**Data Structures**

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or docstring. (More about docstrings can be found in the section Documentation Strings.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a global statement, or, for variables of enclosing functions, named in a nonlocal statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object). 1 When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function.

**list.append(x)**

Add an item to the end of the list. Equivalent to a[len(a):] = [x].

**list.extend(iterable)**

Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

**list.insert(i, x)**

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

**list.remove(x)**

Remove the first item from the list whose value is equal to x. It raises a [ValueError](https://docs.python.org/3/library/exceptions.html#ValueError) if there is no such item.

**list.pop([i])**

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

**list.clear()**

Remove all items from the list. Equivalent to del a[:].

**list.index(x[, start[, end]])**

Return zero-based index in the list of the first item whose value is equal to x. Raises a [ValueError](https://docs.python.org/3/library/exceptions.html#ValueError) if there is no such item.

The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.

**list.count(x)**

Return the number of times x appears in the list.

**list.sort(key=None, reverse=False)**

Sort the items of the list in place (the arguments can be used for sort customization, see [sorted()](https://docs.python.org/3/library/functions.html#sorted) for their explanation).

**list.reverse()**

Reverse the elements of the list in place.

**list.copy()**

Return a shallow copy of the list. Equivalent to a[:].

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The [del](https://docs.python.org/3/reference/simple_stmts.html#del) statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice).

A tuple consists of a number of values separated by commas.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are [immutable](https://docs.python.org/3/glossary.html#term-immutable), and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of [namedtuples](https://docs.python.org/3/library/collections.html#collections.namedtuple)). Lists are [mutable](https://docs.python.org/3/glossary.html#term-mutable), and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses).

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the [set()](https://docs.python.org/3/library/stdtypes.html#set) function can be used to create sets.

**Dictionaries**

Another useful data type built into Python is the dictionary (see [Mapping Types — dict](https://docs.python.org/3/library/stdtypes.html#typesmapping)). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().