**Common Python Data Structures (Guide)**

* [What is a Data Structure?](https://www.edureka.co/blog/data-structures-in-python/#datastructure)
* [Types of Data Structures in Python](https://www.edureka.co/blog/data-structures-in-python/#types)
* [Built-in Data Structures](https://www.edureka.co/blog/data-structures-in-python/#builtin)
  + [List](https://www.edureka.co/blog/data-structures-in-python/#list)
  + [Dictionary](https://www.edureka.co/blog/data-structures-in-python/#dictionary)
  + [Tuple](https://www.edureka.co/blog/data-structures-in-python/#tuple)
  + [Sets](https://www.edureka.co/blog/data-structures-in-python/#set)
* [User-Defined Data Structures](https://www.edureka.co/blog/data-structures-in-python/#user)
  + [Arrays vs. List](https://www.edureka.co/blog/data-structures-in-python/#arrayvslist)
  + [Stack](https://www.edureka.co/blog/data-structures-in-python/#stack)
  + [Queue](https://www.edureka.co/blog/data-structures-in-python/#queue)
  + [Trees](https://www.edureka.co/blog/data-structures-in-python/#tree)
  + [Linked Lists](https://www.edureka.co/blog/data-structures-in-python/#linkedlist)
  + [Graphs](https://www.edureka.co/blog/data-structures-in-python/#graph)
  + [HashMaps](https://www.edureka.co/blog/data-structures-in-python/#hashmap)

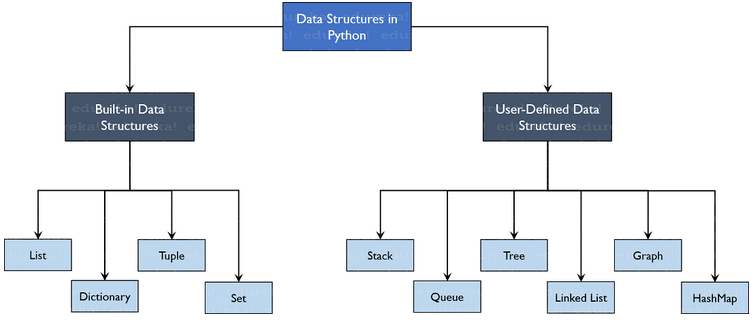
**What is a Data Structure?**

**Organizing, managing and storing data is important as it enables easier access and efficient modifications. Data Structures allows you to organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly.**

**Types of Data Structures in Python**

**Python has implicit support for Data Structures which enable you to store and access data. These structures are called List, Dictionary, Tuple and Set.**

Python allows its users to create their own Data Structures enabling them to have **full control** over their [functionality](https://www.edureka.co/blog/python-functions). The most prominent Data Structures are Stack, Queue, Tree, Linked List and so on which are also available to you in other programming languages. So now that you know what are the types available to you, why don’t we move ahead to the Data Structures and implement them using Python.

****

**Built-in Data Structures**

**As the name suggests, these Data Structures are built-in with Python which makes programming easier and helps programmers use them to obtain solutions faster. Let’s discuss each of them in detail.**

[**Lists**](https://www.edureka.co/blog/lists-in-python/)**:**

What Is A List In Python?

The Python List is a general data structure widely used in Python programs. They are found in other languages, often referred to as dynamic arrays. They are both mutable and a sequence data type that allows them to be indexed and sliced. The list can contain different types of objects, including other list objects.

List is a mutable sequence of objects

* Internally list uses an array data structure
* It is ordered- (appear in same position)
* It allows duplicate
* It allows indexing using positive or negative numbers a[1] or a[-1]
* Stores different types

To declare a list, we use the square brackets. [ ]

Sequential Indexing#

To access the elements of a list or a string which exists inside another list, we can use the concept of sequential indexing.

Each level of indexing takes us one step deeper into the list, allowing us to access any element in a complex list.

All we have to do is specify all the indices in a sequence:

world\_cup\_winners = [[2006, "Italy"], [2010, "Spain"],

                     [2014, "Germany"], [2018, "France"]]

print(world\_cup\_winners[1])

print(world\_cup\_winners[1][1])  # Accessing 'Spain'

print(world\_cup\_winners[1][1][0])  # Accessing 'S'

Output:

[2010, 'Spain']

Spain

S

List is like any other array that we declare in other programming languages. Lists in python are often used to implement stacks and queues. The lists are mutable in nature. Therefore, the values can be changed even after a list is declared.

**Why Use A List?**

While choosing a data type for storing our data, we must keep in mind the properties and features of the data type. It becomes more efficient and secure if we make the right choice in first place.

A list is preferred because it can store multiple data at the same time. It becomes easy to replace and modify the values inside a list. We can store the sequence in a list and perform several iterations using the loops as well. There are numerous operations we can perform on a list as well, lets understand the various operations that we have for lists in python.

|  |  |  |
| --- | --- | --- |
| 1. append 2. clear 3. copy 4. count | 1. extend 2. insert 3. index 4. pop | 1. remove 2. reverse 3. sort |

List Operations in Python

append

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

a.append(6)

print(a)

#the output will have 6 at the end of the list.

clear

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

a.clear()

#this will clear the list or empty the list.

copy

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

b = a.copy()

print(b)

#it makes the copy of the list.

count

a = [1,1,1,3,3,3,4,4,4,4,5,5,5,5,5]

a.count(5)

#this will give the number of times 5 is present in the list.

extend

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

a.extend(range(6,11))

#this will add the values in this list from the iterable object range.

insert

a = ['edureka', 'python', 'data science']

a.insert(2,'artificial intelligence')

#this will add the string at the index value 2

index

a = ['edureka', 'python', 'programming', 'data science', 'AI', 'machine learning']

a.index('data science')

#this will get the index value at the string 'data science' which is 3.

pop

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

a.pop()

#this will pop the value from the end of the list i.e 5. the list will no longer have 5 after this.

remove

a = [1,2,3,4,11,5]

a.remove(11)

#this will remove 11 from the list.

reverse

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

a.reverse()

#this will reverse the list.

#another statement to reverse the list a = a[: :-1]

sort

a = [3,1,2,6,4,5,9,6,7,8]

a.sort()

#you will get a sorted list as a result.

Replacing a value in a list

a = ['edureka', 'python', 'data science', 'tennis', 'machine learning']

a[3] = 'artificial intelligence'

#this will replace the value at the given index with the mentioned value.

**If we just want to verify the existence of an element in a list, we can use the in operator:**

cities = ["London", "Paris", "Los Angeles", "Beirut"]

print("London" in cities)

print("Moscow" not in cities)

Output: True

True

**Iterate through a list**

Lists can be used for iterations as well. Below is the code to iterate a list and print values using a control statement.

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

for x in a:

if x == 4:

break

print(x)

#this will iterate through the list and print the values until it encounters 4.

**The List Constructor**

The list constructor is used to create/declare a list.

a = list((1,2,3,4,5))

print(a)

#you will get a list with the values declared in the constructor.

As you can see, the list constructor takes the tuple as argument. Similarly, you can declare any other data type like a dictionary or a set inside the list constructor as well.

**Slicing A List In Python**

Suppose you have a list with numbers from 0-10. But you only want to get the numbers from 5-10, you must not access all the elements typing the index values of all those numbers. Instead, you can follow the approach in the code below.

Slicing a list gives us a sublist:

a = [1,2,3,4,5,6,7,8,9,10]

a[4:11]

#this will get all the numbers starting from index 4 to index 11.

a[-1:-6]

#this will get all the numbers from the index 11 to index 6.

a[4:]

#this will print all the numbers starting from index 4 until the end of the list.

a[:6]

#this will print all the numbers from index 0 until the index 6.

You might have noticed that methods like insert, remove or sort that only modify the list have no return value printed – they return the default None. 1 This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, [None, 'hello', 10] doesn’t sort because integers can’t be compared to strings and None can’t be compared to other types. Also, there are some types that don’t have a defined ordering relation. For example, 3+4j < 5+7j isn’t a valid comparison.

**List Comprehensions**

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.

**List comprehension is a technique that uses a for loop and a condition to create a new list from an existing one.**

**Structure#**

A list comprehension statement is always enclosed in square brackets, [].

The comprehension consists of three main parts:

**[expression - for loop - if condition]**

* The expression is an operation used to create elements in the new list.
* The for loop will iterate an existing list. The iterator will be used in the expression.
* New elements will only be added to the new list when the if condition is fulfilled. This component is optional.

Let’s create a new list whose values are the doubles of the values of an existing list.

nums = [10, 20, 30, 40, 50]

# List comprehension

nums\_double = [n \* 2 for n in nums]

Let’s break down the loop above into the three components of a list comprehension.

1. The expression is equivalent to n \* 2 since it’s used to create each value in the new list.
2. Our for loop is for n in nums, where n is the iterator.
3. An if condition doesn’t exist in this case.

nums = [10, 20, 30, 40, 50]

# List comprehension

nums\_double = [n \* 2 for n in nums if n % 4 == 0]

output: [10, 20, 30, 40, 50]

[40, 80]

List comprehension can also be performed on more than one list. **The number of for loops in the comprehension will correspond to the number of lists we’re using.**

Let’s write a list comprehension which creates tuples out of the values in two lists when their sum is greater than 100. These tuples are the elements of the new list.

list1 = [30, 50, 110, 40, 15, 75]

list2 = [10, 60, 20, 50]

sum\_list = [(n1, n2) for n1 in list1 for n2 in list2 if n1 + n2 > 100]

print(sum\_list)

**[(50, 60), (110, 10), (110, 60), (110, 20), (110, 50), (75, 60), (75, 50)]**

square =list(map(lambda x: x\*\*2,range(10)))

print(square) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

a=[12,'mohammad',True,3.4,None,3+4j]

print(type(a))

for i in range(0,len(a)):

    print("This is element at index: "+str(i)+" type of "+str(type(a[i]))+" with a value of:\t "+ str(a[i]))

<class 'list'>

This is element at index: 0 type of <class 'int'> with value of: 12

This is element at index: 1 type of <class 'str'> with value of: mohammad

This is element at index: 2 type of <class 'bool'> with value of: True

This is element at index: 3 type of <class 'float'> with value of: 3.4

This is element at index: 4 type of <class 'NoneType'> with value of: None

This is element at index: 5 type of <class 'complex'> with value of: (3+4j)

print(a[-1]) # (3+4j)

**Dictionary**

It is a collection data type just like a list or a set, but there are certain features that make python dictionary unique. A dictionary in python is not ordered and is changeable as well. We can make changes in a dictionary unlike tuples or strings which are immutable in nature. Dictionary contains **key**-**value** pairs like a map that we have in other programming languages. A dictionary has indexes. Since the value of the keys we declare in a dictionary are always unique, we can use them as indexes to access the elements in a dictionary.

Dictionaries are **unordered** because the entries are not stored in a linear structure.

In Python, we must put the dictionary’s content inside curly brackets, {}

**Note**: Since the dictionary is an unordered data structure, the order of the output will not necessarily match the order in which we wrote the entries. Key-value pairs are accessed in a random or *unordered* manner.

1. Internal implementation is hash-table data structure.
2. Key is unique and value maybe duplicate.
3. dictionaries are indexed by keys, which can be any immutable type;
4. strings and numbers can always be keys
5. Tuples can be used as keys if they contain only strings, numbers, or tuples;
6. A pair of braces creates an empty dictionary: {}.

**Why Use A Dictionary In Python?**

First of all, it is not like any other object or data type in python programming language. A dictionary has key value pairs resembling a map. It is often used for unordered data with distinct key values.

**Lists vs Dictionary**

|  |  |
| --- | --- |
| **Lists** | **Dictionary** |
| Ordered | Not ordered |
| Access elements use index values | Access elements use keys as index values |
| Collection of elements | Collection of key value pairs |
| Allows duplicate members | No duplicate members |
| Preferred for ordered data | Preferred for data with unique key values |

**How To Implement A Dictionary? To declare a dictionary in python, we use the curly brackets. The keys and values are separated with a colon and the pairs are separated with a comma.** **mydictionary = { 'key1' : 'value1' , 'key2': 'value2' , 'key3': 'value3'}**

A popular practice is to create an empty dictionary and add entries later. Let’s refactor the empty\_dict and phone\_book examples to make them work with dict():

empty\_dict = dict()  # Empty dictionary

print(empty\_dict)

phone\_book = dict(Batman=468426, Cersei=237734, Ghostbusters=44678)

# Keys will automatically be converted to strings

print(phone\_book)

# Alternative approach

phone\_book = dict([('Batman', 468426),

                   ('Cersei', 237734),

                   ('Ghostbusters', 44678)])

print(phone\_book)

output:

{}

{'Ghostbusters': 44678, 'Batman': 468426, 'Cersei': 237734}

{'Ghostbusters': 44678, 'Batman': 468426, 'Cersei': 237734}

**Operations In A Python Dictionary**

**Accessing Values#**

For many, this is where a dictionary has an edge over a list or a tuple. Since there are no linear indices, we do not need to keep track of where values are stored.

Instead, we can access a value by enclosing its key in square brackets, []. This is more meaningful than the integer indices we use for tuples and lists.

Alternatively, we can use the get() method as follows: a\_dictionary.get(key)

**Accessing an element**

mydictionary = { 1: 'edureka' , 2: 'python' , 3: 'data science'}

mydictionary[1]

#this will get the key value pair with the key 1.

mydictionary.get(1)

#this is another function which will serve the same purpose.

**Replacing an element**

mydictionary = { 1: 'edureka', 2: 'python' , 3: 'data science'}

mydictionary[3] = 'artificial intelligence'

print(mydictionary)

#this will replace the value at key 3 to artificial intelligence.

**Adding/Updating Entries**[**#**](https://www.educative.io/module/lesson/data-structures-algorithms-in-python/g7ZwYyx2B9Y#Adding/Updating-Entries)

We can add new entries in a dictionary by simply assigning a value to a key. Python automatically creates the entry. If a value already exists at this key, it will be updated:

phone\_book = {"Batman": 468426,

              "Cersei": 237734,

              "Ghostbusters": 44678}

print(phone\_book)

phone\_book["Godzilla"] = 46394  # New entry

print(phone\_book)

phone\_book["Godzilla"] = 9000  # Updating entry

print(phone\_book)

**Removing an element**

To delete an entry, we can use the del keyword:

mydictionary = { 1:'edureka' , 2 : 'python', 3: 'data science'}

del mydictionary[3]

print(mydictionary)

#this will remove the key value pair from the dictionary with the specified key.

**pop**( ) – removes the element with the specified key.

a = { 1: 'edureka' , 2: 'data science' , 3: 'python' }

a.pop(3)

#this will remove the value 'python' from the dictionary.

**popitem**( ) – removes the last inserted key values pair from the dictionary.

a = { 1: 'edureka' , 2: 'python' , 3: 'data science'}

a.popitem()

#this will remove the last inserted key value pair from the dictionary.

If we want to use the deleted value, the pop() or popitem() methods would work better

phone\_book = {"Batman": 468426,

              "Cersei": 237734,

              "Ghostbusters": 44678}

print(phone\_book)

cersei = phone\_book.pop("Cersei")

print(phone\_book)

print(cersei)

# Removes and returns the last inserted pair, as a tuple

# In Python versions before 3.7, popitem() removes and returns the random item

lastAdded = phone\_book.popitem()

print(lastAdded)

{'Batman': 468426, 'Cersei': 237734, 'Ghostbusters': 44678}

{'Batman': 468426, 'Ghostbusters': 44678}

237734

('Ghostbusters', 44678)

**Checking Key Existence**

The in keyword can be used to check if a key exists in a dictionary:

phone\_book = {"Batman": 468426,

              "Cersei": 237734,

              "Ghostbusters": 44678}

print("Batman" in phone\_book) # True

print("Godzilla" in phone\_book) # False

**Following are the operations we have for dictionary in python:**

|  |  |
| --- | --- |
| 1. clear() 2. copy() 3. values() 4. update() 5. fromkeys() 6. get() | 1. items() 2. keys() 3. pop() 4. popitem() 5. setdefault() |

**clear**( ) – removes all the elements from the dictionary.

a = { 1: 2 , 2: 3 , 3: 5}

a.clear()

print(a)

#you will get a empty dictionary as the output.

**copy**( ) – returns a copy of the dictionary.

a = {1:2, 2: 3, 3: 4}

b = a.copy()

print(b)

#b will be a copy of the dictionary a.

**values**( ) – returns all the values in a dictionary.

a = {1: 2, 2: 3, 3:4}

a.values( )

#this will get you the list of all the values in the dictionary.

**update**( ) – it updates the values of the dictionary with the specified key value pairs.

a = {1 : 2, 2: 3, 3: 5}

a.update({4: 6})

#this will update the dictionary with the specified key value pair.

**fromkeys**( ) – returns a dictionary with the specified keys and values.

a = {1: 'edureka' , 2: 'data science'}

b = {1: 2, 2: 3, 3: 'python'}

a.Fromkeys(b)

#this will get the dictionary with the specified keys and values.

**items( ) –**returns the list for a tuple of each key value pair in the dictionary.

a = {1: 'edureka', 2; 'python'}

a.items()

#this will get the list of tuple for each key value pair.

**keys**( ) – returns a list containing all the keys in the dictionary.

a = { 1: 'edureka' , 2 : 'python' , 3 : 'data science'}

a.keys()

#this will get the list of all the keys from the dictionary.

**setdefault**( ) – returns the value of the specified key, if not present insert the key with the specified value.

a = { 1: 'edureka' , 2: 'python' }

a.setdefault(1, 'edureka')

**Dict( ) constructor**

Dictionary constructor is used to declare a dictionary in python.

a = dict( 1= 'edureka' , 2= 'python' , 3= 'data science')

print(a)

#this will declare a dictionary with the name a and specified key value pairs.

**Use Case – Nested Dictionary**

Nested dictionary is nothing but a dictionary which incorporates another dictionary. Let us implement a dictionary where we have the statistical data for all the Indian batsmen. We will implement a dictionary with the batsmen names and incorporate other dictionary with the statistics inside the same dictionary to make it a nested dictionary. We will also use the pandas library to get the stats into a dataframe for better understanding.

import pandas as pd

squad = {'Batsmen': {'Rohit Sharma': {'Matches': 206,

                                      'Runs': 8010,

                                      'Average':47.4,

                                      'Highest Score': 264 },

                     'Shikhar Dhawan': {'Matches':128,

                                        'Runs': 5355,

                                        'Average': 44.62,

                                        'Highest Score': 143},

                     'Virat Kohli': {'Matches': 227,

                                     'Runs': 10843,

                                     'Average': 59.58,

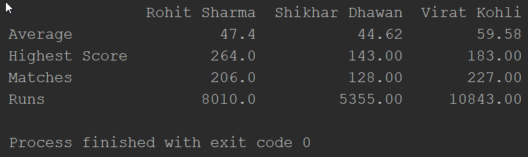
                                     'Highest Score': 183}}

df = pd.DataFrame(squad['Batsmen'])

print(df)

This will print the data frame of the stats of the batsmen from the dictionary. Similarly, you can make this dictionary for bowlers, all rounders and wicket keepers to practice the implementation of nested dictionary.

This will print the data frame of the stats of the batsmen from the dictionary. Similarly, you can make this dictionary for bowlers, all rounders and wicket keepers to practice the implementation of nested dictionary.



my\_dict = {'First': 'Python', 'Second': 'Java'}

print(my\_dict)

my\_dict['Second'] = 'C++' #changing element

print(my\_dict)

my\_dict['Third'] = 'Ruby' #adding key-value pair

print(my\_dict)

my\_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}

a = my\_dict.pop('Third') #pop element

print('Value:', a)

print('Dictionary:', my\_dict)

b = my\_dict.popitem() #pop the key-value pair

print('Key, value pair:', b)

print('Dictionary', my\_dict)

my\_dict.clear() #empty dictionary

print('n', my\_dict)

my\_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}

print(my\_dict.keys()) #get keys

print(my\_dict.values()) #get values

print(my\_dict.items()) #get key-value pairs

print(my\_dict.get('First'))

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key. Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead). To check whether a single key is in the dictionary, use the [in](https://docs.python.org/3/reference/expressions.html#in) keyword. Here is a small example using a dictionary:

>>> tel = {'jack': 4098, 'sape': 4139}

>>> tel['guido'] = 4127

>>> tel

{'jack': 4098, 'sape': 4139, 'guido': 4127}

>>> tel['jack']

4098

>>> del tel['sape']

>>> tel['irv'] = 4127

>>> tel

{'jack': 4098, 'guido': 4127, 'irv': 4127}

>>> list(tel)

['jack', 'guido', 'irv']

>>> sorted(tel)

['guido', 'irv', 'jack']

>>> 'guido' in tel

True

>>> 'jack' not in tel

False

The [dict()](https://docs.python.org/3/library/stdtypes.html#dict) constructor builds dictionaries directly from sequences of key-value pairs:

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

{'sape': 4139, 'guido': 4127, 'jack': 4098}

**Dictionary Comprehension**

Python also supports dictionary comprehensions, which work very similar to list comprehensions. We’ll be creating new key-value pairs based on an existing dictionary.

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

>>> {x: x\*\*2 for x in (2, 4, 6)}

{2: 4, 4: 16, 6: 36}

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

>>> dict(sape=4139, guido=4127, jack=4098)

{'sape': 4139, 'guido': 4127, 'jack': 4098}

However, to iterate the dictionary, we’ll use the dict.items() operation which turns a dictionary into a list of (key, value) tuples. Here’s a simple example where the keys of the original dictionary are squared and '!' is appended to each string value:

houses = {1: "Gryffindor", 2: "Slytherin", 3: "Hufflepuff", 4: "Ravenclaw"}

new\_houses = {n\*\*2: house + "!" for (n, house) in houses.items()}

print(houses)

print(new\_houses)

{1: 'Gryffindor', 2: 'Slytherin', 3: 'Hufflepuff', 4: Ravenclaw'}

{16: 'Ravenclaw!', 1: 'Gryffindor!', 4: 'Slytherin!', 9: 'Hufflepuff!'}

**Tuple**

Tuples are the same as lists are with the exception that the data once entered into the tuple cannot be changed no matter what. The only exception is when the data inside the tuple is mutable, only then the tuple data can be changed.

|  |
| --- |
| #creating a tuple  a = ('python', 'edureka')  #another approach  b = 'python' , 'edureka'  print(a)  print(b) |

**Output:** ('python' , 'edureka')

('python' , 'edureka')

>>> singleton = 'hello’, # <-- note trailing comma A special problem for tuples with 0 or 1 items.

* Tuples are immutable, and usually contain a heterogeneous element.
* Elements accessed via unpacking or indexing (or by attribute in the case of [namedtuples](https://docs.python.org/3/library/collections.html#collections.namedtuple)).
* Allows indexing

myTuple = 1,2,3,4,'hello'

print(type(myTuple)) #<class ‘tuple’>

print(myTuple) #(1,2,3,’hello’)

The statement t = 12345, 54321, 'hello!' is an example of tuple packing: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

x,y,u,z,m= myTuple

print(myTuple[0]) # 1

This is called, appropriately enough, sequence unpacking and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

**Indexing**

It is a data structure technique to effectively retrieve information from a data structure. In python, several data types support indexing like lists, string, etc.

In python, we can use negative indexing as well to access elements in a tuple or any other data type that supports indexing.

a = (1,2,3,4,5,6,7,8,9,10)

print(a[1:8])

print(a[1:])

print(a[:5])

**Output:**(2,3,4,5,6,7,8)

(2,3,4,5,6,7,8,9,10)

(1,2,3,4,5)

The index value before the slicing operator is the starting index and the index value after the slicing operator is the value that will not be included in the output. Only until the value before the ending index will be included in the output. We can even use the negative index values with the slicing operator to get the range of values from the tuple.

a = (1,2,3,4,5,6,7,8,9,10)

print(a[-8:])

**Output:** (3,4,5,6,7,8,9,10)

**Changing A Tuple**

Even though tuples in python are immutable in nature, a nested object in a tuple can be changed. Or in general, a tuple in python can be reassigned with a different value.

a = (1,2,3,[4,5])

a[3][0] = 14

print(a)

#reassigning the value

a = ('eureka', 'python')

print(a)

**Output:**(1,2,3,4,5,6,7,8,9,10)

**Concatenating Two Tuples**

Joining two tuples is a very easy task. You just to assign the addition of the two tuples to another variable and it will return the concatenated tuple with the values of both the tuples. Consider the example below to understand this.

|  |
| --- |
| a = (1,2,3,4,5)  b = (6,7,8,9,10)  c = a + b  print(c) |

**Output:**(1,2,3,4,5,6,7,8,9,10)

**Deleting A Tuple**

Being an immutable data type, a tuple in python does not allow any changes and you cannot even remove an element from a tuple after the declaration. But there is a keyword ‘del’ which will delete the tuple altogether.

a = (1,2,3,4,5)

del a

print(a)

You will get a Name error if you run the above program because there is no tuple named as present since we have deleted it.

**List vs Tuple**

|  |  |
| --- | --- |
| **List** | **Tuple** |
| Used for homogenous data types | Generally used for heterogeneous data types |
| Mutable in nature | Immutable in nature, which helps in faster iteration |
| Does not have immutable elements | Immutable elements can be used as a key for a dictionary |
| No guarantee that the data is write-protected | Implementing a tuple with data that does not change guarantees that it is write-protected |

**Iterating Through A Tuple**

Using a for loop we can iterate through a tuple in python. The following example shows how we can iterate through a tuple using a for loop.

a = ("edureka", "for data science", "for Artificial Intelligence")

for i in a:

    print("python", i)

**Output:**python edureka

python for data science

python for artificial intelligence

**Tuple Constructor**

It is possible to create a tuple using a tuple() constructor as well. We can even use the tuple constructor to change a list to a tuple.

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

b = tuple(a)

print(b)

c = tuple(('edureka', 'python'))

print(c)

**Output:**(1,2,3,4,5)

('edureka', 'python')

**Membership Test In A Tuple**

Using the membership operator ‘in’ in python we can check whether an element is present in a tuple or not. The following example shows how we can check if an element is present in a tuple or not.

**Search**

We can check whether an element exists in a tuple by using the in operator:

cities = ("London", "Paris", "Los Angeles", "Tokyo")

print("Moscow" in cities) #False

**Nested Tuples#**

In the previous coding example, instead of merging the two tuples, we could create a new tuple with these two tuples as its members:

hero1 = ("Batman", "Bruce Wayne")

hero2 = ("Wonder Woman", "Diana Prince")

awesome\_team = (hero1, hero2)

print(awesome\_team)

#(('Batman', 'Bruce Wayne'), ('Wonder Woman', 'Diana Prince'))

**Immutability#**

Since tuples are immutable, we can’t add or delete elements from them. Furthermore, it isn’t possible to append another tuple to an existing tuple.

**Sets**

Sets are a collection of unordered elements that are unique. Meaning that even if the data is repeated more than one time, it would be entered into the set only once. It resembles the sets that you have learnt in arithmetic. The operations also are the same as is with the arithmetic sets. An example program would help you understand better.

* Curly braces { } or the [set()](https://docs.python.org/3/library/stdtypes.html#set) function can be used to create sets.
* Internally implemented in hash table
* 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.
* Note: to create an empty set you have to use set()

The data is not indexed, so we can’t access elements using indices or get().

Mutable data structures like lists or dictionaries can’t be added to a set. However, adding a tuple is perfectly fine.

**Creating a set**

Sets are created using the flower braces but instead of adding key-value pairs, you just pass values to it.

my\_set = {1, 2, 3, 4, 5, 5, 5} #create set

**Adding elements**

To add a single item, we can use the add() method. To add multiple items, we’d have to use update().

The input for update() must be another set, list, tuple, or string.

Let’s add elements to an empty set:

empty\_set = set()

print(empty\_set)

empty\_set.add(1)

print(empty\_set)

empty\_set.update([2, 3, 4, 5, 6])

print(empty\_set)

**output:**

set()

{1}

{1, 2, 3, 4, 5, 6}

**Deleting Elements**

The discard() or remove() operations can be used to delete a particular item from a set:

random\_set = set({"Educative", 1408, 3.142, (True, False)})

print(random\_set)

random\_set.discard(1408)

print(random\_set)

random\_set.remove((True, False))

print(random\_set)

{1408, (True, False), 3.142, 'Educative'}

{(True, False), 3.142, 'Educative'}

{3.142, 'Educative'}

The remove() method generates an error if the item is not found, unlike the discard() method.

**Iterating a Set**

The for loop can be used on unordered data structures like sets. However, we wouldn’t know the order in which the iterator moves meaning elements will be picked randomly.

**Operations in sets**

The different operations on set such as union, intersection and so on are shown below.

set1 = {1, 2, 3, 4} set2 = {3, 4, 5, 6}

A **union** of two sets is the collection of all unique elements from both sets.

print(set1.union(set2)) or print(set1 | set2) #{1, 2, 3, 4, 5, 6}

The **intersection** of two sets is the collection of unique elements which are common between them.

print(set1 & set2) or print( set1.intersection(set2)) #{3, 4}

The **difference** between two sets is the collection of all unique elements present in the first set but not in the second.

print(set1 - set2) or print(set1.difference(set2)) # {1, 2}

* The difference() function deletes the data present in both and outputs data present only in the set passed.
* The symmetric\_difference() does the same as the difference() function but outputs the data which is remaining in both sets.

print(set1.symmetric\_difference(set2))

print(set1)

print(set2)

**Output:**

{1, 2, 5, 6}

{1, 2, 3, 4}

{3, 4, 5, 6}

Demonstrate set operations on unique letters from two words

>>> a = set('abracadabra')

>>> b = set('alacazam')

>>> a # unique letters in a

{'a', 'r', 'b', 'c', 'd'}

>>> a - b # letters in a but not in b

{'r', 'd', 'b'}

>>> a | b # letters in a or b or both

{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

>>> a & b # letters in both a and b

{'a', 'c'}

>>> a ^ b # letters in a or b but not both

{'r', 'd', 'b', 'm', 'z', 'l'}

Similarly, to [list comprehensions](https://docs.python.org/3/tutorial/datastructures.html#tut-listcomps), set comprehensions are also supported:

>>> a = {x for x in 'abracadabra' if x not in 'abc'}

>>> a

{'r', 'd'}

**Data Structure Conversions**

Explicit Conversion#

The template for explicitly converting from one data structure to another is as follows:

destination\_structure\_name(source\_structure\_object)

**Converting to a List#**

We can convert a tuple, set, or dictionary to a list using the list() constructor. In the case of a dictionary, only the keys will be converted to a list.

star\_wars\_tup = ("Anakin", "Darth Vader", 1000)

print(star\_wars\_tup)

star\_wars\_set = {"Anakin", "Darth Vader", 1000}

print(star\_wars\_set)

star\_wars\_dict = {1: "Anakin", 2: "Darth Vader", 3: 1000}

print(star\_wars\_dict)

star\_wars\_list = list(star\_wars\_tup)  # Converting from tuple

print(star\_wars\_list)

star\_wars\_list = list(star\_wars\_set)  # Converting from set

print(star\_wars\_list)

star\_wars\_list = list(star\_wars\_dict)  # Converting from dictionary

print(star\_wars\_list)

**output:**

('Anakin', 'Darth Vader', 1000)

{1000, 'Darth Vader', 'Anakin'}

{1: 'Anakin', 2: 'Darth Vader', 3: 1000}

['Anakin', 'Darth Vader', 1000]

[1000, 'Darth Vader', 'Anakin']

[1, 2, 3]

We can also use the dict.items() method of a dictionary to convert it into an iterable of (key, value) tuples. This can further be cast into a list of tuples using list():

star\_wars\_dict = {1: "Anakin", 2: "Darth Vader", 3: 1000}

print(star\_wars\_dict)

star\_wars\_list = list(star\_wars\_dict.items())

print(star\_wars\_list)

{1: 'Anakin', 2: 'Darth Vader', 3: 1000}

[(1, 'Anakin'), (2, 'Darth Vader'), (3, 1000)]

**Converting to a Tuple**

Any data structure can be converted to a tuple using the tuple() constructor. In the case of a dictionary, only the keys will be converted to a tuple:

star\_wars\_list = ["Anakin", "Darth Vader", 1000]

print(star\_wars\_list)

star\_wars\_set = {"Anakin", "Darth Vader", 1000}

print(star\_wars\_set)

star\_wars\_dict = {1: "Anakin", 2: "Darth Vader", 3: 1000}

print(star\_wars\_dict)

star\_wars\_tup = tuple(star\_wars\_list)  # Converting from list

print(star\_wars\_tup)

star\_wars\_tup = tuple(star\_wars\_set)  # Converting from set

print(star\_wars\_tup)

star\_wars\_tup = tuple(star\_wars\_dict)  # Converting from dictionary

print(star\_wars\_tup)

['Anakin', 'Darth Vader', 1000]

{1000, 'Darth Vader', 'Anakin'}

{1: 'Anakin', 2: 'Darth Vader', 3: 1000}

('Anakin', 'Darth Vader', 1000)

(1000, 'Darth Vader', 'Anakin')

(1, 2, 3)

Converting to a Set#

The set() constructor can be used to create a set out of any other data structure. In the case of a dictionary, only the keys will be converted to a set:

star\_wars\_list = ["Anakin", "Darth Vader", 1000]

print(star\_wars\_list)

star\_wars\_tup = ("Anakin", "Darth Vader", 1000)

print(star\_wars\_tup)

star\_wars\_dict = {1: "Anakin", 2: "Darth Vader", 3: 1000}

print(star\_wars\_dict)

star\_wars\_set = set(star\_wars\_list)  # Converting from list

print(star\_wars\_set)

star\_wars\_set = set(star\_wars\_tup)  # Converting from tuple

print(star\_wars\_set)

star\_wars\_set = set(star\_wars\_dict)  # Converting from dictionary

print(star\_wars\_set) # Same output as above

Converting to a Dictionary#

The dict() constructor cannot be used in the same way as the others because it requires key-value pairs instead of just values. Hence, the data must be stored in a format where pairs exist.

For example, a list of tuples where the length of each tuple is 2 can be converted into a dictionary.

Those pairs will then be converted into key-value pairs:

star\_wars\_list = [[1,"Anakin"], [2,"Darth Vader"], [3, 1000]]

print (star\_wars\_list)

star\_wars\_tup = ((1, "Anakin"), (2, "Darth Vader"), (3, 1000))

print (star\_wars\_tup)

star\_wars\_set = {(1, "Anakin"), (2, "Darth Vader"), (3, 1000)}

print (star\_wars\_set)

star\_wars\_dict = dict(star\_wars\_list) # Converting from list

print(star\_wars\_dict)

star\_wars\_dict = dict(star\_wars\_tup) # Converting from tuple

print(star\_wars\_dict)

star\_wars\_dict = dict(star\_wars\_set) # Converting from set

print(star\_wars\_dict)

[[1, 'Anakin'], [2, 'Darth Vader'], [3, 1000]]

((1, 'Anakin'), (2, 'Darth Vader'), (3, 1000))

{(3, 1000), (1, 'Anakin'), (2, 'Darth Vader')}

{1: 'Anakin', 2: 'Darth Vader', 3: 1000}

{1: 'Anakin', 2: 'Darth Vader', 3: 1000}

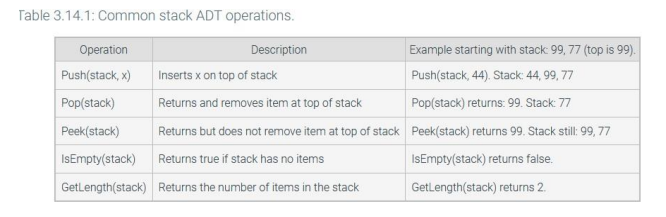
{1: 'Anakin', 2: 'Darth Vader', 3: 1000}

There are also other ways to convert between data structures that don’t involve explicit conversion.

We urge you to explore such methods on your own.

Stack

A stack is an ADT in which items are only inserted on or removed from the top of a stack. The stack push operation inserts an item on the top of the stack. The stack pop operation removes and returns the item at the top of the stack. Ex: After the operations "Push 7", "Push 14", "Push 9", and "Push 5", "Pop" returns 5. A second "Pop" returns 9. A stack is referred to as a last-in first-out ADT. A stack can be implemented using a linked list, an array, or a vector.



Stack Data Structure.

"""

|  |  |  |
| --- | --- | --- |
| class Stack():  def \_\_init\_\_(self):  self.items = []  def push(self,item):  self.items.append(item)    def pop(self):  return self.items.pop()    def peek(self):  return self.items[-1]    def getLength(self):  return len(self.items)    def isEmpty(self):  return self.items == list()    def getStack(self):  return self.items | myStack = Stack()  print(myStack.isEmpty())  myStack.push('A')  myStack.push('B')  myStack.push('C')  myStack.push('D')  print(myStack.isEmpty())  print(myStack.getStack())  print(myStack.peek())  print(myStack.getStack())  print(myStack.getLength())  print(myStack.pop())  print(myStack.getStack()) | True  False  ['A', 'B', 'C', 'D']  D  ['A', 'B', 'C', 'D']  4  D  ['A', 'B', 'C'] |

For the sake of this course, we will go over the following different types of linked lists and implement them in Python:

1. Singly Linked Lists
2. Doubly Linked Lists
3. Circular Linked List

**Arrays vs. Linked Lists**

|  |  |  |
| --- | --- | --- |
|  | Arrays | Linked Lists |
| Insertion/Deletion at the beginning of the array or linked list given a value | O(n) | O(1) |
| Access Element | O(1) | O(n) |
| Contiguous Memory | Yes | No |