

# Python Programming

## BOOKS:

1. Kenneth Lambert - “Fundamentals of Python\_ Data Structures”, *Cengage Learning PTR* (2013).
2. Gowrishankar S, Veena A, “Introduction to Python Programming”, 1st Edition, *CRC Press/Taylor & Francis*, 2018. ISBN-13: 978-0815394372.
3. Mark Lutz, “Programming Python”, 4th Edition, *O’Reilly Media*, 2011. ISBN-13: 978-9350232873.
4. Zed A. Shaw, ““Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code”, Addison-Wesley Professional, Year: 2017, ISBN: 0134692888, 9780134692883.

## REFERENCE MATERIALS:

1. Cody Jackson ,“Learning to Program using Python”, Second Edition, 2014.
2. Michael DAWSON, ”Python Programming”,3rd Edition, Course technology PTR, 2010
3. John V Guttag. "Introduction to Computation and Programming Using Python", Prentice Hall of India
4. Kent D. Lee, Steve Hubbard, "Data Structures and Algorithms with Python, Springer, 2015
5. Charles R. Severance, “Python for Everybody: Exploring Data Using Python 3”, 1st Edition, Create Space Independent Publishing Platform, 2016. [http://do1.drchuck.com/pythonlearn/EN\\_us/pythonlearn.pdf](http://do1.drchuck.com/pythonlearn/EN_us/pythonlearn.pdf)
6. Allen B. Downey, “Think Python: How to Think Like a Computer Scientist”, 2nd Edition, Green Tea Press, 2015.(<http://greenteapress.com/thinkpython2/thinkpython2.pdf>)

# Online Resources

1. <https://nptel.ac.in/courses/106/106/106106182/>
2. <https://nptel.ac.in/courses/115/104/115104095/>
3. <https://www.edx.org/learn/python>
4. <https://www.coursera.org/courses?query=python>
5. <https://www.udemy.com/topic/python/>
6. <https://online-learning.harvard.edu/subject/python>
7. <https://www.codecademy.com/learn/learn-python>
8. <https://www.geeksforgeeks.org/python-programming-language/>
9. <https://www.lynda.com/Python-training-tutorials/415-0.html>
10. <https://www.python.org/>

# Module - 1

- ▶ **Introduction to Python Programming:** History, Application of Python, Identifiers, Keywords, Statements and Expressions, Variables, Operators, Data Types, Type Conversions.
- ▶ **Control Flow Statements:** The if, if...else, if...elif...else, Decision Control Flow Statement, Nested if Statement, The while, For Loop, The continue and break Statements,
- ▶ **Functions:** Built-In Functions, Commonly Used Modules, Function Definition and Calling the Function, The return Statement and void Function,
- ▶ **Strings:** Basic String Operations, Accessing Characters in String by Index Number, String Slicing and Joining, String Methods.

# Introduction



**Guido van Rossum**

Author of the Python programming language

*Why the name Python??*

Inspired by the TV show

*The Complete Monty Python's Flying Circus*



# Python Versions

Python released on 20-02-1991

-Ver-0.9.9

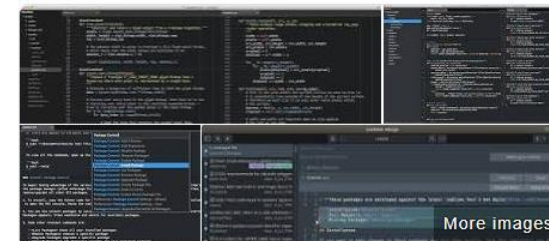
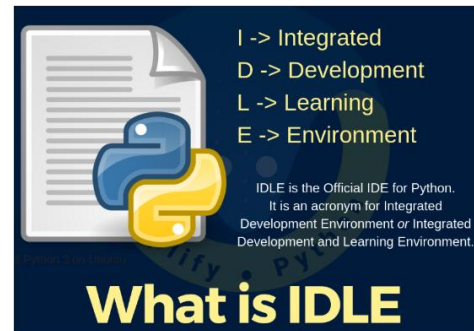
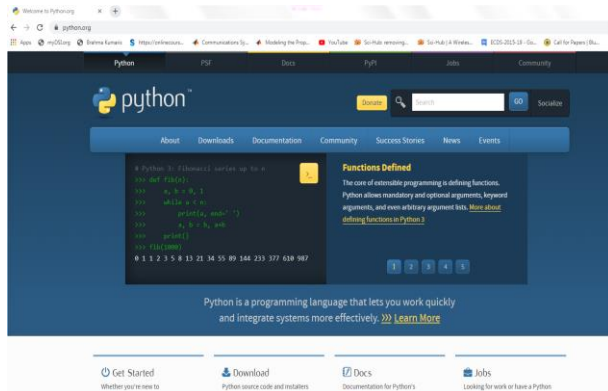
- ▶ **Python 1.0 - Jan 1994**
- ▶ Python 1.5 - 31 Dec 1997
- ▶ Python 1.5.2 - April 1999
- ▶ Python 1.6 - 05 Sep 2000
- ▶ **Python 2.0 - 16 Oct 2000**
- ▶ Python 2.0.1 - 22 Jun 2001
- ▶ Python 2.1 - 17 Apr 2001
- ▶ Python 2.2 - 21 Dec 2001
- ▶ Python 2.3 - 29 Jul 2003
- ▶ Python 2.4 - 30 Nov 2004
- ▶ Python 2.5 - 19 Sep 2006
- ▶ Python 2.6 - 01 Oct 2008
- ▶ Python 2.7 - 03 Jul 2010

- ▶ **Python 3.0 - 03 Dec 2008**
- ▶ Python 3.1 - 27 Jun 2009
- ▶ Python 3.2 - 20 Feb 2011
- ▶ Python 3.3 - 29 Sep 2012
- ▶ Python 3.4 - 16 Mar 2014
- ▶ Python 3.5 - 13 Sep 2015
- ▶ Python 3.6 - 23 Dec 2016
- ▶ Python 3.7 - 27 Jun 2018
- ▶ Python 3.8 - 14 Oct 2019
- Python 3.9 - 05 Oct 2020

[ May Not be backward compatible with 2.x]

# Introduction

- ▶ File Extension - py [Command Window Execution - py prog.py or python prog.py]
- ▶ More popularity because of its simplicity, Concise code, Applications Include Machine learning, Deep Learning, Artificial Intelligence, Neural Networks, Data Science, IoT etc..
- ▶ The **Python Software Foundation (PSF)** is a non-profit organization devoted to the Python programming language
- ▶ Python software and few IDEs: <https://www.python.org/>, Jupyter notebook, etc..



Sublime Text Python

<https://www.jetbrains.com/pycharm/>

```
C:\Users>py
Python 3.8.1 <tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24> [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```



# Introduction

- ▶ Python is an all rounder
  - ▶ Functional programming features from C
  - ▶ OOP features from C++
  - ▶ Scripting language features from Perl, Shell Script
    - Group of lines executed one by one
- ▶ Use of python:
  - ▶ Desktop Applications
  - ▶ Web Applications
  - ▶ Network Applications
  - ▶ Games Development
  - ▶ Data Analysis, Data Science
  - ▶ Machine Learning, Deep Learning, Neural Network, Artificial Language, IoT..



# Features of Python

- ▶ Simple and easy to learn
- ▶ Freeware and Open Source
- ▶ High level Programming language
- ▶ Platform independent, Portable
- ▶ Dynamically typed language
- ▶ Procedure and Object Oriented
- ▶ Interpreted
- ▶ Extensible and Embedded
- ▶ Extensive Library



# Limitations and Flavours of Python

## Limitations of Python

### ➤ Not suitable for

- ❑ Mobile applications
- ❑ Enterprise applications - Banking Application, Telecom Application-End to End Support
- ❑ Performance is low because of Interpreted nature

### ➤ Flavours of Python

- ❑ Free ware and open source-Customised python versions
  - ✓ C-Python- C-language applications
  - ✓ Jpython/ Jython- Java Applications
  - ✓ Iron Python- To work with C#, .net
  - ✓ Ruby Python- Ruby applications
  - ✓ Anaconda Python- Large volume data, ML, Data Science..
  - ✓ Stackless- Concurrent applications
  - ✓ PyPy- Python for speed - PVM+JIT(Just in Time) Compiler

<b>Paradigm</b>	Multi-paradigm: functional, imperative, object-oriented, reflective
<b>Designed by</b>	Guido van Rossum
<b>Developer</b>	Python Software Foundation
<b>First appeared</b>	1990; 30 years ago <sup>[1]</sup>
<b>Stable release</b>	3.8.1 / 18 December 2019; 33 days ago <sup>[2]</sup>
<b>Preview release</b>	3.9.0a2 / 18 December 2019; 33 days ago <sup>[3]</sup>
<b>Typing discipline</b>	Duck, dynamic, gradual (since 3.5) <sup>[4]</sup>
<b>License</b>	Python Software Foundation License
<b>Filename extensions</b>	.py, .pyi, .pyc, .pyd, .pyo (prior to 3.5), <sup>[6]</sup> .pyw, .pyz (since 3.5) <sup>[6]</sup>
<b>Website</b>	<a href="http://www.python.org">www.python.org</a> 
<b>Major implementations</b>	
CPython, PyPy, Stackless Python, MicroPython, CircuitPython, IronPython, Jython, RustPython	
<b>Dialects</b>	
Cython, RPython, Starlark <sup>[7]</sup>	
<b>Influenced by</b>	
ABC, <sup>[8]</sup> Ada <sup>[9]</sup> , ALGOL 68, <sup>[10]</sup> APL, <sup>[11]</sup> C, <sup>[12]</sup> C++, <sup>[13]</sup> CLU, <sup>[14]</sup> Dylan, <sup>[15]</sup> Haskell, <sup>[16]</sup> Icon, <sup>[17]</sup> Java, <sup>[18]</sup> Lisp, <sup>[19]</sup> Modula-3, <sup>[13]</sup> Perl, Standard ML <sup>[11]</sup>	
<b>Influenced</b>	
Apache Groovy, Boo, Cobra, CoffeeScript, <sup>[20]</sup> D, F#, Genie, <sup>[21]</sup> Go, JavaScript, <sup>[22][23]</sup> Julia, <sup>[24]</sup> Nim, Ring, <sup>[25]</sup> Ruby, <sup>[26]</sup> Swift <sup>[27]</sup>	
 Python Programming at Wikibooks	



# Identifiers

- An identifier is a name given to a variable, function, class or module.

Identifiers may be one or more characters in the following format:

- ❑ Identifiers can be a combination (a to z) or (A to Z) or (0 to 9) or an underscore (\_).
- ❑ A Python identifier can begin with an alphabet (A - Z and a - z and \_).
- ❑ An identifier cannot start with a digit but is allowed everywhere else.
- ❑ Keywords or reserved words cannot be used as identifiers.
- ❑ One cannot use spaces and special symbols like !, @, #, \$, % etc. as identifiers.
- ❑ Identifier can be of any length ( But not recommended to take lengthy variable)
- ❑ Case Sensitive

# Keywords

- ▶ Keywords are a list of reserved words that have predefined meaning.
- ▶ Keywords are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name.
- ▶ Attempting to use a keyword as an identifier name will cause an error.
- ▶ There 33 Keywords -
  - ❑ contains only alphabets
  - ❑ Except True, False, None- all contains only lowercase alphabet
  - ❑ Switch concept is not there in python
  - ❑ do-while is not there
  - ❑ Int, float, complex .. keywords are not there in python

## List of Keywords in Python

---

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

---

## 33 - Keywords

```
>> import keyword  
>> print(keyword.kwlist)
```

▶ False True None

▶ and not or is

▶ if elif else

▶ break for while continue return in yield

▶ try except finally raise assert

▶ Import from as class def pass

▶ nonlocal global del with lambda

- ▶ `True` and `False` are truth values in Python. They are the results of comparison operations or logical (Boolean) operations in Python
- ▶ `None` is a special constant in Python that represents the absence of a value or a `null` value.
- ▶ `and`, `or`, `not` are the logical operators in Python.
- ▶ `is` is used in Python for testing object identity
- ▶ `if`, `else`, `elif` are used for conditional branching or decision making
- ▶ `break` and `continue` are used inside for and while loops to alter their normal behaviour.
- ▶ `For`, `while` used for looping
- ▶ `return` statement is used inside a function to exit it and return a value.
- ▶ `yield` is used inside a function like a `return` statement
- ▶ `in` is used to test if a sequence (list, tuple, string etc.) contains a value. It returns `True` if the value is present, else it returns `False`
- ▶ `assert` is used for debugging purposes.
- ▶ `except`, `raise`, `try` are used with exceptions in Python.
- ▶ `finally` is used with `try...except` block to close up resources or file streams



- ▶ `from...import` is used to import specific attributes or `functions` into the current namespace
- ▶ `pass` is a null statement in Python. Nothing happens when it is executed. It is used as a placeholder.
- ▶ `as` is used to create an `alias` while importing a module. It means giving a different name (user-defined) to a module while importing it.
- ▶ `class` is used to define a new user-defined class in Python
- ▶ `def` is used to `define` a user-defined function
- ▶ `del` is used to `delete` the reference to an object.
- ▶ `global` is used to declare that a variable inside the function is global (outside the function).
- ▶ `lambda` is used to create an `anonymous function` (function with no name). It is an inline function that does not contain a return statement. It consists of an expression that is evaluated and returned.
- ▶ `with` statement is used to wrap the execution of a block of code within methods defined by the context manager.
- ▶ The use of `nonlocal keyword` is very much similar to the `global keyword`. `nonlocal` is used to declare that a variable inside a nested function (function inside a function) is not local to it, meaning it lies in the outer enclosing function
- ▶ The `async` and `await` keywords are provided by the `asyncio` library in Python. They are used to write `concurrent code` in Python

Keyword	Description
<u>and</u>	A logical operator
<u>as</u>	To create an alias
<u>assert</u>	For debugging
<u>break</u>	To break out of a loop
<u>class</u>	To define a class
<u>continue</u>	To continue to the next iteration of a loop
<u>def</u>	To define a function
<u>del</u>	To delete an object
<u>elif</u>	Used in conditional statements, same as else if
<u>else</u>	Used in conditional statements
<u>except</u>	Used with exceptions, what to do when an exception occurs
<u>False</u>	Boolean value, result of comparison operations
<u>finally</u>	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
<u>for</u>	To create a for loop
<u>from</u>	To import specific parts of a module
<u>global</u>	To declare a global variable
<u>if</u>	To make a conditional statement

<u>import</u>	To import a module
<u>in</u>	To check if a value is present in a list, tuple, etc.
<u>is</u>	To test if two variables are equal
<u>lambda</u>	To create an anonymous function
<u>None</u>	Represents a null value
<u>nonlocal</u>	To declare a non-local variable
<u>not</u>	A logical operator
<u>or</u>	A logical operator
<u>pass</u>	A null statement, a statement that will do nothing
<u>raise</u>	To raise an exception
<u>return</u>	To exit a function and return a value
<u>True</u>	Boolean value, result of comparison operations
<u>try</u>	To make a try...except statement
<u>while</u>	To create a while loop
with	Used to simplify exception handling
yield	To end a function, returns a generator

# Statements and Expressions

- ▶ A statement is an instruction that the Python interpreter can execute. Python program consists of a sequence of statements.

**Ex: `z = 1` is an assignment statement.**

- ▶ Expression is an arrangement of values and operators which are evaluated to make a new value. Expressions are statements as well.

EX: `>>> 20` or `>>> z` or `>>> z + 20`

- ▶ A value is the representation of some entity like a letter or a number that can be manipulated by a program.
- ▶ `>>> 8 + 2`

10

# Variables

- ▶ Variable is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution.
- ▶ In Python, there is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type.
- ▶ *To define a new variable in Python, we simply assign a value to a name.*
- ▶ If a need for variable arises you need to think of a variable name based on the rules mentioned in the following subsection and use it in the program.

# Legal Variable Names

Follow the below-mentioned rules for creating legal variable names in Python.

- Variable names can consist of any number of letters, underscores and digits.
- Variable should not start with a number.
- Python Keywords are not allowed as variable names.
- Variable names are case-sensitive. For example, computer and Computer are different variables.
- Ensure variable names are descriptive and clear enough. This allows other programmers to have an idea about what the variable is representing.

# Assigning Values to Variables

- ▶ The general format for assigning values to variables is as follows:

*variable\_name = expression*

```
number = 100
```

```
>>> print(number)
```

```
100
```

```
>>> print(type(number))
```

```
<class 'int'>
```

```
>>> number = 100.0
```

```
>>> print(type(number))
```

```
<class 'float'>
```

```
>>> number = " one two"
```

```
>>> print(number)
```

```
one two
```

```
>>> print(type(number))
```

```
<class 'str'>
```



# Operators

- ▶ Operators are symbols, such as +, -, =, >, and <, that perform certain mathematical or logical operation to manipulate data values and produce a result based on some rules.
- ▶ An operator manipulates the data values called operands.
- ▶ Python language supports a wide range of operators. They are
  1. Arithmetic Operators
  2. Assignment Operators
  3. Comparison Operators
  4. Logical Operators
  5. Bitwise Operators

## List of Arithmetic Operators

Operator	Operator Name	Description	Example
+	Addition operator	Adds two operands, producing their sum.	$p + q = 5$
-	Subtraction operator	Subtracts the two operands, producing their difference.	$p - q = -1$
*	Multiplication operator	Produces the product of the operands.	$p * q = 6$
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	$q / p = 1.5$
%	Modulus operator	Divides left hand operand by right hand operand and returns a remainder.	$q \% p = 1$
**	Exponent operator	Performs exponential (power) calculation on operators.	$p^{**}q = 8$
//	Floor division operator	Returns the integral part of the quotient.	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$

*Note:* The value of p is 2 and q is 3.

# Assignment Operators

- ▶ **Assignment operators** are used for **assigning the values** generated after evaluating the right operand to the left operand. Assignment operation always works from **right to left**.
- ▶ Assignment operators are either simple assignment operator or compound assignment operators. Simple assignment is done with the equal sign (=) and simply assigns the value of its right operand to the variable on the left.

```
>>> x=5
```

```
>>> x=x+1
```

```
>>> x
```

```
6
```

```
>>> x+=1
```

```
>>> x
```

```
7
```

## List of Assignment Operators

Operator	Operator Name	Description	Example
=	Assignment	Assigns values from right side operands to left side operand.	$z = p + q$ assigns value of $p + q$ to $z$
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand.	$z += p$ is equivalent to $z = z + p$
-=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand.	$z -= p$ is equivalent to $z = z - p$
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand.	$z *= p$ is equivalent to $z = z * p$
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand.	$z /= p$ is equivalent to $z = z / p$
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand.	$z **= p$ is equivalent to $z = z ** p$
//=	Floor Division Assignment	Produces the integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	$z //= p$ is equivalent to $z = z // p$
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	$z \% = p$ is equivalent to $z = z \% p$

Python programming language doesn't support Auto increment (+ +) and Auto decrement (- -) operators.

# Comparison Operators

- ▶ When the values of two operands are to be compared then comparison operators are used.
- ▶ The output of these comparison operators is always a Boolean value, either True or False.
- ▶ The operands can be Numbers or Strings or Boolean values.
- ▶ Strings are compared letter by letter using their ASCII values, thus, “P” is less than “Q”, and “Aston” is greater than “Asher”.

## List of Comparison Operators

Operator	Operator Name	Description	Example
==	Equal to	If the values of two operands are equal, then the condition becomes True.	(p == q) is not True.
!=	Not Equal to	If values of two operands are not equal, then the condition becomes True.	(p != q) is True
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes True.	(p > q) is not True.
<	Lesser than	If the value of left operand is less than the value of right operand, then condition becomes True.	(p < q) is True.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes True.	(p >= q) is not True.
<=	Lesser than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes True.	(p <= q) is True.

*Note:* The value of p is 10 and q is 20.

1. >>>10 == 12

False

2. >>>10 != 12

3. >>>10 < 12

4. >>>10 > 12

5. >>>10 <= 12

6. >>>10 >= 12

7. >>> "P" < "Q"

8. >>> "Aston" > "Asher"

9. >>> True == True



1. >>>10 == 12

False

2. >>>10 != 12

True

3. >>>10 < 12

True

4. >>>10 > 12

False

5. >>>10 <= 12

True

6. >>>10 >= 12

False

7. >>> "P" < "Q"

True

8. >>> "Aston" > "Asher"

True

9. >>> True == True

True

# Logical Operators

- ▶ The logical operators are used for comparing or negating the logical values of their operands and to return the resulting logical value.
- ▶ The values of the operands on which the logical operators operate evaluate to either True or False. The result of the logical operator is always a Boolean value

List of Logical Operators

Operator	Operator Name	Description	Example
and	Logical AND	Performs AND operation and the result is True when both operands are True	p and q results in False
or	Logical OR	Performs OR operation and the result is True when any one of both operand is True	p or q results in True
not	Logical NOT	Reverses the operand state	not p results in False

Note: The Boolean value of p is True and q is False.

Boolean Logic Truth Table

P	Q	P and Q	P or Q	Not P
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

1. >>> True and False

False

2. >>> True or False

3. >>> not(True) and False

4. >>> not(True and False)

5. >>> (10 < 0) and (10 > 2)

6. >>> (10 < 0) or (10 > 2)

7. >>> not(10 < 0) or (10 > 2)

8. >>> not(10 < 0 or 10 > 2)

1. >>> True and False

False

2. >>> True or False

True

3. >>> not(True) and False

False

4. >>> not(True and False)

True

5. >>> (10 < 0) and (10 > 2)

False

6. >>> (10 < 0) or (10 > 2)

True

7. >>> not(10 < 0) or (10 > 2)

True

8. >>> not(10 < 0 or 10 > 2)

False

# Bitwise Operators

- ▶ Bitwise operators treat their operands as a sequence of bits (zeroes and ones) and perform bit by bit operation.
- ▶ For example, the decimal number ten has a binary representation of 1010. Bitwise operators perform their operations on such binary representations, but they return standard Python numerical values.

Bitwise Truth Table

P	Q	P & Q	P   Q	P ^ Q	~ P
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

### List of Bitwise Operators

Operator	Operator Name	Description
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands are 1s.
	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.
~	Binary Ones Complement	Inverts the bits of its operand.
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.

*Note:* The value of p is 60 and q is 13.

## List of Bitwise Operators

Operator	Operator Name	Description	Example
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands are 1s.	$p \& q = 12$ (means 0000 1100)
	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.	$p   q = 61$ (means 0011 1101)
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.	$(p \wedge q) = 49$ (means 0011 0001)
~	Binary Ones Complement	Inverts the bits of its operand.	$(\sim p) = -61$ (means 1100 0011 in 2's complement form due to a signed binary number.)
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	$p \ll 2 = 240$ (means 1111 0000)
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	$p \gg 2 = 15$ (means 0000 1111)

*Note:* The value of p is 60 and q is 13.



## Examples of bitwise logical operators.

Bitwise and ( & )		Bitwise or (   )	
a	= 0011 1100 → (60)	a	= 0011 1100 → (60)
b	= 0000 1101 → (13)	b	= 0000 1101 → (13)
<hr/> a & b = 0000 1100 → (12) <hr/>		<hr/> a   b = 0011 1101 → (61) <hr/>	
Bitwise exclusive or ( ^ )		One's Complement ( ~ )	
a	= 0011 1100 → (60)	a	= 0011 1100 → (60)
b	= 0000 1101 → (13)	<hr/> ~ a	= 1100 0011 → (-61) <hr/>
<hr/> a ^ b = 0011 0001 → (49) <hr/>			
Binary left shift ( << )		Binary right shift ( >> )	
a	= 0011 1100 → (60)	a	= 0011 1100 → (60)
<hr/> a << 2 = 1111 0000 → (240) <hr/>		<hr/> a >> 2 = 0000 1111 → (15) <hr/>	
left shift of 2 bits		right shift of 2 bits	

# Precedence and Associativity

- ▶ Operator precedence determines the way in which operators are parsed with respect to each other. Operators with higher precedence become the operands of operators with lower precedence.
- ▶ Associativity determines the way in which operators of the same precedence are parsed. Almost all the operators have left-to-right associativity.
- ▶ `>>> 2 + 3 * 6`
- ▶ `>>> (2 + 3) * 6`
- ▶ `>>> 6 * 4 / 2`

## Operator Precedence in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=,	Comparisons,
is, is not, in, not in	Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

# Data Types

- ▶ Data types specify the type of data like numbers and characters to be stored and manipulated within a program. Basic data types of Python are
  - ❑ Numbers
  - ❑ Boolean
  - ❑ Strings
  - ❑ None

## Numbers

- ▶ Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as `int`, `float` and `complex` class in Python.
- ▶ Integers can be of any length; it is only limited by the memory available.
- ▶ A floating point number is accurate up to 15 decimal places.
- ▶ Integer and floating points are separated by decimal points.
- ▶ Complex numbers are written in the form,  $x + yj$ , where  $x$  is the real part and  $y$  is the imaginary part.

# Data Types

## Boolean

- ▶ Booleans -useful in **conditional statements**. since a condition is really just a **yes-or-no** question, the answer to that question is a **Boolean value**, either **True** or **False**.
- ▶ The Boolean values, True and False are treated as **reserved words**.

## Strings

- ▶ A string consists of a **sequence of one or more characters**, which can include **letters**, **numbers**, and other types of **characters**.
- ▶ A string can also contain **spaces**.
- ▶ You can use **single quotes** or **double quotes** to represent strings and it is also called a **string literal**.
- ▶ **Multiline strings** can be denoted using **triple quotes**, `'''` or `"""`. These are fixed values, not variables that you literally provide in your script.

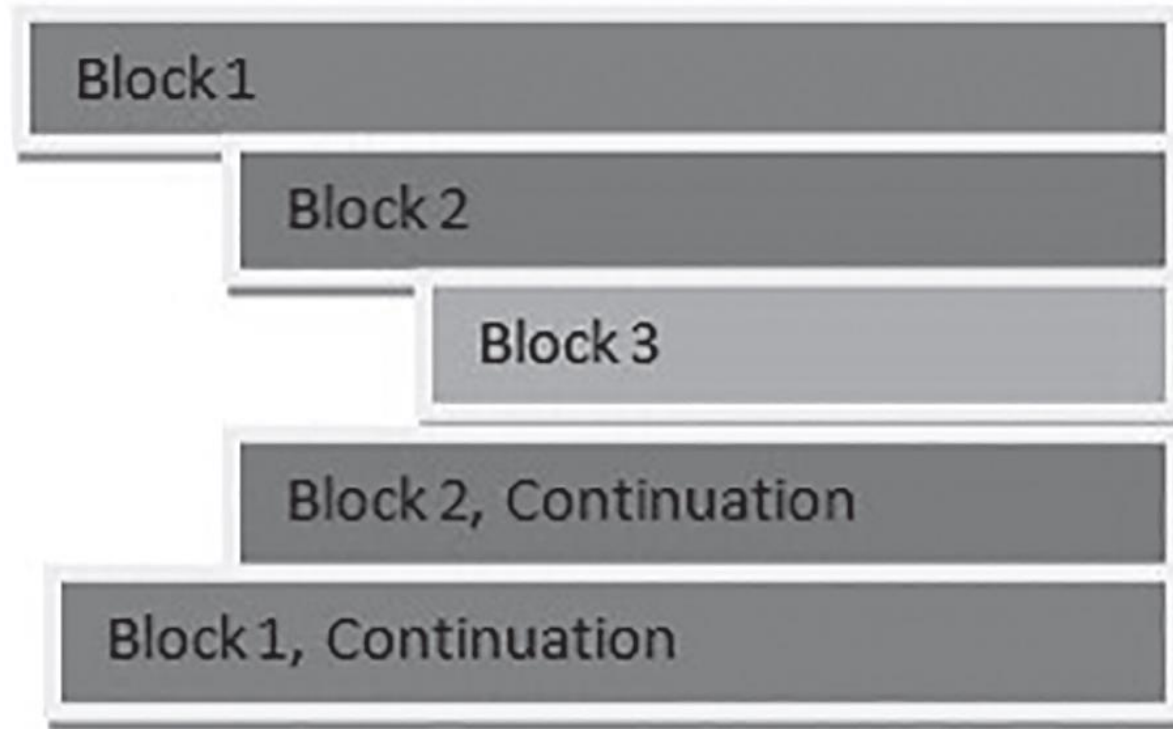
## Example

- ❑ `print('ECE')`
- ❑ `print("ENGG")`
- ❑ `print("ECE BENGALURU")`
- ❑ `s= (" ECE  
ENGG  
Bengaluru ")`
- ❑ `print(s)`
- ❑ `print(" ECE 'ENGG' Bengaluru")`

## None

- ▶ `None` is another special data type in Python.  
`None` is frequently used to represent the **absence of a value**.

# Indentation



Code blocks and indentation in Python.

# Comments

- ❖ Comments are an important part of any program.
- ❖ A comment is a text that describes what the program or a particular part of the program is trying to do and is ignored by the Python interpreter.
- ❖ Comments are used to help you and other programmers understand, maintain, and debug the program.
- ❖ Python uses two types of comments: single-line comment

`#This is single line Python comment`

- ❖ multiline comments.

`#This is`

`#multiline comments`

`#in Python`



# Reading Input

- ❑ In Python, **input()** function is used to gather data from the user.

The syntax for input function is,

***variable\_name = input([prompt])***

```
>>> person = input("What is your name?")
```

```
What is your name? Carrey
```

```
>>> person
```

```
'Carrey'
```

# Print Output

- ▶ The *print()* function allows a program to display text onto the console.

```
print("Hello World!!")
```

Hello World!!

- ▶ Two major string formats which are used inside the *print()* function to display the contents onto the console

1. *str.format()*

2. f-strings

# str.format() Method

- ▶ Use `str.format()` method if you need to insert the value of a variable, expression or an object into another string and display it to the user as a single string.
- ▶ The syntax for `format()` method is, `str.format(p0, p1, ..., k0=v0, k1=v1, ...)`  
`p0, p1,...` are called as positional arguments and, `k0, k1,...` are keyword arguments with their assigned values of `v0, v1,...` respectively.

## Example:

```
USN = input("Enter your USN: ")
```

```
Name = input("Enter your Name: ")
```


```
print("Student USN is {0} and name {1}".format(USN, Name))
```

```
print("Student name is {1} and USN is {0}".format(USN, Name))
```

```
print("Student belongs to {Section}, ECE".format(Section = "6ABCD"))
```

## str.format() Method

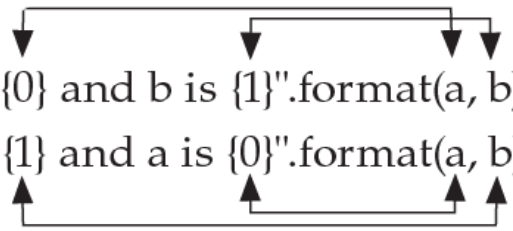
```
country = input("Which country do you live in?")  
print("I live in {0}".format(country))
```



### OUTPUT

```
Which country do you live in? India  
I live in India
```


```
a = 10  
b = 20  
print("The values of a is {0} and b is {1}".format(a, b))  
print("The values of b is {1} and a is {0}".format(a, b))
```



### OUTPUT

```
The values of a is 10 and b is 20  
The values of b is 20 and a is 10
```

```
>>> print("Give me {ball} ball".format(ball = "tennis"))  
Give me tennis ball
```



# f-strings

- ▶ Formatted strings or **f-strings** were introduced in **Python 3.6**.
- ▶ A **f-string** is a string literal that is prefixed with **"f"**. These strings may contain **replacement fields**, which are expressions enclosed within **curly braces { }**. The expressions are replaced with their **values**.

## Example:

```
USN = input("Enter your USN: ")
```

```
Name = input("Enter your Name: ")
```

```
print(f"Student USN {USN} and {Name}")
```

- Write a pgm to find area and circumference of circle given radius

```
radius= int(input("Enter radius"))
```

```
area_circle = 3.1415*radius*radius
```

```
circumference_circle=2*3.1415*radius
```

```
Print(f"Area={area_circle} and Circumference={circumference_circle}")
```

Write a pgm to convert number of days to measure of time given in years, weeks and days(Ignore leap year)

```
number_of_days=int(input("Enter number of days"))
```

```
number_of_years=int(number_of_days/365)
```

```
number_of_weeks=int(number_of_days%365/7)
```

```
remaining_days=int(number_of_days%365%7)
```

```
print(f "Years={number_of_years}, Weeks={number_of_weeks},  
Days={remaining_days}")
```

# Type Conversions

You can explicitly cast, or convert, a variable from one type to another.

The `int()` Function

The `float()` Function

The `str()` Function

The `complex()` Function

The `chr()` Function

The `ord()` Function

The `hex()` Function

The `oct()` Function

The `type()` Function and is Operator: `type(object)`

The `type()` function returns the data type of the given object

```
type(6.4)
```

```
<class 'float'>
```



*int()* Function : To convert from other types to int

**int(10.998)**

**int(10+20j)**

**int(True)**

**int(False)**

**int('15')**

**int('0b1111')**

**int('10.5')**

*float()* Function : To convert from other types to float

**float(15)**

**float(0b1111)**

**float(0xfa)**

**float(10+20j)**

**float(True)**

**float(False)**

**float('15')**

**float('0b1111')**

**float('10.5')**

*complex()* function : To convert from other types to complex

`complex(15)`

`complex(0b1111)`

`complex(10.5)`

`complex(True)`

`complex(False)`

`complex('15')`

`complex('0b1111')`

`complex('10.5')`

`complex(10,20)`

`complex(10.5,20.6)`

`complex("10", "20")`

`complex("10", 20)`

`complex(10, "20")`

***bool()* Function : To convert from other types to bool**

**bool(15)**

**bool(0b1111)**

**bool(0)**

**bool(0.0)**

**bool(0.1)**

**bool(0+0j)**

**bool(10+0j)**

**bool(0+10j)**

**bool('True')**

**bool('False')**

**bool('yes')**

**bool('no')**

**bool(' ')**

*str()* function : To convert from other types to string

**str(15)**

**str(0b1111)**

**str(10+20j)**

**str(True)**

**str(False)**

**str(10.5)**

*chr()* Function : To convert from other types to character

- This function converts the specified integer value into character
- Character system : ASCII and UNICODE
- chr(100)  
d
- chr(49)  
1

# ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

*ord()* function : To convert the specified characters into integer

- We have to pass single character
- `ord('A')`  
65
- `ord('4')`  
52
- `ord('ab')`  
Error

*hex()* function : To convert any other to hex prefixed with 0x

- `hex(255)`  
0xff
- `hex(0b1101)`  
d
- `hex(0o13)`  
0xb



*oct()* function : To convert any other to octal prefixed with 0o

- `oct(100)`  
`0o144`
- `oct(0b1101)`  
`0o15`
- `oct(0xfA)`  
`0o372`

*bin()* function : To convert any other to binary prefixed with 0b

- `bin(100)`  
`0b1100100`
- `bin(0o65)`  
`0b110101`
- `bin(0xfA)`  
`0b11111010`

# is , is not

- ▶ The operators *is* and *is not* are *identity operators*.
- ▶ Operator *is* evaluates to *True* if the values of operands on either side of the operator point to the same object and *False* otherwise.
- ▶ The operator *is not* evaluates to *False* if the values of operands on either side of the operator point to the same object and *True* otherwise.

```
>>> a=2
```

```
>>> b=2
```

```
>>> a is b
```

```
True
```

```
>>> a is not b
```

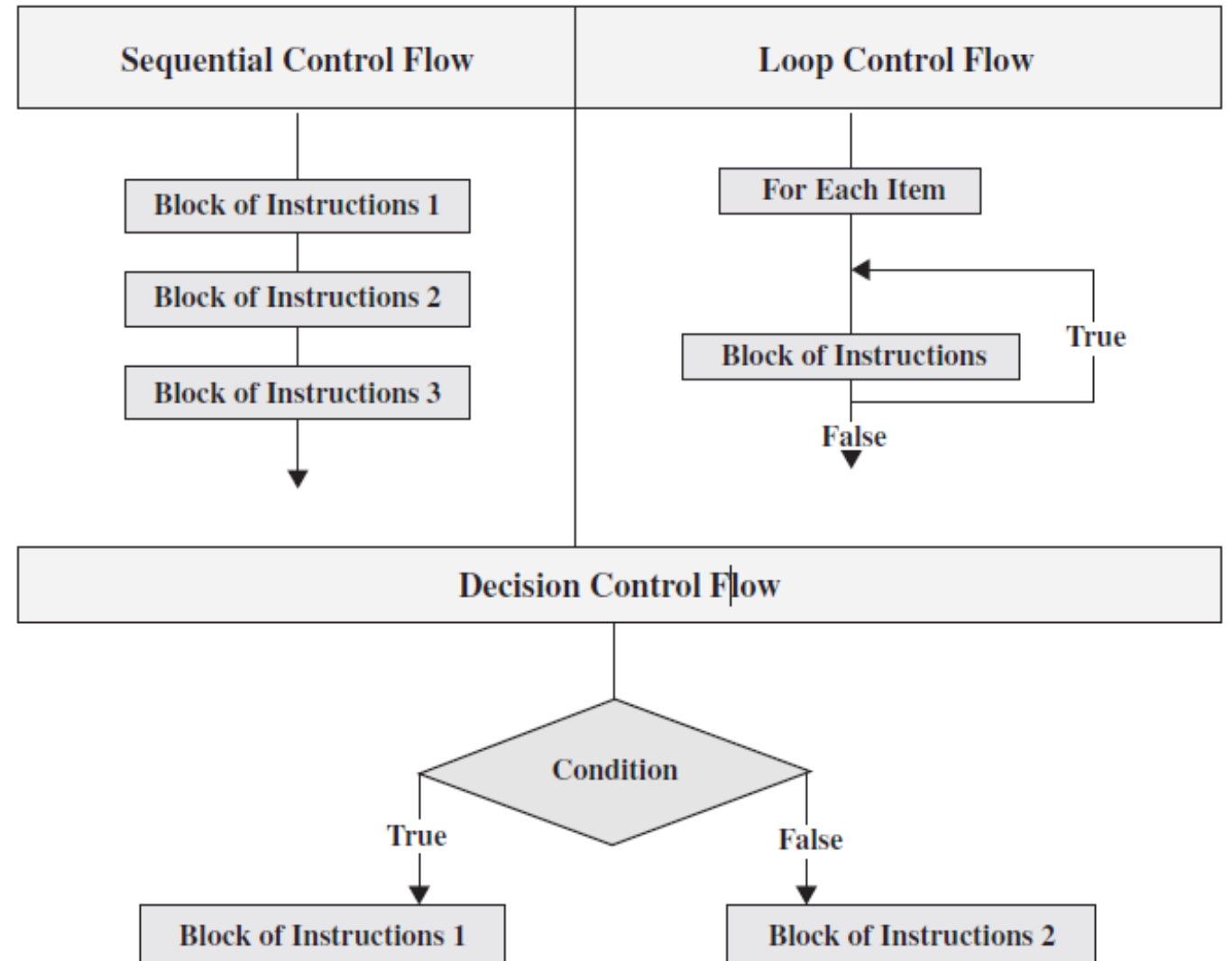
```
False
```

```
>>> c=4
```

```
>>> a is not c
```

```
True
```

## Control Flow Statements



**FIGURE 3.1**  
Forms of control flow statements.

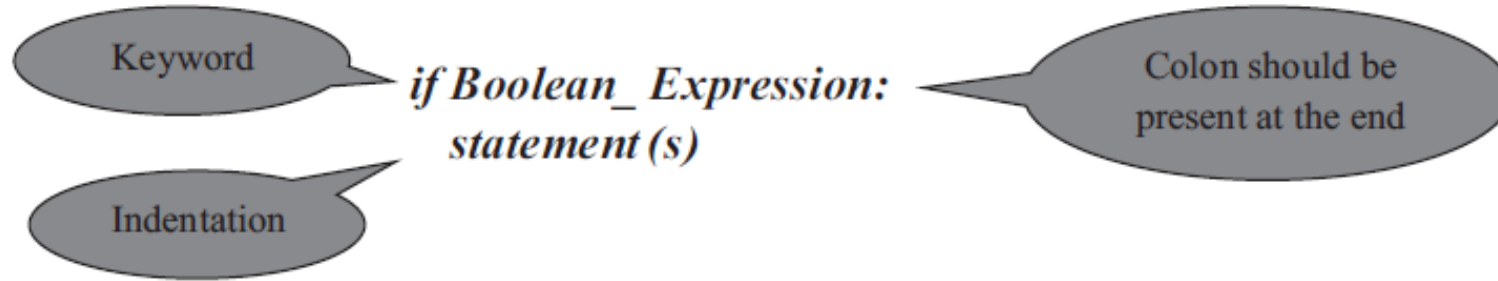
# control flow statements

The control flow statements in Python Programming Language are:

- ▶ 1. **Sequential Control Flow Statements:** This refers to the **line by line execution**, in which the statements are executed **sequentially**, in the same order in which they appear in the program.
- ▶ 2. **Decision Control Flow Statements:** Depending on whether a **condition** is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if...else and if...elif...else).
- ▶ 3. **Loop Control Flow Statements:** This is a control structure that allows the execution of a block of statements **multiple times** until a loop termination **condition is met** (*for* loop and *while* loop). Loop Control Flow Statements are also called **Repetition statements** or **Iteration statements**.

# The if Decision Control Flow Statement

The syntax for *if* statement is,



If the Boolean expression evaluates to *True* then statements in the *if* block will be executed; otherwise the result is *False* then none of the statements are executed.

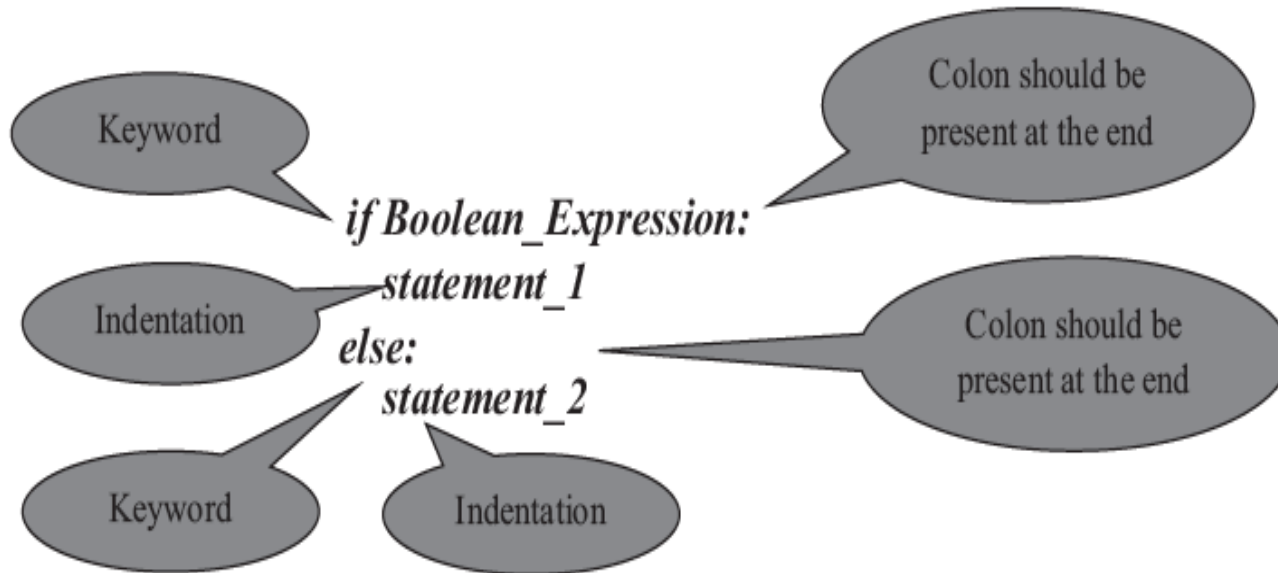
In Python, the *if* block statements are determined through indentation and the first unindented statement marks the end.

## Ex: Program Reads a Number and Prints a Message If It Is Positive

```
number = int(input("Enter a number"))  
if number >= 0:  
    print(f"The number entered by the user is a positive number")
```

# The if...else Decision Control Flow Statement

The syntax for *if...else* statement is,



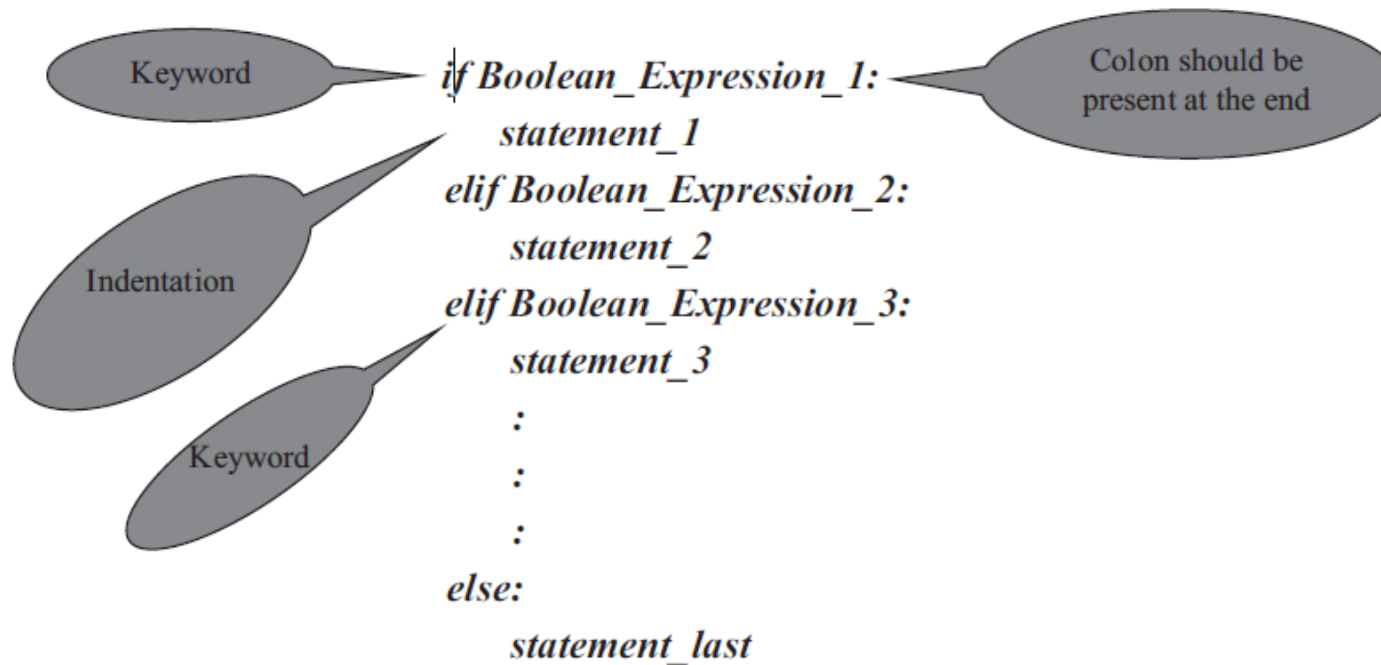
- If **Boolean\_Expression** evaluates to **True**, then **statement\_1 is executed**, otherwise it is evaluated to **False** then **statement\_2 is executed**.
- **Indentation** is used to **separate the blocks**.
- After the **execution** of either **statement\_1** or **statement\_2**, the control is transferred to the **next statement after the if statement**.
- Also, **if** and **else** keywords should be aligned at the **same column position**.

# Program to Find If a Given Number Is Odd or Even

```
number = int(input("Enter a number"))  
if number % 2 == 0:  
    print(f"{number} is Even number")  
else:  
    print(f"{number} is Odd number")
```

# The if...elif...else Decision Control Statement

The syntax for *if...elif...else* statement is,



- This *if...elif...else* decision control statement is executed as follows:
- In the case of multiple Boolean expressions, only the first logical Boolean expression which evaluates to *True* will be executed.
- If *Boolean\_Expression\_1* is *True*, then *statement\_1* is executed.
- If *Boolean\_Expression\_1* is *False* and *Boolean\_Expression\_2* is *True*, then *statement\_2* is executed.
- If *Boolean\_Expression\_1* and *Boolean\_Expression\_2* are *False* and *Boolean\_Expression\_3* is *True*, then *statement\_3* is executed and so on.
- If none of the Boolean\_Expression is *True*, then *statement\_last* is executed.



# Write a Program to Prompt for a Score between 0 and 100

## Score Grade:

$\geq 90$  S,

$\geq 80$  A,

$\geq 70$  B,

$\geq 60$  C,

$\geq 50$  D,

$\geq 40$  E,

$< 40$  F

Write a Program to Prompt for a Score between 0 and 100

Score Grade:  $\geq 90$  S,  $\geq 80$  A,  $\geq 70$  B,  $\geq 60$  C,  
 $\geq 50$  D,  $\geq 40$  E,  $< 40$  F

```
score = float(input("Enter your score"))
```

```
if score < 0 or score > 100:
```

```
    print('Wrong Input')
```

```
elif score  $\geq 90$ :
```

```
    print('Your Grade is "S" ')
```

```
elif score  $\geq 80$ :
```

```
    print('Your Grade is "A" ')
```

```
elif score  $\geq 70$ :
```

```
    print('Your Grade is "B" ')
```

```
elif score  $\geq 60$ :
```

```
    print('Your Grade is "C" ')
```

```
elif score  $\geq 50$ :
```

```
    print('Your Grade is "D" ')
```

```
elif score  $\geq 40$ :
```

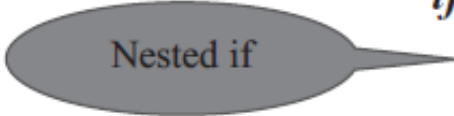
```
    print('Your Grade is "E" ')
```

```
else:
```

```
    print('Your Grade is "F" ')
```

# Nested if Statement

The syntax of the nested *if* statement is,



```
if Boolean_Expression_1:  
    if Boolean_Expression_2:  
        statement_1  
    else:  
        statement_2  
else:  
    statement_3
```

If the **Boolean\_Expression\_1** is evaluated to **True**, then the **control shifts** to **Boolean\_Expression\_2** and if the expression is evaluated to **True**, then **statement\_1** is executed, if the **Boolean\_Expression\_2** is evaluated to **False** then the **statement\_2** is executed. If the **Boolean\_Expression\_1** is evaluated to **False**, then **statement\_3** is executed.

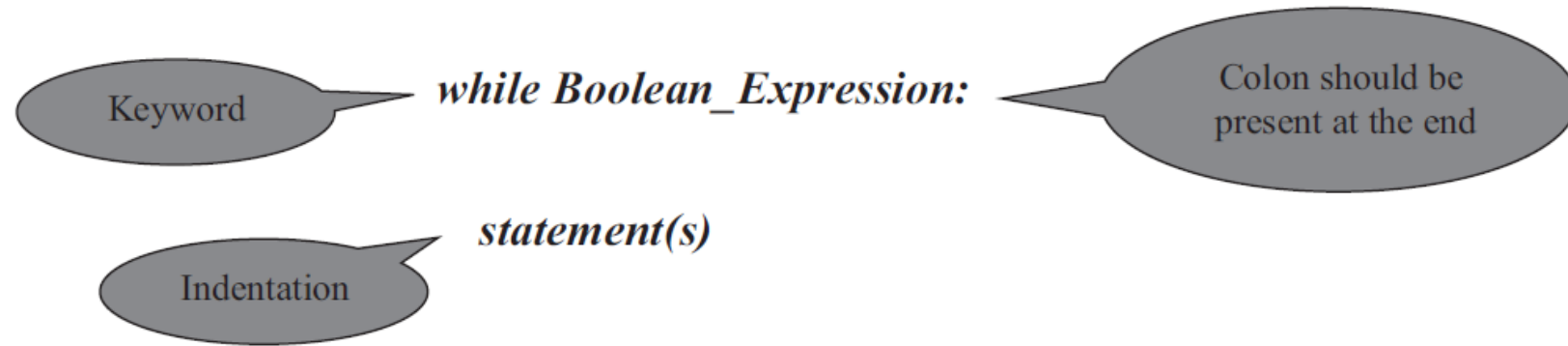
## Program to Check If a Given Year Is a Leap Year

```
year = int(input('Enter a year'))

if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print(f'{year} is a Leap Year')
        else:
            print(f'{year} is not a Leap Year')
    else:
        print(f'{year} is a Leap Year')
else:
    print(f'{year} is not a Leap Year')
```

# The while Loop

The syntax for *while* loop is,



With a **while** statement, the first thing that happens is that the **Boolean expression is evaluated** before the statements in the *while* loop block is executed.

If the **Boolean expression** evaluates to **False**, then the statements in the *while* loop block are **never executed**.

If the **Boolean expression** evaluates to **True**, then the *while* loop block is **executed**.

After **each iteration** of the loop block, the Boolean expression is again **checked**, and if it is **True**, the loop is **iterated again**.

Each **repetition** of the loop block is called an **iteration of the loop**.

This process continues until the **Boolean expression** evaluates to **False** and at this point the *while* statement **exits**. Execution then continues with the first statement after the *while* loop.

# Write Python Program to Display First 10 Numbers Using while Loop Starting from 0

```
i = 0
while i < 10:
    print(f"Current value of i is {i}")
    i = i + 1
```

# Write a Program to Find the Average of $n$ Natural Numbers Where $n$ Is the Input

```
n = int(input("Enter a number up to which you want to find the average "))
```

```
i = 0
```

```
sum = 0
```

```
count = 0
```

```
while i < n:
```

```
    i = i + 1
```

```
    sum = sum + i
```

```
    count = count + 1
```

```
    average = sum/count
```

```
print(f"The average of {n} natural numbers is {average}")
```

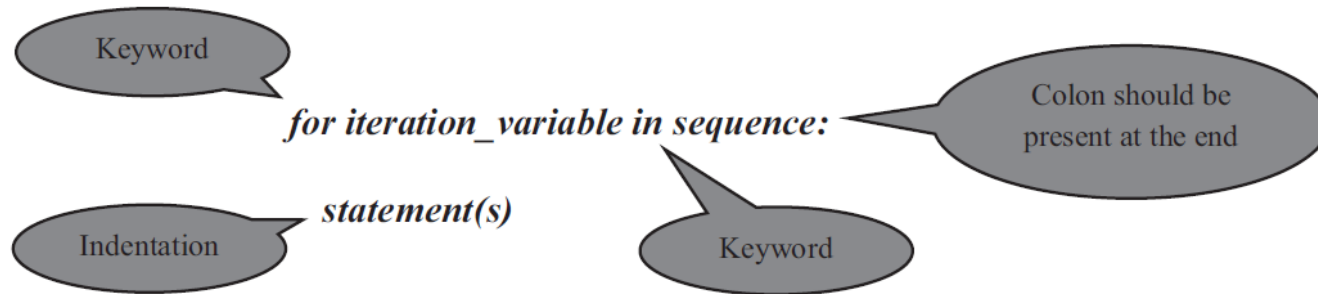
# Write Python Program to Find the Sum of Digits in a Number

```
number = int(input('Enter a number'))  
result = 0  
remainder = 0  
while number != 0:  
    remainder = number % 10  
    result = result + remainder  
    number = int(number / 10)  
print(f"The sum of all digits is {result}")
```



# The for Loop

The syntax for the *for* loop is,



The *for* loop starts with *for* keyword and ends with a colon.

The first item in the sequence gets assigned to the iteration variable *iteration\_variable*.

Here, *iteration\_variable* can be any valid variable name.

Then the *statement block is executed*.

This process of assigning items from the sequence to the *iteration\_variable* and then executing the statement continues until *all the items in the sequence are completed*.

# range() function

- ▶ The *range()* function generates a **sequence of numbers** which can be iterated through using *for* loop.
- ▶ The syntax for *range()* function is,  
*range([start ,] stop [, step])*

Both **start and step arguments are optional** and the range argument value should always be an integer.

- ▶ **start** → value indicates the **beginning** of the sequence. If the start argument is not specified, then the sequence of numbers start from **zero by default**.
- ▶ **stop** → Generates numbers **up to this value but not including the number** itself.
- ▶ **step** → indicates the **difference between every two consecutive numbers** in the sequence.  
The step value can be both **negative and positive** but not zero.
- ▶ **NOTE:** The **square brackets** in the syntax indicate that these **arguments are optional**. You can leave them out.

## Demonstrate for Loop Using range() Function

```
print("Only \"stop\" argument value specified in range function")
```

```
for i in range(9):
```

```
    print(f"{i}")
```

```
print("Both \"start\" and \"stop\" argument values specified in range function")
```

```
for i in range(2, 9):
```

```
    print(f"{i}")
```

```
print("All three arguments \"start\", \"stop\" and \"step\" specified in range  
function")
```

```
for i in range(2, 9, 3):
```

```
    print(f"{i}")
```

```
print("All three arguments \"start\", \"stop\" and \"step\" specified in range  
function")
```

```
for i in range(20, 1, -5):
```

```
    print(f"{i}")
```

```
print("Iterate through Each Character in the String")
```

```
for i in "BENGALURU":
```

```
    print(i)
```

# Write a Program to Find the Sum of All Odd and Even Numbers up to a Number

```
number = int(input("Enter a number"))  
even = 0  
odd = 0  
for i in range(number):  
    if i % 2 == 0:  
        even = even + i  
    else:  
        odd = odd + i  
print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")
```

# Programs

1. Write a program to display multiplication table
2. Write a Program to Find the Factorial of a Number
3. Write a program to generate a OTP
4. Write a Program to Display the Fibonacci Sequences up to nth Term Where n is Provided by the User
5. Write a Program to Display the Fibonacci Sequences between the given limits and count the number of odd and even Fibonacci number within the limits.
6. Write a Program to Display the Pattern

## #Write a Program to Display the Fibonacci Sequences up to nth Term Where n is Provided by the User

```
nterms = int(input('How many terms? '))
previous ,current, next_term, count = 1,0,0,0
if nterms <= 0:
    print('Please enter a positive number')
elif nterms == 1:
    print('Fibonacci sequence')
    print('0')
else:
    print("Fibonacci sequence")
while count < nterms:
    print(next_term)
    current = next_term
    next_term = previous + current
    previous = current
    count += 1
```

## Write a Program to Find the Factorial of a Number

```
number = int(input('Enter a number'))  
factorial = 1  
if number < 0:  
    print("Factorial doesn't exist for negative numbers")  
elif number == 0:  
    print('The factorial of 0 is 1')  
else:  
    for i in range(1, number + 1):  
        factorial = factorial * i  
    print(f"The factorial of number {number} is {factorial}")
```

# Patterns

Enter the number of rows 9

Pattern-1

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-2

```
*
* *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-3

```
*
* *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-4

```

  *
 * *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-5

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-7

```
* * * * *
 * * * * *
  * * * * *
   * * * * *
    * * * * *
     * * * * *
      * * * * *
       * * * * *
        * * * * *
         * * * * *
```

Pattern-8

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-9

```
*
* *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-10

```

  *
 * *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-6

```

  *
 * *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```



# Patterns

```
  * * *
*       *
*       *
* * * * *
*       *
*       *
*       *
```

```
* * * * *      * * * *
*              *
*              *
* * *          *
*              *
*              *
* * * * *      * * * *
* * * * *      * * * *
```

# The *continue* and *break* Statements

- ▶ The *break* and *continue* statements provide greater control over the execution of code in a loop.
- ▶ Whenever the *break* statement is encountered, the execution control immediately jumps to the first instruction following the loop.
- ▶ To pass control to the next iteration without exiting the loop, use the *continue* statement.
- ▶ Both *continue* and *break* statements can be used in *while* and *for* loops.

## Program to Demonstrate Infinite while Loop and break

```
n = 0
while True:
    print(f"The latest value of n is {n}")
    n = n + 1
```

```
n = 0
while True:
    print(f"The latest value of n is {n}")
    n = n + 1
    if n > 5:
        print(f"The value of n is greater than 5")
        break
```

## Program to Demonstrate break

```
available=10
num=int(input("Enter "))

i=1
while i<=num:
    if i>available:
        print('out of stock')
        break
    print("chocolates")
    i=i+1
print("Thanks")
```

## Program to Demonstrate *continue* and *Pass* Statement

### Example:

```
n=10
while n>0:
    print(f"current value is{n}")
    if n==5:
        print(f"breakin at{n}")
        n=10
        continue
    n=n-1
```

### Scenario-1:

```
for i in range(1,24):
    if i%3==0 or i%5==0:
        continue
    print(i)
print("Thats it")
```

### Scenario-2:

```
for i in range(1,24):
    if i%3==0 or i%5==0:
        print(i)
print("Thats it")
```

### Example

```
for i in range(1,21):
    if (i%2==0):
        pass
    else:
        print(i)
print("Thanks")
```

# Catching Exceptions Using try and except Statement

- ▶ There are at least two distinguishable kinds of errors:

1. Syntax Errors: most common

2. Exceptions

- ▶ Exception handling is one of the most important feature of Python programming language that allows us to handle the errors caused by exceptions.
- ▶ Errors detected during execution are called **exceptions**.
- ▶ Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- ▶ An exception is an unwanted event that interrupts the normal flow of the program.
- ▶ When an exception occurs in the program, execution gets terminated. In such cases, we get a system-generated error message. However, these exceptions can be handled in Python.
- ▶ By handling the exceptions, we can provide a meaningful message to the user about the issue rather than a system-generated message, which may not be understandable to the user.

## Exceptions can be either built-in exceptions or user-defined exceptions

Exceptions can be either **built-in exceptions** or **user-defined exceptions**. The interpreter or built-in functions can generate the built-in exceptions while user-defined exceptions are custom exceptions created by the user.

When the exceptions are not handled by programs it results in error messages as shown below.

```
>>> 10 * (1/0)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
```

```
>>> 4 + spam*3
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
```

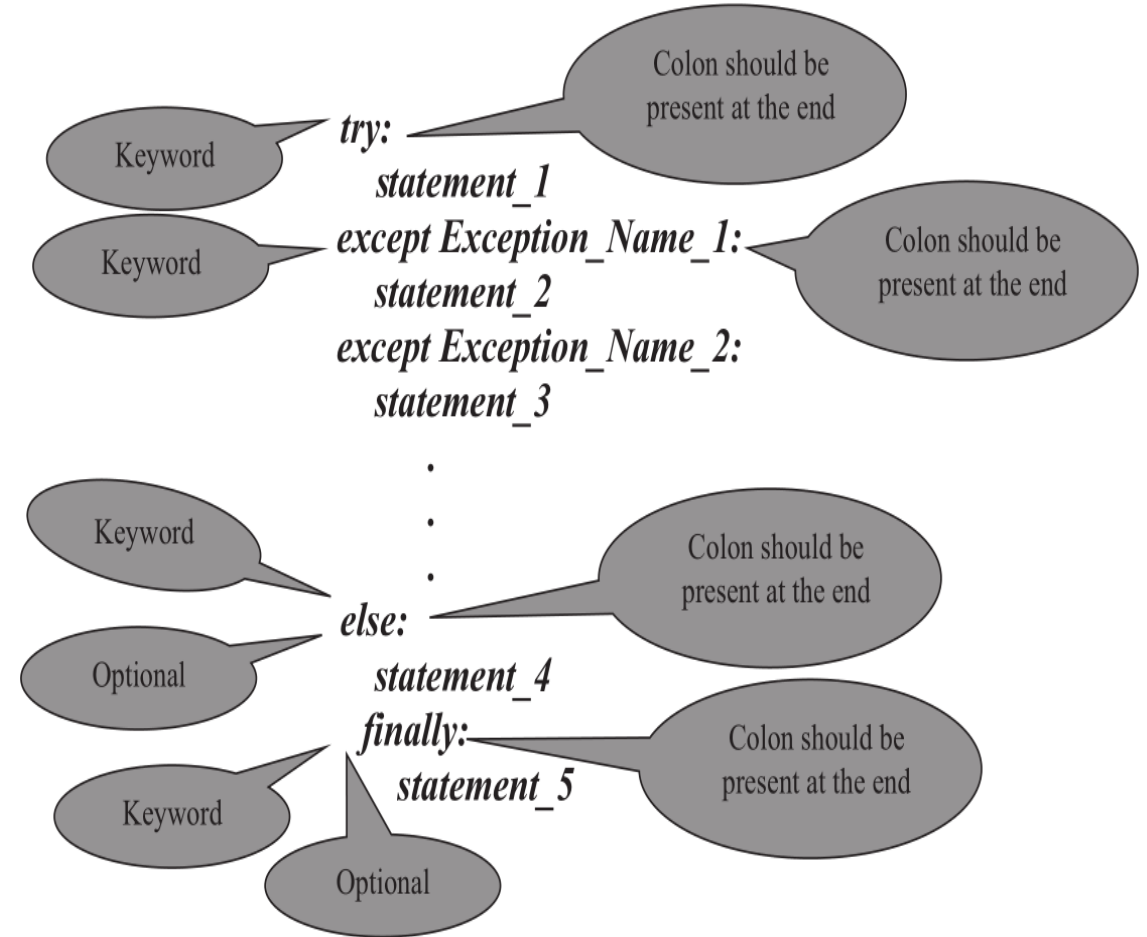
```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: Can't convert 'int' object to str implicitly
```

# Exception Handling Using try...except...finally

- ▶ Handling of exception ensures that the **flow of the program does not get interrupted** when an exception occurs which is done by trapping **run-time errors**.
- ▶ Handling of exceptions results in the execution of all the statements in the program.
- ▶ **Run-time errors** are those errors that occur during the **execution of the program**.
- ▶ These errors are **not detected by the Python interpreter**, because the code is syntactically correct.
- ▶ The **else block**, which, when present, must follow all except blocks. It is useful for code that must be executed if the **try block does not raise an exception**.
- ▶ A **finally** block is always executed before leaving the try statement, whether an exception has occurred or not.





## Program to check for ValueError Exception

```
1 while True:
2     try:
3         number = int(input("Please enter a number: "))
4         print(f"The number you have entered is {number}")
5         break
6     except ValueError:
7         print("Oops! That was no valid number. Try again...")
```

Please enter a number: g

Oops! That was no valid number. Try again...

Please enter a number: 4

The number you have entered is 4

## Example:

### ► Program to Check for ZeroDivisionError Exception

```
1 x = int(input("Enter value for x: "))
2 y = int(input("Enter value for y: "))
3 try:
4     result = x / y
5 except ZeroDivisionError:
6     print("Division by zero!")
7 else:
8     print(f"Result is {result}")
9 finally:
10    print("Executing finally clause")
```

```
Enter value for x: 8
Enter value for y: 0
Division by zero!
Executing finally clause
```

```
Enter value for x: 4
Enter value for y: 2
Result is 2.0
Executing finally clause
```

The *else* block gets executed if the *try* block does not raise an exception.

# Try except finally

#compile time error- syntax error, #logical error-getting wrong answers,  
#runtime error-6/0-dividebyzero error

a=5

b=0

try:

print("resource open")

print(a/b)

k=int(input("Enter "))

print(k)

except ZeroDivisionError as e:

print(" cant divide by zero",e)

except ValueError as v:

print(" invalid input",v)

except Exception as p:

print(" oops: something wrong",p)

finally:

print("resource closed")

print("thanks")

Write a Program Which Repeatedly Reads Numbers Until the User Enters 'done'. Once 'done' Is Entered, Print Out the Total, Count, and Average of the Numbers. If the User Enters Anything Other Than a Number, Detect Their Mistake Using try and except and Print an Error Message and Skip to the Next Number

```
1 total = 0
2 count = 0
3 while True:
4     num = input("Enter a number: ")
5     if count != 0 and num == 'done':
6         print(f"Sum of all the entered numbers is {total}")
7         print(f"Count of total numbers entered {count}")
8         print(f"Average is {total / count}")
9         break
10    else:
11        try:
12            total += float(num)
13        except:
14            print("Invalid input")
15            continue
16        count += 1
```

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: 5
Enter a number: done
Sum of all the entered numbers is 15.0
Count of total numbers entered 5
Average is 3.0
```

## Example

```
year = int(input('Enter a year'))
```

```
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print(f'{year} is a Leap Year')
        else:
            print(f'{year} is not a Leap Year')
    else:
        print(f'{year} is a Leap Year')
else:
    print(f'{year} is not a Leap Year')
```

```
1 year=int(input('enter the year'))
2 if(year%4==0 and (year%400==0 or year%100!=0)):
3     print(f'lear year')
4 else:
5     print(f'not a leap year')
```

```
enter the year2020
lear year
```

## ► Pattern Generation

1. To print \* in same line

```
n=int(input("Enter n value:"))
```

```
for i in range(n):
```

```
    print('*')
```

```
    print('* ',end=' ')
```

```
Enter n value:3
```

```
*  
*  
*
```

```
Enter n value:3
```

```
* * *
```

```
>>> help(print)  
Help on built-in function print in module builtins:  
  
print(...)  
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)  
  
    Prints the values to a stream, or to sys.stdout by default.  
    Optional keyword arguments:  
    file: a file-like object (stream); defaults to the current sys.stdout.  
    sep:  string inserted between values, default a space.  
    end:  string appended after the last value, default a newline.  
    flush: whether to forcibly flush the stream.
```

## 2. To print square pattern with \*

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print('* ' * n)
```

```
Enter number of rows:3
```

```
* * *
```

```
* * *
```

```
* * *
```

### 3. To print square pattern with fixed digit

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print((str(n)+ ' ') *n)
```

```
Enter number of rows:4
```

```
4 4 4 4
```

```
4 4 4 4
```

```
4 4 4 4
```

```
4 4 4 4
```



#### 4. To print square pattern with fixed alphabet

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print('A ' *n)
```

```
Enter number of rows:3
```

```
A A A
```

```
A A A
```

```
A A A
```

## 5. To print pattern with digits

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print((str(i+1)+ ' ' ) *n)
```

```
Enter number of rows:4
```

```
1 1 1 1
```

```
2 2 2 2
```

```
3 3 3 3
```

```
4 4 4 4
```

## 6. To print pattern with alphabets

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print((chr(65+i)+ ' ' ) * (i+1))
```

```
Enter number of rows:5
```

```
A
```

```
B B
```

```
C C C
```

```
D D D D
```

```
E E E E E
```

## 7. To print right angled triangle

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print(("* ")*(i+1))
```

```
Enter number of rows:3
```

```
*
```

```
* *
```

```
* * *
```

## 8. To print Inverted right angled triangle

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print(("* ")*(n-i))
```

```
Enter number of rows:4
```

```
*****
```

```
****
```

```
***
```

```
**
```

```
*
```

## 9. To print right angled triangle

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print((" "*(n-i)+ "*"*(i))
```

```
Enter number of rows:4
```

```
    *  
  * *  
 * * *  
* * * *
```

## 10. To print triangle

```
n=int(input("Enter number of rows:"))  
for i in range(n):  
    print((" "*(n-i)+ "*"*(i))
```

Enter number of rows:5

```
  *  
 * *  
* * *  
* * * *  
* * * * *
```

## 11. To print inverted triangle

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print((" " * i + "*" * (n-i))
```

Enter number of rows:4

\* \* \* \*

\* \* \*

\* \*

\*



## 12. To print diamond pattern

```
n=int(input("Enter number of rows:"))
```

```
for i in range(n):
```

```
    print((" "*(n-i-1)+ "*"*(i+1))
```

```
for i in range(n-1):
```

```
    print((" "*(i+1)+ "*"*(n-i-1))
```

```
Enter number of rows:4
    *
   * *
  * * *
 * * * *
  * * *
   * *
    *
```

► To print Alphabet

for row in range(7): # 0 to 6

for col in range(5): # 0 to 4

if row == 0 and (col in {1,2,3}):

print('\*', end=' ')

elif row in {1,2,4,5,6} and (col in {0,4}):

print('\*', end=' ')

elif row == 3:

print('\*', end=' ')

else:

print(' ', end=' ')

print()



## Program it:

- ❑ program for the following scenario:
  - print all the numbers till n(entered by the user) but ..
  - print “CLAP” for the numbers divisible by 3
  - print “Hha Ha” for the numbers divisible By 5
  - print both for the numbers divisible by both
  
- ❑ Enter the numbers till the user enters word “end”, Count and print the number of
  - Odd number
  - Even number
  - Positive number
  - Negative number
  - Prime number
  - is Armstrong number
  - Is perfect number
  - Is Strong number
  - If it belongs to Fibonacci series, mention its position in the series

## Armstrong number.

- ▶ Start with the procedure for testing to see if a number is an Armstrong number.
  - ❑ Determine how many digits are in the number. Call that  $n$ .
  - ❑ Then take every digit in the number and raise it to the  $n$  power.
  - ❑ Add all those together, and if your answer is the original number then it is an Armstrong number.
  - ❑ All single digit numbers are Armstrong numbers.

### For example:

- ✓ **Test 6** : One digit ....  $6^1 = 6$ . is Armstrong number.
- ✓ **Test 371** : Three digits ....  $3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$ . is Armstrong number
- ✓ **Test 1634** : Four digits ....  $1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634$ . is Armstrong number.

## STRONG number

If the **sum of factorial of the digits in any number** is equal the given **number** then the number is called as STRONG number.

**Example=check 145:**  $1! + 4! + 5! = 1 + 24 + 120 = 145$

## Perfect number

In number theory, a perfect number is a positive integer that is equal to the **sum of its positive divisors, excluding the number itself**.

**For Example:**

6 has divisors 1, 2 and 3 (excluding itself), and  $1 + 2 + 3 = 6$ , so 6 is a perfect number.

## 6174 is known as Kaprekar's constant

- ▶ the Indian mathematician D. R. Kaprekar.
- ▶ This number is renowned for the following rule:
  - ▶ Take any four-digit number, using at least two different digits (leading zeros are allowed).
  - ▶ Arrange the digits in descending and then in ascending order to get two four-digit numbers, adding leading zeros if necessary.
  - ▶ Subtract the smaller number from the bigger number.
  - ▶ Go back to step 2 and repeat.
- ▶ The above process, known as Kaprekar's routine, will always reach its fixed point, 6174, in at most 7 iterations. Once 6174 is reached, the process will continue yielding  $7641 - 1467 = 6174$ .
- ▶ For example, choose 1495:
$$9541 - 1459 = 8082$$
$$8820 - 0288 = 8532$$
$$8532 - 2358 = 6174$$
$$7641 - 1467 = 6174$$
- ▶ The only four-digit numbers for which Kaprekar's routine does not reach 6174 are repdigits such as 1111, which give the result 0000 after a single iteration. All other four-digit numbers eventually reach 6174 if leading zeros are used to keep the number of digits at 4.

## Program it:

- print the grade report considering both thresholds for the grade and grades considered in increasing order.
- print the numbers from 1 to 46 excluding the numbers divisible by 3 , 5  
print the numbers from 1 to 46, only divisible by 3 , 5  
print the numbers from 1 to 46 divisible by both 3 and 5
- check kaprekar constant: 6174( [https://youtu.be/d8TRcZklX\\_Q](https://youtu.be/d8TRcZklX_Q))
- Example: 4 digit number - 9218

9 8 2 1	8 5 3 2	7 6 4 1
-	-	-
1 2 8 9	2 3 5 8	1 4 6 7
=	=	=
8 5 3 2	6 1 7 4	6 1 7 4

## *Program it:*

- ▶ Program to accept 5 digits and print all possible combinations from these digits
- ▶ Python Program to Check if given input is a Palindrome
- ▶ Program for the following:
- ▶ Python Program to Read a Number n And Print the Series "1+2+....+n= “
- ▶ Count for vowels and consonants in the given alphabets



# Functions

- ▶ Built-In Functions
- ▶ Commonly Used Modules
- ▶ Function Definition and Calling the Function
- ▶ The return Statement and void Function

```
def ppsession( , , ):  
    '''  
    '''  
  
ppsession( , , )
```

# Functions

- ❑ Functions are used when you have a **block of statements that needs to be executed multiple times** within the program.
- ❑ Rather than writing the block of statements repeatedly to perform the action, you can use a **function to perform that action**.
- ❑ This block of statements are grouped together and is given a name which can be used to **invoke it from other parts of the program**.
- ❑ You write a function once but can **execute it any number of times** you like.
- ❑ Functions also **reduce the size of the program** by eliminating rudimentary code. Functions can be either **Built-in Functions or User-defined functions**.

# Functions

You can define functions to provide the required functionality.

Here are simple rules to define a function in Python.

- Function blocks begin with the keyword `def` followed by the function name and parentheses `( )`.
- Any input `parameters or arguments` should be placed within these parentheses. You can also `define parameters` inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or `docstring`.
- The `code block` within `every function starts with a colon (:) and is indented`.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

# Functions

- Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.
- Function is a subprogram that works on data and produces some output.

There are two types of Functions.

- ✓ a) **Built-in Functions**: Functions that are predefined and organized into a library. We have used many predefined functions in Python.
- ✓ b) **User- Defined**: Functions that are created by the programmer to meet the requirements.

## A Few Built-in Functions in Python

Function Name	Syntax	Explanation
<code>abs()</code>	<i>abs(x)</i> where x is an integer or floating-point number.	The <i>abs()</i> function returns the absolute value of a number.
<code>min()</code>	<i>min(arg_1, arg_2, arg_3, ..., arg_n)</i> where arg_1, arg_2, arg_3 are the arguments.	The <i>min()</i> function returns the smallest of two or more arguments.
<code>max()</code>	<i>max(arg_1, arg_2, arg_3, ..., arg_n)</i> where arg_1, arg_2, arg_3 are the arguments.	The <i>max()</i> function returns the largest of two or more arguments.
<code>divmod()</code>	<i>divmod(a, b)</i> where a and b are numbers representing numerator and denominator.	The <i>divmod()</i> function takes two numbers as arguments and return a pair of numbers consisting of their quotient and remainder. For example, if a and b are integer values, then the result is the same as (a // b, a % b). If either a or b is a floating-point number, then the result is (q, a % b), where q is the whole number of the quotient.
<code>pow()</code>	<i>pow(x, y)</i> where x and y are numbers.	The <i>pow(x, y)</i> function returns x to the power y which is equivalent to using the power operator: $x^{**}y$ .
<code>len()</code>	<i>len(s)</i> where s may be a string, byte, list, tuple, range, dictionary or a set.	The <i>len()</i> function returns the length or the number of items in an object.

# Built-In Functions

```
>>> abs(-3)
```

```
3
```

```
>>> min(1, 2, 3, 4, 5)
```

```
1
```

```
>>> max(4, 5, 6, 7, 8)
```

```
8
```

```
>>> divmod(5, 2)
```

```
(2, 1)
```

```
>>> divmod(8.5, 3)
```

```
(2.0, 2.5)
```

```
>>> pow(3, 2)
```

```
9
```

```
>>> len("Japan")
```

```
5
```

The divmod() function takes two numbers as arguments and return a pair of numbers consisting of their quotient and remainder.

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

<https://docs.python.org/3/library/functions.html> , [https://www.w3schools.com/python/python\\_ref\\_functions.asp](https://www.w3schools.com/python/python_ref_functions.asp)

Built-in Functions				
<a href="#"><u>abs()</u></a>	<a href="#"><u>delattr()</u></a>	<a href="#"><u>hash()</u></a>	<a href="#"><u>memoryview()</u></a>	<a href="#"><u>set()</u></a>
<a href="#"><u>all()</u></a>	<a href="#"><u>dict()</u></a>	<a href="#"><u>help()</u></a>	<a href="#"><u>min()</u></a>	<a href="#"><u>setattr()</u></a>
<a href="#"><u>any()</u></a>	<a href="#"><u>dir()</u></a>	<a href="#"><u>hex()</u></a>	<a href="#"><u>next()</u></a>	<a href="#"><u>slice()</u></a>
<a href="#"><u>ascii()</u></a>	<a href="#"><u>divmod()</u></a>	<a href="#"><u>id()</u></a>	<a href="#"><u>object()</u></a>	<a href="#"><u>sorted()</u></a>
<a href="#"><u>bin()</u></a>	<a href="#"><u>enumerate()</u></a>	<a href="#"><u>input()</u></a>	<a href="#"><u>oct()</u></a>	<a href="#"><u>staticmethod()</u></a>
<a href="#"><u>bool()</u></a>	<a href="#"><u>eval()</u></a>	<a href="#"><u>int()</u></a>	<a href="#"><u>open()</u></a>	<a href="#"><u>str()</u></a>
<a href="#"><u>breakpoint()</u></a>	<a href="#"><u>exec()</u></a>	<a href="#"><u>isinstance()</u></a>	<a href="#"><u>ord()</u></a>	<a href="#"><u>sum()</u></a>
<a href="#"><u>bytearray()</u></a>	<a href="#"><u>filter()</u></a>	<a href="#"><u>issubclass()</u></a>	<a href="#"><u>pow()</u></a>	<a href="#"><u>super()</u></a>
<a href="#"><u>bytes()</u></a>	<a href="#"><u>float()</u></a>	<a href="#"><u>iter()</u></a>	<a href="#"><u>print()</u></a>	<a href="#"><u>tuple()</u></a>
<a href="#"><u>callable()</u></a>	<a href="#"><u>format()</u></a>	<a href="#"><u>len()</u></a>	<a href="#"><u>property()</u></a>	<a href="#"><u>type()</u></a>
<a href="#"><u>chr()</u></a>	<a href="#"><u>frozenset()</u></a>	<a href="#"><u>list()</u></a>	<a href="#"><u>range()</u></a>	<a href="#"><u>vars()</u></a>
<a href="#"><u>classmethod()</u></a>	<a href="#"><u>getattr()</u></a>	<a href="#"><u>locals()</u></a>	<a href="#"><u>repr()</u></a>	<a href="#"><u>zip()</u></a>
<a href="#"><u>compile()</u></a>	<a href="#"><u>globals()</u></a>	<a href="#"><u>map()</u></a>	<a href="#"><u>reversed()</u></a>	<a href="#"><u>__import__()</u></a>
<a href="#"><u>complex()</u></a>	<a href="#"><u>hasattr()</u></a>	<a href="#"><u>max()</u></a>	<a href="#"><u>round()</u></a>	

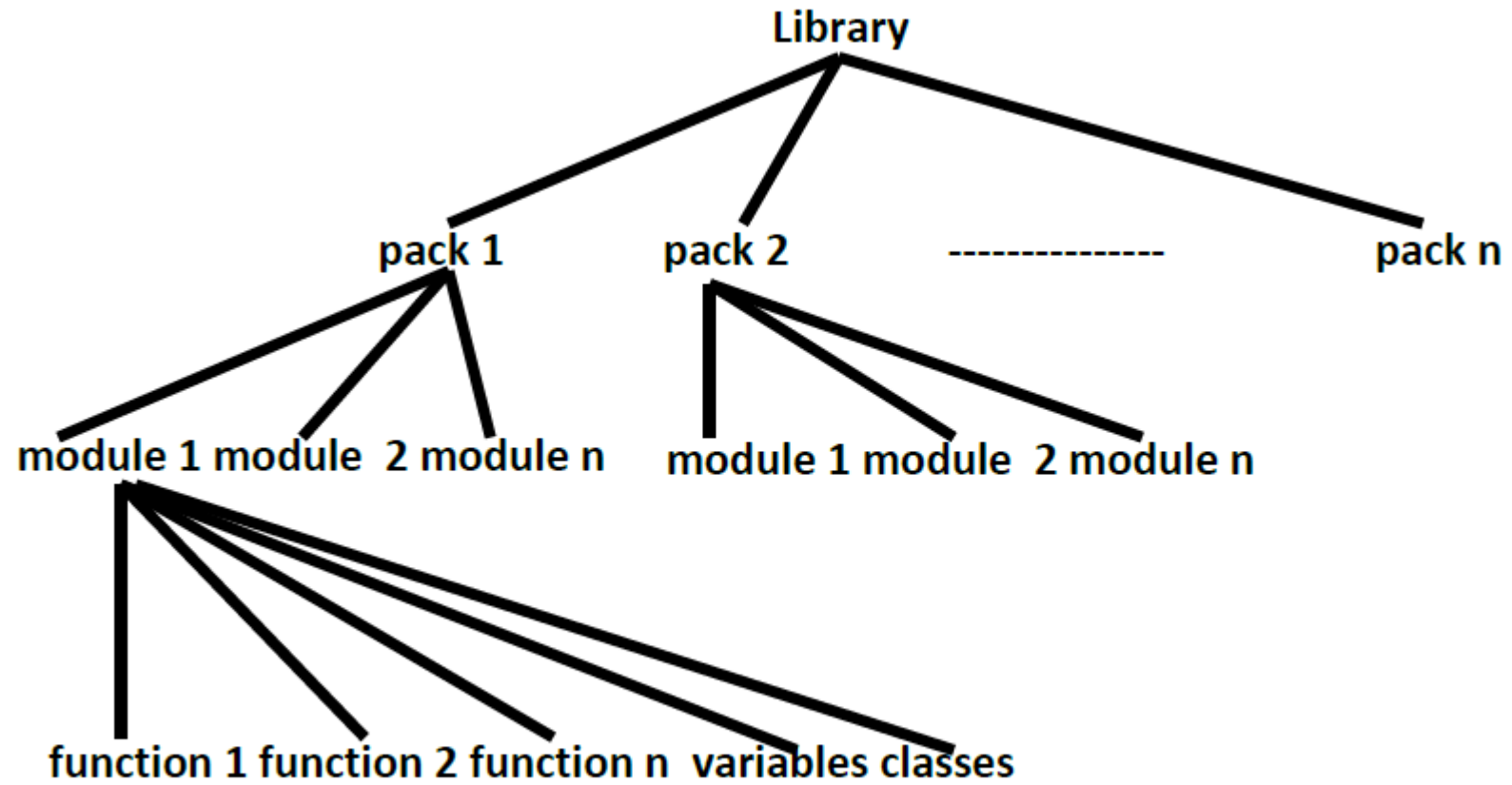
# Commonly Used Modules

- ▶ Modules in Python are **reusable libraries of code having .py extension**, which implements a group of methods and statements. Python comes with many built-in modules as part of the standard library.
- ▶ To use a module in your program, **import the module using *import* statement**. All the ***import*** statements are placed at the **beginning of the program**.
- ▶ **The syntax for import statement** is,

For example, you can import the *math* module as

```
>>>import math
```





# Import()

The syntax for using a function defined in a module is,

*module\_name.function\_name()*

The module name and function name are separated by a **dot**.

Here we list some of the functions supported by *math* module.

```
>>> import math
```

```
>>> print(math.ceil(5.4))
```

```
6
```

```
>>> print(math.sqrt(4))
```

```
2.0
```

```
>>> print(math.pi)
```

```
3.141592653589793
```

# dir()

- ▶ The built-in function *dir()* returns a sorted list of comma separated strings containing the names of functions, classes and variables as defined in the module.

1. >>> dir(math)

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',  
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc',  
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',  
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',  
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

```
>>> print(math.sqrt(4))
```

# help()

- ▶ Another built-in function you will find useful is the *help()* function which invokes the built-in help system.

```
>>> help(math.gcd)
```

```
Help on built-in function gcd in module math:
```

```
gcd(...)
```

```
gcd(x, y) -> int
```

```
greatest common divisor of x and y
```

# Random module

Another useful module in the Python standard library is the *random* module which generates random numbers.

The syntax for *random.randint()* function is *random.randint(start, stop)* which generates a integer number between start and stop argument numbers (including both)

```
>>> import random
```

```
>>> print(random.random())
```

```
0.2551941897892144
```

```
>>> print(random.randint(5,10))
```

```
9
```

► OTP generation

```
import random
```

```
print(random.randint(0,9),random.randint(0,9),random.randint(0,9),
```

```
random.randint(0,9))
```

# pip

- ▶ **Third-party modules or libraries** can be installed and managed using Python's package manager *pip*.
- ▶ The syntax for pip is, *pip install module\_name*
- ▶ **Arrow** is a popular Python library that offers a sensible, human-friendly approach to creating, manipulating, formatting and converting **dates, times, and timestamps**.
- ▶ To install the *arrow* module, open a command prompt window and type the below command from any location.

Below code shows a simple usage of arrow module.

```
C:\> pip install arrow
```

```
>>> import arrow
```

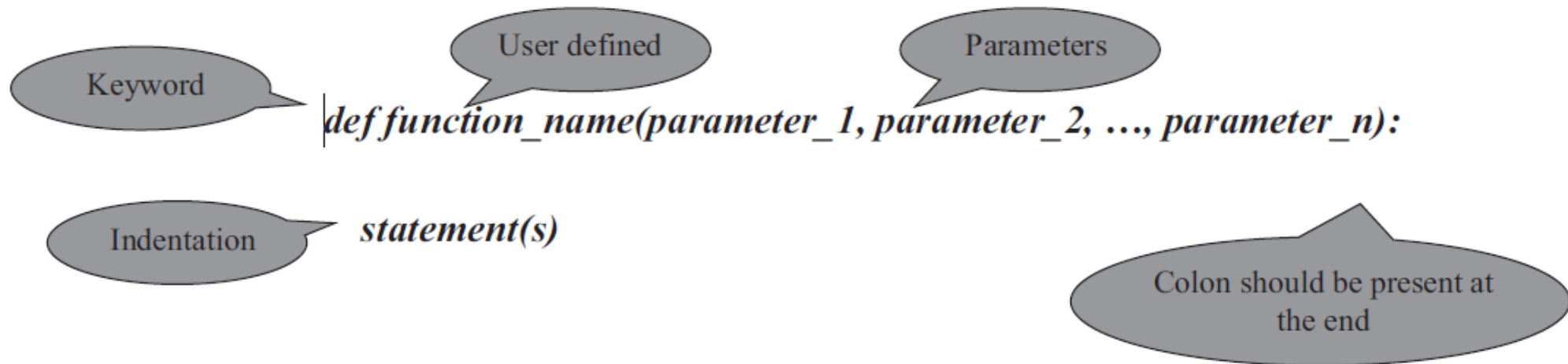
```
>>> a = arrow.utcnow()
```

```
>>> a.now()
```

```
<Arrow [2017-12-23T20:45:14.490380+05:30]>
```

# Function Definition and Calling the Function

- ▶ You can create your **own functions** and use them as and where it is needed. **User-defined functions** are reusable code blocks created by users to perform some specific task in the program.
- ▶ The syntax for function definition is,



# docstring

- ▶ The *def* keyword introduces a function definition. The term parameter or formal parameter is often used to refer to the variables as found in the function definition.
- ▶ The first statement among the block of statements within the function definition can optionally be a documentation string or docstring. There are tools which use docstrings to produce online documents or printed documentation automatically. Triple quotes are used to represent docstrings. For example,

```
""" This is single line docstring """
```

OR

```
""" This is  
multiline  
docstring """
```



# Calling a function

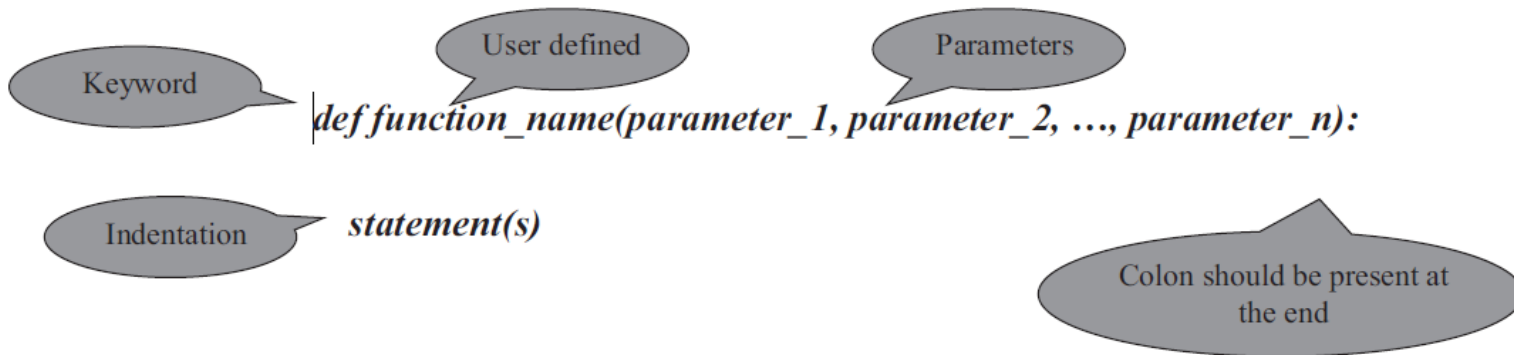
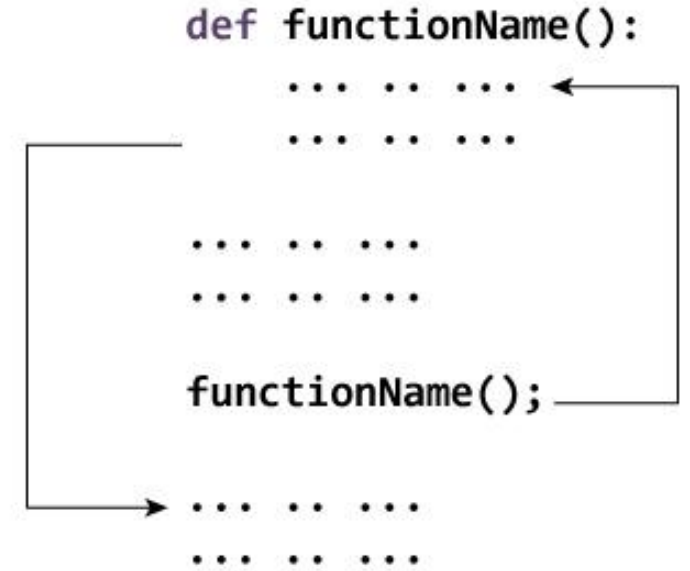
- ▶ Calling the function actually performs the specified actions with the indicated parameters.

The syntax for function call or calling function is,

*function\_name(argument\_1, argument\_2,...,argument\_n)*

- ▶ Arguments are the actual value that is passed into the calling function. There must be a one to one correspondence between the formal parameters in the function definition and the actual arguments of the calling function.

```
def functionName():  
    ... ..  
    ... ..  
  
    ... ..  
    ... ..  
  
functionName();
```



# Function flow

- ▶ A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function.
- ▶ Normally, statements in the Python program are executed one after the other, in the order in which they are written.
- ▶ Function definitions do not alter the flow of execution of the program. When you call a function, the control flows from the calling function to the function definition.
- ▶ Once the block of statements in the function definition is executed, then the control flows back to the calling function and proceeds with the next statement.
- ▶ Python interpreter keeps track of the flow of control between different statements in the program.
- ▶ When the control returns to the calling function from the function definition then the formal parameters and other variables in the function definition no longer contain any values.

# if `__name__ == "__main__":`:

Before executing the code in the source program, the Python interpreter automatically defines a few special variables. If the Python interpreter is running the source program as a stand-alone main program, it sets the special built-in `__name__` variable to have a string value of `"__main__"`. After setting up these special variables, the Python interpreter reads the program to execute the code found in it. All of the code that is at indentation level 0 gets executed. Block of statements in the function definition is not executed unless the function is called.




Double Underscore

```
if __name__ == "__main__":  
    statement(s)
```

The special variable, `__name__` with `"__main__"`, is the entry point to your program. When Python interpreter reads the *if* statement and sees that `__name__` does equal to `"__main__"`, it will execute the block of statements present there.

# The *return* Statement and *void* Function

- ▶ Most of the times you may want the function to perform its specified task to calculate a value and return the value to the calling function so that it can be stored in a variable and used later. This can be achieved using the optional *return* statement in the function definition.
- ▶ The syntax for *return* statement is,  `return [expression_list]`
- ▶ If an expression list is present, it is evaluated, else *None* is substituted. The *return* statement leaves the current function definition with the `expression_list` (or *None*) as a return value.
- ▶ The *return* statement terminates the execution of the function definition in which it appears and returns control to the calling function. It can also return an optional value to its calling function.
- ▶ In Python, it is possible to define functions without a *return* statement. Functions like this are called **void functions**, and they return *None*.

# The *return* Statement and *void* Function

- ▶ Functions without a return statement do return a value, albeit a rather boring one. This value is called *None* (it is a built-in name) which stands for “nothing”. Writing the value *None* is normally suppressed by the interpreter if it would be the only value written.
- ▶ If you want to return a value using *return* statement from the function definition, then you have to assign the result of the function to a variable.
- ▶ In some cases, it makes sense to return **multiple values** if they are related to each other. If so, return the multiple values separated by a comma which by default is constructed as a tuple by Python.
- ▶ When calling function receives a tuple from the function definition, it is common to assign the result to multiple variables by specifying the same number of variables on the left-hand side of the assignment as there were returned from the function definition. This is called tuple unpacking.

# The return Statement

- ▶ The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

```
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2
    print "Inside the function : ", total
    return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

```
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2; # Here total is local variable.
    print "Inside the function local total : ", total
    return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

#EX without return

```
def sumit(num1,num2):
```

```
    sum=num1+num2
```

```
    print(sum)
```

```
sumit(4,3)
```

7

```
x=sumit(4,3)
```

7

```
Print(x)
```

None

#Ex with return

```
def returnsum(num1,num2):
```

```
    sum=num1+num2
```

```
    return(sum)
```

```
returnsum(4,3)
```

7

```
y=returnsum(4,3)
```

7

```
print (y)
```

7

## Returning multiple values from a function

- ▶ In python, a function can return any number of values.

```
def sum_sub(a,b):  
    sum=a+b  
    sub=a-b  
    return sum,sub  
x,y = sum_sub(100,50)  
print("The sum is:",x)  
print("The subtraction is:",y)
```

```
The sum is: 150  
The subtraction is: 50
```



Program to Find the Area of Trapezium Using the Formula  $\text{Area} = (1/2) * (a + b) * h$

Where a and b Are the 2 Bases of Trapezium and h Is the Height

```
def area_trapezium(a, b, h):  
    area = 0.5 * (a + b) * h  
    print(f"Area of a Trapezium is {area}")  
def main():  
    area_trapezium(10, 15, 20)
```

#### Program 4.5: Calculate the Value of sin(x) up to n Terms Using the Series

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \text{ Where } x \text{ Is in Degrees}$$

```
1  #Program 4.5: Calculate the Value of sin(x) up to n Terms Using the Series
2  import math
3  degrees = int(input("Enter the degrees= "))
4  nterms = int(input("Enter the number of terms= "))
5  radians = degrees * math.pi / 180
6  def calculate_sin():
7      result = 0
8      numerator = radians
9      denominator = 1
10     for i in range(1, nterms+1):
11         single_term = numerator / denominator
12         result = result + single_term
13         numerator = -numerator * radians * radians
14         denominator = denominator * (2 * i) * (2 * i + 1)
15     return result
16 def main():
17     result = calculate_sin()
18     print(f"value is sin(x) calculated using the series is {result} ")
19 if __name__ == "__main__":
20     main()
```

# Example

```
colg_name = input('your college name ' )  
def NameOfCollege(college_name):  
    print(f"The college name is {colg_name}")  
NameOfCollege(colg_name)
```

- Example: The following function takes a string as input parameter and prints it on standard screen

```
def printme( str ):  
    "This prints a passed string into this function"  
    print (str)  
    return;
```

# Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.
- Following is the example to call printme() function –

# Function definition is here

```
>>> def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;
```

# Now you can call printme function

```
printme("ECE DSCE")
printme("Bengaluru")
```

# Function Arguments

- You can call a function by using the following types of formal arguments
  - ❖ Positional arguments
  - ❖ Keyword arguments
  - ❖ Default arguments
  - ❖ Variable-length arguments

# Positional arguments

- Required arguments are the arguments passed to a function in correct positional order.
- Here, the number of arguments in the function call should match exactly with the function definition.

## Example:

```
def calculation(a,b,c,d):
```

```
    summ= a+b+c+d
```

```
    subb= c-d
```

```
    mult= a*b
```

```
    divi= a/a
```

```
    return summ, subb, mult, divi
```

```
addition, subtraction, multiplication, division = calculation(2,4, 5, 1)
```

```
print( "addtion= ", addition, "sub=",subtraction,"Mul=",multiplication,"div=",division)
```

# Keyword arguments

- Keyword arguments are related to the function calls. When you use **keyword arguments** in a **function call**, the **caller** identifies the arguments by the parameter name.

```
def calculation(a,b,c,d):  
    summ= a+b+c+d  
    subb= c-d  
    mult= a*b  
    divi= a/a  
    print("a=",a,"b=",b,"c=",c,"d=",d,"sum=",summ, "sub=",subb,"Mul=",mult,"div=",divi)  
  
#Keyword arguments  
calculation(c=9,a=5,b=3,d=2)
```

# Default arguments

- ▶ A default argument is an argument that **assumes a default value** if a value is **not provided in the function call** for that argument.
- ▶ The following example gives an idea on default arguments, it prints default age if it is not passed

```
def calculation(a,b,c=6,d=9):  
    summ= a+b+c+d  
    subb= c-d  
    mult= a*b  
    divi= a/a  
    return summ, subb, mult, divi
```

```
addition, subtraction, multiplication, division=calculation(2,4, 5, 1)  
print( "addition= ", addition, "sub=",subtraction,"Mul=",multiplication,"div=",division)
```

*# Function-Arguments or parameters*

```
def calculation(a,b,c=5,d=6):  
    summ= a+b+c+d  
    subb= c-d  
    mult= a*b  
    divi= a/a  
    print("a=",a,"b=",b,"c=",c,"d=",d,"sum=",summ, "sub=",subb,"Mul=",mult,"div=",divi)
```

*#default arguments*  
calculation(2,9)



# Variable-length arguments

- You may need to process a function for **more arguments than you specified** while defining the function. These arguments are called **variable-length arguments** and are not named in the function definition, unlike required and default arguments.
- An **asterisk (\*)** is placed before the **variable name** that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.
- **\*args** and **\*\*kwargs** are mostly used as parameters in function definitions. \*args and \*\*kwargs allows you to pass a variable number of arguments to the calling function. Here variable number of arguments means that **the user does not know in advance about how many arguments will be passed** to the calling function.
- **\*args as parameter** in function definition allows to pass **non-keyworded, variable length tuple argument** list to calling function.
- **\*\*kwargs as parameter** in function definition allows to pass **keyworded, variable length dictionary argument** list to calling function
- **\*args must come after all the positional parameters** and **\*\*kwargs must come right at the end**.

```
#*variable length arguments
def sumadd(*z):
    sum_var= 0
    for i in z:
        sum_var+=i
    print("sum of var length=",i,sum_var)

sumadd(1,2,3,4,5,6,7,8,9)
```

```
def information(name,sec,**data):
    print("Name and section=",name,sec)
    print("Info=",data)

    for i,j in data.items():
        print(i,j)

information('DSCE',"5B",PIN=560078, city="Benaluru", Phone=1234567890)
```

# Scope and Lifetime of Variables

- ▶ Python programs have two scopes: **global** and **local**.
- ▶ A variable is a **global variable** if its value is **accessible and modifiable throughout your program**. Global variables have a global scope.
- ▶ A variable that is defined **inside a function** definition is a **local variable**.
- ▶ The **lifetime** of a variable refers to the **duration of its existence**. The local variable is created and destroyed every time the **function is executed**, and it **cannot be accessed** by any code **outside the function** definition. Local variables inside a function definition have local scope and exist as long as the function is executing.
- ▶ It is **possible to access global variables** from inside a function, as long as you have not defined a local variable with the same name.
- ▶ A **local variable** can have the **same name as a global variable**, but they are totally different so changing the value of the local variable has **no effect on the global variable**. Only the local variable has meaning inside the function in which it is defined.
- ▶ You can nest a function definition within another function definition. A nested function (inner function definition) can **“inherit” the arguments and variables of its outer function definition**. In other words, the inner function contains the scope of the outer function.
- ▶ The inner function can use the arguments and variables of the outer function, while the **outer function cannot use** the arguments and variables of the inner function.

# Program to Demonstrate the Scope of Variables

```
test_variable = 5
```

```
def outer_function():
```

```
    test_variable = 60
```

```
    def inner_function():
```

```
        test_variable = 100
```

```
        print(f"Local variable value of {test_variable} having local scope to inner function is displayed")
```

```
    inner_function()
```

```
    print(f"Local variable value of {test_variable} having local scope to outer function is displayed ")
```

```
outer_function()
```

```
print(f"Global variable value of {test_variable} is displayed ")
```

# The Anonymous Functions

- Those functions are called anonymous which **are not declared in the standard manner by using the `def` keyword**. You can use the **`lambda`** keyword to create small anonymous functions.
- Lambda forms can **take any number of arguments** but **return just one value** in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have **their own local namespace** and cannot access variables other than those in **their parameter list** and those in the **global namespace**.

## lambda()

The syntax of *lambda()* functions contains only a single statement,

`lambda [arg1 [,arg2,.....argn]]:expression`

```
1 x = lambda a : a + 10
2 print(x(5))
```

15

```
1 x = lambda a, b : a * b
2 print(x(2, 4))
```

8

```
1 x = lambda a, b, c : a + b + c
2 print(x(1, 2, 3))
```

6

# The Anonymous Functions

Syntax:

`lambda arguments: expression`

Ex:

#Add two numbers using lambda function

```
sum=(lambda x,y : x+y)
```

```
print("sum", sum(2,3))
```

sum 5

## filter()

- ▶ filter() function is used to filter values from the given sequence based on some condition.
- ▶ filter(function, sequence)

# pgm to filter only even numbers from the list using filter() function

```
L=[0,5,10,15,20,25,30]
```

```
L1=list(filter(lambda x:x%2 == 0, L))
```

```
print(L1)
```

```
[0, 10, 20, 30]
```

## map()

- ▶ map() function is used when for every element present in the sequence, some functionality can be applied and generate new element with the required modification.
- ▶ map(function, sequence)

# pgm for every element present in the list, perform double and  
generate new list of doubles.

```
L=[1,2,3,4,5]
```

```
L1=list(map(lambda x:2*x, L))
```

```
print(L1)
```

```
[2, 4, 6, 8, 10]
```



## reduce()

- ▶ reduce() function reduces sequence of elements into a single element by applying the specified function.
- ▶ reduce(function, sequence)
- ▶ reduce() function is present in functools module and hence we should use import statement

```
from functools import *
```

```
L=[10,20,30,40,50]
```

```
result=reduce(lambda x,y:x+y, L)
```

```
print(result)
```

```
150
```

## #Pgm to find Fibonacci of number using function

```
def fib(n):  
    a = 0  
    b = 1  
    if n == 1:  
        print(a)  
    else:  
        print(a)  
        print(b)  
        for i in range(2,n):  
            c = a + b  
            a = b  
            b = c  
            print(c)
```

fib(10)

0
1
1
2
3
5
8
13
21
34

## #Pgm to find factorial of number using function

```
def fact(num):  
    result=1  
    while num>=1:  
        result= result*num  
        num = num-1  
    return(result)
```

```
fact(4)  
24
```

## Example - lambda, filter, map, reduce

```
1  from functools import reduce
2
3  marks=[5,4,6,3,4,6,5,9]
4  print("marks=",marks)
5  odd_marks=list(filter(lambda m:m%2,marks))
6  print("odd_marks=",odd_marks)
7  extra_marks=list(map(lambda p:p*2,marks))
8  print("extra_marks=",extra_marks)
9  sum=reduce(lambda a,b:a+b, marks)
10 average=sum/len(marks)
11 print("sum=", sum,"\naverage", average)
```

```
marks= [5, 4, 6, 3, 4, 6, 5, 9]
odd_marks= [5, 3, 5, 9]
extra_marks= [10, 8, 12, 6, 8, 12, 10, 18]
sum= 42
average 5.25
```

#Pgm to convert milliseconds to hours, minutes and seconds using function

```
milli=int(input("Enter time in millisecond:"))
```

```
def convertMillis(milli):
```

```
    seconds=int((milli/1000)%60)
```

```
    minutes=int((milli/(1000*60))%60)
```

```
    hours=int((milli/(1000*60*60))%24)
```

```
    print(hours,":",minutes,":",seconds)
```

```
convertMillis(mili)
```

#Pgm to calculate area of hexagon using function  
 $\text{Area} = (3\sqrt{3}(s)^2)/2$

```
s=int(input("Enter side of hexagon:"))
```

```
from math import sqrt
```

```
def findArea(s):
```

```
    area=(3*sqrt(3)*(s*s))/2
```

```
    return area
```

```
print("Area of Hexagon",findArea(s))
```

# Example: Program it

Regular pentagon

Solve for area ▾

$$A = \frac{1}{4} \sqrt{5(5+2\sqrt{5})} a^2$$

Regular octagon

Solve for area ▾

$$A = 2(1+\sqrt{2}) a^2$$

Regular hexagon

Solve for area ▾

$$A = \frac{3\sqrt{3}}{2} a^2$$

Regular decagon

Solve for area ▾

$$A = \frac{5}{2} a^2 \sqrt{5+2\sqrt{5}}$$

# Strings

- ▶ A string consists of a sequence of characters, which includes letters, numbers, punctuation marks and spaces.
- ▶ To represent strings, you can use a single quote, double quotes or triple quotes.

## The *str()* Function

- ▶ The *str()* function returns a string which is considered an informal or nicely printable representation of the given object. The syntax for *str()* function is *str(object)*
- ▶ It returns a string version of the object. If the object is not provided, then it returns an empty string.



# Basic String Operations

concatenated using + sign and \* operator is used to create a repeated sequence of strings

```
>>> str1="ECE"
```

```
>>> str2="BENGALURU"
```

```
>>> TotalStr1=str1+str2
```

```
>>> TotalStr2="ECE "+"BENGALURU"
```

```
>>> Totalstr3=560078+ "PIN"
```

```
>>> Totalstr4=str(560078)+"PIN"
```

```
>>> Totalstr4="560078"+"PIN"
```

```
>>> Totalstr5="BENGALURU"*4
```

```
>>> str3="a"
```

```
>>> str4=''
```

```
>>> Totalstr3=560078+ "PIN"
```

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

```
Totalstr3=560078+ "PIN"
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

```
>>> Totalstr4=str(560078)+"PIN"
```

```
>>> Totalstr4
```

```
'560078PIN'
```

```
>>> Totalstr4="560078"+"PIN"
```

```
>>> Totalstr4
```

```
'560078PIN'
```

```
>>> Totalstr5="bangaluru"*4
```

```
>>> Totalstr5
```

## Membership operators - *in* and *not in*

check for the presence of a string in another string using *in* and *not in* membership operators. It returns either a Boolean True or False

```
>>> colgstrng="Engg college"
```

```
>>> substr="Engg"
```

```
>>> substr in colgstrng
```

```
>>> substr1="Medical"
```

```
>>> substr1 not in colgstrng
```

# String Comparison

- ▶ You can use (>, <, <=, >=, ==, !=) to compare two strings resulting in either Boolean *True* or *False* value. Python compares strings using ASCII value of the characters.
- ▶ >>> "january"=="jane"
- ▶ >>> "january"<"jane"
- ▶ >>> "january">"jane"
- ▶ >>> "january"!="jane"

# Built-in functions used on strings

## Built-In Functions Used on Strings

Built-In Functions	Description
len()	The <i>len()</i> function calculates the number of characters in a string. The white space characters are also counted.
max()	The <i>max()</i> function returns a character having highest ASCII value.
min()	The <i>min()</i> function returns character having lowest ASCII value.

```
>>> Count=len("bengaluru")
```

```
>>> Count1=len("place mysuru")
```

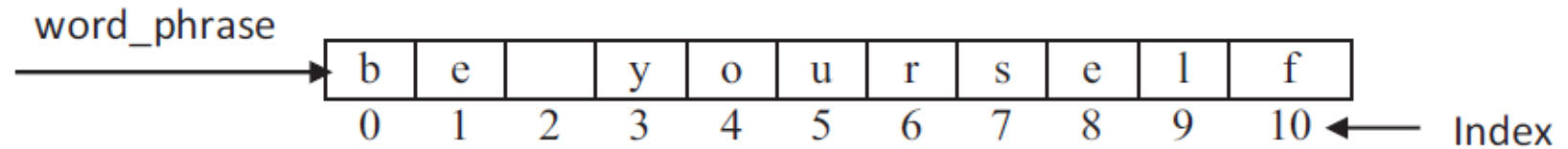
```
>>> max("bengaluru")
```

```
>>> min("bengaluru")
```

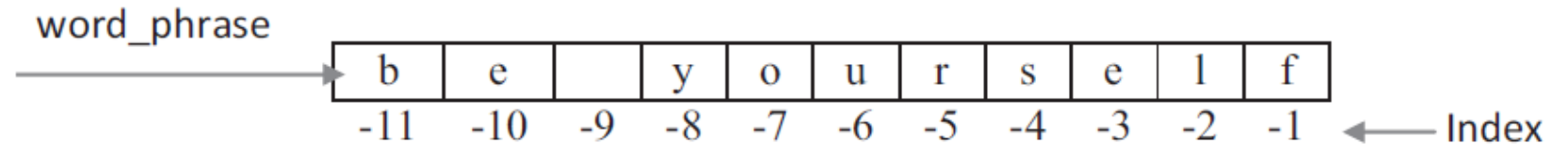
Characters with highest and lowest ASCII value are calculated

# Accessing Characters in String by Index Number

- ▶ The index breakdown for the string "be yourself" assigned to `word_phrase` string variable is shown



- ▶ The syntax for accessing an individual character in a string is as shown below. **`string_name[index]`**
- ▶ `>>> word_phrase[1]`
- ▶ The last character in string is referenced by an index value (size of the string - 1) or `(len(string) - 1)`
- ▶ The negative index breakdown for string "be yourself" assigned to `word_phrase` string variable is shown



- ▶ `>>> word_phrase[-2]`

# String Slicing and Joining

- ▶ The "slice" syntax is a way to refer to sub-parts of sequence of characters within an original string.
- ▶ The syntax for string slicing is, *string\_name[start:end[:step]]*

```
>>> ClgDept="ECE Dept Bengaluru"
```

```
>>> ClgDept[4:15:2]
```

```
>>> ClgDept[:,2]
```

```
>>> clgdept[:,1]
```

```
>>> clgdept[:, -1]
```

```
>>> ClgDept[:,3]
```

```
>>> ClgDept[-9:-2]
```

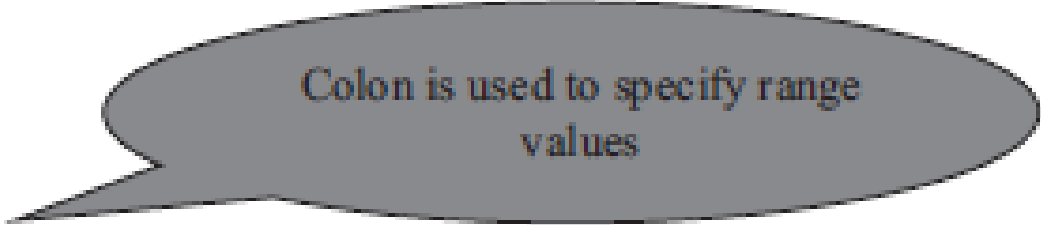
```
>>> ClgDept[2:]
```

```
>>> ClgDept[:5]
```

```
>>> ClgDept[:]
```

```
>>> ClgDept[5:50]
```

```
>>> ClgDept[5:5]
```



Colon is used to specify range values

*string\_name[start:end[:step]]*

Write Program to Determine Whether Given String Is a Palindrome or Not, Using Slicing.

```
Enterstr = input("Enter string: ")
if Enterstr == Enterstr[::-1]:
    print(f" palindrome")
else:
    print(f" Not a palindrome")
print(Enterstr, Enterstr[::-1])
```



## Joining Strings Using join() Method is string\_name.join(sequence)

- ▶ Here sequence can be string or list.
- ▶ If the sequence is a string, then join() function inserts string\_name between each character of the string sequence and returns the concatenated string.
- ▶ If the sequence is a list, then join() function inserts string\_name between each item of list sequence and returns the concatenated string. It should be noted that all the items in the list should be of string type.

```
>>> DOB=["01","02","2020"] but wanted like 01-02-2020
```

```
>>> "-".join(DOB)
```

```
>>> clgg=["ECE","Bengaluru","Engg"]
```

```
>>> ":".join(clgg)
```

```
>>> num="20"
```

```
>>> chrt="AbC$"
```

```
>>> password=num.join(chrt)
```

```
>>> password
```

## Split Strings Using split() Method

- ▶ The `split()` method returns a **list of string items** by breaking up the string using the **delimiter string**.
- ▶ The syntax of `split()` method is, **`string_name.split([separator [, maxsplit]])`**

```
>>> datestr="01-02-2020"
```

```
>>> datestr.split("-")
```

```
>>> datestr="01 02 2020"
```

```
>>> datestr.split(" ")
```

```
>>> datestr.split()
```

# Strings Are Immutable

- ▶ As strings are immutable, **it cannot be modified**.
- ▶ The **characters in a string cannot be changed** once a string value is assigned to string variable.
- ▶ However, you can **assign different string values** to the same string variable.

```
>>> clg1="bengaluru"
```

```
>>> id(clg1)
```

```
>>> clg1="engineering"
```

```
>>> id(clg1)
```

# String Traversing

- ▶ Since the string is a sequence of characters, each of these characters can be traversed using the *for* loop.
- ▶ Program to Demonstrate String Traversing Using the *for* Loop

```
1 Example = "Bengaluru"
2 index = 0
3 print(f"In the string '{Example}'")
4 for i in Example:
5     print(f"Character '{i}' has an index value of {index}")
6     index += 1
```

```
In the string 'Bengaluru'
Character 'B' has an index value of 0
Character 'e' has an index value of 1
Character 'n' has an index value of 2
Character 'g' has an index value of 3
Character 'a' has an index value of 4
Character 'l' has an index value of 5
Character 'u' has an index value of 6
Character 'r' has an index value of 7
Character 'u' has an index value of 8
```

## Program to Print the Characters Which Are Common in Two Strings

```
1 string_1="Jasmine"  
2 string_2="Jasan"  
3 for letter in string_1:  
4     if letter in string_2:  
5         print(f"Character '{letter}' is found in both the strings")  
6
```

Character 'J' is found in both the strings  
Character 'a' is found in both the strings  
Character 's' is found in both the strings  
Character 'n' is found in both the strings

## Write Python Program to Count the Total Number of Vowels, Consonants and Blanks in String

```
#Write Python Program to Count the Total Number of Vowels, Consonants and Blanks in a String
user_string = input("Enter a string: ")
vowels, consonants, blanks = 0, 0, 0
for each_ch in user_string:
    if(each_ch=='a' or each_ch=='e' or each_ch=='i' or each_ch=='o' or each_ch=='u'):
        vowels += 1
    elif "a" < each_ch < "z":
        consonants += 1
    elif each_ch == " ":
        blanks += 1
print(f"Total Vowels in user entered string is {vowels}")
print(f"Total Consonants in user entered string is {consonants}")
print(f"Total Blanks in user entered string is {blanks}")
```

## Write Python Program to Calculate the Length of a String Without Using Built-In *len()* Function

```
user_string = input("Enter a string: ")
count_ch = 0
for each_ch in user_string:
    count_ch += 1
print(f"The length of user entered string is {count_ch} ")
#using function
print("The length of user entered string using function",len(user_string))
```

# String Methods

```
>>> dir(str)
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',  
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',  
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__  
mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',  
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',  
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',  
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',  
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',  
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',  
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```



## Various String Methods

String Methods	Syntax	Description
<code>capitalize()</code>	<code>string_name.capitalize()</code>	The <i>capitalize()</i> method returns a copy of the string with its first character capitalized and the rest lowercased.
<code>casefold()</code>	<code>string_name.casefold()</code>	The <i>casefold()</i> method returns a casefolded copy of the string. Casefolded strings may be used for caseless matching.
<code>center()</code>	<code>string_name.center(width[, fillchar])</code>	The method <i>center()</i> makes <code>string_name</code> centered by taking <code>width</code> parameter into account. Padding is specified by parameter <code>fillchar</code> . Default filler is a space.
<code>count()</code>	<code>string_name.count(substring [, start [, end]])</code>	The method <i>count()</i> , returns the number of non-overlapping occurrences of substring in the range <code>[start, end]</code> . Optional arguments <code>start</code> and <code>end</code> are interpreted as in slice notation.
<code>endswith()</code>	<code>string_name.endswith(suffix[, start[, end]])</code>	This method <i>endswith()</i> , returns <i>True</i> if the <code>string_name</code> ends with the specified suffix substring, otherwise returns <i>False</i> . With optional <code>start</code> , test beginning at that position. With optional <code>end</code> , stop comparing at that position.
<code>find()</code>	<code>string_name.find(substring[, start[, end]])</code>	Checks if substring appears in <code>string_name</code> or if substring appears in <code>string_name</code> specified by starting index <i>start</i> and ending index <i>end</i> . Return position of the first character of the first instance of string substring in <code>string_name</code> , otherwise return <code>-1</code> if substring not found in <code>string_name</code> .
<code>isalnum()</code>	<code>string_name.isalnum()</code>	The method <i>isalnum()</i> returns Boolean <i>True</i> if all characters in the string are alphanumeric and there is at least one character, else it returns Boolean <i>False</i> .

<code>isalpha()</code>	<code>string_name.isalpha()</code>	The method <i>isalpha()</i> , returns Boolean <i>True</i> if all characters in the string are alphabetic and there is at least one character, else it returns Boolean <i>False</i> .
<code>isdecimal()</code>	<code>string_name.isdecimal()</code>	The method <i>isdecimal()</i> , returns Boolean <i>True</i> if all characters in the string are decimal characters and there is at least one character, else it returns Boolean <i>False</i> .
<code>isdigit()</code>	<code>string_name.isdigit()</code>	The method <i>isdigit()</i> returns Boolean <i>True</i> if all characters in the string are digits and there is at least one character, else it returns Boolean <i>False</i> .
<code>isidentifier()</code>	<code>string_name.isidentifier()</code>	The method <i>isidentifier()</i> returns Boolean <i>True</i> if the string is a valid identifier, else it returns Boolean <i>False</i> .
<code>islower()</code>	<code>string_name.islower()</code>	The method <i>islower()</i> returns Boolean <i>True</i> if all characters in the string are lowercase, else it returns Boolean <i>False</i> .
<code>isspace()</code>	<code>string_name.isspace()</code>	The method <i>isspace()</i> returns Boolean <i>True</i> if there are only whitespace characters in the string and there is at least one character, else it returns Boolean <i>False</i> .
<code>isnumeric()</code>	<code>string_name.isnumeric()</code>	The method <i>isnumeric()</i> , returns Boolean <i>True</i> if all characters in the string_name are numeric characters, and there is at least one character, else it returns Boolean <i>False</i> . Numeric characters include digit characters and all characters that have the Unicode numeric value property.

## Various String Methods

String Methods	Syntax	Description
<code>istitle()</code>	<code>string_name.istitle()</code>	The method <i>istitle()</i> returns Boolean <i>True</i> if the string is a title cased string and there is at least one character, else it returns Boolean <i>False</i> .
<code>isupper()</code>	<code>string_name.isupper()</code>	The method <i>isupper()</i> returns Boolean <i>True</i> if all cased characters in the string are uppercase and there is at least one cased character, else it returns Boolean <i>False</i> .
<code>upper()</code>	<code>string_name.upper()</code>	The method <i>upper()</i> converts lowercase letters in string to uppercase.
<code>lower()</code>	<code>string_name.lower()</code>	The method <i>lower()</i> converts uppercase letters in string to lowercase.
<code>ljust()</code>	<code>string_name.ljust(width[, fillchar])</code>	In the method <i>ljust()</i> , when you provide the string to the method <i>ljust()</i> , it returns the string left justified. Total length of string is defined in first parameter of method width. Padding is done as defined in second parameter fillchar. (default is space).
<code>rjust()</code>	<code>string_name.rjust(width[, fillchar])</code>	In the method <i>rjust()</i> , when you provide the string to the method <i>rjust()</i> , it returns the string right justified. The total length of string is defined in the first parameter of the method, width. Padding is done as defined in second parameter fillchar. (default is space).
<code>title()</code>	<code>string_name.title()</code>	The method <i>title()</i> returns “titlecased” versions of string, that is, all words begin with uppercase characters and the rest are lowercase.
<code>swapcase()</code>	<code>string_name.swapcase()</code>	The method <i>swapcase()</i> returns a copy of the string with uppercase characters converted to lowercase and vice versa.

<code>splitlines()</code>	<code>string_name. splitlines([keepends])</code>	The method <i>splitlines()</i> returns a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless <i>keepends</i> is given and <i>true</i> .
<code>startswith()</code>	<code>string_name.startswith(prefix[, start[, end]])</code>	The method <i>startswith()</i> returns Boolean <i>True</i> if the string starts with the prefix, otherwise return <i>False</i> . With optional start, test string_name beginning at that position. With optional end, stop comparing string_name at that position.
<code>strip()</code>	<code>string_name.strip([chars])</code>	The method <i>strip()</i> returns a copy of the string_name in which specified <i>chars</i> have been stripped from both side of the string. If char is not specified then space is taken as default.
<code>rstrip()</code>	<code>string_name.rstrip([chars])</code>	The method <i>rstrip()</i> removes all trailing whitespace of string_name.
<code>lstrip()</code>	<code>string_name.lstrip([chars])</code>	The method <i>lstrip()</i> removes all leading whitespace in string_name.
<code>replace()</code>	<code>string_name.replace(old, new[, max])</code>	The method <i>replace()</i> replaces all occurrences of old in string_name with new. If the optional argument max is given, then only the first max occurrences are replaced.
<code>zfill()</code>	<code>string_name.zfill(width)</code>	The method <i>zfill()</i> pads the string_name on the left with zeros to fill width.

---

*Note:* Replace the word *string\_name* mentioned in the syntax with the actual string name in your code.

## String Methods

- ▶ String methods like *capitalize()*, *lower()*, *upper()*, *swapcase()*, *title()* and *count()* are used for **conversion purpose**.
- ▶ String methods like *islower()*, *isupper()*, *isdecimal()*, *isdigit()*, *isnumeric()*, *isalpha()* and *isalnum()* are used for **comparing strings**.
- ▶ Some of the string methods used for **padding** are *rjust()*, *ljust()*, *zfill()* and *center()*.
- ▶ The string method *find()* is used to **find substring** in an existing string. You can use string methods like *replace()*, *join()*, *split()* and *splitlines()* to **replace** a string in Python.



# Examples

1. >>> fact = "Abraham Lincoln was also a champion wrestler"

2. >>> fact.isalnum()

False

3. >>> "sailors".isalpha()

True

4. >>> "2018".isdigit()

True

5. >>> fact.islower()

False

6. >>> "TSAR BOMBA".isupper()

True

7. >>> "columbus".islower()

True

8. >>> warriors = "ancient gladiators were vegetarians"

9. >>> warriors.endswith("vegetarians")

True

10. >>> warriors.startswith("ancient")

True

# Examples

11. >>> warriors.startswith("A")

False

12. >>> warriors.startswith("a")

True

13. >>> "cucumber".find("cu")

0

14. >>> "cucumber".find("um")

3

15. >>> "cucumber".find("xyz")

-1

16. >>> warriors.count("a")

5

17. >>> species = "charles darwin discovered galapagos tortoises"

18. >>> species.capitalize()

'Charles darwin discovered galapagos tortoises'

19. >>> species.title()

'Charles Darwin Discovered Galapagos Tortoises'

20. >>> "Tortoises".lower()

'tortoises'

## Examples

21. >>> "galapagos".upper()

'GALAPAGOS'

22. >>> "Centennial Light".swapcase()

'cENTENNIAL lIGHT'

23. >>> "history does repeat".replace("does", "will")

'history will repeat'

24. >>> quote = " Never Stop Dreaming "

25. >>> quote.rstrip()

' Never Stop Dreaming'

26. >>> quote.lstrip()

'Never Stop Dreaming '

27. >>> quote.strip()

'Never Stop Dreaming'

28. >>> 'ab c\n\nde fg\rkl\r\n'.splitlines()

['ab c', '\n', 'de fg', '\r', '\n']

29. >>> "scandinavian countries are rich".center(40)

'scandinavian countries are rich'



## Program using Sort()

### ▶ # List of Integers

```
numbers = [1, 3, 4, 2,5,9,6,0]
```

```
numbers.sort()
```

```
print(numbers)
```

### ▶ # List of Floating point numbers

```
decimalnumber = [2.01, 2.00, 3.67, 3.28, 1.68]
```

```
decimalnumber.sort()
```

```
print(decimalnumber)
```

### ▶ # List of strings

```
words = ["ECE", "DSCE", "Bengaluru"]
```

```
words.sort()
```

```
print(words)
```

## Formatting Strings

- ▶ "f-strings"
- ▶ `str.format()`

## Escape Sequences

- ▶ Escape Sequences are a combination of a **backslash (\)** followed by **either a letter or a combination of letters and digits**. Escape sequences are also called as **control sequences**.
- ▶ The **backslash (\)** character is used to escape the meaning of characters that follow it by substituting their special meaning with an alternate interpretation. So, all escape sequences consist of two or more characters.

## List of Escape Sequences

Escape Sequence	Meaning
\	Break a Line into Multiple lines while ensuring the continuation of the line
\\	Inserts a Backslash character in the string
\'	Inserts a Single Quote character in the string
\"	Inserts a Double Quote character in the string
\n	Inserts a New Line in the string
\t	Inserts a Tab in the string
\r	Inserts a Carriage Return in the string
\b	Inserts a Backspace in the string
\u	Inserts a Unicode character in the string
\0oo	Inserts a character in the string based on its Octal value
\xhh	Inserts a character in the string based on its Hex value

# Examples

1. >>> print("You can break \  
... single line to \  
... multiple lines")

You can break single line to multiple lines

2. >>> print('print backslash \\ inside a string ')  
print backslash \ inside a string

3. >>> print('print single quote \' within a string')  
print single quote ' within a string

4. >>> print("print double quote \" within a string")  
print double quote " within a string

5. >>> print("First line \nSecond line")  
  
First line  
  
Second line

6. >>> print("tab\tspacing")  
tab spacing

7. >>> print("same\rlike")  
like

8. >>> print("He\bi")  
Hi

9. >>> print("\u20B9")

10. >>> print("\046")  
&

11. >>> print("\x24")  
\$

# Raw Strings

- ▶ A raw string is created by prefixing the character *r* to the string.
- ▶ In Python, a raw string ignores all types of formatting within a string including the escape characters.

```
>>> print(r" Saying, \"Time and tide\n wait for no\tte; do good be good.\")
```

## Unicode

```
1. >>> unicode_string = u'A unicode \u018e string \xf1'
2. >>> unicode_string
'A unicode string ñ'
```

## Summary

- A string is a sequence of characters.
- To access values through slicing, square brackets are used along with the index.
- Various string operations include conversion, comparing strings, padding, finding a substring in an existing string and replace a string in Python.
- Python strings are immutable which means that once created they cannot be changed.

### Program 5.7: Write Python Program to Convert Uppercase Letters to Lowercase and Vice Versa

```
def case_conversion(user_string):  
    convert_case = str()  
    for each_char in user_string:  
        if each_char.isupper():  
            convert_case += each_char.lower()  
        else:  
            convert_case += each_char.upper()  
    print(f"The modified string is {convert_case}")  
  
input_string = input("Enter a string ")  
case_conversion(input_string)
```

# Example

```
text = input("Enter the String: ")
vowels = ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']
newtext = ""
textlen = len(text)
for i in range(textlen):
    if text[i] not in vowels:
        newtext = newtext + text[i]
        #newtext1=''.join(newtext1)
print("\nString after removing Vowels: ", newtext)
```



## #Program 5.9: Write Python Program to Count the Occurrence of User-Entered #Words in a Sentence

```
def count_word(word_occurrence, user_string):  
    word_count = 0  
    for each_word in user_string.split():  
        if each_word == word_occurrence:  
            word_count += 1  
    print(f"The word '{word_occurrence}' has occurred {word_count} times")  
  
input_string = input("Enter a string ")  
user_word = input("Enter a word to count its occurrence ")  
count_word(user_word, input_string)
```

```
#Program 5.6: Write Python Program That Accepts a Sentence and Calculate
#the Number of Words, Digits, Uppercase Letters and Lowercase Letters
def string_processing(user_string):
    word_count = 0
    digit_count = 0
    upper_case_count = 0
    lower_case_count = 0
    for each_char in user_string:
        if each_char.isdigit():
            digit_count += 1
        elif each_char.isspace():
            word_count += 1
        elif each_char.isupper():
            upper_case_count += 1
        elif each_char.islower():
            lower_case_count += 1
        else:
            pass
    print(f"Number of digits in sentence is {digit_count}")
    print(f"Number of words in sentence is {word_count + 1}")
    print(f"Number of upper case letters in sentence is {upper_case_count}")
    print(f"Number of lower case letters in sentence is {lower_case_count}")

user_input = input("Enter a sentence ")
string_processing(user_input)
```