**Python Keywords**

Keywords are the reserved words in Python.

We cannot use a keyword as a [variable name](https://www.programiz.com/python-programming/variables-datatypes), [function](https://www.programiz.com/python-programming/function) name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive.

There are 33 keywords in Python 3.7. This number can vary slightly in the course of time.

All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords is given below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass |  |
| break | except | in | raise |  |
| Keywords in Python | | | | |

**Python Identifiers**

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

**Rules for writing identifiers**

1. Identifiers can be a combination of letters in lowercase **(a to z)** or uppercase **(A to Z)** or digits **(0 to 9)** or an underscore \_. Names like myClass, var\_1 and print\_this\_to\_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.

**Multi-line statement**

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

1. a = 1 + 2 + 3 + \
2. 4 + 5 + 6 + \
3. 7 + 8 + 9

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

1. a = (1 + 2 + 3 +
2. 4 + 5 + 6 +
3. 7 + 8 + 9)

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

1. colors = ['red',
2. 'blue',
3. 'green']

We could also put multiple statements in a single line using semicolons, as follows

1. a = 1; b = 2; c = 3

a=1

Example: Python Statements

msg="Hello World"

code=123

name="Steve"

Example: Continuation of Statement

msg="Hello Pythonista \

Welcome to Python Tutorial \

from TutorialsTeacher.com"

Example: Multiple Statements in Single Line

msg="Hello World";code=123;name="Steve"

Example: Comments

# this is a comment

print ("Hello World")

print ("Welcome to Python Tutorial") #this is also a comment but after a statement.

Example: Multi-line Comments

'''

comment1

comment2

comment3

'''

print ("Hello World")

The input() function always reads the input as a string, even if comprises of digits. The type() function used earlier confirms this behaviour.

>>> name=input("Enter your name: ")  
Enter your name: Steve   
>>> type(name)   
<class 'str'>   
>>> age=input("Enter your age: ")   
Enter your age: 21   
>>> type(age)   
<class 'str'>

Multiple values can be displayed by the single print() function separated by comma.

>>> name="Ram"   
>>> age=21   
>>> print("Name:", name, "Age:",age)  
Name: Ram Age: 21

By default, a single space (' ') acts as a separator between values. However, any other character can be used by providing a **sep** parameter. In the following example, "=" is used as a separator character.

>>> name="Ram"   
>>> age=21   
>>> print(name,age)  
Ram 21   
>>> print(name,age,sep=",")  
Ram,21

Any suitable identifier can be used as a name of a variable, based on the following rules:

1. The name of the variable should start with either an alphabet letter (lower or upper case) or an underscore (\_), but it cannot start with a digit.
2. More than one alpha-numeric characters or underscores may follow.
3. The variable name can consist of alphabet letter(s), number(s) and underscore(s) only. For example, myVar, MyVar, \_myVar, MyVar123 are valid variable names but m\*var, my-var, 1myVar are invalid variable names.
4. Identifiers in Python are case sensitive. So, NAME, name, nAME, and nAmE are treated as different variable names.

# Python Data Types

Data types are the classification or categorization of data items. Data types represent a kind of value which determines what operations can be performed on that data. Numeric, non-numeric and Boolean (true/false) data are the most used data types. However, each programming language has its own classification largely reflecting its programming philosophy.

Python has the following standard or built-in data types:

## Numeric

A numeric value is any representation of data which has a numeric value. Python identifies three types of numbers:

* **Integer:** Positive or negative whole numbers (without a fractional part)
* **Float:** Any real number with a floating point representation in which a fractional component is denoted by a decimal symbol or scientific notation
* **Complex number:** A number with a real and imaginary component represented as x+yj. x and y are floats and j is -1(square root of -1 called an imaginary number)

## Boolean

Data with one of two built-in values True or False. Notice that 'T' and 'F' are capital. true and false are not valid booleans and Python will throw an error for them.

## Sequence Type

A sequence is an ordered collection of similar or different data types. Python has the following built-in sequence data types:

* **String**: A string value is a collection of one or more characters put in single, double or triple quotes.
* **List** : A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.
* **Tuple**: A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

## Dictionary

A dictionary object is an unordered collection of data in a key:value pair form. A collection of such pairs is enclosed in curly brackets. For example: {1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}

### type() function

Python has an in-built function **type()** to ascertain the data type of a certain value. For example, enter type(1234) in Python shell and it will return <class 'int'>, which means 1234 is an integer value. Try and verify the data type of different values in Python shell, as shown below.

>>> type(1234)   
<class 'int'>   
>>> type(55.50)   
<class 'float'>   
>>> type(6+4j)   
<class 'complex'>   
>>> type("hello")   
<class 'str'>   
>>> type([1,2,3,4])   
<class 'list'>   
>>> type((1,2,3,4))   
<class 'tuple'>   
>>> type({1:"one", 2:"two", 3:"three"})  
<class 'dict'>

# Python - Number Type

An object of Number data type represents a numeric literal. In computer science, a literal is a notation for representing a fixed value in the source code. For example, in the assignment statement:

>>> X=10

Here 10 is a literal, as the numeric value representing 10 is directly stored in memory. However,

>>> Y=X\*2

Here, even if the expression evaluates to 20, it is not literally included in the source code.

Python identifies three numeric types.

## Integer:

Zero, positive and negative whole numbers without a fractional part and having unlimited precision, e.g. 1234, 0, -456.

A number having **0o**or **0O** as prefix represents an **octal** number.

For example: 0O12: equivalent to 10 (ten) in the decimal number system.

A number with **0x** or **0X** as prefix represents **hexadecimal** number.

For example: 0x12: equivalent to 18 (Eighteen) in the decimal number system.

## Float:

Positive and negative real numbers with a fractional part denoted by the decimal symbol or the scientific notation using E or e, e.g. 1234.56, 3.142, -1.55, 0.23.

Scientific notation is used as a short representation to express floats having many digits.

For example:

345600000000 is represented as 3.456e11 or 3.456E11

345.56789 is represented as 3.4556789e2 or 3.4556789E2

## Complex:

A complex number is a number with real and imaginary components. For example, 5 + 6j is a complex number where 5 is the real component and 6 multiplied by j is an imaginary component.

Examples: 1+2j, 10-5.5J, 5.55+2.33j, 3.11e-6+4j

## Arithmetic Operators

| Operator | Description | Example |
| --- | --- | --- |
| + (Addition) | Adds operands on either side of the operator. | >>> a=21  >>> b=10  >>> c=a+b  >>> c  31 |
| - (Subtraction) | Subtracts the right-hand operand from the left-hand operand. | >>> a=21  >>> b=10  >>> c=a-b  >>> c  -11 |
| \* (Multiplication) | Multiplies values on either side of the operator. | >>> a=21  >>> b=10  >>> c=a\*b  >>> c  210 |
| / (Division) | Divides the left-hand operand by the right-hand operand. | >>> a=21  >>> b=10  >>> c=a/b  >>> c  2.1 |
| % (Modulus) | Returns the remainder of the division of the left-hand operand by right-hand operand. | >>> a=21  >>> b=10  >>> c=a%b  >>> c  1 |
| \*\* (Exponent) | Calculates the value of the left-operand raised to the right-operand. | >>> a=21  >>> b=10  >>> c=a\*\*b  >>> c  16679880978201 |
| // (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): | >>> a=9  >>> b=2  >>> c=a//b  >>> c  4 |

## Arithmetic Operations on Complex Numbers

Addition and subtraction of complex numbers is straightforward. Real and imaginary parts are added/subtracted to get the result.

>>> a=6+4j  
>>> b=3+2j  
>>>a+b   
(9+6j)   
>>>a-b   
(3+2j)

The process of multiplying these two complex numbers is very similar to multiplying two binomials. Multiply each term in the first number by each term in the second number.

a=6+4j   
b=3+2j   
c=a\*b   
c=(6+4j)\*(3+2j)   
c=(18+12j+12j+8\*-1)  
c=10+24j

Verify this result using the Python interpreter.

>>> a=6+4j  
>>> b=3+2j  
>>>a\*b   
(10+24j)

To obtain the division of two complex numbers, multiply both sides by the **conjugate** of the denominator, which is a number with the same real part and the opposite imaginary part.

a=6+4j   
b=3+2j   
c=a/b   
c=(6+4j)\*(3-2j)/(3+2j)(3-2j)   
c=(18-12j+12j-8\*-1)/(9-6j+6j-4\*-1)  
c=26/13   
c=2+0j

Verify this using the Python interpreter.

>>> a=6+4j  
>>> b=3+2j  
>>>a/b   
(2+0j)

## Built-in Functions

A numeric object of one type can be converted in another type using the following functions:

| Built-in Function | Description |
| --- | --- |
| [int](https://www.tutorialsteacher.com/python/python-int) | Returns the integer object from a float or a string containing digits. |
| [float](https://www.tutorialsteacher.com/python/python-float) | Returns a floating-point number object from a number or string containing digits with decimal point or scientific notation. |
| [complex](https://www.tutorialsteacher.com/python/python-complex) | Returns a complex number with real and imaginary components. |
| [hex](https://www.tutorialsteacher.com/python/python-hex) | Converts a decimal integer into a hexadecimal number with 0x prefix. |
| [oct](https://www.tutorialsteacher.com/python/python-oct) | Converts a decimal integer in an octal representation with 0o prefix. |
| [pow](https://www.tutorialsteacher.com/python/python-pow) | Returns the power of the specified numbers. |
| [abs](https://www.tutorialsteacher.com/python/python-abs) | Returns the absolute value of a number without considering its sign. |
| [round](https://www.tutorialsteacher.com/python/python-round) | Returns the rounded number. |

# Python - String

A string object is one of the sequence data types in Python. It is an immutable sequence of Unicode characters. Strings are objects of Python's built-in class 'str'. String literals are written by enclosing a sequence of characters in single quotes ('hello'), double quotes ("hello") or triple quotes ('''hello''' or """hello""").

>>> str1='hello'  
>>> str1  
'hello'  
>>> str2="hello"  
>>> str2  
'hello'  
>>> str3='''hello'''  
>>> str3  
'hello'  
>>> str4="""hello"""  
>>> str4  
'hello'

Note that the value of all the strings, as displayed by the Python interpreter, is the same ('hello'), irrespective of whether single, double or triple quotes were used for string formation. If it is required to embed double quotes as part of a string, the string itself should be put in single quotes. On the other hand, if a single-quoted text is to be embedded, the string should be written in double quotes.

>>> str1='Welcome to "Python Tutorial" from TutorialsTeacher'  
>>> str1  
'Welcome to "Python Tutorial" from TutorialsTeacher'  
>>> str2="Welcome to 'Python Tutorial' from TutorialsTeacher"  
>>> str2  
"Welcome to 'Python Tutorial' from TutorialsTeacher"

A sequence is defined as an ordered collection of items. Hence, a string is an ordered collection of characters. The sequence uses an index (starting with zero) to fetch a certain item (a character in case of a string) from it.

>>> myString='hello'  
>>> myString[0]  
'h'  
>>> myString[1]  
'e'  
>>> myString[4]  
'o'

The string is an immutable object. Hence, it is not possible to modify it. The attempt to assign different characters at a certain index results in errors.

>>> myString[1]='a'  
TypeError: 'str' object does not support item assignment

## Triple Quoted String

The triple quoted string is useful when a multi-line text is to be defined as a string literal.

>>> myString="""Welcome to  
Python Tutorial  
from TutorialsTeacher"""  
>>> myString  
'Welcome to "Python Tutorial" from TutorialsTeacher'

## Escape Sequences

The escape character is used to invoke an alternative implementation of the subsequent character in a sequence. In Python backslash \ is used as an escape character. Here is a list of escape sequences and their purpose.

| Escape sequence | Description | Example | Result |
| --- | --- | --- | --- |
| \a | Bell or alert | "\a" | Bell sound |
| \b | Backspace | "ab\bc" | ac |
| \f | Formfeed | "hello\fworld" | hello world |
| \n | Newline | "hello\nworld" | Hello |
| \nnn | Octal notation, where n is in the range 0-7 | '\101' | A |
| \t | Tab | 'Hello\tPython' | Hello Python |
| \xnn | Hexadecimal notation, where n is in the range 0-9, a-f, or A-F | '\x41' | A |

## String Operators

Obviously, arithmetic operators don't operate on strings. However, there are special operators for string processing.

| Operator | Description | Example |
| --- | --- | --- |
| + | Appends the second string to the first | >>> a='hello'  >>> b='world'  >>> a+b  'helloworld' |
| \* | Concatenates multiple copies of the same string | >>> a='hello' >>> a\*3 'hellohellohello' |
| [] | Returns the character at the given index | >>> a = 'Python Tutorials' >>> a[7]  T |
| [ : ] | Fetches the characters in the range specified by two index operands separated by the : symbol | >>> a = 'Python Tutorials' >>> a[7:15] 'Tutorial' |
| in | Returns *true* if a character exists in the given string | >>> a = 'Python Tutorials' >>> 'X' in a  False  >>> 'Python' in a  True  >>> 'python' in a  False |
| not in | Returns *true* if a character does not exist in the given string | >>> a = 'Python Tutorials' >>> 'X' not in a  True  >>> 'Python' not in a  False |

## Converting to String

Python has an in-built function str() which returns a printable string representation of any object. In the previous chapter we have used int(), float() and complex() functions. They convert the string representation into integer, float and complex numbers, respectively. The str() function converts any number to a string object.

>>> str(12)  
'12'  
>>> str(6+5j)  
'(6+5j)'  
>>> str(1.11)  
'1.11'

## String Formatting

Interpolation of objects of different types at placeholders inside a string is called string formatting. The % operator (otherwise an arithmetic operator used to return the remainder of division) is used to perform string formatting also. Format specification symbols (%d, %c, %f, %s, etc) used in C language are utilized as placeholders in a string.

In the following example, name is a string and age is an integer variable. Their values are inserted in the string with %s and %d format specification symbols, respectively. These symbols are interpolated to values in a tuple with the % operator in front.

>>> name="Bond"  
>>> "My name is %s." % name  
'My name is Bond.'

You can have multiple parameters too.

>>> name="Bond"  
>>> age=30  
>>> "My name is %s and age is %d years." % (name, age)  
'My name is Bond and age is 30 years.'

All C style format specification symbols are permitted.

| Format Symbol | Conversion |
| --- | --- |
| %c | character |
| %s | string conversion via str() prior to formatting |
| %i | signed decimal integer |
| %d | signed decimal integer |
| %u | unsigned decimal integer |
| %o | octal integer |
| %x / %X | hexadecimal integer (lowercase letters) |
| %e / %E | exponential notation (with lowercase 'e') |
| %f | floating point real number |

You can specify the width of integer and float objects. Here, integers a, b and c will occupy the width of 3 characters in the formatted string. Additional spaces will be padded to the left.

>>> a=1  
>>> b=11  
>>> c=111  
>>> "a=%3d b=%3d c=%3d" % (a, b, c)  
'a=  1 b= 11 c=111'

The following specifies the width of the float variable.

>>> percent=55.50  
>>> "%5.2f" % percent  
'55.50'  
>>> "%6.2f" % percent  
' 55.50'  
>>> "%6.3f" % percent  
'55.500'  
>>> "%7.3f" % percent  
' 55.500'

In the above example, %5.2 specifies the width of float, where 5 is for total characters and 2 is for decimals. So, the result would be '55.50'.

The width of a string can also be specified. The default alignment is right. For left alignment, give a negative sign to width.

>>>'%4s' % 'abc'   
>>>' abc'   
  
>>>'%6s' % 'abc'   
>>>' abc'   
  
>>>'%-6s' % 'abc'   
>>>'abc '   
  
>>>a='abc'   
>>>'%-6s' % a   
>>>'abc '

### format() method

The format() method can handle complex string formatting more efficiently. This method of in-built string class provides the ability to do complex variable substitutions and value formatting. This new formatting technique is regarded as more elegant. The general syntax of the format() method is as follows:

string.format(str1, str2,...)

The string itself contains placeholders {}, in which the values of variables are successively inserted.

>>>name="Bill"  
>>>age=25  
>>>"My name is {} and I am {} years old.".format(name, age)  
'My name is Bill and I am 25 years old.'   
  
>>>myStr = "My name is {} and I am {} years old."  
>>>myStr.format(name, age)  
'my name is Bill and I am 25 years old.'

You can also specify formatting symbols by using : instead of %. For example, instead of %s use {:s} and instead of %d use {:d}.

>>> "My name is {:s} and I am {:d} years old.".format(name, age)  
'My name is Bill and I am 25 years old.'

Precision formatting of numbers can be done accordingly.

>>> percent=55.50  
>>> "I have scored {:6.3f} percent marks.".format(percent)  
'I have scored 55.500 percent marks.'

String alignment is done with <, > and ^ symbols in the place holder causing left, right and center alignment, respectively. Default is left alignment.

>>> '{:>10}'.format('test')  
'      test'  
>>> '{:<10}'.format('test')  
'test      '  
>>> '{:^10}'.format('test')  
'   test   '

## Built-in String Methods

### capitalize():

Converts the first character of a string to uppercase letters.

>>> mystr='python'  
>>> mystr.capitalize()  
'Python'

### upper()

Replaces the lowercase characters in a string with corresponding uppercase characters.

>>> mystr='Python'  
>>> mystr.upper()  
'PYTHON'

### lower()

Replaces the uppercase characters in a string with corresponding lowercase characters.

>>> mystr='PYTHON'  
>>> mystr.lower()  
'python'

### title():

Returns the string with the first character of each word converted to uppercase.

>>> mystr='python tutorial from tutorials teacher'  
>>> mystr.title()  
'Python Tutorial From Tutorials Teacher'

### find()

The find() method finds the first occurrence of a substring in another string. If not found, the method returns -1.

>>> mystr='Python Tutorial From Tutorials Teacher'  
>>> mystr.find('From')  
16  
>>> mystr.find('xyz')  
-1

Substring 'From' first occurs at position 16 (the count starts from 0). 'xyz' is not found, hence it returns -1.

### count()

The count() method returns the number of occurrences of a substring in the given string.

>>> mystr='Python Tutorial From Tutorials Teacher'  
>>> mystr.count('Tutorial')  
2

### isalpha()

The isalpha() method returns true if all the characters in a string are alphabetic letters (a-z or A-Z), otherwise it returns false.

>>> mystr='TutorialsTeacher'  
>>> mystr.isalpha()  
True  
>>> mystr='Tutorials Teacher'  
>>> mystr.isalpha()  
False

### isdigit()

The isdigit() is method returns true if all the characters in string are digits (0-9), if not, it returns false.

>>> str1='2000'  
>>> str1.isdigit()  
True  
>>> str2='2,000'  
>>> str2.isdigit()  
False

### islower()

The islower() method returns true if all the characters in the string are lowercase characters, else it returns false.

>>> str1='python'  
>>> str1.islower()  
True  
>>> str2='Python'  
>>> str2.islower()  
False

### isupper()

The isupper() method returns true if all the characters in the string are uppercase characters, else it returns false.

>>> var='TUTORIALSTEACHER'  
>>> var.isupper()  
True  
>>> var='TUTORIALSTeacher'  
>>> var.isupper()  
False

# Python - List

In Python, the list is a collection of items of different data types. It is an ordered sequence of items. A list object contains one or more items, not necessarily of the same type, which are separated by comma and enclosed in square brackets []. It is mutable.

Syntax:

list = [value1, value2, value3,...valueN]

The following declares a list type variable.

>>> names=["Jeff", "Bill", "Steve", "Mohan"]

**A list can also contain elements of different types.**

>>> orderItem=[1, "Jeff", "Computer", 75.50, True]

The above list orderItem includes five elements. Each individual element in the sequence is accessed by the index in the square brackets []. An index starts with zero, as shown below.

>>> orderItem=[1, "Jeff", "Computer", 75.50, True]  
>>> orderItem[0]  
1   
>>> orderItem[1]  
'Jeff'   
>>> orderItem[2]  
'Computer'   
>>> orderItem[3]  
75.50   
>>> orderItem[4]  
True

The list object is mutable. It is possible to modify its contents, which will modify the value in the memory.

For instance, item at index 2 in orderItem can be modified as shown below.

>>> orderItem=[1, "Jeff", "Computer", 75.50, True]  
>>> orderItem[2]="Laptop"  
>>> orderItem  
[1, "Jeff", "Laptop", 75.50, True]

It will throw an error "index out of range" if the element at the specified index does not exist.

>>> orderItem=[1, "Jeff", "Computer", 75.50, True]  
>>> orderItem[5]  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>   
IndexError: list index out of range

Lists are generally used to store homogenous collections, i.e. items of similar types.

>>> languages=['Python', 'Java', 'C#', 'PHP']  
>>> temperatures=[56, 65, 78]

Use the *del* keyword to delete the list object.

>>> del languages  
>>> languages  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>   
NameError: name 'languages' is not defined

## List Operators

Like the string, the list is also a sequence. Hence, the operators used with strings are also available for use with the list (and tuple also).

| Operator | Description | Example |
| --- | --- | --- |
| + Concatenation | Returns a list containing all the elements of the first and the second list. | >>> L1=[1,2,3] >>> L2=[4,5,6]  >>> L1+L2  [1, 2, 3, 4, 5, 6] |
| \* Repetition | Concatenates multiple copies of the same list. | >>> L1\*4  [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3] |
| [] slice | Returns the item at the given index. A negative index counts the position from the right side. | >>> L1=[1, 2, 3, 4, 5, 6] >>> L1[3]  4  >>> L1[-2]  5 |
| [ : ] - Range slice | Fetches items in the range specified by the two index operands separated by : symbol.  If the first operand is omitted, the range starts from the zero index. If the second operand is omitted, the range goes up to the end of the list. | >>> L1=[1, 2, 3, 4, 5, 6] >>> L1[1:4]  [2, 3, 4]  >>> L1[3:]  [4, 5, 6]  >>> L1[:3]  [1, 2, 3] |
| in | Returns true if an item exists in the given list. | >>> L1=[1, 2, 3, 4, 5, 6] >>> 4 in L1  True  >>> 10 in L1  False |
| not in | Returns true if an item does not exist in the given list. | >>> L1=[1, 2, 3, 4, 5, 6] >>> 5 not in L1  False  >>> 10 not in L1  True |

## Built-in List Methods

### len()

The len() method returns the number of elements in the list/tuple.

>>> L1=[12,45,43,8,35]  
>>> len(L1)  
5

### max()

The max() method returns the largest number, if the list contains numbers. If the list contains strings, the one that comes last in alphabetical order will be returned.

>>> L1=[12,45,43,8,35]  
>>> max(L1)  
45  
>>> L2=['Python', 'Java', 'C++']  
>>> max(L2)  
'Python'

### min()

The min() method returns the smallest number, if the list contains numbers. If the list contains strings, the one that comes first in alphabetical order will be returned.

>>> L1=[12, 45, 43, 8, 35]  
>>> min(L1)  
8  
>>> L2=['Python', 'Java', 'C++']  
>>> min(L2)  
'C++'

Note that if a list object contains strings as well as numeric items, max() and min() functions throw an error.

>>> L1=[1,'aa',12.22]  
>>> max(L1)  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>   
TypeError: '>' not supported between instances of 'str' and 'int'

### append()

Adds an item at the end of the list.

>>> L2=['Python', 'Java', 'C++']  
>>> L2.append('PHP')  
>>> L2  
['Python', 'Java', 'C++', 'PHP']

### insert()

Inserts an item in a list at the specified index.

>>> L2=['Python', 'Java', 'C++']  
>>> L2.insert(1,'Perl')  
>>> L2  
['Python', 'Perl', 'Java', 'C++']

### remove()

Removes a specified object from the list.

>>> L2=['Python', 'Perl', 'Java', 'C++']  
>>> L2.remove('Java')  
>>> L2  
['Python', 'Perl', 'C++']

### pop()

Removes and returns the last object in the list.

>>> L2=['Python', 'Perl', 'Java', 'C++']  
>>> L2.pop()  
'C++'  
>>> L2  
['Python', 'Perl', 'Java']

### reverse()

Reverses the order of the items in a list.

>>> L2=['Python', 'Perl', 'Java', 'C++']  
>>> L2.reverse()  
>>> L2  
['C++', 'Java', 'Perl', 'Python']

### sort()

Rearranges the items in the list according to the alphabetical order. Default is the ascending order. For descending order, put reverse=True as an argument in the function bracket.

>>> L2=['Python', 'C++', 'Java', 'Ruby']  
>>> L2.sort()  
>>> L2  
['C++', 'Java', 'Python', 'Ruby']  
>>> L2.sort(reverse=True)  
>>> L2  
['Ruby', 'Python', 'Java', 'C++']

The following utility functions help in converting one sequence data type to another.

### list()

Converts a tuple or string to a list object.

>>> t2=('python', 'java', 'C++')  
>>> list(t2)  
['python', 'java', 'C++']  
>>> s1="TutorialsTeacher"  
>>> list(s1)  
['T', 'u', 't', 'o', 'r', 'i', 'a', 'l', 's', 'T', 'e', 'a', 'c', 'h', 'e', 'r']

# Python - Tuple

Tuple is a collection of items of any Python data type, same as the list type. Unlike the list, tuple is immutable.

The tuple object contains one or more items, of the same or different types, separated by comma and enclosed in parentheses ().

Syntax:

tuple = (value1, value2, value3,...valueN)

The following declares a tuple type variable.

>>> names=("Jeff", "Bill", "Steve", "Mohan")

The tuple type can also contain elements of different types.

>>> orderItem=(1, "Jeff", "Computer", 75.50, True)

It is however not necessary to enclose the tuple elements in parentheses. The tuple object can include elements separated by comma without parentheses.

>>> names="Jeff", "Bill", "Steve", "Mohan"   
>>> type(names)   
<class 'tuple'>

The above tuple object orderItem includes five elements. Each individual element in the sequence is accessed by the index in the square brackets []. An index starts with zero, as shown below.

>>> orderItem=(1, "Jeff", "Computer", 75.50, True)  
>>> orderItem[0]  
1   
>>> orderItem[1]  
'Jeff'   
>>> orderItem[2]  
'Computer'   
>>> orderItem[3]  
75.50   
>>> orderItem[4]  
True

You can use backward indexing also.

>>> orderItem=(1, "Jeff", "Computer", 75.50, True)  
>>> orderItem[-5]  
1   
>>> orderItem[-4]  
'Jeff'   
>>> orderItem[-3]  
'Computer'   
>>> orderItem[-2]  
75.50   
>>> orderItem[-1]  
True

If the element at the specified index does not exist, error "index out of range" will be returned.

>>> orderItem=(1, "Jeff", "Computer", 75.50, True)  
>>> orderItem[5]  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>   
IndexError: tuple index out of range

Tuple is immutable. So, once a tuple is created, any operation that seeks to change its contents is not allowed.

For instance, item orderItem cannot be modified and an attempt to do so will result in an error.

>>> orderItem=(1, "Jeff", "Computer", 75.50, True)  
>>> orderItem[2]="Laptop"  
TypeError: 'tuple' object does not support item assignment

Tuple is typically used to represent a data structure which may contain objects of different types.

>>> date=(2018, June, 15)  
>>> student=('Kapil', 'A001', 23, 78.75, 'M', True)

Use the del keyword to delete the tuple object.

>>> del student  
>>> student  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>   
NameError: name 'student' is not defined

## Tuple Operators

Like string, tuple objects are also a sequence. Hence, the operators used with strings are also available for tuple.

| Operator | Description | Example |
| --- | --- | --- |
| + Concatenation | Returns a tuple containing all the elements of the first and the second tuple object. | >>> t1=(1,2,3)  >>> t2=(4,5,6)  >>> t1+t2  (1, 2, 3, 4, 5, 6)  >>> t2+(7,)  (4, 5, 6, 7) |
| \* Repetition | Concatenates multiple copies of the same tuple. | >>> t1=(1,2,3)  >>> t1\*4  (1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3) |
| [] slice | Returns the item at the given index. A negative index counts the position from the right side. | >>> t1=(1,2,3,4,5,6)  >>> t1[3]  4  >>> t1[-2]  5 |
| [ : ] - Range slice | Fetches the items in the range specified by two index operands separated by the : symbol.  If the first operand is omitted, the range starts at zero index. If the second operand is omitted, the range goes up to the end of tuple. | >>> t1=(1,2,3,4,5,6)  >>> t1[1:3]  (2, 3)  >>> t1[3:]  (4, 5, 6)  >>> t1[:3]  (1, 2, 3) |
| in | Returns true if an item exists in the given tuple. | >>> t1=(1,2,3,4,5,6)  >>> 5 in t1  True  >>> 10 in t1  False |
| not in | Returns true if an item does not exist in the given tuple. | >>> t1=(1,2,3,4,5,6)  >>> 4 not in t1  False  >>> 10 not in t1  True |

## Built-in Tuple Methods

### len()

Returns the number of elements in the tuple.

>>> t1=(12,45,43,8,35)  
>>> len(t1)  
5

### max()

If the tuple contains numbers, the heighest number will be returned. If the tuple contains strings, the one that comes last in alphabetical order will be returned.

>>> t1=(12, 45, 43, 8, 35)  
>>> max(t1)  
45  
>>> t2=('python', 'java', 'C++')  
>>> max(t2)  
'python'

### min()

If the tuple contains numbers, the lowest number will be returned. If the tuple contains strings, the one that comes first in alphabetical order will be returned.

>>> t1=(12,45,43,8,35)  
>>> min(t1)  
8  
>>> t2=('python', 'java', 'C++')  
>>> min(t2)  
'C++'

Note that if the tuple object contains strings as well as numeric items, max() and min() functions throw error.

>>> t1=(1,'Test', 12.22)  
>>> max(t1)  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>   
TypeError: '>' not supported between instances of 'str' and 'int'

# Python - Set

A set is a collection of data types in Python, same as the [list](https://www.tutorialsteacher.com/python/python-list) and [tuple](https://www.tutorialsteacher.com/python/python-tuple). However, it is not an ordered collection of objects. The set is a Python implementation of the set in Mathematics. A set object has suitable methods to perform mathematical set operations like union, intersection, difference, etc.

A set object contains one or more items, not necessarily of the same type, which are separated by comma and enclosed in curly brackets {}.

Syntax:

set = {value1, value2, value3,...valueN}

The following defines a set object.

>>> S1={1, "Bill", 75.50}

A set doesn't store duplicate objects. Even if an object is added more than once inside the curly brackets, only one copy is held in the set object. Hence, indexing and slicing operations cannot be done on a set object.

>>> S1={1, 2, 2, 3, 4, 4, 5, 5}  
>>> S1  
{1, 2, 3, 4, 5}

## set() function

Python has an in-built function set(), using which a set object can be constructed out of any sequence such as a string, list or a tuple object.

>>> s1=set("Python")   
>>> s1  
{'t', 'h', 'o', 'n', 'P', 'y'}  
>>> s2=set([45,67,87,36, 55])  
>>> s2  
{67, 36, 45, 87, 55}  
>>> s3=set((10,25,15))  
>>> s3  
{25, 10, 15}

The order of elements in the set is not necessarily the same as the order given at the time of assignment. Python optimizes the structure of a set for performing operations over it, as defined in mathematics.

Only immutable (and hashable) objects can be a part of a set object. Numbers (integer, float, as well as complex), strings, and tuple objects are accepted, but list and dictionary objects are not.

>>> S1={(10,10), 10,20}  
>>> S1  
{10, 20, (10, 10)}  
>>> S2={[10,10], 10,20}  
TypeError: unhashable type: 'list'

In the above example, (10,10) is a tuple, hence it becomes part of the set. However, [10,10] is a list, hence an error message is displayed saying that the list is unhashable. ([Hashing](https://en.wikipedia.org/wiki/Hash_function) is a mechanism in computer science which enables quicker search of objects in the computer's memory.)

Even though mutable objects are not stored in a set, the set itself is a mutable object.

## Set Operations

As mentioned earlier, the set data type in Python implements as the set defined in mathematics. Various set operations can be performed. Operators |, &, - and ^ perform union, intersection, difference and symmetric difference operations, respectively. Each of these operators has a corresponding method associated with the built-in set class.

| Operation | Operator/Method | Example |
| --- | --- | --- |
| The union of two sets is a set of all elements from both the collections. | | | >>> s1={1,2,3,4,5} >>> s2={4,5,6,7,8} >>> s1|s2  {1, 2, 3, 4, 5, 6, 7, 8} |
| union() | >>> s1={1,2,3,4,5} >>> s2={4,5,6,7,8} >>> s1.union(s2)  {1, 2, 3, 4, 5, 6, 7, 8}  >>> s2.union(s1)  {1, 2, 3, 4, 5, 6, 7, 8} |
| The intersection of two sets is a set containing elements common to both collections. | & | >>> s1={1,2,3,4,5} >>> s2={4,5,6,7,8} >>> s1&s2  {4, 5}  >>> s2&s1  {4, 5} |
| intersection() | >>> s1={1,2,3,4,5}  >>> s2={4,5,6,7,8}  >>> s1.intersection(s2)  {4, 5}  >>> s2.intersection(s1)  {4, 5} |
| The difference of two sets results in a set containing elements only in the first set, but not in the second set. | - | >>> s1={1,2,3,4,5} >>> s2={4,5,6,7,8} >>> s1-s2  {1, 2, 3}  >>> s2-s1  {8, 6, 7} |
| difference() | >>> s1={1,2,3,4,5}  >>> s2={4,5,6,7,8}  >>> s1.difference(s2)  {1, 2, 3}  >>> s2.difference(s1)  {8, 6, 7} |
| Symmetric Difference: the result of symmetric difference is a set consisting of elements in both sets, excluding the common elements. | ^ | >>> s1={1,2,3,4,5} >>> s2={4,5,6,7,8} >>> s1^s2  {1, 2, 3, 6, 7, 8}  >>> s2^s1  {1, 2, 3, 6, 7, 8} |
| Symmetric\_difference() | >>> s1={1,2,3,4,5}  >>> s2={4,5,6,7,8}  >>> s1.symmetric\_difference(s2) {1, 2, 3, 6, 7, 8}  >>> s2.symmetric\_difference(s1) {1, 2, 3, 6, 7, 8} |

## Built-in Set Methods

### add()

Adds a new element in the set object.

>>> S1   
{'Java', 'Python', 'C++'}   
>>> S1.add("Perl")   
>>> S1   
{'Java', 'Python', 'Perl', 'C++'}

### update()

Adds multiple items from a list or a tuple.

>>> S1={"Python", "Java", "C++"}  
>>> S1.update(["C", "Basic"])   
>>> S1   
{'C++', 'Java', 'Python', 'Basic', 'C'}   
>>> S1.update(("Ruby", "PHP"))   
>>> S1   
{'C++', 'Ruby', 'Java', 'PHP', 'Python', 'Basic', 'C'}

### clear()

Removes the contents of set object and results in an empty set.

>>> S1.clear()  
>>> S1   
set()

### copy()

Creates a copy of the set object.

>>> S1={"Python", "Java", "C++"}  
>>> S2=S1.copy()   
>>> S2   
{'Java', 'Python', 'C++'}

### discard()

Returns a set after removing an item from it. No changes are done if the item is not present.

>>> S1={"Python", "Java", "C++"}  
>>> S1.discard("Java")   
>>> S1   
{'Python', 'C++'}   
>>> S1={"Python", "Java", "C++"}

>>> S1.discard("SQL")   
>>> S1   
{'Java', 'Python', 'C++'}

### remove()

Returns a set after removing an item from it. Results in an error if the item is not present.

>>> S1={"Python", "Java", "C++"}  
>>> S1.remove("C++")   
>>> S1   
{'Java', 'Python'}   
>>> S1={"Python", "Java", "C++"}  
>>> S1.remove("SQL")   
KeyError: 'SQL'

# Python - Dictionary

Like the list and the tuple, dictionary is also a collection type. However, it is not an ordered sequence, and it contains key-value pairs. One or more key:value pairs separated by commas are put inside curly brackets to form a dictionary object.

Syntax:

dict = { key1:value1, key2:value2,...keyN:valueN }

The following declares a dictionary object.

>>> capitals={"USA":"Washington, D.C.", "France":"Paris", "India":"New Delhi"}

In the above example, capitals is a dictionary object. The left side of : is a key and right side of : is a value. The key should be an immutable object. A number, string or tuple can be used as key. Hence, the following definitions of dictionary are also valid:

>>> numNames={1:"One", 2: "Two", 3:"Three"}  
>>> decNames={1.5:"One and Half", 2.5: "Two and Half", 3.5:"Three and Half"}  
>>> items={("Parker","Reynolds","Camlin"):"pen",("LG","Whirlpool","Samsung"): "Refrigerator"}

However, a dictionary with a list as a key is not valid, as the list is mutable:

>>>dict={["Mango","Banana"]:"Fruit", ["Blue", "Red"]:"Colour"}  
TypeError: unhashable type: 'list'

But, a list can be used as a value.

>>>dict={"Fruit":["Mango","Banana"], "Colour":["Blue", "Red"]}

The same key cannot appear more than once in a collection. If the key appears more than once, only the last will be retained. The value can be of any data type. One value can be assigned to more than one key.

>>>staff={"Krishna":"Officer", "Steve":"Manager", "John":"officer", "Anil":"Clerk", "John":"Manager"}  
>>>staff  
{"Krishna":"Officer", "Steve":"Manager", "Anil":"Clerk", "John":"Manager"}

In above example the same value parameter is used more than once. However, when key 'John' is assigned two different values, only the latest is retained, overwriting the previous value.

## Accessing a Dictionary

Dictionary is not an ordered collection, so a value cannot be accessed using an index in square brackets. A value in a dictionary can be fetched using the associated key, using the get() method. Specify the key in the get() method to retrieve its value.

>>>capitals={"USA":"New York", "France":"Paris", "Japan":"Tokyo", "India":"New Delhi"}  
>>>capitals.get("France")  
'Paris'  
>>>points={"p1":(10,10), "p2":(20,20)}  
>>>points.get("p2")  
(20,20)  
>>>numbers={1:"one", 2:"Two", 3:"three",4:"four"}  
>>>numbers.get(2)  
'Two'

Use the for loop to iterate a dictionary in the Python script.

capitals={"USA":"Washington, D.C.", "France":"Paris", "Japan":"Tokyo", "India":"New Delhi"}

for key in capitals:

print("Key = " + key + ", Value = " + capitals[key])

Output

Key = 'USA', Value = 'Washington, D.C.'  
Key = 'France', Value = 'Paris'   
Key = 'Japan', Value = 'Tokyo'   
Key = 'India', Value = 'New Delhi'

## Updating a Dictionary

As mentioned earlier, the key cannot appear more than once. Use the same key and assign a new value to it to update the dictionary object.

>>> captains={"England":"Root", "Australia":"Smith", "India":"Dhoni"}  
>>> captains['India']='Virat'  
>>> captains['Australia']='Paine'  
>>> captains  
{'England': 'Root', 'Australia': 'Paine', 'India': 'Virat'}

Use a new key and assign a value to it. The dictionary will show an additional key-value pair in it.

>>> captains['SouthAfrica']='Plessis'  
>>> captains  
{'England': 'Root', 'Australia': 'Paine', 'India': 'Virat', 'SouthAfrica': 'Plessis'}

## Deleting Values from a Dictionary

Use the **del** keyword to delete a pair from a dictionary or the dictionary object itself. To delete a pair, use its key as parameter. To delete a dictionary object, use its name.

>>> captains={'England': 'Root', 'Australia': Paine', 'India': 'Virat', 'Srilanka': 'Jayasurya'}  
>>> del captains['Srilanka']  
>>> captains  
{'England': 'Root', 'Australia': Paine', 'India': 'Virat'}  
>>> del captains  
>>> captains  
NameError: name 'captains' is not defined

NameError indicates that the dictionary object has been removed from memory.

## View Keys and Values

The keys() and values() methods of Python dictionary class return a view object consisting of keys and values respectively, used in the dictionary.

>>> d1 = {'name': 'Steve', 'age': 21, 'marks': 60, 'course': 'Computer Engg'}  
>>>d1.keys()  
dict\_keys(['name', 'age', 'marks', 'course'])

The result of the keys() method is a view which can be stored as a list object. If a new key-value pair is added, the view object is automatically updated.

>>> keys=d1.keys()  
>>> keys  
dict\_keys(['name', 'age', 'marks', 'course'])  
>>>d1.update({"college":"IITB"})  
>>> keys  
dict\_keys(['name', 'age', 'marks', 'course', 'college'])

This is similar for the values() method.

>>> d1= {'name': 'Steve', 'age': 21, 'marks': 60, 'course': 'Computer Engg'}  
>>>values=d1.values()  
dict\_values(['Steve', 21, 60, 'Computer Engg'])<

The result of the values() method is a view which can be stored as a list object. If a new key-value pair is added, the view is dynamically updated.

>>> d1.update({"college":"IITB"})  
>>> values  
dict\_values(['Steve', 21, 60, 'Computer Engg', 'IITB'])<

## Multi-dimensional Dictionary

Let's assume there are three dictionary objects, as below:

>>> d1={"name":"Steve","age":25, "marks":60}  
>>> d2={"name":"Anil","age":23, "marks":75}  
>>> d3={"name":"Asha", "age":20, "marks":70}<

Let us assign roll numbers to these students and create a multi-dimensional dictionary with roll number as key and the above dictionaries at their value.

>>> students={1:d1,2:d2,3:d3}  
>>> students  
{1: {'name': 'Steve', 'age': 25, 'marks': 60}, 2: {'name': 'Anil', 'age': 23, 'marks': 75}, 3: {'name': 'Asha', 'age': 20, 'marks': 70}}<

The students object is a two-dimensional dictionary. Here d1, d2 and d3 are assigned as values to keys 1,2, and 3, respectively. students [1] returns d1.

>>> students[1]  
{'name': 'Steve', 'age': 25, 'marks': 60}

The value of a key inside d1 can be obtained as below:

>>> students[1]['age']  
25

## Built-in Dictionary Methods

### len()

Returns the number of key:value pairs in the dictionary.

>>> lang={'A':('Ada','ActionScript'), 'P':("Python", "Perl","PHP")}  
>>> len(lang)  
2

### max()

If all keys in the dictionary are numbers, the heighest number will be returned. If all keys in the dictionary are strings, the one that comes last in alphabetical order will be returned.

>>> lang={'J':'Java', 'A': 'ActionScript', 'P':'Python'}  
>>> max(lang)  
'P'  
>>> num={5:"five", 100:"hundred",3:"three"}  
>>> max(num)  
100

### min()

If all keys in the dictionary are numbers, the lowest number will be returned. If all keys in the dictionary are strings, the one that comes first in alphabetical order will be returned.

>>> lang={'J':'Java', 'A': 'ActionScript', 'P':'Python'}  
>>> min(lang)  
'A'  
>>> num={5:"five", 100:"hundred",3:"three"}  
>>> min(num)  
3

### pop()

Returns the value associated with the key and the corresponding key-value pair is removed.

>>> captains={'England': 'Root', 'Australia': 'Smith', 'India': 'Virat', 'Pakistan': 'Sarfraz'}  
>>> captains.pop('India')  
'Virat'  
>>> captains  
{'England': 'Root', 'Australia': 'Smith', 'Pakistan': 'Sarfraz'}

### clear()

Returns empty object by deleting all the key-value pairs.

>>> captains  
{'England': 'Root', 'Australia': 'Smith', 'Pakistan': 'Sarfraz'}  
>>> captains.clear()  
>>> captains  
{}

### items() :

Returns a list of tuples, each tuple containing the key and value of each pair.

>>> captains={'England': 'Root', 'Australia': 'Smith', 'India': 'Virat', 'Pakistan': 'Sarfraz'}  
>>> captains.items()  
dict\_items([('England', 'Root'), ('Australia', 'Smith'), ('India', 'Virat'), ('Pakistan', 'Sarfraz')])

### keys()

Returns a list object comprising of the keys in the dictionary.

>>> captains={'England': 'Root', 'Australia': 'Smith', 'India': 'Virat', 'Pakistan': 'Sarfraz'}  
>>> captains.keys()  
dict\_keys(['England', 'Australia', 'India', 'Pakistan'])

### values():

Returns a list object comprising of the values in the dictionary.

>>> captains={'England': 'Root', 'Australia': 'Smith', 'India': 'Virat', 'Pakistan': 'Sarfraz'}  
>>> captains.values()  
dict\_values(['Root', 'Smith', 'Virat', 'Sarfraz'])

### update()

Adds key-value pairs from the second dictionary object to the first. If the second dictionary contains a key already used in first dictionary object, its value is updated.

>>> mark1={"Sagar":67, "Amrita":88, "Bhaskar":91, "Kiran":49}  
>>> mark2={"Arun":55, "Bhaskar":95, "Geeta":78}  
>>> mark1.update(mark2)  
>>> mark1  
{'Sagar': 67, 'Amrita': 88, 'Bhaskar': 95, 'Kiran': 49, 'Arun': 55, 'Geeta': 78}

mark1 dictionary now has new items from mark2, with the value of one key updated.