**5. Data-Types Introduction**

* 1. Numbers Page 1
  2. String Page7

bool

* 1. List Page 18
  2. Tuple Page 22
  3. Dictionary Page 25
  4. Set Page 31

**Data Types vs Data Structures**

**CRUD : Create Retrieve(Read) Update Delete**

**===============================================**

**age = 25**

**type() id()**

**len() min() max() -- sequencecs**

**Numbers 6 int float long complex**

**String 40**

**List 15**

**Tuple 3**

**Dictionary 10**

**Set 5**

**------------------**

**Collections**

**OrderdDict**

**DefaultDict**

**Counter**

**Deque**

**HTTP FTP UDP**

**m1 | 1 2 3 |**

**| 4 5 6 |**

**| 7 8 9 |**

**m1[2][2]**

**m1[2][3]**

**Sequnce , Series :**

**----------------------**

KG Litres

Every **value** in Python has a datatype . Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classe

**Type** represents the kind of value and determines how the value can be used. All data values in Python are encapsulated in relevant object classes. Everything in Python is an object and every object has an identity, a type, and a value.

**Java : String age = 20;**

**Stores value. And determines type**

**Based on type of value,It will compare/matches with variable type.**

**20 is a value which is being assigned to variable age,And is of type int**

**Python: age = 20**

**Stores value.And determines type**

**While referring to variable age ,directly it will refer and remembers age is of type int.**

**20 is value which is being assigned to variable age**

Like another object-oriented language such as Java or C++, there are several data types which are built into Python. Extension modules which are written in C, Java, or other languages can define additional types.

To determine a variable's type in Python you can use the **type()** function.

**5.1 Numbers:**

**Python supports** four different numerical types

* **int (signed integers)** − They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
* **long (long integers )** − Also called longs, they are integers of **unlimited size**, written like integers and followed by an uppercase or lowercase L.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

* + 7 6 5 4 3 2 1 0 =0
  + 2 2 2 2 2 2 2 2 =255

0 to 255 ASCII char = “Python”

A to Z a to z !@#$%^&\*(\*)\_\_+= 1234567890 \|

* **float (floating point real values)** − Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).
* **complex (complex numbers)** − are of the form a + bJ, where a and b are floats

**5+4j**

**5 –real no.**

**4 imaginary no.**

and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

|  |  |  |  |
| --- | --- | --- | --- |
| **Int** | **Long** | **Float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9456812345867  48987 | 9.322e-36j |
| 080 | 0Xdefabcecbdaecbfbael | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

* Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
* A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

## Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter. int < float < long

Type **int(x)** to convert x to a plain integer.

* Type **long(x)** to convert x to a long integer.
* Type **float(x)** to convert x to a floating-point number.
* Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
* Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

**CRUD operations on DATA:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **SQL (Backend)** | **HTTP method (UI)** | **Data Distribution System** |
| CREATE | INSERT | POST | Write |
| READ/RETRIEVE | SELECT | GET | Read / take |
| UPDATE | UPDATE | PUT | Write |
| DELETE | DELETE | DELETE | Dispose / delete |

**CRUD operations on employee data in organization:**

**Create/Insert : Ex :** create,insert new employee details **# if new employee joins**

**Retrieve : Ex:** select all employees data **# To view information of all employees**

**Update : Ex:** update **#** Salary hike,name update,address change

**Delete : Ex:** delete his data # If employee leaves organisation

**5.2 String:**

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable.

For example – -3 -2 – 1 0 1 2 3

message = 'Hello World' x = 10

## Diagram string Hello World

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ….. |  |  |  |  |  |  |  |

## 5786548522

## Accessing Values in Strings:

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

For example –

var1 ='Hello World!'

var2 ="Python Programming"

print"var1[0]: ", var1[0]

print"var2[1:5]: ", var2[1:5]

var1[0]: H

var2[1:5]: ytho

## Updating Strings

## You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

## For example −

var1 ='Hello World!'

print"Updated String :- ", var1[:6]+'Python'

Updated String :- Hello Python

## Triple Quotes:

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of **three consecutive single or doublequotes.**

para\_str ="""this is a long string that is made up of

several lines and non-printable characters such as

TAB ( \t ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [ \n ], or just a NEWLINE within

the variable assignment will also show up.

"""

print para\_str

this is a long string that is made up of

several lines and non-printable characters such as

TAB ( ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [

], or just a NEWLINE within

the variable assignment will also show up.

print'C:\\nowhere'

C:\nowhere

print'C:\\nowhere'

C:\\nowhere

## 1byte = 8 bits

## 0 to 255 2 \*\* 8

## -65532 to 65531 2 \*\* 16

## Unicode String:

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following –

print('Hello, world!')

Hello, world!

**5.3 Lists:**

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets.

\*\***items in a list need not be of the same type.**

**X =**

1. **2 3**

**4 6 9**

**5 9 2**

**X[3][2]**

List<String> list1 = new ArrayList<String>();

List1.add(‘physics’)

List1.add(‘chemistry’)

list1 = [500, 'physics', 'chemistry', 1997]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | P | H | Y | S | I | C | S | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | C | H | E | M | I | S | T | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | |  |

0 1 2 3

list2 = [1, 2, 3, 4, 5 ]

list3 = ["a", "b", "c", "d"]

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on. [index **:** position **:** step] CRUD 10

X = 10

X = 20

**x**

**12345**

**20**

**342324**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **11001001** | **10100101010** | **1o1ooo1** | |  |  |  | | --- | --- | --- | | **010101** | **101001** | **10001** | | **0** | **1** | **2** | | **101001** | **100101** |

**0 1 2 3 4 5**

**List operations : C R U D**

1. **Accessing** values in Lists:

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index

list1 =['physics','chemistry',1997,2000];

list2 =[1,2,3,4,5,6,7];

print("list1[0]: ",list1[0])

print"list2[1:5]: ", list2[1:5]

list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

1. **Updating** values in Lists:

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator,

list =['physics','chemistry',1997,2000];

print"Value available at index 2 : "

print list[2]

list[2]=2001;

print"New value available at index 2 : "

print list[2]

Value available at index 2 :

1997

New value available at index 2 :

2001

1. **Delete** List Elements in Lists:

To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting or the **remove()** method if you do not know.

For example −

list1 =['physics','chemistry',1997,2000]

print list1

del list1[2];

print"After deleting value at index 2 : "

print list1

['physics', 'chemistry', 1997, 2000]

After deleting value at index 2 :

['physics', 'chemistry', 2000]

**5.4 Tuples:**

A tuple is a sequence of **immutable** Python objects.

Tuples are sequences, just like lists.

The differences between tuples and lists are, the tuples c**annot be changed** unlike lists and tuples use **parentheses**, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values.

Optionally you can put these comma-separated values between parentheses also.

For example −

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";

The empty tuple is written as two parentheses containing nothing −

tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value −

tup1 = (50,);

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

**Accessing Values in Tuples:**

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

#!/usr/bin/python

tup1 =('physics','chemistry',1997,2000);

tup2 =(1,2,3,4,5,6,7);

print"tup1[0]: ", tup1[0];

print"tup2[1:5]: ", tup2[1:5];

When the above code is executed, it produces the following result −

tup1[0]: physics

tup2[1:5]: (2, 3, 4, 5)

## Updating Tuples:

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

tup1 =(12,34.56);

tup2 =('abc','xyz');

# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print tup3;

When the above code is executed, it produces the following result −

(12, 34.56, 'abc', 'xyz')

## Delete Tuple Elements:

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the del statement.

For example −

tup =('physics','chemistry',1997,2000);

print tup;

del tup;

print"After deleting tup : ";

print tup;

This produces the following result. Note an exception raised, this is because after  **del tup** tuple does not exist any more −

('physics', 'chemistry', 1997, 2000)

After deleting tup :

Traceback (most recent call last):

File "test.py", line 9, in <module>

print tup;

NameError: name 'tup' is not defined

**5.5 Dictionary:**

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.

An empty dictionary without any items is written with just two curly braces, like {}

**Keys** are **unique** within a dictionary while **values may not be**.

The values of a dictionary can be of any type, but the **keys must be of an immutable** data type such as **strings, numbers, or tuples**.

## Accessing Values in Dictionary:

## To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

## Following is a simple example −

dict = {'Name':'Zara',

'Age':7,

'Class':'First'

}

print"dict['Name']: ", dict['Name']

print"dict['Age']: ", dict['Age']

dict['Name']: Zara

dict['Age']: 7

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows –

dict ={'Name':'Zara','Age':7,'Class':'First'}

print"dict['Alice']: ", dict['Alice']

dict['Alice']:

Traceback (most recent call last):

File "test.py", line 4, in <module>

print "dict['Alice']: ", dict['Alice'];

KeyError: 'Alice'

## Updating Dictionary:

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

dict ={'Name':'Zara','Age':7,'Class':'First'}

dict['Age']=8;# update existing entry

dict['School']="DPS School";# Add new entry

print"dict['Age']: ", dict['Age']

print"dict['School']: ", dict['School']

dict['Age']: 8

dict['School']: DPS School

## Delete Dictionary Elements:

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

dict ={'Name':'Zara','Age':7,'Class':'First'}

del dict['Name'];# remove entry with key 'Name'

dict.clear();# remove all entries in dict

del dict ;# delete entire dictionary

print"dict['Age']: ", dict['Age']

print"dict['School']: ", dict['School']

an exception is raised because after **del dict** dictionary does not exist any more

dict['Age']:

Traceback (most recent call last):

File "test.py", line 8, in <module>

print "dict['Age']: ", dict['Age'];

TypeError: 'type' object is unsubscriptable

**5.6 Sets:**

A **Set** is an unordered collection data type that is **iterable**, **mutable**, and has **no duplicate elements**.

Python’s set class represents the mathematical notion of a set.

The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

This is based on a data structure known as a[**hash table**](https://www.geeksforgeeks.org/hashing-set-1-introduction/).

**Frozen Sets** : Frozen sets are immutable objects that only support methods and operators that produce a result without a?ecting the frozen set or sets to which they are applied.

|  |
| --- |
| # Python program to demonstrate differences  # between normal and frozen set    # Same as {"a", "b","c"}  normal\_set =set(["a", "b","c"])    # Adding an element to normal set is fine  normal\_set.add("d")    print("Normal Set")  print(normal\_set)    # A frozen set  frozen\_set =frozenset(["e", "f", "g"])    print("Frozen Set")  print(frozen\_set)    # Uncommenting below line would cause error as  # we are trying to add element to a frozen set  # frozen\_set.add("h") |

Output:

Normal Set

set(['a', 'c', 'b', 'd'])

Frozen Set

frozenset(['e', 'g', 'f'])