Module-1

Python Basics

Definition:

Python is a high level, interpreted, interactive and **Object-Oriented Scripting language**. Python is designed to be highly readable.

It uses English keywords frequently unlike other languages which use punctuation, and it has fewer syntactical constructions than other languages.

1. **Python is interpreted:** python is processed at runtime by the interpreter.
2. **Python is interactive:** you can actually sit at a python prompt and interact with the interpreter directly to write your programs.
3. **Python is object-oriented:** python supports object-oriented style or technique of programming that encapsulates code within the object.
4. **Python is a Beginner's Language:** python is a great language for the beginner-level programmers and supports the development of a wide variety of applications from simple text processing to WWW browser to games.
5. **Python is a simple**
6. **Python is a powerful.**
7. **Python is used as general purpose language.**
8. **Python is a programming language.**

Note: python is a programming language and it is **free software & open source.**

What is difference between C and Python?

The main difference between C and Python is that,

- **C** is a **structure oriented programming** language.
- While **Python** is an **object oriented programming** language.
- **Python** has fully formed ***built-in*** and ***pre-defined library functions***, but ***C has only few built-in functions***.
- The following table illustrates comparison of C and Python languages.

SNo.	Key	C Language	Python Language
1	**Definition**	C is a **general-purpose Programming language** that is **extremely popular, simple and flexible**. It is **machine-dependent, structured programming language** which is used in various applications. It uses compiler.	Python is a **general-purpose interpreted, interactive object-oriented** and high level programming language
2	**Type**	C is a **structured type programming** language and follows **imperative programming mode**l. Also, **it is statically typed.**	Python is an object-oriented type programming language and **it is dynamically typed.**
3	**Variable Declaration**	In C, **Variables have to be declared before they are used in code** further.	In Python, **no need of variable declaration** for its use is required.
4	**Compilation**	C uses compiler and hence, it is known as **compiled language.**	Python uses interpreter and hence, it is known as **interpreted language**.
5	**Functions**	C language has a **limited number of built-in functions**.	Python language has a **large number of built-in functions**.
6	**Execution**	C language code is **compiled into a machine code** or object code and the **CPU can directly execute** it.	In Python, **the code is first compiled into byte code and then it is interpreted.**

PYTHON FEATURES

Python's features include-

1. **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.
2. **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
3. **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
4. **A broad standard library:** Python's bulk of the library is very portable and **cross platform compatible** on UNIX, Windows, and Macintosh.
5. **Interactive Mode:** Python has support for an interactive mode, which allows **interactive testing and debugging of snippets of code**.
6. **Portable:** Python can run on a **wide variety of hardware platforms** and has the **same interface on all platforms**.
7. **Extendable:** You can **add low-level modules to the Python interpreter**. These modules enable programmers to add to or customize their tools to be more efficient.
8. **Databases:** Python provides **interfaces to all major commercial databases**.
9. **GUI Programming:** Python supports **GUI applications that can be created and ported to many system calls, libraries and windows systems**, such as Windows MFC, Macintosh, and the X Window system of Unix.
10. **Scalable:** Python **provides a better structure and support** for large programs than **shell scripting**.
11. **No type Declaration:** Automatically declared integer(Ex: int a, a=5)

Apart from the above-mentioned features, Python has a big list of good features. A few are listed below-

- It supports ***functional*** and ***structured programming*** methods as well as ***OOP***.
- It can be used as a ***scripting language*** or ***can be compiled to byte-code*** for building large applications.
- It provides very high-level ***dynamic data types*** and ***supports dynamic type checking.***
- It supports ***automatic garbage collection***.
- It can be **easily integrated with C, C++, COM, ActiveX, CORBA, and Java**.

Applications of Python

1. Web development
2. GUI
3. Business Application
4. Games & 3-D graphics.
5. Accessing the database.

Entering Expression into the Interactive Shell

You run the interactive shell by launching **IDLE**, which you 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**.

A window with the >>> prompt should appear; that's the interactive shell. 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 down*** (that is, **reduced**) to a **single value**.
- That means, you ***can use expressions anywhere in Python code*** that you could also use a 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
```

There are plenty of other operators you can use in Python expressions, too.

For example, Table 1-1 lists all the **math operators in Python**.

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

- 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***;
- The **+** and **-** operators are evaluated ***last*** (also ***from left to right***).
- We can use **parentheses** to **override the usual precedence** if you need to.

Enter the following expressions into the interactive shell:

```
>>> 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
```

- Python will keep evaluating parts of the expression until it becomes a single value, as shown in **Figure 1-1**.

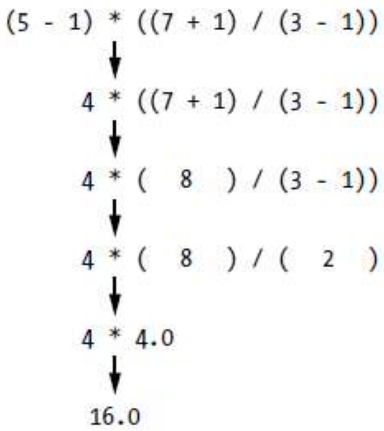


Figure 1-1: Evaluating an expression reduces it to a single value.

- These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate.

Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

- The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you type in a bad Python instruction, Python won't be able to understand it and will display a **Syntax Error** 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
```

- You *can always test to see whether an instruction works by typing it into the interactive shell*.

Literal Constants.

1. **Numbers:** → Numbers can be ***integer, floating point*** and ***Complex*** number.
2. **Stings:**→ ***Single character*** OR ***Group of characters***.

1. **Numbers**

Integer:→ 3, 6, 7, 100, 352 ,……..etc
Floating point number:→ 1.4, 5.6, 3.75, 9.25,…..etc
Complex numbers:→ a+b**i**, 5+3**i** ……..etc

Suppose we want to **store long integers**

- **Long integer** → 'l', OR 'L'.

Ex: 123456789L →suffix

Add '**L**', suffix to the number. Every long integer 'L' must be ***suffix*** otherwise it is not considered as a long integer.

A=5 --→**int**

A=5.4 ---→**float**

A=3456789L --- →**long int**

A=5+4i -→**Complex**.

2. **Strings.**
   - **Single quote** ---- ‘ ‘
   - **Double quote** ---- “ “
   - **Triple quote** ---- ‘’’ ‘’’ } Multiline String.

     The string appears in more than one line

     Ex: ‘’’ welcome to
     Python programming ’’’

Note: For Two lines string, we can use Triple quote

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 1-2**.
- **For 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.

Table 1-2: 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'`

- Python programs can also have **text values** called ***strings***, or ***strs*** (pronounced "***stirs***").
- ***Always surround your string in single quote*** (') characters (as in **'Hello'** or **'Goodbye cruel world!'**)
- So, ***Python knows where the string begins and ends***. You can even have a string with no characters in it, **' '**, called a ***blank string***.
- If you ever see the error message

  ***Syntax Error: EOL* while scanning string literal**,

  you probably forgot the final single quote character at the end of the string, such as in this **example**:

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

String Concatenation and Replication

1. 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.

Example:

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

2. 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**, ***Python will not know how to handle this,*** and it will display an ***error message.***

Example:

```
>>> "ravi" + 2
    Traceback (most recent call last):
     File "<pyshell#3>", line 1, in <module>
      "ravi" + 2
    Type Error: can only concatenate str (not "int") to str
```

The error message **Can't convert 'int'** object to **str** ***implicitly*** means that Python thought you were *trying to concatenate an integer to the string* **'Ravi'**. Your code will have to explicitly convert the integer to a string, because **Python cannot do this automatically**.

The correct format is

```
>>> "ravi" + '2'
'ravi2'
```

3. 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:

   ```
   >>> "ravi" * 5
   'raviraviraviraviravi'
   ```

   - The expression evaluates down to a single string value that repeats the string number of times equal to the integer value. ***String replication is a useful trick, but it's not used as often as string concatenation.***

4. 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 as follows.

```
>>> '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'
```

Storing Values in 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.

Assignment Statements

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.

Think of a variable as a labeled box that a value is placed in, as in Figure 1-1.

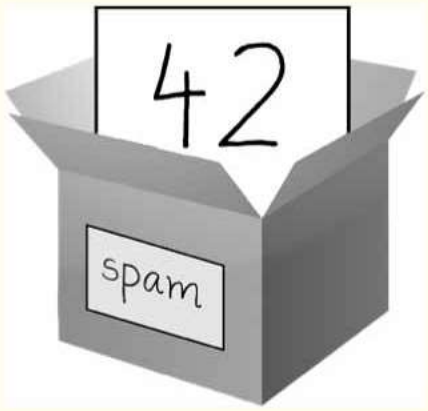


Figure 1-1: spam = 42 is like telling the program, "The variable spam now has the integer value 42 in it."

For example, enter the following into the interactive shell:

```
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'
```

Just like the box in Figure 1-2, the spam variable in this example stores '**Hello**' until you replace the string with '**Goodbye**'.

Figure 1-2: When a new value is assigned to a variable, the old one is forgotten

Variable Names

- A good variable name describes the **data it contains**.
- Though you can name your variables almost anything, Python has some naming restrictions.
- **Table 1-3** shows examples of legal variable names.
- You can name a variable anything as long as it obeys the following rules:

Rules

- **It can be only one word with no spaces.**
- Variables **should not be keywords**.
- **It can use only letters, numbers, and the underscore (_) character.**
- Variable name **should start with letter** or **underscore** (_).

  Note: (characters can be either **upper case** or **lowercase**)
- Variable name is **case sensitive**.
- **It can't begin with a number.**

Table 1-3: 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***. Though Spam is a valid variable you can use in a program, it is a ***Python convention to start your variables with a lowercase letter***.

- Sometimes, camelcase convention is used for variable names instead of underscores; That is, variables **lookLikeThis** instead of **look_like_this**.

Example 1: Add Two Numbers

```
num1 = 1.5

num2 = 6.3

sum = num1 + num2                    # Add two numbers

print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))      # Display the sum
```

Output

```
The sum of 1.5 and 6.3 is 7.8
```

Example 2: Add Two Numbers with User Input

```
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

sum = float(num1) + float(num2)                    # Add two numbers

print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))     # Display the sum
```

Output

```
Enter first number: 1.5
Enter second number: 6.3
The sum of 1.5 and 6.3 is 7.8
```

Example: 3 Python Program to Calculate the Area of a Triangle

```
a = 5
b = 6
c = 7

    # calculate the semi-perimeter
s = (a + b + c) / 2
    # calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f ' %area)
```

```
# Uncomment below to take inputs from the user
# a = float(input('Enter first side: '))
# b = float(input('Enter second side: '))
# c = float(input('Enter third side: '))
```

Output

```
The area of the triangle is 14.70
```

OR

Python Program to Calculate the Area of a Triangle with User Input

```
a =input('enter the first number:')
b = input('enter the Second  number:')
c= input('enter the Third number:')
s= (float(a) + float(b) + float(c)) / 2

# calculate the area
area = (s*(s-float(a))*(s-float(b))*(s-float(c))) ** 0.5
print('The area of the triangle is %0.2f ' %area)
```

Output

```
enter the first number:5
enter the Second  number:6
enter the Third number:7
The area of the triangle is 14.70
```

Python Program to Solve Quadratic Equation

```
# Solve the quadratic equation ax**2 + bx + c = 0
# import complex math module
import cmath
a = 1
b = 4
c = 4
# calculate the discriminant
d = (b**2) - (4*a*c)
# find two solutions
sol1 = (-b+cmath.sqrt(d))/(2*a)
sol2 = (-b-cmath.sqrt(d))/(2*a)

print('The solution are {0} and {1}'.format(sol1, sol2))
```

Output

```
The solution are (-2+0j) and (-2+0j)
```

Python Identifiers

- A Python identifier is the name used to identify a **variable, function, class, module** or **other object**.
- An identifier starts with a letter **A to Z** or **a to z** or an **underscore (_)** followed by **zero or more letters**, **underscores** and **digits (0 to 9)**.
- Python **does not allow punctuation** characters such as **@, $, and %** within identifiers.
- Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are **two different identifiers** in Python.

Here are naming conventions for Python identifiers-

- **Class names** start with an ***uppercase letter***. All other identifiers start with a lowercase letter.
- **Starting an identifier with a single leading underscore** indicates that the **identifier is private**.
- **Starting an identifier with two leading underscores** indicates a ***strong private identifier***.
- **If the identifier also ends with two trailing underscores**, the identifier is a ***language defined special name***.

Reserved Words (Key Words)

The following list shows the **Python keywords**. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python ***keywords contain lowercase letters*** only.

Python keywords					
and	**break**	**continue**	**del**	**def**	**with**
elif	**else**	**except**	**class**	**exec**	**yield**
finally	**for**	**global**	**if**	**import**	**lambda**
in	**is**	**from**	**not**	**or**	**assert**
pass	**print**	**raise**	**try**	**while**	**as**
return					

1. and→it is used for logical operator (check 2 conditions)
2. break→ Same working as 'C' Language.
3. continue→ Same working as in 'C' Language.
4. del→ it is used to delete the element in the list.
5. def→ defined user defined function.
6. with→ used in sequence of elements.
7. elif→ similar to else if keyword.
8. else→ used to control statements (same as 'C' language).
9. except→ it is used for ***exception handling*** in python.
10. class→ used in object oriented program.
11. exec→ execute the python program dynamically.
12. finally→ used in exeption handling.
13. for→ Same working as 'C' Language.
14. global→ main function in user defined function.
15. if→ decision control statement same as 'C ' language.
16. import→ used to import the package in python programming. (Mathematical function.)
17. in→ membership operator used in sequence of elements (Comparison of elements).
18. is→ identity operator (if the two variable refer same object return ***true*** else ***false***).
19. from→ import the sub package.
20. not→ logical not.
21. or→ logical OR.
22. pass→ null statements.
23. print→ output functions.
24. raise→ raising the exception.
25. return→ same as 'C' language.
26. try→ exception handling.
27. while→ same as 'C' language.
28. yield → used in exception handling.
29. lambda → to create anonymous function.
30. assert-→ used to raise the error.
31. as→ To create an alias in Python

Lines and Indentation

Python does not use braces ({}) to indicate blocks of code for ***class*** and ***function*** definitions or flow control. ***Blocks of code are denoted by line indentation***, which is rigidly enforced.

The ***number of spaces in the indentation is variable***, but ***all statements within the block must be indented by the same amount***.

For example:

```
if True:
    print ("True")
else:
  print ("False")
```

Dissecting Your Program

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a ***comment.***

```
❶ # This program says hello and asks for my name.
```

- ***Python ignores comments***, and you 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.
- ***Python also ignores the blank line after the comment***. You can add as many blank lines to your program as you want.

The print() Function

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

```
❷ 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, you 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.

Note: You can also use this function to put a blank line on the screen; just call **print()** with ***nothing in between the parentheses***.

- When writing a function name, the opening and closing parentheses at the end identify it as the name of a function.

The input() Function

The **input( )** function ***waits for the user to type some text on the keyboard*** and press ***enter*** key.

```
❸ 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.

- We can think of 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'**.

Example:

```
>>> myName=input()
Al
>>> myName
'Al'
>>>
```

Printing the User's Name

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

```
❹ print('It is good to meet you, ' + myName)
```

Remember that **expressions can always evaluate to a single value**.

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

Example:

```
>>> myName=input()
Al
>>> myName
'Al'
>>> print('it is good to meet you,'+ myName)
it is good to meet you,Al
>>>
```

- This single string value is then passed to **print(),** which prints it on the screen.

The len( ) Function

You can pass a string value (or a variable containing a string) to the **len( )** function, and the function ***evaluates to the integer value of the number of characters in that string***.

```
❺ print('The length of your name is:')
  print(len(myName))
```

Enter the following into the interactive shell to try this:

Example:

```
>>> len('hello')
5
>>> len('welcome to sjmit')
16
>>> len('')
0
>>> len(' ')
1
>>>
```

Just like those examples,

- **len(myName)** evaluates to an integer. It is then passed to **print( )** to be displayed on the screen.
- Notice that **print( )** ***allows*** you to pass it either ***integer values*** or ***string values***.

But notice the error that shows up when you type the following into the interactive shell:

```
>>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: can only concatenate str (not "int") to str
>>>
```

- The **print()** function isn't causing that error, but rather it's the ***expression*** you tried to pass to print( ) function.

We get the same error message if you type the expression into the interactive shell on its own.

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    'I am ' + 29 + ' years old.'
TypeError: can only concatenate str (not "int") to str
>>>
```

- Python gives an error because you can use the **+** operator only to ***add two integers*** together **or** ***concatenate two strings***.
- We ***can't add an integer to a string*** because this is ***ungrammatical*** in Python.

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

- If you want to concatenate an integer such as 30 with a string to pass to **print( )**, you'll need to get the value '30', which is the string form of 30.
- The **str()** function can be passed an **integer value** and **will evaluate to a string value** version of it, as follows:

Example

```
>>> str(30)
'30'
>>>
print('I am ' + str(30) + ' years old.')
I am 30 years old.
>>>
```

- Because **str(30)** evaluates to **'30'**, the expression **'I am ' + str(30) + ' years old.'** evaluates to **'I am ' + '30' + ' years old.'**, which in turn evaluates to **'I am 30 years old.'**. This is the value that is passed to the **print()** function.

- The **str(), int(),** and **float()** functions will evaluate to the ***string, integer***, and ***floating-point*** forms of the value you pass, respectively.

Converting some values of different datatypes in the interactive shell with these functions, gives the following results:

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

- The **str()** function is useful when you have an **integer or float** that **you want to concatenate to a string**. The **int()** function is also helpful if you have a number as a string value that you want to use in some mathematics.

For example,

The **input()** function ***always returns a string***, even if the user enters a number.

Enter **spam = input()** into the interactive shell and enter **100** when it waits for your text.

```
>>> spam=input()
100
>>> spam
'100'
>>>
```

The value stored inside variable **spam** is **not the integer 100** but the **string '100'**.

If you want to do math using the value in **spam,**

- Use the **int()** function to get the **integer form** of **spam** and then store this as the new value in **spam.**
- Use the **float()** function to get the **floating form** of **spam** and then store this as the new value in spam.

```
>>> spam=int(spam)
>>> spam
100
>>> spam=float(spam)
>>> spam
100.0
>>>
```

Now you should be able to treat the spam variable as an integer instead of a string.

```
>>> spam=int(spam)
>>> 100
100
>>> spam*10/5
200.0
```

```
>>> spam*10//5
200
>>>
```

```
>>> spam=float(spam)
>>> spam
100.0
>>> spam*10/5
200.0
>>> spam*10//5
200.0
>>>
```

Note that if you pass a value to **int()** that **it cannot evaluate as an integer**, Python will display an ***error message***.

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
```

- The **int()** function is also used to truncate a **floating-point number** down.

```
>>> int(7.7)
7
>>> int(7.7+1)
8
```

- The **float()** function is also used to round a floating-point number down.

```
>>> float(7)
7.0
>>> float(7+1)
8.0
```

Below, you used the **int()** and **str()** functions in the third line to get a value of the appropriate data type for the code.

```
>>> print('What is your age?') # ask for their age
What is your age?
>>> myAge = input()
35
>>> print('You will be ' + str(int(myAge) + 1) + ' in a year.')
You will be 36 in a year.
```

- The **myAge** variable contains the value returned from **input( )**.
- Because the **input( )** function ***always returns a string*** (**even if the user typed in a number**), you can use the **int(myAge)** code to ***return an integer value*** of the ***string*** in **myAge.**

- This **integer value** is then **added to 1** in the expression **int(myAge) + 1.** The **result of this addition is passed** to the **str()** function: **str(int(myAge) + 1).**
- The **string value returned** is then **concatenated** with the strings ***'You will be '*** and ***' in a year.'*** to evaluate **to one large string value**.
- This large string is finally passed to **print()** to be displayed on the screen.

Let's say the user enters the string **'4'** for **myAge**. The string '**4**' is converted to an integer, so you **can add one to it**. The result is **5**.

The **str()** function converts the **result back to a string**, so **you can concatenate** it with the second string, **'in a year.'**, to create the final message.

These evaluation steps would look something like Figure 1-4.

```
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 1-4: The evaluation steps, if 4 was stored in myAge

Flow Control

The real strength of programming isn't just running (or executing) one instruction after another. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them.

- *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.

Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

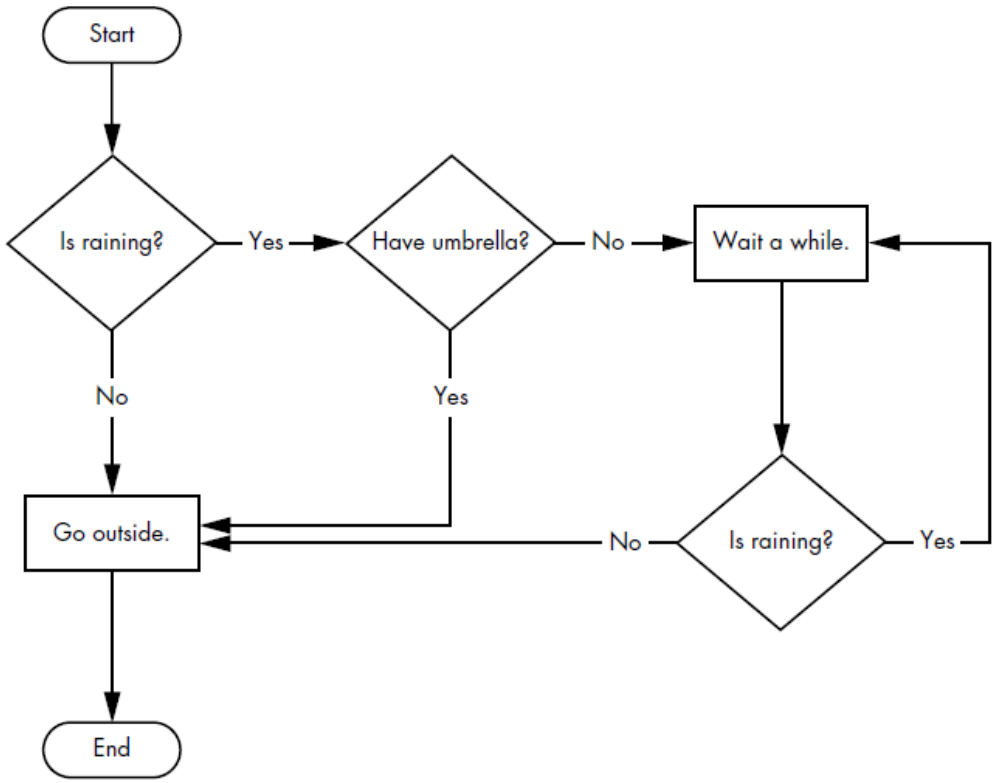


Figure 2-1: A flowchart to tell you what to do if it is raining

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.

1. Flowcharts represent these **branching** points with **diamonds**, while the **other steps a**re represented with **rectangles.**

2. The **starting and ending steps** are represented with **rounded rectangles**.
3. First we need to learn how to represent those ***yes*** and ***no*** options, and we need to understand how to write those **branching points** as Python code.
4. To that end, let's explore **Boolean values**, **comparison operators**, **and Boolean operators.**

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**.
- The Boolean value always **starts** with a **capital T** or **F** with the rest in **Lowercase**.
- Enter the following into the interactive shell.

Example:

```
>>> spam=True
>>> True
True
>>> spam
True
>>> true
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    true
NameError: name 'true' is not defined
>>> True=2+2
SyntaxError: cannot assign to True
>>>
```

1. Boolean values are used in expression & can be stored in variable
2. If you don't use the proper case OR If you try to use **True** or **False** for variable names,

   Python will give you an **error message**.

Comparison Operators

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

The below table-2-1: shows list the comparison operators.

Table 2-1: 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 you give them. Let's try some operators now, starting with == and !=.

```
>>> 50==50
True
>>> 50==98
False
>>> 50!=98
True
>>> 50!=50
False
>>>
```

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

Another example:

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

1. Note that an **integer** or **floating-point** value will always be **unequal** to a string value.
2. The expressions **42 == '42'** evaluates to **False** because Python considers the integer ***42 to be different from the string '42'.***
3. The **<, >, <=,** and **>=** operators, on the other hand, work properly only with integer and floating-point values as shown in below..

```
>>> 50<100
True
>>> 50>100
False
>>> 50<50
False
>>> Count=50
>>> Count<=50
True
>>> myAge=30
>>> myAge>=20
True
>>>
```

- We *use comparison operators to compare a variable's value to some other value*, like in the **Count <= 50** and **myAge >= 20**

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.**

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

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the **and** operator.

Table 2-2: 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

```
>>> 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 any one of two Boolean values is **True**. If both are **False,** it evaluates to **False.**

```
>>> False or True
True
>>> False or False
False
>>>
```

The **or** operator's truth table, is shown in Table 2-3

Table 2-3: 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

```
>>> True or False
True
>>> 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*. The **not** operator simply evaluates to the opposite Boolean value

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

Much like using double negatives in speech and writing, *you can nest* **not** operators Table 2-4 shows the truth table for not.

Table 2-4: The not Operator's Truth Table

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

Mixing Boolean and Comparison Operators

- The comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.
- Recall that 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.

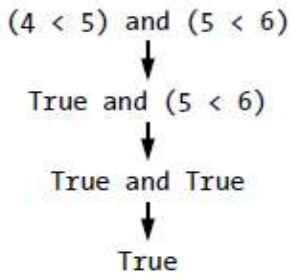
Example for some Boolean expressions that use comparison operators into the interactive shell.

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

```
>>> (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.

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



- Also 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: Python evaluates the **not** operators first, then the **and** operators, and then the **or** operators.

Elements of Flow Control

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

❖ Conditions

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**.

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.*

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program,

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

The **first block** of code 1 starts at the line **print('Hello Mary')** and contains all the lines after it. Inside this block is **another block** 2, which has only a single line in it: **print('Access Granted.'). The third block** 3 is also one line long: **print('Wrong password.')**

Flow Control Statements

Control statements enable us to specify the flow of program control, i.e, the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

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 plain English, an if statement could be read as, "If this condition is true, execute the code in the clause." In Python, an if statement consists of the following:

1. The **if** keyword.
2. A **condition** (that is, an expression that evaluates to **True** or **False**).
3. A **colon**.
4. **Starting on the *next line***, an ***indented block*** **of code** (called the **if *clause***)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Assume name was assigned some value earlier.)

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

All flow control statements end with a colon and are followed by a new block of code (the clause). This if statement's clause is the block with print('Hi, Alice.'). Figure 2-3 shows what a flowchart of this code would look like.

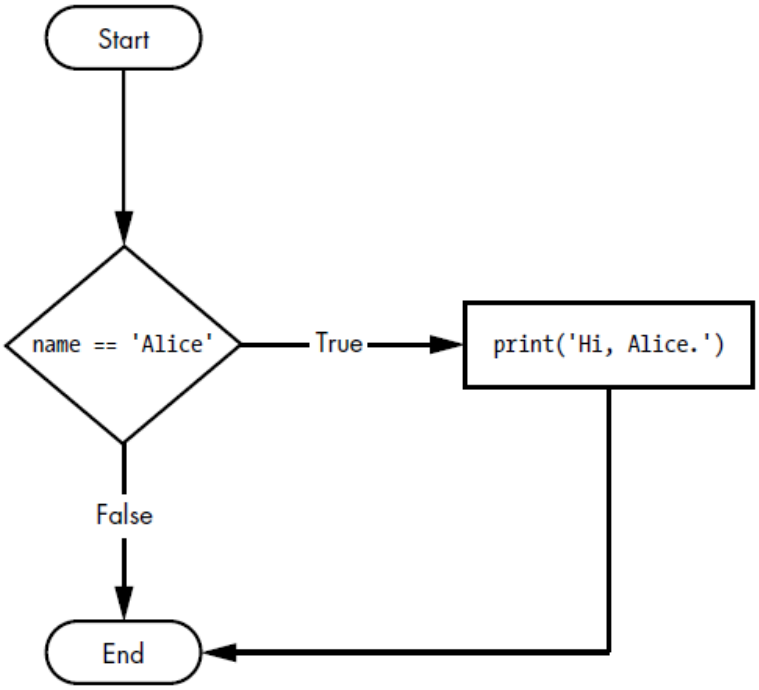


Figure 2-3: The flowchart for an if statement

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:

1. The **else** keyword.
2. A **colon**.
3. Starting on the next line, an **indented block of code** (called the ***else clause***)

Returning to the Alice example, let's look at some code that uses an else statement to offer a different greeting if the person's name isn't Alice.

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

Figure 2-4 shows what a flowchart of this code would look like.

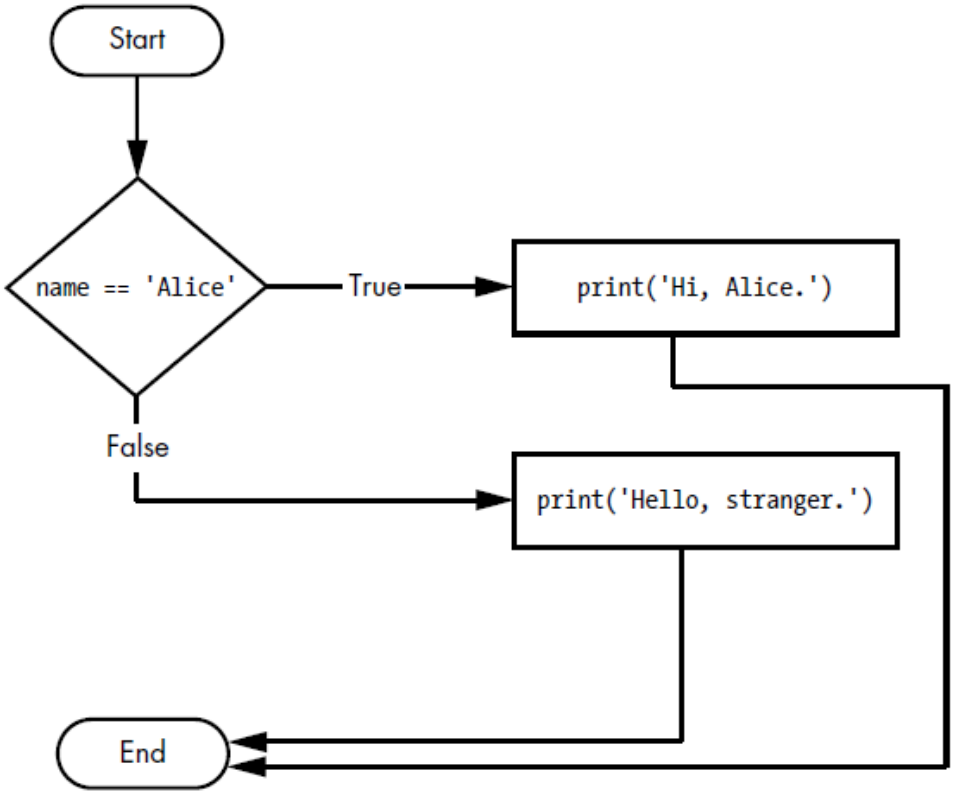


Figure 2-4: The flowchart for an else statement

elif Statements

- While only one of the **if** or **else** clauses will execute, you may have a case where you 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. In code, an **elif** statement always consists of the following:
  1. The **elif** keyword.
  2. A **condition**. (that is, an expression that evaluates to **True** or **False**)
  3. A **colon**.
  4. Starting on the next line, an **indented block of code** (called the **elif clause**)

Example:

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

This time, we check the person's age, and the program will tell them something different if they're younger than 12. You can see the flowchart for this in **Figure 2-5**.

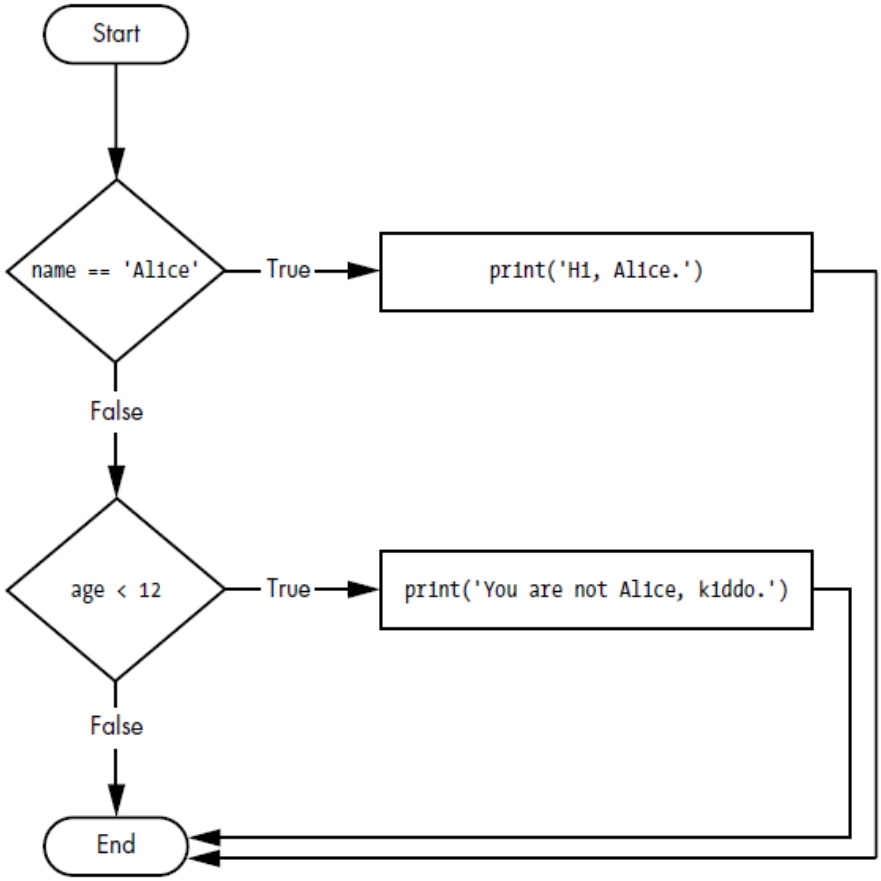


Figure 2-5: The flowchart for an elif statement

The **elif** clause executes if **age < 12** is **True** and **name == 'Alice'** is **False**. However, if both of the conditions are False, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed.

Python Code (Program)

Example Program-1

Python Program to check if a Number is Positive Negative or Zero

The user is asked to enter the number, the input number is stored in a *variable* ***number*** and then we have checked the number using ***if..elif..else*** statement.

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

# checking the number
if number < 0: print("The entered number is negative.")
elif number > 0:
    print("The entered number is positive.")
elif number == 0:
    print("Number is zero.")
else:
    print("The input is not a number")
```

OUTPUT

```
Enter number: 200
The entered number is positive.

Enter number: -25
The entered number is negative.

Enter number: 0
Number is zero.
```

Example Program-2

Python Program to Check whether Year is a Leap Year or not

In this program, user is asked to enter a year. Program checks whether the entered year is leap year or not.

```
# User enters the year
year = int(input("Enter Year: "))

# Leap Year Check
if year % 4 == 0 and year % 100 != 0:
    print(year, "is a Leap Year")
elif year % 100 == 0:
    print(year, "is not a Leap Year")
elif year % 400 ==0:
    print(year, "is a Leap Year")
else:
    print(year, "is not a Leap Year")
```

OUTPUT

```
Enter Year: 2010
2010 is not a Leap Year
>>>
==========
Enter Year: 2011
2011 is not a Leap Year
>>>
==========
Enter Year: 2012
2012 is a Leap Year
>>>
==========
Enter Year: 2013
2013 is not a Leap Year
>>>
```

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**.

In code, a ***while*** statement always consists of the following:

1. The ***while*** keyword.
2. A **condition** (that is, an expression that evaluates to True or False)
3. A **colon.**
4. Starting on the next line, an **indented block of code** (called the **while clause**)

- **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.
- While clause is often called the ***while loop*** or just the ***loop***. Let's look at an **if** statement and a **while** loop that use the same condition and take the same actions based on that ***condition***. Here is the code with an if statement:

```
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
```

For the **if** statement, the output is simply **"Hello, world.".** But for the **while** statement, it's **"Hello, world."** repeated **five times**! Take a look at the flowcharts for

these two pieces of code, Figures 2-9 and 2-10,

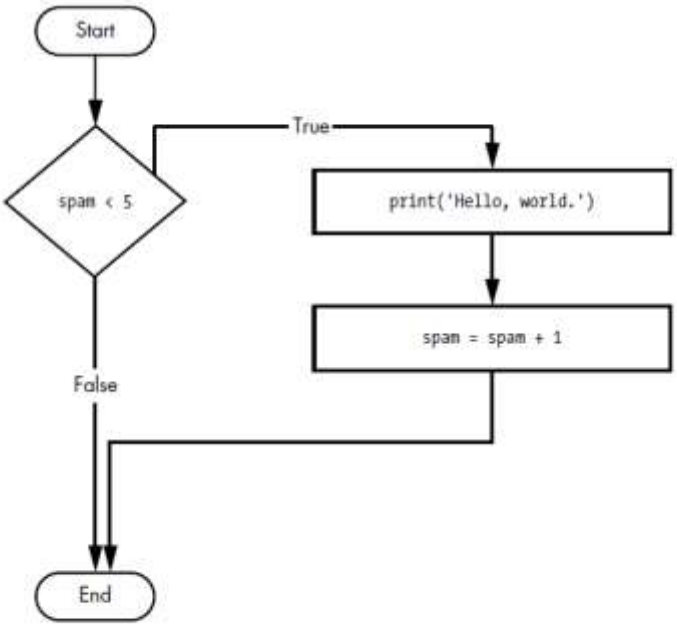


Figure 2-9: The flowchart for the if statement code

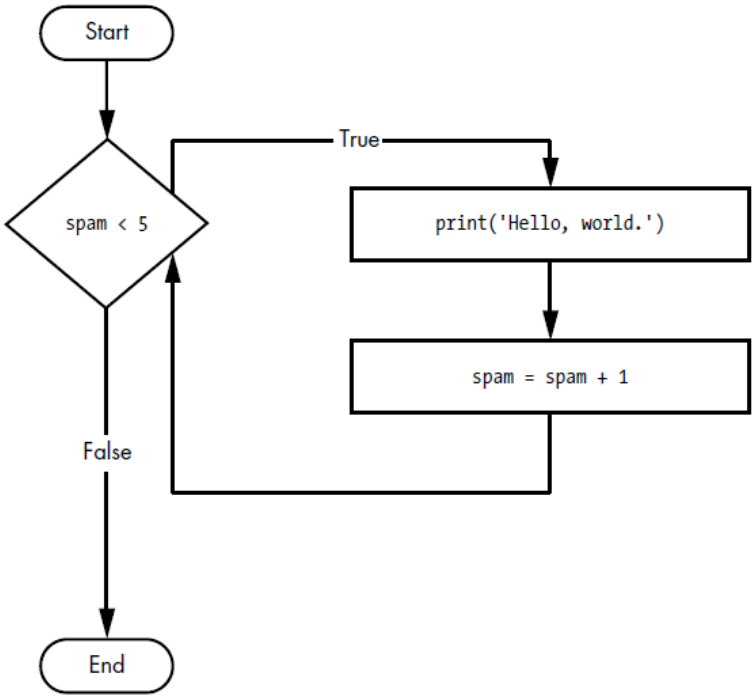


Figure 2-10: The flowchart for the while statement code

- The code with **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**.

- It stops after five prints because the integer in spam is incremented by one at the end of the each loop iteration, which means that the loop will execute five times before **spam < 5** is **False.**

- A ***while*** loop is a control statement using which the programmer can give instructions to the computer to execute a *set of statements repeatedly* as long as specified condition is **True**. Once the specified condition is **False**, control comes *out of the loop*.
- In the ***while*** loop, the *condition is always checked at the start* of each ***iteration*** (that is, each time the loop is executed).
- 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

- A ***break*** statement simply contains the ***break*** keyword. A ***break*** statement is a *jump statement* which can be used in looping statements.
- If the ***break*** is executed in a loop, the *control comes out of the loop* and the statement following the loop will be executed.
- The ***break*** statement is used mainly to terminate the loop when specific condition is reached.

For Example, Observe the following code-1

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

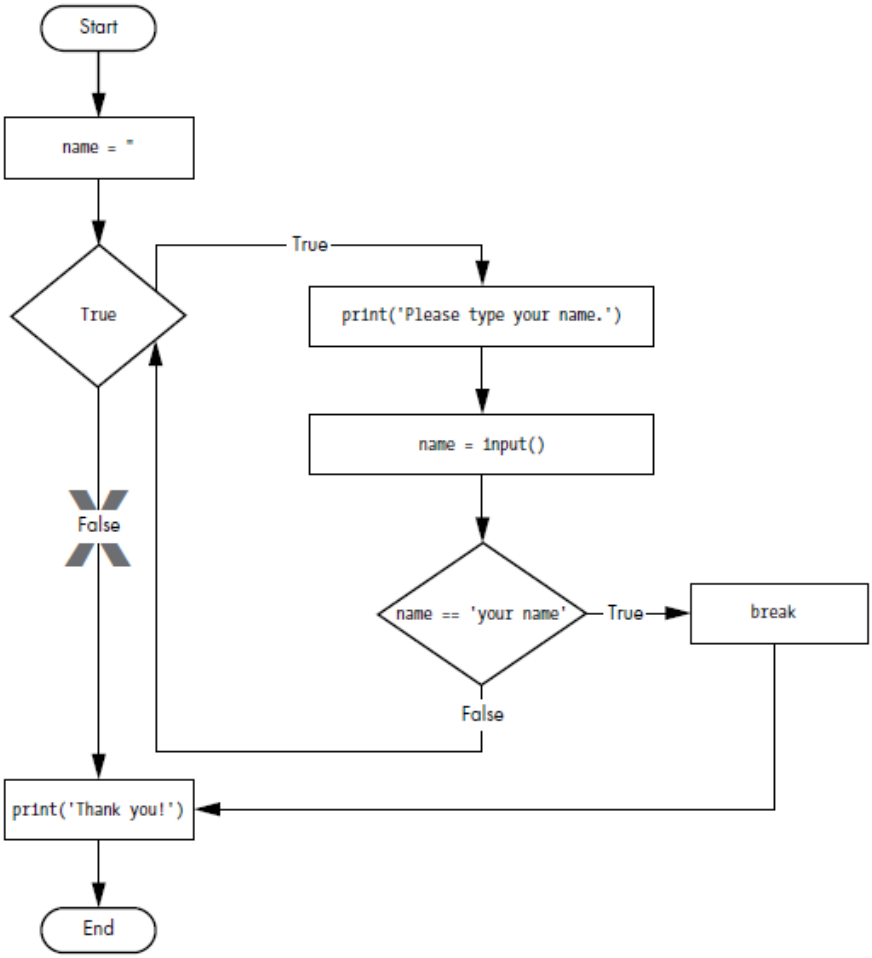
1. The first line creates an ***infinite loop***; it is a while loop whose condition is always True. The program execution will always enter the loop and will exit it

only when a break statement is executed. (An infinite loop that *never* exits is a *common programming bug*.)

2. In this program, asks the user to type **your name**
3. While the execution is still inside the while loop, an **if** statement gets executed to check whether name is equal to your name.
4. If this condition is **True**, the break statement is run. And the execution moves out of the loop to **print('Thank you!')**
5. Otherwise, the **if** statement's clause with 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 1 to recheck the condition.

The flowchart for the program with an infinite loop.

Note that the X path will logically never happen because the loop condition is always **True**.



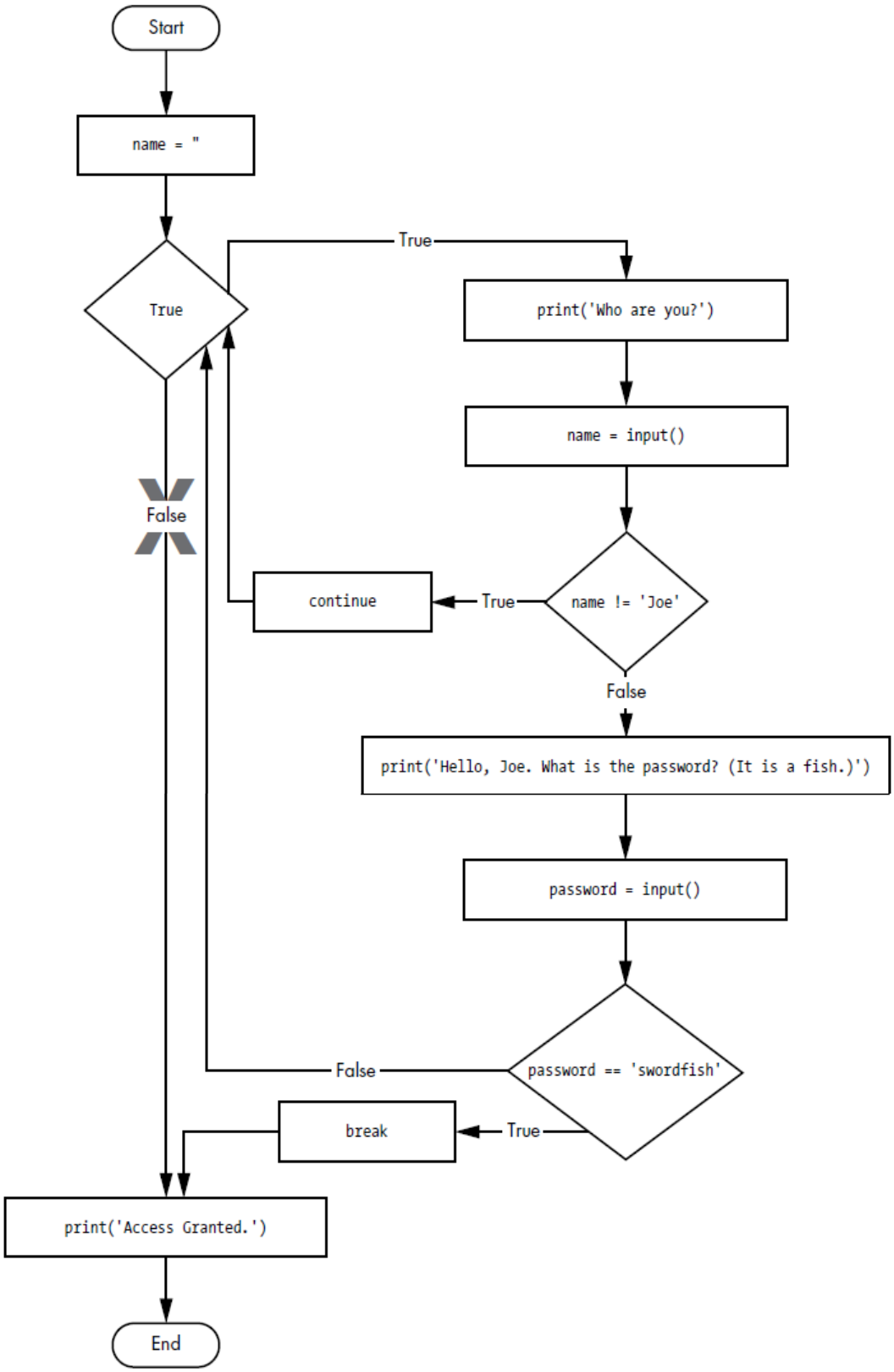
Continue Statements

- 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***.
- The ***continue*** statement is used only in the loops to terminate the current iteration and continue with remaining iterations.

Let's use continue to write a program that asks for a name and password. Enter the following code into a new file editor window

```
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 **1,** the ***continue*** statement **2** causes the program execution to jump back to the start of the loop.
- When it re-evaluates the ***condition***, the execution will always enter the loop, since the condition is simply the value **True**.
- Once they make it past that **if** statement, the user is asked for a password **3**. If the password entered is swordfish, then the ***break*** statement **4** is run, and the execution jumps out of the ***while*** loop to print Access granted **5.**
- Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop



The above a flowchart for swordfish. The X path will logically never happen because the loop condition is always **True.**

```
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.')
```

OUTPUT

Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.

---

for Loops and the *range( )* Function

- The ***while*** loop keeps looping while its condition is **True**, but what if you want to *execute* a *block of code only a certain number of times*? You can do this with a ***for*** loop statement and the ***range( )*** function.

In code, a ***for*** statement looks something like for **i** in **range(5):** and always includes the following:

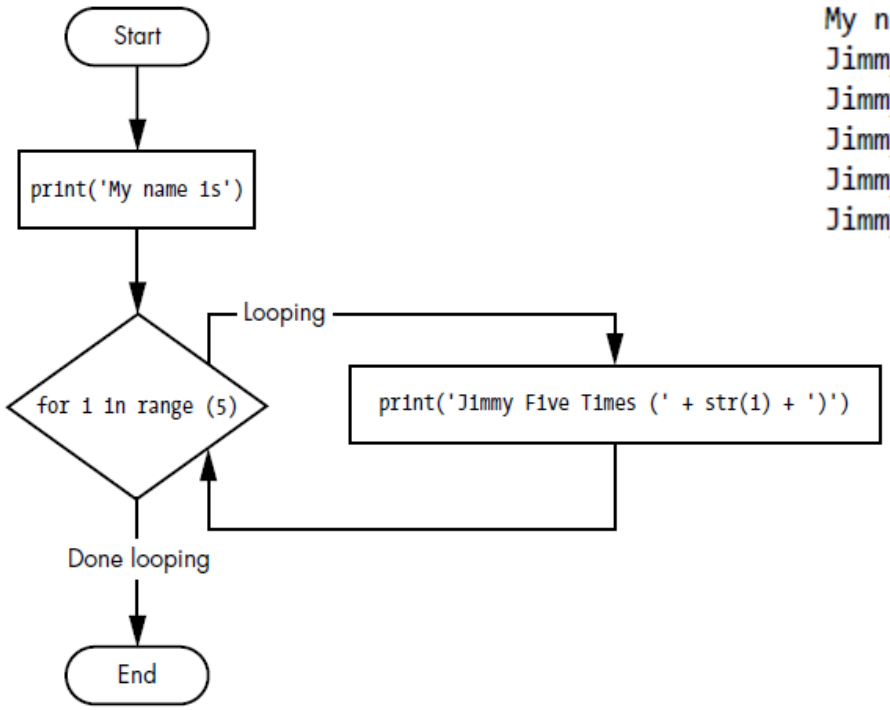
1. The for ***for*** keyword
2. A ***variable name***
3. The ***in*** keyword
4. A **call** to the ***range( )*** method with up to ***three integers passed to it***
5. A **colon**
6. Starting on the next line, an **indented block of code** (called the ***for* clause**)

Let's create a new program called *fiveTimes.py* to help you see a ***for*** loop in action.

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

1. The code in the ***for*** loop's clause is **run five times**. The first time it is run, the ***variable*** **i** is set to **0**. The **print( )** call in the clause will print Jimmy **Five Times (0)**.
2. After Python finishes an iteration through all the code inside the ***for*** loop's clause, the execution goes back to the top of the loop, and the ***for*** statement increments **i** by **one**.
3. This is why **range(5)** results in ***five iterations*** through the clause, with **i** being set to **0**, then **1**, then **2,** then **3**, and then **4.**
4. The variable **i** will ***go up to***, but ***will not include***, the integer passed to **range( ).**

Figure 2-14 shows a flowchart for the *fiveTimes.py* program.




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

Figure 2-14: The flowchart for fiveTimes.py

- When you run this program, it should ***print Jimmy Five Times*** followed by the value of **i** five times before leaving the ***for*** loop.

Importing Modules

- All Python programs can call a basic set of functions called ***built-in functions,*** including the **print(), input(),** and **len()** functions you've seen before.
- 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 your programs.

For example, the ***math*** module has mathematics related functions, the ***random*** module has random number–related functions, and so on.

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

1. The **import** keyword
2. The **name of the module**
3. Optionally, **more module names**, as long as they are separated by Commas

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

Let's give it a try with the random module, which will give us access to the **random.randint()** function. Enter this code into the file editor

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

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

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

- Here's an example of an ***import statement*** that imports ***four different modules***:

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

Ending a Program Early with sys.exit( )

- The last ***flow control*** concept to cover is how to ***terminate the program***. This always happens if the program execution reaches the bottom of the instructions.
- However, you can ***cause the program to terminate, or exit***, by calling the **sys.exit( )** function. Since this function is in the ***sys module***, you have to **import sys** before your program can use it.

Open a new file editor window and enter the following code

```
import sys

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

- Run this program in IDLE. This program has an infinite loop with no break statement inside.
- The only way this program will end is if the user enters exit, causing **sys.exit()** to be called.
- When response is equal to **exit**, the program ends. Since the response variable is set by the **input() function**, the user must enter exit in order to stop the program.

Example Program

Python Program to Check Vowel or Consonant

- In this program, user is asked to input a character. The program checks whether the entered character is equal to the **lowercase** or **uppercase *vowels***, if it is a ***vowel,*** then the program prints a message saying that the ***character is a Vowel*** else it prints that the ***character is a Consonant***.

```
# taking user input
ch = input("Enter a character: ")

if(ch=='A' or ch=='a' or ch=='E' or ch =='e' or ch=='I'
 or ch=='i' or ch=='O' or ch=='o' or ch=='U' or ch=='u'):
    print(ch, "is a Vowel")
else:
    print(ch, "is a Consonant")
```

OUTPUT

```
Enter a character: A
A is a Vowel

Enter a character: a
a is a Vowel

Enter a character: k
k is a Consonant
```

Functions

Python provides several **built-in functions** like these, but you can also write your own functions. **A *function* is like a mini-program within a program**.

Definition: In general a large program can be divided in to manageable pieces called **modules**. Each module also called a function is *self-contained small program* called ***program segment***.

To better understand how functions work. Enter this program into the file editor.

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

❸ hello()
  hello()
  hello()
```

1. The first line is a **def** statement **(1),** which defines a function named **hello().**
2. The code in the block that follows the **def** statement **(2)** is the ***body of the function***. This ***code is executed when the function is called, not when the function is first defined***.
3. The **hello()** lines after the function **(3)** 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.

When you run this program, the output looks like this:

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

Example:

Python program to find Area of a circle

```
def findArea(r):
        PI = 3.142
        return PI * (r*r);

# Driver method

print("Area is %.6f" %findArea(5));
```

OUTPUT

```
Area is 78.550000
```

Python program to find factorial of given number

```
def factorial(n):
        if n < 0:
                return 0
        elif n == 0 or n == 1:
                return 1
        else:
                fact = 1
                while(n > 1):
                        fact *= n
                        n -= 1
                return fact


num=int(input('enter the number:'))

print("Factorial of",num,"is", factorial(num))
```

OUTPUT

```
enter the number:5
Factorial of 5 is 120
enter the number:0
Factorial of 0 is 1
```

Python program to find simple interest for given principle amount, time and rate of interest.

```
def simple_interest(p,t,r):
        print('The principal is', p)
        print('The time period is', t)
        print('The rate of interest is',r)

        si = (p * t * r)/100

        print('The Simple Interest is', si)
        return si

# Driver code
simple_interest(8, 6, 8)
```

OUTPUT

```
The principal is 8
The time period is 6
The rate of interest is 8
The Simple Interest is 3.84
```

def Statements with Parameters

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

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

❸ hello('Alice')
  hello('Bob')
```

When you run this program, the output looks like this:

```
Hello Alice
Hello Bob
```

- The definition of the **hello()** function in this program has a *parameter called* **name** 1.
- 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'** **3**.
- The program execution enters the function, and the ***variable name*** is automatically set to **'Alice'**, which is what gets printed by the **print()** statement **2**.

Return Values and return Statements

- When you 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, you ***can specify what the return value should be with a return statement***.

A return statement consists of the following:

1. The **return** keyword
2. 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.

```
❶ import random

❷ def getAnswer(answerNumber):
❸     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'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

1. Python ***first imports the random module*** (1)
2. Then the **getAnswer()** function is ***defined*** (2)

3. Next, the **random.randint()** function ***is called with two arguments, 1 and 9*** (4).

It evaluates to a random integer between 1 and 9 (including 1 and 9 themselves), and this value is stored in a variable named r.

4. The **getAnswer()** function is called with **r** as the ***argument*** (5).
5. The ***program execution moves to the top*** of the **getAnswer()** function (3), and the value **r** is stored in a parameter named ***answerNumber***. Then, depending on this value in **answerNumber**,
6. The program execution returns to the line at the bottom of the program that originally called **getAnswer()** (5).
7. The ***returned string is assigned to a variable*** named **fortune**, which then gets passed to a **print()** call (6) And ***prints*** on the screen.

Note that since you can pass return values as an argument to another function call, you could shorten these three lines:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

to this single equivalent line:

```
print(getAnswer(random.randint(1, 9)))
```

A function call can be used in an expression because it evaluates to its return value.

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.***
- 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 you 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***

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

- 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:

Keyword Arguments and *print()*

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**.
- 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(keyword)*** to ***specify what should be printed at the end of its arguments*** and ***between its arguments (separating them),*** respectively.

EXAMPLE CODE

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

the output would look like this:

```
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, ***you can set the end keyword argument*** to ***change this to a different string***. **For example**, if the program were this:

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

the output would look like this:

```
HelloWorld
```

- The ***output is printed*** on ***a single line*** because ***there is no longer a newline printed*** after **'Hello'**. ***Instead, the blank string is printed***. This is useful if you 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 you could replace the default separating string by passing*** the ***sep keyword*** argument.

  Example

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

Local and Global Scope

Local Scope: ***Parameters and variables*** that are assigned in a ***called function*** are said to **exist** in that function's ***local scope***.

Global 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 either ***local*** *or* ***global***; it **cannot be both** ***local*** *and* ***global***.

- Think of 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 your* ***program begins***. When *your program terminates*, the ***global scope*** is ***destroyed***, and ***all its variables are forgotten***.
- 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***.

Scopes matter for several reasons:

1. Code in the ***global scope*** cannot use any ***local variables***.
2. However, a ***local scope*** can access ***global variables***.
3. Code in a function's ***local scope*** cannot use variables in any other ***local scope***.
4. You ***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***.

- The reason Python has ***different scopes instead of just making everything a global variable*** is so that ***when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value***.

Local Variables Cannot Be Used in the Global Scope

Global Variables

- In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Example:

```
x = "global"
def foo():
print("x inside:", x)

foo()
print("x outside:", x)
```

OUTPUT

x inside: global
x outside: global

- A variable declared ***inside the function's body*** or ***in the local scope*** is known as a ***local variable***.
- Normally, we ***declare a variable inside the function*** to create ***a local variable***.

```
def foo():
    y = "local"
    print(y)
    foo()
```

OUTPUT

Local

Global and local variables

- Here, we will show *how to use global variables* and ***local variables*** in the ***same code***.

```
x = "global "
def foo():
    global x
    y = "local"
```

OUTPUT
global global
local

```
    x = x * 2
    print(x)
    print(y)

foo()
```

Example : Global variable and Local variable with same name

```
x = 5
def foo():
    x = 10
    print("local x:", x)
foo()
print("global x:", x)
```

OUTPUT
local x: 10
global x: 5

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

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

If you run this program, the output will look like this:

NameError: name 'eggs' is not defined

The error happens because the **eggs *variable exists only in the local scope*** created when **spam()** is ***called***. *Once the program execution* **returns from spam()**, that ***local scope is destroyed***, and there is ***no longer a variable named eggs exists***. So when your program tries to run **print(eggs),** Python **gives you an error** saying that **eggs is not defined**.

Local Scopes Cannot Use Variables in Other Local Scopes

- A new ***local scope*** is created whenever a function is called, from another function.

```
   def spam():
❶      eggs = 99
❷      bacon()
❸      print(eggs)

   def bacon():
       ham = 101
❹      eggs = 0

❺ spam()
```

- When the program starts, the **spam()** function is ***called* 5,** and a ***local scope*** is created. The ***local variable* eggs** 1 is set to **99**.
- Then the **bacon()** function is ***called* 2**, and a ***second local scope*** is created. ***Multiple local scopes can exist at the same time***.
- In this ***new local scope***, the ***local variable* ham** is set to **101**, and a ***local variable* eggs**—which is *different from the one* in **spam()'s** local scope—is also created **4** and set to **0.**
- When **bacon()** ***returns***, the *local scope* for that call is ***destroyed.***
- The *program execution continues* in the **spam()** function to print the value of **eggs 3**, and ***since the local scope for the call*** to **spam()** ***still exists*** here, the **eggs** variable is set to **99**.

Global Variables Can Be Read from a Local Scope

Consider the following example:

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

- Since there is no parameter named **eggs** or any code that assigns **eggs** a value in the **spam()** ***function***.
- when **eggs** is used in **spam(),** ***Python considers it a reference to the global variable eggs***. This is why **42** is printed when the previous program is run.

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***. But technically, it's perfectly legal to do so in Python.

To see what happens..

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

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

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

- When you run this program, it **outputs** the following:

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

- There are actually ***three different variables*** in this program, but confusingly they *are all named* **eggs**.

The variables are as follows:

1. A ***variable named*** **eggs** that exists in a ***local scope*** when **spam()** is called.
2. A ***variable named*** **eggs** that exists in a ***local scope*** when **bacon()** is called.
3. A ***variable named*** **eggs** that exists in the ***global scope***.

- Since ***these three separate variables*** all have the ***same name***, it can be confusing to keep track of which one is being used at any given time.
- This is why ***you should avoid using the same variable name in different scopes***.

The global Statement

- If you need to modify a ***global variable*** from ***within a function***, use the ***global statement***. If you have a line such as ***global eggs*** at the ***top of a function***, it tells Python, "In this function, **eggs** refers to the ***global variable***, so don't create a ***local variable*** with ***this name***."

  For example,

```
def spam():
❶     global eggs
❷     eggs = 'spam'

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

The output of the above code is: spam

Because **eggs** is declared ***global*** at the top of **spam() 1**, when **eggs** is set to ***'spam'*** **2**, this assignment is done to the ***globally scoped eggs***. ***No local eggs variable*** is ***created***.

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***.

To get a better feel for these rules here's an example program

```
def spam():
❶   global eggs
    eggs = 'spam' # this is the global

def bacon():
❷   eggs = 'bacon' # this is a local


def ham():
❸   print(eggs) # this is the global

eggs = 42 # this is the global
spam()
print(eggs)
```

- In the **spam()** function, **eggs** is the **global eggs variable**, because there's a ***global statement*** for **eggs** at the beginning of the function **1**.
- In **bacon(),eggs** is a ***local variable***, because **there's an assignment statement** for it in that function **2.**
- In **ham() 3**, **eggs** is the ***global variable***, because there is **no assignment statement** or ***global statement*** for it in that function.
- The output of the above code is: spam

Exception Handling

- In python program, if getting an **error, or exception**, the entire program will crash.
- You don't want this to happen in real-world programs. Instead, ***you want the program*** to ***detect errors, handle them, and then continue to run***.

For example, consider the following program, which has a **"divide-byzero"** error.

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined ***a function called*** **spam( ),** given it a parameter, and then ***printed the value of that function*** with ***various parameters*** to see what happens.

The **output of the above code** is:

```
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***.
- You 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))
```

When ***code*** in a ***try*** **clause** causes an **error**, the **program execution immediately moves to the *code*** in the ***except*** **clause**. After running that code, the execution continues as normal.

The output of the code is as follows:

```
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.')
```

The output of the code is:

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***.

A Short Program: Guess the Number

- We will see how we can ***create a number guessing game*** using python.
- The ***Number guessing game*** is all about guessing the number randomly chosen by the computer in the given number of chances.
- A simple "**guess the number**" game is as follows:

```
# 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))
```

The output of the above code is

```
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!
```

Let's look at this code line by line, starting at the top.

```
import random
secretNumber = random.randint(1, 20)
```

1. The **program imports** the ***random module*** so that it can use the **random.randint()** function to generate a number for the user to guess. The **return value**, a random integer **between 1 and 20**, is stored in the variable ***secretNumber.***

```
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())
```

2. The *program tells the player that it has come up with a* ***secret number*** and will give the ***player six chances to guess it***. The **code** that lets the player enter a guess and checks that guess is in a **for loop** that will loop ***at most six times***.

3. The first thing that happens in the loop is that the player types in a guess. Since **input()** ***returns a string***, its ***return value is passed straight into*** **int()**, which translates the ***string into an integer*** value. This gets stored in a ***variable*** named ***guess***.

```
if guess < secretNumber:
print('Your guess is too low.')
elif guess > secretNumber:
print('Your guess is too high.')
```

4. These *few lines of code check to see whether the guess is less than or greater than* the ***secret number***. In either case, a hint is printed to the screen.

```
else:
    break                    # This condition is the correct guess!
```

5. If the ***guess*** is *neither higher nor lower* than the ***secret number***, then it must be **equal to** the ***secret number***, in which case you want the program execution to ***break*** out of the **for loop**.

```
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))
```

6. After **for loop**, the previous **if...else** statement checks whether the player has correctly guessed the number and **prints** an **appropriate message** to the screen.

7. In both cases, the **program displays** a ***variable*** that contains an **integer value** (***guessesTaken*** and ***secretNumber***).

8. Since it **must concatenate on these integer values to strings**, it passes these variables to the **str()** function, which **returns the string value form of these integers**.

9. Finally, Now these **strings can be concatenated** with the **+ operators** before finally being passed to the **print()** function call.

EXAMPLE PROGRAMS

Fibonacci number

Time Complexity: T(n) = T(n-1) + T(n-2) which is exponential.

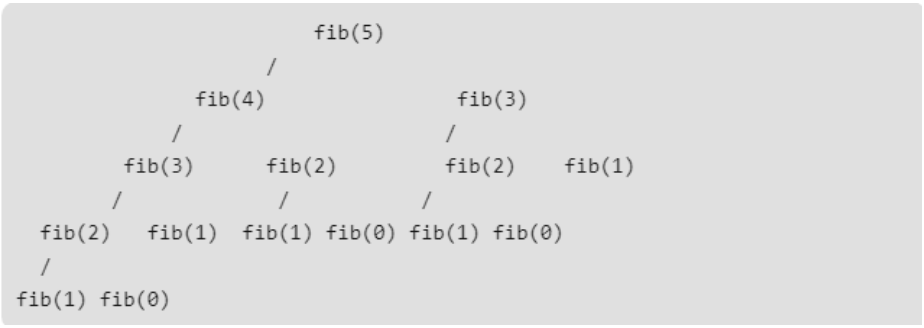
We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

Fn = Fn-1 + Fn-2

We seed values

$F_0 = 0$ and $F_1 = 1$.



1. Python program to find Function for nth Fibonacci number

```
def Fibonacci(n):
        if n<0:
                print("Incorrect input")
        # First Fibonacci number is 0
        elif n==1:
                return 0
        # Second Fibonacci number is 1
        elif n==2:
                return 1
        else:
                return Fibonacci(n-1)+Fibonacci(n-2)

# Driver Program
n=int(input("enter the number"))
print(Fibonacci(n))
```

OUTPUT

```
enter the number5
3
```

Python Program for Sum of squares of first n natural numbers

Given a positive integer N. The task is to find $1^2 + 2^2 + 3^2 + ... + N^2$.

Examples:

```
Input : N = 4
Output : 30
1^2 + 2^2 + 3^2 + 4^2
= 1 + 4 + 9 + 16
= 30

Input : N = 5
Output : 55
```

The idea is to run a loop from 1 to n and for each i, 1 <= i <= n, find i^2 to sum.

```
def squaresum(n) :

                        # Iterate i from 1
                        # and n finding
                        # square of i and
                        # add to sum.
    sm = 0
    for i in range(1, n+1) :
            sm = sm + (i * i)

    return sm

            # Driven Program
n =int(input("enter the number:"))
print(squaresum(n))
```

OUTPUT

```
enter the number:4
30
```