Data structures

Before considering data structures, one needs to consider algorithms.

What is an algorithm?

An **algorithm** is a sequence of clear, well-defined, instructions which is used to solve a problem. An algorithm is straight forward, unambiguous and has a start and an end.

What is a data structure?

A **data structure** is a way to store data, so that this data can be processed in an efficient manner during a particular type of operation. The way that data is stored, affects the efficiency of the algorithm that is used to process them.

The algorithm used to process the data, combined with the data structure used to store them, make up a programme. In Object Oriented Programming, the two are encapsulated within objects.

Common operations on data structures

1. **Insert** an item in the data structure
2. **Delete** an item from the data structure
3. **Update** an item that already exists within the data structure
4. **Sort** the items within the data structure in a specific order
5. **Search** the items within the data structure

Common data structures

1. Arrays
2. Lists
3. Stacks
4. Queues – Priority queues
5. Trees – Heap Trees

Arrays
An array is a data structure that contains elements of the same data type. An example is:

[5, 45, 73, 6, 59, 80, 70, 23, 34, 24]

The elements are stored in a sequence of memory locations:

Position number (index)	Memory address	Content
	0x147A	
9	0x1479	24
8	0x1478	34
7	0x1477	23
6	0x1476	70
5	0x1475	80
4	0x1474	59
3	0x1473	6
2	0x1472	73
1	0x1471	45
0	0x1470	5

Important things about lists:

1. The first element within an array is placed at position 0, the second at position 1, etc.

2. The position of an element in an array is known as the **index** of the element. When accessing the members of an array, the index of the element of interest is added to the base address of the block where the array is stored. Therefore, it is very easy and fast to traverse an array and locate a specific element within it by using either a **while**, or a **for** loop.

3. Accessing a specific element of an array has a constant time complexity, that is it takes the same time to access any element of the array.

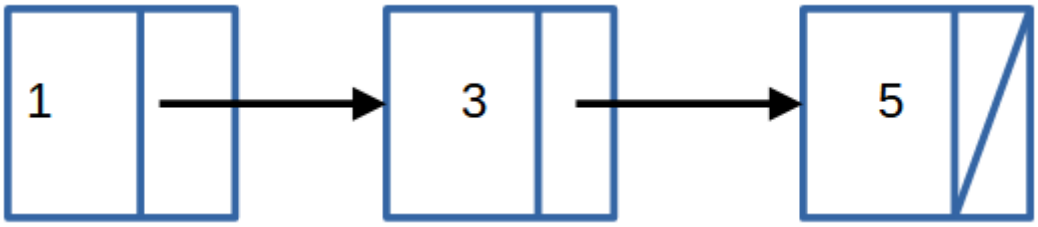
4. The size of an array denotes the total number of elements it can hold. There can be N-dimensional arrays.

Python does not have built-in support for Arrays, but Python Lists can be used instead. Alternatively, the array module can be imported and used (see the array.py example)

Note: The four built-in data structures of Python are: Lists, Tuple, Set and Dictionary

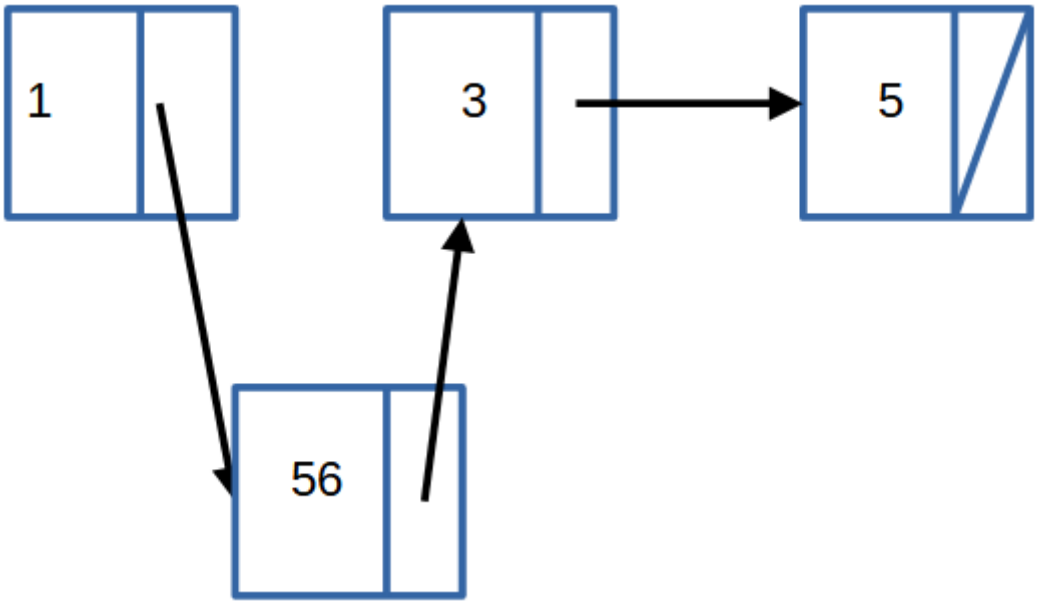
Lists

A list is an ordered collection of values of various types, that does not require a predefined fixed size. Each node of the list is stored in memory as distinct blocks at various memory locations, which are connected together by using pointers, therefore, Lists are a **linked data structure**. They are also an **aggregate data structure**. For example, a list containing three nodes, each holding an integer: 1, 3, 5, is implemented in memory as follows:

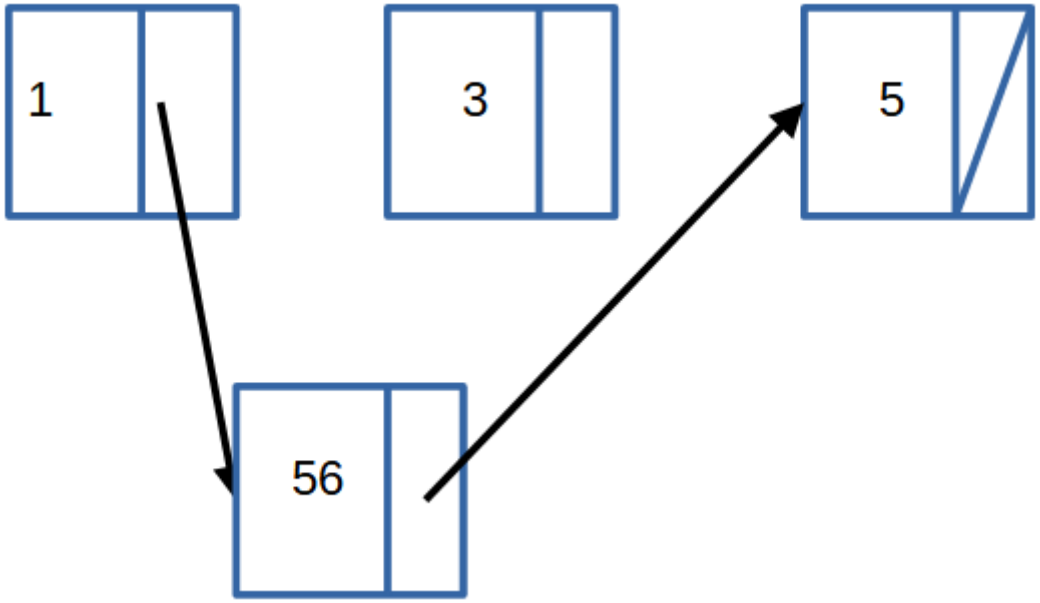


Adding/removing a node of the list requires the update of the pointer that:

1. leads to the node that is added/removed
2. leads to the node following the member that is added/removed



A node carrying the integer 56 is inserted between the first and second nodes.



The node carrying integer 3 is removed from the list.

Because the pointer present at the last element of the list does not point to a valid node, it is called the **null pointer**. We denote the absence of a following node by inserting a diagonal line at the pointer box.

Important things about lists:

1. A list is a **mutable** data structure. This means that we can add elements, delete elements, or modify existing ones.

2. A list can contain the same value multiple times as **each occurrence** is considered **a distinct element**.

3. Lists have variable length.

4. The **first item** in a list is the **Head** of the list (in the example above, it is the element carrying the value “1”).

5. The **last item** in a list is the **Tail** of the list (in the example above, it is the element carrying the value “5”)

6. Because there is no item index to use, there is no way to locate a specific element directly. A list is traversed by visiting each item in turn:
   a) The pointer to the first element (the Head) is used to reach the memory location holding the element.
   b) The pointer held in that element towards the following element, is read.
   c) Steps a) and b) are repeated until the specific element required is reached.

7. Because of 6), locating an item in a list has a linear time complexity: there is a linear relationship between the length of the list and the time needed to access its last element.

8. A list can be divided in two parts: the **first** element and the **rest** of the elements. Thanks to this property, **recursion** can be used very effectively to handle a list.

From:
https://docs.python.org/3/tutorial/datastructures.html

we see that Python offers a number of methods to handle lists:

Method	Description
append()	Adds an element at the end of the list
extend()	Add the elements of a list, to the end of the current list
insert()	Adds an element at the specified position
remove()	Removes the item with the specified value
pop()	Removes the element at the specified position
clear()	Removes all the elements from the list
index()	Returns the index of the first element with the specified value
count()	Returns the number of elements with the specified value
sort()	Sorts the list
reverse()	Reverses the order of the list
copy()	Returns a copy of the list

Lists in Python use square brackets to enclose their elements.

(see the lists.py example)

Aside: Pointers in Python

Pointers are essential to the functioning of lists. However, they do not play an important role to the user in Python, as they do in other languages, because Python attempts to abstract the underlying mechanisms involved in its functionality and focuses on usability, instead of efficiency.

However, they deserve a quick look.

Pointers are variables that hold the memory address of another variable. In Python, the memory address where a variable is held is available through the **id(variablename)** method and is compared to the memory address of another variable through the **is** method.

Because everything is an object in Python, each time an operation takes place on a variable, a new object is created in memory and the **name** of the variable is made to **point** to that new object. Each object takes hold of an area within memory where the following are stored:

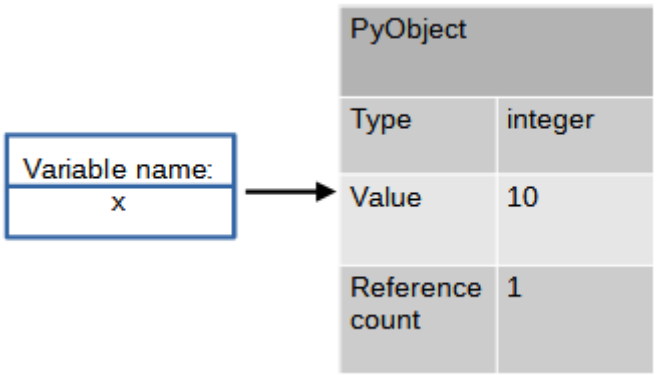
- the type of the object (eg. integer)
- the value
- a reference count (how many variable names point to the specific object)

The assignment: x = y, makes the name x to point to the memory location holding the value of the variable and the name y to point to the same memory location. In this case, the reference count is equal to 2.

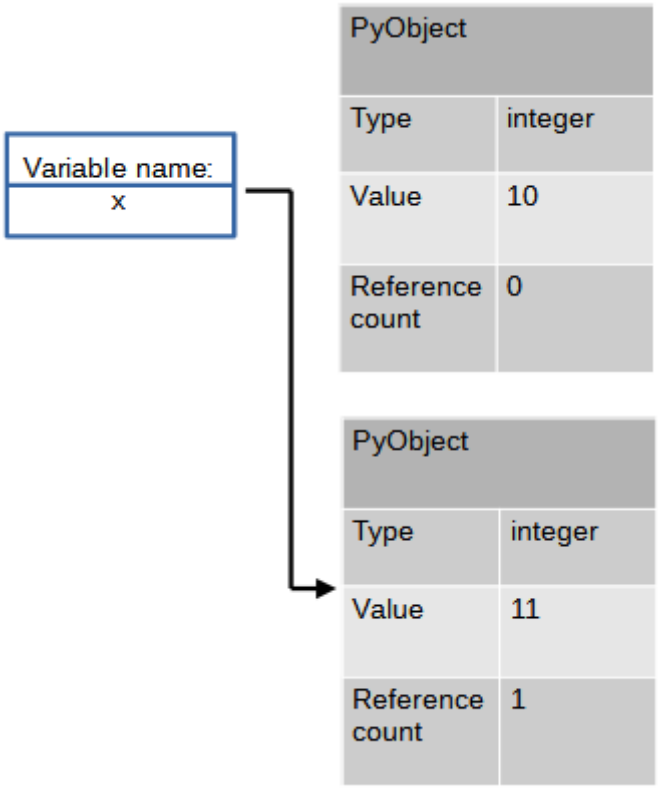
If no variables point to the object in question, the reference count is equal to 0. This flags to the memory management system that the area is available and can be released from use by the memory garbage collector.

Based on the pointers.py example, the above can be illustrated as follows:

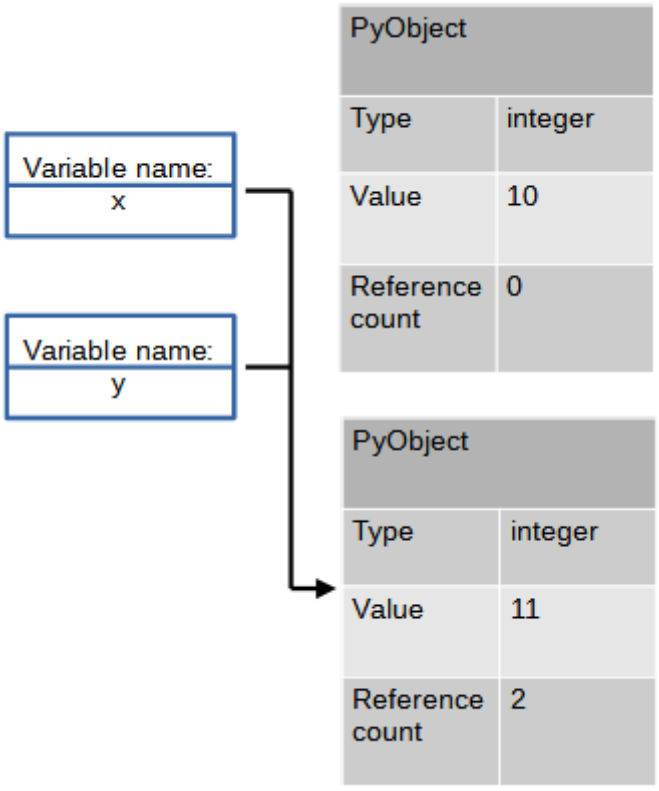
When the line x = 10 is executed:



When x = x + 1


Variable name:
x
PyObject
Type
integer
Value
10
Reference count
0
PyObject
Type
integer
Value
11
Reference count
1

When y = x



There are techniques that can be used to **simulate** the use of pointers in Python, but these go beyond the scope of this tutorial.

Even though users are not given free access to pointers in Python, pointers can still be used to pass parameters to a function call.

(see the pointers.py example)

Tuples
Tuples in Python are very similar to lists in the sense that they are collections of potentially heterogeneous elements. They are an **aggregate data structure**. They are accessed in the same way as lists.

Important things about tuples:

1. A tuple is an **immutable** data structure. This means that we **cannot** add elements, delete elements, or modify existing ones.

2. A tuple uses parentheses to enclose its elements.

Note: mutable elements within a tuple can be modified. For example, if an element of a tuple consists of a list, the contents of that list can be modified by using the list modification methods.

A tuple can be used to:

1. Group data even though they might be heterogeneous

2. Return values from functions to their callers

Dictionaries
Dictionaries in Python are very similar to lists in the sense that they are collections of potentially heterogeneous elements. They are an **aggregate data structure**. They are created by using curly brackets in Python, and they consist of **key value pairs**.

Important things about dictionaries:

1. A dictionary is a **mutable** data structure. Elements can be added, deleted, or modified.
2. They are ordered (as of Python 3.7).
3. Their elements consist of key value pairs. A dictionary provides a map from a key to a value.
4. They are indexed based on the keys of the elements.

(see the dictionary.py example)

Sets

Sets in Python are very similar to lists in the sense that they are collections of objects, called elements or members. They are an aggregate data structure. They are created by using either curly brackets, or the **set** method in Python.

Important things about sets:

1. Sets are unordered.
2. Set elements are unique. Duplicate elements are not allowed.
3. A set itself may be modified, but its elements cannot. For example, a tuple (immutable) may be included in a set. Lists and dictionaries (mutable) cannot be included in a set.
4. Python offers a full set of operators on sets

(see the sets.py example)

Sets in Python are implemented by using hash tables (which are beyond the scope of this tutorial)

Stack
A stack is a linear data structure, an ordered list of homogeneous elements. Implementing a stack requires a single **linked list** and a pointer to its top.

Items are added at the tail of the list through **push** operations and removed from the tail of the list through **pop** operations. Items that were pushed last, are popped first (LIFO).

Starting from an empty stack:

Stack empty	
Stack top	

Pushing byte 121:

Pushed 1 byte	
Stack top	
	121

Pushing byte 221:

Pushed 2 bytes	
Stack top	
	221
	121

Pushing byte 45:

Pushed 3 bytes	
Stack top	
	45
	221
	121

Popping one byte:

Popped 1 byte	
Stack top	
	221
	121

Stacks are the basis of subroutine calls. Just before branching to a subroutine, the CPU pushes to the stack the variables of the status of the CPU. Upon return, the variables are popped, restoring the status of the CPU.

Queue
Similar to a stack, a queue is a linear data structure, an ordered list of homogeneous elements. Implementing a queue requires a double **linked list** and pointers to the beginning and the end of the queue.

Items are added at the rear of the queue through **enqueue** operations and removed from the front of the queue through **dequeue** operations. Items that entered the queue first, are removed first (FIFO).

Starting from an empty stack:

Queue empty	

Adding byte 121:

Added 1st byte	
Queue front	
	121

Adding byte 221:

Added 2nd byte	
Queue front	
	121
	221

Adding byte 45:

Added 3rd byte	
Queue front	
	121
	221
	45

Removing one byte:

Removed 1 byte	
Queue front	
	221
	45

Queues find particular use as buffers, both linear and circular and can be implemented as linked lists.