# Apply Python

## LU3 – Develop Python Application

**NDAYISENGA Gilbert**
UNIVERSITY OF KIGALI

## Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

## Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

| Operator | Name | Example |
|---|---|---|
| + | Addition | x + y |
| - | Subtraction | x - y |
| * | Multiplication | x * y |
| / | Division | x / y |

| | | |
|---|---|---|
| % | Modulus | x % y |
| ** | Exponentiation | x ** y |
| // | Floor division | x // y |

## Python Assignment Operators

Assignment operators are used to assign values to variables:

| Operator | Example | Same as |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| //= | x //= 3 | x = x // 3 |
| **= | x **= 3 | x = x ** 3 |
| &= | x &= 3 | x = x & 3 |
| \|= | x \|= 3 | x = x \| 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |

| | | |
|---|---|---|
| <<= | x <<= 3 | x = x << 3 |

## Python Comparison Operators

Comparison operators are used to compare two values:

| Operator | Name | Example |
|---|---|---|
| == | Equal | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

## Python Logical Operators

Logical operators are used to combine conditional statements:

| Operator | Description | Example |
|---|---|---|
| and | Returns True if both statements are true | x < 5 and x < 10 |
| or | Returns True if one of the statements is true | x < 5 or x < 4 |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x < 10) |

## Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

| Operator | Description | Example |
|---|---|---|
| is | Returns True if both variables are the same object | x is y |
| is not | Returns True if both variables are not the same object | x is not y |

## Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

| Operator | Description | Example |
|---|---|---|
| in | Returns True if a sequence with the specified value is present in the object | x in y |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y |

## Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

| Operator | Name | Description | Example |
|---|---|---|---|
| & | AND | Sets each bit to 1 if both bits are 1 | x & y |
| \| | OR | Sets each bit to 1 if one of two bits is 1 | x \| y |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 | x ^ y |
| ~ | NOT | Inverts all the bits | ~x |
| << | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off | x << 2 |
| >> | Signed right shift | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off | x >> 2 |

## Operator Precedence

Operator precedence describes the order in which operations are performed.

## Example

Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

```
print ((4 + 7) - (8 + 1))
```

Multiplication * has higher precedence than addition +, and therefor multiplications are evaluated before additions:

```
print (10 + 15 * 2)
```

The precedence order is described in the table below, starting with the highest precedence at the top:

| **Operator** | **Description** |
|---|---|
| () | Parentheses |
| ** | Exponentiation |
| +x, -x, ~x | Unary plus, unary minus, and bitwise NOT |
| *, /, //, % | Multiplication, division, floor division, and modulus |
| +, - | Addition and subtraction |
| <<, >> | Bitwise left and right shifts |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| \| | Bitwise OR |
| ==, !=, >, >=, <, <=; **is, is not; in, not in** | Comparisons; identity; and membership operators |
| not | Logical NOT |

| | |
|---|---|
| and | AND |
| or | OR |

Note: **If two operators have the same precedence, the expression is evaluated from left to right.**

**Example**

Addition + and subtraction - has the same precedence, and therefore we evaluate the expression from left to right:

```
print (5 + 6 - 10 + 2)
```

## Python Lists, Tuples, Sets and Dictionaries

### 1. Lists

Lists provide a general mechanism for storing a collection of objects indexed by a number in python. The elements of the list are arbitrary — they can be numbers, strings, functions, user-defined objects or even other lists, making complex data structures very simple to express in python. You can input a list to the python interpreter by surrounding a comma separated list of the objects you want in the list with square brackets ([ ]) Thus, a simple list of numbers can be created as follows:

```
>>> mylist = [1,7,9, 13, 22, 31]
```

Python ignores spaces between the entries of a list. If you need to span multiple lines with a list entry, you can simply hit the return key after any comma in the list:

```
>>> newlist = [7, 9, 12, 15, ... 17,19,103]
```

Note that the python interpreter changes its prompt when it recognizes a continuation line, and that no indentation is necessary to continue a line like this. Inside a script, your input can also be broken up after commas in a similar fashion. To create an empty list, use square brackets with no elements in between them ([]).

The elements of a list need not be of the same type. The following list contains numeric, string and list data, along with a function:

```
>>> mixlist = [7,'dog','tree',[1,5,2,7],abs]
```

Since the token abs was entered in the list without quotes, python interprets it as a named object; in this case it represents the builtin function abs. Since functions and many other objects can't be displayed in the same fashion as a number or a string, python uses a special notation to display them:

```
>>> mixlist
>>>[7, 'dog', 'tree', [1, 5, 2, 7], <built-in function abs>]
```

The angle brackets surrounding the phrase "built-in function abs" indicate that that element of the list is a special object of some sort.

To access the individual elements of a list, use square brackets after the list's name surrounding the index of the desired element. Recall that the first element of a sequence in python is numbered zero. Thus, to extract the abs function from mylist in the example above, we would refer to element 4, i.e. mylist[4]:

```
>>> mixlist[4](-5)
>>>5
```

This example shows that once an element is extracted from a list, you can use it in any context in which the original element could be used; by providing an argument in parentheses to mixlist[4], we call the function abs which was stored in that position in the list. Furthermore, when accessing lists inside of lists, you can simply use additional sets of square brackets to access individual elements:

```
>>> nestlist = [1,2,[10,20,30,[7,9,11,[100,200,300]]],[1,7,8]]
>>> nestlist[2]
>>> [10, 20, 30, [7, 9, 11, [100, 200, 300]]]
>>> nestlist[2][3]
>>> [7, 9, 11, [100, 200, 300]]
>>> nestlist[2][3][3]
>>> [100, 200, 300]
>>> nestlist[2][3][3][0]
>>>100
```

While this is a competely artificial example, it shows that you can nest lists to any level you choose, and that the individual elements of those lists are always extracted in a simple, consistent way.

## ➢ *LIST INDEXING AND SLICING*

You may notice a similarity between the way you access elements in a list, and the way you access individual characters in a character string. This is because both strings and lists are examples of python sequences, and they behave consistently. For example, to find out the number of elements in a list, you can use the built-in function len, just as you would to find the number of characters in a character string. Calling len with a non-sequence argument result in a TypeError exception, however.

```
>>> len(mixlist)
>>>5
>>> len(mixlist[3])
>>>4
>>> len(mixlist[1])
>>>3
>>> len(mixlist[0])
>>>Traceback (innermost last):
    File "<stdin>", line 1, in ?
TypeError: len() of unsized object
```

In the first case, calling len with the argument mixlist returns the number of elements in the list, namely 5. Similarly, refering to mixlist[3], corresponding to the list [1, 5, 2, 7] returns 4, the number of elements in that list. Calling len with mixlist[1] (which is the string “dog”) returns the number of characters in the string, but calling len with the scalar argument mixlist[0] (which is the integer 7), results in an exception.

To convert a string into a list, making each character in the string a separate element in the resulting list, use the list function.

You can set the value of elements in a list by using a slice on the left-hand side of an equal sign. In python terminology, this is because lists are mutable objects, while strings are immutable. Simply put, this means that once a string's value is established, it can't be changed without creating a new variable, while a list can be modified (lengthened, shortened, rearranged, etc.) without having to store the results in a new variable, or reassign the value of an expression to the original variable name.

Consider a list with 5 integer elements:

```
>>> thelist = [0,5,10,15,20]
```

Now suppose we wish to change the central three elements (5, 10 and 15, at positions 1, 2 and 3 in the list) to the values 6, 7, and 8. As with a string, we could extract the three elements with a statement like:

```
>>> thelist[1:4] [5, 10, 15]
```

But with a list, we can also assign values to that slice:

```
>>> thelist[1:4] = [6,7,8]
>>> thelist
>>>[0, 6, 7, 8, 20]
```

If the number of elements in the list on the right-hand side of the equal sign is not equal to the number of elements implied by the subscript of the slice, the list will expand or shrink to accommodate the assignment. (Recall that the number of elements in a slice is the higher valued subscript minus the lower valued subscript.) The following examples illustrate this point:

```
>>> words = ['We', 'belong', 'to', 'the', 'knights', 'who', 'say', '"Ni"']
>>> words[1:4] = ['are']
>>> words
>>> ['We', 'are', 'knights', 'who', 'say', '"Ni"']
>>> words[1:2] = ['are','a','band','of']
>>> ['We', 'are', 'a', 'band', 'of', 'knights', 'who', 'say', '"Ni"']
```

Note that when we are replacing a slice with a single element, it must be surrounded by square brackets, effectively making it into a list with one element, to avoid a TypeError exception.

Assignments through slicing differ from those done with simple subscripting in that a slice can change the length of a list, while assignments done through a single subscript will always preserve the length of the list. This is true for slices where both of the subscripts are the same. Notice the difference between the two expressions shown below:

```
>>> # using a single subscript
>>> x = ['one','two','three','four','five']
>>> x[1] = ['dos','tres','cuatro']
>>> x
>>> ['one', ['dos', 'tres', 'cuatro'], 'three', 'four', 'five']
>>> # using a slice
>>> x = ['one','two','three','four','five']
>>> x[1:1] = ['dos','tres','cuatro']
>>> x
>>> ['one', 'dos', 'tres', 'cuatro', 'two', 'three', 'four', 'five']
```

In the final example, we were able to insert three elements into an list without replacing any elements in the list by assigning to a slice where both subscripts were the same.

Another use of slices is to make a separate modifiable copy of a list. In this case, you create a slice without either a starting or ending index. Python will then make a complete copy of the list

```
>>> x = ['one','two','three']
>>> y = x[:]
>>> y
>>>['one', 'two', 'three']
```

One final use of slices is to remove elements from an array. If we try to replace a single element or slice of an array with an empty list, that empty list will literally replace the locations to which it's assigned. But if we replace a slice of an array with an empty list, that slice of the array is effectively removed:

```
>>> a = [1,3,5,7,9]
>>> a[2] = []
>>> a
>>> [1, 3, [], 7, 9]
>>> b = [2,4,6,8]
>>> b[2:3] = []
>>> b
>>> [2, 4, 8]
```

Another way to remove items from a list is to use the del statement. You provide the del statement with the element or slice of a list which you want removed, and that element or slice is removed without a trace. So, to remove the second element from the list a in the previous example, we would use the del statement as follows:

```
>>> del a[2]
>>> a
>>> [1, 3, 7, 9]
```

The del statement is just as effective with slices:

```
>>> nums = [’one’,’two’,’three’,’four’,’five’]
>>> del nums[0:3]
>>> nums
>>> [’four’, ’five’]
```

In the previous example, the same result could be obtained by assigning an empty list to nums[0:3].

- ***List Operators***

A number of operators are overloaded with respect to lists, making some common operations very simple to express.

- **Concatenation**

To combine the contents of two lists, use a plus sign (+) between the two lists to be concatenated. The result is a single list whose length is the total of the length of the two lists being combined, and which contains all of the elements of the first list followed by all of the elements of the second list.

```
>>> first = [7,9,’dog’]
>>> second = [’cat’,13,14,12]
>>> first + second
>>> [7, 9, ’dog’, ’cat’, 13, 14, 12]
```

Alternatively, you can combine two lists with the expand method.

Note that list concatenation only works when you're combining two lists. To add a scalar to the end of a list, you can either surround the scalar with square brackets ([ ]), or invoke the append method.

- **Repetition**

Like strings, the asterisk (*) is overloaded for lists to serve as a repetition operator. The result of applying repetition to a list is a single list, with the elements of the original list repeated as many times as you specify:

```
>>> ['a','b','c'] * 4
>>>['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

Special care needs to be taken when the list being repeated contains only one element. Note that the expression 3 * 5 is a simple numeric computation, while

```
>>> 3 * [5] [5, 5, 5]
```

produces a list containing the element 5 repeated 3 times. By surrounding a list to be repeated with an extra set of square brackets, lists consisting of other lists can be constructed:

```
>>> littlelist = [1,2,3]
>>> 3 * littlelist
>>> [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 3 * [littlelist] [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

- **The “in” operator**

The in operator provides a very convenient way to determine if a particular value is contained in a list. As its name implies, you provide a value on the left-hand side of the operator, and a list on the right-hand side; an expression so constructed will return 1 if the value is in the list and 0 otherwise, making it ideal for conditional statements. The left-hand side can be a literal value, or an expression; the value on the right-hand side can be a list, or an expression that evaluates to a list.

Since python’s lists are so flexible, you need to be careful about how you construct expressions involving the in operator; only matches of the same type and value will return a value of 1. Consider the list squarepairs, each of whose elements are a list consisting of a small integer and its square:

```
>>> squarepairs = [[0,0],[1,1],[2,4],[3,9]]
```

To see if a list containing the elements 2 and 4 is contained in the squarepairs list, we could use an expression like the following:

```
>>> [2,4] in squarepairs
>>>1
```

Python responds with a 1, indicating that such a list is an element of squarepairs. But notice that neither element of the list alone would be found in squarepairs:

```
>>> 2 in squarepairs
>>>0
>>> 4 in squarepairs
>>>0
```

The in operator works just as well with expressions that evaluate to elements in a list:

```
>>> nums = [0,1,2,3,4,5]
>>> [nums[2]] + [nums[4]] in squarepairs
>>>1
```

The in operator is also used to iterate over the elements of a list in a for loop.

- ***Functions and Methods for Lists***

As mentioned previously the **len** function will return the number of elements in a list. When called with a single list argument, the **min** function returns the smallest element of a list, and the max function returns the largest. When you provide more than one argument to these

functions, the return the smallest **(min)** or largest **(max)** element among the arguments passed to the function.

The functions mentioned above accept a list as an argument, and return the desired result. However, many list operations are performed through methods instead of functions. Unlike strings, lists are mutable objects, so many methods which operate on lists will change the list – the return value from the method should *not* be assigned to any object.

To add a single element to a list, use the append method. Its single argument is the element to be appended.

```
>>> furniture = ['couch','chair','table']
>>> furniture.append('footstool')
>>> furniture
>>> ['couch', 'chair', 'table', 'footstool']
```

If the argument to the append method is a list, then a single (list) element will be appended to the list to which the method is applied; if you need to add several elements to the end of a list, there are two choices. You can use the concatenation operator or the extend method. The extend method takes a single argument, which must be a list, and adds the elements contained in the argument to the end of a list. Notice the difference between invoking the extend and append methods in the following example:

```
>>> a = [1,5,7,9]
>>> b = [10,20,30,40]
>>> a.extend(b)
>>> a
>>> [1, 5, 7, 9, 10, 20, 30, 40]
>>> a = [1,5,7,9]
>>> a.append(b)
>>> a
>>> [1, 5, 7, 9, [10, 20, 30, 40]]
```

(Since the extend method changes the object on which it operates, it was necessary to reinitialize a before invoking the append method. An alternative would have been to use the copy function to make a copy of a for the second method.) Note that with extend, the elements of b were added to a as individual elements; thus, the length of a is the total of its old length and the length of b, but when using append, b is added to a as a single element; the length of a list is always increased by exactly one when append operates on it.

In many cases we want to create a new list incrementally, by adding one element to the list at a time. As a simple example, suppose we have a list of numbers and we want to create a second list containing only those numbers which are less than 0. If we try to append an element to a non-existent list, it raises a NameError exception:

```
>>> oldlist = [7, 9, -3, 5, -8, 19]
>>> for i in oldlist:
>>>...    if i < 0 : newlist.append(i)
>>>...Traceback (innermost last):
      File "<stdin>", line 2, in ?
   NameError: newlist
```

The solution is to assign an empty list to newlist before we begin the loop. Such a list will have a length of zero, and no elements; the purpose of the assignment is simply to inform python that we will eventually refer to newlist as a list.

```
>>> oldlist = [7, 9, -3, 5, -8, 19]
>>> newlist = [ ]
>>> for i in oldlist:
>>>...    if i < 0 : newlist.append(i)
>>>...
>>> newlist
>>> [-3, -8]
```

To add an item to a list at a location other than the end, the insert method can be used. This method takes two arguments; the first is the index at which the item is to be inserted, and the second is the item itself. Note that only one element can be inserted into a list using this method; if you need to insert multiple items, slicing can be used.

To remove an item from a list, based on its value, not its subscript, python provides the remove method. It accepts a single argument, the value to be removed. Note that remove only removes the first occurrence of a value from a list.

The reverse and sort methods perform their operations on a list in place; in other words, when these methods are invoked on a list, the ordering of the elements in the list is changed, and the original ordering is lost. The methods **do not** return the reversed or sorted lists, so you should never set a list to the value returned by these methods! If you wish to retain the list with the elements in their original order, you should copy the list (using the copy module, not a regular assignment).

By default, the sort method sorts its numeric arguments in numerical order, and string arguments in alphabetical order Since a list can contain arbitrary objects, sort needs to be very flexible; it generally sorts scalar numeric values before scalar string values, and it sorts lists by first comparing their initial elements, and continuing through the available list elements until one list proves to be different than the other.

To sort in some other order than the method's default, a comparison function accepting exactly two arguments can be supplied as an argument to sort This function should be modeled on the built-in function cmp, which returns 1 if its first argument is greater than the second, 0 if the two arguments are equal, and -1 if the second argument is greater than the first. We'll look at function definitions in more detail later, but let's say we wish to sort strings, ignoring the case

of the strings. The lower function in the string module can be used to return the lower-case version of a string, and if we use that to compare pairs of values, the strings will be sorted without regard to case. We can write and use a function to do this comparison as follows:

```
>>> def nocase(a,b):
>>>...    return cmp(a.lower(),b.lower()) ...
>>> names = ['fred','judy','Chris','Sam','alex','Heather']
>>> copynames = names[:]
>>> names.sort()
>>> names
>>> ['Chris', 'Heather', 'Sam', 'alex', 'fred', 'judy']
>>> names = copynames[:]
>>> names.sort(nocase)
>>> names
>>> ['alex', 'Chris', 'fred', 'Heather', 'judy', 'Sam']
```

The count method counts how many times a particular value appears in a list. It accepts a single argument which represents the value you're looking for, and returns the number of times that the value appears in the list.

The index method accepts a single argument, and, if that argument is found in the list, it returns the index (subscript) of its first occurrence. If the value is not in the list, a ValueError exception is raised.

The following example demonstrates the use of count and index.

```
>>> food = ['spam','spam','spam','sausage','spam']
>>> food.count('spam') 4
>>> food.index('spam')
>>>0
```

Even though "spam" appears four times in the food list, the index method always returns the index of the first occurence of the value only. If the index method fails to find the requested element, a ValueError exception is raised.

**Related Modules**: copy, array, struct
**Related Exceptions**: IndexError, ValueError

## 2. Tuple

Tuples are very much like lists, except for one important difference. While lists are mutable, tuples, like strings, are not. This means that, once a tuple is created, its elements can't be modified in place. Knowing that a tuple is immutable, python can be more efficient in manipulating tuples than lists, whose contents can change at any time, so when you know you won't need to change the elements within a sequence, it may be more efficient to use a tuple instead of a list. In addition, there are a number of situations (argument passing and string formatting for example) where tuples are required.

Tuples are created in a similar fashion to lists, except that there is no need for square brackets surrounding the value. When the python interpreter displays a tuple, it always surrounds it with parentheses; you can use parentheses when inputting a tuple, but it's not necessary unless the tuple is part of an expression. This creates a slight syntactic problem when creating a tuple with either zero or one element; python will not know you're creating a tuple. For an empty (zero-element) tuple, a pair of empty parentheseis (()) can be used. But surrounding the value with parentheses is not enough in the case of a tuple with exactly one element, since parentheses are used for grouping in arithmetic expression. To specify a tuple with only one element in an assignment statement, simply follow the element with a comma. In arithmetic expressions, you need to surround it with parentheses, and follow the element with a comma before the closing parenthesis.

For example, suppose we wish to create a new tuple which concatenates the value 7 to the end of an existing tuple:

```
>>> values = 3,4,5
>>> values + (7)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>> values + (7,) (3, 4, 5, 7)
>>> newvalue = 7,
>>> values + newvalue (3, 4, 5, 7)
```

Without the closing comma, python regards the value (7) as just a single number, and raises a TypeError exception when we try to concatenate it to an existing tuple. The closing comma identifies the expression as a singleelement tuple, and concatenation can be performed.

Like lists, the contents of tuples are arbitrary. The only difference is that lists are mutable and tuples are not.

### ➢ *Operators and Indexing for Tuples*

The same operators mentioned for lists apply to tuples as well, keeping in mind that tuples are immutable. Thus, slicing operations for tuples are more similar to strings than lists; slicing can be used to extract parts of a tuple, but not to change them.

## ➢ *Functions and Methods for Tuples*

Since tuples and lists are so similar, it's not surprising that there are times when you'll need to convert between the two types, without changing the values of any of the elements. There are two built-in functions to take care of this: list, which accepts a tuple as its single argument and returns a list with identical elements, and tuple which accepts a list and returns a tuple.

These functions are very useful for resolving TypeError exceptions involving lists and tuples.

There are no methods for tuples; however, if a list method which doesn't change the tuple is needed, you can use the list function to temporarily change the tuple into a list to extract the desired information.

Suppose we wish to find how many times the number 10 appears in a tuple. The **count** method cannot be used on a tuple directly, since an AttributeError is raised:

```
>>> v = (7,10,12,19,8,10,4,13,10)
>>> v.count(10)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'count'
```

To solve the problem, call the list function with the tuple as its argument, and invoke the desired method directly on the object that is returned:

```
>>> list(v).count(10)
>>>3
```

Note, however, that if you invoke a method which changes or reorders the values of a temporary list, python will not print an error message, but no change will be made to the original tuple.

```
>>> a = (12,15,9)
>>> list(a).sort()
>>> a
>>> (12, 15, 9)
```

In case like this, you'd need to create a list, invoke the method on that list, and then convert the result back into a tuple.

```
>>> aa = list(a)
>>> aa.sort()
>>> a = tuple(aa)
>>> a
>>> (9, 12, 15)
```

## 3. Set

Sets are used to store multiple items in a single variable. A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

Sets are written with curly brackets:

```
>>>thisset = {"apple", "banana", "cherry"}
>>>print(thisset)
```

Set items are unordered, unchangeable, and do not allow duplicate values. Unordered means that the items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key. Set items are unchangeable, meaning that we cannot change the items after the set has been created. **Sets cannot have two items with the same value.**

```
>>>thisset = {"apple", "banana", "cherry", "apple"}
>>>print(thisset)
>>> {'banana', 'cherry', 'apple'}
#True and 1 is considered the same value:
>>>thisset = {"apple", "banana", "cherry", True, 1, 2}
>>>print(thisset)
>>> {True, 2, 'apple', 'banana', 'cherry'}
```

To determine how many items a set has, use the **len()** function.

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

A set can contain different data types:

>>>set1 = {"abc", 34, True, 40, "male"}

It is also possible to use the **set()** constructor to make a set.

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

- **Access Items**

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

```python
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

- **Add Items**

To add one item to a set use the **`add()`** method.

```python
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

- **Add Sets**

To add items from another set into the current set, use the **`update()`** method.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

The object in the **`update()`** method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]

thisset.update(mylist)

print(thisset)
```

- **Remove Item**

To remove an item in a set, use the **`remove()`**, or the **`discard()`** method.

```python
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

```python
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

You can also use the **`pop()`** method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.

The return value of the **`pop()`** method is the removed item.

```python
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

The **clear()** method empties the set:

```
>>>thisset = {"apple", "banana", "cherry"}

>>>thisset.clear()

>>>print(thisset)
```

The del keyword will delete the set completely:

```
>>>thisset = {"apple", "banana", "cherry"}

>>>del thisset

>>>print(thisset)
```

- ***Join Two Sets***

There are several ways to join two or more sets in Python.

You can use the **`union()`** method that returns a new set containing all items from both sets, or the **`update()`** method that inserts all the items from one set into another:

The **union()** method returns a new set with all items from both sets:

```
>>> set1 = {"a", "b" , "c"}
>>> set2 = {1, 2, 3}

>>> set3 = set1.union(set2)
>>> print(set3)
>>> {1, 2, 'c', 3, 'a', 'b'}
```

The **update()** method inserts the items in set2 into set1:

```
>>>set1 = {"a", "b" , "c"}
>>>set2 = {1, 2, 3}

>>>set1.update(set2)
>>>print(set1)
>>> {'a', 1, 2, 3, 'b', 'c'}
```

**Note:** Both **union()** and **update()** will exclude any duplicate items.

- **Keep ONLY the Duplicates**

The **`intersection_update()`** method will keep only the items that are present in both sets.

Keep the items that exist in both set x, and set y:

```
>>>x = {"apple", "banana", "cherry"}
>>>y = {"google", "microsoft", "apple"}
>>>x.intersection_update(y)
>>>print(x)
>>>>{'apple'}
```

The **intersection()** method will return a *new* set, that only contains the items that are present in both sets.

Return a set that contains the items that exist in both set x, and set y:

```
>>>x = {"apple", "banana", "cherry"}
>>>y = {"google", "microsoft", "apple"}
>>>z = x.intersection(y)
>>>print(z)
>>>{'apple'}
```

- **Keep All, But NOT the Duplicates**

The **symmetric_difference_update()** method will keep only the elements that are NOT present in both sets.

Keep the items that are not present in both sets:

```
>>>x = {"apple", "banana", "cherry"}
>>>y = {"google", "microsoft", "apple"}

>>>x.symmetric_difference_update(y)

>>>print(x)
>>> {'google', 'cherry', 'microsoft', 'banana'}
```

The **symmetric_difference()** method will return a new set, that contains only the elements that are NOT present in both sets.

Return a set that contains all items from both sets, except items that are present in both:

```
>>>x = {"apple", "banana", "cherry"}
>>>y = {"google", "microsoft", "apple"}

>>>z = x.symmetric_difference(y)

>>>print(z)
>>> {'microsoft', 'google', 'banana', 'cherry'}
```

**Note:** The values True and 1 are considered the same value in sets, and are treated as duplicates:

```
>>>x = {"apple", "banana", "cherry", True}
>>>y = {"google", 1, "apple", 2}

>>>z = x.symmetric_difference(y)

>>>print(z)

>>> {'google', 2, 'cherry', 'banana'}
```

## ➢ *Set Methods*

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

| Method | Description |
|---|---|
| **add()** | Adds an element to the set |
| **clear()** | Removes all the elements from the set |
| **copy()** | Returns a copy of the set |
| **difference()** | Returns a set containing the difference between two or more sets |
| **difference_update()** | Removes the items in this set that are also included in another, specified set |
| **discard()** | Remove the specified item |
| **intersection()** | Returns a set, that is the intersection of two other sets |

| | |
|---|---|
| **intersection_update()** | Removes the items in this set that are not present in other, specified set(s) |
| **isdisjoint()** | Returns whether two sets have a intersection or not |
| **issubset()** | Returns whether another set contains this set or not |
| **issuperset()** | Returns whether this set contains another set or not |
| **pop()** | Removes an element from the set |
| **remove()** | Removes the specified element |
| **symmetric_difference()** | Returns a set with the symmetric differences of two sets |
| **symmetric_difference_update()** | inserts the symmetric differences from this set and another |
| **union()** | Return a set containing the union of sets |
| **update()** | Update the set with the union of this set and others |

## 4. Dictionaries

Dictionaries (sometimes refered to as associative arrays or hashes) are very similar to lists in that they can contain arbitrary objects and can be nested to any desired depth, but, instead of being indexed by integers, they can be indexed by any immutable object, such as strings or tuples. Since humans can more easily associate information with strings than with arbitrary

numbers, dictionaries are an especially convenient way to keep track of information within a program.

As a simple example of a dictionary, consider a phonebook. We could store phone numbers as tuples inside a list, with the first tuple element being the name of the person and the second tuple element being the phone number:

```
>>> phonelist = [('Fred','555-1231'),('Andy','555-1195'),('Sue','555-2193')]
```

However, to find, say, Sue's phone number, we'd have to search each element of the list to find the tuple with Sue as the first element in order to find the number we wanted. With a dictionary, we can use the person's name as the index to the array. In this case, the index is usually refered to as a key. This makes it very easy to find the information we're looking for:

```
>>> phonedict = {'Fred':'555-1231','Andy':'555-1195','Sue':'555-2193'}
>>> phonedict['Sue']
>>>'555-2193'
```

As the above example illustrates, we can initialize a dictionary with a commaseparated list of key/value pairs, separated by colons, and surrounded by curly braces. An empty dictionary can be expressed by a set of empty curly braces ({}).

Dictionary keys are not limited to strings, nor do all the keys of a dictionary need be of the same type. However, mutable objects such as lists can not be used as dictionary keys and an

attempt to do so will raise a TypeError. To index a dictionary with multiple values, a tuple can be used:

```
>>> tupledict = {(7,3):21,(13,4):52,(18,5):90}
```

Since the tuples used as keys in the dictionary consist of numbers, any tuple containing expressions resulting in the same numbers can be used to index the dictionary:

```
>>> tupledict[(4+3,2+1)]
>>>21
```

In addition to initializing a dictionary as described above, you can add key/value pairs to a dictionary using assignment statements:

```
>>> tupledict[(19,5)] = 95
>>> tupledict[(14,2)] = 28
```

To eliminate a key/value pair from a dictionary use the del statement.

## ➢ *Functions and Methods for Dictionaries*

As is the case for strings and lists, the **len()** function returns the number of elements in a dictionary, that is, the number of key/value pairs. In addition, a number of methods for dictionaries are available.

Since referring to a non-existent key in a dictionary raises a **KeyError** exception, you should avoid indexing a dictionary unless you're sure of the existence of an entry with the key you are using. One alternative is to use a try/except clause to trap the **KeyError**. Two methods are also useful in this situation. The **has_key** method returns 1 (true) if the specified dictionary has the key which the method accepts as an argument, and 0 (false) otherwise. Thus, an if clause can be used to act if a specified key does not exist. The get method also accepts a key as an argument, but it returns the value stored in the dictionary under that key if it exists, or an optional second argument if there is no value for the key provided. With only one argument, get returns the value None when there is no value corresponding to the key provided.

As an illustration of these methods, suppose we are counting how many times a word appears in a file by using each word as a key to a dictionary called counts, and storing the number of times the word appears as the corresponding value. The logic behind the program would be to add one to the value if it already exists, or to set the value to one if it does not, since that will represent the first time the word is encountered. Here are three code fragments illustrating the different ways of handling a value stored in word which may or may not already be a key in

the dictionary. (Remember that counts would have to be initialized to an empty dictionary before any elements could be added to it.)

```
# method 1:
        exceptions try:
                counts[word] = counts[word] + 1
        except KeyError:
                counts[word] = 1

# method 2: check with has_key if counts.has_key(word):
                counts[word] = counts[word] + 1
        else:
                counts[word] = 1

# method 3: use get counts[word] = counts.get(word,0) + 1
```

Although the third method is the shortest, it may not be obvious why it works. When an entry with the specified key already exists, the get method will return that entry and increment it by 1. If the key does not exist, the optional second argument to get forces it to return a value of 0, which, when incremented by 1 gives the correct initial value.

It is possible to iterate over a dictionary using a for loop; the operation will return the keys of the dictionary in an arbitrary order. In addition, the in operator can be used as an alternative to the has_key method to determine if a particular key is present in a dictionary.

A few other methods provide alternative ways of accessing the keys, values or key/value pairs of a dictionary. The keys method returns a list of just the keys of a dictionary, and the values method returns a list of just the values. While the returned lists have their elements stored in an arbitrary order, corresponding elements in the two lists will be key/value pairs. The items method returns a list of tuples consisting of all the key/value pairs. Returning to the example using names as keys and phone numbers as values, here are the results of invoking these three methods on the phonedict dictionary:

```
>>> phonedict = {'Fred':'555-1231','Andy':'555-1195','Sue':'555-2193'}
>>> phonedict.keys()
>>> ['Fred', 'Sue', 'Andy']
>>> phonedict.values()
>>> ['555-1231', '555-2193', '555-1195']
>>> phonedict.items()
>>> [('Fred', '555-1231'), ('Sue', '555-2193'), ('Andy', '555-1195')]
```

To remove all the elements of a dictionary, use the clear method. This differs from using the del operator on the dictionary in that the dictionary, although empty, still exists. To make a copy of a dictionary, use the copy method; remember that assignment of a dictionary to another variable may not do what you think.

Python provides the update method to allow you to merge all the key/value pairs in one dictionary with those in another. If the original dictionary has an entry with the same key as

one in the dictionary used for updating, the value from the dictionary used for updating will be placed in the original dictionary.

```
>>> master = {'orange':3,'banana':5,'grapefruit':2}
>>> slave = {'apple':7,'grapefruit':4,'nectarine':3}
>>> master.update(slave)
>>> master
>>> {'banana': 5, 'orange': 3, 'nectarine': 3, 'apple': 7, 'grapefruit': 4}
```

Since both master and slave had entries for the key "grapefruit", the value found in master after the call to update is the one which was in slave.