MODULE I - FUNDAMENTALS OF PYTHON

Features of Python – Data types: Numbers, Strings & its operations, Boolean – Operators – List & its operations, Tuples & its operations, Dictionaries & its operations – Arrays – Input and Output – Conditions statements: if, if-else, if-elif-else – Looping statements: while, for

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

FEATURES OF PYTHON

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

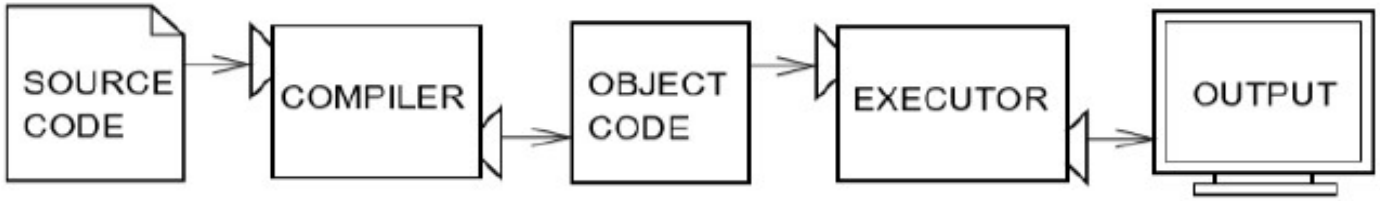
What is Python

Python is a general purpose, created by Guido Van Rossum, high level programming language, which makes coding easy and comfortable by simplifying the task.

Python, as a high level programming language, allows you to focus on core functionality of the application by taking care of common programming tasks

Python is interpreted programming language which is being used by the developers worldwide to develop various things like websites, machine learning algorithms and also simplifying and automating day to day tasks.

There are two ways to use the interpreter- Interactive mode & Script mode



WHY TO LEARN PYTHON?

Python is a

- high-level & interpreted,
- interactive and object-oriented scripting language.
- Python is designed to be highly readable.
- It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.
- **Python** is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain.

Some of the key advantages of learning Python:

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

FEATURES OF PYTHON

Following are important Features of **Python Programming**

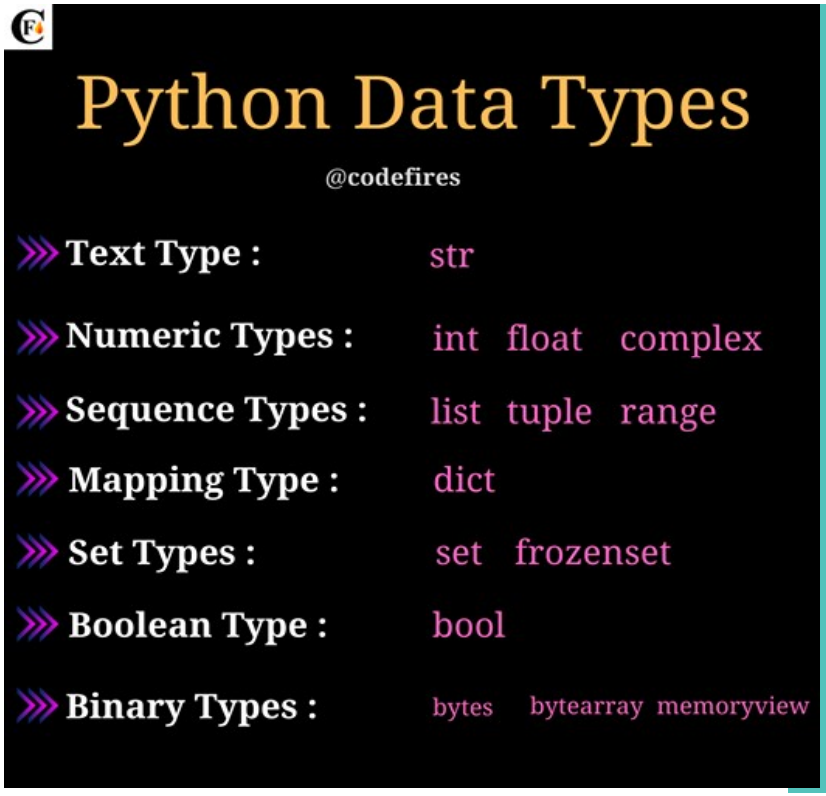
- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.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

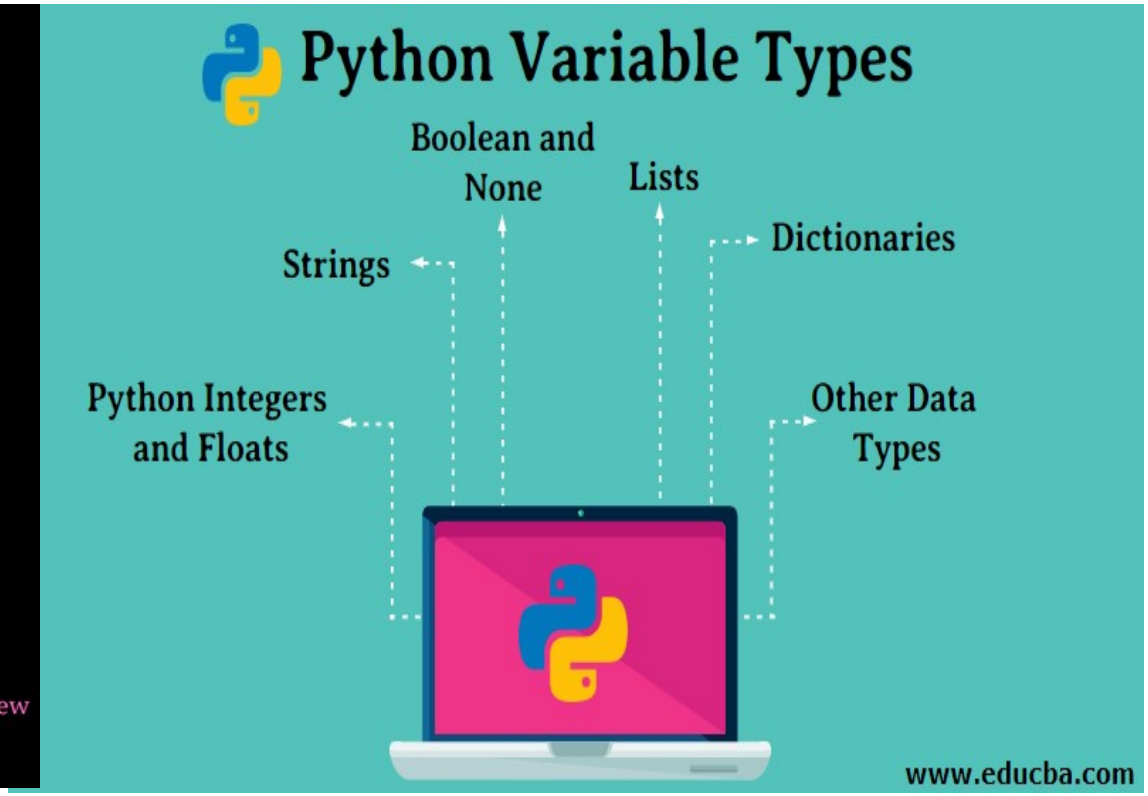
B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

DATA TYPES: NUMBERS, STRINGS & ITS OPERATIONS

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

DATA TYPES: NUMBERS, STRINGS & ITS OPERATIONS





Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Values and Variables

- In this chapter we explore some building blocks that are used to develop Python programs.
- We experiment with the following concepts:
  - ✓ numeric values
  - ✓ variables
  - ✓ assignment
  - ✓ Identifiers
  - ✓ reserved words

INTEGER VALUES

- The number four (4) is an example of a numeric value.
- In mathematics, 4 is an integer value.
- Integers are whole numbers, which means they have no fractional parts, and they can be positive, negative, or zero.
- Examples of integers include 4, −19, 0, and −1005.
- In contrast, 4.5 is not an integer, since it is not a whole number.
- Python supports a number of numeric and non-numeric values.
- In particular, Python programs can use integer values.
- The Python statement print(4) prints the value 4

Example –Integer value entered in python editor and its behavior

```
Python 3.2.1 (default, Jul 10 2011, 21:51:15) [MSC v.1500 32 bit (Intel)] o
32
Type "help", "copyright", "credits" or "license" for more information.
>>> 4
4
>>>
```

The interactive shell attempts to evaluate both expressions and statements. In this case, the expression 4 evaluates to 4. The shell executes what is commonly called the *read, eval, print loop*. This means the interactive shell's sole activity consists of

1. *reading* the text entered by the user,
2. attempting to *evaluate* the user's input in the context of what the user has entered up that point, and
3. *printing* its evaluation of the user's input.

Use of + symbol for normal arithmetic addition

Python uses the + symbol with integers to perform normal arithemtic addition, so the interactive shell can serve as a handy adding machine:

```
>>> 3 + 4
7
>>> 1 + 2 + 4 + 10 + 3
20
>>> print(1 + 2 + 4 + 10 + 3)
20
```

The last line evaluated shows how we can use the + symbol to add values within a `print` statement that could be part of a Python program.

Use of Int() & str() functions in python

Python associates the type name `int` with integer expressions and `str` with string expressions.

The built in `int` function converts the string representation of an integer to an actual integer, and the `str` function converts an integer expression to a string:

```
>>> 4
4
>>> str(4)
'4'
>>> '5'
'5'
>>> int('5')
5
```

The expression `str(4)` evaluates to the string value `'4'`, and `int('5')` evaluates to the integer value 5. The `int` function applied to an integer evaluates simply to the value of the integer itself, and similarly `str` applied to a string results in the same value as the original string:

```
>>> int(4)
4
>>> str('Judy')
'Judy'
```

How + operator works with string

The plus operator (+) works differently for strings; consider:

```
>>> 5 + 10
15
>>> '5' + '10'
'510'
>>> 'abc' + 'xyz'
'abcxyz'
```

As you can see, the result of the expression `5 + 10` is very different from `'5' + '10'`. The plus operator splices two strings together in a process known as *concatenation*. Mixing the two types directly is not allowed:

but the `int` and `str` functions can help:

```
>>> 5 + int('10')
15
>>> '5' + str(10)
'510'
```

“type” function to find the expression type(whether its int or str)

The type function can determine the type of the most complicated expressions:

```
>>> type(4)
<class 'int'>
>>> type('4')
<class 'str'>
>>> type(4 + 7)
<class 'int'>
>>> type('4' + '7')
<class 'str'>
>>> type(int('3') + int(4))
<class 'int'>
```

Variables and Assignment

- Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.
- Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory.
- Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example –

Live Demo

```
#!/usr/bin/python

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result –

```
100
1000.0
John
```

Multiple Assignment

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

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example –

```
a,b,c = 1,2,"john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Python Identifiers

- A Python identifier is a 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.

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase **(a to z)** or uppercase **(A to Z)** or digits **(0 to 9)** or an underscore `_`. Names like `myClass`, `var_1` and `print_this_to_screen`, all are valid example.
2. An identifier cannot start with a digit. `1variable` is invalid, but `variable1` is perfectly fine.
3. Keywords cannot be used as identifiers.

```
a.
b.  >>> global = 1
c.    File "<interactive input>", line 1
d.      global = 1
e.             ^
f.  SyntaxError: invalid syntax
```

4. We cannot use special symbols like **!**, **@**, **#**, **$**, **%** etc. in our identifier.

```
a.
b.  >>> a@ = 0
c.    File "<interactive input>", line 1
d.      a@ = 0
e.       ^
f.  SyntaxError: invalid syntax
```

5. Identifier can be of any length.

Python Keywords

Keywords are the reserved words in Python.

We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive.

There are 33 keywords in Python 3.7. This number can vary slightly in the course of time.

All the keywords except `True`, `False` and `None` are in lowercase and they must be written as it is. The list of all the keywords is given below.

Keywords in Python

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

Floating-point Types

- Python supports non-integer numbers, and they are called floating point numbers.
- The name implies that during mathematical calculations the decimal point can move or “float” to various positions within the number to maintain the proper number of significant digits.
- The Python name for the floating-point type is **float**.

Floating-Point Numbers

The `float` type in Python designates a floating-point number. `float` values are specified with a decimal point. Optionally, the character `e` or `E` followed by a positive or negative integer may be appended to specify scientific notation:

Python >>>

```
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2

>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Python supports four different numerical types –

- **int (signed integers)** – They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
- **long (long integers)** – Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
- **float (floating point real values)** – Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 ($2.5e2 = 2.5 \times 10^2 = 250$).
- **complex (complex numbers)** – are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Examples

Here are some examples of numbers

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEL	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J

STRING

String data type written is python in single or double code

x = ‘hello’

y = “Crescent”

z = input()

print(“ here is an example”, x, y, z)

Name = “Crescent”

Len(Name)

Output : ?

STRING

```
In [18]: Name="Crescent"
         len(Name)

Out[18]: 8

In [ ]:
```

Index no start o and lenth7

Name[2]

Output e

Name[2:8]

Output escent

Name.upper()

Output 'CRESCENT'

Python Strings

String Literals

String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

Example

```
print("Hello")
print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three double quotes Or three single quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

String Methods

Python has a set of built-in methods that you can use on strings.

Example

The strip() method removes any whitespace from the beginning or the end
:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

The lower() method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

The upper() method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

String Methods

The replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

Check String

To check if a certain phrase or character is present in a string, we can use the keywords in or not in.

Example

Check if the phrase "ain" is present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

String Methods

Example

Check if the phrase "ain" is NOT present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" not in txt
print(x)
```

String Concatenation

To concatenate, or combine, two strings you can use the + operator.

Example

Merge variable a with variable b into variable c:

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Example

To add a space between them, add a " ":

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

But we can combine strings and numbers by using the `format()` method!

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example

Use the `format()` method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

Write a Python program to check whether two strings are equal or not.

PROGRAM:

```
string1 = input("Enter first string: ")
string2 = input("Enter second string: ")
        if string1 == string2:
                print("\nBoth strings are equal to each other.")
        print(string1,"==",string2)
        else:
        print("\nStrings are not equal.")
        print(string1,"!=",string2)
```

OUTPUT:

```
Enter first string: CRESCENT
Enter second string: CRESCENT

Both strings are equal to each other.
CRESCENT == CRESCENT
```

Write a Python program to display reverse string.

PROGRAM:

```
# Python Program to Reverse String

string = input("Please enter your own String : ")
string2 = ‘ '
i = len(string) - 1
while(i >= 0):
    string2 = string2 + string[i]
    i = i - 1
print("\nThe Original String = ", string)
print("The Reversed String = ", string2)
```

OUTPUT:

```
Please enter your own String : Python
The Original String =  Python
The Reversed String =  nohtyP
```

Python Built-in String Methods

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found

Python Built-in String Methods

isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string

Python Built-in String Methods

maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Formatting Operator

%s - String (or any object with a string representation, like numbers)

%d – Integers

%f - Floating point numbers

%.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.
%x/%X - Integers in hex representation (lowercase/uppercase)

script.py

```
1  # This prints out "Hello, John!"
2  name = "John"
3  print("Hello, %s!" % name)
```

script.py

```
1  # This prints out "John is 23 years old."
2  name = "John"
3  age = 23
4  print("%s is %d years old." % (name, age))
```

Control codes within strings.

- The characters that can appear within strings include letters of the alphabet (A-Z, a-z), digits (0-9), punctuation (., :, ,, etc.), and other printable symbols (#, &, %, etc.).
- In addition to these "normal" characters, we may **embed special characters known as control codes.**
- Control codes control the way text is rendered in a console window or paper printer.
- **The backslash symbol (\) signifies that the character that follows it is a control code**, not a literal character.
- **The string '\n' thus contains a single control code.** The backslash is known as the escape symbol, and in this case we say the n symbol is escaped.
- The **\n control code represents the newline control code** which moves the text cursor down to the next line in the console window. Other control codes include **\t for tab,** \f for a form feed (or page eject) on a printer, **\b for backspace**, and \a for alert (or bell)

Control codes within strings Example Program

Listing 2.8: specialchars.py

```
1 print('A\nB\nC')
2 print('D\tE\tF')
3 print('WX\bYZ')
4 print('1\a2\a3\a4\a5\a6')
```

OUTPUT

```
A
B
C
D       E       F
WYZ
123456
```

Listing 2.9: escapequotes.py

```
1 print("Did you know that 'word' is a word?")
2 print('Did you know that "word" is a word?')
3 print('Did you know that \'word\' is a word?')
4 print("Did you know that \"word\" is a word?")
```

The output of Listing 2.9 (escapequotes.py) is

```
Did you know that 'word' is a word?
Did you know that "word" is a word?
Did you know that 'word' is a word?
Did you know that "word" is a word?
```

B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

BOOLEAN EXPRESSION & OPERATORS

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Boolean Expressions

- Arithmetic expressions evaluate to numeric values; a *Boolean* expression, sometimes called a *predicate*, may have only one of two possible values: *false* or *true*.
- The simplest Boolean expressions in Python are True and False.

```
>>> True
True
>>> False
False
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

We see that `bool` is the name of the class representing Python's Boolean expressions. Listing 4.1 (`boolvars.py`) is a simple program that shows how Boolean variables can be used.

Boolean Expressions

Expression	Meaning
$x == y$	True if $x = y$ (mathematical equality, not assignment); otherwise, false
$x < y$	True if $x < y$; otherwise, false
$x <= y$	True if $x \leq y$; otherwise, false
$x > y$	True if $x > y$; otherwise, false
$x <= y$	True if $x \geq y$; otherwise, false
$x != y$	True if $x \neq y$; otherwise, false

Table 4.1: The Python relational operators

Expression	Value
`10 < 20`	`True`
`10 >= 20`	`False`
`x < 100`	`True` if `x` is less than 100; otherwise, `False`
`x != y`	`True` unless `x` and `y` are equal

Table 4.2: Examples of some Simple Relational Expressions

Example : Boolean Expressions

Listing 4.1: boolvars.py

```
1  #  Assign some Boolean variables
2  a = True
3  b = False
4  print('a =', a, ' b =', b)
5  #  Reassign a
6  a = False;
7  print('a =', a, ' b =', b)
```

Listing 4.1 (`boolvars.py`) produces

```
a = True b = False
a = False b = False
```

3.1 Expressions

A literal value like 34 and a variable like x are examples of a simple expressions. Values and variables can be combined with operators to form more complex expressions.

sum = value1 + value2;

This is an **assignment statement** because is contains the **assignment operator (=).** The variable sum appears to the left of the assignment operator, so sum will receive a value when this statement executes. To the right of the assignment operator is an arithmetic expression involving two variables and

Listing 3.1: adder.py

```
1 value1 = eval(input('Please enter a number: '))
2 value2 = eval(input('Please enter another number: '))
3 sum = value1 + value2
4 print(value1, '+', value2,  '=', sum)
```

Expression	Meaning
$x + y$	x added to y, if x and y are numbers x concatenated to y, if x and y are strings
$x - y$	x take away y, if x and y are numbers
$x * y$	x times y, if x and y are numbers x concatenated with itself y times, if x is a string and y is an integer y concatenated with itself x times, if y is a string and x is an integer
x / y	x divided by y, if x and y are numbers
$x // y$	Floor of x divided by y, if x and y are numbers
$x \% y$	Remainder of x divided by y, if x and y are numbers
$x ** y$	x raised to y power, if x and y are numbers

Table 3.1: Commonly used Python arithmetic binary operators

x = 10, y = 10

1. x + y
   If x and y are numbers
   - 10 + 10 = 20

   If x and y are string
   - 10 + 10 = 1010
2. x - y
   If x and y are numbers
   - 10 - 10 = 0
3. x * y
   If x and y are numbers
   - 10 * 10 = 100

   If x is string and y is number
   - 10101010101010101010

   If x is number and y is string
   - 10101010101010101010
4. x / y
   If x and y are numbers
   - 10 /10 = 1
5. x // y
   If x and y are numbers
   - 10 // 10 = 1
6. x % y
   If x and y are numbers
   - 10 % 10 = 1
7. x ** y
   If x and y are numbers
   - 10 ** 10 = 10^{10} = 10000000000

Arithmetic operators in Python

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

Example #1: Arithmetic operators in Python

```
1.  x = 15
2.  y = 4
3.
4.  # Output: x + y = 19
5.  print('x + y =',x+y)
6.
7.  # Output: x - y = 11
8.  print('x - y =',x-y)
9.
10. # Output: x * y = 60
11. print('x * y =',x*y)
12.
13. # Output: x / y = 3.75
14. print('x / y =',x/y)
15.
16. # Output: x // y = 3
17. print('x // y =',x//y)
18.
19. # Output: x ** y = 50625
20. print('x ** y =',x**y)
```

When you run the program, the output will be:

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

ADDITION, SUBTRACTION, MULTIPLICATION AND DIVISION

1. Write a program for addition, subtraction, multiplication and division of two numbers

ALGORITHM:

STEP 1: Display choice of operation in screen, “Choice 1: Add, 2: Sub, 3: Div, 4: Mul, 5: Quit “.

STEP 2: Receive input choice from the user.

STEP 3: Convert the input to integer (n = int (n1)).

STEP 4: If choice of option is 1 then receive input for A and B, Initiate operation c = A + B.

STEP 5: If choice of option is 2 then receive input for A and B, Initiate operation c = A - B.

STEP 6: If choice of option is 3 then receive input for A and B, Initiate operation c = A / B.

STEP 7: If choice of option is 4 then receive input for A and B, Initiate operation c = A * B.

PROGRAM:

```
print ("Choice 1: Add, 2: Sub, 3: Div, 4: Mul, 5: Quit")
n1 = input("Enter the choice of operation  :")
n = int(n1)
if (n == 1):
   print (" You have chosen addition Option :")
   a1 = input(" Enter the value for A :")
   b1 = input(" Enter the value for B :")
   a = int(a1)
   b = int(b1)
   c = a + b
   print ("The Result",a,"+",b,"=",c)
elif (n == 2):
   print (" You have chosen subtraction Option :")
   a1 = input(" Enter the value for A :")
   b1 = input(" Enter the value for B :")
   a = int(a1)
   b = int(b1)
   c = a - b
   print ("The Result",a,"-",b,"=",c)
```

```
elif (n == 3):
   print (" You have chosen division Option :")
   a1 = input(" Enter the value for A :")
   b1 = input(" Enter the value for B :")
   a = int(a1)
   b = int(b1)
   c = a / b
   print ("The Result",a,"/",b,"=",c)
elif (n == 4):
   print (" You have chosen multiplication Option :")
   a1 = input (" Enter the value for A :")
   b1 = input (" Enter the value for B :")
   a = int (a1)
   b = int (b1)
   c = a * b
   print ("The Result”, a,"*", b,"=",c)
elif (n==5):
   print (“You have given a wrong option”, n)
   exit ()
```

SAMPLE OUTPUT:

If option 1 -> Then Addition

C = A + B --> C = 5 + 5 --> C = 10.

If option 2 -> Then Subtraction

C = A - B --> C = 5 - 2 --> C = 03.

If option 3 -> Then Division

C = A / B --> C = 6 / 2 --> C = 03.

If option 4 -> Then Multiplications

C = A * B --> C = 5 * 3 --> C = 15.

If option 5 -> You have given a wrong option

Exit**.**

Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Example #1: Arithmetic operators in Python

```
1.  x = 15
2.  y = 4
3.
4.  # Output: x + y = 19
5.  print('x + y =',x+y)
6.
7.  # Output: x - y = 11
8.  print('x - y =',x-y)
9.
10. # Output: x * y = 60
11. print('x * y =',x*y)
12.
13. # Output: x / y = 3.75
14. print('x / y =',x/y)
15.
16. # Output: x // y = 3
17. print('x // y =',x//y)
18.
19. # Output: x ** y = 50625
20. print('x ** y =',x**y)
```

When you run the program, the output will be:

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Example #2: Comparison operators in Python

```
1.  x = 10
2.  y = 12
3.
4.  # Output: x > y is False
5.  print('x > y  is',x>y)
6.
7.  # Output: x < y is True
8.  print('x < y  is',x<y)
9.
10. # Output: x == y is False
11. print('x == y is',x==y)
12.
13. # Output: x != y is True
14. print('x != y is',x!=y)
15.
16. # Output: x >= y is False
17. print('x >= y is',x>=y)
18.
19. # Output: x <= y is True
20. print('x <= y is',x<=y)
```

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Example #3: Logical Operators in Python

```
1.  x = True
2.  y = False
3.
4.  # Output: x and y is False
5.  print('x and y is',x and y)
6.
7.  # Output: x or y is True
8.  print('x or y is',x or y)
9.
10. # Output: not x is False
11. print('not x is',not x)
```

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example	Try it
is	Returns true if both variables are the same object	x is y	Try it »
is not	Returns true if both variables are not the same object	x is not y	Try it »

Example #4: Identity operators in Python

```
1.  x1 = 5
2.  y1 = 5
3.  x2 = 'Hello'
4.  y2 = 'Hello'
5.  x3 = [1,2,3]
6.  y3 = [1,2,3]
7.
8.  # Output: False
9.  print(x1 is not y1)
10.
11. # Output: True
12. print(x2 is y2)
13.
14. # Output: False
15. print(x3 is y3)
```

Membership operators

`in` and `not in` are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

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

Example #5: Membership operators in Python

```
1.  x = 'Hello world'
2.  y = {1:'a',2:'b'}
3.
4.  # Output: True
5.  print('H' in x)
6.
7.  # Output: True
8.  print('hello' not in x)
9.
10. # Output: True
11. print(1 in y)
12.
13. # Output: False
14. print('a' in y)
```

Here, `'H'` is in `x` but `'hello'` is not present in `x` (remember, Python is case sensitive). Similary, `1` is key and `'a'` is the value in dictionary `y`. Hence, `'a' in y` returns `False`.

Bitwise operators

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

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

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

Bitwise operators in Python

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

FIZZ BUZZ PROGRAM

3. Write a program to incorporate FIZZ for any number divisible by 3 and Buzz for any number divisible for 5 and FIZZBUZZ for any number divisible by 3 and 5 as well.

ALGORITHM:

STEP 1: Print “FizzBuzz Program”

STEP 2: prompt for user input for value N1

STEP 3: Convert the received value N1 to integer value N

STEP 4: Create a loop which will execute from 0 to N+1 times.

If I modules 3 and I modules 5 is zero then

Print I value + “= FIZZBUZZ”

If I modules 3 is zero then

Print I value + “= FIZZ”

If I modules 5 is zero then

Print I value + “= Buzz”

else

Print I Value

PROGRAM:

```
print ("Fizz Buzz Program :")
n1 = input("Enter the number : ")
n = int(n1)
i = 0
for i in range (n+1):
    if (i % 3 == 0 and i % 5 == 0):
        print (str(i) + "= Fizz Buzz")
    elif (i % 3 == 0):
        print (str(i) + "= Fizz ")
    elif (i % 5 == 0):
        print (str(i) + "= Buzz ")
    else:
        print(i)
```

OUTPUT:

```
FizzBuzz Program
Enter the number: 15
0
1
2
3 = FIZZ
4
5 = BUZZ
6
7
8
9 = FIZZ
10 = BUZZ
11
12 = FIZZ
13
14
15 = FIZZBUZZ
```

B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

LIST & ITS OPERATIONS, TUPLES & ITS OPERATIONS, DICTIONARIES & ITS OPERATIONS

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Python Lists

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Python List- Word Doc -- Click Here

List Programs - Click Here

Python Tuples

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Python Tuples

Python List VS Array VS Tuple

Array: An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The following are the main characteristics of an Array:

An array is an ordered collection of the similar data types.
An array is mutable.
An array can be accessed by using its index number.

List: A list is of an ordered collection data type that is mutable which means it can be easily modified and we can change its data values and a list can be indexed, sliced, and changed and each element can be accessed using its index value in the list. The following are the main characteristics of a List:

The list is an ordered collection of data types.
The list is mutable.
List are dynamic and can contain objects of different data types.
List elements can be accessed by index number.

Tuple: A tuple is an ordered and an immutable data type which means we cannot change its values and tuples are written in round brackets. We can access tuple by referring to the index number inside the square brackets. The following are the main characteristics of a Tuple:

Tuples are immutable and can store any type of data type.
it is defined using ().
it cannot be changed or replaced as it is an immutable data type.

Python Tuples

Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

Example

Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Python Tuples

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

Example

Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Note: The search will start at index 2 (included) and end at index 5 (not included).

Python Tuples

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

Example

This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Python Tuples

Loop Through a Tuple

You can loop through the tuple items by using a for loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Check if Item Exists

To determine if a specified item is present in a tuple use the in keyword:

Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Tuple Length

To determine how many items a tuple has, use the len() method:

Example

Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

Python Tuples

Add Items

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

Example

You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

Join Two Tuples

To join two or more tuples you can use the `+` operator:

Example

Join two tuples:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

Python Tuples

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Python Tuples

The tuple() Constructor

It is also possible to use the `tuple()` constructor to make a tuple.

Example

Using the tuple() method to make a tuple:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Dictionary

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
x = thisdict["model"]
```

Dictionary

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
x = thisdict["model"]
```

There is also a method called `get()` that will give you the same result:

Example

Get the value of the "model" key:

```
x = thisdict.get("model")
```

Dictionary

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Dictionary

Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.

When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example

Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

Example

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

Dictionary

Example

You can also use the `values()` method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

Example

Loop through both *keys* and *values*, by using the `items()` method:

```
for x, y in thisdict.items():
  print(x, y)
```

Dictionary

Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Example

Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Dictionary

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the `len()` function.

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

Dictionary

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Dictionary

Removing Items

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Dictionary

Example

The `popitem()` method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Example

The `del` keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Dictionary

Example

The `del` keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Dictionary

Example

The `clear()` method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Dictionary

Copy a Dictionary

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.

There are ways to make a copy, one way is to use the built-in Dictionary method `copy()`.

Example

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Dictionary

Another way to make a copy is to use the built-in function `dict()`.

Example

Make a copy of a dictionary with the `dict()` function:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Dictionary

Nested Dictionaries

A dictionary can also contain many dictionaries, this is called nested dictionaries.

Example

Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Dictionary

Example

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

Dictionary

The dict() Constructor

It is also possible to use the `dict()` constructor to make a new dictionary:

Example

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Dictionary

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

SET

B.S. Abdur Rahman Crescent Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956

Set is collection just like list or dictionary but it is unordered and no duplicate entries are present.

Example:

animals = (“Lion” , “Monkey” , “Snake”’ , “Fox”)

print(animals)

```
In [12]: myset = {10,20,30,40, 40,30, 'hello', 'courses'}

In [13]: myset

Out[13]: {10, 20, 30, 40, 'courses', 'hello'}
```

B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

ARRAYS

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
x = cars[0]
```

Example

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

The Length of an Array

Use the `len()` method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the `cars` array:

```
x = len(cars)
```

Adding Array Elements

You can use the `append()` method to add an element to an array.

Example

Add one more element to the `cars` array:

```
cars.append("Honda")
```

Removing Array Elements

You can use the `pop()` method to remove an element from the array.

Example

Delete the second element of the `cars` array:

```
cars.pop(1)
```

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

INPUT-OUTPUT

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

User Input

The `print` function enables a Python program to display textual information to the user. Programs may use the `input` function to obtain information from the user. The simplest use of the `input` function assigns a string to a variable:

```
x = input()
```

The parentheses are empty because, the `input` function does not require any information to do its job. Listing 2.11 (`usinginput.py`) demonstrates that the `input` function produces a string value.

Listing 2.11: usinginput.py

```
1 print('Please enter some text:')
2 x = input()
3 print('Text entered:', x)
4 print('Type:', type(x))
```

The following shows a sample run of Listing 2.11 (`usinginput.py`):

```
Please enter some text:
My name is Rick
Text entered: My name is Rick
Type: <class 'str'>
```

User Input- Example

Listing 2.12: addintegers.py

```
1 print('Please enter an integer value:')
2 x = input()
3 print('Please enter another integer value:')
4 y = input()
5 num1 = int(x)
6 num2 = int(y)
7 print(num1, '+', num2, '=', num1 + num2)
```

```
Please enter an integer value:
2
Please enter another integer value:
17
2 + 17 = 19
```

eval()

Definition and Usage

The `eval()` function evaluates the specified expression, if the expression is a legal Python statement, it will be executed.

Syntax

```
eval(expression, globals, locals)
```

Parameter Values

Parameter	Description
expression	A String, that will be evaluated as Python code
globals	Optional. A dictionary containing global parameters
locals	Optional. A dictionary containing local parameters

The eval function: eval()

- Python provides the eval function that attempts to evaluate a string in the same way that the interactive shell would evaluate it.

Example Program-Implementation of eval()

Listing 2.15: evalfunc.py

```
1  x1 = eval(input('Entry x1? '))
2  print('x1 =', x1, ' type:', type(x1))
3
4  x2 = eval(input('Entry x2? '))
5  print('x2 =', x2, ' type:', type(x2))
6
7  x3 = eval(input('Entry x3? '))
8  print('x3 =', x3, ' type:', type(x3))
9
10 x4 = eval(input('Entry x4? '))
11 print('x4 =', x4, ' type:', type(x4))
12
13 x5 = eval(input('Entry x5? '))
14 print('x5 =', x5, ' type:', type(x5))
```

Output : Implementation of eval() Output : Implementation of eval()

```
Entry x1? 4
x1 = 4  type: <class 'int'>
Entry x2? 4.0
x2 = 4.0  type: <class 'float'>
Entry x3? 'x1'
x3 = x1  type: <class 'str'>
Entry x4? x1
x4 = 4  type: <class 'int'>
Entry x5? x6
Traceback (most recent call last):
  File "C:\Users\rick\Documents\Code\Other\python\changeable.py", line 13,
    x5 = eval(input('Entry x5? '))
  File "<string>", line 1, in <module>
NameError: name 'x6' is not defined
```

B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

CONDITIONS STATEMENTS: IF, IF-ELSE, IF-ELIF-ELSE – LOOPING STATEMENTS: WHILE, FOR

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Simple if statement & if/else.

- **if** statement allows code to be optionally executed

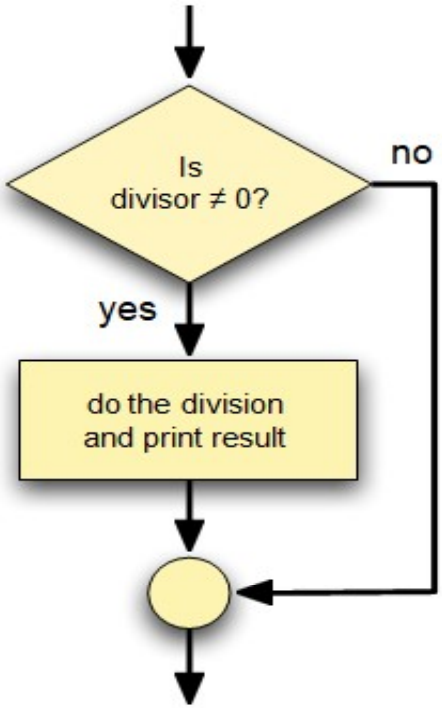
Listing 4.2: betterdivision.py

```
1  #  File betterdivision.py
2
3  #  Get two integers from the user
4  dividend, divisor = eval(input('Please enter two numbers to divide: '))
5  #  If possible, divide them and report the result
6  if divisor != 0:
7      print(dividend, '/', divisor, "=", dividend/divisor)
```

The print statement may not always be executed. In the following run

```
Please enter two numbers to divide: 32, 8
32 / 8 = 4.0
```

Simple if statement(flowchart & Syntax)



The general form of the `if` statement is:

```
if condition :
    block
```

- The reserved word `if` begins a `if` statement.
- The *condition* is a Boolean expression that determines whether or not the body will be executed. A colon (`:`) must follow the condition.
- The *block* is a block of one or more statements to be executed if the condition is true.

Simple if statement

- Python requires the block to be indented. If the block contains just one statement, some programmers will place it on the same line as the if; for example, the following if statement that optionally assigns y

```
if x < 10:
    y = x
```

could be written as

```
if   x < 10: y = x
```

but may *not* be written as

```
if x < 10:
     y = x
```

- because the lack of indentation hides the fact that the assignment statement is optionally executed.
- Indentation is how Python determines which statements make up a block.

Using spaces and tabs in python editor

- It is important not to mix spaces and tabs when indenting statements in a block.
- In many editors you cannot visually distinguish between a tab and a sequence of spaces.
- The number of spaces equivalent to the spacing of a tab differs from one editor to another.
- Most programming editors have a setting to substitute a specified number of spaces for each tab character.
- For Python development you should use this feature.
- It is best to eliminate all tabs within your Python source code
- How many spaces should you indent?
- Python requires at least one, some programmers consistently use two, four is the most popular number, but some prefer a more dramatic display and use eight.
- A four space indentation for a block is the recommended Python style.

If with multiple statement

The if block may contain multiple statements to be optionally executed. Listing 4.3 (alternatedivision.py) optionally executes two statements depending on the input values provided by the user.

Listing 4.3: alternatedivision.py

```
1 #  Get two integers from the user
2 dividend, divisor = eval(input('Please enter two numbers to divide: '))
3 #  If possible, divide them and report the result
4 if divisor != 0:
5     quotient = dividend/divisor
6     print(dividend, '/', divisor, "=", quotient)
7 print('Program finished')
```

The assignment statement and first printing statement are both a part of the block of the if. Given the truth value of the Boolean expression divisor != 0 during a particular program run, either both statements will be executed or neither statement will be executed. The last statement is not indented, so it is not part of the if block. The program always prints *Program finished*, regardless of the user's input.

Remember when checking for equality, as in

```
if x == 10:
    print('ten')
```

to use the relational equality operator (==), not the assignment operator (=).

The if/else Statement

The if statement has an optional else clause that is executed only if the Boolean condition is false.

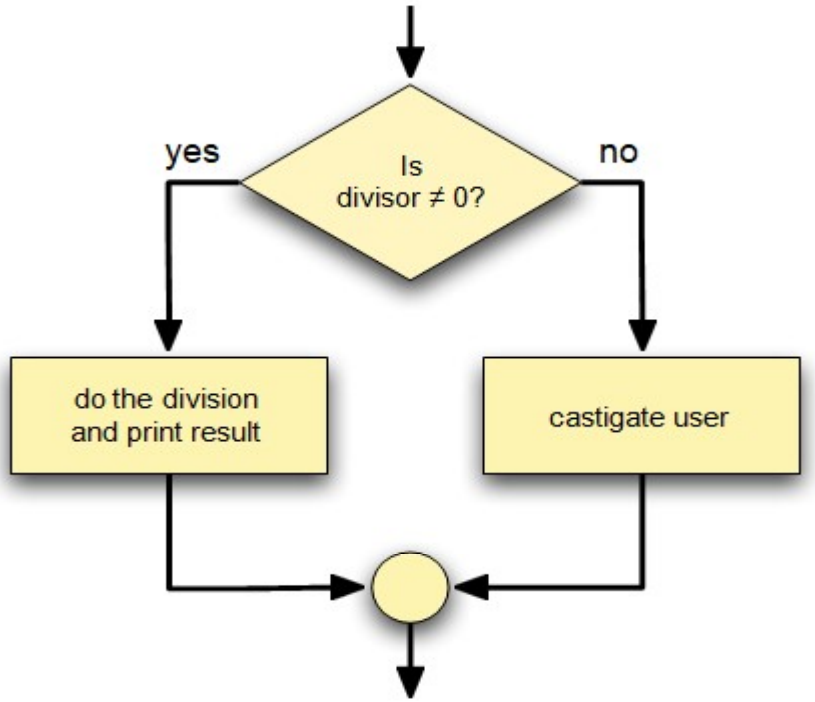
Listing 4.5: betterfeedback.py

```
1  #  Get two integers from the user
2  dividend, divisor = eval(input('Please enter two numbers to divide: '))
3  #  If possible, divide them and report the result
4  if divisor != 0:
5      print(dividend, '/', divisor, "=", dividend/divisor)
6  else:
7      print('Division by zero is not allowed')
```

A given run of Listing 4.5 (betterfeedback.py) will execute exactly one of either the if block or the else block. Unlike in Listing 4.2 (betterdivision.py), a message is always displayed.

```
Please enter two integers to divide: 32, 0
Division by zero is not allowed
```

if else statement(flowchart & Syntax)



The general form of an `if/else` statement is

```
if condition :
    if block
else:
    else block
```

- The reserved word `if` begins the `if/else` statement.
- The *condition* is a Boolean expression that determines whether or not the `if` block or the `else` block will be executed. A colon (`:`) must follow the condition.

if & else (compare & Contrast)

- The *if block* is a block of one or more statements to be executed if the condition is true. As with all blocks, it must be indented more spaces than the `if` line. This part of the `if` statement is sometimes called the body of the `if`.
- The reserved word `else` begins the second part of the `if/else` statement. A colon (`:`) must follow the `else`.
- The *else block* is a block of one or more statements to be executed if the condition is false. It must be indented more spaces than the `else` line. This part of the `if/else` statement is sometimes called the body of the `else`.

The `else` block, like the `if` block, consists of one or more statements indented to the same level.

Nested Conditionals

• The statements in the block of the if or the else may be any Python statements, including other if/else statements.

•These nested if statements can be used to develop arbitrarily complex control flow logic.

Listing 4.7: checkrange.py

```
1 value = eval(input("Please enter an integer value in the range 0...10: ")
2 if value >= 0:          #  First check
3     if value <= 10:     #  Second check
4         print("In range")
5 print("Done")
```

Listing 4.7 (checkrange.py) behaves as follows:

- The first condition is checked. If value is less than zero, the second condition is not evaluated and the statement following the outer if is executed. The statement after the outer if simply prints *Done*.
- If the first condition finds value to be greater than or equal to zero, the second condition then is checked. If the second condition is met, the *In range* message is displayed; otherwise, it is not. Regardless, the program eventually prints the *Done* message.

Nested Conditionals(using and logic)

Listing 4.8: newcheckrange.py

```
1 value = eval(input("Please enter an integer value in the range 0...10: ")
2 if value >= 0 and value <= 10: #  Only one, more complicated check
3     print("In range")
4 print("Done")
```

Nested Conditionals(using 2 if and2 else)

Listing 4.9: enhancedcheckrange.py

```
1 value = eval(input("Please enter an integer value in the range 0...10: ")
2 if value >= 0:          #  First check
3     if value <= 10:     #  Second check
4         print(value, "is in range")
5     else:
6         print(value, "is too large")
7 else:
8     print(value, "is too small")
9 print("Done")
```

Multi-way Decision Statements

Listing 4.11: digittoword.py

```
1  value = eval(input("Please enter an integer in the range 0...5: "))
2  if value < 0:
3      print("Too small")
4  else:
5      if value == 0:
6          print("zero")
7      else:
8          if value == 1:
9              print("one")
10         else:
11             if value == 2:
12                 print("two")
13             else:
14                 if value == 3:
15                     print("three")
16                 else:
17                     if value == 4:
18                         print("four")
19                     else:
20                         if value == 5:
21                             print("five")
22                         else:
23                             print("Too large")
24 print("Done")
```

Need of elif statement

- Notice that each `if` block contains a single printing statement and each `else` block, except the last one, contains an `if` statement. The control logic forces the program execution to check each condition in turn. The first condition that matches wins, and its corresponding `if` body will be executed. If none of the conditions are true, the program prints the last `else`'s *Too large* message. printed.

As a consequence of the required formatting of Listing 4.11 (`digittoword.py`), the mass of text drifts to the right as more conditions are checked. Python provides a multi-way conditional construct called `if`/`elif`/`else` that permits a more manageable textual structure for programs that must check many conditions. Listing 4.12 (`restyleddigittoword.py`) uses the `if`/`elif`/`else` statement to avoid the rightward code drift.

Program with elif statement

Listing 4.12: restyleddigittoword.py

```
1  value = eval(input("Please enter an integer in the range 0...5: "))
2  if value < 0:
3      print("Too small")
4  elif value == 0:
5      print("zero")
6  elif value == 1:
7      print("one")
8  elif value == 2:
9      print("two")
10 elif value == 3:
11     print("three")
12 elif value == 4:
13     print("four")
14 elif value == 5:
15     print("five")
16 else:
17     print("Too large")
18 print("Done")
```

The word `elif` is a contraction of `else` and `if`; if you read `elif` as *else if*, you can see how the code fragment

```
else:
    if value == 2:
        print("two")
```

in Listing 4.11 (`digittoword.py`) can be transformed into

```
elif value == 2:
    print("two")
```

Conditional Expressions

Consider the following code fragment:

```
if a != b:
    c = d
else:
    c = e
```

Here variable c is assigned one of two possible values. As purely a syntactical convenience, Python provides an alternative to the `if/else` construct called a *conditional expression*. A conditional expression evaluates to one of two values depending on a Boolean condition. The above code can be rewritten as

```
c = d if a != b else e
```

The general form of the conditional expression is

$$expression_1 \texttt{ if } condition \texttt{ else } expression_2$$

where

- $expression_1$ is the overall value of the conditional expression if *condition* is true.
- *condition* is a normal Boolean expression that might appear in an `if` statement.
- $expression_2$ is the overall value of the conditional expression if *condition* is false.

Example program for conditional expression

A Simple with if and else condition

Listing 4.14: safedivide.py

```
1 # Get the dividend and divisor from the user
2 dividend, divisor = eval(input('Enter dividend, divisor: '))
3 # We want to divide only if divisor is not zero; otherwise,
4 # we will print an error message
5 if divisor != 0:
6     print(dividend/divisor)
7 else:
8     print('Error, cannot divide by zero')
```

Above program rewritten using conditional expression

Listing 4.15: safedivideconditional.py

```
1 # Get the dividend and divisor from the user
2 dividend, divisor = eval(input('Enter dividend, divisor: '))
3 # We want to divide only if divisor is not zero; otherwise,
4 # we will print an error message
5 msg = dividend/divisor if divisor != 0 else 'Error, cannot divide by zero'
6 print(msg)
```

Iteration

- Iteration repeats the execution of a sequence of code.
- Iteration is useful for solving many programming problems.
- Iteration and conditional execution form the basis for algorithm construction.
- The process of executing the **same section of code over and over** is known as ***iteration*, or *looping***.
- Python has two different statements, **while** and **for**, that enable iteration.

Python Program with 'while' statement

Listing 5.2: iterativecounttofive.py

```
1  count = 1              #  Initialize counter
2  while count <= 5:      #  Should we continue?
3      print(count)       #  Display counter, then
4      count += 1         #  Increment counter
```

The while statement in Listing 5.2 (iterativecounttofive.py) repeatedly displays the variable count. The block of statements

```
print(count)
count += 1
```

are executed five times. After each redisplay of the variable count, the program increments it by one. Eventually (after five iterations), the condition `count <= 5` will no longer be true, and the block is no longer executed.

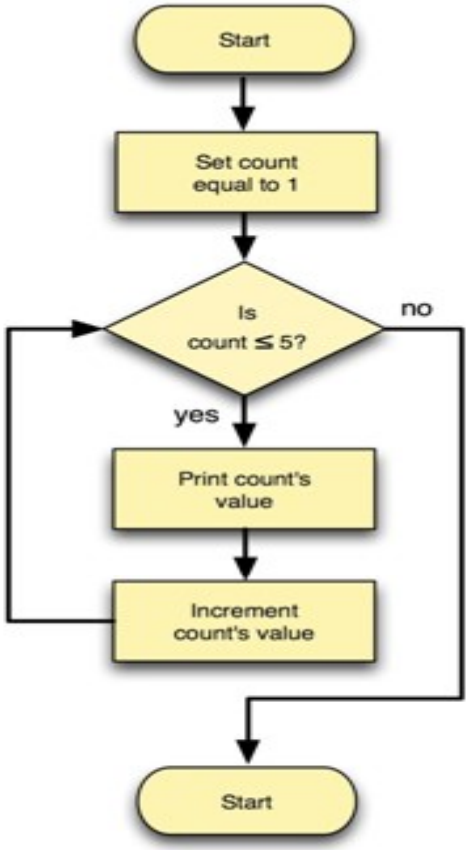
Unlike the approach taken in Listing 5.1 (counttofive.py), it is trivial to modify Listing 5.2 (iterativecounttofive.py) to count up to 10,000—just change the literal value 5 to 10000.

The line

```
while count <= 5:
```

begins the while statement. The expression following the while keyword is the condition that determines if the statement block is executed or continues to execute. As long as the condition is true, the program executes the code block over and over again. When the condition becomes false, the loop is finished. If the condition is false initially, the code block within the body of the loop is not executed at all.

While statement flow and syntax



The `while` statement has the general form:

```
while condition :
    block
```

- The reserved word `while` begins the `while` statement.
- The *condition* determines whether the body will be (or will continue to be) executed. A colon (`:`) must follow the condition.
- *block* is a block of one or more statements to be executed as long as the condition is true. As a block, all the statements that comprise the block must be indented the same number of spaces from the left. As with the `if` statement, the block must be indented more spaces than the line that begins the `while` statement. The block technically is part of the `while` statement.

Example program

Listing 5.3: addnonnegatives.py

```
1  #  Allow the user to enter a sequence of non-negative
2  #  numbers.  The user ends the list with a negative
3  #  number.  At the end the sum of the non-negative
4  #  numbers entered is displayed.  The program prints
5  #  zero if the user provides no non-negative numbers.
6
7  entry = 0       #  Ensure the loop is entered
8  sum = 0         #  Initialize sum
9
10 #  Request input from the user
11 print("Enter numbers to sum, negative number ends list:")
12
13 while entry >= 0:          #  A negative number exits the loop
14     entry = eval(input())  #  Get the value
15     if entry >= 0:         #  Is number non-negative?
16         sum += entry       #  Only add it if it is non-negative
17 print("Sum =", sum)        #  Display the sum
```

Definite Loop vs Indefinite Loop

Definite Loops vs. Indefinite Loops

In Listing 5.5 (definite1.py), code similar to Listing 5.1 (counttofive.py), prints the integers from one to 10.

Listing 5.5: definite1.py

```
1 n = 1
2 while n <= 10:
3     print(n)
4     n += 1
```

We can inspect the code and determine the number of iterations the loop performs. This kind of loop is known as a *definite loop*, since we can predict exactly how many times the loop repeats. Consider Listing 5.6 (definite2.py).

Listing 5.6: definite2.py

```
1 n = 1
2 stop = int(input())
3 while n <= stop:
4     print(n)
5     n += 1
```

Looking at the source code of Listing 5.6 (definite2.py), we cannot predict how many times the loop will repeat. The number of iterations depends on the input provided by the user. However, at the program's point of execution after obtaining the user's input and before the start of the execution of the loop, we would be able to determine the number of iterations the while loop would perform. Because of this, the loop in Listing 5.6 (definite2.py) is considered to be a definite loop as well.

Indefinite loop

Listing 5.7: indefinite.py

```
1  done = False                  #  Enter the loop at least once
2  while not done:
3      entry = eval(input())     #  Get value from user
4      if entry == 999:          #  Did user provide the magic number?
5          done = True           #  If so, get out
6      else:
7          print(entry)          #  If not, print it and continue
```

In Listing 5.7 (indefinite.py), we cannot predict at any point during the loop's execution how many iterations the loop will perform. The value to match (999) is know before and during the loop, but the variable entry can be anything the user enters. The user could choose to enter 0 exclusively or enter 999 immediately and be done with it. The while statement in Listing 5.7 (indefinite.py) is an example of an *indefinite loop*.

For statement

The `for` Statement

The `while` loop is ideal for indefinite loops. As Listing 5.4 (`troubleshootloop.py`) demonstrated, a programmer cannot always predict how many times a `while` loop will execute. We have used a `while` loop to implement a definite loop, as in

```
n = 1
while n <= 10:
    print(n)
    n += 1
```

The `print` statement in this code executes exactly 10 times every time this code runs. This code requires three crucial pieces to manage the loop:

- initialization: `n = 1`
- check: `n <= 10`
- update: `n += 1`

Python provides a more convenient way to express a definite loop. The `for` statement iterates over a range of values. These values can be a numeric range, or, as we shall, elements of a data structure like a string, list, or tuple. The above `while` loop can be rewritten

```
for n in range(1, 11):
    print(n)
```

The expression `range(1, 11)` creates an object known as an *iterable* that allows the `for` loop to assign to the variable `n` the values 1, 2, . . . , 10. During the first iteration of the loop, `n`'s value is 1 within the block. In the loop's second iteration, `n` has the value of 2. The general form of the `range` function call is

range declaration in for loop

range (*begin* , *end* , *step*)

where

- *begin* is the first value in the range; if omitted, the default value is 0
- *end* is **one past** the last value in the range; the *end* value may **not** be omitted
- *change* is the amount to increment or decrement; if the *change* parameter is omitted, it defaults to 1 (counts up by ones)

begin, *end*, and *step* must all be integer values; floating-point values and other types are not allowed.

The `range` function is very flexible. Consider the following loop that counts down from 21 to 3 by threes:

```
for n in range(21, 0, -3):
    print(n, '', end='')
```

It prints

```
21 18 15 12 9 6 3
```

Thus range(21, 0, −3) represents the sequence 21, 18, 15, 12, 9, 3.

Example Program with **range** exploration

The following code computes and prints the sum of all the positive integers less than 100:

```
sum = 0     #  Initialize sum
for i in range(1, 100):
    sum += i
print(sum)
```

The following examples show how range can be used to produce a variety of sequences:

- range(10) → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- range(1, 10) → 1, 2, 3, 4, 5, 6, 7, 8, 9
- range(1, 10, 2) → 1, 3, 5, 7, 9
- range(10, 0, -1) → 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
- range(10, 0, -2) → 10, 8, 6, 4, 2
- range(2, 11, 2) → 2, 4, 6, 8, 10
- range(-5, 5) → −5, −4, −3, −2, −1, 0, 1, 2, 3, 4
- range(1, 2) → 1
- range(1, 1) → (empty)
- range(1, -1) → (empty)
- range(1, -1, -1) → 1, 0
- range(0) → (empty)

Nested Loops

Nested Loops

Just like with `if` statements, `while` and `for` blocks can contain arbitrary Python statements, including other loops. A loop can therefore be nested within another loop. Listing 5.8 (`timestable.py`) prints a multiplication table on the screen using nested `for` loops.

Listing 5.8: `timestable.py`

```
1  #  Print a multiplication table to 10 x 10
2  #  Print column heading
3  print("       1   2   3   4   5   6   7   8   9  10")
4  print("    +-----------------------------------------")
5  for row in range(1, 11):          #  1 <= row <= 10, table has 10 rows
6      if row < 10:                  #  Need to add space?
7          print(" ", end="")
8      print(row, "| ", end="")      #  Print heading for this row.
9      for column in range(1, 11): #  Table has 10 columns.
10         product = row*column;     #  Compute product
11         if product < 100:         #  Need to add space?
12             print(end=" ")
13         if product < 10:          #  Need to add another space?
14             print(end=" ")
15         print(product, end=" ") #  Display product
16     print()                       #  Move cursor to next row
```

Nested Loops

Nested loops are used when an iterative process itself must be repeated. Listing 5.9 (`flexibletimestable.py`) uses a `for` inner loop to print the contents of each row, but multiple rows must be printed. The inner (`column`) loop prints the contents of each row, while the outer (`row`) loop is responsible for printing all the rows.

Listing 5.10 (`permuteabc.py`) uses a triply-nested loop to print all the different arrangements of the letters A, B, and C. Each string printed is a *permutation* of ABC.

Listing 5.10: `permuteabc.py`

```
1  #  File permuteabc.py
2
3  #  The first letter varies from A to C
4  for first in 'ABC':
5      for second in 'ABC': #  The second varies from A to C
6          if second != first:   #   No duplicate letters allowed
7              for third in 'ABC':   #  The third varies from A to C
8                  #  Don't duplicate first or second letter
9                  if third != first and third != second:
10                     print(first + second + third)
```

Notice how the `if` statements are used to prevent duplicate letters within a given string. The output of Listing 5.10 (`permuteabc.py`) is all six permutations of ABC:

```
ABC
ACB
BAC
BCA
CAB
CBA
```

Abnormal Loop Termination

• Normally, a while statement executes until its condition becomes false. This condition is checked only at the ”top” of the loop, so the loop is not immediately exited if the condition becomes false due to activity in the middle of the body.
• Ordinarily this behavior is not a problem because the intention is to execute all the statements within the body as an indivisible unit.
• Sometimes, however, it is desirable to immediately exit the body or recheck the condition from the middle of the loop instead. Python provides the break and continue statements to give programmers more flexibility designing the control logic of loops.

Abnormal Looping-Break Statement

The `break` statement

Python provides the `break` statement to implement middle-exiting control logic. The `break` statement causes the immediate exit from the body of the loop. Listing 5.12 (`addmiddleexit.py`) is a variation of Listing 5.3 (`addnonnegatives.py`) that illustrates the use of `break`.

Listing 5.12: `addmiddleexit.py`

```
1  #  Allow the user to enter a sequence of non-negative
2  #  numbers.  The user ends the list with a negative
3  #  number.  At the end the sum of the non-negative
4  #  numbers entered is displayed.  The program prints
5  #  zero if the user provides no non-negative numbers.
6
7  entry = 0       #  Ensure the loop is entered
8  sum = 0         #  Initialize sum
9
10 #  Request input from the user
11 print("Enter numbers to sum, negative number ends list:")
12
13 while True:                  #  Loop forever
14     entry = eval(input())    #  Get the value
15     if entry < 0:            #  Is number negative number?
16         break                #  If so, exit the loop
17     sum += entry             #  Add entry to running sum
18 print("Sum =", sum)          #  Display the sum
```

The condition of the `while` is a tautology, so the body of the loop will be entered. Since the condition of the `while` can never be false, the `break` statement is the only way to get out of the loop. The `break` statement is executed only when the user enters a negative number. When the `break` statement is encountered during the program's execution, the loop is immediately exited. Any statements following the `break` within the body are skipped. It is not possible, therefore, to add a negative number to the `sum` variable.

Abnormal Looping-continue Statement

The continue Statement

The continue statement is similar to the break statement. During a program's execution, when the break statement is encountered within the body of a loop, the remaining statements within the body of the loop are skipped, and the loop is exited. When a continue statement is encountered within a loop, the remaining statements within the body are skipped, but the loop condition is checked to see if the loop should continue or be exited. If the loop's condition is still true, the loop is not exited, but the loop's execution continues at the top of the loop. Listing 5.14 (continueexample.py) shows how the continue statement can be used.

Listing 5.14: continueexample.py

```
1  sum = 0
2  done = False;
3  while not done:
4      val = eval(input("Enter positive integer (999 quits):"))
5      if val < 0:
6          print("Negative value",  val,  "ignored")
7          continue;  # Skip rest of body for this iteration
8      if val != 999:
9          print("Tallying", val)
10         sum += val
11     else:
12         done = (val == 999);   #  999 entry exits loop
13 print("sum =", sum)
```

The continue statement is not used as frequently as the break statement since it is often easy to transform the code into an equivalent form that does not use continue. Listing 5.15 (nocontinueexample.py) works exactly like Listing 5.14 (continueexample.py), but the continue has been eliminated.

Infinite Looping

Infinite Loops

An infinite loop is a loop that executes its block of statements repeatedly until the user forces the program to quit. Once the program flow enters the loop's body it cannot escape. Infinite loops are sometimes designed. For example, a long-running server application like a Web server may need to continuously check for incoming connections. This checking can be performed within a loop that runs indefinitely. All too often for beginning programmers, however, infinite loops are created by accident and represent logical errors in their programs.

Intentional infinite loops should be made obvious. For example,

```
while True:
    #  Do something forever. . .
```

The Boolean literal `True` is always true, so it is impossible for the loop's condition to be false. The only ways to exit the loop is via a `break` statement, `return` statement (see Chapter 7), or a `sys.exit` call (see Section ??) embedded somewhere within its body.

Intentional infinite loops are easy to write correctly. Accidental infinite loops are quite common, but can be puzzling for beginning programmers to diagnose and repair. Consider Listing 5.16 (`findfactors.py`) that attempts to print all the integers with their associated factors from 1 to 20.

Example program for infinite loop

Listing 5.16: findfactors.py

```
1  #  List the factors of the integers 1...MAX
2  MAX = 20                                #  MAX is 20
3  n = 1   #  Start with 1
4  while n <= MAX:                         #  Do not go past MAX
5      factor = 1                          #  1 is a factor of any integer
6      print(end=str(n) + ': ')            #  Which integer are we examining?
7      while factor <= n:                  #  Factors are <= the number
8          if n % factor == 0:             #  Test to see if factor is a factor of n
9              print(factor, end=' ')      #  If so, display it
10             factor += 1                 #  Try the next number
11     print()  #  Move to next line for next n
12     n += 1
```

It displays

```
1: 1
2: 1 2
3: 1
```

Thank You

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*