What is Python? Executive Summary

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

ENTERING EXPRESSIONS INTO THE INTERACTIVE SHELL

Click the **New** button and save an empty file as *blank.py*. When you run this blank file by clicking the **Run** button or pressing F5, Enter **2 + 2** at the prompt to have Python do some simple math.

```
>>> 2 + 2
4
>>
```

In Python, 2 + 2 is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as 2) and *operators* (such as +), and they can always *evaluate* (that is, reduce) down to a single value, In the previous example, 2 + 2 is evaluated down to a single value, 4. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

Precedence

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).

```
>>> 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
```

Table 1-1: Math Operators from Highest to Lowest Precedence

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

Python won't be able to understand it and will display a `SyntaxError` error message, as shown here:

```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

THE INTEGER, FLOATING-POINT, AND STRING DATA TYPES

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 Table. The values `-2` and `30`, for example, 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.

Table

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'`

STRING CONCATENATION AND REPLICATION

`+` 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 you try to use the `+` operator on a string and an integer value

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Alice' + 42
TypeError: can only concatenate str (not "int") to str
```

The error message `can only concatenate str (not "int") to str` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string because Python cannot do this automatically. (Converting data types will be explained in "Dissecting Your Program" on page 13 when we talk about the `str()`, `int()`, and `float()` functions.)

The `*` operator multiplies 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. Enter a string multiplied by a number into the interactive shell to see this in action.

```
>>> '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, like the following:

```
>>> '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'
```

VARIABLES

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

You'll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, then a variable named `spam` will have the integer value `42` stored in it.

```
❶ >>> spam = 40
   >>> spam
   40
   >>> eggs = 2
❷ >>> spam + eggs
   42
   >>> spam + eggs + spam
   82
❸ >>> spam = spam + 2
   >>> spam
   42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is

why `spam` evaluated to `42` instead of `40` at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Variable Names

A Python variable is a reserved memory location to store values. In other words, a variable in a python program gives data to the computer for processing, variable it obeys the following three rules.

Valid variable names	Invalid variable names
`current_balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`account4`	`4account` (can't begin with a number)
`_42`	`42` (can't begin with a number)
`TOTAL_SUM`	`TOTAL_$UM` (special characters like `$` are not allowed)
`hello`	`'hello'` (special characters like `'` are not allowed)

FLOW CONTROL

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, 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.

BOOLEAN VALUES

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`.

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

Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use `True` and `False` for variable names ❸, Python will give you an error message.

COMPARISON OPERATORS

Comparison operators, also called *relational operators*, compare two values and evaluate down to a single Boolean value. Table shows the lists the comparison operators.

Table

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 you give them

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

The operator, `==` (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
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression `42 == '42'` ❶ evaluates to `False` because Python considers the integer `42` to be 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
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

THE DIFFERENCE BETWEEN THE == AND = OPERATORS

- The == operator (equal to) asks whether two values are the same as each other.
- The = operator (assignment) puts the value on the right into the variable on the left.

BOOLEAN OPERATORS

The three Boolean operators (`and`, `or`, and `not`) are used to compare Boolean values.

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 is 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

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`

Table : The `not` Operator's Truth Table

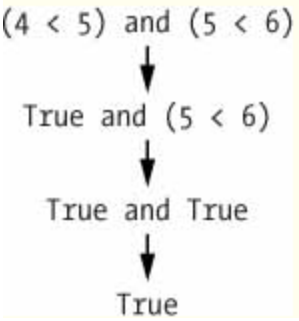
Expression	Evaluates to . . .
`not True`	`False`
`not False`	`True`

MIXING BOOLEAN AND COMPARISON 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.

```
>>> (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.



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
```

ELEMENTS OF FLOW CONTROL

Conditions

The Boolean expressions you've seen so far could all be 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*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

- Blocks begin when the indentation increases.
- Blocks can contain other blocks.
- Blocks end when the indentation decreases to zero or to a containing block's indentation.

Example:

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

The first block of code ❶ starts at the line `print('Hello, Mary')` and contains all the lines after it.

FLOW CONTROL STATEMENTS

A program's control flow is the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

if Statements

The most common type of flow control statement is the `if` statement. An `if` statement is executed if the condition is true else false.

In Python, an `if` statement consists of the following:

- The `if` keyword
- A condition (that 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)

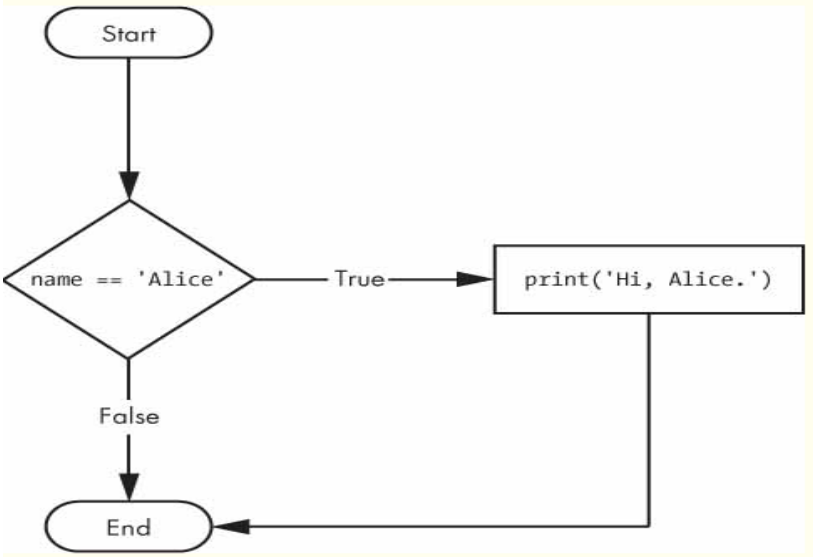


Figure : The flowchart for an *if* statement

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 plain English, an `else` statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An `else` statement doesn't have a condition, and in code, an `else` statement always consists of the following:

- The `else` keyword
- A colon
- Starting on the next line, an indented block of code (called the `else` clause)

The example below uses an `else` statement to offer a different greeting if the person's name isn't Alice.

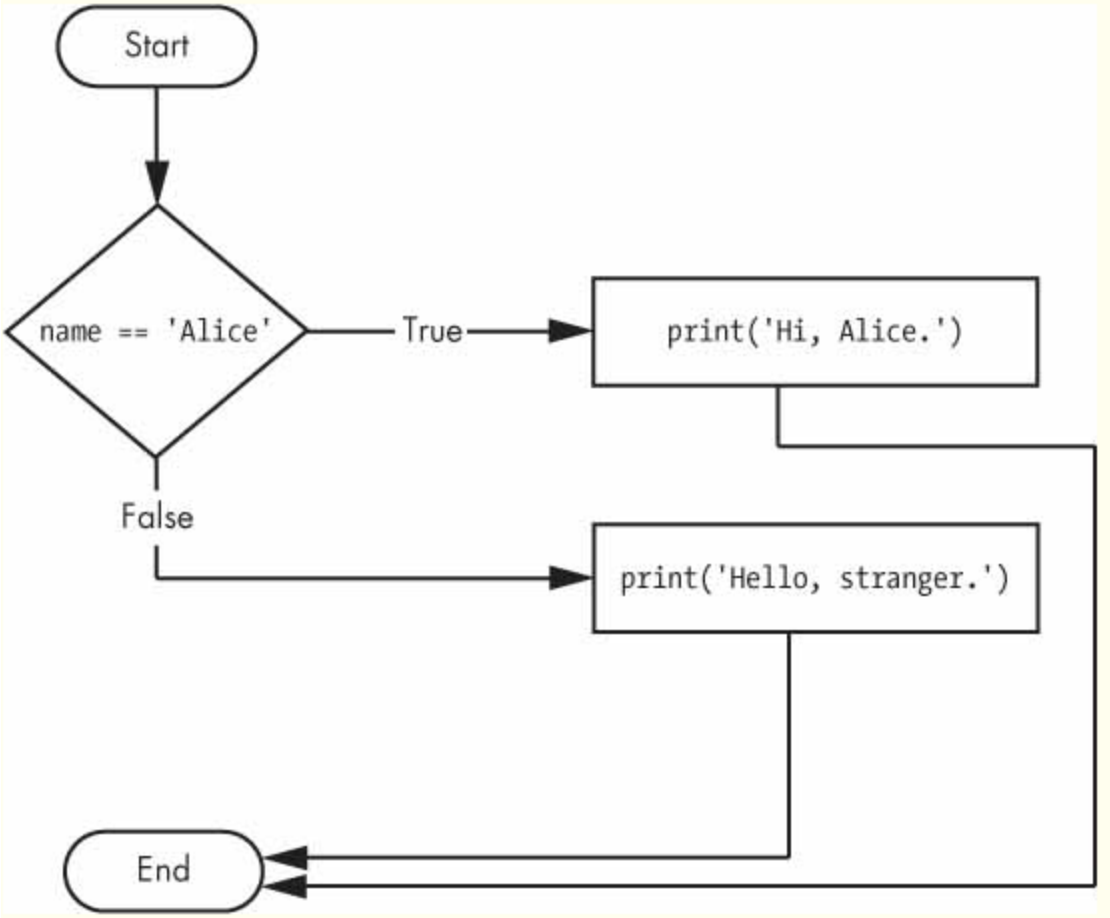


Figure : The flowchart for an *else* statement

elif Statements

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 all of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

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

Starting on the next line, an indented **player_age = 12**

```
if player_age >= 18:
   print("You could be in college.")
elif player_age >= 13:
   print("You can also attend iD Academies!")
elif player_age >= 7:
   print("You can attend iD Tech Camps!")
else:
   print("You're young.")
```

- **if player_age > 18** : #Happens if the age is greater than 18.
- **elif player_age > 13** : #Only happens if the age is not greater than 18 and greater than 13.

An else statement doesn't look for a specific condition. Else statements occur after an if or elif statement in your code.

- **if player_age > 18** : #Happens if the age is greater than 18.
- **else** : #Would happen for any other age not specified above.

Every if/elif/else block must begin with one regular if statement and end with a single else statement, but you can have as many elif statements in the middle as you want!

while Loop Statements

You can make a block of code execute over and over again using a `while` statement. The code in a `while` clause will be executed as long as the `while` statement's condition is `True`. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition (that 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)

A `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if` statement. But at the end of a `while` clause, the program execution jumps back to the start of the `while` statement. The `while` clause is often called the *while loop* or just the *loop*.

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

Here is the code with a `while` statement:

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

These statements are similar—both `if` and `while` check the value of `spam`, and if it's less than 5, they print a message. But when you run these two code snippets, something very different happens for each one. For the `if` statement, the output is simply `"Hello, world."`. But for the `while` statement, it's `"Hello, world."` repeated five times!

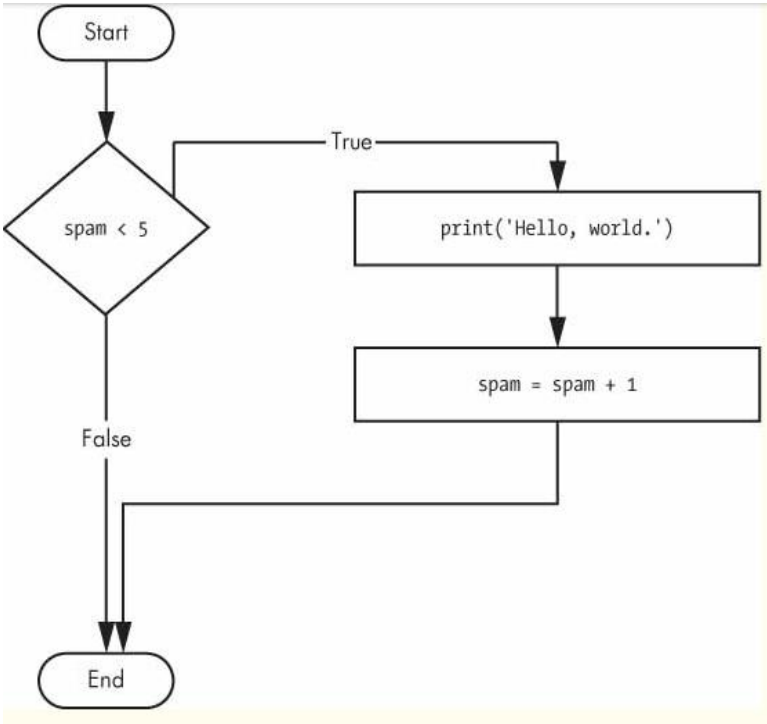


Figure : The flowchart for the *if* statement code

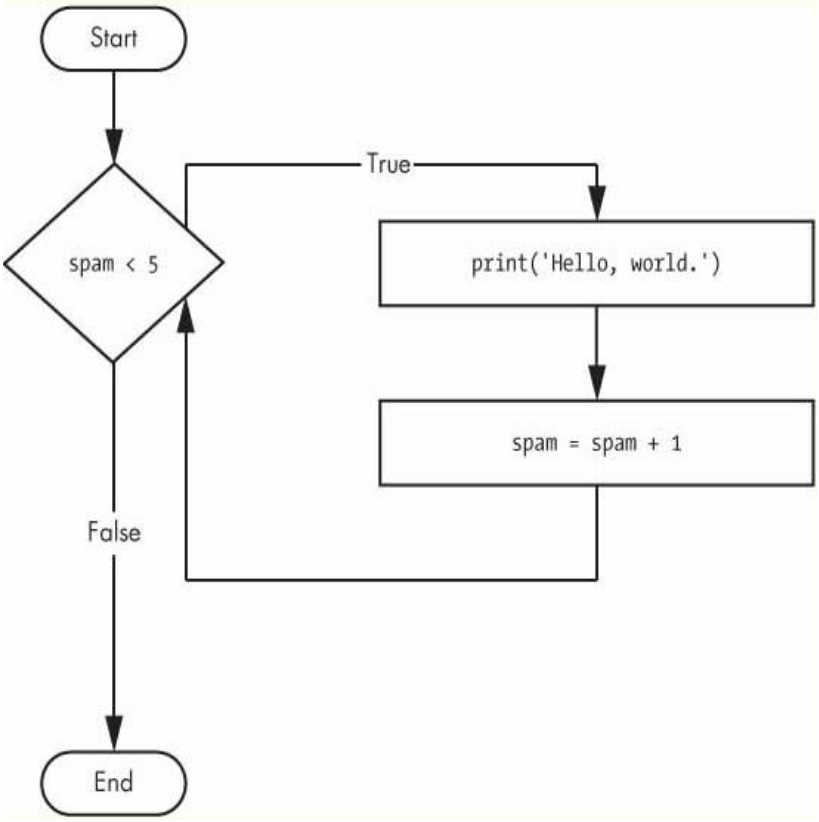


Figure : The flowchart for the *`while`* statement code

The code with the `if` statement checks the condition, and it prints `Hello, world.` only once if that condition is true. The code with the `while` loop, on the other hand, will print it five times. The loop stops after five prints because the integer in `spam` increases by one at the end of each loop iteration, which means that the loop will execute five times before `spam < 5` is `False`.

In the `while` loop, the condition is always checked at the start of each *iteration*, If the condition is `True`, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be `False`, the `while` clause is skipped.

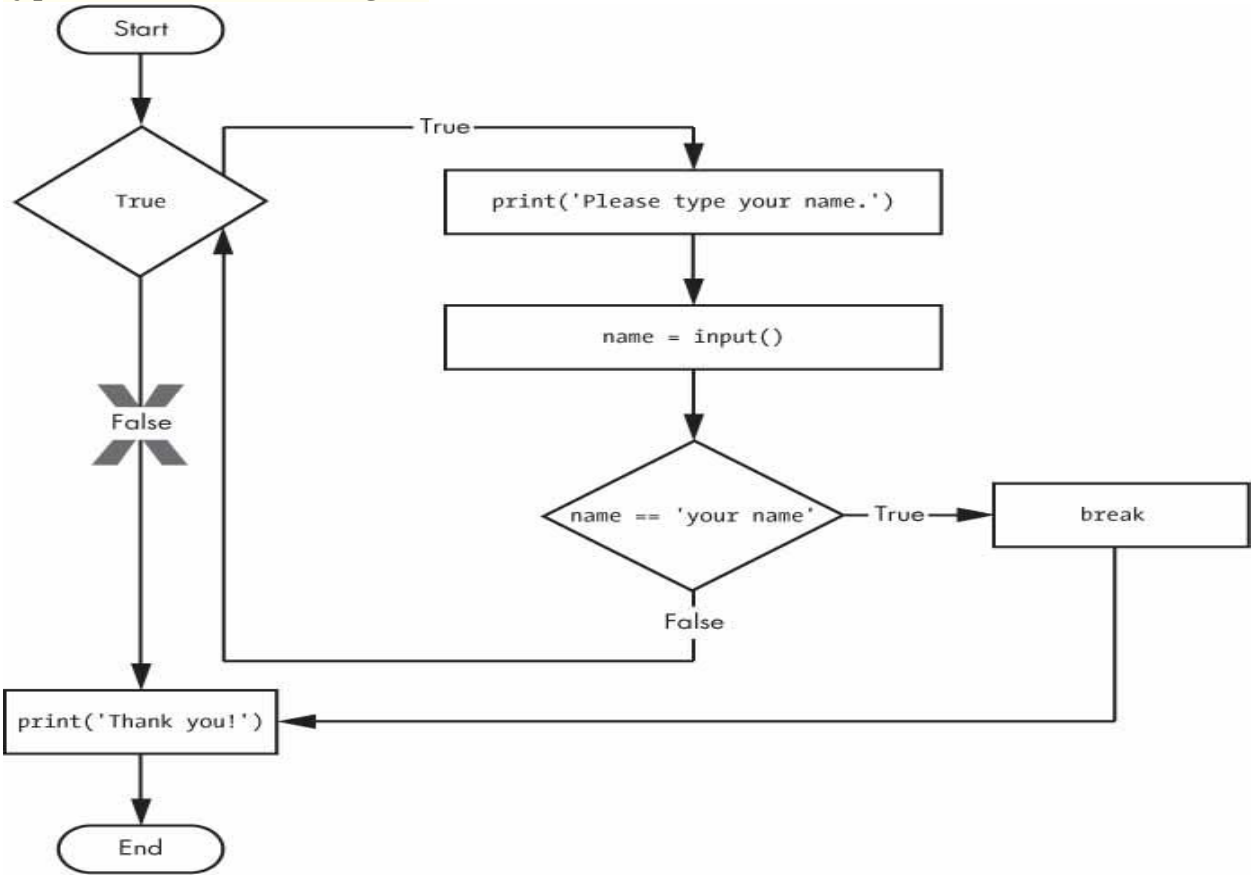
break Statements:

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.

```
❶ while True:
      print('Please type your name.')
   ❷ name = input()
   ❸ if name == 'your name':
      ❹ break
❺ print('Thank you!')
```

The first line ❶ creates an *infinite loop*; it is a `while` loop whose condition is always `True`. After the program execution enters this loop, it will exit the loop only when a `break` statement is executed, this program asks the user to enter `your name` ❷. Now, however, while the execution is still inside the `while` loop, an `if` statement checks ❸ whether `name` is equal to `'your name'`. If this condition is `True`, the `break` statement is run ❹, and the execution moves out of the loop to `print('Thank you!')` ❺. Otherwise, the `if` statement's clause that contains the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement ❶ to recheck the condition. Since this condition is merely the `True` Boolean value, the execution enters the loop to ask the user to

type `you're your name` again



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 reevaluates the loop's condition

Example:

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

If the user enters any name besides `Joe` ❶, the `continue` statement ❷ causes the program execution to jump back to the start of the loop. When the program reevaluates the condition, the execution will always enter the loop, since the condition is simply the value `True`. Once the user makes it past that `if` statement, they are asked for a password ❸. If the password entered is `swordfish`, then the `break` statement ❹ is run, and the execution jumps out of the `while` loop to print `Access granted` ❺. Otherwise, the execution continues to the end of the `while` loop, where it then jumps back to the start of the loop.

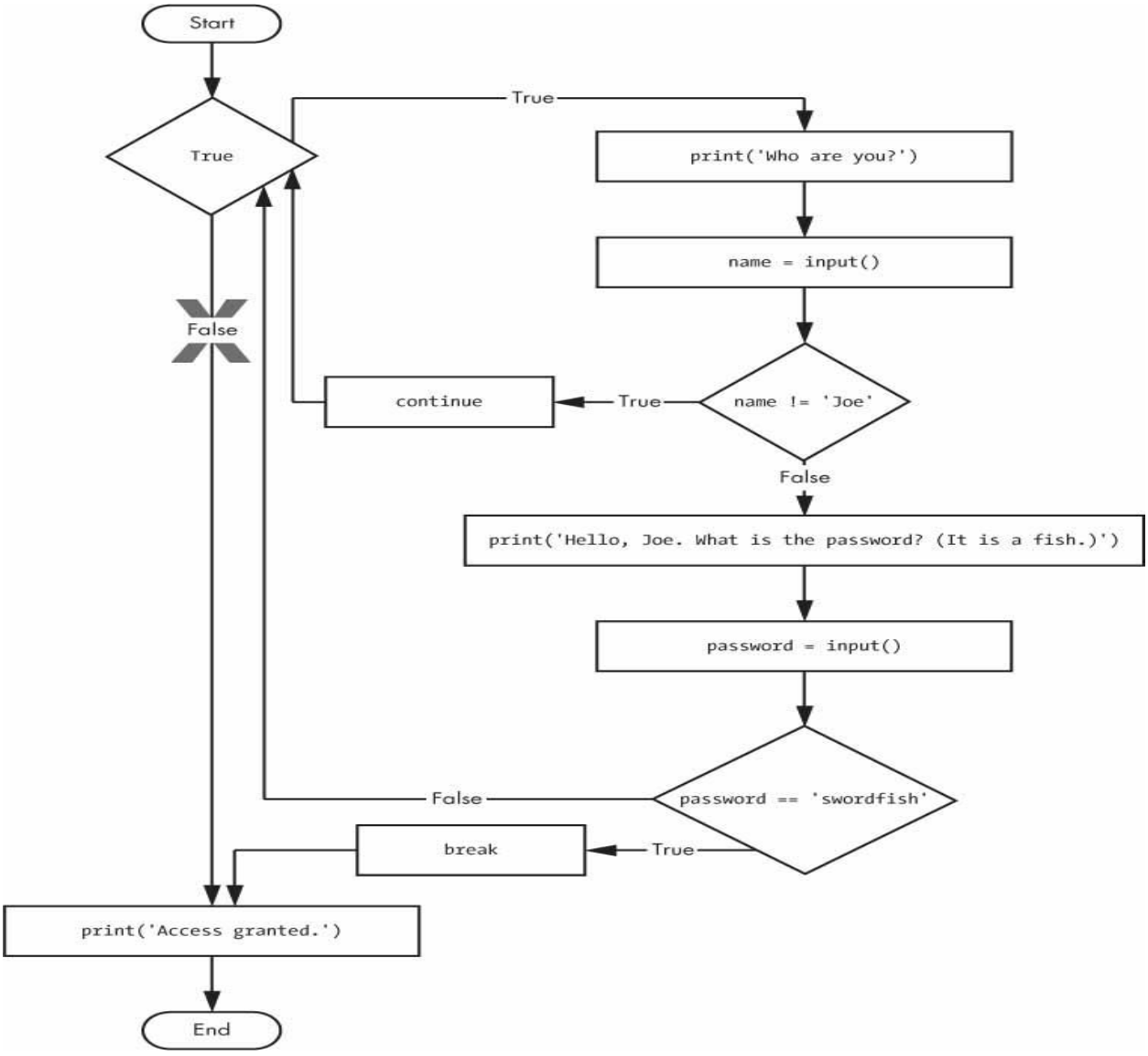


Fig :Flowchart of continue statement

Local & Global variables

Local variables:- local variables are those which are defined inside a function and their scope is limited to that function only.In other words, we can say that local variables are accessible only inside the function in which it was initialized whereas the global variables are accessible throughout the program and inside every function.

Ex:

```
def f():

# local variable
    s = "I love Geeksforgeeks"
    print(s)
f()
```

Output:

I love Geeksforgeeks

- **Can a local variable be used outside a function?**

If we will try to use this local variable outside the function then let's see what will happen.

```
def f():

s = "I love Geeksforgeeks"
    print("Inside Function:", s)
f()
print(s)
```

Output:

NameError: name 's' is not defined

Global Variables:-

These are those which are defined outside any function and which are accessible throughout the program, i.e., inside and outside of every function

Ex:

```
def f():
    print("Inside Function", s)
# Global scope
s = "I love Geeksforgeeks"
f()
print("Outside Function", s)
```

Output

Inside Function I love Geeksforgeeks

Outside Function I love Geeksforgeeks

FUNCTIONS

- A function is a block of code which only runs when it is called.You can pass data, known as parameters, into a function.A function can return data as a result. A *function* is like a miniprogram within a program.
  Ex:

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

❸ hello()
  hello()
  hello()
```

The first line is a def statement ❶, which defines a function named hello(). The code in the block that follows the def statement ❷ is the body of the function. This code is executed when the function is called, The hello() lines after the function ❸ 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. Since this program calls hello() three times, the code in the hello() function is executed three times. When you run this program, the output looks like this:

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

DEF STATEMENTS WITH PARAMETERS

When you call the `print()` or `len()` function, you pass them values, called *arguments*, by typing them between the parentheses.

Ex:

```
❶ def hello(name):
    ❷ print('Hello, ' + name)

❸ hello('Alice')
  hello('Bob')
```

Output:

Hello, Alice
Hello, Bob

The definition of the `hello()` function in this program has a parameter called `name` ❶. *Parameters* are variables that contain arguments. When a function is called with arguments, the arguments are stored in the parameters. The first time the `hello()` function is called, it is passed the argument `'Alice'` ❸. The program execution enters the function, and the parameter `name` is automatically set to `'Alice'`, which is what gets printed by the `print()` statement ❷.

Define, Call, Pass, Argument, Parameter

Ex:

```
❶ def sayHello(name):
      print('Hello, ' + name)
❷ sayHello('Al')
```

To *define* a function is to create it, just like an assignment statement like spam = 42 creates the spam variable. The def statement defines the sayHello() function ❶. The sayHello('Al') line ❷ *calls* the now-created function, sending the execution to the top of the function's code. This function call is also known as *passing* the string value 'Al' to the function. A value being passed to a function in a function call is an *argument*. The argument 'Al' is assigned to a local variable named name. Variables that have arguments assigned to them are *parameters*.

RETURN VALUES AND RETURN STATEMENTS

A **return statement** is used to end the execution of the function call and "returns" the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned. A **return statement** is overall used to invoke a function so that the passed statements can be executed.

Note: *Return statement can not be used outside the function.*

Syntax:

```
def fun():
    statements
    .
    .
    return [expression]
```

```
Ex:def cube(x):
    r=x**3
    return r
```

Return values:

To let a function return a value, use the `return` statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Output:

```
15
25
45
```

KEYWORD ARGUMENTS AND THE PRINT() FUNCTION

Arguments: The terms parameter and argument can be used for the same thing: information that are passed into a function. From a function's perspective: A parameter is the variable listed inside the parentheses in the function definition. An argument is the value that are sent to the function when it is called.

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).

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.

Ex:

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

output:

```
Hello
World
```

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

Ex:

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

Output:

```
HelloWorld
```

Similarly, when you pass multiple string values to `print()`, the function will automatically separate them with a single space.

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

By passing the `sep` keyword

Ex:

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

THE CALL STACK

When a function call returns, Python removes a frame object from the top of the stack and moves the execution to the line number stored in it. Note that frame objects are always added and removed from the top of the stack and not from any other place.

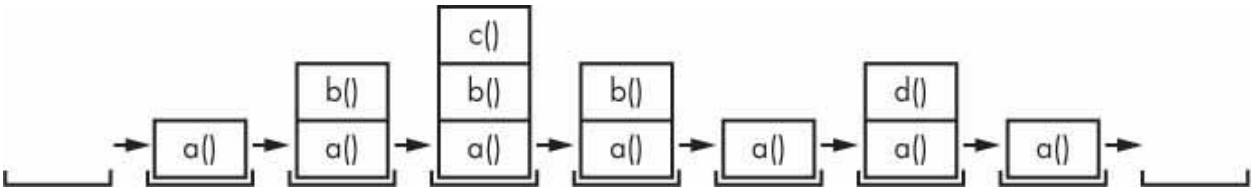


Fig:The frame objects of the call stack as *abcdCallStack.py* calls and returns from functions

The top of the call stack is which function the execution is currently in.

EXCEPTION HANDLING

Right now, getting an error, or *exception*, in your 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.

As discussed there is a chance of runtime error while doing some program.

One of the possible reasons is wrong input.

For example, consider the following code segment –

```
a=int(input("Enter a:"))

b=int(i
nput("
Enter
b:"))
c=a/b

print(c)
```

When you run the above code, one of the possible situations would be –

```
Enter a:12
Enter b:0
Traceback (most
    recent call
    last):c=a/b

ZeroDivisionError: division by zero
```

For the end-user, such type of system-generated error messages is difficult to handle.

So the code which is prone to runtime error must be executed conditionally within *try* block.

The ***try*** block contains the statements involving suspicious code and the ***except*** block contains the possible remedy (or instructions to user informing what went wrong and what could be the way toget out of it).

- If something goes wrong with the statements inside ***try*** block, the ***except*** block will be executed.
- Otherwise, the except-block will be skipped.
- Consider the example –

```
a=int(input("Enter a:"))
b=int(input("Enter b:"))

try:
c=a/b
```

```
        print(c)

except:
        print("Division by zero is not possible")
```

Output:

```
Enter a:12

Enter b:0

Division by zero is not possible
```

Handling an exception using *try* is called as ***catching*** an exception.
In general, catching an exception gives the programmer to fix the probable problem, or to try againor at least to end the program gracefully.

Question Bank

1.Need for role of precedence,Illustrate the rules of precedence
2.Explain the integer,floating point,and string data types
3.what is string concantination and repplication explain with examples
4.what are variables explain how values are storing to the variables
5.what is an expression made up of?what do all expressions do?
6.Explain with examples Local and Global variables(Scope)
7.What are comparission operators explain with examples?
8.Explain the 3 binary Boolean operators
9.Explain with examples the mixing Boolean with comparision operators
10.Explain all the flow control statements.
11.what are functions?Explain python functions with parameters and return statements
12.brief note on callstack
13.Define exception handling.Explain how exceptions are handled in python.