



INTRODUCTION TO PYTHON PROGRAMMING

UNIT III

For loops

- A `for` loop is used for iterating over a particular sequence (can be a list, a tuple, a dictionary, a set, or a string).
- This is less like the `for` keyword in other programming languages
- Similar to an iterator method as found in other object-orientated programming languages.
- With the `for` loop a set of statements, once for each item in a list, tuple, set etc can be executed.
- Used for sequential traversal.

A simple 'for' loop

Syntax:

```
for iterator_var in sequence:  
    statements(s)
```

Example 1:

```
n = 4  
for i in range(0, n):  
    print(i)
```

output: 0 1 2 3

Example 2:

```
for i in [4, 3, 2, 1] :  
    print(i)  
print('stop!')
```

output: 4 3 2 1 stop!

Example 3

```
friends = ['Joseph', 'Glenn', 'Sally']  
for friend in friends :  
    print('Hello:', friend)  
print('Done!')
```

Output:

Hello: Joseph

Hello: Glenn

Hello: Sally

Done!

Example 4

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

Loop through the letters in the word "banana":

```
for x in "banana":  
    print(x)
```

Nested Loops

Loops Inside 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":

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]
```

```
for x in adj:  
    for y in fruits:  
        print(x, y)
```

Control Statements

- Break
- Continue
- Pass


Break Statement

The break statement is used to terminate the loop immediately when it is encountered.

The syntax of the break statement is

```
for val in sequence:
    # code
    if condition:
        break
    # code
```

```
while condition:
    # code
    if condition:
        break
    # code
```



Break Statement

With the break statement we can stop the loop before it has looped through all the items:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        break  
    print(x)
```

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

Break Statement

```
# program to find first 5 multiples of 6
```

```
i = 1
```

```
while i <= 10:
```

```
    print('6 * ',(i), '=',6 * i)
```

```
    if i >= 5:
```

```
        break
```

```
    i = i + 1
```

Continue Statement

```
→ for val in sequence:  
    # code  
    if condition:  
        continue
```

```
    # code
```

```
→ while condition:  
    # code  
    if condition:  
        continue
```

```
    # code
```

Continue Statement

Python continue Statement with for Loop

We can use the continue statement with the for loop to skip the **current iteration of the loop**.

Then the control of the program jumps to the next iteration.
For example,

```
for i in range(5):  
    if i == 3:  
        continue  
    print(i)
```

Continue Statement

Python continue Statement with While Loop

We can use the continue statement with the While loop to skip the current iteration of the loop.

Then the control of the program jumps to the next iteration.

For example,

```
num = 0

while num < 10:
    num += 1

    if (num % 2) == 0:
        continue

    print(num)
```

Continue Statement

While Loop

```
i = 1
while i < 6:
    print(i)
    i += 1
```

1
2
3
4
5

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

1
2
4
5
6

for Loop

```
names = ["Arun", "Raju", "Charan"]
for x in names:
    print(x)
```

Arun
Raju
Charan

```
names = ["Arun", "Raju", "Charan"]
for x in names:
    if x == "Raju":
        continue
    print(x)
```

Arun
Charan

Pass Statement

- In Python programming, the pass statement is a **null statement**
 - which can be used as a placeholder for future code.
 - Suppose we have a loop or a function that is not
 - implemented yet, but we want to implement it in the future.
 - In such cases, we can use the pass statement.
 - The syntax of the pass statement is : **pass**
-
- `n = 10`
 - `# use pass inside if statement`
 - `if n > 10:`
 - `pass`
 - `print('Hello')`

Strings

- Accessing Strings
- Basic Operations
- String slices
- Function and Methods

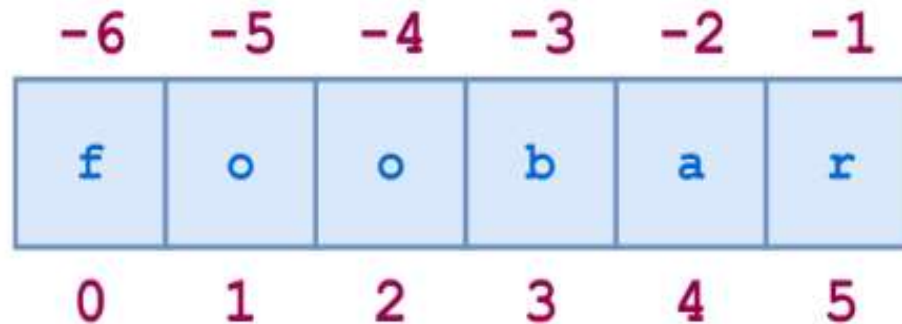
Strings

- String Initialization
a='Welcome, to RVCE!'
b = "Welcome, to RVCE!"
c="""Welcome,
to RVCE!"""
- Accessing characters in a string
a[1]

Strings

- String Indexing
- Let us say we have word “foobar”

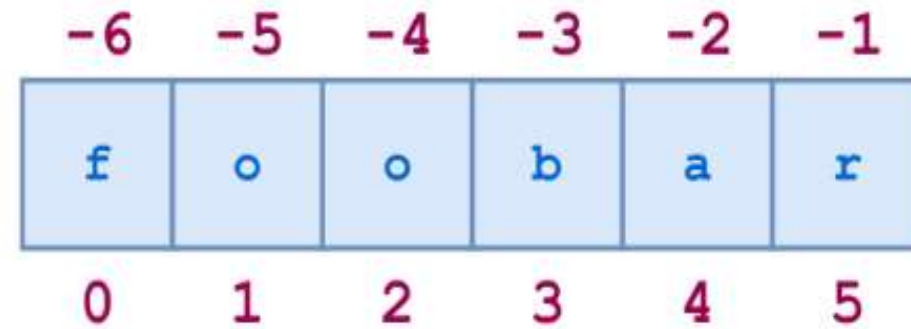
String indexing in Python is zero-based: the first character in the string has index 0, the next has index 1, and so on



Positive and Negative String Indices

Strings

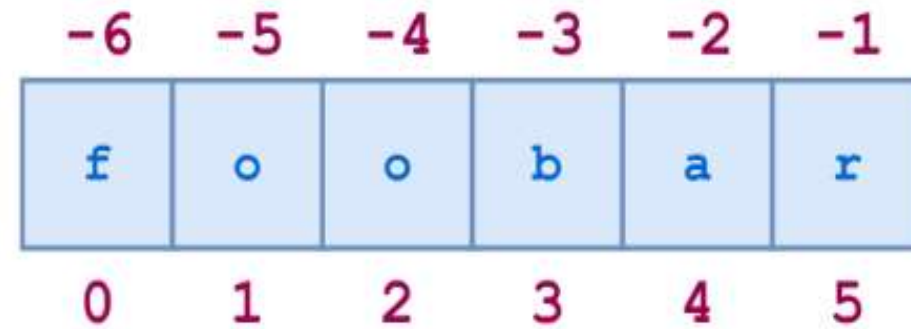
- The individual characters can be accessed by index as follows:
- `>>> s = 'foobar'`
- `>>> s[0]`
- `'f'`
- `>>> s[1]`
- `'o'`
- `>>> s[-1]`
- `'r'`



Positive and Negative String Indices

Strings

- The individual characters can be accessed by index as follows:
- `>>> s = 'foobar'`
- `>>> s[0]`
- `'f'`
- `>>> s[1]`
- `'o'`
- `>>> s[-1]`
- `'r'`



Positive and Negative String Indices

String Slicing

- Python also allows a form of indexing syntax that extracts substrings from a string, known as **string slicing**.
- If **s** is a string, an expression of the form **s[m:n]** returns the portion of **s** starting with position **m**, and up to but not including position **n**:
- **a='Welcome, to RVCE!'**
- **print(a)**
- **print(a[2:11])**
- **print(a[2:])** -- if you omit the second index as in **s[n:]**, the slice extends from the first index through the end of the string.
- **print(a[:3])** -- If you omit the first index, the slice starts at the beginning of the string. Thus, **s[:m]** and **s[0:m]** are equivalent

String- In built functions

strip() -- removes whitespace from the beginning or end

```
a = "    Welcome, to RVCE!    "  
print(a.strip())
```

```
#The strip() method removes any whitespace from the beginning or the end:  
a = "    Welcome, to RVCE!    "  
print(a.strip()) # returns "Hello, World!"  
print(a)
```

```
Welcome, to RVCE!
```

```
Welcome, to RVCE!
```

String- In built functions

- `len()` -- returns the length of a string

```
a = " Welcome, to RVCE! "  
print(len(a))
```

```
#The len() method returns the length of a string:  
a='Welcome, to RVCE!'  
print(len(a))
```

17

String- In built functions

- The `split()` method splits a string into a list.
- You can specify the separator, default separator is any whitespace.
- Syntax
- *`string.split('separator', maxsplit)`*

Parameter	Description
<i>separator</i>	Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator
<i>maxsplit</i>	Optional. Specifies how many splits to do.

- *`string.split ()`*

String- In built functions

Defining both Separator and MaxSplit

```
a = " Welcome, to, RVCE! "
```

```
print(a)
```

```
print(a.split(', ', 1))
```

#No Maxsplit is Specified

```
a = "Welcome,to, RVCE!"
```

```
print(a)
```

```
print(a.split(','))
```

#No Separator and Maxsplit are specified

```
a = "Welcome,to, RVCE!"
```

```
print(a)
```

```
print(a.split())
```

String- In built functions

lower() -- returns the lower case of the string

```
#The lower() method returns the string in lower case:  
a='Welcome, to RVCE!'  
print(a.lower())
```

welcome, to rvce!

upper() -- returns the upper case of the string

```
#The upper() method returns the string in upper case:  
a='Welcome, to RVCE!'  
print(a.upper())
```

WELCOME, TO RVCE!

replace() -- returns the replaced string

```
#The replace() method replaces a string with another string:  
a='Welcome, to RVCE!'  
print(a.replace("e", "i"))
```

Wilcomi, to RVCE!

String manipulation functions

- **capitalize()** Capitalizes first letter of string
- Eg: **Syntax:** `string_name.capitalize()`
- **Parameter:** *The capitalize() function does not takes any parameter.*
- **Return:** *The capitalize() function returns a string with the first character in the capital.*
- Eg: `name = "geeks FOR geeks"`
`print(name.capitalize())`

Output:

Geeks for geeks

String manipulation functions

- [centre \(width, fillchar\)](#)
- Python String center() method creates and returns a new string that is padded with the specified character.
- Syntax: `string.center(length[, fillchar])`
- Parameters:
- length: length of the string after padding with the characters.
- fillchar: (optional) characters which need to be padded. If it's not provided, **space is taken as the default argument**.
- Returns: Returns a string padded with specified fillchar and it doesn't modify the original string.

String manipulation functions

- [centre \(width, fillchar\)](#)
- Python String center() method creates and returns a new string that is padded with the specified character.
- Syntax: `string.center(length[, fillchar])`
- `string = "geeks for geeks"`
- `new_string = string.center(24, '#')`
- `print(new_string)`
- **If the length argument value is less than original String length, then the string is unchanged**

String manipulation functions

- **Count** : The **count()** method returns the number of times a specified value appears in the string.
- Syntax
- **string.count(value, start, end)**

Parameter Values

Parameter	Description
<i>value</i>	Required. A String. The string to value to search for
<i>start</i>	Optional. An Integer. The position to start the search. Default is 0
<i>end</i>	Optional. An Integer. The position to end the search. Default is the end of the string

String manipulation functions

- `txt = "I love apples, apple are my favorite fruit"`
- `x = txt.count("apple", 10, 24)`
- `print(x)`

String manipulation functions

- **endswith:**

The endswith() method returns True if a string ends with the specified suffix. If not, it returns False.

The syntax

str.endswith(suffix[, start[, end]])

Example

message = 'Python is fun'

check if the message ends with fun

print(message.endswith('fun'))

String manipulation functions

- **Find**: The `find()` method returns the index of first occurrence of the substring (if found). If not found, it returns -1.
- The syntax of the `find()` method is:
`str.find(sub[, start[, end]])`
- The `find()` method takes maximum of three parameters:
- **sub** - It is the substring to be searched in the `str` string.
- **start and end (optional)** - The range `str[start:end]` within which substring is searched.
- Eg : `message = 'Python is a fun programming language'`
- `# check the index of 'fun'`
- `print(message.find('fun'))`
- `# Output: 12`

String manipulation functions

- [isalnum\(\)](#) Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
- **# string contains either alphabet or number**
- name1 = "Python3"
- print(name1.isalnum()) #True
- **# string contains whitespace**
- name2 = "Python 3"
- print(name2.isalnum()) #False
- [isdigit\(\)](#) Returns true if string contains only digits and false otherwise.
- [islower\(\)](#) Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
- [isspace\(\)](#) Returns true if string contains only whitespace characters and false otherwise.

String manipulation functions

- [isdigit\(\)](#) Returns true if string contains only digits and false otherwise.
- Example:
- `str1 = '342'`
- `print(str1.isdigit())`
- `str2 = 'python'`
- `print(str2.isdigit())`
- # Output: True and False respectively

String manipulation functions

- **islower()** The islower() method returns True if all alphabets in a string are lowercase alphabets. If the string contains at least one uppercase alphabet, it returns False
- **s = 'this is good'**
- **print(s.islower())**

- **s = 'th!s is aLso g00d'**
- **print(s.islower())**

- **s = 'this is Not good'**
- **print(s.islower())**

String manipulation functions

- [isspace\(\)](#) Returns true if string contains only whitespace characters and false otherwise.
- `str1 = ' '`
- `print(str1.isspace())`

String Formatting Operator

- %c for character
- %s for strings
- %d for numbers
- %x for hexadecimal numbers
- %f for floating point numbers

```
print('Joe stood up and %s to the crowd.' % 'spoke')
```

```
print('There are %d dogs.' % 4)
```

Tuples in Python

- **Tuples are ordered collections of heterogeneous data that are unchangeable.** Heterogeneous means tuple can store variables of all types.
 - *Tuple has the following characteristics*
- **Ordered:** Tuples are part of sequence data types, which means they hold the order of the data insertion. It maintains the index value for each item.
- **Unchangeable:** Tuples are unchangeable, which means that we cannot add or delete items to the tuple after creation.
- **Heterogeneous:** Tuples are a sequence of data of different data types (like integer, float, list, string, etc;) and can be accessed through indexing and slicing.
- **Contains Duplicates:** Tuples can contain duplicates, which means they can have items with the same value

Tuples in Python

PYnative.com

Tuples in Python

```
T = ( 20, 'Jessa', 35.75, [30, 60, 90] )
```

↑
T[0]↑
T[1]↑
T[2]↑
T[3]

- ✓ **Ordered:** Maintain the order of the data insertion.
- ✓ **Unchangeable:** Tuples are immutable and we can't modify items.
- ✓ **Heterogeneous:** Tuples can contains data of types
- ✓ **Contains duplicate:** Allows duplicates data

Creating 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.).

```
Eg : my_tuple = (1, "Hello", 3.4)
      print(my_tuple)
```

In Python, 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 a tuple**,

```
var1 = ("Hello") # This is string
var2 = ("Hello",) # This is tuple
```

Accessing Values in Tuple

- You can access tuple items by referring to the index number, inside square brackets:
- `thistuple = ("apple", "banana", "cherry")`
- `print(thistuple[1])`
- `print(thistuple[-3])`
- `Print(thistuple[0:2])`

Updating values in Tuple

- Tuples are immutable which means you cannot update or change the values of tuple elements.
 - **But there are some Hacks**
- **Add tuple to a tuple.** You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:
- Example :

```
thistuple = ("apple", "banana", "cherry")  
y = ("orange",)  
thistuple += y  
print(thistuple)
```

Updating values in Tuple

```
#take portions of existing tuples to create new tuples  
Tup1=('Amit', 'Anu', 'Akshay')  
Tup2 = (1, 2, 3, 4, 5 )  
Tup3=Tup1+Tup2  
print(Tup3)
```

```
('Amit', 'Anu', 'Akshay', 1, 2, 3, 4, 5)
```

Delete Tuple Elements

We will be using the operator called **del**

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple)  
del thistuple # This deletes the tuple  
print(thistuple)
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Python Expression	Results	Description
<code>len((1, 2, 3))</code>	3	Length
<code>(1, 2, 3) + (4, 5, 6)</code>	<code>(1, 2, 3, 4, 5, 6)</code>	Concatenation
<code>('Hi!') * 4</code>	<code>('Hi!', 'Hi!', 'Hi!', 'Hi!')</code>	Repetition
<code>3 in (1, 2, 3)</code>	True	Membership
<code>for x in (1, 2, 3): print x,</code>	1 2 3	Iteration

Example Code

```
thistuple = ("apple", "banana", "cherry")
y = (1,2,3)
print(len(thistuple))
print (thistuple+y)
print(y*2)
print(2 in y)
for x in y:
    print(x)
```

Basic Tuples Operations

```
Tup1=('Amit', 'Anu', 'Akshay')
Tup2 = (1, 2, 3, 4, 5 )
#Length
print(len(Tup1))
print(len(Tup2))
#Concatenation
Tup3=Tup1+Tup2
print(Tup3)
#Repetition
print(Tup2*3)
#Membership
print("Anu" in Tup3)
print("A" in Tup3)
print(5 in Tup2)
#iteration
for x in Tup3:
    print(x)
```

```
3
5
('Amit', 'Anu', 'Akshay', 1, 2, 3, 4, 5)
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
True
False
True
Amit
Anu
Akshay
1
2
3
4
5
```

Indexing, Slicing, and Matrixes

```
Tup = ("Apple", "Ball", "Camera", "Doll")  
print(Tup[3])  
print(Tup[-3])  
print(Tup[1:])  
print(Tup[:3])
```

Doll

Ball

('Ball', 'Camera', 'Doll')

('Apple', 'Ball', 'Camera')

Built-in Tuple Functions

Sr.No.	Function with Description	Code
1	<u>cmp(tuple1, tuple2)</u> Compares elements of both tuples.	
2	<u>len(tuple)</u> Gives the total length of the tuple.	
3	<u>max(tuple)</u> Returns item from the tuple with max value.	
4	<u>min(tuple)</u> Returns item from the tuple with min value.	
5	<u>tuple(seq)</u> Converts a list into tuple.	

Built-in Tuple Functions

```
Tup1=('Amit', 'Anu', 'Akshay')
Tup2 = (1, 2, 3, 4, 5 )
#print(cmp(Tup1,Tup2))
print(len(Tup1))
print(len(Tup2))
print(max(Tup1))
print(max(Tup2))
print(min(Tup1))
print(min(Tup2))
list=[1,2,3,'a']
lis=tuple(list)
print(lis)
```

```
3
5
Anu
5
Akshay
1
(1, 2, 3, 'a')
```

Lists in Python

- Lists are used to store multiple items using a single variable.
- Lists is a built-in data types
- Lists are used to store collections of data,
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] and so on
- List items can be of any data type
- Example : `list1 = ["abc", 34, True, 40, "male", "abc"]`

Lists in Python

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
```

```
tinylist = [123, 'john']
```

```
print (list)
```

Prints complete list

```
print (list[0])
```

Prints first element of the list

```
print (list[1:3])
```

Prints elements starting from 2nd till 3rd

```
print (list[2:])
```

Prints elements starting from 3rd element

```
print (tinylist * 2)
```

Prints list two times

```
print (list + tinylist)
```

Prints concatenated lists

Lists in Python

Output:

```
['abcd', 786, 2.23, 'john', 70.2]
```

```
abcd
```

```
[786, 2.23]
```

```
[2.23, 'john', 70.2]
```

```
[123, 'john', 123, 'john']
```

```
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Basic Operations of Lists in Python

Add Elements to a Python List

1. The [append\(\)](#) method adds an item at the end of the list

```
numbers = [21, 34, 54, 12]
```

```
numbers.append(32)
```

```
print(numbers)
```

2. **The insert() method** inserts an element to the list at the specified index

Syntax : `list.insert(position, item)`

Eg:

```
numbers = [21, 34, 54, 12]
```

```
numbers.insert(0, 32)
```

```
print(numbers)
```

Basic Operations of Lists in Python

To Remove Elements

The syntax of the `remove()` method is:

```
list.remove(element)
```

Eg: `numbers = [21, 34, 54, 12]`

```
numbers.remove(34)
```

```
print(numbers)
```

If a list contains duplicate elements, the `remove()` method only removes the first matching element.

If you need to delete elements based on the index (like the fourth element), you can use the [pop\(\) method](#). Eg : `numbers.pop(0)`

Dictionary

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates
- Key is separated from its value by a colon (:)
- Items are separated by commas
- Dictionary is enclosed in curly braces
- Keys are unique within a dictionary while values may not be.

```
thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964}
```


Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}  
print (dict['Name'])  
print (dict['Age'])
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows

```
print (dict['Height'])
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}  
dict['Age'] = 8; # update existing entry  
dict['School'] = "DPS School"; # Add new entry
```

Delete Dictionary Elements

You can either remove **individual dictionary elements** or **clear the entire contents of a dictionary**. You can also **delete entire dictionary** in a single operation

1. To remove individual dictionary elements

: We will use **del** on Key element which has to be removed

Eg :

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
```

```
del dict['Name']; # remove entry with key 'Name'
```

2. To Clear the contents of dictionary : I will use the function **clear**

```
dict.clear(); # remove all entries in dict
```

3. To delete entire dictionary : **del dict**

Clear Dictionary elements

```
#Clear Dictionary elements
```

```
D = {'Name': 'Raghu', 1:28, 'Designation': 'Manager'}
```

```
print(D)
```

```
D.clear()
```

```
print("After deleting\n",D)
```

```
{'Name': 'Raghu', 1: 28, 'Designation': 'Manager'}
```

```
After deleting
```

```
{}
```

Properties of Dictionary Keys

- Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects.
- There are two important points to remember about dictionary keys –
- **(a)** More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.
- For example – `dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}`
- `print (dict['Name'])`

Properties of Dictionary Keys

Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Following is a simple example

```
dict = {'Name': 'Zara', 'Age': 7}
```

```
print (dict['Name']) # You will get the error
```

Built-in Dictionary Functions

- `len(dict)` Gives the total length of the dictionary

```
dict = {'Name': 'Zara', 'Age': 7};  
print(len (dict))
```

- `str(dict)` Produces a string representation of dictionary

```
dict = {'Name': 'Zara', 'Age': 7};  
print(str (dict))
```

- `type(variable)` Returns the type of the passed variable.

```
dict = {'Name': 'Zara', 'Age': 7};  
print(type (dict))
```

Built-in dictionary methods

Function	Explanation	Syntax	Example /Code
copy()	returns a shallow copy of the dictionary.	dict.copy()	dict1 = {'Name': 'Zara', 'Age': 7}; dict2 = dict1.copy() print(dict2)
get()	returns a value for the given key	dict.get(key)	dict1 = {'Name': 'Zara', 'Age': 7}; print(dict.get('Name'))
has_key()	returns true if a given <i>key</i> is available in the dictionary, otherwise it returns a false.	dict.has_key(key))	dict1 = {'Name': 'Zara', 'Age': 7}; print(dict.has_Key('Name'))
update()	adds dictionary dict2's key-values pairs in to dict		dict = {'Name': 'Zara', 'Age': 7} dict2 = {'School': 'ABC'} dict.update(dict2) Print(dict)
keys()	returns a list of all the available keys in the dictionary.	dict.keys	dict = {'Name': 'Zara', 'Age': 7} print(dict.keys())

THANK
YOU