Tuple

A tuple is a sequence of elements of any type and is immutable. In python, they are initialized inside parenthesis "()" instead of square brackets "[]". To create a tuple, we simply have to do the following:

Code	Output
`example_tuple = ("banana", "mango", "apple")` `print(type(example_tuple))`	`<class 'tuple'>`

Tuples can hold values of any type. They can be homogeneous as well as heterogeneous. But we need to remember that once we declare those values, we cannot change them. For example:

Code	Output

`mixed_type = ('C','S','E', 1, 1, 0)`
 	
`for char in mixed_type:`
`  print(char,":",type(i))`
	`C : <class 'str'>`
	`S : <class 'str'>`
	`E : <class 'str'>`
	`1 : <class 'int'>`
	`1 : <class 'int'>`
	`0 : <class 'int'>`

`#Trying to change 'E' to 'O'` `mixed_type[2] = 'O'`	`TypeError: 'tuple' object does not support item assignment`

Here, we are getting the error message because we are not allowed to change values inside a tuple.

Immutability of Tuple:

The word "immutability" in python means an object with a fixed value/id. Here "id" is the identity of a **location** of an object in **memory**. For example:

Code	Output
#1) Declare a Tuple named 'nth_tuple' `nth_tuple = (21,21,34,47)`	

#2) Items with the same value have the same id.
 	
`if id(nth_tuple[0]) == id(nth_tuple[1]):`
`  print("True")`
`print(id(nth_tuple[0]))`
	`True`
	
	`9756832`
	
	`9756832`

`print(id(nth_tuple[1]))`	

#3) Items with different values have different ids.
 	
`if id(nth_tuple[0]) == id(nth_tuple[2]):`
`  print ("True")`
`else:`
`  print ("False")`
 	
`print(id(nth_tuple[0]))`
`print(id(nth_tuple[2]))`
	`False`
	
	`9756832`
	
	`9757248`

#4) append function is not applicable in a tuple `nth_tuple.append(5)`	`AttributeError: 'tuple' object has no attribute 'append'`

Tuples are efficient:

Tuples provide no access to data values and are considered as faster than the lists. For example:

Code	Output

`#Execution time for tuple:`
`import timeit`
`print(timeit.timeit('x=(1,2,3,4,5,6,7,8,9)', number=100000))`
	`0.001523201004602015`

`#Execution time for List:` `print(timeit.timeit('x=[1,2,3,4,5,6,7,8,9]', number=100000)))`	`0.009876493015326656`

In the above example, we have used the timeit() method imported from python library timeit to calculate the difference between the execution time of tuple and list. It is seen that tuple takes less time to execute than a list.

To access the Items of Tuple:

1. Print the third item in the tuple:

Code	Output
`example_tuple = ("Banana", "Mango", "Apple")` `print(example_tuple[2])`	`Apple`

2. In tuple, negative indexing means beginning from the end. Example: -1 refers to the last item, -2 refers to the second-last item, etc.:

Code	Output

`example_tuple = ("Banana", "Mango", "Apple")`
 	
`print(example_tuple[-1])`
`print(example_tuple[-2])`
	`Apple`
	`Mango`

3. While specifying a range in a tuple, the return value will give a new tuple with the specified items:

Code	Output

`example_tuple = ("Banana", "Mango", "Apple", "Orange","Grape","Jackfruit")`
 	
`print(example_tuple[2:5])`
	`('Apple','Orange', 'Grape')`

4. While specifying a range of negative indexes in a tuple, we can start the search from the end of a tuple. Given example returns the items from index -5 (included) to index -1 (excluded):

Code	Output

`example_tuple = ("Banana", "Mango", "Apple", "Orange","Grape","Jackfruit")`
 	
`print(example_tuple[-5:-1])`
	`('Mango', 'Apple', 'Orange', 'Grape')`

Tuple Unpacking:

To store elements of a tuple in separate variables is called unpacking. This operation allows us to take multiple return values from a function and store them in separate variables.

Basic example:

Code	Output

`example_tup = (3, 2, 1)`
`a, b, c = example_tup`
 	
`print("a:", a, "b:", b, "c:", c)`
	`a: 3 b: 2 c: 1`

`#swapping values`
`b, c, a = example_tup`
`print("a:", a, "b:", b, "c:", c)`
	`a: 1 b: 3 c: 2`

Advanced example: [Try this after function chapter]

Code	Output

`def convert_days(days):`
`  years = days // 365`
`  months = (days - years * 365) // 30`
`  remaining_days = days - years * 365 - months *30`
`  return years, months, remaining_days`
`result = convert_days(4320)`
`print(result)`
	`(11, 10, 5)`

`years, months,days = result` `print(years, months, days)`	`11 10 5`
`years,months,days = convert_days(4000)` `print(years, months, days)`	`10 11 20`

In the above example, a function is written which takes the number of days as input and returns the number of years, month, and remaining days. That is, the function returns a tuple of three elements. After that, we've split the tuple into three separate variables (years, months, days) each of which has its own value.

Basic Operations of Tuple:

Example:	Output

#1) Looping through tuple using for loop:
 	
`example_tuple = ("Banana", "Mango", "Apple", "Orange","Grape","Jackfruit")`
 	
`for fruit in example_tuple:`
`  print(fruit)`
	`Banana`
	`Mango`
	`Apple`
	`Orange`
	`Grape`

#2) To check if an item exists in the tuple:
 	
`example_tuple = ("Banana", "Mango", "Apple", "Orange","Grape","Jackfruit")`
 	
`if "Grape" in example_tuple:`
`  print("Yes,Grape is in the tuple")`
	`Yes,Grape is in the tuple`

#4) To create a tuple with single item, we need to put comma:
 	
`example_tuple = ("Grape",)`
`print(type(example_tuple))`
 	
`#NOT a tuple`
 	
`example_tuple = ("Grape")`
`print(type(example_tuple))`
	`<class 'tuple'>`
	`<class 'str'>`

#5) Because of its immutability, we cannot remove items in a tuple but, we can delete the tuple completely:
 	
`Delete_the_tuple = ("Anna", "Lukas", "Julia")`
`del Delete_the_tuple`
`print(Delete_the_tuple)`
	`NameError: name 'Delete_the_tuple' is not defined`

#6) For joining two tuples we can use '+' operator:
 	
`tuple1 = ("Anna", "Lukas","Julia")`
`tuple2 = (1, 2, 3)`
 	
`tuple3 = tuple1 + tuple2`
`print(tuple3)`
	`('Anna', 'Lukas', 'Julia', 1, 2, 3)`

Tuple Methods:

Method name	Description	Example

len()	Finds the number of items in the tuple	`example_tuple = ("Banana", "Mango", "Apple", "Orange")`
		`print(len(example_tuple))`
		
		`Output:`
		`4`

tuple()	Returns a tuple using double round-brackets.	`tuple_constructor = tuple(("Anna", "Lukas", "Julia"))`
		`print(tuple_constructor)`
		
		`Output:`

		('Anna', 'Lukas', 'Julia')
count()	Returns the number of times the value appears in the tuple.	tuple_count = (53,32,78,12,2,3,6,7,53,53) number= tuple_count.count(53) print(number) Output: 3
index()	Searches for the first occurrence of the value 12, and returns its position.	tuple_index = (53,32,78,12,2,3,6,7,53,53) x = tuple_index.index(2) print(x) Output: 4

enumerate()	It takes a list as a parameter and returns a tuple for each element in the list.	grocery = ['bread', 'milk', 'butter']
		newGrocery = enumerate(grocery)
		
		# converting to tuple
		print(tuple(newGrocery))
		
		Output:
		((0, 'bread'), (1, 'milk'), (2, 'butter'))
		((10, 'bread'), (11, 'milk'), (12, 'butter'))

To change tuple values:

Although tuples are unchangeable or immutable, we can still change the items of a tuple by converting the tuple into a list, change the list, and then, convert the list back to the tuple.

Example:	**Output**
given_tuple = ("Banana", "Mango", "Apple") to_list = list(given_tuple) to_list[1] = "Kiwi" new_tuple = tuple(to_list) print(new_tuple)	('Banana', 'Kiwi', 'Apple')

Dictionary

Dictionary

A dictionary is an unordered collection that consists of *key-value* pairs. It is python's inbuilt mapping type. It is similar to lists, but we know after studying so far that lists are sequential collections. Lists index their entries based on the position in the list which is sequenced from left to right, whereas dictionaries index their items **using *keys* and are unordered**.

The necessity of Dictionaries

Suppose, you want to specify a different index for a specific value as sometimes it makes sense to have keys for different elements. You cannot do that using lists as we know that list is a sequential collection. Think of a phonebook, if you want to collect a number, you search by name and get the number. Similarly, if you are asked to mention the page number of chapter 8 in a book. You can turn over the pages one by one and eventually find the heading Chapter 8 and then tell the page number. However, it is faster to go to the index, look for chapter 8, and tell the page number. In the above examples, the 'name' in the phonebook and the 'chapter 8' work as keys. So, the dictionary is needed for this kind of **direct searching**.

Key-value pair

As mentioned above, the dictionary is python's mapping type, here the mapping is from a *key* which can of any type that is immutable (unchangeable), to a *value* that can be any Python Data Object.

Creating a Dictionary

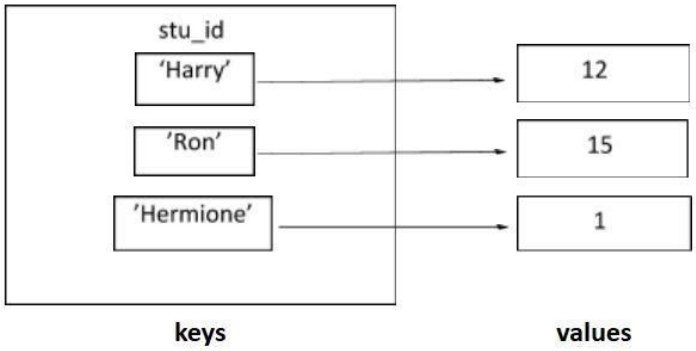
Example 1:

Code	Output

`stu_id = {}`
`stu_id ['Harry'] = 12`
`stu_id ['Ron'] = 15`
`stu_id ['Hermione'] = 1`
`print(stu_id)`
	`{'Harry': 12,`
	`'Ron': 15,`
	`'Hermione': 1}`

`(Can also be set in a single line)`
 	
`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`print(stu_id)`
	`{'Harry': 12,`
	`'Ron': 15,`
	`'Hermione': 1}`

If we visualize the code, it looks like this.



Example 2:

Code	Output

`eng_to_fr = {}`
`eng_to_fr ['one'] = 'une'`
`eng_to_fr ['two'] = 'deux'`
`eng_to_fr ['three'] = 'trois'`
`print(eng_to_fr)`
	`{'one': 'une',`
	`'two': 'deux',`
	`'three': 'trois'}`

`eng_to_fr = { 'one': 'une', 'two': 'deux',`
`'three': 'trois'}`
`print(eng_to_fr)`
	`{'one': 'une',`
	`'two': 'deux',`
	`'three': 'trois'}`

In the above examples,

Key	Value	Key	Value
'Harry'	12	'one'	'une'
'Ron'	15	'two'	'deux'
'Hermione'	1	'three'	'trois'

So, dictionary literals have curly braces and have a list of **key: value** pairs which are separated by comma (,). Dictionary keys are **case sensitive**, same name but different cases of **the key** will be treated distinctly.

Empty Dictionary

Empty Dictionaries can be made using empty curly braces.

Code	Output
`abc = { }` `print(abc)`	`{ }`

Retrieving values

It does not depend on which order the pairs are written. To retrieve a value from the dictionary, it is the key that is needed. So you don't have to think about which order they are.

Code	Output

`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`value = stu_id['Hermione']`
`print(value)`
`print(stu_id['Harry'])`
	1
	12

You can get the same output by calling **get()** function**.**

Code	Output

`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`value = stu_id.get('Hermione')`
`print(value)`
	1

The **get()** function returns the value of the key if the key exists in the dictionary. Otherwise, it shows **None** if the key is not present.

Code	Output

`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`print(stu_id.get('Hermione'))`
`print(stu_id.get('Draco'))`
	1
	None

Dictionaries are Mutable

Dictionaries are mutable because their values can be modified. Dictionaries are something that needs constant changing - growing, shrinking, updating and deleting entries, etc. The examples below further illustrate how dictionary values are modified in place. For example:

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`print(box)`
`box['pen']= 2`
`print(box)`
	`{'pencil': 3, 'pen': 4, 'eraser': 2}`
	
	`{'pencil': 3, 'pen': 2, 'eraser': 2}`

`box = {'pencil':3,'pen':4,'eraser':2}`
`print(box)`
`box['pencil']= box['pencil']+3`
`print(box)`
	`{'pencil': 3, 'pen': 4, 'eraser': 2}`
	`{'pencil': 6, 'pen': 4, 'eraser': 2}`

Finding the length of Dictionary

We can find the total number of key-value pairs in the dictionary using the **len()** function.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`num_items = len(closet)`
`print(num_items)`
	`3`

Looping through a Dictionary

We can loop through a dictionary by using for loop. For example, if you want to display all the key names from the dictionary:

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for key in box:`
`  print(key)`
	`pencil`
	`pen`
	`eraser`

And, for printing all the values in the dictionary:

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for key in box:`
`  print(box[key])`
	`3`
	`4`
	`2`

The values of the dictionary can also be returned by using **the values()** function.

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for val in box.values():`
`  print(val)`
	`3`
	`4`
	`2`

We can also loop through both keys and values, by using **items()** function.

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for key,val in box.items():`
`  print(key,val)`
	`eraser 2`
	`pencil 3`
	`pen 4`

Using the **key()** function, we get the keys of the dictionary as a list.

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`key_values = box.keys()`
`print(key_values)`
	`dict_keys(['pencil',`
	`'pen', 'eraser'])`

If you update the dictionary, then the updated list of keys will be displayed.

Code	Output
box = {'pencil':3,'pen':4,'eraser':2} key_values = box.keys() box['cutter'] = 1 print(key_values)	dict_keys(['pencil', 'cutter', 'pen', 'eraser'])

Determining if a key exists in Dictionary

For checking whether a key is in the dictionary or not, we use the membership operator, **in**.

Code	Output
closet = {'shirt':3,'pant':4,'scarf':2} if 'scarf' in closet: print('Yes')	Yes
closet = {'shirt':3,'pant':4,'scarf':2} print('scarf' in closet)	True
closet = {'shirt':3,'pant':4,'scarf':2} print('shoes' in closet)	False

Adding elements

If we want to add a new element to the dictionary, we can do it by using a new key and providing value to it.

Code	Output
closet = {'shirt':3,'pant':4,'scarf':2} closet['shoes'] = 1 print(closet)	{'shirt': 3, 'pant': 4, 'scarf': 2, 'shoes': 1}

You can also insert an element to your dictionary using update() method.

Code	Output
closet={'shirt':3, 'pant':4, 'scarf':2} closet.update({'shoes':1}) print(closet)	{'shirt': 3, 'pant': 4, 'scarf': 2, 'shoes': 1}

Removing elements

To remove an element from the dictionary, we use **del** keyword with the specified key name.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`del closet['scarf']`
`print(closet)`
	`{'shirt': 3, 'pant': 4}`

The **del** keyword can also be used to remove the whole dictionary.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`del closet`
`print(closet)`
	`NameError: name 'closet' is not defined`
	
	Here, trying to print the dictionary closet is showing error as the dictionary has been deleted.

Sorting Dictionary

You can also sort the elements of the dictionary by values or keys by using **sorted().**

By keys,

Code	Output

`box = {'cutter':3,'pen':4,'eraser':2}`
`ksort = sorted(box.keys())`
`print(ksort)`
	`['cutter', 'eraser', 'pen']`

By values,

Code	Output

`box = {'cutter':3,'pen':4,'eraser':2}`
`ksort = sorted(box.values())`
`print(ksort)`
	`[2, 3, 4]`

Copying Dictionary

We can also make a copy of the dictionary by using the copy**()** function. For example: suppose, we have a dictionary of a library. The library includes both fiction and non-fiction genres. But we want to see fiction and non-fiction parts separately.

Code	Output

`library = {'sci-fi': 25,'mystery':17,'mythology':12,'biography':22, 'history':20}`
`fictions = {'sci-fi': 25,'mystery':17,'mythology':12}`
`nonfictions = {'biography':22,'history':20}`
 	
`fiction_dict = fictions.copy()`
`non_fiction_dict = nonfictions.copy()`
`print(fiction_dict)`
`print(non_fiction_dict)`
	`{'sci-fi': 25, 'mystery': 17, 'mythology': 12}`
	`{'biography': 22, 'history': 20}`

Now look at the above example, we have declared the items of fictions and non-fictions genres in separate dictionaries named `fictions` and `nonfictions`. Then the items of both the fictions and non-fictions dictionaries are copied into the new dictionaries named `fiction_dict` and `non_fiction_dict` using the **copy()** function.

So now if you want to check both genres separately, you need not look at the main dictionary library. You just print `fiction_dict` and `non_fiction_dict` and see how many books are available in each category.

When to use which data type?

Before knowing when to use which data types, we need to know the difference between them. The table below illustrates their difference.

Data type	Type of elements	Mutability	Indexed By	Supports "in" operator	Supports " +" and "*" operator
String	Characters	Immutable	integers	Yes	Yes
List	Any type	Mutable	Integers	Yes	Yes
Tuple	Any type	Immutable	Integers	Yes	Yes
Dictionary	Values any type, but keys immutable type	Mutable	Any immutable data type	Yes	No
Range Function	Integers	Immutable	Integers	Yes	Yes

Advantage-Disadvantage of List:

Due to lists mutability, it is more popular among the programmer's community than Tuple. For example, we can use it to store any values of any data types while still building it. But due to the same characteristic (mutability), it cannot be used as a key for dictionaries and it is prone to aliasing problems. So, when we do not want the original list to be modified, copies of the original list are made before making any kind of modifications. For example, the slicing method makes a new copy of the original list automatically every time before returning the modified list as a return value.

Advantage-Disadvantage of Tuple:

Since tuples are immutable, they do not have any side effects as well as any aliasing problems. For example, for lessening the chances of any possible aliasing problems, tuples can be used to pass values as an argument to functions. Additionally, tuples can hold any data types as its elements. But for a tuple to work as a dictionary key, its elements have to be immutable.

Advantage-Disadvantage of String:

Unlike tuples and lists, strings can only have characters as its elements and these characteristics make strings less flexible. But, in Python3, strings have many built-in methods compared to other data types

which make its use quite effortless. Additionally, due to the string's immutability, it has no side effects and it can be used as dictionary keys.

Advantage-Disadvantage of Dictionaries:

Dictionary keys must be unique and immutable data types. However, its values can be any data type including the user-defined custom object types (will be discussed later in this course) and there are no mandatory restrictions for its values to be unique. Since dictionaries are mutable, they can have aliasing problems. For solving this, duplicates of the original dictionary are made, before making any kind of modifications.

Advantage-Disadvantage of Range:

Range function returns an immutable sequence of integers (actually returns a range object) which can be converted into a list. It can also be used in "for loop iteration" (iteration a fixed number of times). For iterating, before the iteration, the "immutable sequence" is automatically converted into a "list".

Code	Output

#1)converting immutable sequence of integers to list
 	
`odd_numbers = list(range(1,12,2))`
`print(odd_numbers)`
	`[1, 3, 5, 7, 9, 11]`

#2)used in for loop iteration (manually converting it into list)
 	
`for num in list(range(1,12,2)):`
`  print(num)`
	1
	3
	5
	7
	9
	11

#3)used in for loop iteration (automatically converting it into a list)
 	
`for num in range(1,12,2):`
`  print(num)`
	1
	3
	5
	7
	9
	11

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/