**STANDARD DATA TYPES**

.........................................................................................................................................................

The data stored in memory can be of many types. Python has five standard data types:

**1.Numbers**

**2.String   
3.List**

**4.Tuple**

**5.Dictionary**

Python Numbers

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.

Python supports different numerical types

1. int()
2. float()
3. complex(x,y)

## **Mathematical Functions**

Python includes following functions that perform mathematical calculations and all these functions cannot be used directly and for that we need to import the math function in it. Example-

|  |  |
| --- | --- |
| **Function** | **What it’ll Return ( description )** |
| abs(x) | The absolute value of x: the (positive) distance between x and zero. |
| ceil(x) | The ceiling of x: the smallest integer not less than x |
| cmp(x, y) | -1 if x < y, 0 if x == y, or 1 if x > y |
| exp(x) | The exponential of x: ex |
| fabs(x) | The absolute value of x. |
| floor(x) | The floor of x: the largest integer not greater than x |
| log(x) | The natural logarithm of x, for x> 0 |
| log10(x) | The base-10 logarithm of x for x> 0 . |
| max(x1, x2,...) | The largest of its arguments: the value closest to positive infinity |
| min(x1, x2,...) | The smallest of its arguments: the value closest to negative infinity |
| modf(x) | The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float. |
| pow(x, y) | The value of x\*\*y. |
| round(x [,n]) | x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0. |
| sqrt(x) | The square root of x for x > 0 |

## **Random Number Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| [**choice(seq)**](https://www.tutorialspoint.com/python/number_choice.htm) | A random item from a list, tuple, or string. |
| [**randrange ([start,] stop [,step])**](https://www.tutorialspoint.com/python/number_randrange.htm) | A randomly selected element from range(start, stop, step) |
| [**random()**](https://www.tutorialspoint.com/python/number_random.htm) | A random float r, such that 0 is less than or equal to r and r is less than 1 |
| [**shuffle(lst)**](https://www.tutorialspoint.com/python/number_shuffle.htm) | Randomizes the items of a list in place. Returns None. |
|  |  |

1. **Choice()**
2. import random
3. print "choice([1, 2, 3, 5, 9]) : ", random.choice([1, 2, 3, 5, 9])
4. print "choice('A String') : ", random.choice('A String')
5. When we run above program, it produces following result:
6. choice([1, 2, 3, 5, 9]) : 2
7. choice('A String') : n

**2. randrange()**

import random

# Select an even number in 100 <= number < 1000

print "randrange(100, 1000, 2) : ", random.randrange(100, 1000, 2)

# Select another number in 100 <= number < 1000

print "randrange(100, 1000, 3) : ", random.randrange(100, 1000, 3)

When we run above program, it produces following result:

randrange(100, 1000, 2) : 976

randrange(100, 1000, 3) : 520

**3. random()**

The method **random()** returns a random float r, such that 0 is less than or equal to r and r is less than 1.

import random

# First random number

print "random() : ", random.random()

# Second random number

print "random() : ", random.random()

When we run above program, it produces following result:

random() : 0.281954791393

random() : 0.309090465205

1. **shuffle()**

The method **shuffle()** randomizes the items of a list in place.

list = [20, 16, 10, 5];

random.shuffle(list)

print "Reshuffled list : ", list

random.shuffle(list)

print "Reshuffled list : ", list

When we run above program, it produces following result:

Reshuffled list : [16, 5, 10, 20]

Reshuffled list : [16, 5, 20, 10]

***STRINGS*** – They are bunch of characters. In other words they are identified as contiguous(IN SEQUENCE) set of characters. Subset of strings can be taken using slice operator [],[:]

## Accessing Values in Strings

var1 = 'Hello World!'

var2 = "Python Programming"

print "var1[0]: ", var1[0]

print "var2[1:5]: ", var2[1:5]

When the above code is executed, it produces the following result −

var1[0]: H

var2[1:5]: ytho

## Updating Strings

var1 = 'Hello World!'

print ("Updated String :- ", var1[:6] + 'Python')

String Formatting Operator

One of Python's coolest features is the string format operator %. Following is a simple example −

print "My name is %s and weight is %d kg!" % ('Mac', 21)

When the above code is executed, it produces the following result −

My name is Mac and weight is 21 kg!

Here is the list of complete set of symbols which can be used along with % −

|  |  |
| --- | --- |
| **Format Symbol** | **Conversion** |
| %c | Character |
| %s | string conversion via str() prior to formatting |
| %d | signed decimal integer |
| %f | floating point real number |

## Built-in String Methods

1. ***Capitalize() Method***

It returns a copy of the string with only its first character capitalized.

### **Syntax**

str.capitalize()

### **Example**

str = "this is string example....wow!!!"

print ("str.capitalize() : ", str.capitalize())

### **Result**

str.capitalize() : This is string example....wow!!!

1. ***Count() Method***

## Description

The method **count()** returns the number of occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

## Syntax

str.count(sub, start, end)

## Parameters

* **sub** -- This is the substring to be searched.
* **start** -- Search starts from this index. First character starts from 0 index. By default search starts from 0 index.
* **end** -- Search ends from this index. First character starts from 0 index. By default search ends at the last index.

## Return Value

Centered in a string of length width.

## Example

str = "this is string example....wow!!!"

sub = "i"

print "str.count(sub, 4, 40) : ", str.count(sub, 4, 40)

sub = "wow"

print "str.count(sub) : ", str.count(sub)

## Result

str.count(sub, 4, 40) : 2

str.count(sub, 4, 40) : 1

print "Encoded String: " + Str

print "Decoded String: " + Str.decode('base64','strict')

1. ***Find() Method***

## Description

It determines if string *str* occurs in string, or in a substring of string if starting index *beg* and ending index *end* are given.

## Syntax

str.find(str)

## Parameters

* **str** -- This specifies the string to be searched.
* **beg** -- This is the starting index, by default its 0.
* **end** -- This is the ending index, by default its equal to the lenght of the string.

## Return Value

Index if found and -1 otherwise.

## Example

str1 = "this is string example....wow!!!"

str2 = "exam"

print (str1.find(str2))

## Result

15

1. ***index() Method***

## Description

It determines if string *str* occurs in string or in a substring of string if starting index *beg* and ending index *end* are given. This method is same as find(), but raises an exception if sub is not found.

## Syntax

str.index(str, beg=0 end=len(string))

## Parameters

* **str** -- This specifies the string to be searched.
* **beg** -- This is the starting index, by default its 0.
* **end** -- This is the ending index, by default its equal to the length of the string.

## Return Value

Index if found otherwise raises an exception if str is not found.

## Example

str1 = "this is string example....wow!!!"

str2 = "exam"

print (str1.index(str2))

pri nt (str1.index(str2, 10))

print (str1.index(str2, 40))

## Result

15

15

Traceback (most recent call last):

File "test.py", line 8, in

print str1.index(str2, 40);

ValueError: substring not found

shell returned 1

1. ***isalnum() Method***

## Description

The method **isalnum()** checks whether the string consists of alphanumeric characters.

## Return Value

This method returns true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

## Example

The following example shows the usage of isalnum() method.

str = "this2009" # No space in this string

print (str.isalnum())

str = "this is string example....wow!!!"

print (str.isalnum())

When we run above program, it produces following result:

True

False

1. ***Isalpha() Method***

## Description

The method **isalpha()** checks whether the string consists of alphabetic characters only.

## Syntax

Following is the syntax for **islpha()** method −

str.isalpha()

## Example

The following example shows the usage of isalpha() method.

str = "this" # No space & digit in this string

print str.isalpha()

str = "this is string example....wow!!!"

print str.isalpha()

When we run above program, it produces following result −

True

False

1. ***islower() Method()***

## Description

The method **islower()** checks whether all the case-based characters (letters) of the string are lowercase.

## Syntax

Following is the syntax for **islower()** method −

str.islower()

## Example

The following example shows the usage of islower() method.

str = "THIS is string example....wow!!!";

print str.islower()

str = "this is string example....wow!!!";

print str.islower()

When we run above program, it produces following result −

False

True

1. ***join() Method***

## Description

The method **join()** returns a string in which the string elements of sequence have been joined by *str* separator.

## Syntax

Following is the syntax for **join()** method:

str.join(sequence)

## Example

The following example shows the usage of join() method.

s = "-"

seq = ("a", "b", "c") # This is sequence of strings.

print s.join( seq )

When we run above program, it produces following result −

a-b-c

1. **len() Method**

## Description

The method **len()** returns the length of the string.

## Syntax

Following is the syntax for **len()** method −

len( str )

## Example

The following example shows the usage of len() method.

str = "this is string example....wow!!!"

print "Length of the string: ", len(str)

When we run above program, it produces following result −

Length of the string: 32

1. ***ljust() Method***

## Description

The method **ljust()** returns the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if width is less than len(s).

## Syntax

Following is the syntax for **ljust()** method −

str.ljust(width[, fillchar])

## Parameters

* **width** -- This is string length in total after padding.
* **fillchar** -- This is filler character, default is a space.

## Return Value

This method returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

## Example

The following example shows the usage of ljust() method.

str = "this is string example....wow!!!"

print str.ljust(50, '0')

When we run above program, it produces following result −

this is string example....wow!!!000000000000000000

1. **lower() Method**

## Description

The method **lower()** returns a copy of the string in which all case-based characters have been lowercased.

## Syntax

Following is the syntax for **lower()** method −

str.lower()

## Example

The following example shows the usage of lower() method.

str = "THIS IS STRING EXAMPLE....WOW!!!"

print str.lower()

When we run above program, it produces following result −

this is string example....wow!!!

1. **lstrip() Method.**

## Description

The method **lstrip()** returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

## Syntax

Following is the syntax for **lstrip()** method −

str.lstrip([chars])

## Example

The following example shows the usage of lstrip() method.

str = " this is string example....wow!!! "

print str.lstrip()

str = "88888888this is string example....wow!!!8888888"

print str.lstrip('8')

When we run above program, it produces following result −

this is string example....wow!!!

this is string example....wow!!!8888888

# **rstrip() Method**

## **Description**

The method **rstrip()** returns a copy of the string in which all *chars* have been stripped from the end of the string (default whitespace characters).

## **Syntax**

Following is the syntax for **rstrip()** method −

str.rstrip([chars])

## **Example**

The following example shows the usage of rstrip() method.

#!/usr/bin/python

str = " this is string example....wow!!! "

print str.rstrip()

str = "88888888this is string example....wow!!!8888888"

print str.rstrip('8')

When we run above program, it produces following result −

this is string example....wow!!!

88888888this is string example....wow!!!

1. **rstrip() Method**

## **Description**

The method **strip()** returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

## **Syntax**

Following is the syntax for **strip()** method −

str.strip([chars])

## **Example**

The following example shows the usage of strip() method.

#!/usr/bin/python

str = "0000000this is string example....wow!!!0000000";

print str.strip( '0' )

When we run above program, it produces following result −

this is string example....wow!!!

1. ***maketrans() Method***

## Description

The method **maketrans()** returns a translation table that maps each character in the *intabstring* into the character at the same position in the *outtab* string. Then this table is passed to the translate() function.

**Note:** Both intab and outtab must have the same length.

## Syntax

Following is the syntax for **maketrans()** method −

str.maketrans(intab, outtab])

## Parameters

* **intab** -- This is the string having actual characters.
* **outtab** -- This is the string having corresponding mapping character.

## Example

The following example shows the usage of maketrans() method. Under this, every vowel in a string is replaced by its vowel position −

from string import maketrans # Required to call maketrans function.

intab = "aeiou"

outtab = "12345"

trantab = maketrans(intab, outtab)

str = "this is string example....wow!!!"

print str.translate(trantab)

When we run above program, it produces following result −

th3s 3s str3ng 2x1mpl2....w4w!!!

1. ***max() Method***

## Description

The method **max()** returns the max alphabetical character from the string *str*.

## Syntax

Following is the syntax for **max()** method −

max(str)

## Example

The following example shows the usage of max() method.

str = "this is really a string example....wow!!!"

print "Max character: " + max(str)

str = "this is a string example....wow!!!"

print "Max character: " + max(str)

When we run above program, it produces following result −

Max character: y

Max character: x

1. ***min() Method***

## Description

The method **min()** returns the min alphabetical character from the string *str*.

## Syntax

Following is the syntax for **min()** method:

min(str)

**Example**

The following example shows the usage of min() method.

str = "this-is-real-string-example....wow!!!"

print "Min character: " + min(str)

str = "this-is-a-string-example....wow!!!"

print "Min character: " + min(str)

When we run above program, it produces following result −

Min character: !

Min character: !

1. ***replace() method***

## Description

The method **replace()** returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

## Syntax

Following is the syntax for **replace()** method −

str.replace(old, new[, max])

## Parameters

* **old** -- This is old substring to be replaced.
* **new** -- This is new substring, which would replace old substring.
* **max** -- If this optional argument max is given, only the first count occurrences are replaced.

## Return Value

This method returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first count occurrences are replaced.

## Example

The following example shows the usage of replace() method.

str = "this is string example....wow!!! this is really string"

print str.replace("is", "was")

print str.replace("is", "was", 3)

When we run above program, it produces following result −

thwas was string example....wow!!! thwas was really string

thwas was string example....wow!!! thwas is really string

1. **split() Method**

## Description

The method **split()** returns a list of all the words in the string, using *str* as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to *num*.

## Example

The following example shows the usage of split() method.

str = “black,color”

print str.split( )

print str.split(',')

When we run above program, it produces following result −

[‘black,color’]

[‘black’,’color’]

1. ***upper() Method***

## Description

The method **upper()** returns a copy of the string in which all case-based characters have been uppercased.

## Syntax

Following is the syntax for **upper()** method −

str.upper()

## Example

The following example shows the usage of upper() method.

str = "this is string example....wow!!!"

print "str.capitalize() : ", str.upper()

When we run above program, it produces following result −

THIS IS STRING EXAMPLE....WOW!!!

### ***LISTS :***

List are the most versatile(ABLE TO ADAPT) of python. Lists contain item separated by comma and enclosed within square bracket.[]. List are similar to arrays in C. Its used for storing all data types.ex- int, float, string, etc.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

list1 = ['Lucy', 'Avergers', 1997, 2000]

## Accessing Values in Lists

list1 = ['physics', 'chemistry', 1997, 2000]

list2 = [1, 2, 3, 4, 5, 6, 7 ]

print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]

When the above code is executed, it produces the following result −

list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

## Updating Lists

list = ['physics', 'chemistry', 1997, 2000]

print "Value available at index 2 : "

print list[2]

list[2] = 2001

print "New value available at index 2 : "

print list[2]

When the above code is executed, it produces the following result −

Value available at index 2 :

1997

New value available at index 2 :

2001

## Delete List Elements

list1 = ['physics', 'chemistry', 1997, 2000]

print list1

del list1[2]

print "After deleting value at index 2 : "

print list1

## Basic List Operations

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

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

Python List Builtin Methods

1. append() Method

Description

The method **append()** appends a passed *obj* into the existing list.

Syntax

Following is the syntax for **append()** method −

list.append(obj)

Example

The following example shows the usage of append() method.

aList = [123, 'xyz', 'zara', 'abc']

aList.append( 2009 )

print "Updated List : ", aList

When we run above program, it produces following result −

Updated List : [123, 'xyz', 'zara', 'abc', 2009]

1. Python List count() Method

Description

The method **count()** returns count of how many times *obj* occurs in list.

Syntax

Following is the syntax for **count()** method −

list.count(obj)

Example

The following example shows the usage of count() method.

aList = [123, 'xyz', 'zara', 'abc', 123]

print "Count for 123 : ", aList.count(123)

print "Count for zara : ", aList.count('zara')

When we run above program, it produces following result −

Count for 123 : 2

Count for zara : 1

1. Python List extend() Method

Description

The method **extend()** appends the contents of *seq* to list.

Syntax

Following is the syntax for **extend()** method −

list.extend(seq)

Example

The following example shows the usage of extend() method.

aList = [123, 'xyz', 'zara', 'abc', 123]

bList = [2009, 'manni']

aList.extend(bList)

print ("Extended List : ", aList)

When we run above program, it produces following result −

Extended List : [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']

1. Python List index() Method

Description

The method **index()** returns the lowest index in list that *obj* appears.

Syntax

Following is the syntax for **index()** method −

list.index(obj)

Example

The following example shows the usage of index() method.

aList = [123, 'xyz', 'zara', 'abc']

print "Index for xyz : ", aList.index( 'xyz' )

print "Index for zara : ", aList.index( 'zara' )

When we run above program, it produces following result −

Index for xyz : 1

Index for zara : 2

1. Python List insert() Method

Description

The method **insert()** inserts object *obj* into list at offset *index*.

Syntax

Following is the syntax for **insert()** method −

list.insert(index, obj)

Parameters

* **index** -- This is the Index where the object obj need to be inserted.
* **obj** -- This is the Object to be inserted into the given list.

Return Value

This method does not return any value but it inserts the given element at the given index.

Example

The following example shows the usage of insert() method.

aList = [123, 'xyz', 'zara', 'abc']

aList.insert( 3, 2009)

print "Final List : ", aList

When we run above program, it produces following result −

Final List : [123, 'xyz', 'zara', 2009, 'abc']

1. Python List pop() Method

Description

The method **pop()** removes and returns last object or *obj* from the list.

Syntax

Following is the syntax for **pop()** method −

list.pop(obj=list[-1])

Parameters

* **obj** -- This is an optional parameter, index of the object to be removed from the list.

Return Value

This method returns the removed object from the list.

Example

The following example shows the usage of pop() method.

aList = [123, 'xyz', 'zara', 'abc']

print "A List : ", aList.pop()

print "B List : ", aList.pop(2)

When we run above program, it produces following result −

A List : abc

B List : zara

1. Python List remove() Method

Parameters

* **obj** -- This is the object to be removed from the list.

Return Value

This method does not return any value but removes the given object from the list.

Example

The following example shows the usage of remove() method.

aList = [123, 'xyz', 'zara', 'abc', 'xyz']

aList.remove('xyz')

print "List : ", aList

aList.remove('abc')

print "List : ", aList

When we run above program, it produces following result −

List : [123, 'zara', 'abc', 'xyz']

List : [123, 'zara', 'xyz']

1. Python List reverse() Method

Description

The method **reverse()** reverses objects of list in place.

Syntax

Following is the syntax for **reverse()** method −

list.reverse()

Example

The following example shows the usage of reverse() method.

aList = [123, 'xyz', 'zara', 'abc', 'xyz']

aList.reverse()

print "List : ", aList

When we run above program, it produces following result −

List : ['xyz', 'abc', 'zara', 'xyz', 123]

# Python List sort() Method

## Description

The method **sort()** sorts objects of list, use compare *func* if given.

## Syntax

Following is the syntax for **sort()** method −

list.sort([func])

## Example

The following example shows the usage of sort() method.

aList = [123, 'xyz', 'zara', 'abc', 'xyz']

aList.sort()

print "List : ", aList

When we run above program, it produces following result −

List : [123, 'abc', 'xyz', 'xyz', 'zara']

***TUPLES :***

They are just like list. They cannot be adjusted. We use () in tUPLES. It is a sequence of immutable(UNABLE TO CHANGE) objects. Value can’t be deleted or updated in it.

TUPLES VS LIST :

1. More memory efficient 1. Takes more memory
2. Cannot be adjusted. Means nothing can be changed 2. It’s adjustable. Can be updated

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

tup1 = ('physics', 'chemistry', 1997, 2000)

tup2 = (1, 2, 3, 4, 5 )

Accessing Values in Tuples:

tup1 = ('physics', 'chemistry', 1997, 2000)

tup2 = (1, 2, 3, 4, 5, 6, 7 )

print "tup1[0]: ", tup1[0]

print "tup2[1:5]: ", tup2[1:5]

When the above code is executed, it produces the following result −

tup1[0]: physics

tup2[1:5]: [2, 3, 4, 5]

## Updating Tuples

tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');

# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print tup3

When the above code is executed, it produces the following result −

(12, 34.56, 'abc', 'xyz')

## Delete Tuple Elements

tup = ('physics', 'chemistry', 1997, 2000)

print tup

del tup;

print "After deleting tup : "

print tup

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more −

('physics', 'chemistry', 1997, 2000)

After deleting tup :

Traceback (most recent call last):

File "test.py", line 9, in <module>

print tup;

NameError: name 'tup' is not defined

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.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

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

## Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

L = ('spam', 'Spam', 'SPAM!')

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | 'SPAM!' | Offsets start at zero |
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

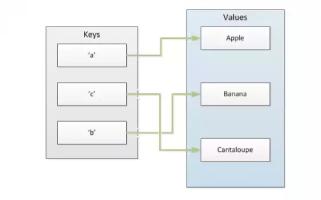
Built-in Tuple Functions

Python includes the following tuple functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 |  |
| 2 | [**len(tuple)**](http://www.tutorialspoint.com/python/tuple_len.htm)  Gives the total length of the tuple. |
| 3 | [**max(tuple)**](http://www.tutorialspoint.com/python/tuple_max.htm)  Returns item from the tuple with max value. |
| 4 | [**min(tuple)**](http://www.tutorialspoint.com/python/tuple_min.htm)  Returns item from the tuple with min value. |
| 5 | [**tuple(seq)**](http://www.tutorialspoint.com/python/tuple_tuple.htm)  Converts a list into tuple. |

Python Dictionary

A [python](https://deals.makeuseof.com/sales/2016-pure-python-bundle) dictionary is a data structure similar to an associative array found in other programming languages. An array or a list indexes elements by position. A dictionary, on the other hand, indexes elements by keys which can be strings. Think of a dictionary as unordered sets of key-value pairs.



Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be.

## **Accessing Values in Dictionary:**

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

print "dict['Name']: ", dict['Name']

print "dict['Age']: ", dict['Age']

When the above code is executed, it produces the following result −

dict['Name']: Zara

dict['Age']: 7

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

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

print "dict['Alice']: ", dict['Alice']

When the above code is executed, it produces the following result −

dict['Alice']:

Traceback (most recent call last):

File "test.py", line 4, in <module>

print "dict['Alice']: ", dict['Alice'];

KeyError: 'Alice'

## **Updating Dictionary**

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

dict['Age'] = 8 # update existing entry

dict['School'] = "DPS School" # Add new entry

print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']

When the above code is executed, it produces the following result −

dict['Age']: 8

dict['School']: DPS School

## **Delete Dictionary Elements**

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

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

dict.clear() # remove all entries in dict

del dict # delete entire dictionary

print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more −

dict['Age']:

Traceback (most recent call last):

File "test.py", line 8, in <module>

print "dict['Age']: ", dict['Age'];

TypeError: 'type' object is unsubscriptable

**Nested Dictionaries Examples**

DICTIONARY IN DICTIONARY

dict1={'Name':{'first':'will','last':'smith'},'Age':'17','occup':['Teaching','superviser','manager']}

print dict1['Name']

print dict1['Name']['last']

print dict1['Name']['first']

print (dict1['occup'][1])

print (dict1['occup'][0])

dict1['occup'].append('VP')

print dict1 ["occup"]

## **Built-in Dictionary Functions**

# Python dictionary copy() Method

## **Description**

The method **copy()** returns a shallow copy of the dictionary.

## **Syntax**

Following is the syntax for **copy()** method −

dict.copy()

## **Example**

The following example shows the usage of copy() method.

dict1 = {'Name': 'Zara', 'Age': 7}

dict2 = dict1.copy()

print "New Dictinary : %s" % str(dict2)

When we run above program, it produces following result −

New Dictinary : {'Age': 7, 'Name': 'Zara'}

# Python dictionary fromkeys() Method

## **Description**

The method **fromkeys()** creates a new dictionary with keys from *seq* and*values* set to value.

## **Syntax**

Following is the syntax for **fromkeys()** method −

dict.fromkeys(seq[, value]))

## **Example**

The following example shows the usage of fromkeys() method.

seq = ('name', 'age', 'sex')

dict = dict.fromkeys(seq)

print ("New Dictionary : %s" % str(dict))

dict = dict.fromkeys(seq, 10)

print "New Dictionary : %s" % str(dict)

When we run above program, it produces following result −

New Dictionary : {'age': None, 'name': None, 'sex': None}

New Dictionary : {'age': 10, 'name': 10, 'sex': 10}

# Python dictionary has\_key() Method

## **Description**

The method **has\_key()** returns true if a given *key* is available in the dictionary, otherwise it returns a false.

## **Syntax**

Following is the syntax for **has\_key()** method −

dict.has\_key(key)

## **Example**

The following example shows the usage of has\_key() method.

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

print "Value : %s" % dict.has\_key('Age')

print "Value : %s" % dict.has\_key('Sex')

When we run above program, it produces following result −

Value : True

Value : False

# Python dictionary keys() Method

## **Description**

The method **keys()** returns a list of all the available keys in the dictionary.

## **Syntax**

Following is the syntax for **keys()** method −

dict.keys()

## **Example**

The following example shows the usage of keys() method.

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

print "Value : %s"% dict.keys()

When we run above program, it produces following result −

Value : ['Age', 'Name']

# Python dictionary values() Method

## **Description**

The method **values()** returns a list of all the values available in a given dictionary.

## **Syntax**

Following is the syntax for **values()** method −

dict.values()

## **Example**

The following example shows the usage of values() method.

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

print "Value : %s" % dict.values()

When we run above program, it produces following result −

Value : [7, 'Zara']

# Python dictionary update() Method

## **Description**

The method **update()** adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

## **Syntax**

Following is the syntax for **update()** method −

dict.update(dict2)

## **Example**

The following example shows the usage of update() method.

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

dict2 = {'Sex': 'female' }

dict.update(dict2)

print "Value : %s" % dict

When we run above program, it produces following result −

Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}

## **Dictionary as a set of counters**

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.  
  
Here is what the code might look like:

word = 'brontosaurus'

d = dict()

for c in word:

if c not in d:

d[c] = 1

else:

d[c] = d[c] + 1

print (d)

The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c].  
  
Here's the output of the program:

{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}

## **Looping and dictionaries**

If you use a dictionary as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}

for key in counts:

print key, counts[key]

Here's what the output looks like:

jan 100

chuck 1

annie 42

Again, the keys are in no particular order.  
  
We can use this pattern to implement the various loop idioms that we have described earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}

for key in counts:

if counts[key] > 10 :

print key, counts[key]

The for loop iterates through the *keys* of the dictionary, so we must use the index operator to retrieve the corresponding *value* for each key. Here's what the output looks like:

jan 100

annie 42

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the keys method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key printing out key/value pairs in sorted order as follows as follows:

counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}

lst = counts.keys()

print lst

lst.sort()

for key in lst:

print key, counts[key]

Here's what the output looks like:

['jan', 'chuck', 'annie']

annie 42

chuck 1

jan 100

First you see the list of keys in unsorted order that we get from the keys method. Then we see the key/value pairs in order from the for loop.

Specifying Key-Value Pairs

You can also create and initialize a dictionary using name value pairs as keyword arguments to the dict() constructor.

p=dict(california=33,Colorado=46356)

print p

Output:

{'california': 33, 'Colorado': 46356}

Array of Key-Value Tuples

Yet another way of creating a dictionary is to use an array of key-value tuples. Here is the same example as above

pairs = [('California', 37253956), ('Colorado', 56)]

print dict(pairs)

Output:

{'California': 37253956, 'Colorado': 56}

Dict Comprehension

print {x: x\*\*2 for x in range(10, 20)}

Output:

{10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225, 16: 256, 17: 289, 18: 324, 19: 361}

How does it work? The latter part (for x in range(10, 20)) returns a range of numbers in the specified range. The dict comprehension part ({x: x\*\*2 ..}) loops over this range and initializes the dictionary.

Checking for Existence

What if you want to check for the presence of a key without actually attempting to access it (and possibly encountering a KeyError as above)? You can use the in keyword in the form key in dct which returns a boolean

dic={'Name':'Aman','Age':'17','occup':'Teaching'}

print 'Name' in dic

print 'name' in dic

Output:

True

False

Deleting elements

Use the pop() method instead, when you want the deleted value back

dic={'Name':'Aman','Age':'17','occup':'Teaching'}

print dic.pop('Age')

print dic

Output:

17

{'Name': 'Aman', 'occup': 'Teaching'}