Unit 3
Data Types

Prepared By:
Tanvi Patel
Asst. Prof. (CSE)

Contents

List

- Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
- A list is a container which holds comma-separated values (items or elements) between square brackets.
- Lists need not be homogeneous always which makes it a most powerful tool in Python.
- A single list may contain Data types like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

List (Cont.)

- List in Python are ordered and have a definite count.
- The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index.
- Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

List (Cont.)

- **Create a List:** To create python list of items, you need to mention the items, separated by commas, in square brackets. Then assign it to a variable.

  Example: colors=['red','green','blue']

- # empty list

  my_list = []

  # list of integers

  my_list = [1, 2, 3]

  # list with mixed data types

  my_list = [1, "Hello", 3.4]

List (Cont.)

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

  ```
  # nested list
  my_list = ["mouse", [8, 4, 6], ['a']]
  ```

- A list may also contain tuples or so.

  languages=[('English','Albanian'),'Gujarati','Hindi','Romanian','Spanish']

  languages[0] ➔ ('English', 'Albanian')

  type(languages[0]) ➔ tuple

List (Cont.)

Access Python List: To access a Python list as a whole, all you need is its name.

days = ['Mon','Tue','Wed',4,5,6,7.0]

days➔ ['Mon', 'Tue', 'Wed', 4, 5, 6, 7.0]

- Or, you can put it in a print statement.

  languages=[['English'],['Gujarati'],['Hindi'],'Romanian','Spanish']

  print(languages) ➔ [[‘English’], [‘Gujarati’], [‘Hindi’], ‘Romanian’, ‘Spanish’]

- To access a single element, use its index in square brackets after the list’s name. Indexing begins at 0.

  languages[0] ➔ [‘English’]

- An index cannot be a float value.

  languages[1.0] ➔ TypeError: list indices must be integers or slices, not float

List (Cont.)

Slicing Python List: When you want only a part of a Python list, you can use the slicing operator [].

indices=['zero','one','two','three','four','five']

indices[2:4] ➔ [‘two’, ‘three’] # This returns items from index 2 to index 4-1 (i.e., 3)

- To return items from the beginning of the list to index 3.

  indices[:3] ➔ ['zero','one','two','three’]

- To return whole list

  indices[:] ➔ [‘zero’, ‘one’, ‘two’, ‘three’, ‘four’, ‘five’]

List (Cont.)

Negative indices- The indices we mention can be negative as well. A negative index means traversal from the end of the list.

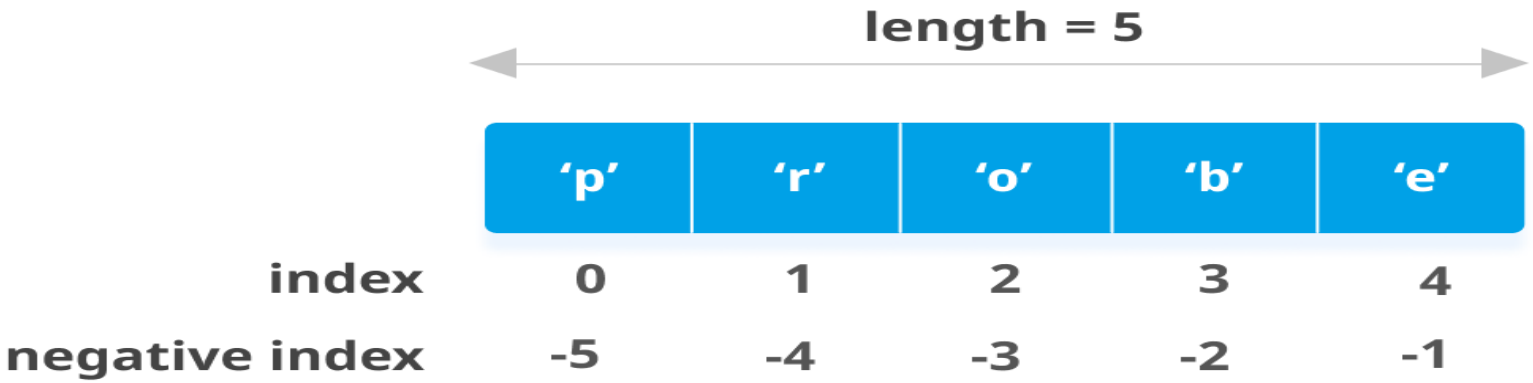
- To return item from the list's beginning to two items from the end.

  indices[:-2] ➔ ['zero', 'one', 'two', 'three']
- To return items from the item at index 1 to two items from the end.

  indices[1:-2] ➔ ['one', 'two', 'three']
- To return items from two from the end to one from the end.

  indices[-2:-1] ➔ ['four']

List (Cont.)



List (Cont.)

- **Reassigning a Python List (Mutable):** Python Lists are mutable. This means that you can reassign its items, or you can reassign it as a whole.

  colors=['red','green','blue']

a. **Reassigning the whole Python list:** We can reassign a Python list by assigning it like a new list.

   colors=['caramel','gold','silver','occur']

   colors ➔ [‘caramel’, ‘gold’, ‘silver’, ‘occur’]

b. **Reassigning a few elements:** You can also reassign a slice of a list in Python.

   colors[2:]=['bronze','silver']

   colors ➔ [‘caramel’, ‘gold’, ‘bronze’, ‘silver’]

List (Cont.)

- If we had instead put two values to a single one in the left, see what would've happened.

  colors=['caramel','gold','silver','occur']

  colors[2:3]=['bronze','silver']

  colors ➔ ['caramel', 'gold', 'bronze', 'silver', 'occur']

  colors[2:3] reassigns the element at index 2, which is the third element.

  2:2 works too.

  colors[2:2]=['occur']

  colors ➔ ['caramel', 'gold', 'occur', 'bronze', 'silver','occur']

List (Cont.)

c. **Reassigning a single element:** We can reassign individual elements too.

colors=['caramel','gold','silver','occur']

colors[3]='bronze'

colors ➔ [‘caramel’, ‘gold’, ‘silver’, ‘bronze’]

- Now if you want to add another item ‘holographic’ to the list, we cannot do it the conventional way.

  colors[4]='holographic‘

  IndexError: list assignment index out of range

- So, you need to reassign the whole list for the same.

  colors=['caramel','gold','silver','bronze','holographic']

  colors ➔ [‘caramel’, ‘gold’, ‘silver’, ‘bronze’, ‘holographic’]

List (Cont.)

- 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) ➔ [1, 3, 5, 7]

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

  print(odd) ➔ [1, 3, 5, 7, 9, 11, 13]

List (Cont.)

- 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.

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

List (Cont.)

Delete a Python List: We can delete a Python list, some of its elements, or a single element.

a. **Deleting the entire Python list:** Use the del keyword for the same.

   del colors

   colors ➔ NameError: name ‘colors’ is not defined

b. **Deleting a few elements:** Use the slicing operator in python to delete a slice.

   colors=['caramel','gold','silver','bronze','holographic']

   del colors[2:4]

   colors ➔ [‘caramel’, ‘gold’, ‘holographic’]

c. **Deleting a single element:** To delete a single element from a Python list, use its index.

   del colors[0]

   colors ➔ [‘gold’, ‘holographic’]

List (Cont.)

- 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 or last in, first out).
- We can also use the clear() method to empty a list.
- Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
my_list = ['p','r','o','b','l','e','m']
my_list[2:3] = []
my_list  ➔  ['p', 'r', 'b', 'l', 'e', 'm']
my_list[2:5] = []
my_list  ➔  ['p', 'r', 'm']
```

List (Cont.)

Copy a List:

- We cannot copy a list simply by typing list2 = list1, because: list2 will only be a *reference* to list1, and changes made in list1 will automatically also be made in list2.
- There are ways to make a copy, one way is to use the built-in List method copy().

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)  ➔ ['apple', 'banana', 'cherry']
```

- Another way to make a copy is to use the built-in method list().

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)  ➔ ['apple', 'banana', 'cherry']
```

List (Cont.)

Multidimensional Lists in Python: We can also put a list in a list. Let's look at a multidimensional list.

grocery_list=[['caramel','P&B','Jelly'],['onions','potatoes'],['flour','oil']]

grocery_list ➔ [[‘caramel’, ‘P&B’, ‘Jelly’], [‘onions’, ‘potatoes’], [‘flour’, ‘oil’]]

- Or, you can choose to go deeper.

  a=[[[1,2],[3,4],5],[6,7]]

  a ➔ [[[1, 2], [3, 4], 5], [6, 7]]

- To access the element 4 here, we type the following code into the shell.

  a[0][1][1] ➔ 4

List (Cont.)

Concatenation of Python List: The concatenation operator works for lists as well. It lets us join two lists, with their orders preserved.

a,b=[3,1,2],[5,4,6]

a+b ➔ [3, 1, 2, 5, 4, 6]

Python List Operations

a. Multiplication: This is an arithmetic operation. Multiplying a Python list by an integer makes copies of its items that a number of times while preserving the order.

a*=3 ➔ a = a*3

a ➔ [3, 1, 2, 3, 1, 2, 3, 1, 2]

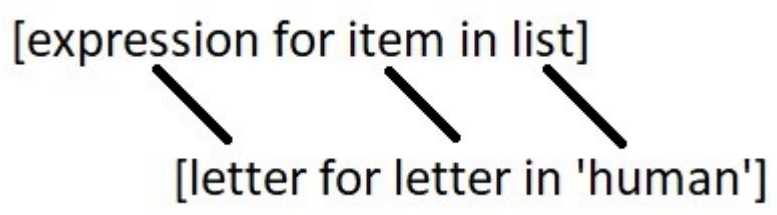
- However, you can't multiply it by a float.

a*3.0 ➔ TypeError: can't multiply sequence by non-int of type 'float'

List (Cont.)

List comprehension: List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

- **Syntax of List Comprehension:** [expression for item in list]



List (Cont.)

- Example (without comprehension):

```
h_letters = []
for letter in 'human':
    h_letters.append(letter)
print(h_letters)
```

- Example (with comprehension):

```
h_letters = [ letter for letter in 'human' ]
print( h_letters)
```

List (Cont.)

Conditionals in List Comprehension

- List comprehensions can utilize conditional statement to modify existing list (or other tuples). We will create list that uses mathematical operators, integers, and range().
- **Example**

```
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list)
```

List (Cont.)

Key Points to Remember for List Comprehension

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating list.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in for loop, but every for loop can’t be rewritten in the form of list comprehension.

List (Cont.)

Other List Operations in Python

- **List Membership Test**
- We can test if an item exists in a list or not, using the keyword in.

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

  print('p' in my_list) ➔ True

  print('a' in my_list) ➔ False

  print('c' not in my_list) ➔ True
- **Iterating Through a List**
- Using a for loop we can iterate through each item in a list.

```
for fruit in ['apple','banana','mango']:
    print("I like",fruit)
```

List (Cont.)

Built-in List Functions: There are some built-in functions in Python that you can use on python lists.

- even = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

Sr. No.	Name	Description	Example
1	len()	It calculates the length of the list.	len(even)
2	max()	It returns the item from the list with the highest value.	max(even)
3	min()	It returns the item from the Python list with the lowest value.	min(even)
4	sum()	It returns the sum of all the elements in the list.	sum(even)
5	sorted()	It returns a sorted version of the list, but does not change the original one.	sorted(a)

List (Cont.)

Sr. No.	Name	Description	Example
6	list()	It converts a different data type into a list.	list("abc") ➔ [‘a’, ‘b’, ‘c’]
7	any()	It returns True if even one item in the Python list has a True value.	any(["","",'1']) ➔ TRUE
8	all()	It returns True if all items in the list have a True value.	all(["","",'1']) ➔ FALSE all([]) ➔ TRUE

List (Cont.)

Built-in Methods: While a function is what you can apply on a construct and get a result, a method is what you can do to it and change it. To call a method on a construct, you use the dot-operator(.). Python supports some built-in methods to alter a Python list.

Sr. No.	Name	Description	Example
1	append()	It adds an item to the end of the list.	a = [2, 1, 3] a.append(4)
2	insert()	It inserts an item at a specified position.	a.insert(3,5)
3	remove()	It removes the first instance of an item from the Python list.	a = [2,1,3,5,2,4] a.remove(2)

List (Cont.)

Sr. No.	Name	Description	Example
4	pop()	It removes the element at the specified index, and prints it to the screen.	a.pop(3)
5	clear()	It empties the Python list.	a.clear()
6	index()	It returns the first matching index of the item specified.	a=[1,3,5,3,4] a.index(3)
7	count()	It returns the count of the item specified.	a.count(3)
8	sort()	It sorts the list in an ascending order.	a.sort()
9	reverse()	It reverses the order of elements in the Python lists.	a.reverse()

Tuple

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and **unchangeable i.e., immutable**.
- Tuples are written with round brackets.
- Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
- A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).
- Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also.

Tuple (Cont.)

- Example:

  tup1 = () #empty tuple

  tup2 = (1, 2, 3, 4, 5) # tuple having integers

  tup3 = ('physics', 'chemistry', 1997, 2000) #tuple with mixed datatype

  tup4 = ("mouse", [8, 4, 6], (1, 2, 3)) #nested tuple

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

  tup5 = "a", "b", "c", "d“

  my_tuple = 3, 4.6, "dog“

  # tuple unpacking is also possible

  a, b, c = my_tuple

Tuple (Cont.)

- Creating a tuple with one element is a bit tricky.
- Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("hello")
print(type(my_tuple)) ➔  str
# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple)) ➔ tuple
# Parentheses is optional
my_tuple = "hello",
print(type(my_tuple)) ➔ tuple
```

Tuple (Cont.)

Access Tuple Elements

- There are various ways in which we can access the elements of a tuple.

1. Indexing

- We can use the index operator [] to access an item in a tuple, where the index starts from 0.
- So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an IndexError.
- The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Tuple (Cont.)

- **Accessing tuple elements using indexing**

  my_tuple = ('p','e','r','m','i','t')

  print(my_tuple[0]) ➔ 'p'

  print(my_tuple[5]) ➔ 't'

  print(my_tuple[6]) ➔ IndexError: list index out of range

  my_tuple[2.0] ➔ TypeError: list indices must be integers, not float

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

  print(n_tuple[0][3]) ➔ 's'

  print(n_tuple[1][1]) ➔ 4

Tuple (Cont.)

2. 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.

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[-1]) ➔  't'
print(my_tuple[-6]) ➔ 'p'
```

Tuple (Cont.)

3. Slicing

Accessing tuple elements using slicing

my_tuple = ('p','r','o','g','r','a','m')

print(my_tuple[1:4]) ➔ (‘r’,’o’,’g’)

print(my_tuple[:-7]) ➔ ()

print(my_tuple[5:]) ➔ (‘a’,’m’)

print(my_tuple[:]) ➔ (‘p’,’r’,’o’,’g’,’r’,’a’,’m’)

Tuple (Cont.)

Changing a Tuple

- Unlike lists, tuples are immutable.
- This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed.
- We can also assign a tuple to different values (reassignment).

  my_tuple = (4, 2, 3, [6, 5])

  my_tuple[1] = 9 ➔ TypeError: 'tuple' object does not support item assignment
- However, item of mutable element can be changed

  my_tuple[3][0] = 9

  print(my_tuple) ➔ (4, 2, 3, [9, 5])

Tuple (Cont.)

- We can use + operator to combine two tuples. This is called **concatenation**.
- We can also **repeat** the elements in a tuple for a given number of times using the * operator.
- Both + and * operations result in a new tuple.

  print((1, 2, 3) + (4, 5, 6)) ➔ (1, 2, 3, 4, 5, 6)

  print(("Repeat",) * 3) ➔ ('Repeat', 'Repeat', 'Repeat')

Tuple (Cont.)

Deleting a Tuple

- We cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.
- Deleting a tuple entirely, however, is possible using the keyword del.

  my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm')
- We can't delete items

  del my_tuple[3] ➔ TypeError: 'tuple' object doesn't support item deletion
- We can delete an entire tuple

  del my_tuple

  print(my_tuple) ➔ NameError: name 'my_tuple' is not defined

Tuple (Cont.)

Built-in List Functions: A lot of functions that work on lists work on tuples too. A function applies on a construct and returns a result. It does not modify the construct.

- my_tuple = (1, 2, 3, [6, 5])

Sr. No.	Name	Description	Example
1	len()	It calculates the length of the tuple.	len(my_tuple)
2	max()	It returns the item from the tuple with the highest value.	a=(3,1,2,5,4,6) max(a)
3	min()	It returns the item from the Python tuple with the lowest value.	min(a)
4	sum()	It returns the sum of all the elements in the tuple.	sum(a)
5	sorted()	It returns a sorted version of the tuple, but does not change the original one.	sorted(a)

Tuple (Cont.)

Sr. No.	Name	Description	Example
6	tuple()	This function converts another construct into a Python tuple	list1=[1,2,3] tuple(list1) ➔ (1, 2, 3) string1="string" tuple(string1) ➔ (‘s’, ‘t’, ‘r’, ‘i’, ‘n’, ‘g’)
7	any()	It returns True if even one item in the Python tuple has a True value.	any((",'0',"))➔ TRUE any((",0,"))➔ FALSE
8	all()	It returns True if all items in the tuple have a True value.	all(('1',1,True,‘ ')) ➔ FALSE

Tuple (Cont.)

Tuple Methods

- Methods that add items or remove items are not available with tuple. Only the following two methods are available.
- Some examples of Python tuple methods:
- my_tuple = ('a', 'p', 'p', 'l', 'e',)
- print(my_tuple.count('p')) ➔ 2
- print(my_tuple.index('l')) ➔ 3

Tuple (Cont.)

Other Tuple Operations

1. Tuple Membership Test

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

  my_tuple = ('a', 'p', 'p', 'l', 'e',)

  print('a' in my_tuple) ➔ True

  print('b' in my_tuple) ➔ False

  print('g' not in my_tuple) ➔ True

Tuple (Cont.)

2. Iterating Through a Tuple

- We can use a for loop to iterate through each item in a tuple.

```
for name in ('John', 'Kate'):
        print("Hello", name)
```

Output:

Hello John

Hello Kate

Tuple (Cont.)

Advantages of Tuple over List: Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Sets

- Sets are used to store multiple items in a single variable.
- 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.

Sets (Cont.)

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.
- 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, sets or dictionaries as its elements.
- Set of integers

  my_set = {1, 2, 3}
- Set of mixed datatypes

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

Sets (Cont.)

- Set cannot have duplicate items.

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

  print(my_set) ➔ {1, 2, 3, 4}
- We can make a set from list

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

  print(my_set) ➔ {1, 2, 3}
- Set cannot have mutable items. In below example [3, 4] is a mutable list so, this will cause an error.

  my_set = {1, 2, [3, 4]} ➔ TypeError: unhashable type: 'list'

Sets (Cont.)

- 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.
- Distinguish set and dictionary while creating empty set

  a = {}

  print(type(a)) ➔ <class 'dict'>

  a = set()

  print(type(a)) ➔ <class 'set'>

Sets (Cont.)

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.

Sets (Cont.)

my_set = {1, 3}

my_set[0] ➔ TypeError: 'set' object does not support indexing

- Add an element

  my_set.add(2)

  print(my_set) ➔ {1, 2, 3}

- Add multiple elements

  my_set.update([2, 3, 4])

  print(my_set) ➔ {1, 2, 3, 4}

- Add list and set

  my_set.update([4, 5], {1, 6, 8})

  print(my_set) ➔ {1, 2, 3, 4, 5, 6, 8}

Sets (Cont.)

Removing elements from a set

- 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).

  my_set = {1, 3, 4, 5, 6}
- Discard an element

  my_set.discard(4)

  print(my_set) ➔ {1, 3, 5, 6}

Sets (Cont.)

- Remove an element

  my_set.remove(6)

  print(my_set) ➔ {1, 3, 5}
- Discard an element not present in a set

  my_set.discard(2)

  print(my_set) ➔ {1, 3, 5}
- Remove an element not present in a set

  my_set.remove(2) ➔ KeyError: 2

Sets (Cont.)

- Similarly, we can remove and return an item using the pop() method.
- Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.
- We can also remove all the items from a set using the clear() method.

  my_set = set("HelloWorld")

  print(my_set) ➔ {'H', 'l', 'r', 'W', 'o', 'd', 'e'}
- Pop an element

  print(my_set.pop()) ➔ Gives an random element ➔ H

  my_set.pop()

  print(my_set) ➔ {'r', 'W', 'o', 'd', 'e'}

  my_set.clear()

  print(my_set) ➔ set()

Sets (Cont.)

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.
- Let us consider the following two sets for the following operations.

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

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

Sets (Cont.)

Set Union

- 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.

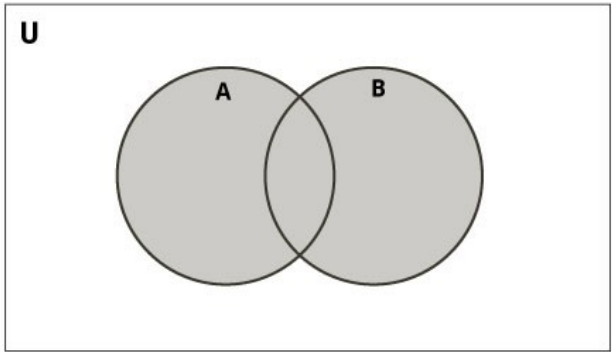
  A = {1, 2, 3, 4, 5}

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

  print(A | B) ➔ {1, 2, 3, 4, 5, 6, 7, 8}

  A.union(B) ➔ {1, 2, 3, 4, 5, 6, 7, 8}

  B.union(A) ➔ {1, 2, 3, 4, 5, 6, 7, 8}



Sets (Cont.)

Set Intersection

- 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.

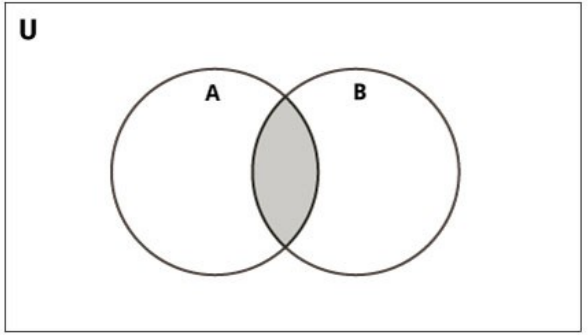
  A = {1, 2, 3, 4, 5}

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

  print(A & B) ➔ {4, 5}

  A.intersection(B) ➔ {4, 5}

  B.intersection(A) ➔ {4, 5}



Sets (Cont.)

Set Difference

- 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.

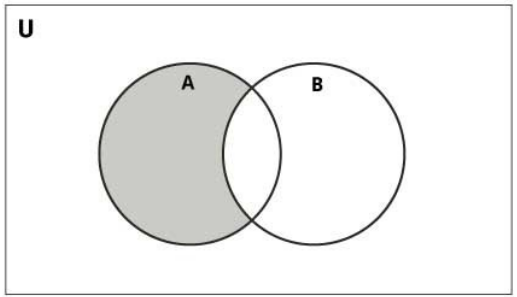
A = {1, 2, 3, 4, 5}

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

print(A - B) ➔ {1, 2, 3}

A.difference(B) ➔ {1, 2, 3}

B – A ➔ {8, 6, 7}
B.difference(A) ➔ {8, 6, 7}



Sets (Cont.)

Set Symmetric Difference

- 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().

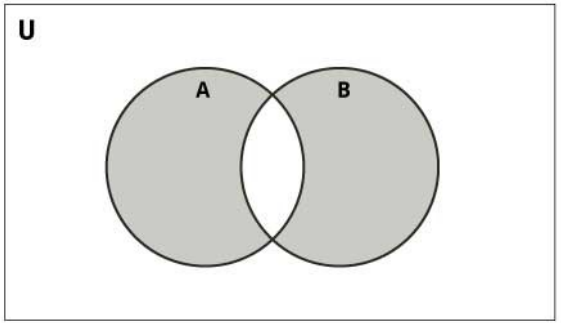
  A = {1, 2, 3, 4, 5}

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

  print(A ^ B) ➔ {1, 2, 3, 6, 7, 8}

  A.symmetric_difference(B) ➔ {1, 2, 3, 6, 7, 8}

  B.symmetric_difference(A) ➔ {1, 2, 3, 6, 7, 8}



Sets (Cont.)

Other Python Set Methods

- There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another

Sets (Cont.)

Method	Description
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises KeyError if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a KeyError
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Sets (Cont.)

Other Set Operations

- **Set Membership Test**
- We can test if an item exists in a set or not, using the in keyword.

  my_set = set("apple")

  print('a' in my_set) ➔ True

  print('p' not in my_set) ➔False
- **Iterating Through a Set**
- We can iterate through each item in a set using a for loop.

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

Output:

a p l e

Sets (Cont.)

Built-in Functions with Set

- Built-in functions like all(), any(),len(), max(), min() etc. are commonly used with sets to perform different tasks.

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

Sets (Cont.)

Python Frozenset

- Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.
- Frozensets can be created using the frozenset() function.
- This data type supports methods
  like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable, it does not have methods that add or remove elements.

Sets (Cont.)

- A = frozenset([1, 2, 3, 4])
- B = frozenset([3, 4, 5, 6])
- A.isdisjoint(B) ➔ False
- A.difference(B) ➔ frozenset({1, 2})
- A | B ➔ frozenset({1, 2, 3, 4, 5, 6})
- A.add(3) ➔ AttributeError: 'frozenset' object has no attribute 'add'

Sets (Cont.)

- Operations that can't be performed on frozenset, but can be performed on a set:
- A few operations can't be performed on frozenset due to the nature of it being immutable. Below are such operations:
- update
- intersection_update
- difference_update
- symmetric_difference_update
- add
- remove
- discard
- pop
- clear

Exercise

- fs = frozenset([1, 2, 3, 4, 5])
- size = len(fs)
- print('frozenset size =', size)
- contains_item = 5 in fs
- print('fs contains 5 =', contains_item)
- not_contains_item = 6 not in fs
- print('fs not contains 6 =', not_contains_item)
- is_disjoint = fs.isdisjoint(frozenset([1, 2]))
- print(is_disjoint)
- is_disjoint = fs.isdisjoint(frozenset([10, 20]))
- print(is_disjoint)

Exercise

- is_subset = fs.issubset(set([1, 2]))
- print(is_subset)
- is_subset = fs.issubset(set([1, 2, 3, 4, 5, 6, 7]))
- print(is_subset)
- is_superset = fs.issuperset(frozenset([1, 2]))
- print(is_superset)
- is_superset = fs.issuperset(frozenset([1, 2, 10]))
- print(is_superset)
- fs1 = fs.union(frozenset([1, 2, 10]), set([99, 98]))
- print(fs1)
- fs1 = fs.intersection(set((1, 2, 10, 20)))
- print(fs1)

Exercise

- fs1 = fs.difference(frozenset([1, 2, 3]))
- print(fs1)

- fs1 = fs.symmetric_difference(frozenset([1, 2, 10, 20]))
- print(fs1)

- fs1 = fs.copy()
- print(fs1)

Booleans

- A Boolean is another data type that Python offer.
- You can evaluate any expression in Python, and get one of two answers, True or False.
- **Value of a Boolean:** A Boolean value may either be True or be False. Some methods like isalpha() or issubset() return a Boolean value.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer.

  ```
  print(10 > 9)
  print(10 == 9)
  print(10 < 9)
  ```

- **Declaring a Boolean:** You can declare a Boolean just like you would declare an integer.

  ```
  days=True
  ```

Booleans

- **The bool() function:** the bool() function converts another value into the Boolean type and allows you to evaluate any value, and give you True or False in return.

  bool(‘Tanvi') ➔ True

  bool([]) ➔ False
- **Boolean Values of Various Constructs:** Different values have different equivalent Boolean values. In this example, we use the bool() Python set function to find the values.
- For example, 0 has a Boolean value of False.

  bool(0) ➔ False
- 1 has a Boolean value of True, and so does 0.00000000001.

  bool(0.000000000001) ➔ True

Booleans

- **Most of the values are True**
- Almost any value is evaluated to True if it has some sort of content.
- Any string is True, except empty strings.
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.
- The following will return True:

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

Booleans

- **Some values are False**
- In fact, there are not many values that evaluates to False, except empty values, such as (), [], {}, "", the number 0, and the value None. And of course the value False evaluates to False.
- The following will return False:

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

Booleans (Cont.)

Operations on Booleans

- **1. Arithmetic**
  You can apply some arithmetic operations to a set. It takes 0 for False, and 1 for True, and then applies the operator to them.
- **Addition**

  True+False

  True+True

  False+True

  False+False
- **Subtraction and Multiplication:** The same strategy is adopted for subtraction and multiplication.

  False-True

Booleans (Cont.)

- **Division:** Division results in a float.

  False/True

  True/False

- **Modulus, Exponentiation, and Floor Division**

  False%True

  True**False

  False**False

  0//1

  (True+True)*False+True

Booleans (Cont.)

2. **Relational:** The relational operators we've learnt so far are >, <, >=, <=, !=, and ==. All of these apply to Boolean values. We will show you a few examples, you should try the rest of them.

   False>True

   False<=True

Booleans (Cont.)

3. **Bitwise:** Normally, the bitwise operators operate bit-by bit. For example, the following code ORs the bits of 2(010) and 5(101), and produces the result 7(111).

- 2|5 ➔ 7
- **Bitwise &:** It returns True only if both values are True.

  True&False

  True&True
- **Bitwise | :** It returns False only if both values are False.

  False|True
- **Bitwise XOR (^):** This returns True only if one value is True and one is False.

  False^True

  False^False

  True^True

Booleans (Cont.)

- **Binary 1’s Complement:** This calculates 1’s complement for True(1) and False(0).

  ~True

  ~False

- **Left-shift(<<) and Right-shift(>>) Operators:** These operators shift the value by specified number of bits left and right, respectively.

  False>>2

  True<<2

- True is 1. When shifted two places two the left, it results in 100, which is binary for 4. Hence, it returns 4.

Booleans (Cont.)

4. **Identity:** The identity operators ‘is’ and ‘is not’ apply to Booleans.

   False is False

   False is 0

5. **Logical**: Finally, even the logical operators apply on Booleans.

   False and True

Dictionary

- Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.
- Dictionary is a mutable data type in Python.
- A python dictionary is a collection of key and value pairs separated by a colon (:), enclosed in curly braces {}.
- Here we have a dictionary. Left side of the colon(:) is the key and right side of the : is the value.

  mydict = {'Name': ‘Tanvi', 'Age': 27, 'City': ‘Vadodara'}
- Dictionaries are optimized to retrieve values when the key is known.
- 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.

Dictionary (Cont.)

- Empty dictionary

  my_dict = {}
- Dictionary with integer keys

  my_dict = {1: 'apple', 2: 'ball'}
- Dictionary with mixed keys

  my_dict = {'name': Tanvi', 1: [2, 4, 3]}
- We can also create a dictionary using the built-in dict() function

  dict({1:'apple', 2:'ball'})
- From sequence having each item as a pair

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

Dictionary (Cont.)

Accessing Elements from Dictionary

- While indexing is used with other data types to access values, a dictionary uses keys.
- Keys can be used either inside square brackets [] or with the get() method.
- If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

  my_dict = {'name': ‘Tanvi', 'age': 27}

  print(my_dict['name']) ➔ Tanvi

  print(my_dict.get('age')) ➔ 27

Dictionary (Cont.)

- Trying to access keys which doesn't exist throws error

  print(my_dict.get('address')) ➔ None

  print(my_dict['address']) ➔ KeyError: 'address‘

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.

Dictionary (Cont.)

- my_dict = {'name': ‘Tanvi', 'age': 27}
- To update the value

  my_dict['age'] = 28

  print(my_dict) ➔ {'name': ‘Tanvi‘, 'age': 28}
- To add the item

  my_dict[‘city'] = ‘Vadodara'

  print(my_dict) ➔ { 'name': ‘Tanvi' , 'age': 28, ‘city': ‘Vadodara' }

Dictionary (Cont.)

Removing elements from Dictionary

- We can remove a particular item in a dictionary by using the pop() method.
- This method removes an item with the provided key and returns the value.
- The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary.
- All the items can be removed at once, using the clear() method.
- We can also use the del keyword to remove individual items or the entire dictionary itself.

Dictionary (Cont.)

- squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
- **Remove a particular item, returns its value**

  print(squares.pop(4)) ➔ 16

  print(squares) ➔ {1: 1, 2: 4, 3: 9, 5: 25}
- **Remove an arbitrary item, return (key,value)**

  print(squares.popitem()) ➔ (5, 25)

  print(squares) ➔ {1: 1, 2: 4, 3: 9}
- **Remove all items**

  squares.clear()

  print(squares) ➔ {}
- **Delete the dictionary itself**

  del squares

  print(squares) ➔ NameError: name 'squares' is not defined

Dictionary (Cont.)

Python Dictionary Methods

- Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Returns the value of the key. If the key does not exist, returns d (defaults to None).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.

Dictionary (Cont.)

Method	Description
pop(key[,d])	Removes the item with the key and returns its value or d if key is not found. If d is not provided and the key is not found, it raises KeyError.
popitem()	Removes and returns an arbitrary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of d and returns d (defaults to None).
update([other])	Updates the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Returns a new object of the dictionary's values

Dictionary (Cont.)

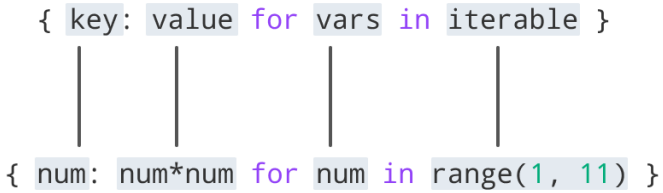
- **Dictionary Built-in Functions**

Function	Description
all()	Return True if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries. (Not available in Python 3)
sorted()	Return a new sorted list of keys in the dictionary.

Dictionary (Cont.)

Python Dictionary Comprehension

- Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.
- Dictionary comprehension consists of an expression pair (**key: value**) followed by a for statement inside curly braces {}.
- Syntax for dictionary comprehension:
- dictionary = {key: value for vars in iterable}



Dictionary (Cont.)

- Here is an example to make a dictionary with each item being a pair of a number and its square.

  squares = {x: x*x for x in range(6)}

  print(squares) ➔ {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
- This code is equivalent to

  squares = {}

  for x in range(6):

      squares[x] = x*x

  print(squares) ➔ {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
- A dictionary comprehension can optionally contain more for or if statements.
- An optional if statement can filter out items to form the new dictionary.
- Here are some examples to make a dictionary with only odd items.
- odd_squares = {x: x*x for x in range(11) if x % 2 == 1}
- print(odd_squares) ➔ {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

Dictionary (Cont.)

Other Dictionary Operations

- **Dictionary Membership Test**
- We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.

  squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

  print(1 in squares) ➔

  print(2 not in squares) ➔
- Membership tests for key only not value

  print(49 in squares) ➔

Dictionary (Cont.)

- **Iterating Through a Dictionary**
- We can iterate through each key in a dictionary using a for loop.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

- O/p:

```
1
9
25
49
81
```

Built-in Data Types

- In programming, data type is an important concept.
- 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
Mapping Type:	Dict
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray, memoryview

Built-in Data Types (Cont.)

- Setting the Data Type: In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	Str
x = 20	Int
x = 20.5	Float
x = 1j	Complex
x = ["apple", "banana", "cherry"]	List
x = ("apple", "banana", "cherry")	Tuple
x = range(6)	Range

Built-in Data Types (Cont.)

Example	Data Type
x = {"name" : "John", "age" : 36}	Dict
x = {"apple", "banana", "cherry"}	Set
x = frozenset({"apple", "banana", "cherry"})	Frozenset
x = True	Bool
x = b"Hello"	Bytes
x = bytearray(5)	Bytearray
x = memoryview(bytes(5))	Memoryview

Python Constructs and Logic

Python Constructs and Logic

- **Code Blocks**
- Code blocks in Python use indentation. Many other programming languages use curly braces for indicate code blocks; not Python.
- Here, we use 4 spaces, that is the standard in Python. We could use a different amount of spaces or tabs, but 4 spaces are recommended by the Python creators - so we will use it.

Python Constructs and Logic (Cont.)

- **The three basic programming constructs**
- **Programs** are designed using common building blocks. These building blocks, known as programming constructs (or programming concepts), form the basis for all programs.
- There are three basic building blocks to consider:
- **sequence** is the order in which **instructions** occur and are processed
- **selection** determines which path a program takes when it is running
- **iteration** is the repeated **execution** of a section of code when a program is running

Python If Elif Else

- All programs use one or more of these constructs. The longer and more complex the program, the more these constructs will be used repeatedly.
- **Decision Making in Python**
- 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.
- **Python if Statement Syntax**
- 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.

Python If Elif Else (Cont.)

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

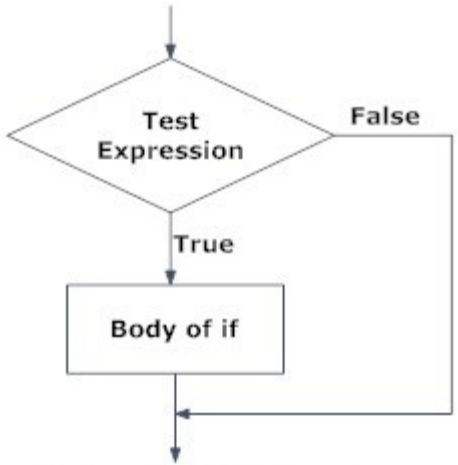


Fig: Operation of if statement

Python If Elif Else (Cont.)

- **Example: Python if Statement**

```
# 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.")
```

Python If Elif Else (Cont.)

- **Python if...else Statement**
- **Syntax of if...else**

```
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.

Python If Elif Else (Cont.)

Python if..else Flowchart

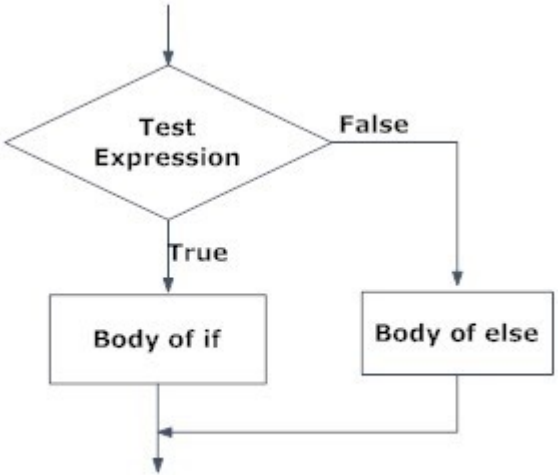


Fig: Operation of if...else statement

Python If Elif Else (Cont.)

- **Example of if...else**
- # Program checks if the number is positive or negative
- num = 3
- if num >= 0:
- print("Positive or Zero")
- else:
- print("Negative number")

Python If Elif Else (Cont.)

- **Python if...elif...else Statement**
- **Syntax of if...elif...else**
- 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.

Python If Elif Else (Cont.)

- **Flowchart of if...elif...else**

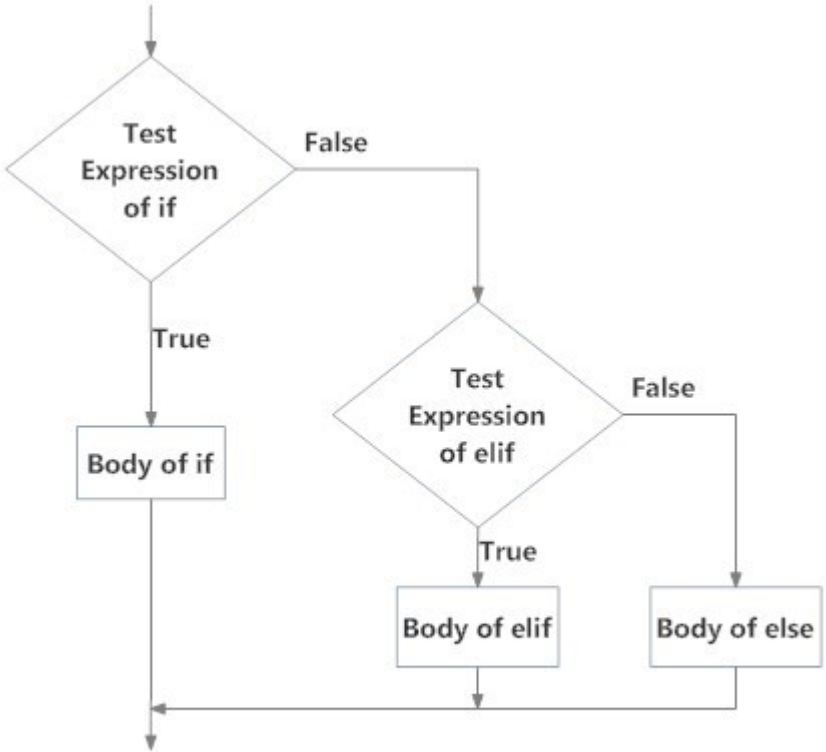


Fig: Operation of if...elif...else statement

Python If Elif Else (Cont.)

- **Example of if...elif...else**
- num = 3.4
- if num > 0:
- print("Positive number")
- elif num == 0:
- print("Zero")
- else:
- print("Negative number")

Python If Elif Else (Cont.)

Python Nested if statements

- We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.
- Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting.

Python Nested if Example

```
num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Python 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.
- **Syntax of while Loop in Python**

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

Python while Loop (Cont.)

- **Flowchart of while Loop**

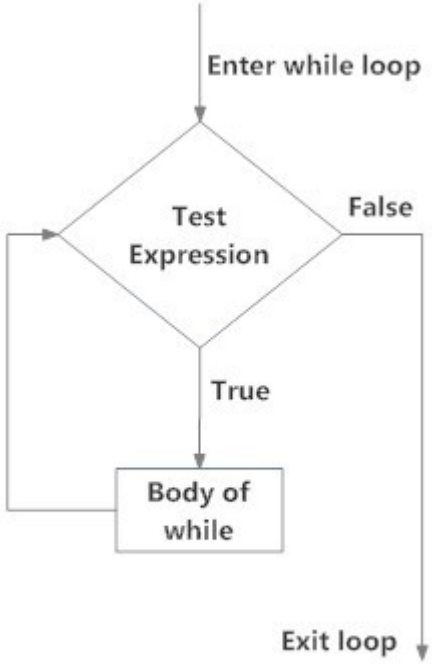


Fig: operation of while loop

Python while Loop (Cont.)

- **Example: Python while Loop:** Program to add natural numbers up to sum = 1+2+3+...+n
- n = 10
- sum = 0
- i = 1
- while i <= n:
    sum = sum + i
    i = i+1
  print("The sum is", sum)

Python while Loop (Cont.)

- **While loop with else**
- 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.

```
counter = 0
while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Loop Control Statements

- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.
- 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.
- **Continue:** It returns the control to the beginning of the loop.
- **Break:** It brings control out of the loop
- **Pass:** We use pass statement to write empty loops. Pass is also used for empty control statements, functions and classes.

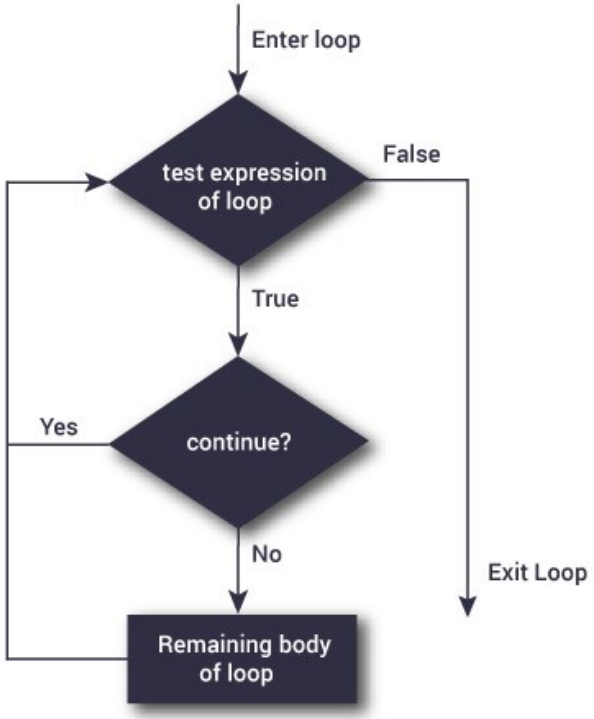
Loop Control Statements (Cont.)

Continue: With the continue statement we can stop the current iteration, and continue with the next.

- 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.

Example:

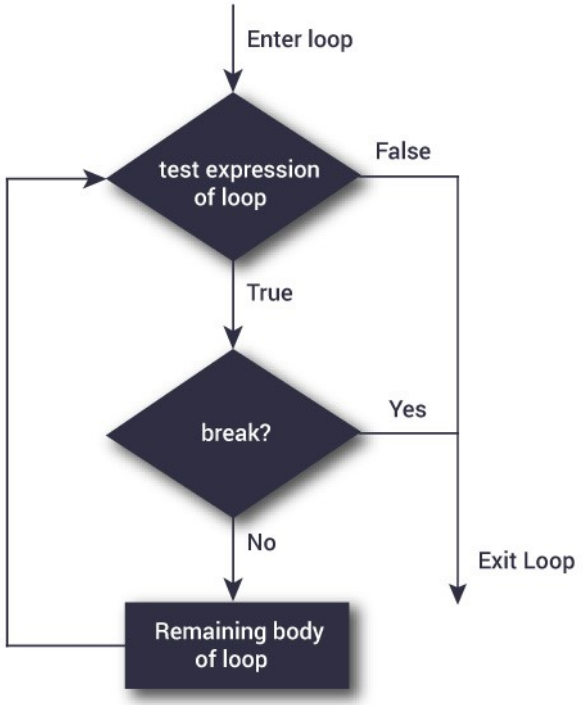
- ```
  i = 0
  while i < 6:
    i += 1
    if i == 3:
      continue
    print(i)
  ```


  ```

Loop Control Statements (Cont.)

Break: With the break statement we can stop the loop even if the while condition is true.

- Control of the program flows to the statement immediately after the body of the loop.
- If the break statement is inside a nested loop, the break statement will terminate the innermost loop.
- Example
- ```
  i = 1
  while i < 6:
    print(i)
    if i == 3:
      break
    i += 1
  ```


  ```

Loop Control Statements (Cont.)

Pass: In Python programming, the pass statement is a null statement.

- The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.
- However, nothing happens when the pass is executed. It results in no operation (NOP).
- Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would give an error. So, we use the pass statement to construct a body that does nothing.

Loop Control Statements (Cont.)

Example:

```
i=0
a = "Good Afternoon Students"
while(i<len(a)):
    i+=1
    pass
 print(i)
```

Python 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.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.
- **Syntax of for Loop**
- 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.

Python for loop (Cont.)

- Flowchart of for loop

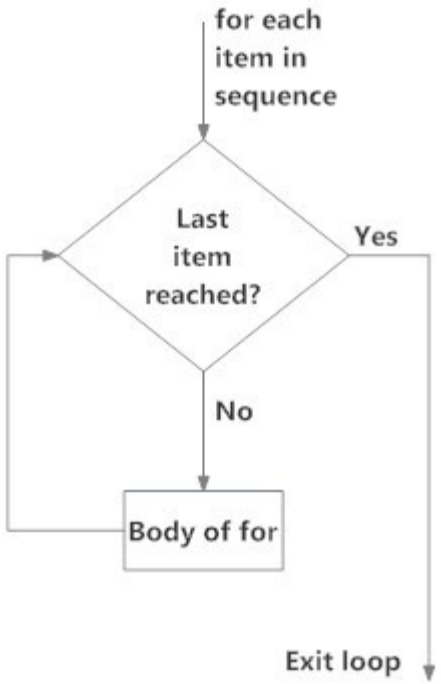


Fig: operation of for loop

Python for loop (Cont.)

- Example: Program to find the sum of all numbers stored in a list

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
    sum = sum+val
print("The sum is", sum)
```

0,1,1,2,3,5,8

Python for loop (Cont.)

Nested Loops

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":
- Example:
- ```
  adj = ["red", "big", "tasty"]
  fruits = ["apple", "banana", "cherry"]
  for x in adj:
    for y in fruits:
      print(x, y)
  ```
  ```

Python for loop (Cont.)

The **range()** Function

- We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as range(start, stop,step_size). step_size defaults to 1 if not provided.
- This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.
- To force this function to output all the items, we can use the function list().

  print(range(10)) ➔ range(0, 10)

  print(list(range(10))) ➔ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

  print(list(range(2, 8))) ➔ [2, 3, 4, 5, 6, 7]

  print(list(range(2, 20, 3))) ➔ [2, 5, 8, 11, 14, 17]

Python for loop (Cont.)

- We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.
- **Example:** Program to iterate through a list using indexing

```
a = ['pop', 'rock', 'jazz']
# iterate over the list using index
for i in range(len(a)):
    print("I like", a[i])
```

- **Output**

```
I like pop
I like rock
I like jazz
```

Python for loop (Cont.)

- **Else in For Loop**
- The else keyword in a for loop specifies a block of code to be executed when the loop is finished.
- **Example:** Print all numbers from 0 to 5, and print a message when the loop has ended.
- ```
  for x in range(6):
    print(x)
  else:
    print("Finally finished!")
  ```
  ```

Python for loop (Cont.)

Note: The else block will NOT be executed if the loop is stopped by a break statement.

- Example:
- ```
  for x in range(6):
    if x == 3: break
    print(x)
  else:
    print("Finally finished!")
  ```

The pass Statement

- for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.
- Example:
- ```
  for x in [0, 1, 2]:
    pass
  ```

Thank You