## **Python Tuples**

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are:

* Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed.
* Tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as **read-only** lists.

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

tinytuple = (123, 'john')

print(tuple) # Prints complete list

print(tuple[0]) # Prints first element of the list

print(tuple[1:3]) # Prints elements starting from 2nd till 3rd

print(tuple[2:]) # Prints elements starting from 3rd element

print(tinytuple \* 2) # Prints list two times

print(tuple + tinytuple) # Prints concatenated lists

This produce the following result −

('abcd', 786, 2.23, 'john', 70.200000000000003)

abcd

(786, 2.23)

(2.23, 'john', 70.200000000000003)

(123, 'john', 123, 'john')

('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

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

print(tuple[3:5])

['john', 70.2]

print(list[3:5])

['john', 70.2]

# Update element in tuple.

tuple[2] = 1000 **# Invalid syntax with tuple**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

list[2] = 1000 **# Valid syntax with list**

print(list)

['abcd', 786, 1000, 'john', 70.2]

del tuple[2]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object doesn't support item deletion

del list[2]

print(list)

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

## **Python Dictionary**

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

#!/usr/bin/python

dict = {}

dict['one'] = "This is one"

dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print(dict['one']) # Prints value for 'one' key

print(dict[2]) # Prints value for 2 key

print(tinydict) # Prints complete dictionary

print(tinydict.keys()) # Prints all the keys

print(tinydict.values()) # Prints all the values

This produce the following result −

This is one

This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'}

['dept', 'code', 'name']

['sales', 6734, 'john']

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

**Extend and update the dictionaries.**

**# Dictionary extended.**

dict['Three'] = "This is three"

dict[4] = "This is four"

print(dict)

{'one': 'This is one', 2: 'This is two', 'Three': 'This is three', 4: 'This is four'}

dict['CG Bangalore'] = "Whitefield","ATP","DTP"

dict['CG Bangalore']

('Whitefield', 'ATP', 'DTP')

dict['CG Bangalore'][0]

'Whitefield'

dict['CG Bangalore'][1]

'ATP'

dict['CG Bangalore'][2]

'DTP'

**# Dictionary updated.**

dict[4] = "Today is 4th of the month"

print(dict)

{'one': 'This is one', 2: 'This is two', 'Three': 'This is three', 4: 'Today is 4th of the month'}

**# Delete an item from a dictionary**

del dict['Three']

print(dict)

{'one': 'This is one', 2: 'This is two', 4: 'Today is 4th of the month'}

dict.pop(4)

'Today is 4th of the month'

print(dict)

{'one': 'This is one', 2: 'This is two' }

**# Replace or recreate the whole items in a dictionary**

dict = {'EmpNo':114649, 'Ename':'Hari Yadav', 'Phone':'666777', 'State':'MH', 'Pin':556622}

print(dict)

{'EmpNo': 114649, 'Ename': 'Hari Yadav', 'Phone': '666777', 'State': 'MH', 'Pin': 556622}

del dict

emp\_rec = {7369 : ("SMITH","CLERK",7902,"17-DEC-80",80000,20), 7499 : ("ALLEN","SALESMAN",7698, "20-FEB-81",16000,300,30)}

print(emp\_rec)

{7369: ('SMITH', 'CLERK', 7902, '17-DEC-80', 80000, 20), 7499: ('ALLEN', 'SALESMAN', 7698, '20-FEB-81', 16000, 300, 30)}

Display data in vertical format like tablular record.

## **Data Type Conversion**

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

|  |  |
| --- | --- |
| **Function** | **Description** |
| int(x [,base]) | Converts x to an integer. base specifies the base if x is a string.  **# integer**  print("int(123) is:", int(123))  int(123) is: 123  # float  print("int(123.23) is:", int(123.23))  int(123.23) is: 123  # string  print("int('123') is:", int('123'))  int('123') is: 123  Like 123 = 1\*102 +2\*101+3\*100  123 = 100 + 20 + 3  **# binary 0b or 0B**  print("For 1010, int is:", int('1010', 2))  For 1010, int is: 10  Like 1010 = 1\*23 + 0\*22 + 1\*21 + 0\*20  1010 = 8+0+2+0 => 10  print("For 101010, int is:", int('101010', 2))  For 101010, int is: 42  print("For 0b101010, int is:", int('0b101010', 2))  For 0b101010, int is: 42  print("For 0B101010, int is:", int('0B101010', 2))  For 0B101010, int is: 42  **# octal 0o or 0O**  print("For 234, int is:", int('234', 8))  For 234, int is: 156  Like (234)8 = 2\*82 + 3\*81 + 4\*80  (234)8 = 128 + 24 + 4 => 156  print("For 0o234, int is:", int('0o234', 8))  For 0o234, int is: 156  **# hexadecimal**  print("For A, int is:", int('A', 16))  For A, int is: 10  print("For ABC, int is:", int('ABC', 16))  For ABC, int is: 2748  Like (ABC)16 = A\*162 + B\*161 + C\*160  (ABC)16 = 10\*162 + 11\*161 + 12\*160 => 2748  (ABC)16 = 10\*256 + 11\*16 + 12\*1 => 2748  (ABC)16 = 2560 + 176 + 12 => 2748  print("For 0xA, int is:", int('0xA', 16))  For 0xA, int is: 10  print("For 0xABC, int is:", int('0xABC', 16))  For 0xABC, int is: 2748 |
| long(x [,base] ) | Converts x to a long integer. base specifies the base if x is a string. As Python 3 treats all integers as long integer this long() has been removed. (Python 3's int is the same as Python 2's long). there is no distinction between long and int in python3. (long does not exist...)  long('234')  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  NameError: name 'long' is not defined  int(float('234.89'))  234 |
| float(x) | Converts x to a floating-point number.  float('12345')  12345.0  float('12345.55')  12345.55 |
| complex(real [,imag]) | Creates a complex number.  complex(3,5)  (3+5j) |
| str(x) | Converts object x to a string representation.  str(12)  '12'  str(99.99)  '99.99'  str('abc')  'abc'  user = "Dev"  lines = 50  print("Congratulations, " + user + "! You just wrote " + str(lines) + " lines of code.")  Congratulations, Dev! You just wrote 50 lines of code.  lines\_yesterday = "50"  lines\_today = "108"  lines\_more = int(lines\_today) - int(lines\_yesterday)  print(lines\_more)  58 |
| repr(x) | * Converts object x to an expression string. repr() [compute the “official” string representation of an object](http://docs.python.org/reference/datamodel.html#object.__repr__)(a representation that has all information about the object) and str() is used to [compute the “informal” string representation of an object](http://docs.python.org/reference/datamodel.html#object.__str__) (a representation that is useful for printing the object).   import datetime  today = datetime.datetime.now()    # Prints readable format for date-time object  print(str(today))    # prints the official format of date-time object  print(repr(today))  Output:  2016-02-22 19:32:04.078030  datetime.datetime(2016, 2, 22, 19, 32, 4, 78030) |
| eval(str) | Evaluates a string and returns an object.  x = 1  eval('x+1')  2 |
| tuple(s) | Converts s to a tuple.  In Python:   * a [**tuple**](https://www.digitalocean.com/community/tutorials/understanding-tuples-in-python-3) is an immutable ordered sequence of elements contained within parentheses ( ).   sea\_creatures = **[**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**]**  print(type(sea\_creatures))  <class 'list'>  x = tuple(sea\_creatures);  print(x, '\t', type(x));  **(**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**)** <class 'tuple'> |
| list(s) | Converts s to a list.  In Python:   * a [**list**](https://www.digitalocean.com/community/tutorials/understanding-lists-in-python-3) is a mutable ordered sequence of elements that is contained within square brackets [ ].   sea\_creatures = **(**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**)**  print(type(sea\_creatures));  <class 'tuple'>  x = list(sea\_creatures);  print(x, '\t', type(x));  **[**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**]** <class 'list'> |
| set(s) | Converts s to a set prior to 2.6  lang = ("Perl", "Python", "Java")  print(type(lang));  <class 'tuple'>  x = set(lang)  print(x, '\t', type(x));  {'Python', 'Java', 'Perl'} <class 'set'>  We can define sets (since Python2.6) without using the built-in set function. We can use curly braces instead:  x = {"cheap","expensive","inexpensive","economical"}  x  {'economical', 'inexpensive', 'expensive', 'cheap'}  print(x,'\t',type(x))  {'economical', 'inexpensive', 'expensive', 'cheap'} <class 'set'> |
| Immutable Sets | Sets are implemented in a way, which doesn't allow mutable objects. The following example demonstrates that we can include tuples as elements in the sets but cannot include multiple lists as elements (**single list is allowed**):  x = {"cheap","expensive","inexpensive","economical"}  cities = set((("Python","Perl"), ("Paris", "Berlin", "London")))  # Here we used tuples.  print(cities)  {('Paris', 'Berlin', 'London'), ('Python', 'Perl')}  # Here we are going to use lists. Single list is allowed as in frozenset.  cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: unhashable type: 'list'  cities = set(["Paris", "Berlin", "London"],["Delhi","Mumbai"])  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  **TypeError: set expected at most 1 arguments, got 2** |
| frozenset(s) | Converts s to a frozen set.  Though sets can't contain mutable objects (list), multiple lists are not allowed single list is allowed, sets are mutable:  **# This is mutable set because it have only 1 mutable object (list).**  cities = set(["Paris", "Berlin", "London"])  cities.add("Rome")  cities  {'London', 'Berlin', 'Rome', 'Paris'}  Frozensets are like sets except that they cannot be changed, i.e. they are immutable:  cities = frozenset(["Paris", "Berlin", "London"])  cities  frozenset({'London', 'Berlin', 'Paris'})  cities.add("Rome")  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  AttributeError: 'frozenset' object has no attribute 'add' |
| Set Operations | add(element)  A method which adds an element, which has to be immutable, to a set.  colours = {"red","green"}  colours.add("yellow")  colours  {'green', 'yellow', 'red'}  colours.add(["black","white"]) # List cannot be a added.  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: unhashable type: 'list'  Of course, an element will only be added, if it is not already contained in the set. If it is already contained, the method call has no effect.  colours.add("yellow")  colours  {'yellow', 'green', 'red'} clear colours.clear()  colours  set() copy cities = {"Delhi","Mumbai","Chennai","Kolkata"};  metro = cities.copy();  metro  {'Chennai', 'Kolkata', 'Mumbai', 'Delhi'}  metro1 = cities;  metro1  {'Chennai', 'Kolkata', 'Mumbai', 'Delhi'} difference() This method returns the difference of two or more sets as a new set.  x = {"a","b","c","d","e"}  y = {"b","c"}  z = {"c","d"}  x.difference(y)  {'a', 'e', 'd'}  x  {'a', 'c', 'e', 'd', 'b'}  x.difference(y).difference(z)  {'a', 'e'}    Instead of using the method difference, we can use the operator "-":  x - y  {'a', 'e', 'd'}  x - y - z  {'a', 'e'} difference\_update() The method difference\_update removes all elements of another set from this set. x.difference\_update(y) is the same as "x = x - y"  x = {"a","b","c","d","e"}  y = {"b","c"}  x.difference\_update(y)  x  {'a', 'e', 'd'}    x = {"a","b","c","d","e"}  y = {"b","c"}  x = x - y  x  {'a', 'e', 'd'} discard(el) An element el will be removed from the set, if it is contained in the set. If el is not a member of the set, nothing will be done.  x = {"a","b","c","d","e"}  x.discard("a")  x  {'c', 'b', 'e', 'd'}  x.discard("z")  x  {'c', 'b', 'e', 'd'}    x["b"]  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'set' object is not subscriptable  del x["b"]  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'set' object does not support item deletion remove(el) works like discard(), but if el is not a member of the set, a KeyError will be raised.  x = {"a","b","c","d","e"}  x.remove("a")  x  {'c', 'b', 'e', 'd'}  x.remove("z")  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  KeyError: 'z' union(s) This method returns the union of two sets as a new set, i.e. all elements that are in either set.  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x.union(y)  {'d', 'a', 'g', 'c', 'f', 'b', 'e'}  This can be abbreviated with the pipe operator "|":  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x | y  {'d', 'a', 'g', 'c', 'f', 'b', 'e'} intersection(s) Returns the intersection of the instance set and the set s as a new set. In other words: A set with all the elements which are contained in both sets is returned.  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x.intersection(y)  {'c', 'e', 'd'}    This can be abbreviated with the ampersand operator "&":  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x & y  {'c', 'e', 'd'} isdisjoint() This method returns True if two sets have a null intersection.  x = {"a","b","c"}  y = {"c","d","e"}  x.isdisjoint(y)  False    x = {"a","b","c"}  y = {"d","e","f"}  x.isdisjoint(y)  True   issubset() x.issubset(y) returns True, if x is a subset of y. "<=" is an abbreviation for "Subset of" and ">=" for "superset of" and "<" is used to check if a set is a proper subset of a set.  x = {"a","b","c","d","e"}  y = {"c","d"}  x.issubset(y)  False  y.issubset(x)  True  x < y  False  y < x # y is a proper subset of x  True  x < x # a set can never be a proper subset of oneself.  False  x <= x  True issuperset() x.issuperset(y) returns True, if x is a superset of y. ">=" is an abbreviation for "issuperset of" ">" is used to check if a set is a proper superset of a set.  x = {"a","b","c","d","e"}  y = {"c","d"}  x.issuperset(y)  True  x > y  True  x >= y  True  x >= x  True  x > x  False  x.issuperset(x)  True pop() pop() removes and returns an arbitrary set element. The method raises a KeyError if the set is empty  x = {"a","b","c","d","e"}  x.pop()  'a'  x.pop()  'c'  x.pop()  'e'  x.pop()  'd'  x.pop()  'b'  x.pop()  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  KeyError: 'pop from an empty set' |
| dict(d) | Creates a dictionary. d must be a sequence of (key,value) tuples.  state\_population = {"UP":199581477, "MH":112372972, "BR":103804637, "WB": 91347736, "AP": 84665533, "MP":72597565, "TN":72138958, "RJ":68621012, "KA":61130704, "GJ":60383628, "OD":41947358};  print(type(state\_population))  <class 'dict'>  state\_population["UP"]  199581477  state\_population["MH"]  112372972  state\_population["Punjab"] = 27704236;  # Add new element  state\_population["Punjab"] = 27704236;  state\_population  {'UP': 199581477, 'MH': 112372972, 'BR': 103804637, 'WB': 91347736, 'AP': 84665533, 'MP': 72597565, 'TN': 72138958, 'RJ': 68621012, 'KA': 61130704, 'GJ': 60383628, 'OD': 41947358, 'Punjab': 27704236}  # Convert tuple to dictionary.  t = (('a','Apple'), ('b','Ball'), ('c', 'Cat'), ('d', 'Dog'));  print(t, type(t))  (('a', 'Apple'), ('b', 'Ball'), ('c', 'Cat'), ('d', 'Dog')) <class 'tuple'>  x = dict(t);  print(x, type(x))  {'a': 'Apple', 'b': 'Ball', 'c': 'Cat', 'd': 'Dog'} **<class 'dict'>**  y = dict((x, y) for x, y in t);  print(y, type(y))  {'a': 'Apple', 'b': 'Ball', 'c': 'Cat', 'd': 'Dog'} <class 'dict'>  y['a']  'Apple'  y['d']  'Dog'  z = dict(map(reversed, t));  print(z, type(z))  {'Apple': 'a', 'Ball': 'b', 'Cat': 'c', 'Dog': 'd'} <class 'dict'>  # Convert list to dictionary.  x=[('hi','goodbye')]  print(x,type(x));  [('hi', 'goodbye')] <class 'list'>  y = dict(x);  print(y,type(y));  {'hi': 'goodbye'} <class 'dict'> |

Character Values in ASCII



|  |  |
| --- | --- |
| chr(x) | Converts an integer to a character.  The chr() method takes a single parameter, an integer.  The valid range of the integer is from 0 through 1,114,111.  print(chr(97));  print(chr(65));  print(chr(90));  print(chr(1119));  print(chr(63))  a  A  Z  џ  ? |
| unichr(x) | Converts an integer to a Unicode character.  Can't use unichr in Python 3.1  print(unichr(1119));  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  NameError: name 'unichr' is not defined |
| ord(x) | The ord() method returns an integer representing Unicode code point for the given Unicode character.  The ord() method is the inverse of chr().  The ord(x) method takes a single parameter:  x - character string of length 1 whose Unicode code point is to be found.  print(ord('1'))  49  print(ord('5'))  53  print(ord('A'))  65  print(ord('Z'))  90 |
| hex(x) | The hex() function converts an integer number to the corresponding hexadecimal string.  The hex() function takes a single argument.  print(hex(110));  print(hex(111));  print(hex(112));  print(hex(113));  print(hex(114));  print(hex(115));  0x6e  0x6f  0x70  0x71  0x72  0x73  Hexadecimal representation of a float.  print(float.hex(2.5));  0x1.4000000000000p+1  print(float.hex(5.5));  0x1.6000000000000p+2 |
| oct(x) | Converts an integer to an octal string.  print(oct(555));  0o1053  print(oct(555.55));  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'float' object cannot be interpreted as an integer |

# Python Basic Operators

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

## **Types of Operator**

Python language supports the following types of operators.

* Arithmetic Operators
* Comparison (Relational) Operators
* Assignment Operators
* Logical Operators
* Bitwise Operators
* Membership Operators
* Identity Operators

## **Python Arithmetic Operators**

Assume variable a holds 10 and variable b holds 20, then −

a = 10

b = 20

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + Addition | Adds values on either side of the operator. | a + b = 30 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -10 |
| \* Multiplication | Multiplies values on either side of the operator | a \* b = 200 |
| / Division | Divides left hand operand by right hand operand | b / a = 2 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 0 |
| \*\* Exponent | Performs exponential (power) calculation on operators | a\*\*b =10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity): | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0 |

## **Python Comparison Operators**

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | If the values of two operands are equal, then the condition becomes true. | (a == b) is not true. |
| != | If values of two operands are not equal, then condition becomes true. |  |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. | (a > b) is not true. |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. | (a < b) is true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | (a >= b) is not true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

## **Python Assignment Operators**

Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assigns values from right side operands to left side operand | c = a + b assigns value of a + b into c |
| += Add AND | It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |
| -= Subtract AND | It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*= Multiply AND | It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /= Divide AND | It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / a |
| %= Modulus AND | It takes modulus using two operands and assign the result to left operand (return remainder) | c %= a is equivalent to c = c % a |
| \*\*= Exponent AND | Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a is equivalent to c = c \*\* a |
| //= Floor Division | It performs floor division on operators and assign value to the left operand (return quotient) | c //= a is equivalent to c = c // a  c=58;  c %=a;  print(c);  8  c=58;  c //=a;  print(c);  5 |

## **Python Logical Operators**

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and Logical AND | If both the operands are true then condition becomes true. | (a and b) is true. |
| or Logical OR | If any of the two operands are non-zero then condition becomes true. | (a or b) is true. |
| not Logical NOT | Used to reverse the logical state of its operand. | Not(a and b) is false. |

Example.

x = True;

y = False;

print('x and y is ',x and y);

x and y is False

print('x or y is ',x or y);

x or y is True

print('not x is ',not x);

not x is False

## **Python Bitwise Operators**

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100 => 60

b = 0000 1101 => 13

-----------------

a&b = 0000 1100 => 12

a|b = 0011 1101 => 61

a^b = 0011 0001 => 49

~a  = 1100 0011 => -61 (-64+3)

There are following Bitwise operators supported by Python language

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & Binary AND | Operator copies a bit to the result if it exists in both operands | (a & b) (means 0000 1100)  a = 60;  b = 13;  x = a&b;  print(x);  12 |
| | Binary OR | It copies a bit if it exists in either operand. | (a | b) = 61 (means 0011 1101)  x = a|b;  print(x);  61 |
| ^ Binary XOR (exclusive OR) | Write the bit as 1 if it is set in one operand otherwise 0 if matches in both operand. | (a ^ b) = 49 (means 0011 0001)  x = a^b;  print(x);  49 |
| ~ Binary Ones Complement | It is unary and has the effect of 'flipping' bits. | (~a ) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.  x = ~a;  print(x);  -61 |
| << Binary Left Shift | The left operands value is moved left by the number of bits specified by the right operand. | a << 2 = 240 (means 1111 0000)  x = a << 2 ;  print(x);  240 |
| >> Binary Right Shift | The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 = 15 (means 0000 1111)  x = a >> 2 ;  print(x);  15 |

## **Python Membership Operators**

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Evaluates to true if it finds a variable in the specified sequence and false otherwise. | x in y, here in results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. | x not in y, here not in results in a 1 if x is not a member of sequence y. |

### **Example**

a = 10

b = 20

l\_num = [1, 2, 3, 4, 5 ];

if ( a in l\_num ):

print("a is available in the given list");

else:

print("a is not available in the given list");

if ( b not in l\_num ):

print("b is not available in the given list");

else:

print("b is available in the given list");

x = 2

if ( x in l\_num ):

print("x is available in the given list");

else:

print("x is not available in the given list");

When you execute the above program it produces the following result:

a is not available in the given list

b is not available in the given list

x is available in the given list

print( a in l\_num );

False

print( b in l\_num );

False

x = 2

print( x in l\_num );

True

## **Python Identity Operators**

Identity operators are used to verify if two variables point to the same memory location or not. Identity operators are of two types:

(1) is

(2) is not.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | x is y, here **is** results in 1 if id(x) equals id(y). |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | x is not y, here **is not** results in 1 if id(x) is not equal to id(y). |

### **Example**

a = 20

b = 20

if ( a is b ):

print("Line 1 - a and b have same identity")

else:

print("Line 1 - a and b do not have same identity")

if ( id(a) == id(b) ):

print("Line 2 - a and b have same identity")

else:

print("Line 2 - a and b do not have same identity")

b = 30

if ( a is b ):

print("Line 3 - a and b have same identity")

else:

print("Line 3 - a and b do not have same identity")

if ( a is not b ):

print("Line 4 - a and b do not have same identity")

else:

print("Line 4 - a and b have same identity")

When you execute the above program it produces the following result −

Line 1 - a and b have same identity

Line 2 - a and b have same identity

Line 3 - a and b do not have same identity

Line 4 - a and b do not have same identity

a = 20

b = 20

id(a)

1539688720

id(b)

1539688720

a = 555

b = 555

id(a)

1390899932912

id(b)

1390901046000

a = 1000

b = 1000

id(a)

1390901046096

id(b)

1390899932912

a = 20

b = 20

id(a)

1539688720

id(b)

1539688720

id(1000)

1390899932912

b = 1000

id(b)

1390899930224

## **Python Operators Precedence**

The following table lists all operators from highest precedence to lowest.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Complement, unary plus and minus (method names for the last two are +@ and -@)  a = 20  b = 1000  -a  -20  +a  20  -b  -1000  +b  1000  ~a  -21  ~b  -1001 |
| \* / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise 'AND' |  |
| ^ | | Bitwise exclusive `OR' and regular `OR' |  |
| <= < > >= | Comparison operators |  |
| <> == != | Equality operators |  |
| = %= /= //= -= += \*= \*\*= | Assignment operators |  |
| is, is not | Identity operators |  |
| in, not in | Membership operators |  |
| not, or, and | Logical operators |  |

Operator precedence affects how an expression is evaluated.

For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so it first multiplies 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

### **Example**

a = 20

b = 10

c = 15

d = 5

e = 0

e = (a + b) \* c / d #( 30 \* 15 ) / 5

print("Value of (a + b) \* c / d is ", e)

e = ((a + b) \* c) / d # (30 \* 15 ) / 5

print("Value of ((a + b) \* c) / d is ", e)

e = (a + b) \* (c / d); # (30) \* (15/5)

print("Value of (a + b) \* (c / d) is ", e)

e = a + (b \* c) / d; # 20 + (150/5)

print("Value of a + (b \* c) / d is ", e)

When you execute the above program, it produces the following result −

Value of (a + b) \* c / d is 90

Value of ((a + b) \* c) / d is 90

Value of (a + b) \* (c / d) is 90

Value of a + (b \* c) / d is 50

**Does Python have a ternary conditional operator?**

**Yes, it was added in version 2.5.**

The syntax is:

a if condition else b

First condition is evaluated, then either a or b is returned based on the Boolean value of condition

If condition evaluates to True a is returned, else b is returned.

'true' if True else 'false'

'true'

'false' if True else 'false'

'false'

'true' if False else 'false'

'false'

'true' if 1 else 'false'

'true'

'true' if 0 else 'false'

'false'

# Python Decision Making

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages −



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| **if statements** | An **if statement** consists of a boolean expression followed by one or more statements. |
| **if...else statements** | An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE. |
| **nested if statements** | You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).  The **elif** Statement  The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.  Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an if. **Syntax:** The syntax of the nested *if...elif...else* construct may be:  if expression1:  statement(s)  if expression2:  statement(s)  elif expression3:  statement(s)  else:  statement(s)  elif expression4:  statement(s)  else:  statement(s) |

var1 = 5555

if var1:

print("1 – Condition evaluated true");

print(var1);

var2 = 0

if var2:

print("2 - Condition evaluated false");

print(var2);

else:

print("Good bye!");

v\_marks = 85

if (v\_marks<50) :

print("He is fail.");

print("He should work hard.");

elif (v\_marks>=50 and v\_marks<60) :

print("He stood 2nd."); print("He need improvments.");

elif (v\_marks>=60 and v\_marks<80) :

print("He stood 1st.");

else :

print("Outstanding.");

print("Well done.");

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

1 - Condition evaluated true

5555

Good bye!

Outstanding.

Well done.

**# Another Example**

var = 100

if var < 200:

print("Expression value is less than 200");

if var == 150:

print("Which is 150");

elif var == 100:

print("Which is 100");

elif var == 50:

print("Which is 50");

elif var < 50:

print("Expression value is less than 50");

else:

print("Could not find true expression");

print("Good bye!"); # This statement is not the of any if.

**# Output**

Expression value is less than 200

Which is 100

>>> print("Good bye!"); # This statement is not the of any if.

Good bye!

## **Single Statement Suites**

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause −

var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"

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

Value of expression is 100

Good bye!

# Python Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements. There are two types of loops in Python, **for** and **while**.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [**while loop**](https://www.tutorialspoint.com/python/python_while_loop.htm) | Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| [**for loop**](https://www.tutorialspoint.com/python/python_for_loop.htm) | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |

**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 beforehand, the number of times to iterate.

## Syntax

while test\_expression:

Body of while

**In Python, the body of the while loop is determined through indentation.**

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.

# Prints out 1,2,3,4,5

count = 1

while count <= 5:

print(count);

count += 1;

print("Good Bye While Loop");

# Output

1

2

3

4

5

Good Bye While Loop

In Python, break and continue statements can alter the flow of a normal loop. The break statement terminates the loop containing it and pass the control of the program to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

# Prints out 1,2,3,4,5

count = 1

while True:

print(count)

count += 1

if count > 5:

break;

print("Good Bye While Loop");

# Output

1

2

3

4

5

Good Bye While Loop

# Prints out 1,2,3,4,6,7,8,9,10

count = 1

while (count<=10):

if (count==5):

count=count+1

continue

else:

print(count)

count=count+1

print("Good Bye While Loop")

# Output

1

2

3

4

6

7

8

9

10

Good Bye While Loop

**We use "else" clause in loops.**

When the loop condition of "for" or "while" statement fails then code part in "else" is executed. If break statement is executed inside for loop then the "else" part is skipped. Note that "else" part is executed even if there is a continue statement.

# While loop with else.

# Print out 1,2,3,4,5 and then it prints "count value reached 5"

count=1

while(count<5):

print(count)

count +=1

else:

print("count value reached %d" %(count))

# Output

1

2

3

4

count value reached 5

**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.

## 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.

for x in (1,2,3,4,5):

print(x)

# Output

1

2

3

4

5

for x in [1,2,3,4,5]:

print(x)

# Output

1

2

3

4

5

for x in ['a','b','c','d','e']:

print(x)

# Output

a

b

c

d

e

for x in range(5):

print(x)

# Output

0

1

2

3

4

for x in range(3, 6):

print(x)

# Output

3

4

5

# range in increment order of 2

for x in range(3, 10, 2):

print(x)

# Output

3

5

7

9

# Prints out only odd numbers - 1,3,5,7,9

for x in range(10):

# Check if x is even

if x % 2 == 0:

**continue;**

print(x);

# Output

1

3

5

7

9

# Prints out 1,2,3,4

for i in range(1, 10):

if(i%5==0):

**break**

print(i)

else:

print("this is not printed because for loop is terminated because of break but not due to fail in condition")

# Output

1

2

3

4

# Program to find the sum of all numbers stored in a list

# List of numbers

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

sum = 0

for val in numbers:

sum = sum+val;

else:

print("The sum is", sum);

# Output

The sum is 48

**Python Pass Statement**

It is used as a placeholder for future implementation of functions, loops, etc.

In Python programming, pass is a **null** statement. The difference between a comment and pass statement is that the interpreter ignores a comment entirely, pass is not ignored it is executed. However, nothing happens when pass is executed. It results into no operation (NOP).

## Syntax of pass

pass

We generally use it as a placeholder.

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 complain. So, we use the pass statement to construct a body that does nothing.

# pass is just a placeholder for

# functionality to be added later.

sequence = [1,2,3,4,5]

for val in sequence:

pass

# Python Numbers

# Integer Objects

All integers are implemented as “long” integer objects of **arbitrary size**.

<https://docs.python.org/3/c-api/long.html>

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a **new allocated object.**

Number objects are created when you assign a value to them. For example −

var1 = 1

var2 = 10

id(var1)

1539688112

id(var2)

1539688400

var1 = 100

var2 = 200

**NOTE: Here object is reallocated.**

id(var1)

1539691280

id(var2)

1539694480

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is −

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the **del** statement. For example:

del var

del var\_a, var\_b

Python supports four different numerical types −

* **int (signed integers)**: They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
* **long (long integers )**: Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L. No longer supported in Version 3.x.
* **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).
* **complex (complex numbers)** : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

**What is the range of int in python?**

**Python** has arbitrary precision integers so there **is no true** fixed maximum. You're only limited by available memory. sys.maxint does not even exist in **Python** 3, since **int** and long were unified into a single arbitrary precision **int** type.

v\_num = 9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999;

print(v\_num);

9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999

### **Examples**

Here are some examples of numbers

|  |  |  |  |
| --- | --- | --- | --- |
| **int** | **long** | **float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9 | 9.322e-36j |
| 080 | 0xDEFABCECBDAECBFBAEL | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

## **Number Type Conversion**

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

* Type **int(x)** to convert x to a plain integer.
* Type **float(x)** to convert x to a floating-point number.
* Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
* Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

|  |
| --- |
| float(2);  2.0  float(5.55);  5.55  complex(5)  (5+0j)  complex(5,2)  (5+2j) |

## **Mathematical Functions**

Python includes following functions that perform mathematical calculations.

|  |  |
| --- | --- |
| **Function** | **Returns ( description )** |
| **abs(x)** | The absolute value of x: the (positive) distance between x and zero.  abs(-5);  5 |
| **ceil(x)** | The ceiling of x: the smallest integer not less than x  import math  math.ceil(4.4); 5  math.ceil(4.7); 5 |
| **cmp(x, y)** | -1 if x < y, 0 if x == y, or 1 if x > y Python 3 doesn't have a cmp function. |
| **exp(x)** | The exponential of x: ex  math.exp(1e-5)  1.00001000005 |
| **fabs(x)** | The absolute value of x.  math.fabs(-5.5)  5.5  abs(-5.5)  5.5 |
| **floor(x)** | The floor of x: the largest integer not greater than x  math.floor(4.4); 4  math.floor(4.8); 4 |
| **log(x)** | The natural logarithm of x, for x> 0  math.log(5);  1.6094379124341003 |
| **log10(x)** | The base-10 logarithm of x for x> 0 .  math.log10(5);  0.6989700043360189 |
| **max(x1, x2,...)** | The largest of its arguments: the value closest to positive infinity  max(10,20,66,45,87,55,23,58)  87 |
| **min(x1, x2,...)** | The smallest of its arguments: the value closest to negative infinity  min(10,20,66,45,87,55,23,58)  10 |
| **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.  math.modf(5);  (0.0, 5.0)  math.modf(1,5);  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: modf() takes exactly one argument (2 given) |
| **pow(x, y)** | The value of x\*\*y.  math.pow(2,5);  32.0 |
| **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.  round(7676567.6556757,3);  7676567.656  round(7676567.6544757,3);  7676567.654 |
| **sqrt(x)** | The square root of x for x > 0  math.sqrt(16)  4.0 |

## **Random Number Functions**

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

|  |  |
| --- | --- |
| **Function** | **Description** |
| **choice(seq)** | A random item from a list, tuple, or string.  import random;  x=**[**1, 2, 3, 5, 9**]**;  print(type(x))  <class 'list'>  x=**(**1, 2, 3, 5, 9**)**;  print(type(x))  <class 'tuple'>  random.choice([1, 2, 3, 5, 9]);  5  random.choice([1, 2, 3, 5, 9]);  1  random.choice((1, 2, 3, 5, 9));  5  random.choice((1, 2, 3, 5, 9));  2  random.choice('India is my country.');  'd'  random.choice('India is my country.');  'm' |
| **randrange ([start,] stop [,step])** | A randomly selected element from range(start, stop, step)  random.randrange(100, 1000, 2);  334  random.randrange(100, 1000, 2);  434  random.randrange(100, 1000, 2);  624 |
| **random()** | A random float r, such that 0 is less than or equal to r and r is less than 1  print("random() : ", random.random());  random() : 0.0623392041549814  print("random() : ", random.random());  random() : 0.9609234212687972 |
| **seed([x])** | Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.4288890546751146  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.20609823213950174  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.81332125135732  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.8235888725334455 |
| **shuffle(lst)** | Randomizes the items of a list in place. Returns None.  list = [20, 16, 10, 5];  random.shuffle(list);  print(list);  [10, 5, 20, 16]  random.shuffle(list);  print(list);  [5, 16, 10, 20]  random.shuffle(list);  print(list);  [20, 16, 5, 10] |
| **uniform(x, y)** | A random float r, such that x is less than or equal to r and r is less than y  print("Random Float uniform(1, 10) : ", random.uniform(1, 10))  print("Random Float uniform(1, 10) : ", random.uniform(1, 10))  Random Float uniform(1, 10) : 6.418091089533576  Random Float uniform(1, 10) : 8.37482423567478  print("Random Float uniform(7, 10) : ", random.uniform(7, 10))  print("Random Float uniform(7, 10) : ", random.uniform(7, 10))  Random Float uniform(7, 10) : 8.370493453174916  Random Float uniform(7, 10) : 9.057584456423784 |

## **Mathematical Constants**

The module also defines two mathematical constants −

|  |  |
| --- | --- |
| **Constants** | **Description** |
| pi | The mathematical constant pi.  import math  math.pi  3.141592653589793 |
| e | The mathematical constant e.  math.e  2.718281828459045 |

# Python Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

var1 = 'Hello World!'

var2 = "Python Programming"

## **Accessing Values in Strings**

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

#!/usr/bin/python

var1 = 'Hello World!'

var2 = "Python Programming"

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

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

print("var2[:10]: ", var2[:10])

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

var1[0]: H

var2[1:5]: ytho

var2[:10]: Python Pro

## **Updating Strings**

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. You cannot update a character of a string. For example −

var1 = 'Hello World!'

print(var1);

var1 = 'Hello World, Welcome to Python!'

print(var1);

var1 = 'ABC';

var1[2] = 'D'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

## **Escape Characters**

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

|  |  |  |
| --- | --- | --- |
| **Backslash notation** | **Hexadecimal character** | **Description** |
| \a | 0x07 | Bell or alert sound generate from your laptop  print('Calling Bell!', '\a');  Calling Bell! |
| \b | 0x08 | Backspace  print('Calling Bell!'+' open the door.');  Calling Bell! open the door.  print('Calling Bell!'+'\b'+' open the door.');  Calling Bell open the door.  print('Calling Bell!'+'\b\b\b\b\b'+' open the door.');  Calling open the door. |
| \e | 0x1b | Escape |
| \f | 0x0c | Formfeed  print('Calling Bell!','\f','Open the door'); |
| \n | 0x0a | Newline  print('Calling Bell!','\n','\bOpen the door');  Calling Bell!  Open the door |
| \nnn |  | Octal notation, where n is in the range 0.7 |
| \r | 0x0d | Carriage return  print('Calling Bell!','\r','Open the door', '\r', 'Guests are welcome.');  Guest are welcome. |
| \s | 0x20 | Space |
| \t | 0x09 | Tab  print('Hello''\t''World');  Hello World |
| \v | 0x0b | Vertical tab |
| \x |  | Character x |
| \xnn |  | Hexadecimal notation, where n is in the range 0.9, a.f, or A.F  print('Hello''\xad''World');  Hello­World  print('Hello''\xae''World');  Hello®World |

## **String Special Operators**

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Concatenation - Adds values on either side of the operator | a + b will give HelloPython  a = 'Hello'  b = 'Python'  a+b  'HelloPython' |
| \* | Repetition - Creates new strings, concatenating multiple copies of the same string | a\*2 will give –HelloHello  a\*3  'HelloHelloHello' |
| [] | Slice - Gives the character from the given index | a[1] will give e |
| [ : ] | Range Slice - Gives the characters from the given range | a[1:4] will give ell |
| in | Membership - Returns true if a character exists in the given string | H in a will give 1 (True)  'l' in var1  True  'w' in var1  False |
| not in | Membership - Returns true if a character does not exist in the given string | M not in a will give 1 (True)  'w' not in var1  True |
| r/R | Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark. | print r'\n' prints \n and print R'\n'prints \n  print('\n')  print(r'\n')  \n |
| % | Format - Performs String formatting | v\_num = 100;  print('v\_num is: %d' %(v\_num));  v\_num is: 100 |

## **String Formatting Operator**

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

#!/usr/bin/python

print("Heavyweight boxer %s, total win %d/58" % ('Michael Gerard Tyson', 50));

Heavyweight boxer Michael Gerard Tyson, total win 50/58

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

|  |  |
| --- | --- |
| **Format Symbol** | **Conversion** |
| %c | Character  fname='Hari'  lname='Yadav'  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 65 kg! |
| %s | string conversion via str() prior to formatting  print("My name is %s" %("Michael Gerard Tyson"));  My name is Michael Gerard Tyson  print("My name is {} {} and surname {} ".format('Michael', 'Gerard', 'Tyson'));  My name is Michael Gerard and surname Tyson  print('%s' % ('test',))  test  print('%10s' % ('test',))  test |
| %i | signed decimal integer |
| %d | signed decimal integer  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 65 kg!  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -65 kg! |
| %u | unsigned decimal integer  Obsolete type – it is identical to ‘d’. See PEP 237. |
| %o | octal integer  print("My name initial is %c%c and weight is %o kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -101 kg!  print("My name initial is %c%c and weight is %o kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 101 kg! |
| %x | hexadecimal integer (lowercase letters)  print("My name initial is %c%c and weight is %x kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 41 kg! |
| %X | hexadecimal integer (UPPERcase letters) |
| %e | exponential notation (with lowercase 'e')  print("My name initial is %c%c and weight is %e kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 6.500000e+01 kg! |
| %E | exponential notation (with UPPERcase 'E')  print("My name initial is %c%c and weight is %E kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -6.500000E+01 kg! |
| %f | floating point real number  print("My name initial is %c%c and weight is %f kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 65.000000 kg! |

Other supported symbols and functionality are listed in the following table −

|  |  |
| --- | --- |
| **Symbol** | **Functionality** |
| \* | argument specifies width or precision |
| - | left justification  print('%10d' %(837.64)); # right justified.  837  print('%-10d' %(837.64)); # left justified.  837 |
| + | display the sign  print('%010d' %(+837.64));  0000000837  print('%+010d' %(837.64));  +000000837  print('%010d' %(-837.64));  -000000837 |
| <sp> | leave a blank space before a positive number  print('% 010d' %(837.64));  000000837 |
| # | add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.  print("%#x" % 17 )  0x11  print("%#x" % 18 )  0x12  print("%x" % 17 )  11  print("%x" % 18 )  12 |
| 0 | pad from left with zeros (instead of spaces)  print('%10d' %(837.64));  837  print('%010d' %(837.64));  0000000837 |
| f | Represent the value in floating point.  print('%f' %(837.64759678));  837.647597  print('%.1f' %(837.64759678));  837.6  print('%.2f' %(837.64759678));  837.65  print('%.4f' %(837.64759678));  837.6476  print('%010.2f' %(837.64759678));  0000837.65  print('%010.4f' %(837.64759678));  00837.6476 |
| % | '%%' leaves you with a single literal '%'  print('I have scored %d%%' %(85));  I have scored 85%  print('I have scored %d%% %d%%' %(85,90));  I have scored 85% 90% |

## **Triple Quotes**

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double**quotes.

para\_str = """this is a long string that is made up of

several lines and non-printable characters such as

TAB ( \t ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [ \n ], or just a NEWLINE within

the variable assignment will also show up.

"""

print(para\_str)

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n) –

this is a long string that is made up of

several lines and non-printable characters such as

TAB ( ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [

], or just a NEWLINE within

the variable assignment will also show up.

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it −

print('C:\\nowhere')

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

C:\nowhere

Now let's make use of raw string. We would put expression in **r'expression'**as follows −

print(r'C:\\nowhere')

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

C:\\nowhere

## **Unicode String**

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world.

Var1 = 'Hello, world!';

Var2 = u'Hello, world!';

print(u'Hello, world!')

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

Hello, world!

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

## **unicode()** global function[#](http://www.diveintopython3.net/porting-code-to-python-3-with-2to3.html#unicode)

Python 2 had two global functions to coerce objects into strings: unicode() to coerce them into Unicode strings, and str() to coerce them into non-Unicode strings. Python 3 has only one string type so the str() function is all you need. (The unicode() function no longer exists.)

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|  | unicode(anything) | str(anything) |

## **Built-in String Methods**

Python includes the following built-in methods to manipulate strings −

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **capitalize()** Capitalizes first letter of string  print(str.capitalize('Michael Gerard Tyson'));  Michael gerard tyson |
| 2 | [**center(width, fillchar)**](https://www.tutorialspoint.com/python/string_center.htm)  Returns a space-padded string with the original string centered to a total of width columns.  vname = 'MICHAEL GERARD TYSON';  print(vname.center(30,'x'));  xxxxxMICHAEL GERARD TYSONxxxxx  print(vname.center(30,' '));  MICHAEL GERARD TYSON  print(vname.center(30,' x '));  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: The fill character must be exactly one character long |
| 3 | [**count(str, beg= 0,end=len(string))**](https://www.tutorialspoint.com/python/string_count.htm)  Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given. |
| 4 | [**decode(encoding='UTF-8',errors='strict')**](https://www.tutorialspoint.com/python/string_decode.htm)  Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. |
| 5 | [**encode(encoding='UTF-8',errors='strict')**](https://www.tutorialspoint.com/python/string_encode.htm)  Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'. |
| 6 | [**endswith(suffix, beg=0, end=len(string))**](https://www.tutorialspoint.com/python/string_endswith.htm) Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise. |
| 7 | [**expandtabs(tabsize=8)**](https://www.tutorialspoint.com/python/string_expandtabs.htm)  Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided. |
| 8 | [**find(str, beg=0 end=len(string))**](https://www.tutorialspoint.com/python/string_find.htm)  Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise. |
| 9 | [**index(str, beg=0, end=len(string))**](https://www.tutorialspoint.com/python/string_index.htm)  Same as find(), but raises an exception if str not found. |
| 10 | [**isalnum()**](https://www.tutorialspoint.com/python/string_isalnum.htm)  Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise. |
| 11 | [**isalpha()**](https://www.tutorialspoint.com/python/string_isalpha.htm)  Returns true if string has at least 1 character and all characters are alphabetic and false otherwise. |
| 12 | [**isdigit()**](https://www.tutorialspoint.com/python/string_isdigit.htm)  Returns true if string contains only digits and false otherwise. |
| 13 | [**islower()**](https://www.tutorialspoint.com/python/string_islower.htm)  Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise. |
| 14 | [**isnumeric()**](https://www.tutorialspoint.com/python/string_isnumeric.htm)  Returns true if a unicode string contains only numeric characters and false otherwise. |
| 15 | [**isspace()**](https://www.tutorialspoint.com/python/string_isspace.htm)  Returns true if string contains only whitespace characters and false otherwise. |
| 16 | [**istitle()**](https://www.tutorialspoint.com/python/string_istitle.htm)  Returns true if string is properly "titlecased" and false otherwise. |
| 17 | [**isupper()**](https://www.tutorialspoint.com/python/string_isupper.htm)  Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise. |
| 18 | [**join(seq)**](https://www.tutorialspoint.com/python/string_join.htm)  Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string. |
| 19 | [**len(string)**](https://www.tutorialspoint.com/python/string_len.htm)  Returns the length of the string |
| 20 | [**ljust(width[, fillchar])**](https://www.tutorialspoint.com/python/string_ljust.htm)  Returns a space-padded string with the original string left-justified to a total of width columns. |
| 21 | [**lower()**](https://www.tutorialspoint.com/python/string_lower.htm)  Converts all uppercase letters in string to lowercase.  print(str.lower('Michael Gerard Tyson'));  michael gerard tyson |
| 22 | [**lstrip()**](https://www.tutorialspoint.com/python/string_lstrip.htm)  Removes all leading whitespace in string. |
| 23 | [**maketrans()**](https://www.tutorialspoint.com/python/string_maketrans.htm)  Returns a translation table to be used in translate function. |
| 24 | [**max(str)**](https://www.tutorialspoint.com/python/string_max.htm)  Returns the max alphabetical character from the string str. |
| 25 | [**min(str)**](https://www.tutorialspoint.com/python/string_min.htm)  Returns the min alphabetical character from the string str. |
| 26 | [**replace(old, new [, max])**](https://www.tutorialspoint.com/python/string_replace.htm)  Replaces all occurrences of old in string with new or at most max occurrences if max given. |
| 27 | [**rfind(str, beg=0,end=len(string))**](https://www.tutorialspoint.com/python/string_rfind.htm)  Same as find(), but search backwards in string. |
| 28 | [**rindex( str, beg=0, end=len(string))**](https://www.tutorialspoint.com/python/string_rindex.htm)  Same as index(), but search backwards in string. |
| 29 | [**rjust(width,[, fillchar])**](https://www.tutorialspoint.com/python/string_rjust.htm)  Returns a space-padded string with the original string right-justified to a total of width columns. |
| 30 | [**rstrip()**](https://www.tutorialspoint.com/python/string_rstrip.htm)  Removes all trailing whitespace of string. |
| 31 | [**split(str="", num=string.count(str))**](https://www.tutorialspoint.com/python/string_split.htm)  Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given. |
| 32 | [**splitlines( num=string.count('\n'))**](https://www.tutorialspoint.com/python/string_splitlines.htm)  Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed. |
| 33 | [**startswith(str, beg=0,end=len(string))**](https://www.tutorialspoint.com/python/string_startswith.htm)  Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise. |
| 34 | [**strip([chars])**](https://www.tutorialspoint.com/python/string_strip.htm)  Performs both lstrip() and rstrip() on string |
| 35 | [**swapcase()**](https://www.tutorialspoint.com/python/string_swapcase.htm)  Inverts case for all letters in string. |
| 36 | [**title()**](https://www.tutorialspoint.com/python/string_title.htm)  Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase. |
| 37 | [**translate(table, deletechars="")**](https://www.tutorialspoint.com/python/string_translate.htm)  Translates string according to translation table str(256 chars), removing those in the del string. |
| 38 | **upper()** Converts lowercase letters in string to uppercase.  print(str.upper('Michael Gerard Tyson'));  MICHAEL GERARD TYSON |
| 39 | [**zfill (width)**](https://www.tutorialspoint.com/python/string_zfill.htm)  Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). |
| 40 | [**isdecimal()**](https://www.tutorialspoint.com/python/string_isdecimal.htm)  Returns true if a unicode string contains only decimal characters and false otherwise. |

# Python Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

## **Python Lists**

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

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

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

list2 = [1, 2, 3, 4, 5 ];

list3 = ["a", "b", "c", "d"]

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

## **Accessing Values in Lists**

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

#!/usr/bin/python

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**

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

#!/usr/bin/python

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]

**Note:** append() method is discussed in subsequent section.

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**

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

#!/usr/bin/python

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

print list1

del list1[2];

print "After deleting value at index 2 : "

print list1

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

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

After deleting value at index 2 :

['physics', 'chemistry', 2000]

**Note:** remove() method is discussed in subsequent section.

## **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 |

## **Indexing, Slicing, and Matrixes**

Because lists are sequences, indexing and slicing work the same way for lists 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 List Functions & Methods:**

Python includes the following list functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(list1, list2)**](https://www.tutorialspoint.com/python/list_cmp.htm)  Compares elements of both lists. |
| 2 | [**len(list)**](https://www.tutorialspoint.com/python/list_len.htm)  Gives the total length of the list. |
| 3 | [**max(list)**](https://www.tutorialspoint.com/python/list_max.htm)  Returns item from the list with max value. |
| 4 | [**min(list)**](https://www.tutorialspoint.com/python/list_min.htm)  Returns item from the list with min value. |
| 5 | [**list(seq)**](https://www.tutorialspoint.com/python/list_list.htm)  Converts a tuple into list. |

Python includes following list methods

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [**list.append(obj)**](https://www.tutorialspoint.com/python/list_append.htm)  Appends object obj to list |
| 2 | [**list.count(obj)**](https://www.tutorialspoint.com/python/list_count.htm)  Returns count of how many times obj occurs in list |
| 3 | [**list.extend(seq)**](https://www.tutorialspoint.com/python/list_extend.htm)  Appends the contents of seq to list |
| 4 | [**list.index(obj)**](https://www.tutorialspoint.com/python/list_index.htm)  Returns the lowest index in list that obj appears |
| 5 | [**list.insert(index, obj)**](https://www.tutorialspoint.com/python/list_insert.htm)  Inserts object obj into list at offset index |
| 6 | [**list.pop(obj=list[-1])**](https://www.tutorialspoint.com/python/list_pop.htm)  Removes and returns last object or obj from list |
| 7 | [**list.remove(obj)**](https://www.tutorialspoint.com/python/list_remove.htm)  Removes object obj from list |
| 8 | [**list.reverse()**](https://www.tutorialspoint.com/python/list_reverse.htm)  Reverses objects of list in place |
| 9 | [**list.sort([func])**](https://www.tutorialspoint.com/python/list_sort.htm)  Sorts objects of list, use compare func if given |

# Python Tuples

A tuple is a sequence of immutable Python objects. 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.

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 );

tup3 = "a", "b", "c", "d";

The empty tuple is written as two parentheses containing nothing −

tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value −

tup1 = (50,);

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

## **Accessing Values in Tuples:**

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

#!/usr/bin/python

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**

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

#!/usr/bin/python

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**

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example:

#!/usr/bin/python

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 |

## **No Enclosing Delimiters**

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz'

x, y = 1, 2;

print "Value of x , y : ", x,y

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

abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2

## **Built-in Tuple Functions**

Python includes the following tuple functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(tuple1, tuple2)**](https://www.tutorialspoint.com/python/tuple_cmp.htm)  Compares elements of both tuples. |
| 2 | [**len(tuple)**](https://www.tutorialspoint.com/python/tuple_len.htm)  Gives the total length of the tuple. |
| 3 | [**max(tuple)**](https://www.tutorialspoint.com/python/tuple_max.htm)  Returns item from the tuple with max value. |
| 4 | [**min(tuple)**](https://www.tutorialspoint.com/python/tuple_min.htm)  Returns item from the tuple with min value. |
| 5 | [**tuple(seq)**](https://www.tutorialspoint.com/python/tuple_tuple.htm)  Converts a list into tuple. |

# Python Dictionary

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. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

## **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 −

#!/usr/bin/python

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 −

#!/usr/bin/python

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**

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 −

#!/usr/bin/python

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**

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.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

#!/usr/bin/python

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

**Note:** del() method is discussed in subsequent section.

## **Properties of Dictionary Keys**

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

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 −

#!/usr/bin/python

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

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

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

dict['Name']: Manni

**(b)** 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:

#!/usr/bin/python

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

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

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

Traceback (most recent call last):

File "test.py", line 3, in <module>

dict = {['Name']: 'Zara', 'Age': 7};

TypeError: list objects are unhashable

## **Built-in Dictionary Functions & Methods −**

Python includes the following dictionary functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(dict1, dict2)**](https://www.tutorialspoint.com/python/dictionary_cmp.htm)  Compares elements of both dict. |
| 2 | [**len(dict)**](https://www.tutorialspoint.com/python/dictionary_len.htm)  Gives the total length of the dictionary. This would be equal to the number of items in the dictionary. |
| 3 | [**str(dict)**](https://www.tutorialspoint.com/python/dictionary_str.htm)  Produces a printable string representation of a dictionary |
| 4 | [**type(variable)**](https://www.tutorialspoint.com/python/dictionary_type.htm)  Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type. |

Python includes following dictionary methods −

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [**dict.clear()**](https://www.tutorialspoint.com/python/dictionary_clear.htm)  Removes all elements of dictionary *dict* |
| 2 | [**dict.copy()**](https://www.tutorialspoint.com/python/dictionary_copy.htm)  Returns a shallow copy of dictionary *dict* |
| 3 | [**dict.fromkeys()**](https://www.tutorialspoint.com/python/dictionary_fromkeys.htm)  Create a new dictionary with keys from seq and values *set* to *value*. |
| 4 | [**dict.get(key, default=None)**](https://www.tutorialspoint.com/python/dictionary_get.htm)  For *key* key, returns value or default if key not in dictionary |
| 5 | [**dict.has\_key(key)**](https://www.tutorialspoint.com/python/dictionary_has_key.htm)  Returns *true* if key in dictionary *dict*, *false* otherwise |
| 6 | [**dict.items()**](https://www.tutorialspoint.com/python/dictionary_items.htm)  Returns a list of *dict*'s (key, value) tuple pairs |
| 7 | [**dict.keys()**](https://www.tutorialspoint.com/python/dictionary_keys.htm)  Returns list of dictionary dict's keys |
| 8 | [**dict.setdefault(key, default=None)**](https://www.tutorialspoint.com/python/dictionary_setdefault.htm)  Similar to get(), but will set dict[key]=default if *key* is not already in dict |
| 9 | [**dict.update(dict2)**](https://www.tutorialspoint.com/python/dictionary_update.htm)  Adds dictionary *dict2*'s key-values pairs to *dict* |
| 10 | [**dict.values()**](https://www.tutorialspoint.com/python/dictionary_values.htm)  Returns list of dictionary *dict*'s values |

# Python Decision Making

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages –

Three ways to denot conditions:

* + True

False

* + 0 for False

1 for True

* + <condition expression>



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| **if statements** | An **if statement** consists of a boolean expression followed by one or more statements. |
| **if...else statements** | An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE. |
| **nested if statements** | You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).  The **elif** Statement  The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.  Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an if. **Syntax:** The syntax of the nested *if...elif...else* construct may be:  if expression1:  statement(s)  if expression2:  statement(s)  elif expression3:  statement(s)  else:  statement(s)  elif expression4:  statement(s)  else:  statement(s) |

var1 = 5555

if var1:

print("1 – Condition evaluated true");

print(var1);

var2 = 0

if var2:

print("2 - Condition evaluated false");

print(var2);

else:

print("Good bye!");

v\_marks = 85

if (v\_marks<50) :

print("He is fail.");

print("He should work hard.");

elif (v\_marks>=50 and v\_marks<60) :

print("He stood 2nd."); print("He need improvments.");

elif (v\_marks>=60 and v\_marks<80) :

print("He stood 1st.");

else :

print("Outstanding.");

print("Well done.");

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

1 - Condition evaluated true

5555

Good bye!

Outstanding.

Well done.

**# Another Example**

var = 100

if var < 200:

print("Expression value is less than 200");

if var == 150:

print("Which is 150");

elif var == 100:

print("Which is 100");

elif var == 50:

print("Which is 50");

elif var < 50:

print("Expression value is less than 50");

else:

print("Could not find true expression");

print("Good bye!"); # This statement is not the of any if.

**# Output**

Expression value is less than 200

Which is 100

>>> print("Good bye!"); # This statement is not the of any if.

Good bye!

## **Single Statement Suites**

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause −

var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"

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

Value of expression is 100

Good bye!

Example 2.

if ( var == 100 ) : print("Value of expression is 100"); print("hello"); else: print("good bye")

File "<stdin>", line 1

if ( var == 100 ) : print("Value of expression is 100"); print("hello"); else: print("good bye")

^

if ( var == 100 ) : print("Value of expression is 100"); print("hello");

else: print("good bye")

Value of expression is 100

hello

if ( var == 0 ) : print("Value of expression is 100"); print("hello");

else: print("good bye")

good bye

# Python Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements. There are two types of loops in Python, **for** and **while**.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [**while loop**](https://www.tutorialspoint.com/python/python_while_loop.htm) | Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| [**for loop**](https://www.tutorialspoint.com/python/python_for_loop.htm) | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |

**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 beforehand, the number of times to iterate.

## Syntax

while test\_expression:

Body of while

**In Python, the body of the while loop is determined through indentation.**

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.

# Prints out 1,2,3,4,5

count = 1

while count <= 5:

print(count);

count += 1;

print("Good Bye While Loop");

# Output

1

2

3

4

5

Good Bye While Loop

In Python, break and continue statements can alter the flow of a normal loop. The break statement terminates the loop containing it and pass the control of the program to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

# Prints out 1,2,3,4,5

count = 1

while True:

print(count)

count += 1

if count > 5:

break;

print("Good Bye While Loop");

# Output

1

2

3

4

5

Good Bye While Loop

# Prints out 1,2,3,4,6,7,8,9,10

count = 1

while (count<=10):

if (count==5):

count=count+1

continue

else:

print(count)

count=count+1

print("Good Bye While Loop")

# Output

1

2

3

4

6

7

8

9

10

Good Bye While Loop

**We use "else" clause in loops.**

When the loop condition of "for" or "while" statement fails then code part in "else" is executed. If break statement is executed inside for loop then the "else" part is skipped. Note that "else" part is executed even if there is a continue statement.

# While loop with else.

# Print out 1,2,3,4,5 and then it prints "count value reached 5"

count=1

while(count<5):

print(count)

count +=1

else:

print("count value reached %d" %(count))

# Output

1

2

3

4

count value reached 5

**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.

## 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.

for x in (1,2,3,4,5):

print(x)

# Output

1

2

3

4

5

for x in [1,2,3,4,5]:

print(x)

# Output

1

2

3

4

5

for x in ['a','b','c','d','e']:

print(x)

# Output

a

b

c

d

e

for x in range(5):

print(x)

# Output

0

1

2

3

4

for x in range(3, 6):

print(x)

# Output

3

4

5

# range in increment order of 2

for x in range(3, 10, 2):

print(x)

# Output

3

5

7

9

# Prints out only odd numbers - 1,3,5,7,9

for x in range(10):

# Check if x is even

if x % 2 == 0:

**continue;**

print(x);

# Output

1

3

5

7

9

# Prints out 1,2,3,4

for i in range(1, 10):

if(i%5==0):

**break**

print(i)

else:

print("this is not printed because for loop is terminated because of break but not due to fail in condition")

# Output

1

2

3

4

# Program to find the sum of all numbers stored in a list

# List of numbers

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

sum = 0

for val in numbers:

sum = sum+val;

else:

print("The sum is", sum);

# Output

The sum is 48

**Python Pass Statement**

It is used as a placeholder for future implementation of functions, loops, etc.

In Python programming, pass is a **null** statement. The difference between a comment and pass statement is that the interpreter ignores a comment entirely, pass is not ignored it is executed. However, nothing happens when pass is executed. It results into no operation (NOP).

## Syntax of pass

pass

We generally use it as a placeholder.

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 complain. So, we use the pass statement to construct a body that does nothing.

# pass is just a placeholder for

# functionality to be added later.

sequence = [1,2,3,4,5]

for val in sequence:

pass

# Python Numbers

# Integer Objects

All integers are implemented as “long” integer objects of **arbitrary size**.

<https://docs.python.org/3/c-api/long.html>

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a **new allocated object.**

Number objects are created when you assign a value to them. For example −

var1 = 1

var2 = 10

id(var1)

1539688112

id(var2)

1539688400

var1 = 100

var2 = 200

**NOTE: Here object is reallocated.**

id(var1)

1539691280

id(var2)

1539694480

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is −

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the **del** statement. For example:

del var

del var\_a, var\_b

Python supports four different numerical types −

* **int (signed integers)**: They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
* **long (long integers )**: Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L. No longer supported in Version 3.x.
* **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).
* **complex (complex numbers)** : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

**What is the range of int in python?**

**Python** has arbitrary precision integers so there **is no true** fixed maximum. You're only limited by available memory. sys.maxint does not even exist in **Python** 3, since **int** and long were unified into a single arbitrary precision **int** type.

v\_num = 9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999;

print(v\_num);

9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999

### **Examples**

Here are some examples of numbers

|  |  |  |  |
| --- | --- | --- | --- |
| **int** | **long** | **float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9 | 9.322e-36j |
| 080 | 0xDEFABCECBDAECBFBAEL | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

## **Number Type Conversion**

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

* Type **int(x)** to convert x to a plain integer.
* Type **float(x)** to convert x to a floating-point number.
* Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
* Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

|  |
| --- |
| float(2);  2.0  float(5.55);  5.55  complex(5)  (5+0j)  complex(5,2)  (5+2j) |

## **Mathematical Functions**

Python includes following functions that perform mathematical calculations.

|  |  |
| --- | --- |
| **Function** | **Returns ( description )** |
| **abs(x)** | The absolute value of x: the (positive) distance between x and zero.  abs(-5);  5 |
| **ceil(x)** | The ceiling of x: the smallest integer not less than x  import math  math.ceil(4.4); 5  math.ceil(4.7); 5 |
| **cmp(x, y)** | -1 if x < y, 0 if x == y, or 1 if x > y Python 3 doesn't have a cmp function. |
| **exp(x)** | The exponential of x: ex  math.exp(1e-5)  1.00001000005 |
| **fabs(x)** | The absolute value of x.  math.fabs(-5.5)  5.5  abs(-5.5)  5.5 |
| **floor(x)** | The floor of x: the largest integer not greater than x  math.floor(4.4); 4  math.floor(4.8); 4 |
| **log(x)** | The natural logarithm of x, for x> 0  math.log(5);  1.6094379124341003 |
| **log10(x)** | The base-10 logarithm of x for x> 0 .  math.log10(5);  0.6989700043360189 |
| **max(x1, x2,...)** | The largest of its arguments: the value closest to positive infinity  max(10,20,66,45,87,55,23,58)  87 |
| **min(x1, x2,...)** | The smallest of its arguments: the value closest to negative infinity  min(10,20,66,45,87,55,23,58)  10 |
| **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.  math.modf(5);  (0.0, 5.0)  math.modf(1,5);  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: modf() takes exactly one argument (2 given) |
| **pow(x, y)** | The value of x\*\*y.  math.pow(2,5);  32.0 |
| **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.  round(7676567.6556757,3);  7676567.656  round(7676567.6544757,3);  7676567.654 |
| **sqrt(x)** | The square root of x for x > 0  math.sqrt(16)  4.0 |

## **Random Number Functions**

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

|  |  |
| --- | --- |
| **Function** | **Description** |
| **choice(seq)** | A random item from a list, tuple, or string.  import random;  x=**[**1, 2, 3, 5, 9**]**;  print(type(x))  <class 'list'>  x=**(**1, 2, 3, 5, 9**)**;  print(type(x))  <class 'tuple'>  random.choice([1, 2, 3, 5, 9]);  5  random.choice([1, 2, 3, 5, 9]);  1  random.choice((1, 2, 3, 5, 9));  5  random.choice((1, 2, 3, 5, 9));  2  random.choice('India is my country.');  'd'  random.choice('India is my country.');  'm' |
| **randrange ([start,] stop [,step])** | A randomly selected element from range(start, stop, step)  random.randrange(100, 1000, 2);  334  random.randrange(100, 1000, 2);  434  random.randrange(100, 1000, 2);  624 |
| **random()** | A random float r, such that 0 is less than or equal to r and r is less than 1  print("random() : ", random.random());  random() : 0.0623392041549814  print("random() : ", random.random());  random() : 0.9609234212687972 |
| **seed([x])** | Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.4288890546751146  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.20609823213950174  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.81332125135732  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.8235888725334455 |
| **shuffle(lst)** | Randomizes the items of a list in place. Returns None.  list = [20, 16, 10, 5];  random.shuffle(list);  print(list);  [10, 5, 20, 16]  random.shuffle(list);  print(list);  [5, 16, 10, 20]  random.shuffle(list);  print(list);  [20, 16, 5, 10] |
| **uniform(x, y)** | A random float r, such that x is less than or equal to r and r is less than y  print("Random Float uniform(1, 10) : ", random.uniform(1, 10))  print("Random Float uniform(1, 10) : ", random.uniform(1, 10))  Random Float uniform(1, 10) : 6.418091089533576  Random Float uniform(1, 10) : 8.37482423567478  print("Random Float uniform(7, 10) : ", random.uniform(7, 10))  print("Random Float uniform(7, 10) : ", random.uniform(7, 10))  Random Float uniform(7, 10) : 8.370493453174916  Random Float uniform(7, 10) : 9.057584456423784 |

## **Mathematical Constants**

The module also defines two mathematical constants −

|  |  |
| --- | --- |
| **Constants** | **Description** |
| pi | The mathematical constant pi.  import math  math.pi  3.141592653589793 |
| e | The mathematical constant e.  math.e  2.718281828459045 |

# Python Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

var1 = 'Hello World!'

var2 = "Python Programming"

## **Accessing Values in Strings**

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

#!/usr/bin/python

var1 = 'Hello World!'

var2 = "Python Programming"

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

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

print("var2[:10]: ", var2[:10])

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

var1[0]: H

var2[1:5]: ytho

var2[:10]: Python Pro

## **Updating Strings**

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. You cannot update a character of a string. For example −

var1 = 'Hello World!'

print(var1);

var1 = 'Hello World, Welcome to Python!'

print(var1);

var1 = 'ABC';

var1[2] = 'D'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

## **Escape Characters**

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

|  |  |  |
| --- | --- | --- |
| **Backslash notation** | **Hexadecimal character** | **Description** |
| \a | 0x07 | Bell or alert sound generate from your laptop  print('Calling Bell!', '\a');  Calling Bell! |
| \b | 0x08 | Backspace  print('Calling Bell!'+' open the door.');  Calling Bell! open the door.  print('Calling Bell!'+'\b'+' open the door.');  Calling Bell open the door.  print('Calling Bell!'+'\b\b\b\b\b'+' open the door.');  Calling open the door. |
| \e | 0x1b | Escape |
| \f | 0x0c | Formfeed  print('Calling Bell!','\f','Open the door'); |
| \n | 0x0a | Newline  print('Calling Bell!','\n','\bOpen the door');  Calling Bell!  Open the door |
| \nnn |  | Octal notation, where n is in the range 0.7 |
| \r | 0x0d | Carriage return  print('Calling Bell!','\r','Open the door', '\r', 'Guests are welcome.');  Guest are welcome. |
| \s | 0x20 | Space |
| \t | 0x09 | Tab  print('Hello''\t''World');  Hello World |
| \v | 0x0b | Vertical tab |
| \x |  | Character x |
| \xnn |  | Hexadecimal notation, where n is in the range 0.9, a.f, or A.F  print('Hello''\xad''World');  Hello­World  print('Hello''\xae''World');  Hello®World |

## **String Special Operators**

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Concatenation - Adds values on either side of the operator | a + b will give HelloPython  a = 'Hello'  b = 'Python'  a+b  'HelloPython' |
| \* | Repetition - Creates new strings, concatenating multiple copies of the same string | a\*2 will give –HelloHello  a\*3  'HelloHelloHello' |
| [] | Slice - Gives the character from the given index | a[1] will give e |
| [ : ] | Range Slice - Gives the characters from the given range | a[1:4] will give ell |
| in | Membership - Returns true if a character exists in the given string | H in a will give 1 (True)  'l' in var1  True  'w' in var1  False |
| not in | Membership - Returns true if a character does not exist in the given string | M not in a will give 1 (True)  'w' not in var1  True |
| r/R | Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark. | print r'\n' prints \n and print R'\n'prints \n  print('\n')  print(r'\n')  \n |
| % | Format - Performs String formatting | v\_num = 100;  print('v\_num is: %d' %(v\_num));  v\_num is: 100 |

## **String Formatting Operator**

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

#!/usr/bin/python

print("Heavyweight boxer %s, total win %d/58" % ('Michael Gerard Tyson', 50));

Heavyweight boxer Michael Gerard Tyson, total win 50/58

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

|  |  |
| --- | --- |
| **Format Symbol** | **Conversion** |
| %c | Character  fname='Hari'  lname='Yadav'  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 65 kg! |
| %s | string conversion via str() prior to formatting  print("My name is %s" %("Michael Gerard Tyson"));  My name is Michael Gerard Tyson  print("My name is {} {} and surname {} ".format('Michael', 'Gerard', 'Tyson'));  My name is Michael Gerard and surname Tyson  print('%s' % ('test',))  test  print('%10s' % ('test',))  test |
| %i | signed decimal integer |
| %d | signed decimal integer  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 65 kg!  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -65 kg! |
| %u | unsigned decimal integer  Obsolete type – it is identical to ‘d’. See PEP 237. |
| %o | octal integer  print("My name initial is %c%c and weight is %o kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -101 kg!  print("My name initial is %c%c and weight is %o kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 101 kg! |
| %x | hexadecimal integer (lowercase letters)  print("My name initial is %c%c and weight is %x kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 41 kg! |
| %X | hexadecimal integer (UPPERcase letters) |
| %e | exponential notation (with lowercase 'e')  print("My name initial is %c%c and weight is %e kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 6.500000e+01 kg! |
| %E | exponential notation (with UPPERcase 'E')  print("My name initial is %c%c and weight is %E kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -6.500000E+01 kg! |
| %f | floating point real number  print("My name initial is %c%c and weight is %f kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 65.000000 kg! |

Other supported symbols and functionality are listed in the following table −

|  |  |
| --- | --- |
| **Symbol** | **Functionality** |
| \* | argument specifies width or precision |
| - | left justification  print('%10d' %(837.64)); # right justified.  837  print('%-10d' %(837.64)); # left justified.  837 |
| + | display the sign  print('%010d' %(+837.64));  0000000837  print('%+010d' %(837.64));  +000000837  print('%010d' %(-837.64));  -000000837 |
| <sp> | leave a blank space before a positive number  print('% 010d' %(837.64));  000000837 |
| # | add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.  print("%#x" % 17 )  0x11  print("%#x" % 18 )  0x12  print("%x" % 17 )  11  print("%x" % 18 )  12 |
| 0 | pad from left with zeros (instead of spaces)  print('%10d' %(837.64));  837  print('%010d' %(837.64));  0000000837 |
| f | Represent the value in floating point.  print('%f' %(837.64759678));  837.647597  print('%.1f' %(837.64759678));  837.6  print('%.2f' %(837.64759678));  837.65  print('%.4f' %(837.64759678));  837.6476  print('%010.2f' %(837.64759678));  0000837.65  print('%010.4f' %(837.64759678));  00837.6476 |
| % | '%%' leaves you with a single literal '%'  print('I have scored %d%%' %(85));  I have scored 85%  print('I have scored %d%% %d%%' %(85,90));  I have scored 85% 90% |

## **Triple Quotes**

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double**quotes.

para\_str = """this is a long string that is made up of

several lines and non-printable characters such as

TAB ( \t ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [ \n ], or just a NEWLINE within

the variable assignment will also show up.

"""

print(para\_str)

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n) –

this is a long string that is made up of

several lines and non-printable characters such as

TAB ( ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [

], or just a NEWLINE within

the variable assignment will also show up.

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it −

print('C:\\nowhere')

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

C:\nowhere

Now let's make use of raw string. We would put expression in **r'expression'**as follows −

print(r'C:\\nowhere')

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

C:\\nowhere

## **Unicode String**

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world.

Var1 = 'Hello, world!';

Var2 = u'Hello, world!';

print(u'Hello, world!')

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

Hello, world!

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

## **unicode()** global function[#](http://www.diveintopython3.net/porting-code-to-python-3-with-2to3.html#unicode)

Python 2 had two global functions to coerce objects into strings: unicode() to coerce them into Unicode strings, and str() to coerce them into non-Unicode strings. Python 3 has only one string type so the str() function is all you need. (The unicode() function no longer exists.)

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|  | unicode(anything) | str(anything) |

## **Built-in String Methods**

Python includes the following built-in methods to manipulate strings −

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **capitalize()** Capitalizes first letter of string  print(str.capitalize('Michael Gerard Tyson'));  Michael gerard tyson |
| 2 | [**center(width, fillchar)**](https://www.tutorialspoint.com/python/string_center.htm)  Returns a space-padded string with the original string centered to a total of width columns.  vname = 'MICHAEL GERARD TYSON';  print(vname.center(30,'x'));  xxxxxMICHAEL GERARD TYSONxxxxx  print(vname.center(30,' '));  MICHAEL GERARD TYSON  print(vname.center(30,' x '));  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: The fill character must be exactly one character long |
| 3 | [**count(str, beg= 0,end=len(string))**](https://www.tutorialspoint.com/python/string_count.htm)  Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given. |
| 4 | [**decode(encoding='UTF-8',errors='strict')**](https://www.tutorialspoint.com/python/string_decode.htm)  Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. |
| 5 | [**encode(encoding='UTF-8',errors='strict')**](https://www.tutorialspoint.com/python/string_encode.htm)  Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'. |
| 6 | [**endswith(suffix, beg=0, end=len(string))**](https://www.tutorialspoint.com/python/string_endswith.htm) Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise. |
| 7 | [**expandtabs(tabsize=8)**](https://www.tutorialspoint.com/python/string_expandtabs.htm)  Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided. |
| 8 | [**find(str, beg=0 end=len(string))**](https://www.tutorialspoint.com/python/string_find.htm)  Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise. |
| 9 | [**index(str, beg=0, end=len(string))**](https://www.tutorialspoint.com/python/string_index.htm)  Same as find(), but raises an exception if str not found. |
| 10 | [**isalnum()**](https://www.tutorialspoint.com/python/string_isalnum.htm)  Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise. |
| 11 | [**isalpha()**](https://www.tutorialspoint.com/python/string_isalpha.htm)  Returns true if string has at least 1 character and all characters are alphabetic and false otherwise. |
| 12 | [**isdigit()**](https://www.tutorialspoint.com/python/string_isdigit.htm)  Returns true if string contains only digits and false otherwise. |
| 13 | [**islower()**](https://www.tutorialspoint.com/python/string_islower.htm)  Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise. |
| 14 | [**isnumeric()**](https://www.tutorialspoint.com/python/string_isnumeric.htm)  Returns true if a unicode string contains only numeric characters and false otherwise. |
| 15 | [**isspace()**](https://www.tutorialspoint.com/python/string_isspace.htm)  Returns true if string contains only whitespace characters and false otherwise. |
| 16 | [**istitle()**](https://www.tutorialspoint.com/python/string_istitle.htm)  Returns true if string is properly "titlecased" and false otherwise. |
| 17 | [**isupper()**](https://www.tutorialspoint.com/python/string_isupper.htm)  Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise. |
| 18 | [**join(seq)**](https://www.tutorialspoint.com/python/string_join.htm)  Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string. |
| 19 | [**len(string)**](https://www.tutorialspoint.com/python/string_len.htm)  Returns the length of the string |
| 20 | [**ljust(width[, fillchar])**](https://www.tutorialspoint.com/python/string_ljust.htm)  Returns a space-padded string with the original string left-justified to a total of width columns. |
| 21 | [**lower()**](https://www.tutorialspoint.com/python/string_lower.htm)  Converts all uppercase letters in string to lowercase.  print(str.lower('Michael Gerard Tyson'));  michael gerard tyson |
| 22 | [**lstrip()**](https://www.tutorialspoint.com/python/string_lstrip.htm)  Removes all leading whitespace in string. |
| 23 | [**maketrans()**](https://www.tutorialspoint.com/python/string_maketrans.htm)  Returns a translation table to be used in translate function. |
| 24 | [**max(str)**](https://www.tutorialspoint.com/python/string_max.htm)  Returns the max alphabetical character from the string str. |
| 25 | [**min(str)**](https://www.tutorialspoint.com/python/string_min.htm)  Returns the min alphabetical character from the string str. |
| 26 | [**replace(old, new [, max])**](https://www.tutorialspoint.com/python/string_replace.htm)  Replaces all occurrences of old in string with new or at most max occurrences if max given. |
| 27 | [**rfind(str, beg=0,end=len(string))**](https://www.tutorialspoint.com/python/string_rfind.htm)  Same as find(), but search backwards in string. |
| 28 | [**rindex( str, beg=0, end=len(string))**](https://www.tutorialspoint.com/python/string_rindex.htm)  Same as index(), but search backwards in string. |
| 29 | [**rjust(width,[, fillchar])**](https://www.tutorialspoint.com/python/string_rjust.htm)  Returns a space-padded string with the original string right-justified to a total of width columns. |
| 30 | [**rstrip()**](https://www.tutorialspoint.com/python/string_rstrip.htm)  Removes all trailing whitespace of string. |
| 31 | [**split(str="", num=string.count(str))**](https://www.tutorialspoint.com/python/string_split.htm)  Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given. |
| 32 | [**splitlines( num=string.count('\n'))**](https://www.tutorialspoint.com/python/string_splitlines.htm)  Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed. |
| 33 | [**startswith(str, beg=0,end=len(string))**](https://www.tutorialspoint.com/python/string_startswith.htm)  Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise. |
| 34 | [**strip([chars])**](https://www.tutorialspoint.com/python/string_strip.htm)  Performs both lstrip() and rstrip() on string |
| 35 | [**swapcase()**](https://www.tutorialspoint.com/python/string_swapcase.htm)  Inverts case for all letters in string. |
| 36 | [**title()**](https://www.tutorialspoint.com/python/string_title.htm)  Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase. |
| 37 | [**translate(table, deletechars="")**](https://www.tutorialspoint.com/python/string_translate.htm)  Translates string according to translation table str(256 chars), removing those in the del string. |
| 38 | **upper()** Converts lowercase letters in string to uppercase.  print(str.upper('Michael Gerard Tyson'));  MICHAEL GERARD TYSON |
| 39 | [**zfill (width)**](https://www.tutorialspoint.com/python/string_zfill.htm)  Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). |
| 40 | [**isdecimal()**](https://www.tutorialspoint.com/python/string_isdecimal.htm)  Returns true if a unicode string contains only decimal characters and false otherwise. |

# Python Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

## **Python Lists**

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

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

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

list2 = [1, 2, 3, 4, 5 ];

list3 = ["a", "b", "c", "d"]

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

## **Accessing Values in Lists**

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

#!/usr/bin/python

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**

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

#!/usr/bin/python

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]

**Note:** append() method is discussed in subsequent section.

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**

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

#!/usr/bin/python

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

print list1

del list1[2];

print "After deleting value at index 2 : "

print list1

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

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

After deleting value at index 2 :

['physics', 'chemistry', 2000]

**Note:** remove() method is discussed in subsequent section.

## **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 |

## **Indexing, Slicing, and Matrixes**

Because lists are sequences, indexing and slicing work the same way for lists 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 List Functions & Methods:**

Python includes the following list functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(list1, list2)**](https://www.tutorialspoint.com/python/list_cmp.htm)  Compares elements of both lists. |
| 2 | [**len(list)**](https://www.tutorialspoint.com/python/list_len.htm)  Gives the total length of the list. |
| 3 | [**max(list)**](https://www.tutorialspoint.com/python/list_max.htm)  Returns item from the list with max value. |
| 4 | [**min(list)**](https://www.tutorialspoint.com/python/list_min.htm)  Returns item from the list with min value. |
| 5 | [**list(seq)**](https://www.tutorialspoint.com/python/list_list.htm)  Converts a tuple into list. |

Python includes following list methods

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [**list.append(obj)**](https://www.tutorialspoint.com/python/list_append.htm)  Appends object obj to list |
| 2 | [**list.count(obj)**](https://www.tutorialspoint.com/python/list_count.htm)  Returns count of how many times obj occurs in list |
| 3 | [**list.extend(seq)**](https://www.tutorialspoint.com/python/list_extend.htm)  Appends the contents of seq to list |
| 4 | [**list.index(obj)**](https://www.tutorialspoint.com/python/list_index.htm)  Returns the lowest index in list that obj appears |
| 5 | [**list.insert(index, obj)**](https://www.tutorialspoint.com/python/list_insert.htm)  Inserts object obj into list at offset index |
| 6 | [**list.pop(obj=list[-1])**](https://www.tutorialspoint.com/python/list_pop.htm)  Removes and returns last object or obj from list |
| 7 | [**list.remove(obj)**](https://www.tutorialspoint.com/python/list_remove.htm)  Removes object obj from list |
| 8 | [**list.reverse()**](https://www.tutorialspoint.com/python/list_reverse.htm)  Reverses objects of list in place |
| 9 | [**list.sort([func])**](https://www.tutorialspoint.com/python/list_sort.htm)  Sorts objects of list, use compare func if given |

# Python Tuples

A tuple is a sequence of immutable Python objects. 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.

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 );

tup3 = "a", "b", "c", "d";

The empty tuple is written as two parentheses containing nothing −

tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value −

tup1 = (50,);

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

## **Accessing Values in Tuples:**

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

#!/usr/bin/python

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**

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

#!/usr/bin/python

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**

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example:

#!/usr/bin/python

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 |

## **No Enclosing Delimiters**

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz'

x, y = 1, 2;

print "Value of x , y : ", x,y

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

abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2

## **Built-in Tuple Functions**

Python includes the following tuple functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(tuple1, tuple2)**](https://www.tutorialspoint.com/python/tuple_cmp.htm)  Compares elements of both tuples. |
| 2 | [**len(tuple)**](https://www.tutorialspoint.com/python/tuple_len.htm)  Gives the total length of the tuple. |
| 3 | [**max(tuple)**](https://www.tutorialspoint.com/python/tuple_max.htm)  Returns item from the tuple with max value. |
| 4 | [**min(tuple)**](https://www.tutorialspoint.com/python/tuple_min.htm)  Returns item from the tuple with min value. |
| 5 | [**tuple(seq)**](https://www.tutorialspoint.com/python/tuple_tuple.htm)  Converts a list into tuple. |

# Python Dictionary

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. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

## **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 −

#!/usr/bin/python

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 −

#!/usr/bin/python

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**

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 −

#!/usr/bin/python

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**

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.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

#!/usr/bin/python

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

**Note:** del() method is discussed in subsequent section.

## **Properties of Dictionary Keys**

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

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 −

#!/usr/bin/python

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

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

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

dict['Name']: Manni

**(b)** 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:

#!/usr/bin/python

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

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

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

Traceback (most recent call last):

File "test.py", line 3, in <module>

dict = {['Name']: 'Zara', 'Age': 7};

TypeError: list objects are unhashable

## **Built-in Dictionary Functions & Methods −**

Python includes the following dictionary functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(dict1, dict2)**](https://www.tutorialspoint.com/python/dictionary_cmp.htm)  Compares elements of both dict. |
| 2 | [**len(dict)**](https://www.tutorialspoint.com/python/dictionary_len.htm)  Gives the total length of the dictionary. This would be equal to the number of items in the dictionary. |
| 3 | [**str(dict)**](https://www.tutorialspoint.com/python/dictionary_str.htm)  Produces a printable string representation of a dictionary |
| 4 | [**type(variable)**](https://www.tutorialspoint.com/python/dictionary_type.htm)  Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type. |

Python includes following dictionary methods −

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [**dict.clear()**](https://www.tutorialspoint.com/python/dictionary_clear.htm)  Removes all elements of dictionary *dict* |
| 2 | [**dict.copy()**](https://www.tutorialspoint.com/python/dictionary_copy.htm)  Returns a shallow copy of dictionary *dict* |
| 3 | [**dict.fromkeys()**](https://www.tutorialspoint.com/python/dictionary_fromkeys.htm)  Create a new dictionary with keys from seq and values *set* to *value*. |
| 4 | [**dict.get(key, default=None)**](https://www.tutorialspoint.com/python/dictionary_get.htm)  For *key* key, returns value or default if key not in dictionary |
| 5 | [**dict.has\_key(key)**](https://www.tutorialspoint.com/python/dictionary_has_key.htm)  Returns *true* if key in dictionary *dict*, *false* otherwise |
| 6 | [**dict.items()**](https://www.tutorialspoint.com/python/dictionary_items.htm)  Returns a list of *dict*'s (key, value) tuple pairs |
| 7 | [**dict.keys()**](https://www.tutorialspoint.com/python/dictionary_keys.htm)  Returns list of dictionary dict's keys |
| 8 | [**dict.setdefault(key, default=None)**](https://www.tutorialspoint.com/python/dictionary_setdefault.htm)  Similar to get(), but will set dict[key]=default if *key* is not already in dict |
| 9 | [**dict.update(dict2)**](https://www.tutorialspoint.com/python/dictionary_update.htm)  Adds dictionary *dict2*'s key-values pairs to *dict* |
| 10 | [**dict.values()**](https://www.tutorialspoint.com/python/dictionary_values.htm)  Returns list of dictionary *dict*'s values |

# Python Date & Time

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

## **What is Tick?**

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970.

There is a popular **time** module available in Python which provides functions for working with times, and for converting between representations. The function *time.time()* returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

## **Example**

#!/usr/bin/python

import time; # This is required to include time module.

ticks = time.time()

print("Number of ticks since 12:00am, January 1, 1970:", ticks);

This would produce a result something as follows −

Number of ticks since 12:00am, January 1, 1970: 1507888743.720282

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

## **What is TimeTuple?**

Many of Python's time functions handle time as a tuple of 9 numbers, as shown below:

|  |  |  |
| --- | --- | --- |
| **Index** | **Field** | **Values** |
| 0 | 4-digit year | 2008 |
| 1 | Month | 1 to 12 |
| 2 | Day | 1 to 31 |
| 3 | Hour | 0 to 23 |
| 4 | Minute | 0 to 59 |
| 5 | Second | 0 to 61 (60 or 61 are leap-seconds) |
| 6 | Day of Week | 0 to 6 (0 is Monday) |
| 7 | Day of year | 1 to 366 (Julian day) |
| 8 | Daylight savings | -1, 0, 1, -1 means library determines DST |

The above tuple is equivalent to **struct\_time** structure. This structure has following attributes:

|  |  |  |
| --- | --- | --- |
| **Index** | **Attributes** | **Values** |
| 0 | tm\_year | 2008 |
| 1 | tm\_mon | 1 to 12 |
| 2 | tm\_mday | 1 to 31 |
| 3 | tm\_hour | 0 to 23 |
| 4 | tm\_min | 0 to 59 |
| 5 | tm\_sec | 0 to 61 (60 or 61 are leap-seconds) |
| 6 | tm\_wday | 0 to 6 (0 is Monday) |
| 7 | tm\_yday | 1 to 366 (Julian day) |
| 8 | tm\_isdst | -1, 0, 1, -1 means library determines DST |

## **Getting current time**

To translate a time instant from a *seconds since the epoch* floating-point value into a time-tuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all nine items valid.

#!/usr/bin/python

import time;

localtime = time.localtime(time.time())

print("Local current time :", localtime);

This would produce the following result, which could be formatted in any other presentable form −

Local current time : time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=16, tm\_min=2, tm\_sec=8, tm\_wday=4, tm\_yday=286, tm\_isdst=0)

**Getting formatted time**

You can format any time as per your requirement, but simple method to get time in readable format is **asctime()** :

#!/usr/bin/python

import time;

localtime = time.asctime( time.localtime(time.time()) )

print("Local current time :", localtime)

This would produce the following result −

Local current time : Fri Oct 13 16:10:30 2017

## **Getting calendar for a month**

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month ( Jan 2017 )

#!/usr/bin/python

import calendar

cal = calendar.month(2017, 1)

print(cal)

This would produce the following result −

January 2017

Mo Tu We Th Fr Sa Su

1

2 3 4 5 6 7 8

9 10 11 12 13 14 15

16 17 18 19 20 21 22

23 24 25 26 27 28 29

30 31

## **The *time* Module**

There is a popular **time** module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | **time.altzone**  The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.  print("time.altzone %d " % time.altzone)  time.altzone -23400 |
| 2 | **time.asctime([tupletime])**  Accepts a time-tuple and returns as character readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.  print(time.gmtime())  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=10, tm\_min=54, tm\_sec=4, tm\_wday=4, tm\_yday=286, tm\_isdst=0)  # Output in UTC (Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT).  time.asctime(time.gmtime())  'Fri Oct 13 10:56:04 2017'  time.asctime(time.localtime())  'Fri Oct 13 16:26:18 2017' |
| 3 | **time.clock( )**  Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time(). On Unix and in Windows it returns wall-clock seconds elapsed since the first call to this function, as a floating point number.  time.clock( )  2.1935462828870445  time.clock( )  4.097568009977395  time.clock( )  5.349900923483391 |
| 4 | **time.ctime([secs])**  Like asctime(localtime(secs)) and without arguments is like asctime( )  time.asctime(time.gmtime())  'Fri Oct 13 11:25:41 2017'  time.asctime(time.localtime())  'Fri Oct 13 16:57:43 2017'  time.ctime()  'Fri Oct 13 16:57:52 2017' |
| 5 | **time.gmtime([secs])**  Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm\_isdst is always 0  time.gmtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=11, tm\_min=31, tm\_sec=33, tm\_wday=4, tm\_yday=286, tm\_isdst=0) |
| 6 | **time.localtime([secs])**  Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm\_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).  time.localtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=17, tm\_min=3, tm\_sec=35, tm\_wday=4, tm\_yday=286, tm\_isdst=0) |
| 7 | **time.mktime(tupletime)**  The method **mktime()** is the inverse function of localtime(). Its argument is the struct\_time or full 9-tuple and it returns a floating point number, for compatibility with time().  If the input value cannot be represented as a valid time, either *OverflowError*or *ValueError* will be raised.  time.localtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=17, tm\_min=18, tm\_sec=50, tm\_wday=4, tm\_yday=286, tm\_isdst=0)  t =(2017,10,13,17,18,50,4,286,0);  time.mktime(t);  1507895330.0  time.mktime(time.localtime())  1507895862.0 |
| 8 | **time.sleep(secs)**  Suspends the calling thread for secs seconds.  print("Start : %s" % time.ctime())  Start : Fri Oct 13 18:46:57 2017  time.sleep( 5 )  print("End : %s" % time.ctime())  End : Fri Oct 13 18:47:02 2017 |
| 9 | **time.strftime(fmt[,tupletime])**  Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.  Syntax  Following is the syntax for strftime() method:  time.strftime(format[, t])  Format elements   * %a - abbreviated weekday name * %A - full weekday name * %b - abbreviated month name * %B - full month name * %c - preferred date and time representation * %C - century number (the year divided by 100, range 00 to 99) * %d - day of the month (01 to 31) * %D - same as %m/%d/%y * %e - day of the month (1 to 31) * %g - like %G, but without the century * %G - 4-digit year corresponding to the ISO week number (see %V). * %h - same as %b * %H - hour, using a 24-hour clock (00 to 23) * %I - hour, using a 12-hour clock (01 to 12) * %j - day of the year (001 to 366) * %m - month (01 to 12) * %M - minute * %n - newline character * %p - either am or pm according to the given time value * %r - time in a.m. and p.m. notation * %R - time in 24 hour notation * %S - second * %t - tab character * %T - current time, equal to %H:%M:%S * %u - weekday as a number (1 to 7), Monday=1. Warning: In Sun Solaris Sunday=1 * %U - week number of the current year, starting with the first Sunday as the first day of the first week * %V - The ISO 8601 week number of the current year (01 to 53), where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week * %W - week number of the current year, starting with the first Monday as the first day of the first week * %w - day of the week as a decimal, Sunday=0 * %x - preferred date representation without the time * %X - preferred time representation without the date * %y - year without a century (range 00 to 99) * %Y - year including the century * %Z or %z - time zone or name or abbreviation   t =(2017,10,13,17,18,50,4,286,0);  t = time.mktime(t)  print(time.strftime("%b %d %Y %H:%M:%S", time.gmtime(t)))  Oct 13 2017 11:48:50 |
| 10 | [**time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')**](https://www.tutorialspoint.com/python/time_strptime.htm)  Parses str according to format string fmt and returns the instant in time-tuple format. The method strptime() parses a string representing a time according to a format. The return value is a struct\_time as returned by gmtime() or localtime().  The format parameter uses the same directives as those used by strftime(); it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by ctime().  If string cannot be parsed according to format, or if it has excess data after parsing, ValueError is raised.  **Syntax**  Following is the syntax for strptime() method:  **time.strptime(string[, format]**  t = time.strptime("30 Nov 2017 16 30 45", "%d %b %Y %H %M %S");  print(t)  time.struct\_time(tm\_year=2017, tm\_mon=11, tm\_mday=30, tm\_hour=16, tm\_min=30, tm\_sec=45, tm\_wday=3, tm\_yday=334, tm\_isdst=-1) |
| 11 | **time.time( )**  Returns the current time instant, a floating-point number of seconds since the epoch.  time.time( )  1507904917.6235135 |
| 12 | **time.tzset()**  Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.  Availability: Unix. |

Let us go through the functions briefly −

There are following two important attributes available with time module:

|  |  |
| --- | --- |
| **SN** | **Attribute with Description** |
| 1 | **time.timezone**  Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).  time.timezone  -19800 |
| 2 | **time.tzname**  Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.  time.tzname  ('India Standard Time', 'India Daylight Time') |

## **The *calendar* Module**

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call **calendar.setfirstweekday()** function.

Here is a list of functions available with the *calendar* module:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | **calendar.calendar(year,w=2,l=1,c=6)**  Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21\*w+18+2\*c. l is the number of lines for each week.  print(calendar.calendar(2017,w=1,l=1,c=1));  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  July August September  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 1 2  2 3 4 5 6 7 8 6 7 8 9 10 11 12 3 4 5 6 7 8 9  9 10 11 12 13 14 15 13 14 15 16 17 18 19 10 11 12 13 14 15 16  16 17 18 19 20 21 22 20 21 22 23 24 25 26 17 18 19 20 21 22 23  23 24 25 26 27 28 29 27 28 29 30 31 24 25 26 27 28 29 30  30 31  October November December  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2  8 9 10 11 12 13 14 5 6 7 8 9 10 11 3 4 5 6 7 8 9  15 16 17 18 19 20 21 12 13 14 15 16 17 18 10 11 12 13 14 15 16  22 23 24 25 26 27 28 19 20 21 22 23 24 25 17 18 19 20 21 22 23  29 30 31 26 27 28 29 30 24 25 26 27 28 29 30  31 |
| 2 | **calendar.firstweekday( )**  Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.  calendar.firstweekday( )  0 |
| 3 | **calendar.isleap(year)**  Returns True if year is a leap year; otherwise, False.  calendar.isleap(2016)  True  calendar.isleap(2017)  False |
| 4 | **calendar.leapdays(y1,y2)**  Returns the total number of leap days in the years within range(y1,y2).  calendar.leapdays(2000,2017)  5 |
| 5 | **calendar.month(year,month,w=2,l=1)**  Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7\*w+6. l is the number of lines for each week.  print(calendar.month(2017,10,w=2,l=1))  October 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31 |
| 6 | **calendar.monthcalendar(year,month)**  Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.  print(calendar.monthcalendar(2017,10))  [[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14], [15, 16, 17, 18, 19, 20, 21], [22, 23, 24, 25, 26, 27, 28], [29, 30, 31, 0, 0, 0, 0]] |
| 7 | **calendar.monthrange(year,month)**  Returns two integers. The first one is the code of the weekday for the first day of the month month in year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.  >>> print(calendar.calendar(2017,w=1,l=1,c=1));  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  calendar.monthrange(2017,1)  (6, 31)  calendar.monthrange(2017,2)  (2, 28)  calendar.monthrange(2017,3)  (2, 31)  calendar.monthrange(2017,4)  (5, 30)  calendar.monthrange(2017,5)  (0, 31)  calendar.monthrange(2017,6)  (3, 30) |
| 8 | **calendar.prcal(year,w=2,l=1,c=6)**  Like print(calendar.calendar(year,w,l,c));  calendar.prcal(2017,w=2,l=1,c=2)  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  July August September  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 1 2  2 3 4 5 6 7 8 6 7 8 9 10 11 12 3 4 5 6 7 8 9  9 10 11 12 13 14 15 13 14 15 16 17 18 19 10 11 12 13 14 15 16  16 17 18 19 20 21 22 20 21 22 23 24 25 26 17 18 19 20 21 22 23  23 24 25 26 27 28 29 27 28 29 30 31 24 25 26 27 28 29 30  30 31  October November December  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2  8 9 10 11 12 13 14 5 6 7 8 9 10 11 3 4 5 6 7 8 9  15 16 17 18 19 20 21 12 13 14 15 16 17 18 10 11 12 13 14 15 16  22 23 24 25 26 27 28 19 20 21 22 23 24 25 17 18 19 20 21 22 23  29 30 31 26 27 28 29 30 24 25 26 27 28 29 30  31 |
| 9 | **calendar.prmonth(year,month,w=2,l=1)**  Like print(calendar.month(year,month,w=1,l=1));  print(calendar.month(2017,1,w=1,l=1));  January 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31  calendar.prmonth(2017,1,w=2,l=1)  January 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31 |
| 10 | **calendar.setfirstweekday(weekday)**  Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).  import calendar  calendar.firstweekday( )  0  calendar.setfirstweekday(calendar.SUNDAY);  calendar.firstweekday( )  6 |
| 11 | **calendar.timegm(tupletime)**  The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.  import calendar  import time  import datetime  d = datetime.datetime(2017, 10, 31)  calendar.timegm(d.timetuple())  1509408000  time.mktime(d.timetuple())  1509388200.0  d = datetime.datetime(2017, 10, 31, 16, 30, 30)  calendar.timegm(d.timetuple())  1509467430  time.mktime(d.timetuple())  1509447630.0 |
| 12 | **calendar.weekday(year,month,day)**  Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).  print(calendar.month(2017,10,w=2,l=1));  October 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31  calendar.weekday(2017,10,3)  1 |

## **Other Modules & Functions:**

If you are interested, then here you would find a list of other important modules and functions to play with date & time in Python:

* [The *datetime* Module](http://docs.python.org/library/datetime.html#module-datetime)
* [The *pytz*Module](http://www.twinsun.com/tz/tz-link.htm)
* [The *dateutil* Module](http://labix.org/python-dateutil)

# Python Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like **print()**, etc. but you can also create your own functions. These functions are called *user-defined functions.*

## **Defining a Function**

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* Function blocks begin with the keyword **def** followed by the **function\_name** and **parentheses ( ( ) )**.
* Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
* The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
* The code block within every function starts with a colon (**:**) and is indented.
* The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

## **Syntax**

def function\_name( parameter1, parameter2, parameter3 )**:**

"""function\_docstring"""

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

## **Example**

The following function takes a string as input parameter and prints it on standard screen.

def print\_myname( str ):

"This function print your name."

print(str)

return

## **Calling a Function**

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call **print\_myname()** function −

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This function print your name."

print(str)

return;

# Now you can call printme function

print\_myname("Hari Yadav")

Hari Yadav

## **Pass by reference vs value**

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

#!/usr/bin/python

# Function definition is here

def change\_mylist( mylist ):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print("Values inside the function: ", mylist)

print(id(mylist))

return (10\*10)

# Now you can call change\_mylist function by using a list variable mylist as reference

v\_list = [10,20,30];

print("Values before passing in the function: ", v\_list)

change\_mylist( v\_list );

print("Values after passing in the function: ", v\_list)

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

Values before passing in the function: [10, 20, 30]

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values after passing in the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

#!/usr/bin/python

# Function definition is here

def change\_mylist1( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assig new reference in mylist

print("Values inside the function: ", mylist)

print("Object ID of mylist inside function: ", id(mylist))

return

# Now you can call change\_mylist1 function

mylist = [10,20,30];

print("Object ID of mylist outside function: ", id(mylist))

change\_mylist1( mylist );

print("Values outside the function: ", mylist)

The parameter *mylist* is local to the function change\_mylist1. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result:

Object ID of mylist outside function: 2301171593544

Values inside the function: [1, 2, 3, 4]

Object ID of mylist inside function: 2301171594056

Values outside the function: [10, 20, 30]

## **Function Arguments**

You can call a function by using the following types of formal arguments:

* Required arguments (**positional order**)
* Keyword arguments (**keyword notation**)
* Default arguments
* Variable-length arguments

## **Required arguments**

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *print\_myname()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This prints a passed string into this function"

print(str);

return;

# Now you can call print\_myname function

print\_myname()

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

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

TypeError: print\_myname() missing 1 required positional argument: 'str'

## **Keyword arguments (parameter named notation)**

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

You can also make keyword calls to the *print\_myname()* function in the following ways −

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This prints a passed string into this function"

print(str)

return;

# Now you can call print\_myname function

print\_myname( str = "Michael Gerard Tyson")

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

Michael Gerard Tyson

The following example gives more clear picture. Note that the order of parameters does not matter.

#!/usr/bin/python

# Function definition is here

def personal\_info( name, age ):

"This prints a passed info into this function"

print("Name: ", name)

print("Age ", age)

return;

# Now you can call personal\_info function

personal\_info( age=50, name="Michael Gerard Tyson")

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

Name: Michael Gerard Tyson

Age 50

## **Default arguments**

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

#!/usr/bin/python

# Function definition is here

def personal\_info( name, age, design = "Trainer" ):

"This prints a passed info into this function"

print("Name: ", name)

print("Age ", age)

print("Designation ", design)

return;

# Now you can call personal\_info function

personal\_info( age=50, name="Michael Gerard Tyson" )

personal\_info( age=50, name="Michael Gerard Tyson" , design='Chairman')

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

Name: Michael Gerard Tyson

Age 50

Designation Trainer

Name: Michael Gerard Tyson

Age 50

Designation Chairman

## **Variable-length arguments**

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

def function\_name([formal\_args,] **\***var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

#!/usr/bin/python

# Function definition is here

def student\_marks( name, **\***score\_tuple ):

"This prints a variable length passed arguments"

print("Student subject details is: ")

print(type(score\_tuple))

print("Student name ",name)

for var in score\_tuple:

print(var)

return;

# Now you can call student\_marks function

student\_marks("Dev Singh")

student\_marks ("Sami", 70, 60, 50 )

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

Student subject details is:

Dev Singh

Student subject details is:

Sami

70

60

50

def student\_marks( name, \*score\_tuple, \*class ):

"This prints a variable passed arguments"

print("Student subject details is: ")

print(name, class)

for var in score\_tuple:

print(var)

return;

**NOTE: Only one variable-length arguments can be specified.**

def student\_marks(\*score\_tuple, name):

"This prints a variable passed arguments"

print("Student subject details is: ")

print(name)

for var in score\_tuple:

print(var)

return;

print("Good Bye. . . . . ")

student\_marks("Dev Singh")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: student\_marks() missing 1 required keyword-only argument: 'name'

student\_marks(name="Dev Singh")

Student subject details is:

Dev Singh

**NOTE: Any statement written after return keyword is not executed.**

student\_marks (70, 60, 50, "Sami" )

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: student\_marks() missing 1 required keyword-only argument: 'name'

student\_marks (70, 60, 50, name="Sami" )

Student subject details is:

Sami

70

60

50

## **The *Anonymous* Functions**

These functions are called anonymous because they are not declared in the standard manner by using the ***def* keyword**. You can use the ***lambda*** keyword to create small anonymous functions.

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

## **Syntax**

The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]**:** expression

Following is the example to show how *lambda* form of function works −

#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2**:** arg1 + arg2;

# Now you can call sum as a function

print("Value of total : ", sum( 10, 20 ))

print("Value of total : ", sum( 20, 20 ))

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

Value of total : 30

Value of total : 40

it\_rate10 = lambda arg1: (arg1\*10)/100;

it\_rate10(300000)

30000.0

it\_rate20 = lambda arg1: (arg1\*20)/100;

it\_rate30 = lambda arg1: (arg1\*30)/100;

it\_rate20(600000)

120000.0

it\_rate30(2000000)

600000.0

## **The *return* Statement**

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None. Any statement written after return keyword is not executed.

All the above examples are not returning any value. You can return a value from a function as follows −

#!/usr/bin/python

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print("Total Inside the function : ", total)

return total;

# Now you can call sum function

v\_total = sum( 10, 20 );

print("Total Outside the function : ", v\_total);

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

Total Inside the function : 30

Total Outside the function : 30

## **Scope of Variables**

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

* Global variables
* Local variables

## **Global vs. Local variables**

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

#!/usr/bin/python

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print("Inside the function local total : ", total)

print(id(total))

return total;

# Now you can call sum function

sum( 10, 20 );

print("Outside the function global total : ", total)

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

Inside the function local total : 30

30

Outside the function global total : 0