



GOVERNMENT POLYTECHNIC AMRAVATI

DIPLOMA PROGRAMME IN COMPUTER ENGINEERING

COURSE : PROGRAMMING WITH PYTHON
COURSE CODE :CM5461

Unit 3

Data Structures

UNIT 3 :- CONTENTS

1

Python List

2

Tuples

3

Sets

4

Dictionaries

COURSE OUTCOMES (COs)

At the end of this course, student will be able to: -

- Write and execute simple 'Python' programs.
- Write 'Python' programs using arithmetic expressions and control structure.
- Develop 'Python' programs using List, Tuples and Dictionary.
- Develop/Use functions in Python programs for modular programming approach.
- Develop 'Python' programs using File Input/output operations.
- Write 'Python' code using Classes and Objects.

Python Data Types

- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:
 - Text Type : str
 - Numeric Types : int, float, complex
 - Sequence Types : list, tuple, range

Python Data Types

- Mapping Type : dict
- Set Types : set, frozenset
- Boolean Type : bool
- Binary Types : bytes, bytearray, memoryview
- **Number data types store numeric values.** They are immutable data types, means that changing the value of a number data type results in a newly allocated object. (supports int, long, float, complex data types)

Python Data Types

- `var1 = 1 var2 = 10`
- You can also delete the reference to a number object by using the `del` statement. The syntax of the `del` statement is –
 - `del var1[,var2[,var3[....,varN]]]`
- You can delete a single object or multiple objects by using the `del` statement. For example –
 - `del var`
 - `del var_a, var_b`

Python Data Types

Number Type Conversion

- Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.
- Type `int(x)` to convert `x` to a plain integer.
- Type `long(x)` to convert `x` to a long integer.
- Type `float(x)` to convert `x` to a floating-point number.

Python Data Types

- Type `complex(x)` to convert `x` to a complex number with real part `x` and imaginary part zero.
- Type `complex(x, y)` to convert `x` and `y` to a complex number with real part `x` and imaginary part `y`. `x` and `y` are numeric expressions
- **Mathematical Functions**
- Python includes following functions that perform mathematical calculations.
`abs(x)`, `ceil(x)`, `cmp(x)`, `exp(x)`, `fabs(x)`, `floor(x)`, `pow(x,y)` etc

Python Data Types

Random Number Functions

- Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

- `import random`

```
print(random.randrange(10, 20))
```

```
x = ['a', 'b', 'c', 'd', 'e']
```

- `# Get random choice`

```
print(random.choice(x))
```

Python Data Types

- # Shuffle x

```
random.shuffle(x)
```

- # Print the shuffled x

```
print(x)
```

- # Print random element

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

- 18

- e

- ['c', 'e', 'd', 'b', 'a']

- 0.5682821194654443

Python Data Types

- Python offers a range of compound data types often referred to as sequences. List is one of the most frequently used and very versatile data types used in Python.
- In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.
- It can have any number of items and they may be of different types (integer, float, string etc.).

Python Data Types

- # empty list

```
list1 = []
```

- # list of integers

```
list2 = [1, 2, 3]
```

- # list with mixed data types

```
list3 = [1, "Hello", 3.4]
```

- A list can also have another list as an item. This is called a nested list.

```
# nested list
```

```
list4 = ["mouse", [8, 4, 6], ['a']]
```

Python Data Types

Access List Elements

- There are various ways in which we can access the elements of a list.
- List Index: We can use the index operator `[]` to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4. $n = \text{no of element}$
- Trying to access indexes other than these will raise an `IndexError`. The index must be an integer. We can't use float or other types, this will result in `TypeError`.

Python Data Types

Access List Elements

- Nested lists are accessed using nested indexing.

```
my_list = ['p', 'r', 'o', 'b', 'e']
```

```
# Output: p
```

```
print(my_list[0])
```

```
# Output: o
```

- ```
print(my_list[2])
```

- ```
# Output: e
```

-

```
GPA print(my_list[4])
```

Python Data Types

p

o

E

a

5

Traceback (most recent call last): File "<string>", line 21, in
<module>

TypeError: list indices must be integers or slices, not float

Python Data Types

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

Negative indexing in lists

```
my_list = ['p','r','o','b','e']
```

```
print(my_list[-1])
```

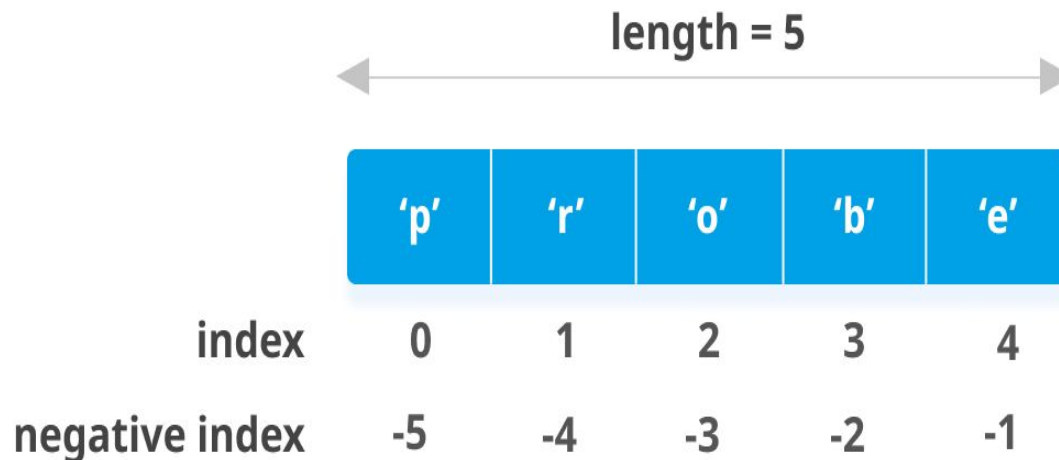
```
print(my_list[-5])
```


Python Data Types

When we run the above program, we will get the following output:

e

p



Python Data Types

Slicing of List

We can access a range of items in a list by using the slicing operator :(colon).

- # List slicing in Python

```
my_list = ['p','r','o','g','r','a','m','i','z']
```

```
# elements 3rd to 5th
```

```
print(my_list[2:5])
```

```
# elements beginning to 4th
```

```
print(my_list[:5])
```

```
# elements 6th to end
```

```
print(my_list[5:])
```

```
# elements beginning to end print(my_list[:])
```

```
['o', 'g', 'r']
```

```
['p', 'r', 'o', 'g']
```

```
['a', 'm', 'i', 'z']
```

```
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Python Data Types

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.

P	R	O	G	R	A	M	I	Z	
0	1	2	3	4	5	6	7	8	9
-9	-8	-7	-6	-5	-4	-3	-2	-1	

Python Data Types

- Python List Comprehension
- List Comprehension is defined as an elegant way to define, create a list in Python and consists of brackets that contains an expression followed by **for** clause. It is efficient in both computationally and in terms of coding space and time.
- Signature
- The list comprehension starts with '[' and ']'.
- [expression **for** item **in** list **if** conditional]
- letters = []
- **for** letter **in** 'Python':
- letters.append(letter)
- **print**(letters)

Python Data Types

```
letters = [ letter for letter in 'Python' ]  
print( letters)
```

```
x = {'chrome': 'browser', 'Windows': 'OS', 'C': 'language'}  
x['mouse'] = 'hardware'  
print(x['Windows'])
```

Python Data Types

- List indexing and splitting
- The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].
- The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.
- We can get the sub-list of the list using the following syntax.
- `list_variable(start:stop:step)`
- The **start** denotes the starting index position of the list.
- The **stop** denotes the last index position of the list.
- The **step** is used to skip the nth element within a **start:stop**

Python Data Types

List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5
---	---	---	---	---	---

List[0] = 0

List[0:] = [0,1,2,3,4,5]

List[1] = 1

List[:] = [0,1,2,3,4,5]

List[2] = 2

List[2:4] = [2, 3]

List[3] = 3

List[1:3] = [1, 2]

List[4] = 4

List[:4] = [0, 1, 2, 3]

List[5] = 5

Python Data Types

List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5
---	---	---	---	---	---

List[0] = 0

List[0:] = [0,1,2,3,4,5]

List[1] = 1

List[:] = [0,1,2,3,4,5]

List[2] = 2

List[2:4] = [2, 3]

List[3] = 3

List[1:3] = [1, 2]

List[4] = 4

List[:4] = [0, 1, 2, 3]

List[5] = 5

Python Data Types

Add/Change List Elements

Lists are mutable, meaning their elements can be changed unlike string or tuple. We can use the assignment operator = to change an item or a range of items.

```
# Correcting mistake values in a list
```

```
odd = [2, 4, 6, 8]
```

```
# change the 1st item
```

```
odd[0] = 1
```

```
print(odd)
```

```
# change 2nd to 4th items
```

```
odd[1:4] = [3, 5, 7]
```

```
print(odd)
```

```
O/P : [1, 4, 6, 8] [1, 3, 5, 7]
```

Python Data Types

We can add one item to a list using the `append()` method or add several items using `extend()` method.

Appending and Extending lists in Python

```
odd = [1, 3, 5]
```

```
odd.append(7)
```

```
print(odd)
```

```
odd.extend([9, 11, 13])
```

```
print(odd)
```

```
[1, 3, 5, 7]
```

```
[1, 3, 5, 7, 9, 11, 13]
```

Python Data Types

We can also use + operator to combine two lists. This is also called concatenation.

The * operator repeats a list for the given number of times.

Concatenating and repeating lists

```
odd = [1, 3, 5]
```

```
print(odd + [9, 7, 5])
```

```
print(["re"] * 3)
```

```
[1, 3, 5, 9, 7, 5]
```

```
['re', 're', 're']
```

Python Data Types

Furthermore, we can insert one item at a desired location by using the method `insert()` or insert multiple items by squeezing it into an empty slice of a list.

Demonstration of list `insert()` method

```
odd = [1, 9]
    odd.insert(1,3)
print(odd)
odd[2:2] = [5, 7]
print(odd)
```

Output [1, 3, 9] [1, 3, 5, 7, 9]

Python Data Types

Delete/Remove List Elements

We can delete one or more items from a list using the keyword **del**. It can even delete the list entirely.

```
# Deleting list items
```

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
```

```
# delete one item
```

```
del my_list[2]
```

```
print(my_list)
```

```
# delete multiple items
```

```
del my_list[1:5]
```

```
print(my_list)
```

```
# delete entire list
```

```
del my_list
```

```
# Error: List not defined
```

```
print(my_list)
```

Python List count()

The count() method returns the number of times the specified element appears in the list.

The syntax of the count() method is:

```
list.count(element)
```

count() Parameters

The count() method takes a single argument:

element - the element to be counted

Return value from count()

The count() method returns the number of times element appears in the list.

Python Data Types

```
# vowels list vowels = ['a', 'e', 'i', 'o', 'i', 'u']
# count element 'i'
count = vowels.count('i')
# print
count print('The count of i is:', count)
# count element 'p'
count = vowels.count('p')
# print count print('The count of p is:', count)
```

Python Data Types

```
# Operating System
```

```
List
```

```
systems = ['Windows', 'macOS', 'Linux']
```

```
print('Original List:', systems)
```

```
# List Reverse
```

```
systems.reverse()
```

```
# updated list print('Updated List:', systems)
```


Python Data Types

Output :['p', 'r', 'b', 'l', 'e', 'm']

['p', 'm']

Traceback (most recent call last):

File "<string>", line 18, in <module>

NameError: name 'my_list' is not defined

We can use remove() method to remove the given item or pop() method to remove an item at the given index.

The pop() method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the clear() method to empty a list.

```
my_list = ['p','r','o','b','l','e','m']
```

```
my_list.remove('p')
```

Python Data Types

```
# Output: ['r', 'o', 'b', 'l', 'e', 'm']  
print(my_list)  
# Output: 'o'  
print(my_list.pop(1))  
# Output: ['r', 'b', 'l', 'e', 'm']  
print(my_list)  
# Output: 'm'  
print(my_list.pop())  
# Output: ['r', 'b', 'l', 'e']  
print(my_list)  
    my_list.clear()
```

Python Data Types

```
# Output: []  
print(my_list)
```

Output

```
['r', 'o', 'b', 'l', 'e', 'm']
```

o

```
['r', 'b', 'l', 'e', 'm']
```

m

```
['r', 'b', 'l', 'e']
```

```
[]
```

Python Data Types

Following built-in functions can be used along with List as well as Tuple.

`len()` Returns number of elements in list/tuple

`max()` If list/tuple contains numbers, largest number will be returned. If list/tuple contains strings, one that comes last in alphabetical order will be returned.

`min()` If list/tuple contains numbers, smallest number will be returned. If list/tuple contains strings, one that comes first in alphabetical order will be returned.

`sum()` Returns addition of all elements in list/tuple

`sort()` Sorts the elements in list/tuple

Python Data Types

```
L1=[10,30,50,20,40]
T1=(10,50,30,40,20)
print (len(L1))
print (len(T1))
print ('max of L1', max(L1))
print ('max of T1', max(T1))
print ('min of L1', min(L1))
print ('min of T1', min(T1))
print ('sum of L1', sum(L1))
print ('sum of T1', sum(T1))
print ('L1 in sorted order', sorted(L1))
print ('T1 in sorted order', sorted(T1))
```

Python Data Types

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them. A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.)

Python Data Types

Empty tuple

```
my_tuple = ()
```

```
print(my_tuple)
```

Tuple having integers

```
my_tuple = (1, 2, 3)
```

```
print(my_tuple)
```

tuple with mixed datatypes

```
my_tuple = (1, "Hello", 3.4)
```

```
print(my_tuple)
```

Python Data Types

```
# nested tuple
```

```
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
```

```
print(my_tuple)
```

```
()
```

```
(1, 2, 3)
```

```
(1, 'Hello', 3.4)
```

```
('mouse', [8, 4, 6], (1, 2, 3))
```

A tuple can also be created without using parentheses. This is known as tuple packing.

Python Data Types

```
my_tuple = 3, 4.6, "dog"
```

```
print(my_tuple)
```

```
# tuple unpacking is also possible
```

```
a, b, c = my_tuple
```

```
print(a) # 3
```

```
print(b) # 4.6
```

```
print(c) # dog
```

```
(3, 4.6, 'dog')
```

```
3
```

```
4.6
```

```
dog
```

Python Data Types

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s. This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.

Python Data Types

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

defining strings in Python

all of the following are equivalent

```
my_string = 'Hello'
```

```
print(my_string)
```

Python Data Types

```
my_string = "Hello"
print(my_string)
my_string = "Hello"
print(my_string)
# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
    the world of Python"""
print(my_string)
Hello
Hello
Hello
Hello, welcome to
the world of Python
```

Python Data Types

```
my_string = "Hello"
print(my_string)
my_string = "Hello"
print(my_string)
# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
    the world of Python"""
print(my_string)
Hello
Hello
Hello
Hello, welcome to
the world of Python
```

Python Data Types

Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.

Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces `{}` separated by commas.

An item has a key and a corresponding value that is expressed as a pair (key: value).

While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Python Data Types

empty dictionary

```
my_dict = {}
```

dictionary with integer keys

```
my_dict = {1: 'apple', 2: 'ball'}
```

dictionary with mixed keys

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

using dict()

```
my_dict = dict({1:'apple', 2:'ball'})
```

from sequence having each item as a pair

```
my_dict = dict([(1,'apple'), (2,'ball')])
```

Python Data Types

```
# empty dictionary
```

```
my_dict = {}
```

```
# dictionary with integer keys
```

```
my_dict = {1: 'apple', 2: 'ball'}
```

```
# dictionary with mixed keys
```

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

we can also create a dictionary using the built-in dict() function.

```
# using dict()
```

```
my_dict = dict({1:'apple', 2:'ball'})
```

```
# from sequence having each item as a pair
```

```
my_dict = dict([(1,'apple'), (2,'ball')])
```


Python Data Types

Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

Changing and adding Dictionary Elements

```
my_dict = {'name': 'Jack', 'age': 26}
```

update value

```
my_dict['age'] = 27
```

```
#Output: {'age': 27, 'name': 'Jack'}
```

```
print(my_dict)
```

add item

```
my_dict['address'] = 'Downtown'
```

```
# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
```

```
print(my_dict)
```

Python Data Types

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed). However, a set itself is mutable. We can add or remove items from it. Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Python Sets

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in `set()` function.

Python Data Types

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists or dictionaries as its elements

Different types of sets in Python

set of integers

```
my_set = {1, 2, 3}
```

```
print(my_set) # set of mixed datatypes
```

```
my_set = {1.0, "Hello", (1, 2, 3)}
```

```
print(my_set)
```

```
{1, 2, 3}
```

```
{1.0, (1, 2, 3), 'Hello'}
```

Python Data Types

set cannot have duplicates

Output: {1, 2, 3, 4}

```
my_set = {1, 2, 3, 4, 3, 2}
```

```
print(my_set)
```

we can make set from a list

Output: {1, 2, 3}

```
my_set = set([1, 2, 3, 2])
```

```
print(my_set)
```

Python Data Types

set cannot have mutable items

here [3, 4] is a mutable list

this will cause an error.

```
my_set = {1, 2, [3, 4]}
```

Python Data Types

Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument

```
# initialize a with {}
```

```
a = {}
```

```
# check data type of a
```

```
print(type(a))
```

```
# initialize a with set()
```

```
a = set()
```

```
# check data type of a
```

```
print(type(a))
```

Python Data Types

Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the `add()` method, and multiple elements using the `update()` method. The `update()` method can take [tuples](#), lists, [strings](#) or other sets as its argument. In all cases, duplicates are avoided.

Python Data Types

A particular item can be removed from a set using the methods `discard()` and `remove()`.

The only difference between the two is that the `discard()` function leaves a set unchanged if the element is not present in the set. On the other hand, the `remove()` function will raise an error in such a condition (if element is not present in the set).

Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Union of A and B is a set of all elements from both sets.

Union is performed using | operator. Same can be accomplished using the union() method.

Set union method

initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

use | operator

Output: {1, 2, 3, 4, 5, 6, 7, 8}

print(A | B)

Python Data Types

Intersection of A and B is a set of elements that are common in both the sets.

Intersection is performed using & operator. Same can be accomplished using the intersection() method.

```
# Intersection of sets
```

```
# initialize A and B
```

```
A = {1, 2, 3, 4, 5}
```

```
B = {4, 5, 6, 7, 8}
```

```
# use & operator
```

```
# Output: {4, 5}
```

```
print(A & B)
```

Python Data Types

Difference of the set B from set A ($A - B$) is a set of elements that are only in A but not in B. Similarly, $B - A$ is a set of elements in B but not in A.

Difference is performed using `-` operator. Same can be accomplished using the `difference()` method.

Python Data Types

Symmetric Difference of A and B is a set of elements in A and B but not in both (excluding the intersection).

Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric_difference().

```
# Symmetric difference of two sets
```

```
# initialize A and B
```

```
A = {1, 2, 3, 4, 5}
```

```
B = {4, 5, 6, 7, 8}
```

```
# use ^ operator
```

```
# Output: {1, 2, 3, 6, 7, 8}
```

```
print(A ^ B)
```

Python Data Types

Set Membership Test

We can test if an item exists in a set or not, using the in keyword.

in keyword in a set

initialize

```
my_set my_set = set("apple")
```

check if 'a' is present

Output: True

```
print('a' in my_set)
```

check if 'p' is present

Output: False

```
print('p' not in my_set)
```

Iterating Through a Set

We can iterate through each item in a set using a for loop.

```
for letter in set("apple"):
```

```
    print(letter)
```

Built-in Functions with Set

Built-in functions

like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` etc

. are commonly used with sets to perform different tasks.

Python Data Types

Function	Description
<code>all()</code>	Returns True if all elements of the set are true (or if the set is empty).
<code>any()</code>	Returns True if any element of the set is true. If the set is empty, returns False.
<code>enumerate()</code>	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
<code>len()</code>	Returns the length (the number of items) in the set.
<code>max()</code>	Returns the largest item in the set.
<code>min()</code>	Returns the smallest item in the set.
<code>sorted()</code>	Returns a new sorted list from elements in the set(does not sort the set itself).
<code>sum()</code>	Returns the sum of all elements in the set.

Operators in Python

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:

```
>>> 2+3 O/P is 5
```

Types of Operator

Python language supports the following types of operators.

Arithmetic Operators

Comparison (Relational) Operators

Assignment Operators

Logical Operators

Bitwise Operators

Membership Operators

Identity Operators

Operators in Python :Arithmetic Operators

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

Operators in Python :Comparison operators

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

Op.	Meaning	Ex.
>	Greater than - True if left operand is greater than the right	$x > y$
<	Less than - True if left operand is less than the right	$x < y$
==	Equal to - True if both operands are equal	$x == y$
!=	Not equal to - True if operands are not equal	$x != y$
>=	Greater than or equal to - True if left operand is greater than or equal to the right	$x >= y$
<=	Less than or equal to - True if left operand is less than or equal to the right	$x <= y$

Operators in Python :Logical operators

Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Operators in Python :Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

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

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

Operators in Python :Assignment operators

Assignment operators are used in Python to assign values to variables. `a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left. There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Operator	Example	Equivalent to
<code>=</code>	<code>x = 5</code>	<code>x = 5</code>
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>
<code>-=</code>	<code>x -= 5</code>	<code>x = x - 5</code>
<code>*=</code>	<code>x *= 5</code>	<code>x = x * 5</code>
<code>/=</code>	<code>x /= 5</code>	<code>x = x / 5</code>
<code>%=</code>	<code>x %= 5</code>	<code>x = x % 5</code>
<code>//=</code>	<code>x //= 5</code>	<code>x = x // 5</code>

Operators in Python :Identity operators

Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

Identity operators

is and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operators in Python :Identity operators

```
x1 = 5
```

```
y1 = 5
```

```
x2 = 'Hello'
```

```
y2 = 'Hello'
```

```
x3 = [1,2,3]
```

```
y3 = [1,2,3]
```

```
# Output: False
```

```
print(x1 is not y1)
```

```
# Output: True
```

```
print(x2 is y2)
```

```
# Output: False
```

```
print(x3 is y3)
```

Operators in Python :Identity operators

operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Operators in Python :Membership operators

Membership operators

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

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

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

Operators in Python :Membership operators

```
x = 'Hello world'
```

```
y = {1:'a',2:'b'}
```

```
# Output: True
```

```
print('H' in x)
```

```
# Output: True
```

```
print('hello' not in x)
```

```
# Output: True
```

```
print(1 in y)
```

```
# Output: False
```

```
print('a' in y)
```

Control Flow

Decision making is required when we want to execute a code only if a certain condition is satisfied.

The if...elif...else statement is used in Python for decision making.

if test expression:

statement(s)

Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True. If the test expression is False, the statement(s) is not executed.

Control Flow : If statement

In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unintended line marks the end. Python interprets non-zero values as True. None and 0 are interpreted as False.

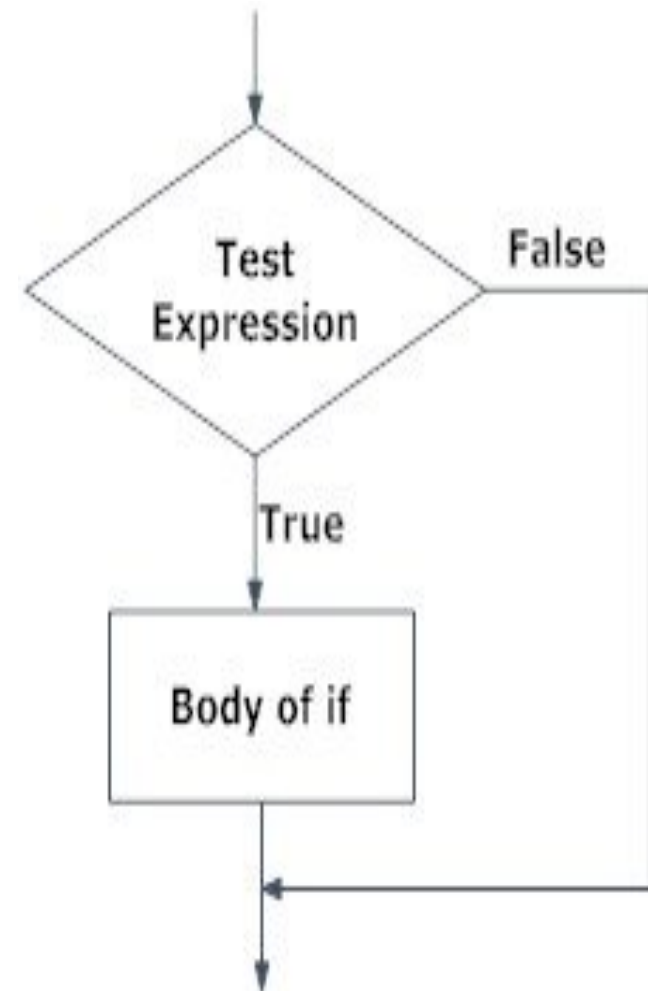


Fig: Operation of if statement

Control Flow

If the number is positive, we print an appropriate message

```
num = 3
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
print("This is always printed.")
```

```
num = -1
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
print("This is also always printed.")
```

Control Flow

If the number is positive, we print an appropriate message

```
num = 3
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
    print("This is always printed.")
```

```
num = -1
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
    print("This is also always printed.")
```

Control Flow

if test expression:

 Body of if

else:

 Body of else

The if....else statement evaluates test expression and will execute the body of if only when the test condition is True.

If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

Control Flow :IF Else Statement

Program checks if the number is positive
or negative

And displays an appropriate message

num = 3

Try these two variations as well.

num = -5

num = 0

```
if num >= 0:
```

```
    print("Positive or Zero")
```

```
else:
```

```
    print("Negative number")
```

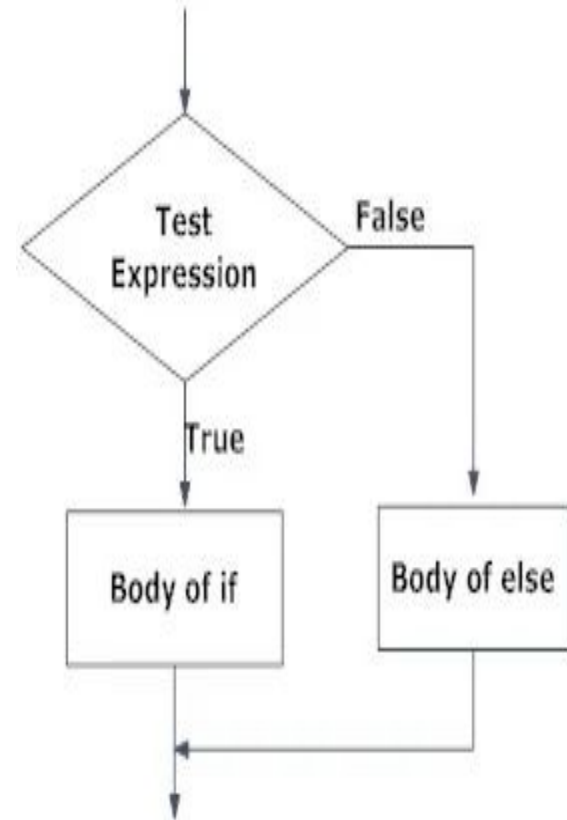


Fig: Operation of if...else statement

Control Flow :IF Else Statement

if test expression:

Body of if

elif test expression:

Body of elif

else:

Body of else

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, the body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

Control Flow :IF ...ElseIf... Else Statement

if test expression:

Body of if

elif test expression:

Body of elif

else:

Body of else

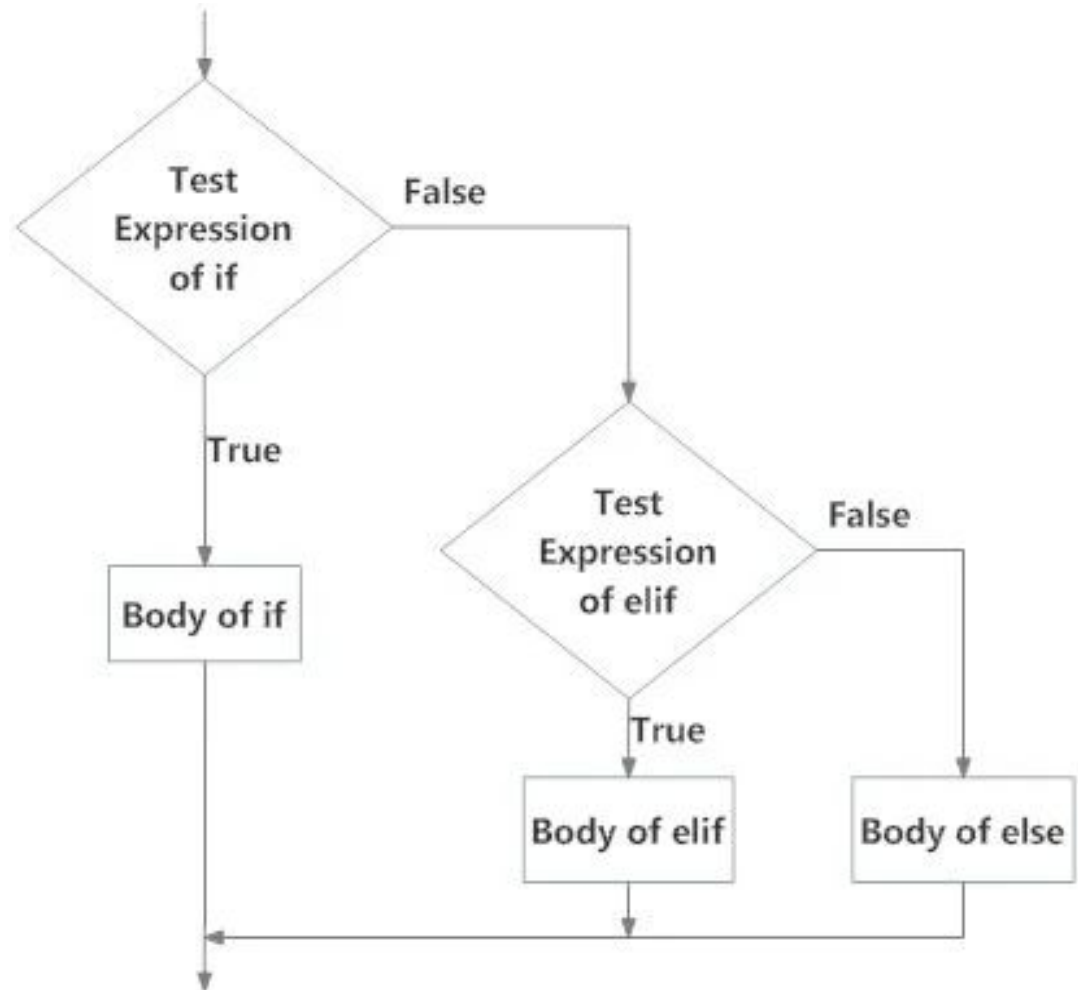


Fig: Operation of if...elif...else statement

Control Flow :IF ...ElseIf... Else Statement

```
num = 3.4
```

```
if num > 0:
```

```
    print("Positive number")
```

```
elif num == 0:
```

```
    print("Zero")
```

```
else:
```

```
    print("Negative number")
```

When variable num is positive,
Positive number is printed.

If num is equal to 0, Zero is
printed.

If num is negative, Negative
number is printed.

Control Flow :IF ...ElseIf... Else Statement

Python Nested if statements

We can have an if...elif...else statement inside another if...elif...else statement. This is called nesting in programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

Control Flow :IF ...ElseIf... Else Statement

```
num = float(input("Enter a number: "))
```

```
if num >= 0:
```

```
    if num == 0:
```

```
        print("Zero")
```

```
    else:
```

```
        print("Positive number")
```

```
else:
```

```
    print("Negative number")
```

Looping :For loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

for val in sequence:

Body of for

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Looping :For loop

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
```

```
sum = 0
```

```
for val in numbers:
```

```
    sum = sum+val
```

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

Looping :For loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. The break keyword can be used to stop a for loop. In such cases, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
    print("No items left.")
```


Looping :For loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. The break keyword can be used to stop a for loop. In such cases, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
    print("No items left.")
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.

Looping :For loop with else

This for...else statement can be used with the break keyword to run the else block only when the break keyword was not executed. Let's take an example:

```
# program to display student's marks from record
student_name = 'Ram'
marks = {'Pawan': 90, 'Rahul': 55, 'Roshan': 77}
for student in marks:
    if student == student_name:
        print(marks[student])
        break
else:
    print('No entry with that name found.')
```

Looping :While loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know the number of times to iterate beforehand.

while test_expression:

Body of while

In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

Looping :While loop

In Python, the body of the while loop is determined through indentation. The body starts with indentation and the first unintended line marks the end. Python interprets any non-zero value as True. None and 0 are interpreted as False

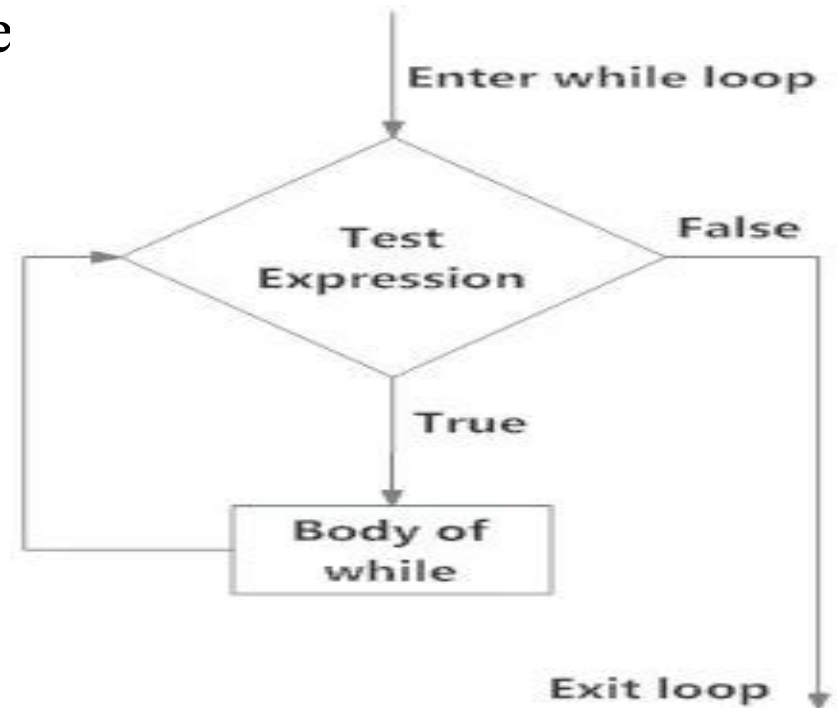


Fig: operation of while loop

Looping :While loop

```
n = 10
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1    # update counter
# print the sum
print("The sum is", sum)
```

In the above program, the test expression will be True as long as our counter variable *i* is less than or equal to *n* (10 in our program).

We need to increase the value of the counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never-ending loop). Finally, the result is displayed.

Looping :While loop with Else

"Example to illustrate
the use of else statement
with the while loop"

```
counter = 0
```

```
while counter < 3:
```

```
    print("Inside loop")
```

```
    counter = counter + 1
```

```
else:
```

```
    print("Inside else")
```

Same as with for loops, while loops can also have an optional else block.

The else part is executed if the condition in the while loop evaluates to False.

The while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

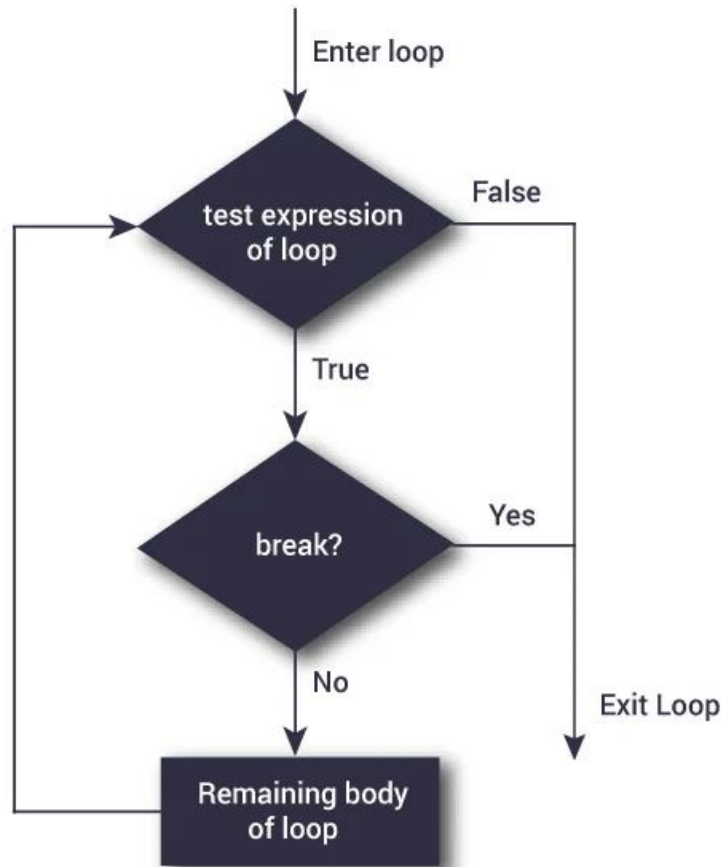
Looping :While loop with Else

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

The break and continue statements are used in these cases.

Looping : While loop with Else




Python break statement


The `break` statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If the `break` statement is inside a nested loop (loop inside another loop), the `break` statement will terminate the innermost loop.

Looping : While loop with Else

```
for var in sequence:
    # codes inside for loop
    if condition:
        break
    # codes inside for loop
# codes outside for loop
```



```
while test expression:
    # codes inside while loop
    if condition:
        break
    # codes inside while loop
# codes outside while loop
```



Python break statement

```
string = ['C','o','m','p','u','t','e','r']
```

```
for val in string:
```

```
    if val == "p":
```

```
        break
```

```
    print(val)
```

```
print("The end")
```

O/P

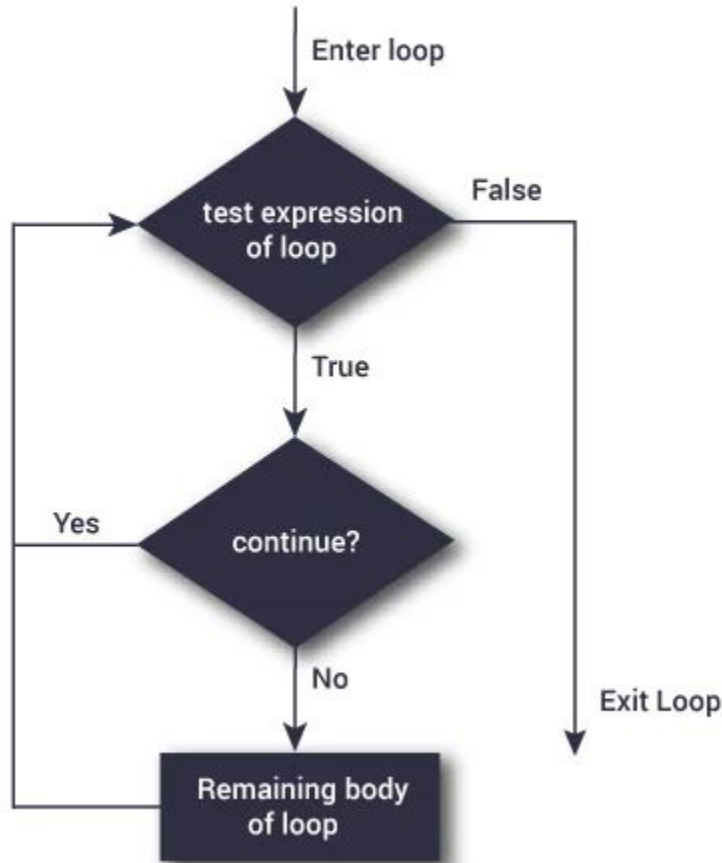
S

t

r

The end

Looping : While loop with Else



The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Looping :While loop with Else

```
for var in sequence:
    # codes inside for loop
    if condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if condition:
        continue
    # codes inside while loop

# codes outside while loop
```

Program to show the use of
continue statement inside loops

```
string = ['C','o','m','p','u','t','e','r']
for val in string:
```

```
    if val == "i":
        continue
```

```
    print(val)
```

```
print("The end")
```

S

t

r

n

g

The end