CST 362
PROGRAMMING IN PYTHON

What is Python? Executive Summary

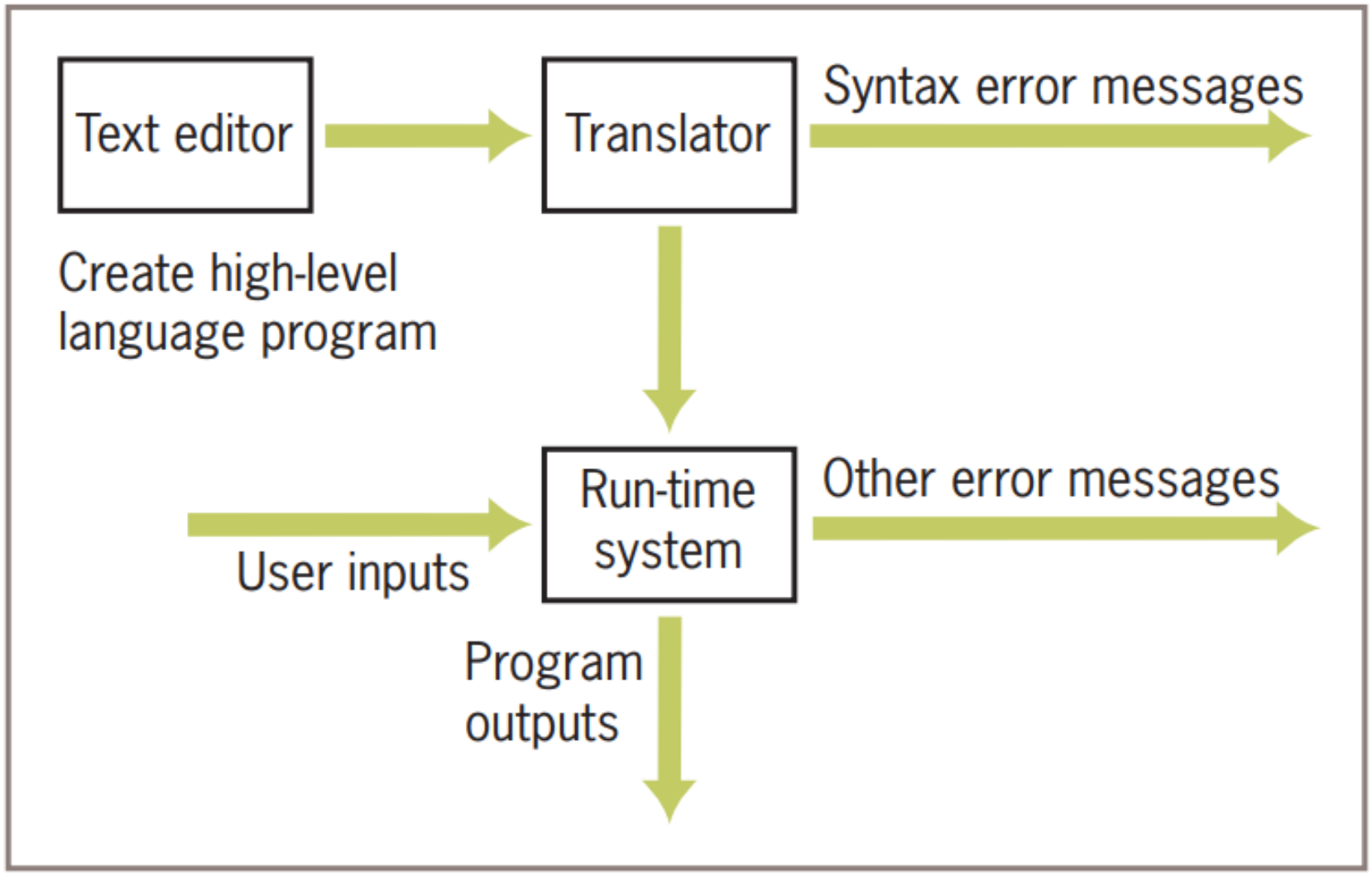
- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics (its variables are dynamic objects).
- Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for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

Guido Van Rossum
Dutch programmer

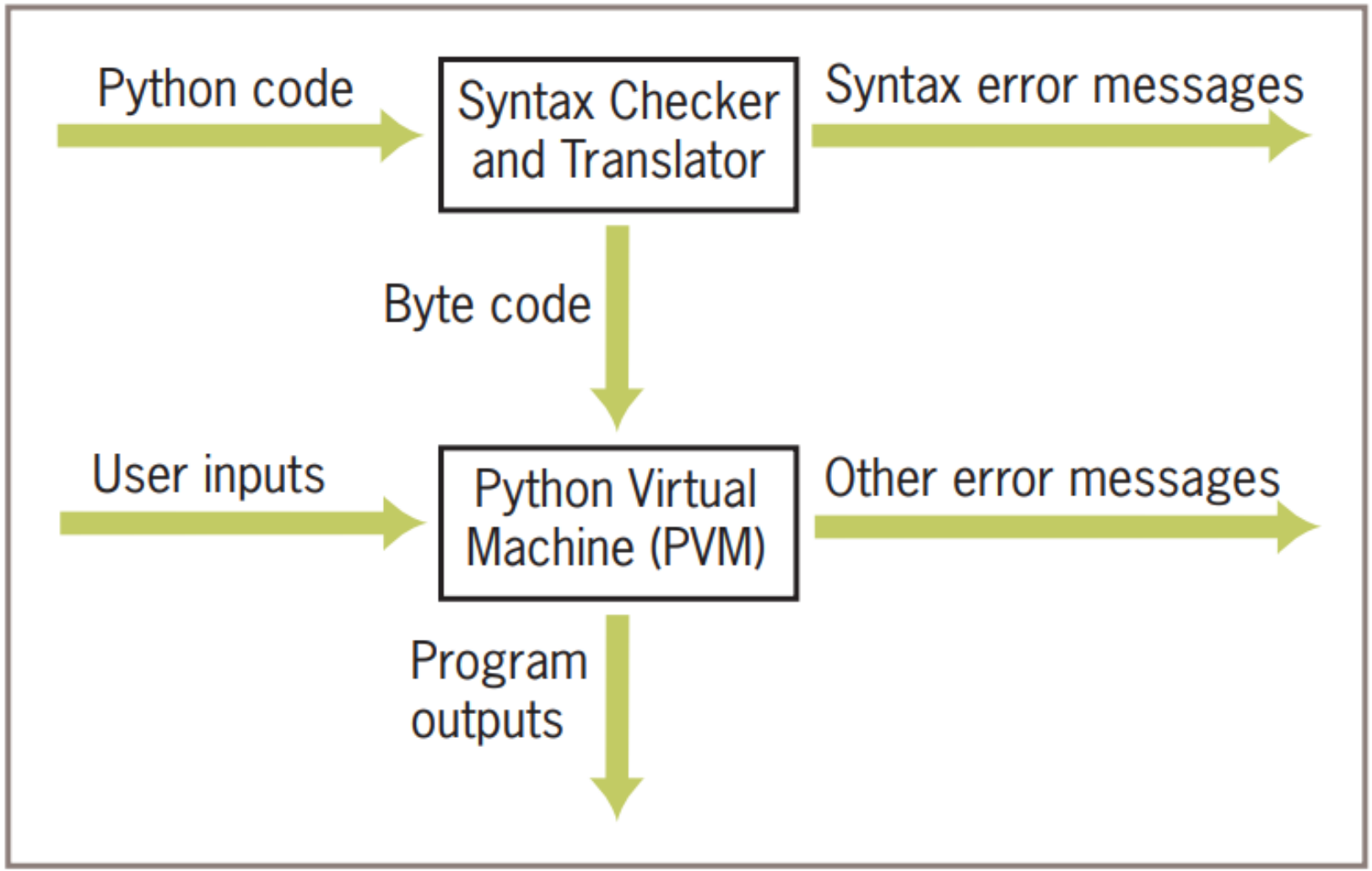
- When he began implementing Python, **Guido van Rossum** was also reading the published scripts from “Monty Python's Flying Circus”, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

Why Python?

- Python has simple, conventional syntax.
- Python has safe semantics.
- Python scales well.
- Python is highly interactive.
- Python is general purpose.
- Python is free and is in widespread use in industry



Software used in the coding process



Steps in interpreting a Python program

Steps in interpreting a Python program

- The interpreter reads a Python expression or statement, also called the source code, and verifies that it is well formed. As soon as the interpreter encounters an error, it halts translation with an error message.
- If a Python expression is well formed, the interpreter then translates it to an equivalent form in a low-level language called byte code. When the interpreter runs a script, it completely translates it to byte code.
- This byte code is next sent to another software component, called the Python virtual machine (PVM), where it is executed. If another error occurs during this step, execution also halts with an error message.

Python Basic syntax, variables and data types

What is syntax?

Syntax refers to the rules that define the structure of a language.

Syntax in computer programming means the rules that control the structure of the symbols, punctuation, and words of a programming language

Example: Use of syntax

- Suppose we consider few words from English language and say like this :

  was killed hunter the by the tiger

- This will not make any sense isn't it. So, if we use the proper syntax or Grammar we can write it correctly as:

  The tiger was killed by the hunter

Therefore, to write a program in any language (python for this course) it is important to learn the syntax of that particular language.

Syntax: Python Identifiers

- A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9)
- Python does not allow punctuation characters within identifier
- Python is a case sensitive language; eg: num1 and Num1 are two different identifier

Syntax: Reserved words

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

Syntax: Comments in Python

- A # sign is used before the text which we want to make comment line

- Example:
- # print(“Hello, Everyone!”)

What is Variable?

Let’s see a statement

age=10;

What is **age** in this statement?

age is the name of the variable or we can say the identifier. Then what is variable? Variable is nothing but reserved memory locations associated with the variable name to store value and this value can change. As in this case value of age will change every year.

What is a constant ?

- Consider another example:

  pi=3.14

  In this case our identifier which is ***pi*** is storing a standard constant value which will never change. So we call it a constant.

- Lets take another example where we take an identifier to store a name of a person:

  name=“Ajay”

In this case our identifier ***name*** is constant as value kept in it will never change for a person.

Python Data Types

- Every value you store in an identifier whether a variable or constant will have a particular data type. Like name will store value of string type (A string is collection of characters).
- Python Numbers
  - Integer
  - Float
  - Complex numbers
- Python List : An ordered sequence of items is called List, There is no need for the value in the list to be of the same data type
- Python Strings
  - Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes

Python Data Types

- Python tuples: The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists

Data Types

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

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

Basic data types	Description	Values	Representation
Boolean	represents two values of logic and associated with conditional statements	True and False	bool
Integer	positive and negative whole numbers	set of all integers, Z	int
Complex	contains real and imaginary part (a+ib)	set of complex numbers	complex
Float	real numbers	floating point numbers	float
String	all strings or characters enclosed between single or double quotes	sequence of characters	str

String Literals

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

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

The last two string literals ('' and "") represent the empty string.

Escape Sequences

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

- The * operator allows you to build a string by repeating another string a given number of times. The left operand is a string, and the right operand is an integer.

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

- Python's ord and chr functions convert characters to their numeric ASCII codes and back again, respectively.

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

Operators

- A symbol that performs a specific operation between two operands is known as an operator.

example: 10 + 5 *(+ is the operator between the operands 10 and 5)*

x=10

y=20

z=x+y

Operators in Python

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators
5. Identity operators
6. Membership operators
7. Bitwise operators

Arithmetic operators

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

Augmented Assignment

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

Comparison Operators

i.	Equal ==	x == y
ii.	Not equal !=	x != y
iii.	Greater than >	x > y
iv.	Less than <	x < y
v.	Greater than or equal to >=	x >= y
vi.	Less than or equal to <=	x <= y

Logical Operators

i. and
ii. or
iii. not

Identity Operators

i. is
ii. is not

Membership operators

i. in
ii. not in

Bitwise operators

i. & (binary and)
ii. | (binary or)
iii. ^ (binary xor)
iv. ~ (negation)
v. << (left shift)
vi >> (right shift)

& (binary and)

- a = 1010 (Binary of 10)
- b = 0100 (Binary of 4)
- a & b =

1010

&

0100

= **0000** = 0 (Decimal)

| (binary or)

- a = 1010 (Binary of 10)
- b = 0100 (Binary of 4)
- a | b =

```
  1010
   |
  0100
= 1110 = 14 (Decimal)
```

^ (binary xor)

- a = 1010 (Binary of 10)
- b = 0100 (Binary of 4)
- a ^ b =

1010
^
0100
= 1110 = 14 (Decimal)

~ (negation)

a= 1010 (Binary of 10)

~a = ~1010

= -(1010 + 1)

= -(1011)

= -11 (Decimal)

<< (left shift)

a= 0000 0101 (Binary of 5)

a << 1 = 0000 1010

= 10 (decimal)

a << 2

= 0001 0100

= 20 (decimal)

>> (right shift)

- a=1010 (Binary of 10)
- a>>2 1010>>2

1	0	1	0

		1	0

10 (Binary) = 2 in Decimal

Precedence rules

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

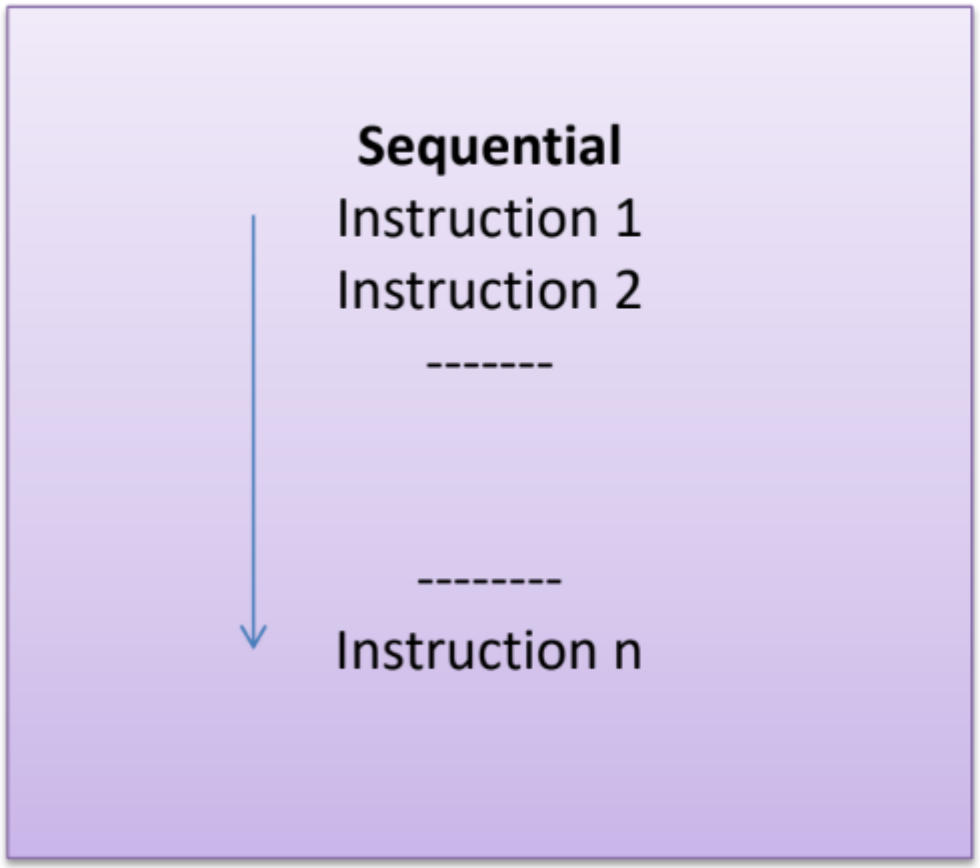
Expression	Evaluation	Value
`5 + 3 * 2`	`5 + 6`	`11`
`(5 + 3) * 2`	`8 * 2`	`16`
`6 % 2`	`0`	`0`
`2 * 3 ** 2`	`2 * 9`	`18`
`-3 ** 2`	`-(3 ** 2)`	`-9`
`(3) ** 2`	`9`	`9`
`2 ** 3 ** 2`	`2 ** 9`	`512`
`(2 ** 3) ** 2`	`8 ** 2`	`64`
`45 / 0`	`Error: cannot divide by 0`	
`45 % 0`	`Error: cannot divide by 0`	

Mixed-Mode Arithmetic and Type Conversions

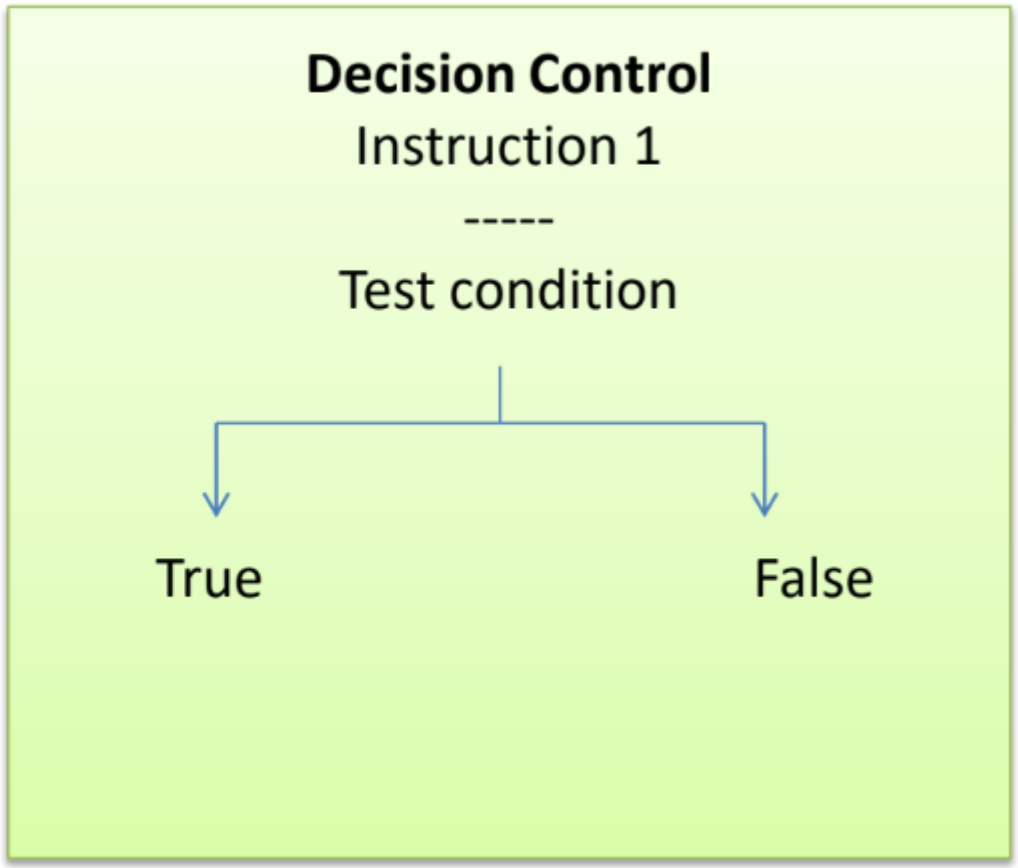
- What happens when one operand is an int and the other is a float?
- Performing calculations involving both integers and floating-point numbers is called mixed-mode arithmetic.
- For instance, if a circle has radius 3, you compute the area as follows:

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

- How does Python perform this type of calculation?
- In a binary operation on operands of different numeric types, the less general type (int) is temporarily and automatically converted to the more general type (float) before the operation is performed.
- Thus, in the example expression, the value 9 is converted to 9.0 before the multiplication.


Sequential
Instruction 1
Instruction 2

Instruction n


Decision Control
Instruction 1

Test condition
True
False

- Thus the decision control statements also known as the *conditional branching statements* allows the programmer to jump from one part of the program to another depending upon a condition.

- Program is a set of instructions.
- Program can be executed:
  - Sequentially: where each instructions are executed one after the another.

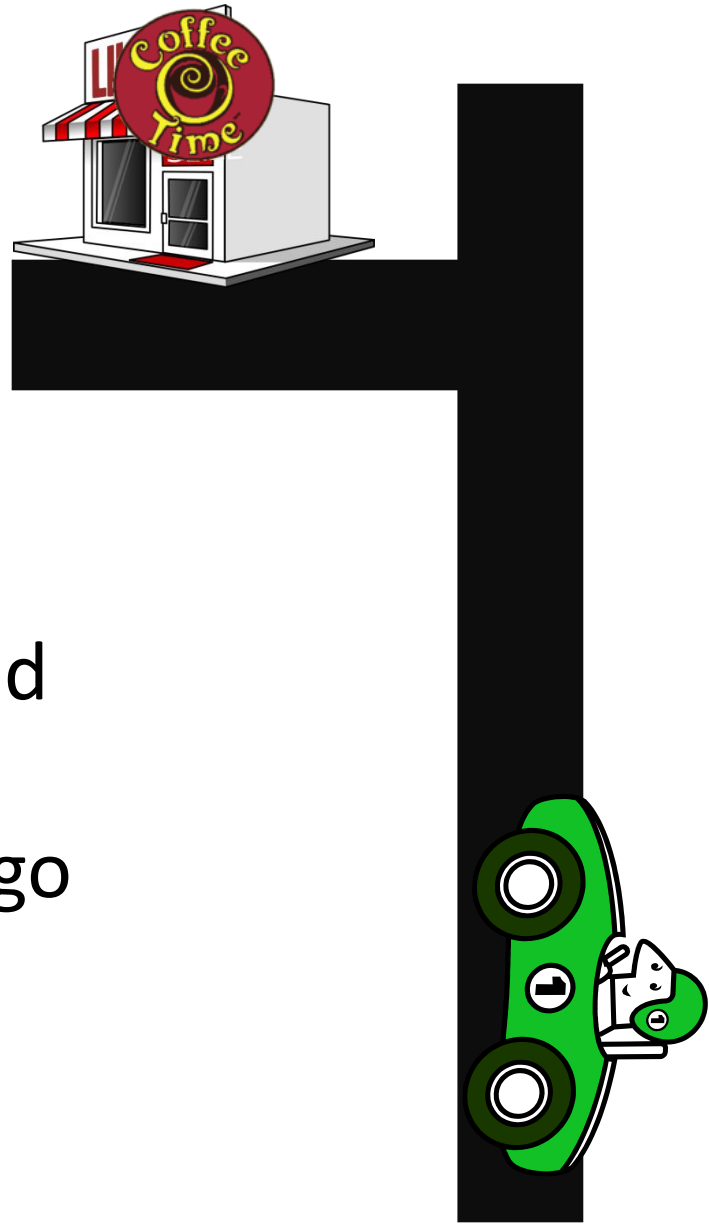
    or

  - We may also alter the sequence of execution of the instructions.

Conditional blocks using if, else and elif.

- The conditional branching statement supported by Python are as follows:
  - if statement
  - if-else statement
  - Nested if statement
  - if-elif-else statement

Conditional Statements

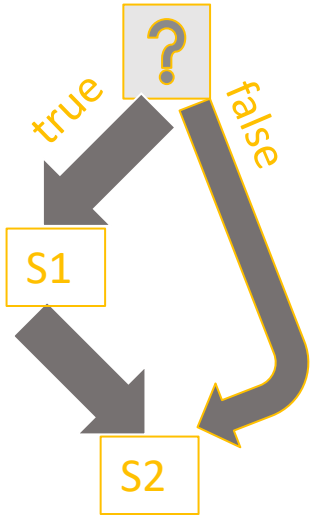


- In daily routine
  - If it is very hot, I will skip exercise.
  - If there is a quiz tomorrow, I will first study and then sleep. Otherwise I will sleep now.
  - If I have to buy coffee, I will go left. Else I will go straight.

if statement (no else!)

- General form of the if statement

```
if boolean-expr :
        S1
S2
```



- Execution of if statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to the S2.
  - If expression evaluates to **false**, then control moves to the S2 directly.

Indentation

- Indentation is **important** in Python
  - grouping of statement (block of statements)
  - no explicit brackets, e.g. { }, to group statements

```
x,y = 6,10

if x < y:
    print (x)
else:
        print (y)
        print('the min')
```

skipped

Run the program

6 10

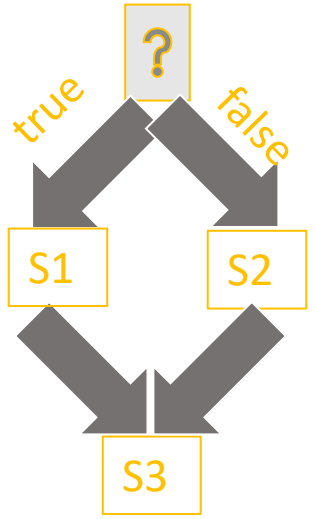
Output

6

if-else statement

- General form of the if-else statement

```
if boolean-expr :
        S1
else:
        S2
S3
```



- Execution of if-else statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to S3.
  - If expression evaluates to **false**, then S2 is executed and then control moves to S3.
  - S1/S2 can be **blocks** of statements!

if-else statement

- Compare two integers and print the min.

```
if  x < y:
        print (x)
else:
        print (y)
print ('is the minimum')
```

1. Check if x is less than y.
2. If so, print x
3. Otherwise, print y.

Nested if, if-else

```
if a <= b:
        if a <= c:
            …
        else:
            …
else:
        if b <= c) :
            …
        else:
            …
```

Elif

- A special kind of nesting is the chain of if-else-if-else-… statements
- Can be written elegantly using if-elif-..-else

```
if cond1:
        s1
else:
    if cond2:
        s2
    else:
        if cond3:
            s3
        else:
            …
```

```
if cond1:
        s1
elif cond2:
    s2
elif cond3:
    s3
elif  …
else
    last-block-of-stmt
```

Write a program to determine whether a person is eligible to drive.

```
age=int(input("Enter the age : "))
if (age>=18):
    print("You are eligible to drive")
```

if-else example

```
age=int(input("Enter the age : "))
if (age>=18):
    print("You are eligible to drive")
else:
    print("You are not eligible to drive")
```

Program to find Largest of two numbers.

```
f_num=int(input("Enter the first number : "))
s_num=int(input("Enter the second number : "))
if(f_num>s_num):
    large = f_num
else:
    large=s_num
print ("Largest is : ",large)
```

elif example

```
marks=int(input("Enter the marks : "))
if (marks>=85 and marks<=100):
    print("Distinction")
elif (marks>=75 and marks<=84):
    print("A+ Grade")
elif (marks>=65 and marks<=74):
    print("A Grade")
elif (marks>=55 and marks<=64):
    print("B+ Grade")
else:
    print("Enter a correct marks")
```

Loops and Selection Statements

- The for Loop

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

- The first line of code in a loop is sometimes called the loop header
- The integer expression denotes the number of iterations that the loop performs.
- The colon (:) ends the loop header
- The **loop body** comprises the statements in the remaining lines of code, below the header
- the statements in the loop body must be indented and aligned in the same column

The range() function

- range() is a inbuilt python function.
- It returns a sequence of numbers which by default starts from 0, increment by 1 and stops at a given value.

- range*(start, stop, step*)
- where:
  - Start is optional and by default begins from 0.
  - Stop is required as any integer value which specifies the stop position.
  - Step is optional and by default is 1.

```
x = range(1, 10, 2)
for n in x:
  print(n)
```

```
x = range(2, 10)
for n in x:
  print(n)
```

```
>>> for count in range(4):
        print(count, end = " ")
0 1 2 3
```

```
for <variable> in range(<lower bound>, <upper bound + 1>):
    <loop body>
```

Off-by-One Error

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

Example

```
for i in range(1,10,2):
    print(i,end=" ")
```

Program to find factorial of a number

```
num=int(input("Enter a number: "))
if (num==0):
    f=1
f=1
for i in range(1,num+1):
    f=f*i
print("Factorial of",num,"is",f)
```

```
for <variable> in <sequence>:
    <do something with variable>
```

```
>>> for number in [6, 4, 8]:
        print(number, end = " ")
6 4 8
>>> for character in "Hi there!":
        print(character, end = " ")
H i  t h e r e !
```

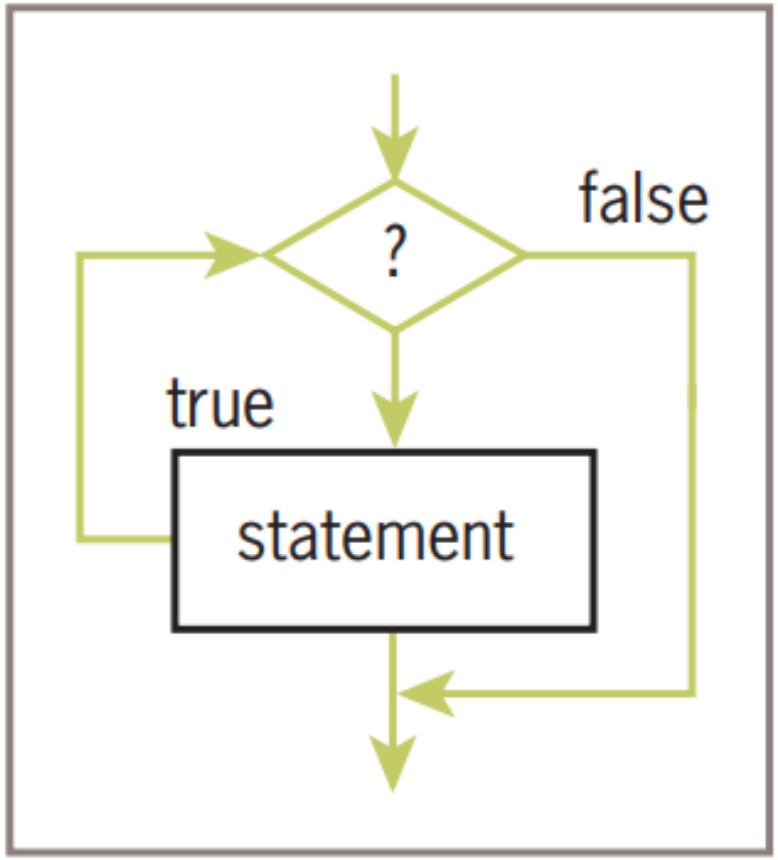
```
for i in range (10):
    print (i)

for i in range (1,10):
    print (i)

for i in range (1,10,2):
    print (i)
```

Conditional Iteration: The while Loop

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



The semantics of a **while** loop

Loop manipulation using pass, continue, break and else

The *break* Statement

- ***break*** statement is used to terminate a loop or bring the control out of a loop when some external condition is triggered.

- ***break*** statement is generally used with ***while*** and ***for*** loop.

- ***if*** statement is used to provide the condition on which break will terminate the loop.

break Example

```
i=1
while i<=10:
    print (i)
    if i==5:
        break
    i=i+1
```

```
s = input("Enter a string: ")
# Using for loop
for letter in s:

    print(letter)

    if letter == 'a' or letter == 'i':
        break

print("Out of for loop")
```

The *continue* Statement

- The ***continue*** statement unconditionally allows the control to jumps to the beginning of the *loop* for the *next iteration.*
- This is just the opposite of the ***break*** statement.

- ***continue*** statement is also generally used with ***while*** and ***for*** loop.

```
i=0
while i<=10:
    i=i+1
    if(i==5):
        continue
    print (i)
```

```
for letter in 'Python‘:
    if letter == 'h':
        continue
    print (letter)
```

The ***pass*** Statement

- It is just a no operation statement.
- You can place a pass statement in the code where you may write the actual set of code later on.

```
for letter in 'Python':
    if letter == 'h':
        pass
        print ("This is pass block")
    print (" Letter :", letter)
```

The ***else*** Statement

- Python supports the use of ***else*** statement with the ***for*** and ***while*** loop.
- The ***else*** statement when used with ***for***, is executed at the termination of the ***for loop***.
- The ***else*** statement when used with ***while***, is executed when the condition becomes false.

```
for letter in "Python":
    print(letter)
else:
    print("Complete")
```

```
count = 0
while (count > 1):
    count = count+1
    print(count)
    break
else:
    print("No Break")
```

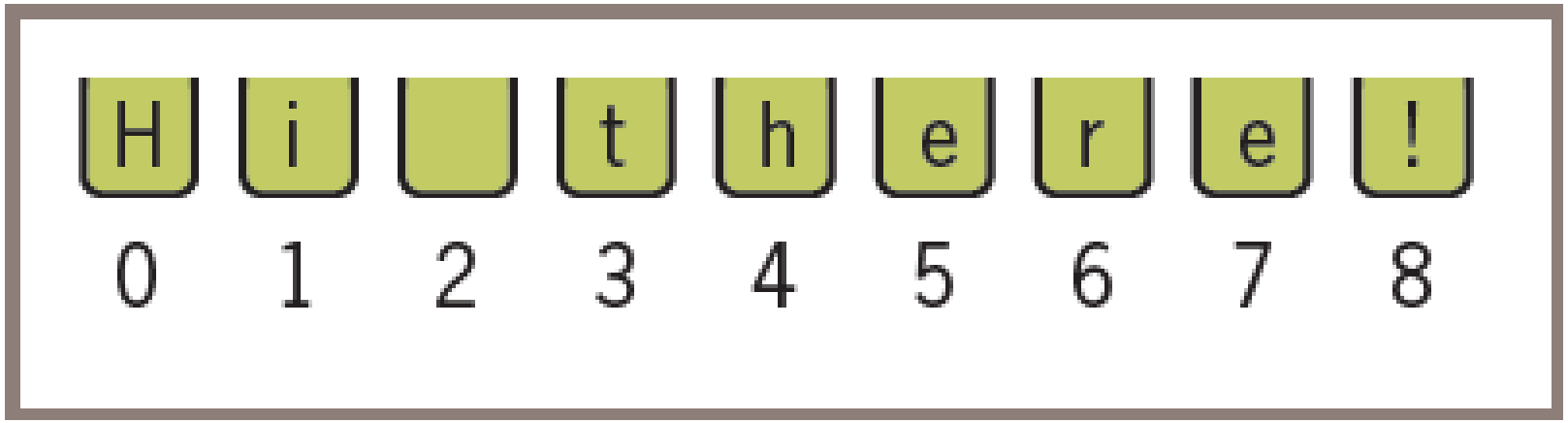
Strings

- A string is a sequence of zero or more characters.
- Unlike an integer, which cannot be decomposed into more primitive parts, a string is a **data structure**.
- A data structure is a compound unit that consists of several other pieces of data.
- You can mention a Python string using either single quote marks or double quote marks.
- The string is an **immutable data structure**. This means that its internal data elements, the characters, can be accessed, but cannot be replaced, inserted, or removed.

Length of a string

- A string's length is the number of characters it contains.
- Python's **len** function returns this value when it is passed a string

```
>>> len("Hi there!")
9
>>> len("")
0
```

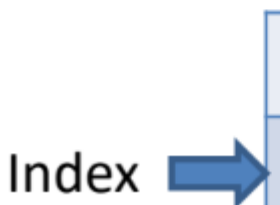


Basic Operations of string

Indexing

- Subscript operator [ ] is used to access the individual characters in a string.
- The value or the expression inside the [ ] brackets is called index.
- This index value is used to access a particular character from a string.
- The index of the first character in a string is 0 (zero) and that of the last character is n-1 where n is the total number of character in a string.

- For example:
- Consider a string : “*PYTHON*”
- Total number of characters in the string is 6 (six) i.e. n=6.



P	Y	T	H	O	N
0	1	2	3	4	5

Concatenation

- Concatenation means joining.
- In python two strings can be joined together using + operator.

- For example:
- first_str = “New”
- second_str= “Delhi”
- first_str + second_str

Appending

- Append is used to add a string at the end of another string.
- The operator used for append operation is **+=**

```
fname = "Sanjay"
lname = "Kapoor"
fname += lname
print(fname)
```

Output:

SanjayKapoor

String slices, function and methods

Slice

- Slice can be used to return a range of characters from a string.

- This is done by specifying the start index and the end index, separated by a colon, to return a part of the string.

- #slicing from index start to index stop-1
- arr[start:stop]

- # slicing from index start to the end
- arr[start:]

- # slicing from the beginning to index stop - 1
- arr[:stop]

- # slicing from the index start to index stop, by skipping step
- arr[start:stop:step]

Example

str='spyder'

*print(str[**2:5**])*

Output: yde

s	p	y	d	e	r
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Example

str='spyder'

*print(str[**3:-1**])*

Output: de

s	p	y	d	e	r
0	1	2	3	4	5

Example

str='spyder'

print(str[:4])

Output: spyd

s	p	y	d	e	r
0	1	2	3	4	5

String Methods

- upper() : To upper case
- lower(): To lower case
- endswith(): Returns true if the string ends with the specified value.
- split(): Splits the string at the specified separator, and returns a list.

- find(): Searches the string for a specified value and returns the position of where it was found.
- strip(): Returns a trimmed version of the string.
- count(): Returns the number of times a specified value occurs in a string.
- isalnum(): Returns True if all characters in the string are alphanumeric

- Find the reverse of a number
- Find the sum of digits of a number
- Convert a decimal number to binary
- Write the python program to print all prime numbers less than 1000
- Python program to check if a string is palindrome or not
- Reverse words in a given String in Python
- Write a Python program to count how many times each character appears in a given string

Practice Problems

- Write the python program to print all prime numbers less than 1000
- Python program to check if a string is palindrome or not
- Decimal to binary conversion
- Binary to decimal conversion
- Write a Python program to count number of even numbers and odd numbers in a given set of n numbers.
- Write a Python program to find distance between two points (x1,y1) and (x2,y2).

Practice Problems

- Count the number of words in a string
- Implement encryption and decryption of a one word lowercase string using Caesar Cipher. Generalize the program to encrypt and decrypt any string (uppercase or lowercase) of multiple words
- Eliminate the vowels in a string
- Covert a string to uppercase
- Find the count of a particular substring in a string

List

Lists in Python

- The data type list is an ordered sequence that is mutable and is made up of one or more elements
- A list can have elements of different data types, such as integer, float, string, tuple or even another list
- A list is very useful to group together elements of mixed data types
- Elements of a list are enclosed in square brackets and are separated by a comma
- List indices also start from 0

Syntax for defining a list

List_variable=[val1, val2,.....]

For example:

list_X= [10, 11, ”ravi” , ’a’, 3.14]

Example

```
first_list= [1,2,3,4,5]
print (first_list)
second_list=['Delhi','b','c',10,20,30,3.14]
print(second_list)
```

- In a List each element has a specific index which starts from zero and is auto incremented for next value. Therefore it can also be considered as an ordered set.

- List items are ordered, changeable, and allow duplicate values.

Example of Index

```
List_city = ["Delhi", "Mumbai", "Kolkata"]
print("\nList Items: ")
print(List_city[0]) # Index 0
print(List_city[2]) # index 2
```

Multi-Dimensional List

```
# Multi-Dimensional List
List = [[1, 2] , [1]]
print(List[0][1])
print(List[1][0])
```

len function

```
List_num = [100, 200, 114,12,34]
print(len(List_num))
```

List Slicing

```
List_A = ['A','B','C','D','E','F','G']
print(List_A[2:5])
```

In the above list note that the range is from 2(Inclusive) to 5(exclusive). Therefore, values at index position 2,3 and 4 will be printed.

Another point to keep in mind is that a the first element of the list has index 0 (zero).

Therefore the output will be: ['C', 'D', 'E']

INDEX	0	1	2	3	4	5	6
ELEMENTS	A	B	C	D	E	F	G

```
List_A = ['A','B','C','D','E','F','G']
print(List_A[:4])
```

In the above list note that the range will begin from 0th index to the 3rd index.

Therefore values at index position 0,1,2, and 3 will be printed.

Therefore the output will be: ['A', 'B', 'C', 'D']

Negative Indexing in a List

- Negative indexing means start from the end of the list.
- Therefore an index value of [-1] refers to the last item of the list.
- Similarly an index value of [-2] refers to the second last item of the list and so on.

Example

```
List = [1, 2, 'a', 4, 'b', 6, 'c']
print(List[-1]) # prints c
print(List[-3]) # prints b
print(List[-5]) # prints a
```

s	p	y	d	e	r
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

remove method

Output:
List after Removal of two elements:
[1, 2, 3, 4, 7]

```
List = [1, 2, 3, 4, 5, 6, 7]
print("Initial List: ")
print(List)
# Removing elements from List using remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)
```

Removing elements from List using iterator method

```
List=[1,2,3,4,5,6,7,8,9]
for i in range(1,5):
        List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)
```

Output:
List after Removing a range of elements:
[5, 6, 7, 8, 9]

list operation Converts an iterable (tuple ,string, set, dictionary) to a list.

For example consider the following code.

```
list1=list("MORNING")
print(list1)
```

```
Output:
['M', 'O', 'R', 'N', 'I', 'N', 'G']
```

Adding elements to list using append.

```
List_new = []
print("Initial blank List: ")
print(List_new)
# Addition of Elements
List_new.append(10)
List_new.append(20)
List_new.append(30)
print("\nList after Addition of Three elements: ")
print(List_new)
```

Few List Methods

```
ListA = [2, 4, 2, 6, 7, 2]
print(ListA.count(2))      #counts no. of times element appears
ListA.insert(2,50)         #Insert object at specific index
print(ListA)
print(ListA.pop(2))        #Removes an element at the specified index
ListA.remove(2)            #removes or deletes and element from the list
print(ListA)
```

reverse and sort method

```
ListA = [2, 4, 2, 6, 7, 2]
ListA.reverse()          #Reverses the elements in the list
print(ListA)
ListA.sort()             #Sorts the elements in the list
print(ListA)
```

extend method

```
ListA = [2, 4, 2, 6, 7, 2]
ListB = [100,200,300]
ListA.extend(ListB)      #adds elements in a list to the end of another
                          list
print(ListA)
```

```
List_A = [1, 3, 6, 2] + [5, 4, 7]          #Concatenation
print (List_A)
print ("Max =", max(List_A))                #prints maximum value
print ("Min =", min(List_A))                #prints minimum value
print ("Sum =", sum(List_A))                #sum operation add
                                            the values in the list of
                                            numbers
```

Operations on Lists

- Concatenation
- Repetition
- Membership
- Slicing

4. Slicing

```
>>> list1 = ['Varun', 'Ravinder', 'Amrit', 'Nitin', 'Rahul', 'Prajwal']
>>> list1[2:4]
['Amrit', 'Nitin']
```

- Practice more!!!!!!!!!!

```
a[start: stop]   # items start through stop-1
a[start:]        # items start through the rest of the array
a[:stop]         # items from the beginning through stop-1
a[:]             # a copy of the whole array
a[-1]            # last item in the array
a[-2:]           # last two items in the array
a[:-2]           # everything except the last two items
a[::-1]          # all items in the array, reversed
a[1::-1]         # the first two items, reversed
a[:-3:-1]        # the last two items, reversed
a[-3::-1]        # everything except the last two items, reversed
```

Practice Problems

- Program to read a list of names and sort in alphabetic order
- Program to remove all duplicate elements from a list
- Consider a list consisting of integers, floating point numbers and strings. Separate them into different lists depending on the data

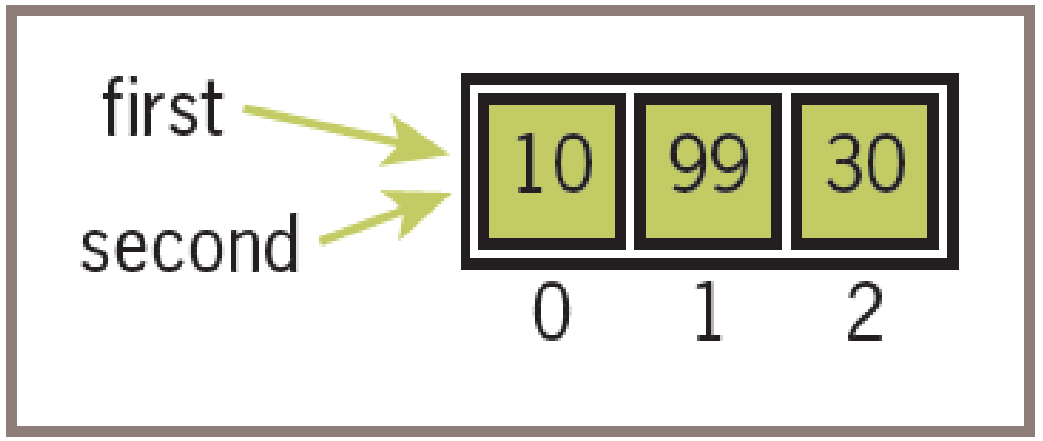
Mutator Methods and the Value None

- Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object. Such methods are called mutators.
- Examples are the *list* methods insert, append, extend, pop, and sort.
- Because a change of state is all that is desired, a mutator method usually returns no value of interest to the caller (but note that pop is an exception to this rule).
- Python nevertheless automatically returns the special value *None* even when a method does not explicitly return a value.

```
aList = aList.sort()
print(aList)                    None
```

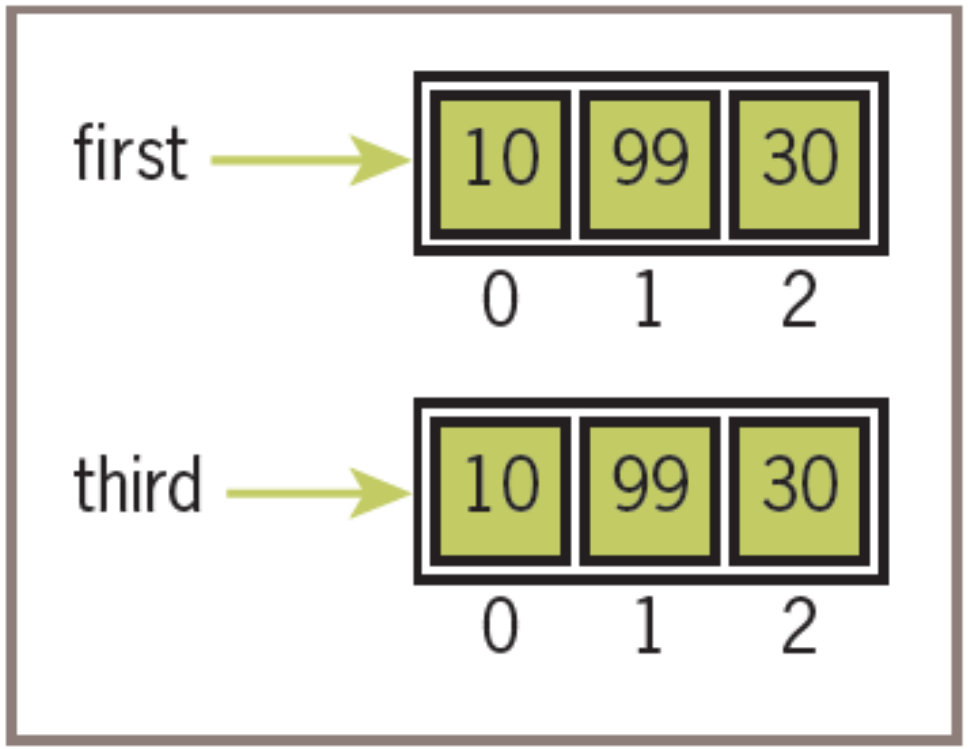
Aliasing and Side Effects

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
[10, 99, 30]
>>> second
[10, 99, 30]
```



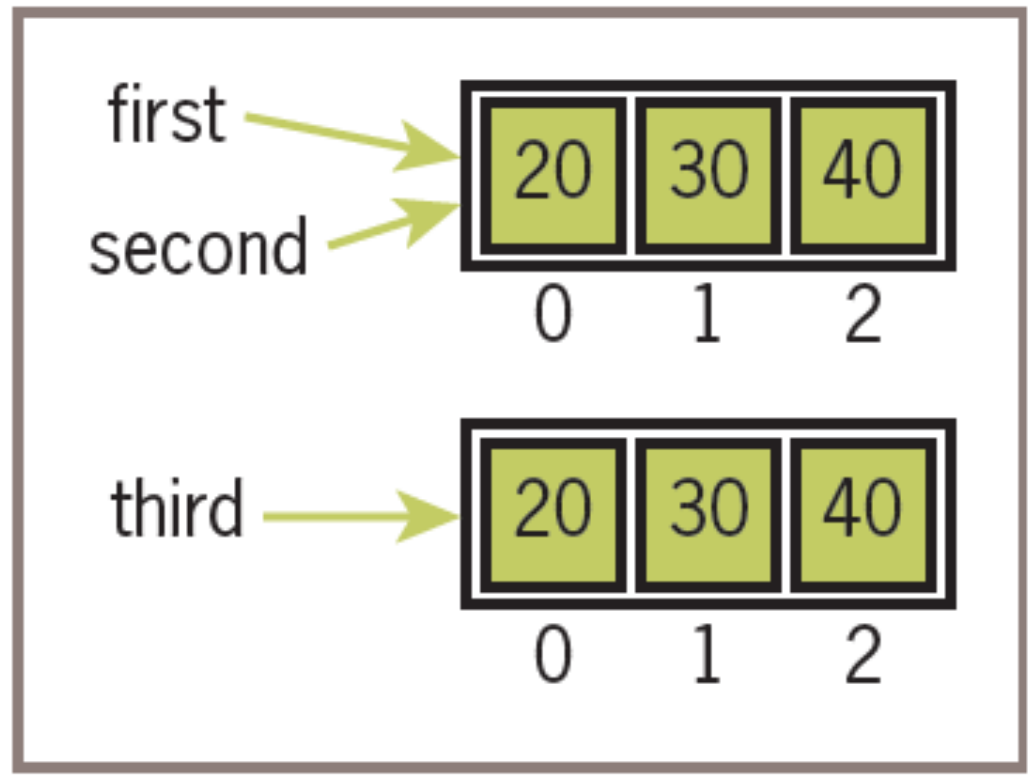
Two variables refer to the same list object

```
>>> third = []
>>> for element in first:
        third.append(element)
>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
>>> first[1] = 100
>>> first
[10, 100, 30]
>>> third
[10, 99, 30]
```



Two variables refer to different list objects

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = list(first)
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```



Three variables and two distinct list objects

List Comprehensions

- List comprehensions provide a concise way to create lists.

For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
squares = [x**2 for x in range(10)]
```

- A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.
- The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

Nested List Comprehensions

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Practice problems

- Program to read a list of numbers and find the median
- Find the mode of a list of numbers
- Program to sort a list of numbers
- Matrix addition

Sequences

- Series of items that are often related
- 3 types of sequences
  - Strings
  - Lists
  - Tuples
- Tuples contain data that are related but not necessarily of the same type
- Tuple items are ordered, unchangeable, and allow duplicate values

  For example, a person’s name, age and birth date

Tuples

- A *tuple* is a type of sequence that resembles a list, except that, unlike a list, a *tuple is immutable*.
- You indicate a tuple literal in Python by enclosing its elements in parentheses instead of square brackets.
- Tuples function much like lists
- Each value in the tuple is an element or item
- Elements can be any Python data type
  - Tuples can have mixed data types
  - Elements can be nested tuples

Dictionaries

- Lists organize their elements by position. This mode of organization is useful when you want to locate the first element, the last element, or visit each element in a sequence.
- However, in some situations, the position of a datum in a structure is irrelevant; we're interested in its association with some other element in the structure.
- A dictionary organizes information by association, not position.
- Phone books, address books, encyclopedias, and other reference sources also organize information by association.
- In Python, a dictionary associates a set of *keys* with *values*.
- Dictionary comprise the set of words, whereas the associated data values are their definitions.

Dictionaries

- A Python dictionary is written as a sequence of key/value pairs separated by commas.
- These pairs are sometimes called entries.
- The entire sequence of entries is enclosed in curly braces { and }
- A colon (:) separates a key and its value.

Example:

A phone book: {"Savannah":"476-3321", "Nathaniel":"351-7743"}

Personal information: {"Name":"Molly", "Age":18}

Adding Keys and Replacing Values

- You add a new key/value pair to a dictionary by using the subscript operator []

<a dictionary>[<a key>] = <a value>

- The next code segment creates an empty dictionary and adds two new entries:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name':'Sandy', 'occupation':'hacker'}
```

- The subscript is also used to replace a value at an existing key, as follows:

```
>>> info["occupation"] = "manager"
>>> info
{'name':'Sandy', 'occupation':'manager'}
```

Accessing Values

```
>>> info["name"]
'Sandy'
>>> info["job"]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    info["job"]
KeyError: 'job'
```

Removing Keys

- To delete an entry from a dictionary, removes its key using the method *pop*.
- This method expects a key and an optional default value as arguments.
- If the key is in the dictionary, it is removed, and its associated value is returned. Otherwise, the default value is returned.
- If pop is used with just one argument, and this key is absent from the dictionary, Python raises an exception.

```
>>> print(info.pop("job", None))
None
>>> print(info.pop("occupation"))
manager
>>> info
{'name':'Sandy'}
```

Traversing a Dictionary

- When a for loop is used with a dictionary, the loop's variable is bound to each key in an unspecified order.

```
for key in info:
    print(key, info[key])
```

- Alternatively, you could use the dictionary method items() to access the dictionary's entries.

```
>>> grades = {90:'A', 80:'B', 70:'C'}
>>> list(grades.items())
[(80,'B'), (90,'A'), (70,'C')]
```

- Note that the entries are represented as tuples within the list. A tuple of variables can then access the key and value of each entry in this list within a *for* loop:

```
for (key, value) in grades.items():
    print(key, value)
```

```
theKeys = list(info.keys())
theKeys.sort()
for key in theKeys:
    print(key, info[key])
```

Python Functions

- Types of Functions
  - Python support two types of functions
    - Built-in function
    - User-defined function

  - Built-in function

    The functions which come along with Python itself are called a built-in function or predefined function. Eg: range(), id(), type(), input(), eval() etc.
  - User-defined function

    Functions which are created by programmer explicitly according to the requirement are called a user-defined function.

Creating a Function

Use the following steps to define a function in Python.

- Use the ***def*** keyword with the function name to define a function.
- Next, pass the number of parameters as per your requirement. (Optional).
- Next, define the function body with a block of code. This block of code is nothing but the action you wanted to perform.

In Python, no need to specify curly braces for the function body. The only indentation is essential to separate code blocks. Otherwise, you will get an error.

Syntax of creating a function

```
def function_name(parameter1, parameter2):
        # function body
        # write some action
return value
```

Creating a function without any parameters

```
# function
def message():
    print("Welcome to CSE")

# call function using its name
message()
```

```
In [2]: runfile('C:/Users/
Zenbook/untitled3.py', wdir='C
Users/Zenbook')
Welcome to CSE
```

Creating a function with parameters

```
# function
def course_func(name, course_name):
    print("Hello", name, "Welcome to CSE")
    print("Your course name is", course_name)

# call function
course_func('John', 'Python')
```

```
In [3]: runfile('C:/Users/
Zenbook/untitled3.py', wdir='C:/
Users/Zenbook')
Hello John Welcome to CSE
Your course name is Python
```

Creating a function with parameters and return value

- Functions can return a value. The return value is the output of the function. Use the return keyword to return value from a function.

```python
# function
def calculator(a, b):
    add = a + b
    # return the addition
    return add

# call function
# take return value in variable
res = calculator(20, 5)

print("Addition :", res)
# Output Addition : 25
```

Calling a function of a module

- You can take advantage of the built-in module and use the functions defined in it.
- For example, Python has a random module that is used for generating random numbers and data. It has various functions to create different types of random data.

Let’s see how to use functions defined in any module.

- First, we need to use the import statement to import a specific function from a module.
- Next, we can call that function by its name.

```python
# import randint function
from random import randint

# call randint function to get random number
print(randint(10, 20))
# Output 14
```

Return Value From a Function

- In Python, to return value from the function, a ***return*** statement is used. It returns the value of the expression following the returns keyword.

Syntax of return statement

```
def fun():
    statement-1
    statement-2
    statement-3
    .
    .
    return [expression]
```

Return Multiple Values

- You can also return multiple values from a function.
- Use the return statement by separating each expression by a comma.

```
def arithmetic(num1, num2):
    add = num1 + num2
    sub = num1 - num2
    multiply = num1 * num2
    division = num1 / num2
    # return four values
    return add, sub, multiply, division

# read four return values in four variables
a, b, c, d = arithmetic(10, 2)

print("Addition: ", a)
print("Subtraction: ", b)
print("Multiplication: ", c)
print("Division: ", d)
```

The pass Statement

- In Python, the pass is the keyword, which won’t do anything. Sometimes there is a situation where we need to define a syntactically empty block. We can define that block using the pass keyword.

- When the interpreter finds a pass statement in the program, it returns no operation.

```python
def addition(num1, num2):
    # Implementation of addition function in comming release
    # Pass statement
    pass

addition(10, 2)
```

Scope and Lifetime of Variables

- When we define a function with variables, then those variables' scope is limited to that function.
- In Python, the scope of a variable is an area where a variable is declared. It is called the variable's local scope.

- We cannot access the local variables from outside of the function. Because the scope is local, those variables are not visible from the outside of the function.
- **Note**: The inner function does have access to the outer function's local scope.

- When we are executing a function, the life of the variables is up to running time.
- Once we return from the function, those variables get destroyed.
- So function does no need to remember the value of a variable from its previous call.

```
global_lang = 'DataScience'

def var_scope_test():
    local_lang = 'Python'
    print(local_lang)

var_scope_test()
# Output 'Python'

# outside of function
print(global_lang)
# Output 'DataScience'

# NameError: name 'local_lang' is not defined
print(local_lang)
```

Local Variable in function

- A local variable is a variable declared inside the function that is not accessible from outside of the function.
- The scope of the local variable is limited to that function only where it is declared.

- If we try to access the local variable from the outside of the function, we will get the error as NameError.

```
def function1():
    # local variable
    loc_var = 888
    print("Value is :", loc_var)

def function2():

    print("Value is :", loc_var)

function1()
function2()
```

```
Value is : 888
print("Value is :", loc_var) # gives error,
NameError: name 'loc_var' is not defined
```

Global Variable in function

- A Global variable is a variable that is declared outside of the function.
- The scope of a global variable is broad.
- It is accessible in all functions of the same module.

```
global_var = 999

def function1():
    print("Value in 1nd function :", global_var)

def function2():
    print("Value in 2nd function :", global_var)

function1()
function2()
```

```
Value in 1nd function : 999
Value in 2nd function : 999
```

```python
# Global variable
global_var = 5

def function1():
    print("Value in 1st function :", global_var)

def function2():
    # Modify global variable
    # function will treat it as a local variable
    global_var = 555
    print("Value in 2nd function :", global_var)

def function3():
    print("Value in 3rd function :", global_var)

function1(
function2()
function3()
```

```
Value in 1st function : 5
Value in 2nd function : 555
Value in 3rd function : 5
```

As you can see, `function2()` treated `global_var` as a new variable (local variable). To solve such issues or access/modify global variables inside a function, we use the `global` keyword.

Global Keyword in Function

- In Python, ***global*** is the keyword used to access the actual global variable from outside the function. we use the ***global*** keyword for two purposes:

- To declare a global variable inside the function.
- Declaring a variable as global, which makes it available to function to perform the modification.

```python
# Global variable
x = 5

# defining 1st function
def function1():
    print("Value in 1st function :", x)

# defining 2nd function
def function2():
    # Modify global variable using global keyword
    global x
    x = 555
    print("Value in 2nd function :", x)

# defining 3rd function
def function3():
    print("Value in 3rd function :", x)

function1()
function2()
function3()
```

```
Value in 1st function : 5
Value in 2nd function : 555
Value in 3rd function : 555
```

Nonlocal Variable in Function

- In Python, nonlocal is the keyword used to declare a variable that acts as a global variable for a nested function (i.e., function within another function).

- We can use a nonlocal keyword when we want to declare a variable in the local scope but act as a global scope.

```
def outer_func():
    x = 777

    def inner_func():
        # local variable now acts as global variable
        nonlocal x
        x = 700
        print("value of x inside inner function is :", x)

    inner_func()
    print("value of x inside outer function is :", x)

outer_func()
```

```
value of x inside inner function is : 700
value of x inside outer function is : 700
```

Python Function Arguments

- The argument is a value, a variable, or an object that we pass to a function or method call.
- In Python, there are four types of arguments allowed.

  - Positional arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Positional Arguments

- Positional arguments are arguments that are pass to function in proper positional order.
- That is, the 1st positional argument needs to be 1st when the function is called. The 2nd positional argument needs to be 2nd when the function is called, etc.

```
def add(a, b):
    print(a - b)


add(50, 10)
# Output 40
add(10, 50)
# Output -40
```

- If you try to use pass more parameters you will get an error.

```
def add(a, b):
    print(a - b)

add(105, 561, 4)
```

```
TypeError: add() takes 2 positional arguments but 3 were given
```

In the positional argument number and position of arguments must be matched. If we change the order, then the result may change. Also, If we change the number of arguments, then we will get an error.

Keyword Arguments

- A keyword argument is an argument value, passed to function preceded by the variable name and an equals sign.

```
def message(name, surname):
    print("Hello", name, surname)

message(name="John", surname="Wilson")
message(surname="Ault", name="Kelly")
```

```
Hello John Wilson
Hello Kelly Ault
```

- In keyword arguments order of argument is not matter, but the number of arguments must match. Otherwise, we will get an error.
- While using keyword and positional argument simultaneously, we need to pass 1st arguments as positional arguments and then keyword arguments. Otherwise, we will get Syntax Error.

```
def message(first_nm, last_nm):
    print("Hello..!", first_nm, last_nm)

# correct use
message("John", "Wilson")
message("John", last_nm="Wilson")

# Error
# SyntaxError: positional argument follows keyword argument
message(first_nm="John", "Wilson")
```

Default Arguments

- Default arguments take the default value during the function call if we do not pass them.
- We can assign a default value to an argument in function definition using the = assignment operator.
- When we call a function with an argument, it will take that value.

```
# function with default argument
def message(name="Guest"):
    print("Hello", name)

# calling function with argument
message("John")

# calling function without argument
message()
```

```
Hello John
Hello Guest
```

Variable-length Arguments

- In Python, sometimes, there is a situation where we need to pass multiple numbers of arguments to the function.
- Such types of arguments are called variable-length arguments.
- We can declare a variable-length argument with the * (asterisk) symbol.
- We can pass any number of arguments to this function. Internally all these values are represented in the form of a tuple.

```python
def addition(*numbers):
    total = 0
    for no in numbers:
        total = total + no
    print("Sum is:", total)


# 0 arguments
addition()

# 5 arguments
addition(10, 5, 2, 5, 4)


# 3 arguments
addition(78, 7, 2.5)
```

```
Sum is: 0
Sum is: 26
Sum is: 87.5
```

There are many advantages of using functions:

- They reduce duplication of code in a program.
- They break the large complex problems into small parts.
- They help in improving the clarity of code.
- A piece of code can be reused as many times as we want with the help of functions

Recursive Function

- A recursive function is a function that calls itself, again and again.
- To prevent a function from repeating itself indefinitely, it must contain at least one selection statement.
- This statement examines a condition called a base case to determine whether to stop or to continue with another recursive step

```python
def factorial(no):
    if no == 0:
        return 1
    else:
        return no * factorial(no - 1)

print("factorial of a number is:", factorial(8))
```

Practice Problems

- Write a Python program to print n^{th} Fibonacci number using recursion.
- Write a python program to compute nCr using a factorial function.
- Write a Python program to find the value for sin(x) up to n terms using the series

  sin(x)=1-x^3/3!+x^5/5!.....

- The advantages of the recursive function are:
    - By using recursive, we can reduce the length of the code.
    - The readability of code improves due to code reduction.
    - Useful for solving a complex problem
- The disadvantage of the recursive function:
    - The recursive function takes more memory and time for execution.
    - Debugging is not easy for the recursive function.

Python Anonymous/Lambda Function