Day_02 Datatypes(String & List & Dictionaries & Tuples)

Strings

Definition

A string in Python is a sequence of characters. It is a derived data type. Strings are immutable. This means that once defined, they cannot be changed. Many Python methods, such as replace(), join(), or split() modify strings. However, they do not modify the original string. They create a copy of a string which they modify and return to the caller.

Purpose

string is a data type used in programming, such as an integer and floating point unit, but is used to represent text rather than numbers. It is comprised of a set of characters that can also contain spaces and numbers. For example, the word "hamburger" and the phrase "I ate 3 hamburgers" are both strings

Importance

As with any programming language, strings are one of the most important things to know about Python. Also as we have experienced in the other languages so far, strings contain characters. Strings are not picky by any means. They can contain almost anything if used properly. The are also not picky by the amount of characters you put in them.

Application

Strings are so widely used component in the any programming language that thaere are so many application of it in the programmings.Like in any program with text data strings are used like in database to store names,addresses and so much.

Strengths

Compilation creates unique strings. At compile time, strings are resolved as far as possible. This includes applying the concatenation operator and converting other literals to strings. So “hi7” and (“hi”+7) both get resolved at compile time to the same string and are identical objects in the class string pool. Compilers differ in their ability to achieve this resolution. You can always check your compiler (e.g., by decompiling some statements involving concatenation) and change it if needed.

Because String objects are immutable, a substring operation doesn’t need to copy the entire underlying sequence of characters. Instead, a substring can use the same char array as the original string and simply refer to a different start point and endpoint in the char array. This means that substring operations are efficient, being both fast and conserving of memory; the extra object is just a wrapper on the same underlying char array with different pointers into that array.

Strings are implemented in the JDK as an internal char array with index offsets (actually a start offset and a character count). This basic structure is extremely unlikely to be changed in any version of Java.

Strings have strong support for internationalization. It would take a large effort to reproduce the internationalization support for an alternative class.

The close relationship with StringBuffers allows Strings to reference the same char array used by the StringBuffer. This is a double-edged sword. For typical practice, when you use a StringBuffer to manipulate and append characters and data types and then convert the final result to a String, this works just fine.

The StringBuffer provides efficient mechanisms for growing, inserting, appending, altering, and other types of String manipulation. The resulting String then efficiently references the same char array with no extra character copying. This is very fast and reduces the number of objects being used to a minimum by avoiding intermediate objects. However, if the StringBuffer object is subsequently altered, the char array in that StringBuffer is copied into a new char array that is now referenced by the StringBuffer.

The String object retains the reference to the previously shared char array. This means that copying overhead can occur at unexpected points in the application. Instead of the copying occurring at the toString( ) method call, as might be expected, any subsequent alteration of the StringBuffer causes a new char array to be created and an array copy to be performed. To make the copying overhead occur at predictable times, you could explicitly execute some method that makes the copying occur, such as StringBuffer.setLength( ). This allows StringBuffers to be reused with more predictable performance.

Weakness

Not being able to subclass String means that it is not possible to add behavior to String for your own needs.

The previous point means that all access must be through the restricted set of currently available String methods, imposing extra overhead.

The only way to increase the number of methods allowing efficient manipulation of String characters is to copy the characters into your own array and manipulate them directly, in which case String is imposing an extra step and extra objects you may not need.

char arrays are faster to process directly.

The tight coupling with StringBuffer can lead to unexpectedly high memory usage. When StringBuffer.toString( ) creates a String, the current underlying array holds the string, regardless of the size of the array (i.e., the capacity of the StringBuffer).

For example, a StringBuffer with a capacity of 10,000 characters can build a string of 10 characters. However, 10-character String continues to use a 10,000-char array to store the 10 characters. If the StringBuffer is now reused to create another 10-character string, the StringBuffer first creates a new internal 10,000-char array to build the string with; then the new String also uses that 10,000-char array to store the 10 characters. Obviously, this process can continue indefinitely, using vast amounts of memory were not expected.

Example 04 :

Task:

Take three variables of String type and store information in them and print them

In [1]:
```
# Code
firstName="Muhammad"
lastName="Umair"
university="Comsats Lahore"
```

In [2]:
```
# Output
print(firstName,lastName,university)
```

```
Muhammad Umair Comsats Lahore
```

Example 05 :

Task:

Use Slicing to print a substring from the string and print it

In [3]:
```
# Code
fullName="Muhammad Umair Akram"
firstName=fullName[0:8]
middleName=fullName[9:14]
lastName=fullName[15:20]
# Output
print(firstName,middleName,lastName)
```

```
Muhammad Umair Akram
```

Example 06 :

Task:

Use + Operator to add(join) two strings togeather and print the output

In [4]:
```
greetingsStr="Hy I am "+middleName+".Nice to meet You"
print(greetingsStr)
```

```
Hy I am Umair.Nice to meet You
```

In [5]:
```
brotherName=fullName[:9]+"Usama Akram"
```

In [6]:
```
print(brotherName)
```

```
Muhammad Usama Akram
```

Example 07 :

Task:

Use the del operator to delete the string and check if the string is realy deleted

In [7]:
```
del greetingsStr
```

In [8]:
```
print(greetingsStr)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-e9c047401d1c> in <module>
----> 1 print(greetingsStr)

NameError: name 'greetingsStr' is not defined
```

Example 08 :

Task:

use the formated string(use + to join the two strings in print function and print them)

In [ ]:
```
print("Hello "+middleName+" How are You?")
```

In [ ]:
```
newStr="I Love Pakistan"
print(((newStr+" "+"\n")*10))
```

Example 09 :

Task:

Use slicing to print chunks from the string

In [ ]:
```
print(newStr[7:])
```

In [ ]:
```
print(newStr[7])
```

In [ ]:
```
print((newStr[2:6]+"\t")*5)
```

Example 10 :

Task:

Use use in operator to serach that specified string in the complete string

In [ ]:
```
print("Umair" in fullName)
```

In [ ]:
```
print("Usama" in fullName)
```

In [ ]:
```
print("Usama" not in fullName)
```

Example 11 :

Task:

Use use String Formator to print the formated string using print function

In [ ]:
```
print("I am %s. I am %d years old. I love to eat Meat"%(name,age))
```

In [ ]:
```
print("I am {name}. I am {age} years old. I love to eat Biryani".format(name=name,age=age))
```

Lists

Difinition

A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ]

Purpose

They are used to store an ordered collection of items, which might be of different types but usually they aren’t. Commas separate the elements that are contained within a list and enclosed in square brackets

Importance

Lists are just like the arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. In Python, list is a type of container in Data Structures, which is used to store multiple data at the same time. Unlike Sets, the list in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

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.Important thing about a list is that items in a list need not be of the same type.List is majorly used with dictionaries when there is large number of data.

Applications

Due to the verstality of hte lists in python they are widely used in puthon to hold huge ammount of data of different datatypes.

Such as there can be the list of lists,tuple,dictionaries.They are used to hold image type data after converting them.

Strengths

Add/remove any objects/items to/from it.Mutable i.e. you can change the items anytime you like

Lists are numerically keyed and can be sorted and have values removed or added.

Lists are mutable. Tuples and Dictionaries are immutable so their values are fixed. There can be no precise explanation to your question since the real question is what do you want your program to do? If you want to collect values and store them somewhere then Lists is the solution you are searching for, since they can be mutated. If you want to get values from something like a database then Tuples or Dictionaries might be a good solution. Dictionaries give you the perk of a key usage also. So it all depends on what you want to achieve.

Weakness

You cannot use a list as a key to a dictoinary because it is mutable which refers to the fact that we want a key to be a constant (non-changing entity).

Example 12 :

Task:

make the list of names and numbers and print them

In [ ]:
```
namesList=["Umair","Usama","Akram","Bilal","Abubakar","Umar","Ali"]
rollNoList=[1,2,3,4,5,6,7]
print(namesList[0],rollNoList[0])
print(namesList[0:7],rollNoList[0:7])
print(namesList[5:7],rollNoList[5:7])
print(namesList)
```

In [ ]:
```
print(rollNoList)
```

Example 13 :

Task:

Append the list or add new string into the list using append function print the output

In [ ]:
```
namesList.append("Usman")
```

In [ ]:
```
print(namesList)
```

Example 14 :

Task:

Delete an Element from the list using del keyword print and check the output

In [ ]:
```
del namesList[3]
```

In [ ]:
```
print(namesList)
```

Example 15 :

Task:

Remove the Element from the list using remove function of list

In [ ]:
```
rollNoList.remove(4)
```

In [ ]:
```
print(namesList+rollNoList)
```

Dictionaries

Definition

Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

Purpose

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair. Key value is provided in the dictionary to make it more optimized

Importance

The Python dictionary makes it easier to read and change data, thereby rendering it more actionable for predictive modeling. A Python dictionary is an unordered collection of data values. Unlike other data types that hold only one value as an element, a Python dictionary holds a key: value pair

Strenghts

The Python dictionary makes it easier to read and change data, thereby rendering it more actionable for predictive modeling. A Python dictionary is an unordered collection of data values. Unlike other data types that hold only one value as an element, a Python dictionary holds a key: value pair.

It improves the readability of your code. Writing out Python dictionary keys along with values adds a layer of documentation to the code. If the code is more streamlined, it is a lot easier to debug. Ultimately, analyses get done a lot quicker and models can be fitted more efficiently.

Apart from readability, there’s also the question of sheer speed. You can look up a key in a Python dictionary very fast. The speed of a task like looking up keys is measured by looking at how many operations it takes to finish. Looking up a key is done in constant time vis-a-vis looking up an item in a large list which is done in linear time.

To look up an item in a huge list, the computer will look through every item in the list. If every item is assigned a key-value pair then you only need to look for the key which makes the entire process much faster. A Python dictionary is basically an implementation of a hash table. Therefore, it hs all the benefits of the hashtable which include membership checks and speedy tasks like looking up keys.`

Weakness

Dictionaries are unordered. In cases where the order of the data is important, the Python dictionary is not appropriate.

Python dictionaries take up a lot more space than other data structures. The amount of space occupied increases drastically when there are many Python Dictionary keys. Of course, this isn’t too much of a disadvantage because memory isn’t very expensive.

Example 16 :

Task:

Create an Dictionary and print different values from it delete an element from dictionary using del keyword and add new element in the dictionary

In [ ]:
```
studentsDict={'Student1':{'name':"Muhammad Umair",'age':20,'Department':"BSE",'Semester':6},
             'Student2':{'name':"Hashim Shakoor",'age':22,'Department':"BSE",'Semester':6},
            'Student3':{'name':"Muhammad Abdullah Tahir",'age':20,'Department':"BSE",'Semester':6}
             }
```

In [ ]:
```
studentsDict['Student1']
```

In [ ]:
```
studentsDict['Student1']['name']
```

In [ ]:
```
studentsDict['Student1']['age']
```

In [ ]:
```
studentsDict['Student1']['Department']
```

In [ ]:
```
studentsDict['Student3']['Department']='BCS'
```

In [ ]:
```
studentsDict['Student3']['Department']
```

In [ ]:
```
del studentsDict['Student2']
```

In [ ]:
```
print(studentsDict)
```

In [ ]:
```
studentsDict['Student2']={'name':"Hamdan Ijaz",'age':22,'Department':"BSE",'Semester':6}
```

In [ ]:
```
studentsDict
```

Tuples

Difinition

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Purpose

A tuple lets us “chunk” together related information and use it as a single thing. Tuples support the same sequence operations as strings. ... So like strings, tuples are immutable. Once Python has created a tuple in memory, it cannot be changed.

Importance

tuples are immutable. The reasons for having immutable types apply to tuples:

copy efficiency: rather than copying an immutable object, you can alias it (bind a variable to a reference)

comparison efficiency: when you're using copy-by-reference, you can compare two variables by comparing location, rather than content

interning: you need to store at most one copy of any immutable value

there's no need to synchronize access to immutable objects in concurrent code

const correctness: some values shouldn't be allowed to change. This (to me) is the main reason for immutable types.

immutable objects can allow substantial optimization; this is presumably why strings are also immutable in Java, developed quite separately but about the same time as Python, and just about everything is immutable in truly-functional languages.

in Python in particular, only immutables can be hashable (and, therefore, members of sets, or keys in dictionaries). Again, this afford optimization, but far more than just "substantial" (designing decent hash tables storing completely mutable objects is a nightmare -- either you take copies of everything as soon as you hash it, or the nightmare of checking whether the object's hash has changed since you last took a reference to it rears its ugly head).

Strengths

Allows you to output the whole tuple

Allows you to output a specific element

Allows you to combine

Allows you find an item using the index function

Allows you to calculat the length of your tuple

Weakness

You can't add an element but in a list you can

You can't sort a tuple but in a list you can

You can't delete an element but you can in a list

You can't replace an element but you can in a list

Example 17 :

Task:

Create a tuple store some values in it print them and add new value in the tuple using + operator delete and element from the tuple using del keyword use the type

In [ ]:
```
studentsResult=(("Muhammad Umair",3.06),("Hamdad Ijaz",2.8),("Muhammad Abdullah Tahir",2.7))
```

In [ ]:
```
print(studentsResult)
```

In [ ]:
```
print(studentsResult[0],studentsResult[1],studentsResult[2])
```

In [ ]:
```
print(studentsResult[1:2])
```

In [ ]:
```
newStudent=(("Muhammad Aaqib Munir",2.5))
```

In [ ]:
```
# studentsResult+=newStudent
```

In [ ]:
```
print(studentsResult)
```

In [ ]:
```
del studentsResult[2]
```

In [ ]:
```
studentsResult=studentsResult-newStudent
```

In [ ]:
```
# del studentsResult
```

In [ ]:
```
name="Muhammad Umair"
```

In [ ]:
```
type(name)
```

In [ ]:
```
name=5
```

In [ ]:
```
type(name)
```

In [ ]:
```
```