>>> **spam**
'Hello'
>>> **spam = 'Goodbye'**
>>> **spam**
'Goodbye'
	The spam variable in this example stores 'Hello' until we replace it with 'Goodbye'.
	
	
	Figure: When a new value is assigned, the old one is forgotten

Q) What are the rules to be followed while naming a Variable.

The following table has examples of legal variable names. We can name a variable anything as long as it obeys the following three rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (_) character.
3. It can't begin with a number.

Table: Valid and Invalid Variable Names



Valid variable names	Invalid variable names
`balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`current_balance`	`4account` (can't begin with a number)
`_spam`	`42` (can't begin with a number)
`SPAM`	`total_$um` (special characters like `$` are not allowed)
`account4`	`'hello'` (special characters like `'` are not allowed)

Variable names are case-sensitive, meaning that spam, SPAM, Spam, and sPaM are four different variables. It is a Python convention to start our variables with a lowercase letter.

Example Program

```
# This program says hello and asks for my name.
print('Hello world!')
print('What is your name?') # ask for their name
myName = input()
print('It is good to meet you, ' + myName)
print('The length of your name is:')
print(len(myName))
print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

The program's output in the interactive shell:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================== RESTART ====================
>>>
Hello world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2 What is your age?
4
You will be 5 in a year.
>>>
```

Q) Briefly explain about the commenet statement in Python.

The following line is called a *comment* statement in Python.

# This program says hello and asks for my name.

Python ignores comments, and we can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when we're trying to figure out why a program doesn't work. We can remove the # later when we are ready to put the line back in.

Python also ignores the blank line after the comment. We can add as many blank lines to our program as we want. This can make our code easier to read, like paragraphs in a book.

Q) Briefly explain print() and input() functions. Give example.

The print() Function

The print() function displays the string value inside the parentheses on the screen.

Example:

```
print('Hello world!')
print('What is your name?') # ask for their name
```

The line print('Hello world!') means "Print out the text in the string 'Hello world!'." When Python executes this line, we say that Python is *calling* the print() function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument*. Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.

The following call to print() actually contains the expression 'It is good to meet you, ' + myName between the parentheses.
Example:
print('It is good to meet you, ' + myName)

If 'Al' is the value stored in myName on the previous line, then this expression evaluates to 'It is good to meet you, Al'.

The input() Function

The input() function waits for the user to type some text on the keyboard and press ENTER.
Example:
myName = input()
This function call evaluates to a string equal to the user's text, and the previous line of code assigns the myName variable to this string value.

The input() function call as an expression that evaluates to whatever string the user typed in. If the user entered 'Al', then the expression would evaluate to myName = 'Al'.

Q) Briefly explain len() functions with an example.

The len() Function
The len() function takes a string value as a parameter (input) to the function, and the function evaluates to the integer value of the number of characters in that string.
Example: print('The length of your name is:')
print(len(myName))

Enter the following into the interactive shell to try this:
```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

Q) Briefly explain str(), int() and float() functions with an example.
OR
What three functions can be used to get the integer, floating-point number, or string version of a value? Give example.

The str(), int() and float() Functions

If you want to concatenate an integer such as 29 with a string to pass to print(), you'll need to get the value '29', which is the string form of 29. The str() function can be passed an integer value and will evaluate to a string value version of it, as follows:
Example:
```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

The str(), int(), and float() functions will evaluate to the string, integer, and floatingpoint forms of the value we pass, respectively.

Example:

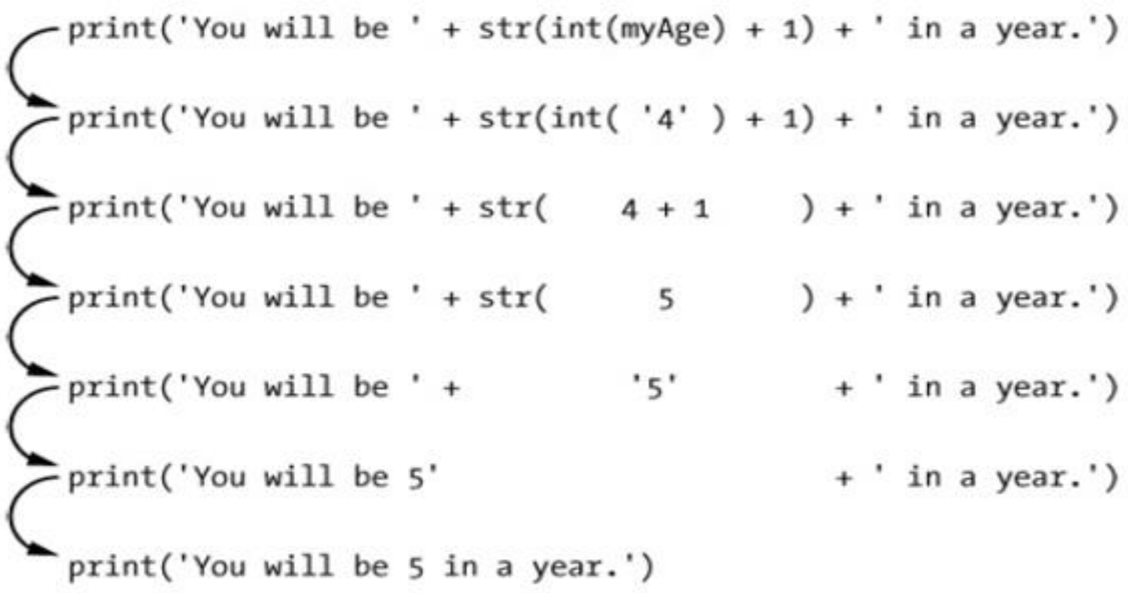
```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1 >>> int(1.99)
1 >>> float('3.14')
3.14
>>> float(10)
10.0
```

Q) Briefly explain about text and number equivalence with an example.

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python makes this distinction because strings are text, while integers and floats are both numbers.



```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')
print('You will be ' + str(    4 + 1     ) + ' in a year.')
print('You will be ' + str(      5       ) + ' in a year.')
print('You will be ' +          '5'        + ' in a year.')
print('You will be 5'                      + ' in a year.')
print('You will be 5 in a year.')
```

Figure: The evaluation steps, if 4 is stored in myAge.

Practice Questions

Q: 1. Which of the following are operators, and which are values?

```
*
‘hello'
-88.8
-
/
+
5
```

Q: 2. Which of the following is a variable, and which is a string?

```
spam
'spam'
```

Q: 3. Name three data types.

Q: 4. What is an expression made up of? What do all expressions do?

Q: 5. What is the difference between an expression and a statement?

Q: 6. What does the variable bacon contain after the following code runs?

```
bacon = 20
bacon + 1
```

Q: 7. What should the following two expressions evaluate to?

```
'spam' + 'spamspam'
'spam' * 3
```

Q: 8. Why is eggs a valid variable name while 100 is invalid?

Q: 9. What three functions can be used to get the integer, floating-point number, or string version of a value?

Q: 10. Why does this expression cause an error? How can you fix it?

```
'I have eaten ' + 99 + ' burritos.'
```

Flow Control

Q) Briefly explain about the flow control statements in Python Programming and give example.

Flow control statements can decide which Python instructions to execute under which conditions. These flow control statements directly correspond to the symbols in a flowchart.
Example: Figure below shows a flowchart for what to do if it's raining.

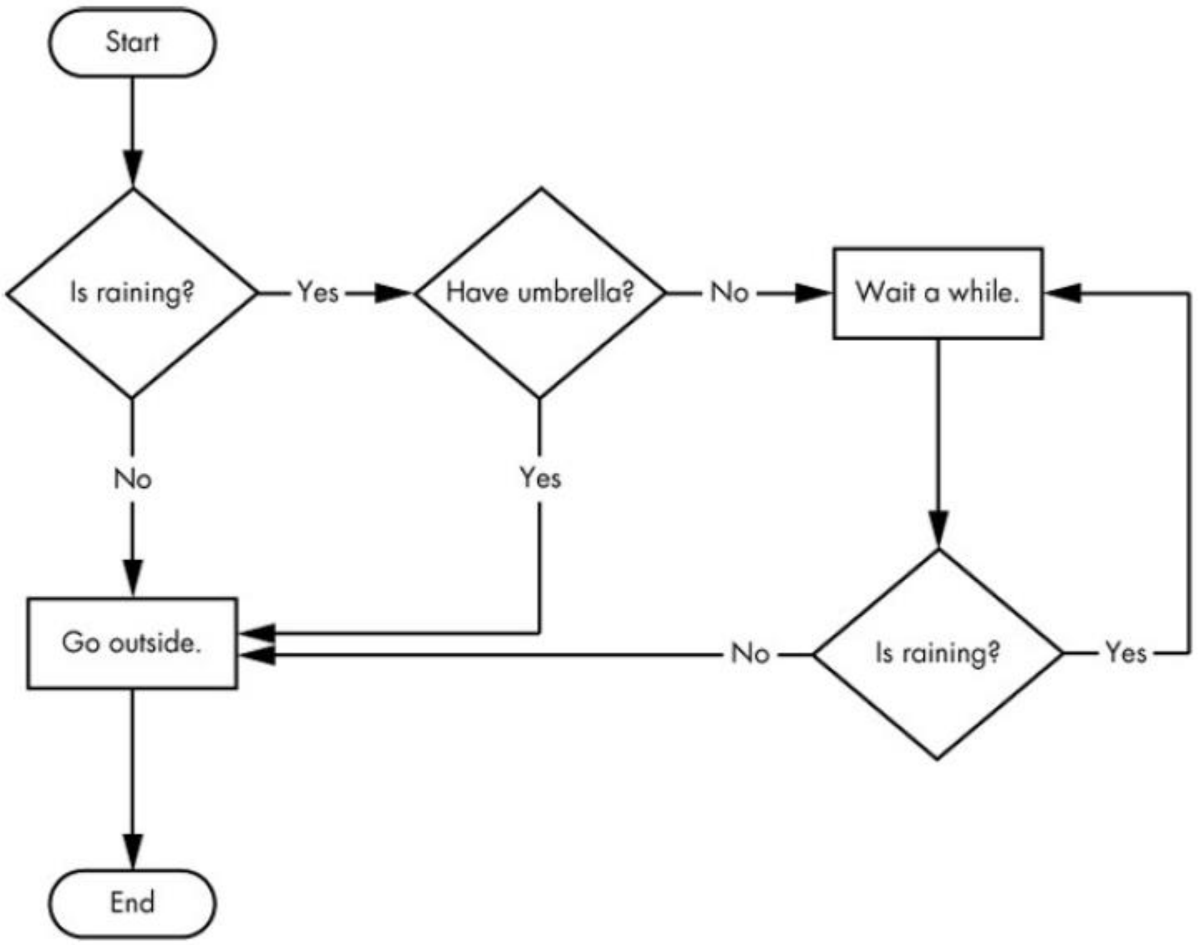


Figure: A flowchart to tell us what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

Q) Briefly explain about Boolean Values in Python programming. Give example.

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: True and False. (Boolean is capitalized because the data type is named after mathematician George Boole.) When typed as Python code, the Boolean Values True and False lack the quotes we place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase.

Example:

```
>>> spam = True
>>> spam
True
>>> true
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
true
NameError: name 'true' is not defined
>>> True = 2 + 2
SyntaxError: assignment to keyword
```

Q) List and explain Comparison Operators used in Python programming. Give example.

Comparison operators compare two values and evaluate down to a single Boolean value.

Table: Comparison Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

These operators evaluate to True or False depending on the values we give them.

Example:

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As we might expect, == (equal to) evaluates to True when the values on both sides are the same, and != (not equal to) evaluates to True when the two values are different. The == and != operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
>>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression 42 == '42' evaluates to False because Python considers the integer 42 tobe different from the string '42'.

The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
>>> eggCount <= 42
True
>>> myAge = 29
>>> myAge >= 10
True
```

Q) Briefly explain about Boolean Operators used in Python programming. Give example.

The three Boolean operators (and, or, and not) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value.

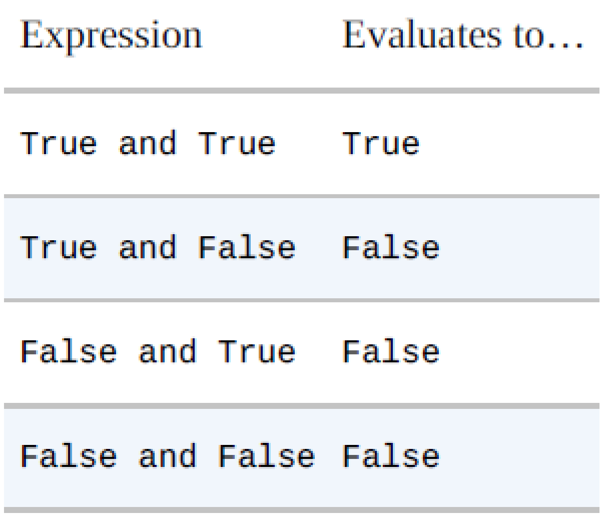
Binary Boolean Operators

The and and or operators always take two Boolean values (or expressions), so they're considered binary operators. The and operator evaluates an expression to True if both Boolean values are True; otherwise, it evaluates to False. Enter some expressions using and into the interactive shell to see it in action.

```
>>> True and True
```

True
>>> **True and False**
False
A truth table shows every possible result of a Boolean operator. Table below shows the truth table for the and operator.

Table. The and Operator’s Truth Table

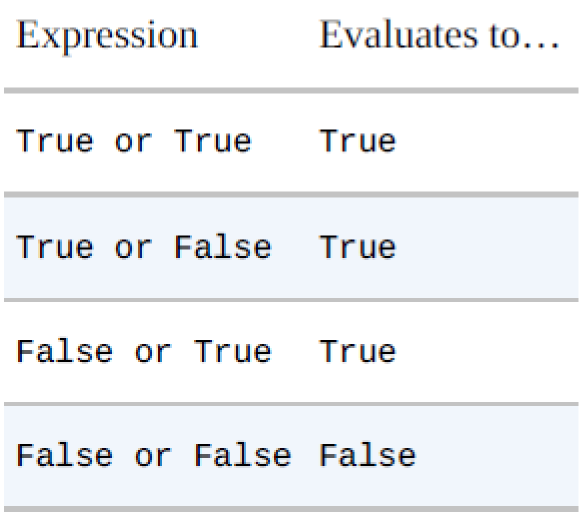


Expression	Evaluates to…
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

On the other hand, the or operator evaluates an expression to True if either of the two Boolean values is True. If both are False, it evaluates to False.
>>> **False or True**
True
>>> **False or False**
False

They possible outcome of the or operator in its truth table, is shown in below Table.

Table. The or Operator’s Truth Table



Expression	Evaluates to…
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

The not Operator
Unlike and and or, the not operator operates on only one Boolean value (or expression). The not operator simply evaluates to the opposite Boolean value.
>>> **not True**

```
False
>>> not not not not True
True
```

The Table below shows the truth table for not.

Table. The not Operator’s Truth Table

Expression	Evaluates to…
`not True`	`False`
`not False`	`True`

Q) Briefly explain about Mixing Boolean and Comparison Operators. Give example.

Since the comparison operators evaluate to Boolean values, we can use them in expressions with the Boolean operators. The and, or, and not operators are called Boolean operators because they always operate on the Boolean values True and False. While expressions like 4 < 5 aren’t Boolean values, they are expressions that evaluate down to Boolean values.

Example:

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer’s evaluation process for (4 < 5) and (5 < 6) as shown in Figure below.

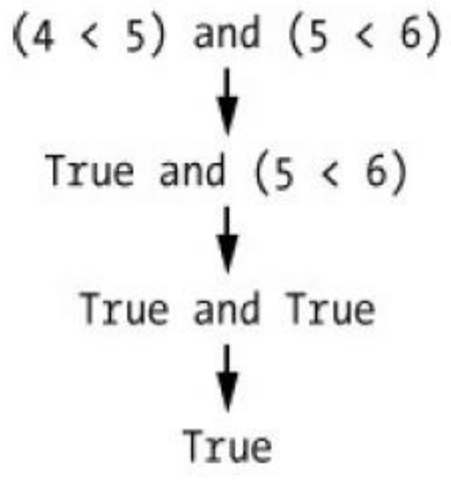


Figure. The process of evaluating (4 < 5) and (5 < 6) to True.

We can also use multiple Boolean operators in an expression, along with the comparison operators.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

Note: The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the not operators first, then the and operators, and then the or operators.

Q) What is meant by Condition and Blocks of Code in Python Programming. Give example.

Flow control statements often start with a part called the condition, and all are followed by a block of code called the clause.

Conditions

The Boolean expressions are considered conditions, which are the same thing as expressions; condition is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, True or False. A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in blocks. We can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.
1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block’s indentation.

Example:

```
if name == 'Mary':
        print('Hello Mary')
if password == 'swordfish':
        print('Access granted.')
else:
        print('Wrong password.')
```

The first block of code starts at the line print('Hello Mary') and contains all the lines after it. Inside this block is another block, which has only a single line in it: print('Access Granted.'). The third block is also one line long: print('Wrong password.').

Q) Briefly Explain Flow Control Statements used in Python Programming. Give syntax and example.

The different control statements used in Python programming are as follows:

i) if statements.
ii) else statements.
iii) elif statements.

if Statements

The most common type of flow control statement is the if statement. An if statement’s clause (that is, the block following the if statement) will execute if the statement’s condition is True. The clause is skipped if the condition is False.
In Python, an if statement consists of the following:

Syntax		
if(condition): statement	OR	if(condition): statement1 statement2 … statementn

Where

- if is the keyword.
- condition is an expression that evaluates to True or False.
- a colon.
- Starting on the next line, an indented block of code (called the if clause).

Example 1:

```
if name == 'Alice':
        print('Hi, Alice.')
```

Example 2:

```
if name == 'Alice':
        print('Hi,')
        print('Alice.')
```

else Statements

An if clause can optionally be followed by an else statement. The else clause is executed only when the if statement's condition is False.

In Python, an if statement consists of the following:

Syntax		

if(condition):
statement1
else:
statement2
	OR	if(condition):
		statement_1
		statement_2
		…
		statement_m
		else
		statement_1
		statement_2
		…
		statement_n

Where

- if is the keyword.
- condition is an expression that evaluates to True or False.
- a colon.
- Starting on the next line, an indented block of code (called the if clause).
- else is a keyword,
- colon Starting on the next line, an indented block of code (called the else clause).

Example1:

```
if name == 'Alice':
        print('Hi, Alice.')
else:
        print('Hello, stranger.')
```

Example2:

```
    if name == 'Alice':
        print('Hi,')
        print('Alice.')

else:
    print('Hello,')
    print('stranger.')
```

elif Statements

While only one of the if or else clauses will execute, we may have a case where we want one of many possible clauses to execute. The elif statement is an “else if” statement that always follows an if or another elif statement. It provides another condition that is checked only if any of the previous conditions were False.

Syntax		

if(condition1):
    statement1
elif(condition2:
    statement2
elif(condition3):
    statement3
	OR	if(condition1):
		    statement_1
		    statement_2
		    …
		    statement_m
		elif(condition2):
		    statement_1
		    statement_2
		    …
		    statement_n
		else
		    statement_1
		    statement_2
		    …
		    statement_p

OR		

if(condition1):
    statement1
elif(condition2:
    statement2
elif(condition3):
    statement3
else
    statement4		

Where

- if is the keyword.
- condition is an expression that evaluates to True or False.
- a colon.
- Starting on the next line, an indented block of code (called the if clause).
- elif is the keyword, A condition is, an expression that evaluates to True or False) and a colon.

Example:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

Q) Briefly explain looping statements used in Python Programming. Give syntax and example.

while Loop Statements

We can make a block of code execute over and over again with a while statement. The code in a while clause will be executed as long as the while statement's condition is True. The syntax is as follows:

Syntax		
while(condition): statement	OR	while(condition): statement1 statement2 … statementn

Where

- while is the keyword,
- condition is, an expression that evaluates to True or False.
- a colon.
- Starting on the next line, an indented block of code (called the while clause).

Example:

```
spam = 0
while spam < 5:
        print('Hello, world.')
        spam = spam + 1
```

Output:

```
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
```

Q) Briefly explain about break and continue statements used in Python Programming. Give syntax and example.

break Statements

There is a shortcut to getting the program execution to break out of a while loop's clause early. If the execution reaches a break statement, it immediately exits the while loop's clause. In code, a break statement simply contains the break keyword.

Example:

```
while True:
        print('Please type your name.')
        name = input()
        if name == 'xyz':
                break
        print('Thank you!')
```

continue Statements

Like break statements, continue statements are used inside loops. When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and re-evaluates the loop's condition.

Example:

```
while True:
        print('Who are you?')
        name = input()
        if name != 'Joe':
                continue
        print('Hello, Joe. What is the password? (It is a fish.)')
        password = input()
        if password == 'swordfish':
                break
        print('Access granted.')
```

Q) Briefly explain for loops and range() function used in Python Programming. Give syntax and example.

for Loops and the range() Function

The while loop keeps looping while its condition is True (which is the reason for its name), but what if we want to execute a block of code only a certain number of times, then we can do this with a for loop statement and the range() function. The syntax is as follows:

Syntax		

for variable in range(n1, n2, n3):
        statement
	OR	for variable in range(n1, n2, n3):
		        statement1
		        statement2
		        …
		        statementn

Where

- for is the keyword.
- variable is the name of the variable.
- in is the keyword.
- range() is a method with up to three integers passed to it.
- A colon.
- Starting on the next line, an indented block of code (called the for clause).

Example:

```
print('My name is')
for i in range(5):
        print('Jimmy Five Times (' + str(i) + ')')
```

Output:

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

Example: Program to add numbers from 0 to 100.

```
total = 0
for num in range(101):
        total = total + num
print(total)
```

Output:

```
5050
```

The Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and range() is one of them.

Example: Python program to print numbers from 12 to 15.

```
for i in range(12, 16):
        print(i)
```

Output:

```
12
13
14
15
```

The range() function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

Example:

```
for i in range(0, 10, 2):
        print(i)
```

Calling range(0, 10, 2) will count from zero to eight by intervals of two.

Output:

```
02468
```

The range() function is flexible in the sequence of numbers it produces for for loops. *For* example, we can even use a negative number for the step argument to make the for loop count down instead of up.

Example:

```
for i in range(5, -1, -1):
        print(i)
```

Running a for loop to print i with range(5, -1, -1) should print from five down to zero.

Output:

```
543210
```

Q) Briefly explain about Importing Modules used in Python programming. Give example.

All Python programs can call a basic set of functions called built-in functions, including the print(), input(), and len() functions. Python also comes with a set of modules called the standard library.

Each module is a Python program that contains a related group of functions that can be embedded in our programs. For example, the math module has mathematics-related functions, the random module has random number–related functions, and so on.

Before we can use the functions in a module, we must import the module with an import statement. In code, an import statement consists of the following:

Syntax:

```
import module_name
```

where

- Import is the keyword.
- The module_name is the name of the module.
- Optionally, more module names, as long as they are separated by commas.

Once we import a module, we can use all the cool functions of that module.

Example:

```
import random
for i in range(5):
        print(random.randint(1, 10))
```

Output:

```
41841
```

The random.randint() function call evaluates to a random integer value between the two integers that we pass it. Since randint() is in the random module, we must first type **random.** in front of the function name to tell Python to look for this function inside the random module.

Another example of an import statement that imports four different modules:

```
import random, sys, os, math
```

from import Statements

An alternative form of the import statement is composed of the from keyword, followed by the module name, the import keyword, and a star; for example, from random import *. With this form of import statement, calls to functions in random will not need the random.prefix. However, using the full name makes for more readable code, so it is better to use the normal form of the import statement.

Q) Explain how to terminate the program without completing its execution?

OR

Briefly explain about sys.exit() function. Give example.

We can cause the program to terminate, or exit, by calling the sys.exit() function. Since this function is in the sys module, we have to import sys before your program can use it.

Example:

```
import sys
while True:
        print('Type exit to exit.')
        response = input()
        if response == 'exit':
                sys.exit()
        print('You typed ' + response + '.')
```

Output:

```
Type exit to exit.
exit
You typed exit.
```

Practice Questions

Q: 1. What are the two values of the Boolean data type? How do you write them?
Q: 2. What are the three Boolean operators?
Q: 3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).
Q: 4. What do the following expressions evaluate to?

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

Q: 5. What are the six comparison operators?
Q: 6. What is the difference between the equal to operator and the assignment operator?
Q: 7. Explain what a condition is and where you would use one.
Q: 8. Identify the three blocks in this code:

```
spam = 0
if spam == 10:
        print('eggs')
if spam > 5:
        print('bacon')
else:
        print('ham')
        print('spam')
        print('spam')
```

Q: 9. Write code that prints Hello if 1 is stored in spam, prints Howdy if 2 is stored in spam, and prints Greetings! If anything else is stored in spam.
Q: 10. What can you press if your program is stuck in an infinite loop?
Q: 11. What is the difference between break and continue?
Q: 12. What is the difference between range(10), range(0, 10), and range(0, 10, 1) in a for loop?
Q: 13. Write a short program that prints the numbers 1 to 10 using a for loop. Then write an equivalent program that prints the numbers 1 to 10 using a while loop.
Q: 14. If you had a function named bacon() inside a module named spam, how would you call it after importing spam?
Extra credit: Look up the round() and abs() functions on the Internet, and find out what they do. Experiment with them in the interactive shell.

Functions

Q) What are functions? Explain Python function with parameters and return statements.

Python provides several built-in functions like print(), input() and len() fuctions, but we can also write our own functions. A function is like a mini-program within a program.

```
def hello():
        print('Howdy!')
        print('Howdy!!!')
        print('Hello there.')
hello()
hello()
hello()
```

The first line is a def statement, which defines a function named hello(). The code in the block that follows the def statement is the body of the function. This code is executed when the function is called, not when the function is first defined.

The hello() lines after the function are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Since this program calls hello() three times, the code in the hello() function is executed three times.

Output:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

def Statements with Parameters

When we call the print() or len() function, we pass in values, called arguments, by typing them between the parentheses. We can also define your own functions that accept arguments.

Example:

```
def hello(name):
        print('Hello ' + name)

hello('Alice')
hello('Bob')
```

Output:

```
Hello Alice
Hello Bob
```

The definition of the hello() function in this program has a parameter called name. A *parameter* is a variable that an argument is stored in when a function is called. The first time the hello() function is called, it's with the argument 'Alice' . The program execution enters the function, and the variable name is automatically set to 'Alice', which is what gets printed by the print() statement.

Return Values and return Statements

When we call the len() function and pass it an argument such as 'Hello', the function call evaluates to the integer value 5, which is the length of the string you passed it. In general, the value that a function call evaluates to is called the return value of the function.

When creating a function using the def statement, we can specify what the return value should be with a return statement. A return statement consists of the following:

- The return keyword
- The value or expression that the function should return

When an expression is used with a return statement, the return value is what this expression evaluates to.

For example, the following program defines a function that returns a different string depending on what number it is passed as an argument.

```
import random
def getAnswer(answerNumber):
        if answerNumber == 1:
                return 'It is certain'
        elif answerNumber == 2:
                return 'It is decidedly so'
        elif answerNumber == 3:
                return 'Yes'
        elif answerNumber == 4:
                return 'Reply hazy try again'
        elif answerNumber == 5:
                return 'Ask again later'
        elif answerNumber == 6:
                return 'Concentrate and ask again'
        elif answerNumber == 7:
                return 'My reply is no'
        elif answerNumber == 8:
                return 'Outlook not so good'
        elif answerNumber == 9:
                return 'Very doubtful'
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

The None Value

In Python there is a value called None, which represents the absence of a value. None is the only value of the NoneType data type. (Other programming languages might call this value null, nil, or undefined.) Just like the Boolean True and False values, None must be typed with a capital *N*.

This value-without-a-value can be helpful when we need to store something that won't be confused for a real value in a variable. One place where None is used is as the return value of print(). The print() function displays text on the screen, but it doesn't need to return anything in the same way len() or input() does. But since all function calls need to evaluate to a return value, print() returns None.

Example:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Q) Explain Keyword Arguments and print() used in Python programming. Give example.

Most arguments are identified by their position in the function call. For example, random.randint(1, 10) is different from random.randint(10, 1). The function call random.randint(1, 10) will return a random integer between 1 and 10, because the first argument is the low end of the range and the second argument is the high end (while random.randint(10, 1) causes an error).

However, *keyword arguments* are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters. For example, the print() function has the optional parameters end and sep to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

Example:

```
print('Hello')
print('World')
```

Output:

```
Hello
World
```

The two strings appear on separate lines because the print() function automatically adds a newline character to the end of the string it is passed. However, we can set the end keyword argument to change this to a different string.

For example,

```
print('Hello', end='')
print('World')
```

Output:

```
HelloWorld
```

The output is printed on a single line because there is no longer a new-line printed after 'Hello'. Instead, the blank string is printed. This is useful if we need to disable the newline that gets added to the end of every print() function call.
Similarly, when you pass multiple string values to print(), the function will automatically separate them with a single space.

Example:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

But we could replace the default separating string by passing the sep keyword argument.

Example:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

Q) Briefly explain about Local and Global Scope. Give example.

Parameters and variables that are assigned in a called function are said to exist in that function’s local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. A variable must be one or the other; it cannot be both local and global.

Consider a scope as a container for variables. When a scope is destroyed, all the values stored in the scope’s variables are forgotten. There is only one global scope, and it is created when our program begins. When our program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran our program, the variables would remember their values from the last time we ran it.

A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time we call this function, the local variables will not remember the values stored in them from the last time the function was called.

Scopes matter for several reasons:

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function’s local scope cannot use variables in any other local scope.
- We can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when we run it:

```
def spam():
        eggs = 31337
spam()
print(eggs)
```

Output:

```
Traceback (most recent call last):
File "C:/test3784.py", line 4, in <module>
print(eggs)
NameError: name 'eggs' is not defined
```

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function.

```
def spam():
        eggs = 99
        bacon()
        print(eggs)

def bacon():
        ham = 101
        eggs = 0
        spam()
```

Output:

```
99
```

Global Variables Can Be Read from a Local Scope

```
def spam():
        print(eggs)
eggs = 42
spam()
print(eggs)
```

Output:

```
42
```

Local and Global Variables with the Same Name

Avoid using local variables that have the same name as a global variable or another local variable.

Example:

```
def spam():
        eggs = 'spam local'
        print(eggs) # prints 'spam local'
def bacon():
        eggs = 'bacon local'
        print(eggs) # prints 'bacon local'
        spam()
        print(eggs) # prints 'bacon local'

eggs = 'global'
bacon()
print(eggs) # prints 'global'
```

Output:

```
bacon local
spam local
bacon local
global
```

The global Statement

If we need to modify a global variable from within a function, use the global statement.

Example:

```
def spam():
        global eggs
        eggs = 'spam'
eggs = 'global'
spam()
print(eggs)
```

Output:

```
spam
```

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

Example:

```
def spam():
        global eggs
        eggs = 'spam' # this is the global
def bacon():
        eggs = 'bacon' # this is a local
def ham():
        print(eggs) # this is the global
eggs = 42 # this is the global
spam()
print(eggs)
```

Output:

```
spam
```

Q) Briefly explain about Exception Handling used in Python programming. Give example.

When getting an error, or *exception*, in our Python program means the entire program will crash. We don’t want this to happen in real-world programs. Instead, we want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a “divide-by-zero” error.

```
def spam(divideBy):
        return 42 / divideBy
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output:

```
21.0
3.5
Traceback (most recent call last):
File "C:/zeroDivide.py", line 6, in <module>
print(spam(0))
File "C:/zeroDivide.py", line 2, in spam
return 42 / divideBy
ZeroDivisionError: division by zero
```

A ZeroDivisionError happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the return statement in spam() is causing an error.

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens.

We can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
        try:
                return 42 / divideBy
        except ZeroDivisionError:
                print('Error: Invalid argument.')
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

```
def spam(divideBy):
        return 42 / divideBy
try:
        print(spam(2))
        print(spam(12))
        print(spam(0))
        print(spam(1))
except ZeroDivisionError:
        print('Error: Invalid argument.')
```

Output:

```
21.0
3.5
Error: Invalid argument.
```

The reason print(spam(1)) is never executed is because once the execution jumps to the code in the except clause, it does not return to the try clause. Instead, it just continues moving down as normal.

Q) Write a program in Python to Guess the Number.

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')
# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
        print('Take a guess.')
        guess = int(input())
        if guess < secretNumber:
                print('Your guess is too low.')
        elif guess > secretNumber:
                print('Your guess is too high.')
        else:
                break # This condition is the correct guess!
        if guess == secretNumber:
                print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
        else:
                print('Nope. The number I was thinking of was ' + str(secretNumber))
```

Output:

```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
Your guess is too low.
```

```
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

Practice Questions

Q: 1. Why are functions advantageous to have in your programs?

Q: 2. When does the code in a function execute: when the function is defined or when the function is called?

Q: 3. What statement creates a function?

Q: 4. What is the difference between a function and a function call?

Q: 5. How many global scopes are there in a Python program? How many local scopes?

Q: 6. What happens to variables in a local scope when the function call returns?

Q: 7. What is a return value? Can a return value be part of an expression?

Q: 8. If a function does not have a return statement, what is the return value of a call to that function?

Q: 9. How can you force a variable in a function to refer to the global variable?

Q: 10. What is the data type of None?

Q: 11. What does the import areallyourpetsnamederic statement do?

Q: 12. If you had a function named bacon() in a module named spam, how would you call it after importing spam?

Q: 13. How can you prevent a program from crashing when it gets an error?

Q: 14. What goes in the try clause? What goes in the except clause?