Python

07-11-2022

Course Indtroduction

1.Please watch the course overview lecture, lots of useful info there!

2. Our course notebooks can be found at this link: <https://github.com/Pierian-Data/Complete-Python-3-Bootcamp>

3. We go over downloading and using the notebooks in the lecture videos, so don't worry about how to use the link above, we will walk you through it all.

4. You can ask questions in the QA Forums (make sure to search first, before posting a new question as there are already 60,000+ answered questions) Guide on using the QA Forums:

https://support.udemy.com/hc/en-us/articles/229233387-How-to-Use-The-Q-A

5. Our chat channel to talk to other students can be found here: https://discord.gg/TztE6B8

6. Video guide on how to use the chat room: https://www.youtube.com/watch?v=bkH89OJ001M

7. I don't control certification, Udemy does, information on this can be found here:  https://support.udemy.com/hc/en-us/articles/229603868-Certificate-of-Completion

8. Course Slides can be found here: https://drive.google.com/drive/folders/1CKqOQzst1cGURXGiRVivi2Xsc0n-X8CR?usp=sharing

Course overview

Python

**History**

* It is created in 1990 by guido van Rossum
* Python 3 release in 2008
* Specifically designed as an easy to use language
* High focus on readability of code

**Why python**

* Designed for clear, logical code that is easy to learn and read
* Lots of existing libraries and frameworks are written in python allowing users to apply python to a wide variety of tasks
* Focus on optimizing developer time, rather than a comp processing time

**What can you do with Python**

* This course first focuses on “base Python, which consists of the core components of the language and writing scripts and small programs.
* Later we begin to learn about outside libraries and frameworks that greatly expand python’s capabilities

**FAQ’s**

**How do I open .ipynb files? What program do I choose? (We show this in a video in the next section)**

After installing Anaconda, search your computer for Anaconda Navigator, then launch jupyter notebook and move through the Jupyter Notebook interface until you reach your desired directory where you have your .ipynb files. Please note, you can not just double click a .ipynb file, it won't open that way.

Using Juptyer Notebook at your command line? Here are the instructions to open .ipynb files through your command line:

In order to open the Notebook Files you'll need to have Python and the Jupyter Notebook system installed, check out the Python Set-up section for more details on the installation of Python and the Jupyter Notebook system (or you can just follow the relevant instructions [here](http://jupyter.readthedocs.org/en/latest/install.html) if you feel more technical). Once you have python and the jupyter notebooks installed you are ready to open the notebooks using the following steps:

1. First open up your Command Prompt (search for **cmd** on a Windows machine) or if you are on a Mac use your terminal (Spotlight search for **terminal**).
2. Next in you terminal/command prompt type **pwd** and press enter (this will print your working directory)
3. Take note of what file directory was displayed, this is where you should save your **.ipynb** files (or a folder containing your **.ipynb** files)
4. Once your **ipynb**files or folder containing the files is in the location displayed from the **pwd** step go back to your terminal and type **jupyter notebook** and press Enter.
5. After Step 4 you should have a browser tab open up with the Jupyter Notebook system running inside of it.
6. Click on your Notebook (or go to your folder of Notebooks) displayed in the Jupyter Notbeook and it will open in a new tab with the Notebook you selected.
7. You should now have successfully opened a Notebook file.

08-11-2022

Command line basics

It will be useful to use the python commands and running the python programs

Find your current directory

Terminal🡪pwd 🡪will give the current working directory

Listing all files in a directory

Terminal🡪ls 🡪will give the list of files in the current working directory

Change directory

Terminal🡪cd 🡪used to change the directory

Terminal🡪cd .. 🡪used to go back from the current directory

Clear the commands

Terminal🡪clear 🡪used to clear the terminal

Installation

Running the python code

Text editors—sublime text,atom—small lightweight programs

Full IDEs –Pycharm and spyder --- large nd heavy programs

Notebook—Jupyter Notebooks--- we can write codes and notes aswell

Sublime text write the program in the sublime text and save it with .py extention

And navigate the terminal to that folder and give the command as—python “filename”

09/11/2022

Python objects and data types

**Data types:-**

These are the basic building blocks when we constructing the large piece of code

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Integers | int | Whole numbers, such as:    **3**     **300**    **200** |
| Floating point | float | Numbers with a decimal point:**2.3     4.6    100.0** |
| Strings | str | Ordered sequence of characters:   **"hello"**   **'Sammy'   "2000"  "楽しい"** |
| Lists | list | Ordered sequence of objects:   **[10,"hello",200.3]** |
| Dictionaries | dict | Unordered Key:Value pairs:  **{"mykey" : "value" , "name" : "Frankie"}** |
| Tuples | tup | Ordered immutable sequence of objects: **(10,"hello",200.3)** |
| Sets | set | Unordered collection of unique objects:  **{"a","b"}** |
| Booleans | bool | Logical value indicating **True** or **False** |

Tuple-it is immutable that means we cannot change the object in that seq

**Python Numbers:**

1. Integers—whole numbers
2. Floating point numbers--- decimal numbers

* **Rules for variable names**
  + Names can not start with a number.
  + There can be no spaces in the name, use \_ instead.
  + Can't use any of these symbols :'",<>/?|\()!@#$%^&\*~-+
* Python uses **Dynamic Typing**
* This means you can reassign variables to different data types.

This makes Python very flexible in assigning data types, this is different than other languages that are **“Statically-Typed”**

**my\_dogs = 2**

**my\_dogs = [ “Sammy” ,  “Frankie” ]**

* Pros of Dynamic Typing:
  + Very easy to work with
  + Faster development time
* Cons of Dynamic Typing:
  + May result in bugs for unexpected data types!
  + You need to be aware of **type()**

**12-11-2022**

Strings

* Strings are sequences of characters, using the syntax of either single  quotes or double quotes:

1. It will help in when ever there is a single Quotes in a scentence or word
2. We can use either single quote or double quote
3. Because strings are **ordered sequences** it means we can using **indexing** and **slicing** to grab sub-sections of the string.
4. Indexing notation uses [ ] notation after the string (or variable assigned the string).
5. Indexing allows you to grab a single character from the string...

**Character :    h     e     l       l     o**

**Index :     0     1     2      3    4**

1. Indexing starts at zero in python
2. It has reverse indexing we can grad the last letter of the word directly

**Character :    h     e     l       l     o**

**Index :     0     1     2      3    4**

**Reverse Index:    0    -4    -3   -2    -1**

1. In python we have slicing concept that is
2. Slicing allows you to grab a subsection of multiple characters, a “slice” of the string.
3. This has the following syntax:
   1. **[start:stop:step]—index numbers**
4. **start** is a numerical index for the slice start
5. **stop** is the index you will go up to (but not include)
6. **step** is the size of the “jump” you take.
7. **In notebook without print statement it will always give the output as it is**
8. **If we give “hello” it will give “hello**
9. **Or if we give 2 statements it will return only last statement so for that we have use print statements so that we can see the actual print result**
10. **“\n” escape sequence ---new line**

**Ex:-print(“hello \n world”)-----**

**o/p----hello**

**world**

1. **“\t” tab spce**

**Ex:-print(“hello \t world”)-----**

**o/p----hello world**

1. **Len length function –it gives the length of the string**

**Ex. len(“hello”)---5**

**len(“I am”)---4 --🡪 it will takethe spaces blw the characters and it will take the special characters as well**

**Ex:-**

len("i'm the only one here")

21

>>> len("im the only one here")

20

>>>

**String Indexing and string slicing**

**Example-string indexing**

>>> mystring="Hi Sowmya"---------define a string

>>> mystring

'Hi Sowmya'------string output

>>> mystring[2]-------string indexing at 2

' '

>>> mystring[1]-------- string indexing at 1

'i'

**Example- reverse string indexing**

>>> mystring

'Hi Sowmya'

>>> mystring[-2]

'y'

>>>

**Example-string slicing**

>>> mystring="i'm the one here"------string

>>> mystring

"i'm the one here"

>>> mystring[:11]—----- o/p from that index to last (stop index)

"i'm the one"

>>> mystring[8:11]------o/p we are giving the step and stop points

'one'

>>> mystring[4:]—----- o/p upto that index not including that index number(starting. Number)

'the one here'

>>>

**To grab the entire string we can use string name or ---stringname[::] this syntax us also valid**

**If we give the step size --it will jump the index positions**

>>> mystring

'ABCDEFGH'

>>> mystring[::2]

'ACEG'

>>> mystring[::3]

'ADG'

>>> mystring[1:5:2]---starts from 1 /stop at 5/jump over every 2 positions

'BD'

>>> mystring[1:5]

'BCDE'

>>> mystring[1:5:2]

'BD'

>>>

In python for reversing a string we can use the step position

>>> mystring[::-1]---all the way to begin and end and backward step to (-1)—so it can reverse a string

'HGFEDCBA'

>>>

**String properties and methods**

**Immutable---we cant chnge the value after declaration==we have to create a new string otherwise we can’t chnge the single letter**

Strings are not mutable! (meaning you can't use indexing to change individual elements of a string)

**Example**

name="sam"

>>> name

'sam'

>>> name[0]='p'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

>>> name="sowmya"

>>> name

'sowmya'

>>> name[5]='o'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

**For this we can use string concatinaton—combining two string**

Ex=1

>>> name="tru"

>>> name

'tru'

>>> name[1:]

'ru'

>>> lst\_letters=name[1:]

>>> lst\_letters

'ru'

>>> 'B'+lst\_letters

'Bru'

>>>

**Ex:2**

>>> name="im the only one here"

>>> name

'im the only one here'

>>> middlename=name[11:15]

>>> middlename

' one'

>>> "some"+middlename

'some one'

**String addition**

>>> x="hello world"

>>> x+"it is beautiful outside"

'hello worldit is beautiful outside'

>>> x+" it is beautiful outside"

'hello world it is beautiful outside'

>>> x=x+" it is beautiful outside"

>>> x

'hello world it is beautiful outside'

>>> x

'hello world it is beautiful outside'

String Mutiplication

>>> x=x\*4

>>> x

'hello world it is beautiful outsidehello world it is beautiful outsidehello world it is beautiful outside it is beautiful outsidehello world it is beautiful outsidehello world it is beautiful outsidehello world it is beautiful outside it is beautiful outsidehello world it is beautiful outsidehello world it is beautiful outsidehello world it is beautiful outside it is beautiful outsidehello world it is beautiful outsidehello world it is beautiful outsidehello world it is beautiful outside it is beautiful outside'

>>>

* **If we give 2+3 it will add them ‘5’ is the output**
* **If we give ‘2’+’3’ it will give ‘23’—this is string concatenation**
* **If we click the string name like this in the jupyter notebook it will provide all the details of string methids**

**Stringname.+tab**

**Stringname.upper()===converts all letters to upcase**

**stringname.lower()====converts all letters to lower case**

>>> x.upper()

'HELLO WORLD'

>>> x.lower()

'hello world'

**stringname.split()==it will split the words with respect to white spaces**

>>> x="hi this is string"

>>> x.split()

['hi', 'this', 'is', 'string']----splits w.r.t to white space

>>> x.split('i'). ]----splits w.r.t to letter specified and removes that letter and keep the spaces as is

['h', ' th', 's ', 's str', 'ng']

>>>

Print formatting strings

* Often you will want to “inject” a variable into your string for printing. For example:
  + **my\_name = “Jose”**
  + **print(“Hello ” + my\_name)**
* There are multiple ways to format strings for printing variables in them.
* This is known as string interpolation.
* Let’s explore two methods for this:
  + **.format()** method
  + **f-strings** (formatted string literals)

Formatting with the .format() method

A Good way to format objects into a strings for print statements is with the string .format() method

Syntax—print(‘This is a string{}’.format(“inserted”))

>>> print('This is a string {}'.format("inserted"))

This is a string inserted

>>>

The words can be inserted in the statement using .format() method

The words can be inserted in index positions

print('Hi {} {} {}' .format("sowmya","javvadi","good morning"))

Hi sowmya javvadi good morning

**Words count should match**

print('Hi {} {} {}' .format("sowmya","javvadi"))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: Replacement index 2 out of range for positional args tuple

>>>

We can reorder or give the particular index positions for the insertion

>>>

>>> print('The {} {} {}'.format('fox','brown','quick'))

The fox brown quick

>>> print('The {2} {1} {0}'.format('fox','brown','quick'))

The quick brown fox

>>>

We can also assign key values to the particular words, with that key word we can insert the words instead of index numbers

>>> print('The {b} {q} {f}'.format(f='fox',b='brown',q='quick'))

The brown quick fox

>>>

**Float formatting follows”{value:width.precision f}”**

If we use division we have all the decimal numbers so for that if we want to give the result with only 3 decimals or req num we can use this

Example

>>> print("the value is {}" .format(result))

the value is 0.1287001287001287------giving the result directly

>>> print("the value is {r}" .format(r=result))

the value is 0.1287001287001287-----provided with keyvalue

>>> print("the value is {r:10.3f}" .format(r=result))

the value is 0.129----giving the precision so it will give in 3 decimal values and 10 represents the spaces before the number it represents the entire width of the value

>>> print("the value is {r:3.3f}" .format(r=result))

the value is 0.129

>>>

**Formatted string literals**

String Formatting

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

player = 'Thomas'

points = 33

'Last night, '+player+' scored '+str(points)+' points.' # concatenation

f'Last night, {player} scored {points} points.' # string formatting

Instead of providing the format method in the print statement we can pass the string directly to this{}

example

>>> name='sowmya'

>>> name

'sowmya'

>>> print("her name is {s}" .format(s='sowmya'))

her name is sowmya

>>> print(f"her name is {name}")---///format literals syntax

her name is sowmya

>>>

example

name="sowmi"

>>> age=23

>>> print(f'{name} is {age} years old')

sowmi is 23 years old

>>>

Anothre formate for string insertion

We have to use %s for value insertion and %’ ‘ for value declaration

print("my name is %s " %'name')

my name is name

>>>

**Notess**

**Format conversion methods.**

It should be noted that two methods %s and %r convert any python object to a string using two separate methods: str() and repr(). We will learn more about these functions later on in the course, but you should note that %r and repr() deliver the *string representation* of the object, including quotation marks and any escape characters.

In [4]:

print('He said his name was %s.' **%**'Fred')

print('He said his name was %r.' **%**'Fred')

He said his name was Fred.

He said his name was 'Fred'.

As another example, \t inserts a tab into a string.

In [5]:

print('I once caught a fish %s.' **%**'this \tbig')

print('I once caught a fish %r.' **%**'this \tbig')

I once caught a fish this big.

I once caught a fish 'this \tbig'.

The %s operator converts whatever it sees into a string, including integers and floats. The %d operator converts numbers to integers first, without rounding. Note the difference below:

In [6]:

print('I wrote %s programs today.' **%3**.75)

print('I wrote %d programs today.' **%3**.75)

I wrote 3.75 programs today.

I wrote 3 programs today.

### Padding and Precision of Floating Point Numbers

Floating point numbers use the format %5.2f. Here, 5 would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits. Next to this, .2f stands for how many numbers to show past the decimal point. Let's see some examples:

In [7]:

print('Floating point numbers: %5.2f' **%**(13.144))

Floating point numbers: 13.14

In [8]:

print('Floating point numbers: %1.0f' **%**(13.144))

Floating point numbers: 13

In [9]:

print('Floating point numbers: %1.5f' **%**(13.144))

Floating point numbers: 13.14400

In [10]:

print('Floating point numbers: %10.2f' **%**(13.144))

Floating point numbers: 13.14

In [11]:

print('Floating point numbers: %25.2f' **%**(13.144))

Floating point numbers: 13.14

For more information on string formatting with placeholders visit <https://docs.python.org/3/library/stdtypes.html#old-string-formatting>

### Multiple Formatting

Nothing prohibits using more than one conversion tool in the same print statement:

In [12]:

print('First: %s, Second: %5.2f, Third: %r' **%**('hi!',3.1415,'bye!'))

First: hi!, Second: 3.14, Third: 'bye!'

## Formatting with the .format() method

A better way to format objects into your strings for print statements is with the string .format() method. The syntax is:

'String here {} then also {}'.format('something1','something2')

For example:

In [13]:

print('This is a string with an {}'**.**format('insert'))

This is a string with an insert

### The .format() method has several advantages over the %s placeholder method:

#### 1. Inserted objects can be called by index position:

In [14]:

print('The {2} {1} {0}'**.**format('fox','brown','quick'))

The quick brown fox

#### 2. Inserted objects can be assigned keywords:

In [15]:

print('First Object: {a}, Second Object: {b}, Third Object: {c}'**.**format(a**=**1,b**=**'Two',c**=**12.3))

First Object: 1, Second Object: Two, Third Object: 12.3

#### 3. Inserted objects can be reused, avoiding duplication:

In [16]:

print('A %s saved is a %s earned.' **%**('penny','penny'))

*# vs.*

print('A {p} saved is a {p} earned.'**.**format(p**=**'penny'))

A penny saved is a penny earned.

A penny saved is a penny earned.

### Alignment, padding and precision with .format()

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

In [17]:

print('{0:8} | {1:9}'**.**format('Fruit', 'Quantity'))

print('{0:8} | {1:9}'**.**format('Apples', 3.))

print('{0:8} | {1:9}'**.**format('Oranges', 10))

Fruit | Quantity

Apples | 3.0

Oranges | 10

By default, .format() aligns text to the left, numbers to the right. You can pass an optional <,^, or > to set a left, center or right alignment:

In [18]:

print('{0:<8} | {1:^8} | {2:>8}'**.**format('Left','Center','Right'))

print('{0:<8} | {1:^8} | {2:>8}'**.**format(11,22,33))

Left | Center | Right

11 | 22 | 33

You can precede the aligment operator with a padding character

In [19]:

print('{0:=<8} | {1:-^8} | {2:.>8}'**.**format('Left','Center','Right'))

print('{0:=<8} | {1:-^8} | {2:.>8}'**.**format(11,22,33))

Left==== | -Center- | ...Right

11====== | ---22--- | ......33

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent:

In [20]:

print('This is my ten-character, two-decimal number:%10.2f' **%13**.579)

print('This is my ten-character, two-decimal number:{0:10.2f}'**.**format(13.579))

This is my ten-character, two-decimal number: 13.58

This is my ten-character, two-decimal number: 13.58

Note that there are 5 spaces following the colon, and 5 characters taken up by 13.58, for a total of ten characters.

For more information on the string .format() method visit <https://docs.python.org/3/library/string.html#formatstrings>

## Formatted String Literals (f-strings)

Introduced in Python 3.6, f-strings offer several benefits over the older .format() string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through .format(var).

In [21]:

name **=** 'Fred'

print(f"He said his name is {name}.")

He said his name is Fred.

Pass !r to get the string representation:

In [22]:

print(f"He said his name is {name!r}")

He said his name is 'Fred'

#### Float formatting follows "result: {value:{width}.{precision}}"

Where with the .format() method you might see {value:10.4f}, with f-strings this can become {value:{10}.{6}}

In [23]:

num **=** 23.45678

print("My 10 character, four decimal number is:{0:10.4f}"**.**format(num))

print(f"My 10 character, four decimal number is:{num:{10}.{6}}")

My 10 character, four decimal number is: 23.4568

My 10 character, four decimal number is: 23.4568

Note that with f-strings, precision refers to the total number of digits, not just those following the decimal. This fits more closely with scientific notation and statistical analysis. Unfortunately, f-strings do not pad to the right of the decimal, even if precision allows it:

In [24]:

num **=** 23.45

print("My 10 character, four decimal number is:{0:10.4f}"**.**format(num))

print(f"My 10 character, four decimal number is:{num:{10}.{6}}")

My 10 character, four decimal number is: 23.4500

My 10 character, four decimal number is: 23.45

If this becomes important, you can always use .format() method syntax inside an f-string:

In [25]:

num **=** 23.45

print("My 10 character, four decimal number is:{0:10.4f}"**.**format(num))

print(f"My 10 character, four decimal number is:{num:10.4f}")

My 10 character, four decimal number is: 23.4500

My 10 character, four decimal number is: 23.4500

For more info on formatted string literals visit <https://docs.python.org/3/reference/lexical_analysis.html#f-strings>

14/11/2022

List, Tuple, Set, and Dictionary are the data structures in python that are used to store and organize the data in an efficient manner.

**Lists**

* Lists are ordered sequences that can hold a variety of object types.
* They use [] brackets and commas to separate objects in the list.
  + **[1,2,3,4,5]**
* Lists support indexing and slicing. Lists can be nested and also have a variety of useful methods that can be called off of them.
* List can be mutable we can chnge the items of the .ist not like the strings
* On lists we can perform indexing,slicing,and we ca add two lists directly

mylist=[1,2,3,4,5]--we can pass directly any type of values to the list

>>> mylist

[1, 2, 3, 4, 5]

>>> mylist=['sowmya',100,23.4]--we can pass directly any type of values to the list

>>> mylist

['sowmya', 100, 23.4]

>>> mylist[2]---we can take the index of the elements in the list

23.4

**List concatination**

anotherlist=['javvadi', 2000,'xyz']

>>> anotherlist

['javvadi', 2000, 'xyz']

>>> newlist=mylist+anotherlist

>>> newlist

['sowmya', 100, 23.4, 'javvadi', 2000, 'xyz']

**We can change the list items like this**

newlist

['sowmya', 100, 23.4, 'javvadi', 2000, 'xyz']

>>> newlist[2]=200

>>> newlist

['sowmya', 100, 200, 'javvadi', 2000, 'xyz']

**append() is used to add items at the end of the list**

newlist.append('hi good morning')

>>> newlist

['sowmya', 100, 200, 'javvadi', 2000, 'xyz', 'hi good morning']

**pop() is used to get the last item from last and if we give the index number it will pop that item out**

**default index position of the pop is ‘-1’**

newlist

['sowmya', 100, 200, 'javvadi', 2000, 'xyz', 'hi good morning']

>>> newlist.pop(2)

200

**It will remove the poped items frm the list**

poppeditem=newlist.pop(4)

>>> poppeditem

'xyz'

>>> newlist

['sowmya', 100, 'javvadi', 2000]

>>>

**sort() function is used to sort the elements in a list …it will not return anything but instead it will arrange the items in the list**

newlist=['a','e','h','y','v','c']

>>> numlist=[3,6,8,23,5,6,7,98]

>>> newlist.sort()—it will sort the items in the list & not return anything we have to check the original list

>>> newlist

['a', 'c', 'e', 'h', 'v', 'y']

>>> numlist.reverse()—reverse the actual list itms

>>> numlist

[98, 7, 6, 5, 23, 8, 6, 3]

>>> numlist.sort()

>>> numlist

[3, 5, 6, 6, 7, 8, 23, 98]

>>>

**Dictionaries in python**

* Dictionaries are unordered mappings for storing objects. Previously we saw how lists store objects in an ordered sequence, dictionaries use  a key-value pairing instead.
* This key-value pair allows users to quickly grab objects without needing to know an index location.
* Dictionaries use curly braces and colons to signify the keys and their associated values.

**{'key1':'value1','key2':'value2'}**

* Value can be of any datatype nd keys always be in string notation
* So when to choose a list and when to choose a dictionary?
* Dictionaries:  Objects retrieved by key name.

Unordered and can not be sorted.

* Lists:  Objects retrieved by location.

Ordered Sequence can be indexed or sliced.

Example

Dictionaries can be written in key value pairs and should be in culry braces

dict={'key1':'value1', 'key2':'value2','key3':'value3'}

>>> dict

{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

>>> dict['key2']----index should be key and that should be in string format

'value2'

prices={'mobile':23546,'laptop':45000,'charger':500}

>>> prices['mobile']

23546

We can give the lists and dictionaries inside the dictionaries

d={'k1':'v1','k2':200,'k3':[2,4,5,6,8],'k4':{'insidekey':'insidevalue'}}

>>> d['k3']---it will return the list inside the dict

[2, 4, 5, 6, 8]

>>> d['k4']----here it will give the k4 value

{'insidekey': 'insidevalue'}

>>> d['k4']['insidekey']---here getting the k4 and inside key value

'insidevalue'

**If you want to change any item in the dicstionary means to upper case or to lower case or to apply any chnge this is the example**

>>> Alp={'k1':['a','b','c'],'k2':['A','B','C','D']}

>>> Alp['k1']

['a', 'b', 'c']

>>> list=Alp['k1']

>>> list

['a', 'b', 'c']

>>> letter=list[1]

>>> letter

'b'

>>> upper=letter.upper()

>>> upper

'B'

>>> Alp['k2']

['A', 'B', 'C', 'D']

>>> letter=Alp['k2'][3]

>>> letter

'D'

>>> lower=letter.lower()

>>> lower

'd'

>>>

**We can add the values for the existing dictionary and can change the value of the key**

>>> Alp={'k1':['a','b','c'],'k2':['A','B','C','D']}

>>> Alp['k3']=300---added the k3 value to the existing dict

>>> Alp

{'k1': ['a', 'b', 'c'], 'k2': ['A', 'B', 'C', 'D'], 'k3': 300}

>>>

>>> Alp['k1']=['s','o','w']---changed the k1 value

>>> Alp

{'k1': ['s', 'o', 'w'], 'k2': ['A', 'B', 'C', 'D'], 'k3': 300}

>>>

**We can get the keys and values of the dictionary separately and items of the disc**

>>> Alp.keys()

dict\_keys(['k1', 'k2', 'k3'])----keys

>>> Alp.values()----values

dict\_values([['s', 'o', 'w'], ['A', 'B', 'C', 'D'], 300])

Alp.items()---items

dict\_items([('k1', ['s', 'o', 'w']), ('k2', ['A', 'B', 'C', 'D']), ('k3', 300)])

**15/11/2022**

**Tuples**

* Tuples are very similar to lists. However they have one key difference - immutability.
* Once an element is inside a tuple, it can not be reassigned.
* Tuples use parenthesis:  (1,2,3)

**Examples**

tuple=(91,"eww",78.89,'yooo')---tuple assiging

>>> tuple

(91, 'eww', 78.89, 'yooo')

>>> firstelement=tuple[0]---extracting the elements frm tuple

>>> firstelement

91

>>>

>>>

>>>

>>> mylist=[1,4,56,767]

>>> mylist

[1, 4, 56, 767]

>>> type(tuple)---type can be know by type function

<class 'tuple'>

>>> type(mylist)

<class 'list'>

>>> type(tuple[1])---type of particular element in tuple

<class 'str'>

>>> tuple=(91,"eww",78.89,'yooo')--

>>> tuple[:1]---tuple slicing

(91,)

>>> tuple[1:]

('eww', 78.89, 'yooo')

>>> tuple[::-1]

('yooo', 78.89, 'eww', 91)

>>>

>>>

>>>

>>> tuple.count('yooo')---count() will give the how many same elements in tuple

1

>>> newtuple=(1,1,2,5,6,'a','a','b')

>>> newtuple

(1, 1, 2, 5, 6, 'a', 'a', 'b')

>>> fulltuple=tuple+newtuple--- tuple concatination

>>> fulltuple

(91, 'eww', 78.89, 'yooo', 1, 1, 2, 5, 6, 'a', 'a', 'b')

>>>

>>>

>>> newtuple.count(1)---count of 1 in the tuple

2

>>> newtuple.index('a')---index of the ‘a’ in the tuple and it will give the first tym index

5

>>>

mylist[0]='new'----we can reassign the list and it can be mutable

>>> mylist

['new', 4, 56, 767]

>>>

>>> tuple[0]='new'--we can’t reassign the list and it is Immutable

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

>>>

**Sets**

* Sets are unordered collections of unique elements.
* Meaning there can only be one representative of the same object.

Examples

set={1,2,3,44,35,32}

>>> set

{32, 1, 2, 3, 35, 44}

>>> set2={'@',76,35}

>>> set2

{35, '@', 76}

>>>

>>> set2.add(23)

>>> set2

{35, '@', 23, 76}

>>> set={'p','a','r','a','l','l','e','l'}

>>> set

{'p', 'e', 'a', 'r', 'l'}

**Booleans**

**Booleans** are operators that allow you to convey **True** or **False** statements.

These are very important later on when we deal with control flow and logic!

**True, False** and when we don’t want to assign a variable value then we can gve that value as **None**

>>> 1==2

False

>>> 3<5

True

>>>

>>> 1>2

False

**Files**