Python Tuples

A tuple is a collection which is **ordered, unchangeable (immutable) and heterogeneous**. In Python tuples are written with round brackets(parenthesis).

```
fruitstuple = ("apple", "banana", "cherry", "orange")
```

- **A tuple has zero or more elements.**
- **The element of the tuple can be of any type. There is no restriction on the type of the element.**
- **A tuple has elements which could be of the same type (homogeneous) or of different. types(heterogeneous)**
- **Each element of a tuple can be referred to by a index or a subscript. An index is an integer**
- **Access to an element based on index or position takes the same time no matter where the element is in the tuple – random access.**
- **Tuples are immutable. Once created, we cannot change the number of elements – no append, no insert, no remove, no delete.**
- **Elements of the tuple cannot be assigned.**
- **A tuple cannot grow and cannot shrink. Size of the tuple can be found using the function len.**
- **Elements in the tuple can repeat**
- **Tuple is a sequence**
- **Tuple is also iterable - is eager and not lazy.**
- **Tuples can be nested. We can have tuple of tuples.**
- **Assignment of one tuple to another causes both to refer to the same tuple.**
- **Tuples can be sliced. This creates a new tuple**
- **You may walk through this program to understand the creation and the operations on a tuple.**
- **If the element of a tuple is a list, then the list can be modified by adding elements or removing them.**
- **We cannot replace the element of the tuple by assignment even if it is a list.**

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Creation of tuple:

- **Empty tuple**
  - `t1 = ()`
  - `t2 = tuple()`
- **Two element tuple**
  - `t3 = (1, 2)`
- **One element tuple**
  - `t4 = (10) # NO`
  - `t5 = (20,) # Yes`

There is an ambiguity when we use parentheses around a single expression. Should we consider this as an expression or a tuple? The language considers this as an expression. **To make a tuple of single element, we require an extra comma.**

```
# name : 1_tuple.py
# tuple
#     is a sequence
#     like a list
#     indexed by int; leftmost element has an index 0
#     select the element using []
#     is immutable
#     once created, cannot be changed
#     length of the tuple cannot change
#     heterogeneous
#     iterable

a = (11, 33, 22, 44, 55)
print(a)
print(a[2]) # 22
print(a[2:4]) # (22, 44)

a[2] = 222 # NO
a.append(66) # Error
```

```
a = (11, 33, 22, 44, 55)
#ok; a new tuple created
a = a + (111, 222)
print(a)       #(11, 33, 22, 44, 55, 111, 222)

b = ([12, 23], {34 : 45}, "56" )
print(b, len(b))

b[0].append(67) #ok
# b[0] = [78, 89] #no
# del b[0]  # no
# b[0] = b[0] + [100]  # no ; assignment forbidden

a = (11, 33, 22, 44, 55)
for i in a :
      print(i, end = " ")
print()

c = [11, 22, 33, 44]
for i in c :
      print("one") # 4 times

for i in [c] :
      print("two") # once

for i in (c) : # not a tuple
      print("three") # 4 times

for i in (c,) : # a tuple
      print("four") # once

print( (3, 4) * 2) # (3, 4, 3, 4)
print( (3 + 4) * 2) # 14
print( (3 + 4,) * 2) # (7, 7)
# tuple of one element requires an extra comma
```

```
d = ()
print(d, type(d))

e = (11, 33, 11, 11, 44, 33)
print(e.count(11)) # 3
print(e.count(33)) # 2
print(e.count(55)) # 0
print(e.index(44)) # 4
print(e.index(11)) # 0
print(e.index(55)) # error
```

In this example, we will consider a few uses of tuples.

```
a = 1, 2, 3
```

The above statement creates a tuple of 3 elements. Parentheses are optional.

```
x, y, z = a
```

This will cause the elements of the tuple to be assigned to x, y and z.

We may also use unnamed tuples while assigning to # of variables.

```
a, b = 11, 22
```

The corresponding elements are assigned thereby a becomes 11 and b becomes 22.

```
(a, b) = (b, a) # swaps two variables
```

This causes swapping of two variables. The expressions on the right of assignment are evaluated before the assignment is made. So, even if the variables occur on both sides of assignment operator, there is no ambiguity.

 # name: 2_tuple.py

```
a = 1, 2, 3
print(a, type(a))
x, y, z = a
print(x, y, z)
x, y = a   # error; # of variables on the left should match the # of elem in the tuple


# use of unnamed tuple
a, b = 11, 22      # equivalent to (a, b) = 11, 22
```

```
print("a : ", a, " b : ", b)
# in case of assignment, the right hand side is completely evaluated before
assignment
(a, b) = (b, a) # swaps two variables  # (a, b) = (22, 11)
print("a : ", a, " b : ", b)

#-------------------------------------------------------------------------
score = { }
score['gavaskar'] = 10000
print(score)
{'gavaskar': 10000}

# many times, key has components : composite key
# should be immutable; cannot be a list
score = { }
#score[['sunil', 'gavaskar']] = 10000 # NO
#score[['rohan', 'gavaskar']] = 1000 # NO
score[('sunil', 'gavaskar')] = 10000
score[('rohan', 'gavaskar')] = 1000
print(score)
print(score['rohan', 'gavaskar'])
```

Tuples are also used as keys of *dictionary* whenever the key has multiple components – key of the *dictionary* is a composite.

The key of a *dictionary* should be immutable.

So the key cannot be a list, but it can be a tuple.

Data structure: str : string

A string is a sequence of characters. Python directly supports a str type; but there is no character type.

- **A string has zero or more characters**
- **Each character of a string can be referred to by a index or a subscript**
- **An index is an integer**
- **Access to an element based on index or position takes the same time no matter where the element is in the string – random access**
- **Strings are immutable. Once created, we cannot change the number of elements – no append, no insert, no remove, no delete.**
- **Elements of the string cannot be assigned.**
- **A string can not grow and cannot shrink. Size of the string can be found using the function len.**
- **String is a sequence**
- **String is also iterable - is eager and not lazy.**
- **Strings cannot be nested.**
- **Strings can be sliced. This creates a new string.**

There are 4 types of string literals or constants

1) Single quoted strings

2) Double quoted strings

There is no difference between the two. In both these strings, escape sequences like \t, \n are expanded.

We can use double quotes in a single quoted string and single quote in double quoted string without escaping.

These strings can span just a line – cannot span multiple lines.

c) Triple quoted strings

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

An object's docsting is defined by including a string constant as the first statement in the object's definition.

It's specified in source code that is used, like a comment, to document a specific segment of code.

Unlike conventional source code comments the docstring should describe what the function does, not how.

All functions should have a docstring.

This allows the program to inspect these comments at run time, for instance as an interactive help system, or as metadata.

Docstrings can be accessed by the __doc__ attribute on objects.

Declaration of docstrings

The following Python file shows the declaration of docstrings within a python source file:

```
"""
Assuming this is file mymodule.py, then this string, being the first statement in the
file, will become the "mymodule" module's docstring when the file is imported.
"""
class MyClass(object):
    """The class's docstring"""
     def my_method(self):
        """The method's docstring"""

def my_function():
    """The function's docstring"""
```

How to access the Docstring

The following is an interactive session showing how the docstrings may be accessed

```
>>> import mymodule
>>> help(mymodule)
```

Assuming this is file mymodule.py then this string, being the first statement in

the file will become the mymodule modules docstring when the file is imported.

```
>>> help(mymodule.MyClass)
The class's docstring

>>> help(mymodule.MyClass.my_method)
The method's docstring

>>> help(mymodule.my_function)
The function's docstring
```

d) Raw strings

There are cases where the escape sequence should not be expanded – we require such strings as patterns in pattern matching using regular expressions.

In such cases, we prefix r to the string literal – it becomes a raw string.

```
# name : str3.py

"""
this program
is about playing
with strings
"""
# stored in a variable : __doc__
# strings
#     sequence
#     indexed
#     leftmost index : 0
#     immutable
#     no character type
#     can be sliced
#     cannot assign

a = "rose"
print(a, type(a), a[2], type(a[2]))
#a[0] = 'b' # error

# make a string:
# 1. single quotes
# 2. double quotes
#     no diference between them
#     escape sequences are expanded

s1 = "this is a \n string"
s2 = 'this is a \n string'
print(s1, type(s1))
print(s2, type(s2))

x = "indira gandhi is nehru's daughter's name"
print(x)
```

```
# 3. raw string
# no escaping
s3 = r"this is a \n string"
print(s3, type(s3))

# 4. triple quoted string
s4 = """
we love python
very much
"""
print(s4, type(s4))
print("document string : ", __doc__)
```

The following example shows how to play with the strings. The str type has lots of useful functions.

Build a string in stages

```
ss = “” # create an empty string
ss = ss + “something”   # create a new string by concatenation
```

Call some utility function like upper or replace

```
x = “abcd”; x.upper(); print(x)
```

We will observe that x has not changed. Remember that x is immutable.

In case we want the original string to change, assign the result of the function call back to the same variable – thus recreating the variable.

```
x = “abcd”; x = x.upper(); print(x)
```

Observe that replace will return a new string modifying every occurrences of the old string with the replace string. We can control the number of changes by using a count as the third argument.

Index finds the leftmost occurrence of a substring in the given string.

Finding the nth occurrence or replacing the nth occurrence become programming exercises.

```
# name : 4_str.py
"""
```

```
s = 'mohanDas Karamchand gandhi'
# print "m K gandhi"
# make a list of words by splitting
# output the first char of each word but for the last; output the last word

ss = ''
namelist = s.split()
for name in namelist[:len(namelist)-1]:
      #print(name[0])
      ss = ss + name[0] + " "
#print(namelist[-1]) # last elem
ss = ss + namelist[-1]
print(ss)
#ss.upper() # NO; does not change the str; returns a new changed string
ss = ss.upper()
print(ss)

ss = ss.title()
print(ss)

s = s.title()
print(s)

"""
mylist = [
"indira gandhi",
"m k gandhi",
"rahul gandhi",
"jawaharlal nehru",
"sardar patel",
"brijesh patel"
]
for w in mylist :
      if w.endswith('gandhi') :
            print(w)
```

```
s = "bad python bad teacher bad lecture"
print(s.replace('bad', 'good')) # default : all occurrences
print(s.replace('bad', 'good', 1))

s = "bad python bad teacher bad lecture"

# fnd the leftmost bad
print(s.index('bad')) #searches the string for a specified value and returns the position

# fnd the second bad from left
print(s.index('bad', s.index('bad') + len('bad')))  #str.index(value,start,end)

i = s.index('bad', s.index('bad') + len('bad'))
print(s[i:].replace('bad', 'worst', 1))     #str.replace(newvalue,oldvalue,count)
print(s[:i] + s[i:].replace('bad', 'worst', 1))
```

In the below first case, the first letter of each word is printed at the end.

In the below second case, after each character, a * is printed.

```
# name: 5_str.py
s = "we love python very much"
for w in s.split():
      print(w[1:], end = "")
      print(w[0], end = " ")
print()

for ch in s:
      print(ch, end = "")
      print('*', end = "")
print()
```

Outputs are:

```
ew ovel ythonp eryv uchm
w*e* *l*o*v*e* *p*y*t*h*o*n* *v*e*r*y* *m*u*c*h*
```

Python String Methods

Python has a set of built-in methods that you can use on strings.

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string

lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Note: All string methods returns new values. They do not change the original string.

Python String capitalize() Method

Example

Upper case the first letter in this sentence:

```
txt = "hello, and welcome to my world."
x = txt.capitalize()
print (x)
```

Definition and Usage

The capitalize() method returns a string where the first character is upper case.

Syntax

```
string.capitalize()
```

Parameter Values

No parameters

Python String casefold() Method

The casefold() method is an aggressive lower() method which convert strings to casefolded strings for caseless matching.

Example

Make the string lower case:

```
txt = "Hello, And Welcome To My World!"
x = txt.casefold()
print(x)
```

Definition and Usage

The casefold() method returns a string where all the characters are lower case.

This method is similar to the lower() method, but the casefold() method is stronger, more aggressive, meaning that it will convert more characters into lower case, and will find more matches when comparing two strings and both are converted using the casefold() method.

Syntax

```
string.casefold()
```

Parameter Values

No parameters

Python String center() Method

Example

Print the word "banana", taking up the space of 20 characters, with "banana" in the middle:

```
txt = "banana"
x = txt.center(20)
print(x)
```

Definition and Usage

The center() method will center align the string, using a specified character (space is default) as the fill character.

Syntax

```
string.center(length, character)
```

Parameter Values

Parameter	Description
length	Required. The length of the returned string
character	Optional. The character to fill the missing space on each side. Default is " " (space)

More Examples

Example

Using the letter "O" as the padding character:

```
txt = "banana"
x = txt.center(20, "O")
print(x)
```

Python String format() Method

The format() method returns the formatted string.

Example

Insert the price inside the placeholder, the price should be in fixed point, two-decimal format:

```
txt = "For only {price:.2f} dollars!"
print(txt.format(price = 49))
```

Definition and Usage

The format() method formats the specified value(s) and insert them inside the string's placeholder.

The placeholder is defined using curly brackets: {}. Read more about the placeholders in the Placeholder section below.

Syntax

```
string.format(value1, value2...)
```

Parameter Values

Parameter	Description
value1, value2...	Required. One or more values that should be formatted and inserted in the string. The values are either a list of values separated by commas, a key=value list, or a combination of both. The values can be of any data type.

The Placeholders

The placeholders can be identified using named indexes {price}, numbered indexes {0}, or even empty placeholders {}.

Example

Using different placeholder values:

```
txt1 = "My name is {fname}, I'am {age}".format(fname = "John", age = 36)
txt2 = "My name is {0}, I'am {1}".format("John",36)
txt3 = "My name is {}, I'am {}".format("John",36)
```

Formatting Types

Inside the placeholders you can add a formatting type to format the result:

:<	Left aligns the result (within the available space)
:>	Right aligns the result (within the available space)
:^	Center aligns the result (within the available space)
:=	Places the sign to the left most position
:+	Use a plus sign to indicate if the result is positive or negative
:-	Use a minus sign for negative values only

:	Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)
:,	Use a comma as a thousand separator
:_	Use a underscore as a thousand separator
:b	Binary format
:c	Converts the value into the corresponding unicode character
:d	Decimal format
:e	Scientific format, with a lower case e
:E	Scientific format, with an upper case E
:f	Fix point number format
:F	Fix point number format, in uppercase format (show inf and nan as INF and NAN)
:g	General format
:G	General format (using a upper case E for scientific notations)
:o	Octal format
:x	Hex format, lower case
:X	Hex format, upper case
:n	Number format
:%	Percentage format

```
#Use "<" to left-align the value:
txt = "We have {:<8} chickens."
print(txt.format(49))
```

```
We have 49       chickens.
```

```
#Use "=" to place the plus/minus sign at the left most position:
txt = "The temperature is {:=8} degrees celsius."
print(txt.format(-5))
```

```
The temperature is -      5 degrees.
```

```
#Use "," to add a comma as a thousand separator:
txt = "The universe is {:,} years old."
print(txt.format(13800000000))
```

```
The universe is 13,800,000,000 years old.
```

```
#Use "b" to convert the number into binary format:
```

```
txt = "The binary version of {0} is {0:b}"
print(txt.format(5))
```

```
The binary version of 5 is 101
```

```
#Use "d" to convert a number, in this case a binary number, into decimal number format:
txt = "We have {:d} chickens."
print(txt.format(0b101))
```

```
We have 5 chickens.
```

```
#Use "%" to convert the number into a percentage format:
txt = "You scored {:%}"
print(txt.format(0.25))
#Or, without any decimals:
txt = "You scored {:.0%}"
print(txt.format(0.25))
```

```
You scored 25.000000%
You scored 25%
```

Python String isidentifier() Method

Example

Check if the string is a valid identifier:

```
txt = "Demo"
x = txt.isidentifier()
print(x)
```

Definition and Usage

The `isidentifier()` method returns True if the string is a valid identifier, otherwise False.

A string is considered a valid identifier if it only contains alphanumeric letters (a-z) and (0-9), or underscores (_). A valid identifier cannot start with a number, or contain any spaces.

Syntax

```
string.isidentifier()
```

Parameter Values

No parameters.

More Examples

Example

Check if the strings are valid identifiers:

```
a = "MyFolder"
b = "Demo002"
c = "2bring"
d = "my demo"

print(a.isidentifier())
print(b.isidentifier())
print(c.isidentifier())
print(d.isidentifier())
```

Python String isprintable() Method

Example

Check if all the characters in the text are printable:

```
txt = "Hello! Are you #1?"

x = txt.isprintable()

print(x)
```

Definition and Usage

The isprintable() method returns True if all the characters are printable, otherwise False.
Example of none printable character can be carriage return and line feed.

Syntax

```
string.isprintable()
```

Parameter Values

No parameters.

More Examples

Check if all the characters in the text are printable:

```
txt = "Hello!\nAre you #1?"
x = txt.isprintable()
print(x)
```

Python String join() Method

Example

Join all items in a tuple into a string, using a hash character as separator:

```
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)
```

Definition and Usage

The join() method takes all items in an iterable and joins them into one string.

A string must be specified as the separator.

Syntax

```
string.join(iterable)
```

Parameter Values

Parameter	Description
iterable	Required. Any iterable object where all the returned values are strings

More Examples

Example 1

Join all items in a dictionary into a string, using a the word "TEST" as separator:

```
myDict = {"name": "John", "country": "Norway"}
mySeparator = "TEST"

x = mySeparator.join(myDict)

print(x)
```

```
nameTESTcountry
```

Note: When using a dictionary as an iterable, the returned values are the keys, not the values.

Example 1

```
list1 = [‘hi’, ‘hello’, ‘how are you?’]
S1 = ’AND’
S2 = s1.join(list1)
Print(s2)
hiANDhelloANDhow are you?
```

Python String index() Method

Example

Where in the text is the word "welcome"?:

```
txt = "Hello, welcome to my world."
x = txt.index("welcome")
print(x)
```

Definition and Usage

The index() method finds the first occurrence of the specified value.

The index() method raises an exception if the value is not found.

The index() method is almost the same as the find() method, the only difference is that the find() method returns -1 if the value is not found.

Syntax

```
string.index(value, start, end)
```

Parameter Values

Parameter	Description
value	Required. The value to search for
start	Optional. Where to start the search. Default is 0
end	Optional. Where to end the search. Default is to the end of the string

More Examples

Example

Where in the text is the first occurrence of the letter "e"?:

```
txt = "Hello, welcome to my world."
x = txt.index("e")
print(x)
```

Python String replace() Method

Example

Replace the word "bananas":

```
txt = "I like bananas"
x = txt.replace("bananas", "apples")
print(x)
```

Definition and Usage

The replace() method replaces a specified phrase with another specified phrase.

Note: *All* occurrences of the specified phrase will be replaced, if nothing else is specified.

Syntax

```
string.replace(oldvalue, newvalue, count)
```

Parameter Values

Parameter	Description
oldvalue	Required. The string to search for
newvalue	Required. The string to replace the old value with
count	Optional. A number specifying how many occurrences of the old value you want to replace. Default is all occurrences

More Examples

Replace all occurrence of the word "one":

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three")
print(x)
```

Example

Replace the two first occurrence of the word "one":

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three", 2)
print(x)
```

Python String partition() Method

Example

Search for the word "bananas", and return a tuple with three elements:

1 - everything before the "match"
2 - the "match"
3 - everything after the "match"

```
txt = "I could eat bananas all day"
x = txt.partition("bananas")
print(x)
```

```
('I could eat ', 'bananas', ' all day')
```

Definition and Usage

The partition() method searches for a specified string, and splits the string into a tuple containing three elements.

The first element contains the part before the specified string.

The second element contains the specified string.

The third element contains the part after the string.

Note: This method search for the *first* occurrence of the specified string.

Syntax

```
string.partition(value)
```

Parameter Values

Parameter	Description
value	Required. The string to search for

More Examples

If the specified value is not found, the partition() method returns a tuple containing: 1 - the whole string, 2 - an empty string, 3 - an empty string:

```
txt = "I could eat bananas all day"
x = txt.partition("apples")
print(x)
```

Python String rsplit() Method

Example

Split a string into a list, using comma, followed by a space (,) as the separator:

```
txt = "apple, banana, cherry"
x = txt.rsplit(", ")
print(x)
```

```
['apple', 'banana', 'cherry']
```

Definition and Usage

The rsplit() method splits a string into a list, starting from the right.

If no "max" is specified, this method will return the same as the split() method.

Note: When max is specified, the list will contain the specified number of elements *plus one*.

Syntax

```
string.rsplit(separator, max)
```

Parameter Values

Parameter	Description
separator	Optional. Specifies the separator to use when splitting the string. Default value is a whitespace
max	Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences"

More Examples

Split the string into a list with maximum 2 items:

```
txt = "apple, banana, cherry"

# setting the max parameter to 1, will return a list with 2 elements!
x = txt.rsplit(", ", 1)

print(x)
```

```
['apple, banana', 'cherry']
```