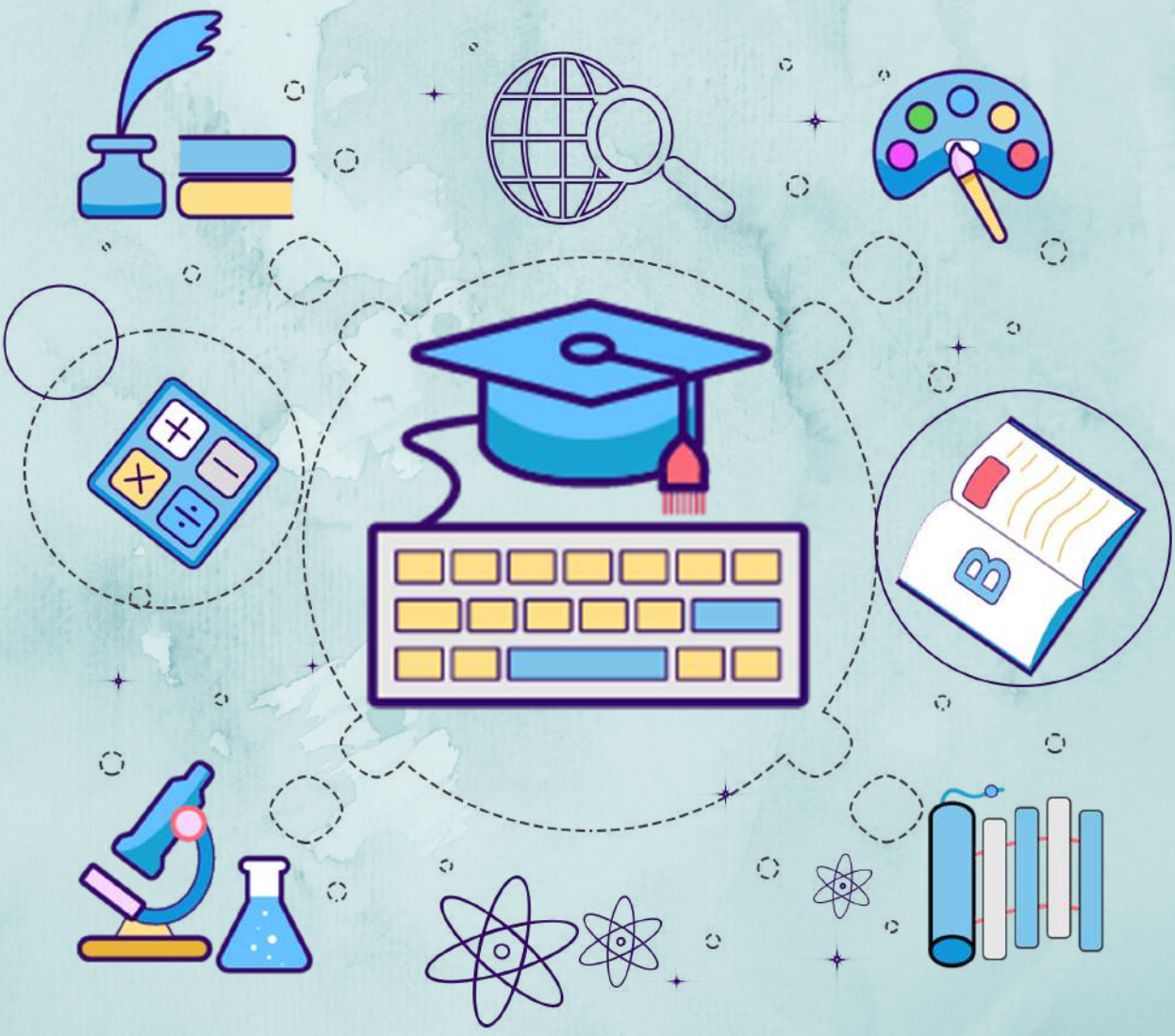
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

PROGRAMMING IN PYTHON

CST 362

Module 1

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

MODULE I

Getting started with Python programming – Interactive shell, IDLE, iPython Notebooks, Detecting and correcting syntax errors, How Python works. The software development process – A case study. Basic coding skills – strings, assignment, and comments, Numeric data types and character sets, Expressions, Using inbuilt functions and modules. Control statements – Iteration with for/while loop, Formatting text for output, A case study, Selection structure (if-else, switch case), Conditional iteration with while, A case study, Testing control statements, Lazy evaluation.

Getting Started with Python Programming

- Python is a high-level, general-purpose programming language for solving problems on modern computer systems.
- The language and many supporting tools are free, and Python programs can run on any operating system.
- It has fewer syntactical constructions than other languages.
- Python is an interpreted language, not a compiled language. That means that unlike applications written in languages such as C, COBOL or Assembler, code written in Python has to run through a process of interpretation by the computer.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Features and Applications:

1. Easy to learn:- It has few keywords, simple structure and clearly defined syntax.
2. Easy to read:- It is clearly defined.
3. Easy to maintain:- It is fairly easy to maintain.
4. Portable:- It can run wide variety of hardware platforms and has the same interface on all platforms.
5. Databases:- It provides interfaces to all major commercial databases.
6. Extendable:- Adding of low level modules to python is possible.
7. Scalable:- It provides a better structure and support for larger programs.
8. It supports functional and structured programming.
9. It can be used as scripting language and can be compiled to byte code to form largerapplications.
10. It supports automatic garbage collection.
11. It can be easily integrated with other programming languages like c,c++, java

Downloading and installing Python

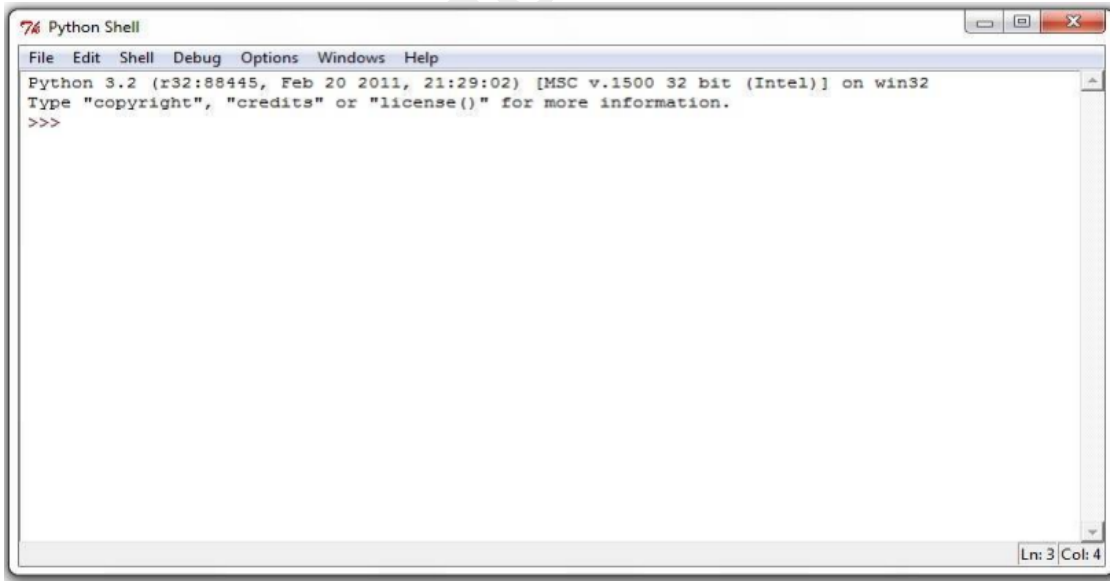
- You can download Python, its documentation, and related materials from www.python.org.
- Python does not come preinstalled on Windows systems.
- Therefore, you must download the Windows installer from http://www.python.org/download/. The installer might then run automatically, or you might have to double click an icon for the installer to

launch it. The installer automatically puts Python into a folder and inserts various command options on the All Programs menu.

- Python that comes preinstalled on Mac OS X, Unix and Linux but the user might need to update the version of Python.

Running Code in the Interactive Shell

- Python is an **interpreted language**, and you can run simple Python expressions and statements in an interactive programming environment called the **shell**.
- The easiest way to open a Python shell is to launch the **IDLE(Integrated Development and Learning Environment)**. This is an integrated program development environment that comes with the Python installation.
- When you do this, a window named Python Shell opens.



- A shell window contains an opening message followed by the special symbol >>>, called a **shell prompt**. The cursor at the shell prompt waits for you to enter a Python command.
- When you enter an expression or statement, Python evaluates it and displays its result, if there is one, followed by a new prompt. To quit the Python shell, you can either select the window's close box or press the Control+D key combination.

Input, Processing, and Output

- Most useful programs accept inputs from some source, process these inputs, and then finally output results to some destination. In terminal-based interactive programs, the input source is the keyboard, and the output destination is the terminal display.

Print Function

- The programmer can also force the output of a value by using the print function. The basic syntax (or grammatical rule) for using the print function:

  print(<expression>)

- When running the **print** function, Python first evaluates the expression and then displays its value.
    1. **print** was used to display some text. In Python code, a string is always enclosed in quotation marks. However, the **print** function displays a string without the quotation marks.

Eg: print("Hi there")

2. You can also write a print function that includes two or more expressions separated by commas. In such a case, the print function evaluates the expressions and displays their results, separated by single spaces, on one line.

Eg: print (2+2, "hello")

- Print function always ends its output with a newline.
- To begin the next output on the same line as the previous one, you can place the expression end = "", which says "end the line with an empty string instead of a newline," at the end of the list of expressions, as follows:

print(<expression>, end = "")

Input Function

- As you create programs in Python, you'll often want your programs to ask the user for input. You can do this by using the input function. This function causes the program to stop and wait for the user to enter a value from the keyboard.
- When the user presses the return or enter key, the function accepts the input value and makes it available to the program.
- The form of an assignment statement with the input function is the following:

<variable identifier>=input(<a string prompt>)

- A **variable identifier**, or **variable** for short, is just a name for a value. When a variable receives its value in an input statement, the variable then refers to this value.

Type conversion

- The input function always builds a string from the user's keystrokes and returns it to the program. After inputting strings that represent numbers, the programmer must convert them from strings to the appropriate numeric types.
- In Python, there are two **type conversion functions** for this purpose, called **int** (for integers) and **float** (for floating point numbers).

Example:

```
>>> first=int(input("enter the first number: "))
enter the first number: 12
>>> second=int(input("enter the second number: "))
enter the second number: 34
>>> print ("the sum is", first+second)
the sum is 46
```

Basic Python functions for input and output

Function	What It Does
`float(<a string of digits>)`	Converts a string of digits to a floating-point value.
`int(<a string of digits>)`	Converts a string of digits to an integer value.
`input(<a string prompt>)`	Displays the string prompt and waits for keyboard input. Returns the string of characters entered by the user.
`print(<expression>, ...,<expression>)`	Evaluates the expressions and displays them, separated by one space, in the console window.
`<string 1> + <string 2>`	Glues the two strings together and returns the result.

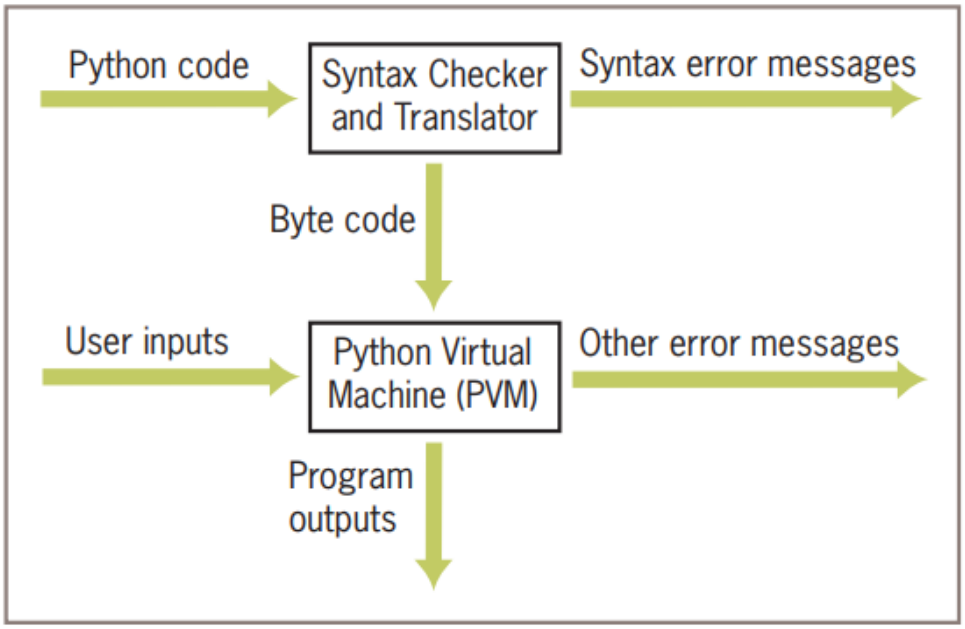
Editing, Saving and Running a Script

- While it is easy to try out short Python expressions and statements interactively at a shell prompt, it is more convenient to compose, edit, and save longer, more complex programs in files.
- We can then run these program files or **scripts** either within IDLE or from the operating system's command prompt without opening IDLE.
- To compose and execute programs in this manner, you perform the following steps:
    1. Select the option New Window from the File menu of the shell window.
    2. In the new window, enter Python expressions or statements on separate lines, in the order in which you want Python to execute them.
    3. At any point, you may save the file by selecting File/Save. If you do this, you should use a .py extension. For example, your first program file might be named myprogram.py.

4. To run this file of code as a Python script, select Run Module from the Run menu or press the F5 key (Windows) or the Control+F5 key (Mac or Linux).

How Python works

- Whether you are running Python code as a script or interactively in a shell, the Python interpreter does a great deal of work to carry out the instructions in your program.
- This work can be broken into a series of steps,



1. The interpreter reads a Python expression or statement, also called the **source code**, and verifies that it is well formed. In this step, the interpreter behaves like a strict English teacher who rejects any sentence that does not

adhere to the grammar rules, or syntax, of the language. As soon as the interpreter encounters such an error, it halts translation with an error message.

2. If a Python expression is well formed, the interpreter then translates it to an equivalent form in a low-level language called **byte code**. When the interpreter runs a script, it completely translates it to byte code.
3. This byte code is next sent to another software component, called the **Python virtual machine (PVM)**, where it is executed. If another error occurs during this step, execution also halts with an error message.

Detecting and Correcting Syntax Errors

- Programmers inevitably make typographical errors when editing programs, and the Python interpreter will nearly always detect them. Such errors are called **syntax errors**.
- When Python encounters a syntax error in a program, it halts execution with an error message.

```
>>> length = int(input("Enter the length: "))
Enter the length: 44

>>> print(lenth)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
NameError: name 'lenth' is not defined
```

- The first statement assigns an input value to the variable **length**. The next statement attempts to print the value of the variable **lenth**. Python responds that this name is not defined.

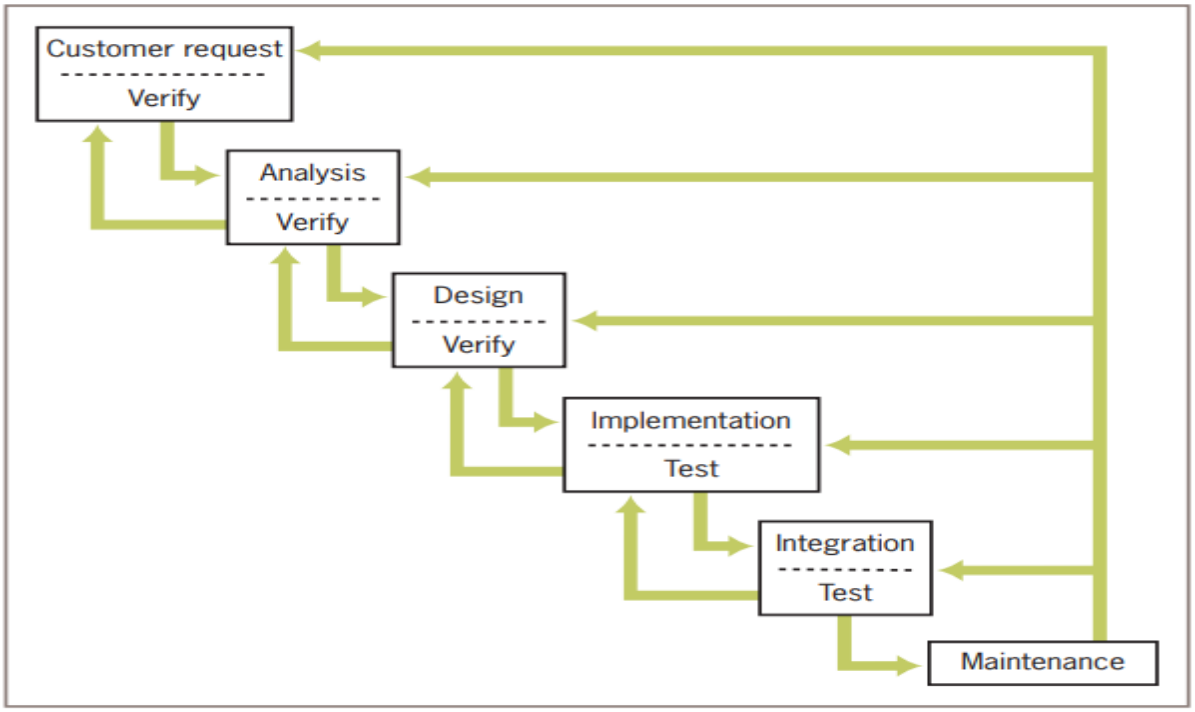
```
>>>  print length
  File "<pyshell#1>", line 1
    print length
    ^
SyntaxError: unexpected indent
```

- In this error message, Python explains that this line of code is unexpectedly indented. In fact, there is an extra space before the word print.
- **Indentation** is significant in Python code. Each line of code entered at a shell prompt or in a script must begin in the leftmost column, with no leading spaces. The only exception to this rule occurs in control statements and definitions, where nested statements must be indented one or more spaces.

The Software Development Process

- The process of planning and organizing a program is called as **software development process**.
- There are several approaches to software development. One version is known as the **waterfall model**.
- The waterfall model consists of several phases:

1. **Customer request**—In this phase, the programmers receive a broad statement of a problem .This step is also called the user requirements phase.
2. **Analysis**—The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.
3. **Design**—The programmers determine how the program will do its task.
4. **Implementation**—The programmers write the program. This step is also called the coding phase.
5. **Integration**—Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.
6. **Maintenance**—Programs usually have a long life; a life span of 5 to 15 years is common for software. During this time, requirements change, errors are detected, and minor or major modifications are made.



- The figure resembles a waterfall, in which the results of each phase flow down to the next.
- However, a mistake detected in one phase often requires the developer to back up and redo some of the work in the previous phase. Modifications made during maintenance also require backing up to earlier phases. Taken together, these phases are also called the **software development life cycle**.
- Modern software development is usually **incremental** and **iterative**. This means that analysis and design may produce a rough draft, skeletal version, or prototype of a system for coding, and then back up to earlier phases to fill in more details after some testing.
- Keep in mind that mistakes found early are much less expensive to correct than those found late. These are not just financial costs but also costs in time and effort.

Basic coding skills:

Strings, Assignment, and Comments

Data Types

- A **data type** consists of a set of values and a set of operations that can be performed on those values.
- A **literal** is the way a value of a data type looks to a programmer. The programmer can use a literal in a program to mention a data value.

Type of Data	Python Type Name	Example Literals
Integers	`int`	`-1, 0, 1, 2`
Real numbers	`float`	`-0.55, .3333, 3.14, 6.0`
Character strings	`str`	`"Hi", "", 'A', "66"`

String Literals

- In Python, a string literal is a sequence of characters enclosed in single or double quotation marks.

```
>>> 'Hello there!'
'Hello there!'
>>> "Hello there!"
'Hello there!'
>>> ' '
' '
>>> " "
' '
```

- The last two string literals (" and "") represent the empty string. Although it contains no characters, the empty string is a string nonetheless.
- To output a paragraph of text that contains several lines

```
>>> print("""This very long sentence extends
all the way to the next line.""")
This very long sentence extends
all the way to the next line.
```

Escape Sequences

- Escape sequences are the way Python expresses special characters, such as the tab, the newline, and the backspace (delete key), as literals.

Escape Sequence	Meaning
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	The \ character
\'	Single quotation mark
\"	Double quotation mark

String Concatenation

- We can join two or more strings to form a new string using the concatenation operator +.

```
>>> "Hi " + "there, " + "Ken!"
'Hi there, Ken!'
```

- The * operator allows you to build a string by repeating another string a given number of times. The left operand is a string, and the right operand is an integer.
- If you want the string "Python" to be preceded by 10 spaces, it would be easier to use the * operator with 10 and one space than to enter the 10 spaces by hand.

```
>>> " " * 10 + "Python"
```

' Python'

Variables and the Assignment Statement

- A variable associates a name with a value, making it easy to remember and use the value later in a program.
    - A variable name must begin with either a letter or an underscore (_).
    - It can contain any number of letters, digits, or other underscores.
    - Python variable names are case sensitive.
- Programmers use all uppercase letters for the names of variables that contain values that the program never changes. Such variables are known as **symbolic constant.**

Assignment Statements

< variable name> = <expression>

- The Python interpreter first evaluates the expression on the right side of the assignment symbol and then binds the variable name on the left side to this value. When this happens to the variable name for the first time, it is called **defining** or **initializing** the variable.

Program Comments and Docstrings

- A comment is a piece of program text that the computer ignores but that provides useful documentation to programmers.

- The author of a program can include his or her name and a brief statement about the program's purpose at the beginning of the program file. This type of comment, called a **docstring**.

```
"""
Program: circle.py
Author: Ken Lambert
Last date modified: 10/10/17

The purpose of this program is to compute the area of a circle. The
input is an integer or floating-point number representing the radius
of the circle. The output is a floating-point number labeled as the
area of the circle.
"""
```

- In addition to docstrings, **end-of-line comments** can document a program. These comments begin with the # symbol and extend to the end of a line. An end-of-line comment might explain the purpose of a variable or the strategy used by a piece of code.

```
>>> RATE = 0.85   # Conversion rate for Canadian to US dollars
```

Numeric Data Types and Character Sets

Numeric Data Types

- **Integers** - the integers include 0, all of the positive whole numbers, and all of the negative whole numbers.

- **Floating - Point Numbers** - A real number in mathematics, such as the value of pi (3.1416…), consists of a whole number, a decimal point, and a fractional part.

Character Sets

- All data and instructions in a program are translated to binary numbers before being run on a real computer. To support this translation, the characters in a string each map to an integer value. This mapping is defined in **character sets**, among them the ASCII set and the Unicode set
- The term ASCII stands for American Standard Code for Information Interchange. In the 1960s, the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 127.

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DCI	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	“	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	I	j	k	l	m
11	n	o	P	q	r	S	t	u	v	w
12	X	y	z	{	|	}	~	DEL		

- The digits in the left column represent the leftmost digits of an ASCII code, and the digits in the top row are the rightmost digits. Thus, the ASCII code of the character 'R' at row 8, column 2 is 82.
- Python's ord() and chr() functions convert characters to their numeric ASCII codes and back again, respectively.

```
>>> ord('a')
97
 >>> ord('A')
65
>>> chr(65)
'A'
>>> chr(66)
'B'
```

Expression

- Expressions provide an easy way to perform operations on data values to produce other data values.

Arithmetic Expressions

- An arithmetic expression consists of operands and operators.

Operator	Meaning	Syntax
-	Negation	-a
**	Exponentiation	a ** b
*	Multiplication	a * b
/	Division	a / b
//	Quotient	a // b
%	Remainder or modulus	a % b
+	Addition	a + b
-	Subtraction	a - b

Precedence rules

- Exponentiation has the highest precedence and is evaluated first.
- Unary negation is evaluated next, before multiplication, division, and remainder.
- Multiplication, both types of division, and remainder are evaluated before addition and subtraction.
- Addition and subtraction are evaluated before assignment.
- With two exceptions, operations of equal precedence are left associative, so they are evaluated from left to right. Exponentiation and assignment operations are right associative, so consecutive instances of these are evaluated from right to left.
- You can use parentheses to change the order of evaluation.

Expression	Evaluation	Value
5 + 3 * 2	5 + 6	11
(5 + 3) * 2	8 * 2	16
6 % 2	0	0
2 * 3 ** 2	2 * 9	18
-3 ** 2	-(3 ** 2)	-9
(3) ** 2	9	9
2 ** 3 ** 2	2 ** 9	512
(2 ** 3) ** 2	8 ** 2	64
45 / 0	Error: cannot divide by 0	
45 % 0	Error: cannot divide by 0	

Mixed-Mode Arithmetic and Type Conversions

- Performing calculations involving both integers and floating-point numbers is called mixed-mode arithmetic.

  Example:

```
>>> 3.14 * 3 ** 2
    28.26
```

- You must use a **type conversion** function when working with the input of numbers.
- A type conversion function is a function with the same name as the data type to which it converts. Because the input function returns a string as its value,

you must use the function int or float to convert the string to a number before performing arithmetic, as in the following

Example:

```
>>> radius = input("Enter the radius: ")
Enter the radius: 3.2
>>> radius
'3.2'
>>> float(radius)
3.2
>>> float(radius) ** 2 * 3.14
32.153600000000004
```

- Another use of type conversion occurs in the construction of strings from numbers and other strings.

```
>>> profit = 1000.55
>>> print('$' + str(profit))
$1000.55
```

- Python is a **strongly typed** programming language. The interpreter checks data types of all operands before operators are applied to those operands. If the type of an operand is not appropriate, the interpreter halts execution with an error message.

Using Functions and Modules

- Python includes many useful functions, which are organized in libraries of code called modules.
- A function is a chunk of code that can be called by name to perform a task. Functions often require arguments, that is, specific data values, to perform their tasks.
- Names that refer to arguments are also known as parameters.
- The process of sending a result back to another part of a program is known as returning a value.

Ex: round(7.563,2) return 7.56

abs(4-5) returns 1

- The above functions belongs to __builtin__ module.

Math Module

- The math module includes several functions that perform basic mathematical operations.
- This list of function names includes some familiar trigonometric functions as well as Python's most exact estimates of the constants pi and e.

Ex: 'sin' , 'sinh' , 'sqrt' , 'tan' , 'tanh' , 'tau' , 'trunc'..... etc.

Syntax for using specific function from math module

Ex:

```
>>>>import math
>>>>math.pi
3.1415926535897931
>>>>math.sqrt(2)
1.4142135623730951
```

- To import specific functions

Ex:

```
from math import pi, sqrt
>>> print(pi, sqrt(2))
3.14159265359 1.41421356237
```

- to import all of the math module's resources

Ex:

```
from math import *
```

Program Format and Structure

- Start with an introductory comment stating the author's name, the purpose of the program, and other relevant information. This information should be in the form of a docstring.
- Then, include statements that do the following:

- Import any modules needed by the program.
- Initialize important variables, suitably commented.
- Prompt the user for input data and save the input data in variables.
- Process the inputs to produce the results.
- Display the results.

Control statements

- Statements that allow the computer to select an action to perform in a particular action or to repeat a set of actions.
- Two types of control statements are there:
  - Selection structure
  - Iteration structure

Iteration Structure

- Also known as loops, which repeat an action.
- Each repetition of the action is known as a pass or an iteration.
- There are two types of loops
  - Those that repeat an action a predefined number of times (definite iteration).
  - Those that perform the action until the program determines that it needs to stop (indefinite iteration).

The for Loop

- The control statement that most easily supports definite iteration.
- The form of this type of loop is:

```
for <variable> in range(<an integer expression>):
    <statement-1>

    <statement-n>
```

- The first line of code in a loop is sometimes called the loop header.
- Integer expression in the header denotes the number of iterations that the loop performs.
- The colon (:) ends the loop header.
- The **loop body** comprises the statements in the remaining lines of code, below the header. Note that the statements in the loop body must be indented and aligned in the same column. These statements are executed in sequence on each pass through the loop.
- Example1:

```
>>> for eachPass in range(4):
        print("It's alive!", end=" ")

It's alive! It's alive! It's alive! It's alive!
>>>
```

- Example 2: Python's exponentiation operator might be implemented in a loop.

```
>>> number = 2
>>> exponent = 3
>>> product = 1
>>> for eachPass in range(exponent):
        product = product * number
        print(product, end = " ")

2 4 8
>>> product
8
```

- Example 3: When Python executes the type of for loop, it actually counts from 0 to the value of the header's integer expression minus 1.

```
>>> for count in range(4):
        print(count, end = " ")

0 1 2 3
>>>
```

- Loops that count through a range of numbers are also called **count controlled loops**. The value of the count on each pass is often used in computations.

  Example 4: factorial of 4

```
>>> product = 1
>>> for count in range(4):
        product = product * (count + 1)

>>> product
24
```

- To count from an explicit lower bound, the programmer can supply a second integer expression in the loop header. When two arguments are supplied to range, the count ranges from the first argument to the second argument minus 1.
- The next code segment uses this variation to simplify the code in the loop body:

```
>>> product = 1
>>> for count in range(1, 5):
        product = product * count

>>> product
24
>>>
```

- Here is an example of a summation, which accumulates the sum of a sequence of numbers from a lower bound through an upper bound:

```
>>> lower = int(input("Enter the lower bound: "))
Enter the lower bound: 1
>>> upper = int(input("Enter the upper bound: "))
Enter the upper bound: 10
>>> sum = 0
>>> for count in range(lower, upper + 1):
        sum = sum + count

>>> sum
55
>>>
```

- Some programs might want a loop to skip some numbers
- **A variant of Python's range function expects a third argument that allows you to nicely skip some numbers**. The third argument specifies a step value, or the interval between the numbers used in the range, as shown in the examples that follow:
- Example: compute the sum of the even numbers between 1 and 10.

```
>>> sum = 0
>>> for count in range(2, 11, 2):
        sum += count

>>> sum
30
>>>
```


Loop Errors: Off-by-One Error

- The loop fails to perform the expected number of iterations. Because this number is typically off by one, the error is called an off-by-one error.
- For the most part, off-by-one errors result when the programmer incorrectly specifies the upper bound of the loop.
- The programmer might intend the following loop to count from 1 through 4, but it counts from 1 through 3:

```
# Count from 1 through 4, we think
>>> for count in range(1,4):
        print(count) 1 2 3
```

Loops That Count Down

- All of our loops until now have counted up from a lower bound to an upper bound. Once in a while, a problem calls for counting in the opposite direction, from the upper bound down to the lower bound.
- Example:

```
>>> for count in range(10, 0, -1):
print(count, end = " ")
10 9 8 7 6 5 4 3 2 1
```

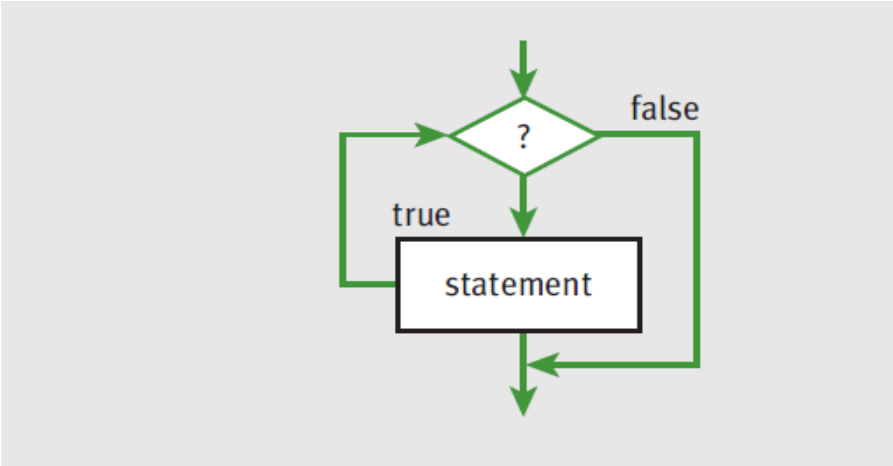
Conditional Iteration: The while Loop

- Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue. Such a condition is called the loop's continuation condition.
- If the continuation condition is false, the loop ends. If the continuation condition is true, the statements within the loop are executed again.
- The while loop is this type of control logic.
- Here is its syntax:

```
while <condition>:
    <sequence of statements>
```

- Clearly, something eventually has to happen within the body of the loop to make the loop's continuation condition become false. Otherwise, the loop will continue forever, an error known as an infinite loop. At least one

statement in the body of the loop must update a variable that affects the value of the condition.



The semantics of a **while** loop

- The while loop is also called an entry-control loop, because its condition is tested at the top of the loop. This implies that the statements within the loop can execute zero or more times.

The while True Loop and the break Statement

```
theSum = 0.0
while True:
    data = input("Enter a number or just enter to quit: ")
    if data == "":
        break
    number = float(data)
```

```
        theSum += number

    print("The sum is", theSum)
```

- The first thing to note is that the loop's entry condition is the Boolean value True.
- Within this body, the input datum is received. It is then tested for the loop's termination condition in a one-way selection statement. If the user wants to quit, the input will equal the empty string, and the **break** statement will cause an exit from the loop.
- Otherwise, control continues beyond the selection statement to the next two statements that process the input.

Selection Structure: if and if-else Statements

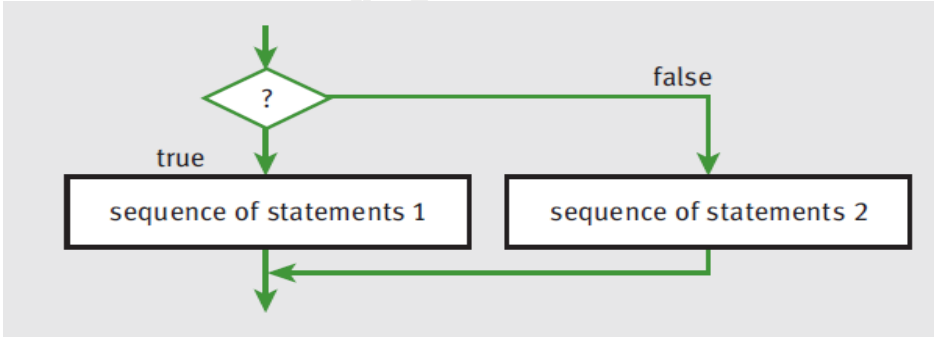
- In some cases, instead of moving straight ahead to execute the next instruction, the computer might be faced with two alternative courses of action.
- The computer must pause to examine or test a **condition.**
- The condition in a selection statement often takes the form of a comparison. The result of the comparison is a Boolean value **True** or **False.**

If-else Statements:

- The if-else statement is the most common type of selection statement.
- It is also called a two-way selection statement, because it directs the computer to make a choice between two alternative courses of action.
- Here is the Python syntax for the **if-else** statement:

```
if <condition>:
    <sequence of statements-1>
else:
    <sequence of statements-2>
```

- The condition in the **if-else** statement must be a Boolean expression—that is, an expression that evaluates to either true or false.
- The two possible actions each consist of a sequence of statements.
- Note that each sequence must be indented at least one space beyond the symbols **if** and **else**.
- Lastly, note the use of the colon (**:**) following the condition and the word **else**.
- A flow diagram of the semantics of the **if-else** statement is given below:



Example for if-else statement:

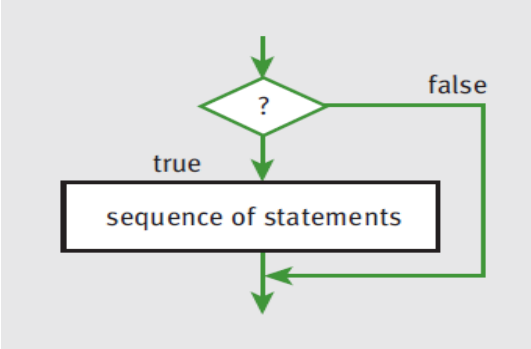
```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first > second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
print("Maximum:", maximum)
print("Minimum:", minimum)
```

If statement:

- This type of control statement is also called a one-way selection statement, because it consists of a condition and just a single sequence of statements.
- If the condition is True, the sequence of statements is run. Otherwise, control proceeds to the next statement following the entire selection statement.
- Here is the syntax for the if statement:

```
if <condition>:
    <sequence of statements>
```

- A flow diagram of the semantics of the if statement is given below:



Multi-Way if Statements:

- Occasionally, a program is faced with testing several conditions that entail more than two alternative courses of action.
- The process of testing several conditions and responding accordingly can be described in code by a **multi-way selection statement**.
- The multi-way **if** statement considers each condition until one evaluates to **True** or they all evaluate to **False**.

- The syntax of the multi-way if statement is the following:

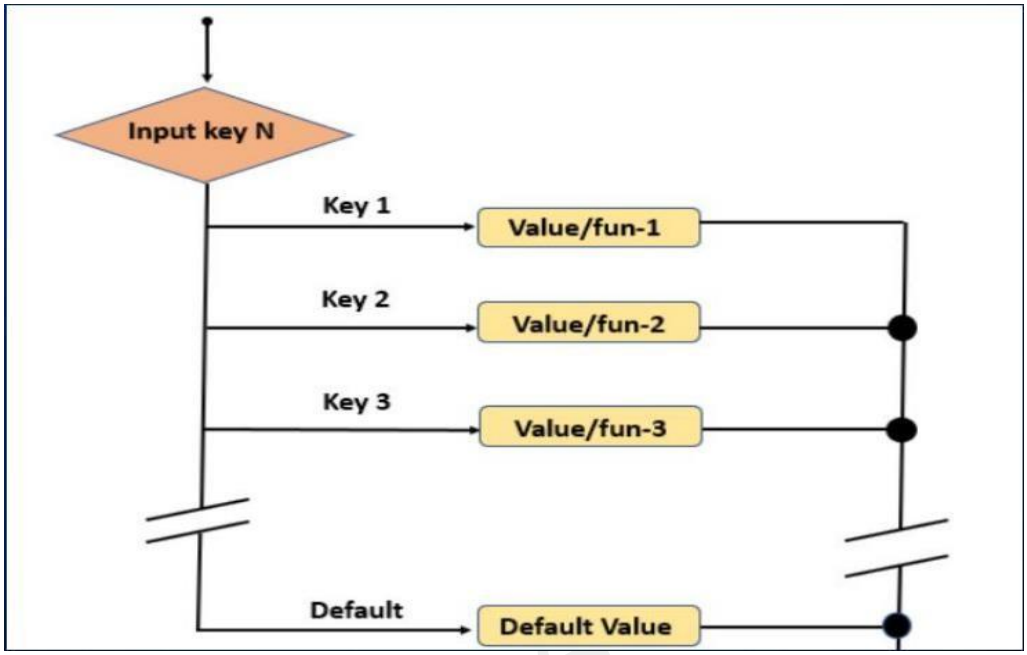
```
if <condition-1>:
    <sequence of statements-1>

elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

- Example:

```
number = int(input("Enter the numeric grade: "))
if number > 89:
    letter = 'A'
elif number > 79:
    letter = 'B'
elif number > 69:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

Switch statements

- A switch statement is a very useful and powerful programming feature. It is an alternate to if-else-if ladder statement and provides better performance and more manageable code than an if-else-if ladder statement.
- Python language does not provide any inbuilt switch statements. We can implement this feature with the same flow and functionality but with different syntax and implementation using Python Dictionary.



Syntax of Switch Statement

```
switcher=
{
        key_1: value_1/method_1(),
        key_2: value_2/method_2(),
        key_3: value_3/method_3(),
        ::
        key_n: value_n/method_n(),
}
```

```
key = N
value = switcher.get(key, "default")
```

Implementation of Switch statement in Python

```
def get_week_day(argument):
switcher =
{
 0: "Sunday" ,
 1: "Monday" ,
 2: "Tuesday" ,
 3: "Wednesday" ,
 4: "Thursday" ,
 5: "Friday" ,
 6: "Saturday"
}
return switcher.get(argument, "Invalid day")
print (get_week_day(6))
print (get_week_day(8))
print (get_week_day(0))
```

Augmented Assignment

- The assignment symbol can be combined with the arithmetic and concatenation operators to provide **augmented assignment operations**.

<variable> <operator>= <expression>

- Examples:

```
a = 17
s = "hi"
a += 3          # Equivalent to a = a + 3
a -= 3          # Equivalent to a = a - 3
a *= 3          # Equivalent to a = a * 3
a /= 3          # Equivalent to a = a / 3
a %= 3          # Equivalent to a = a % 3
s += " there"   # Equivalent to s = s + " there"
```

Formatting Text for Output

- Many data-processing applications require output that has a **tabular format**. In this format, numbers and other information are aligned in columns that can be either left-justified or right-justified.
- A column of data is **left-justified** if its values are vertically aligned beginning with their leftmost characters.
- A column of data is **right-justified** if its values are vertically aligned beginning with their rightmost characters.

- A column of data is centered if there are an equal number of spaces on either side of the data within that column.
- The total number of data characters and additional spaces for a given datum in a formatted string is called its **field width**.
- The **print** function automatically begins printing an output datum in the first available column.
- The next example, which displays the exponents 7 through 10 and the values of 10^7 through 10^{10}, shows the format of two columns produced by the **print** function:

```
>>> for exponent in range(7, 11):
        print(exponent, 10 ** exponent)

7 10000000
8 100000000
9 1000000000
10 10000000000
>>>
```

- Python includes a general formatting mechanism that allows the programmer to specify field widths for different types of data.
- The next session shows how to right-justify and left-justify the string **four** within a field width of 6:

```
>>> "%6s" % "four"        # Right justify
'  four'
>>> "%-6s" % "four"       # Left justify
'four  '
```

- The first line of code right-justifies the string by padding it with two spaces to its left. The next line of code left-justifies by placing two spaces to the string's right.
- The simplest form of this operation is the following:

```
<format string> % <datum>
```

- This version contains a **format string**, the **format operator %**, and a single data value to be formatted.
- The format string can contain string data and other information about the format of the datum.
- To format the string data value in our example, we used the notation **%<*field width*>s** in the format string.
    - When the field width is positive, the datum is right-justified; when the fieldwidth is negative, you get left-justification.
    - If the field width is less than or equal to the datum's print length in characters, no justification is added.
    - The **%** operator works with this information to build and return a formatted string.
    - To format integers, you use the letter **d** instead of **s**.
- To format a sequence of data values, you construct a format string that includes a format code for each datum and place the data values in a tuple following the **%** operator.

```
<format string> % (<datum-1>, …, <datum-n>)
```

- Example:

```
>>> for exponent in range(7, 11):
        print("%-3d%12d" % (exponent, 10 ** exponent))

7      10000000
8     100000000
9    1000000000
10  10000000000
```

- The format information for a data value of type **float** has the form:

```
%<field width>.<precision>f
```

- Here ***.<precision>*** is optional. The next session shows the output of a floating-point number without, and then with, a format string:

```
>>> "%6.3f" % 3.14
' 3.140'
```

Lazy evaluation

When using any programming language, it's important to understand when expressions are evaluated. Consider the simple expression:

a = b = c = 5

d = a + b * c

In Python, once these statements are evaluated, the calculation is immediately (or strictly) carried out, setting the value of *d* to 30. In another programming paradigm, such as in a pure functional programming language like Haskell, the value of *d* might not be evaluated until it is actually used elsewhere. The idea of deferring computations in this way is commonly known as **lazy evaluation**. Python, on the other hand, is a very strict (or eager) language. Nearly all of the time, computations and expressions are evaluated immediately. Even in the above simple expression, the result of *b* * *c* is computed as a separate step before adding it to *a*. There are Python techniques, especially using iterators and generators, which can be used to achieve laziness. When performing very expensive computations which are only necessary some of the time, this can be an important technique in data- intensive applications.

Short-Circuit Evaluation

- The Python virtual machine sometimes knows the value of a Boolean expression before it has evaluated all of its operands.
- For instance, in the expression A and B, if A is false, there is no need to evaluate B. Likewise, in the expression A or B, if A is true, there is no need to evaluate B.
- This approach, in which evaluation stops as soon as possible, is called **short-circuit evaluation**.

```
count = int(input("Enter the count: "))
sum = int(input("Enter the sum: "))
if count > 0 and sum // count > 10:
    print("average > 10")
```

```
else:
        print("count = 0 or average <=10")
```

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

Arithmetic Operators

a = 5
b = 2

Operator	Meaning	Example	Result
+	Addition Operator. Adds two Values.	a + b	7
-	Subtraction Operator. Subtracts one value from another	a – b	3
*	Multiplication Operator. Multiples values on either side of the operator	a * b	10
/	Division Operator. Divides left operand by the right operand.	a / b	2.5
%	Modulus Operator. Gives reminder of division	a % b	1
**	Exponent Operator. Calculates exponential power value. a ** b gives the value of a to the power of b	a ** b	25
//	Integer division, also called floor division. Performs division and gives only integer quotient.	a // b	2

Assignment Operators

x = 3
y = 2
z = 5

Operator	Example	Meaning	Result
=	z = x + y	Assignment operator. Stores right side value into left side variable.	z = 5
+=	z+=x	Addition assignment operator. Adds right operand to the left operand and stores the result into left operand.	z = 8
-=	z-=x	Subtraction assignment operator. subtracts right operand to the left operand and stores the result into left operand.	z = 2
*=	z*=x	Multiplication assignment operator. Multiplies right operand to the left operand and stores the result into left operand.	z = 15
/=	z/=y	Division assignment operator. Divides left operand with the right operand and stores the result into left operand.	z = 2.5
%=	z%=x	Modulus assignment operator. Divides left operand to the right operand and stores the result into left operand.	2
**=	z**=y	Exponential assignment operator. Performs power value and then stores the result into left operands	25
//=	z//=y	Floor division assignment operator. Performs floor division and then stores	2

Comparison Operator

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

Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

Logical Operators

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

Identity Operators

- Used to compare the objects.

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

Membership Operators

- Used to test if a sequence is presented in an object.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Bitwise Operators

- Bitwise Operators are used to compare (binary) numbers.

Name	Symbol	Function
Bitwise AND	&	Sets all bits to 1, if all bits are 1.
Bitwise OR	|	Sets all bits to 1, if one of the bits is 1.
Bitwise XOR	^	Sets all bits to 1, if only one of the bits is 1.
Bitwise NOT	~	Inverts al bits
Left Shift	<<	Shift left by pushing in zeroes from the right and let the leftmost bits fell off
Right Shift	>>	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bit fall off.

Types of Bitwise Operators

Operator	Name	Example	Result
&	Bitwise AND	6 & 3	2
|	Bitwise OR	10 | 10	10
^	Bitwise XOR	2^2	0
~	Bitwise 1's complement	~9	-10
<<	Left-Shift	10<<2	40
>>	Right-Shift	10>>2	2