**PYTHON HELP FILE**

## **Tuples:-**

Tuple is one of 4 built-in data types in Python used to store **collections of data**, the other 3 are [List](https://www.w3schools.com/python/python_lists.asp), [Set](https://www.w3schools.com/python/python_sets.asp), and [Dictionary](https://www.w3schools.com/python/python_dictionaries.asp), all with different qualities and usage.

Tuples are used to store **multiple items in a single variable**.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

thistuple = ("apple", "banana", "cherry")

print(thistuple)

Output: ('apple', 'banana', 'cherry', 'apple', 'cherry')

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Tuple Length -

print(len(thistuple))

Output: 3

## *Create Tuple With One Item*

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Ex:

thistuple = ("apple",)

print(type(thistuple))

Output: <class 'tuple'>

#NOT a tuple

thistuple = ("apple")

print(type(thistuple))

Output: <class 'str'>

A tuple with strings, integers and boolean values:

tuple1 = ("abc", 34, True, 40, "male")

It is also possible to use the tuple() constructor to make a tuple.

thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets

print(thistuple)

O/p: ('apple', 'banana', 'cherry')

Accessing the tuple items: by referring to the index number (starts from 0), inside square brackets

thistuple = ("apple", "banana", "cherry")

print(thistuple[1])

o/p: banana

Negative Indexing: Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

thistuple = ("apple", "banana", "cherry")

print(thistuple[-1])

o/p: cherry

Range of Index:

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(thistuple[2:5])

This prints third, fourth, and fifth items.

The search will start at index 2 (included) and ends at Index 5 (not included)

print(thistuple[:4]) # prints from index 0 to 3.

print(thistuple[2:]) # prints from index 2 till the end of tuple.

Check If item exists :

thistuple = ("apple", "banana", "cherry")

if "apple" in thistuple:

print("Yes, 'apple' is in the fruits tuple")

o/p: Yes, 'apple' is in the fruits tuple

Change tuple values:

Tuples are unchangeable, or immutable as it also is called. But there is a workaround:

Convert the tuple into a list to be able to change it:

Ex:

thistuple = ("apple", "banana", "cherry")

x = list(thistuple)

x[2]= "mango"

thistuple =tuple(x)

print(thistuple)

o/p: ('apple', 'banana', 'mango')

Same workaround for add or delete items:

1. thistuple = ("apple", "banana", "cherry")

y = list(thistuple)

y.append("orange")

thistuple = tuple(y)

o/p: ('apple', 'banana', 'cherry', 'orange')

1. thistuple = ("apple", "banana", "cherry")
2. y = list(thistuple)
3. y.remove("apple")
4. thistuple = tuple(y)

o/p: ('banana', 'cherry')

Add tuple to a tuple:

Ex: Create a new tuple with the value "orange", and add that tuple

thistuple = ("apple", "banana", "cherry")

y = ("orange",)

thistuple += y

print(thistuple)

Delete the tuple completely:

thistuple = ("apple", "banana", "cherry")

del thistuple

print(thistuple) #this will raise an error because the tuple no longer exists

Packing a tuple:

fruits = ("apple", "banana", "cherry")

Unpacking a tuple: means to extract the values back into variables

fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green) # green has value of apple

print(yellow)# yellow has value of banana

print(red) # red has value of cherry

o/p:

apple

banana

cherry

Using \* (asterisk) : If the number of variables is less than the number of values in the tuple, you can add an \* to the variable name and the values will be assigned to the variable as a list.

Ex:

fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, \*red) = fruits

print(green)

print(yellow)

print(red)

o/p:

apple

banana

['cherry', 'strawberry', 'raspberry']

Loop the tuple through index numbers:

Ex: thistuple = ("apple", "banana", "cherry")

for i in range(len(thistuple)):

print(thistuple[i])

Ex: i = 0

while i < len(thistuple):

print(thistuple[i])

i = i + 1

Join the tuples: Use +

tuple1 = ("a", "b" , "c")

tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2

print(tuple3)

o/p: ('a', 'b', 'c', 1, 2, 3)

Multiply the tuple: Use \*

fruits = ("apple", "banana", "cherry")

mytuple = fruits \* 2

print(mytuple)

o/p: ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')

Tuple Methods:

count() - Returns the number of times a specified value occurs in a tuple

Ex: thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)

x = thistuple.count(5)

print(x) # gives 2 as 5 occurs twice in the tuple

index() : Search for the first occurrence of the value, and return its position and raise an exception if value not found in the tuple.

Ex:thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)

x = thistuple.index(8)

print(x) # prints 3

**Python Sets:**

Sets are used to store multiple items (not duplicate, duplicates are ignored) in a single variable.

A set is a collection which is unordered (cannot be referred to by index or key), unchangeable\*(cannot change the items after the set has been created), and unindexed.

Sets are written with curly brackets.

Ex: thisset = {"apple", "banana", "cherry"}

print(thisset)

Sets can contain different data types:

set1 = {"abc", 34, True, 40, "male"}

Length of the set: len(thisset)

type() is <class ‘set’>

set() constructor:

thisset = set(("apple", "banana", "cherry")) # note the double round-brackets

Access the set items:

Loop through the set: for x in thisset: print x

print("banana" in thisset) #Check if "banana" is present in the set, returns True or False

Change items: Once a set is created, you cannot change its items, but you can add new items.

Add item: Add an item to a set, using the add() method

Ex:

thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset) #prints {'orange', 'banana', 'apple', 'cherry'}

Add sets: To add items from another set into the current set, use the update() method

Ex:

thisset = {"apple", "banana", "cherry"}

tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)#prints {'apple', 'mango', 'cherry', 'pineapple', 'banana', 'papaya'}

**Add** elements of a list to set:

thisset = {"apple", "banana", "cherry"}

mylist = ["kiwi", "orange"]

thisset.update(mylist)

print(thisset) # {'banana', 'cherry', 'apple', 'orange', 'kiwi'}

**Remove** items of set:

thisset.remove("banana")# If the item to remove does not exist, remove() will raise an error

thisset.discard("banana") #If the item to remove does not exist, discard() will *not raise an error*

pop() method: this method removes last item, but sets are unordered pop() can remove any item.

Ex:

thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

clear() method: this will empty the set

thisset.clear()

print(thisset) #prints set()

**Del** : this keyword will delete the set completely

del thisset

print(thisset) # this throws an error as no set exists

**Loop** through the set:

for x in thisset: print(x)

**Join** the sets:

union() method that returns a new set containing all items from both sets.

set1 = {"a", "b" , "c"}

set2 = {1, 2, 3}

set3 = set1.union(set2)

print(set3) #prints {1, 'a', 'c', 2, 3, 'b'}

update() method that inserts all the items from one set into another.

set1 = {"a", "b" , "c"}

set2 = {1, 2, 3}

set1.update(set2)

print(set1) #prints 'b', 1, 'c', 2, 3, 'a'}

Both union() and update() will exclude any duplicate items.

**Keep ONLY the duplicates** in sets:

1. intersection\_update() method will keep only the items that are present in both sets

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

x.intersection\_update(y)

print(x) #{'apple'}

1. intersection() method will return a *new* set, that only contains the items that are present in both sets

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

z = x.intersection(y)

print(z) # {'apple'}

**Keep ALL But NOT the duplicates**:

1. symmetric\_difference\_update() method will keep only the elements that are NOT present in both sets

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

x.symmetric\_difference\_update(y)

print(x) #{'google', 'banana', 'microsoft', 'cherry'}

1. symmetric\_difference() method will return a new set, that contains only the elements that are NOT present in both sets

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

z = x.symmetric\_difference(y)

print(z) #{'google', 'banana', 'microsoft', 'cherry'}

**PYTHON Dictionaries**

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in **key:value** pairs, and can be referred to by using the key name.

Duplicate values will overwrite existing values

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964, #this will be overwritten

"year": 2020

}

print(thisdict) #o/p: {'brand': 'Ford', 'model': 'Mustang', 'year': 2020}

Length of the dictionary: len(thisdict)

Type : type(thisdict) # <class dict>

Accessing Dict items:

1. By referring to its key name, inside square brackets.

x = thisdict["model"] #returns Mustang into x

1. get(key):

x = thisdict.get("model")

1. Get the keys in the dict:

keys() : returns list of all the keys in the dict

z = thisdict.keys() #ddict\_keys(['brand', 'model', 'year'])

1. values() : Get the values:

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

X = thisdict.values() #dict\_values(['Ford', 'Mustang', 1964])

1. items(): Get the items in dict

The items() method will return each item in a dictionary, as tuples in a list.

print(thisdict.items()) #dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

Check if key exists in the dict:

if "model" in thisdict:

print(“model key exists in thisdict)

Add a new item to the original dictionary:

car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

X = car.keys() # dict\_keys(['brand', 'model', 'year'])

car[“color”]=”red”

X = car.keys() # dict\_keys(['brand', 'model', 'year', 'color'])

Change Values:

You can change the value of a specific item by referring to its key name

thisdict["year"] = 2018

# dict\_values(['Ford', 'Mustang', 2018, ‘red’])

Update Dict:

update() method will update the dictionary with the items from the given argument.

If the item doesn’t exit, it will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

thisdict.update({"color": "white"})

# dict\_values(['Ford', 'Mustang', 2018, ‘white’])

thisdict.update({“location”:”Delhi”})

# dict\_values(['Ford', 'Mustang', 2018, ‘white’, ‘Delhi’])

**Removing items**:

1. pop(key) method: removes the item with the specified key name

thisdict.pop("model") #this removes model

# dict\_values(['Ford', 2018, ‘white’, ‘Delhi’])

1. popitem() method: removes the last inserted item

thisdict.popitem() #this removes location item

# dict\_values(['Ford', 2018, ‘white’])

1. del keyword:

keyword removes the item with the specified key name

del thisdict["year"]

# dict\_values(['Ford', ‘white’])

del keyword can also delete the dictionary completely

del thisdict #this delete thisdict

1. Clear the empties the dictionaries

thisdict.clear() #this dict is now empty so, {}

**Loop through a dictionary: for, while loops**

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

1. for x in thisdict:

print(x)

# this prints all the keys brand

model

year

1. for x in thisdict:

print(thisdict[x]) # this prints the dict values

Thisdict[x] is same as thisdict.values()

Loop through keys and values:

1. For x , y in thisdict.items(): print(x,y)

Copy the dictionary:

Use copy() method of dict or use dict(thisdict).

mydict = thisdict.copy()

mydict = dict(thisdict)

**Nested Dictionaries -** dictionary containing dictionaries

child1 = {

"name" : "Emil",

"year" : 2004

}

child2 = {

"name" : "Tobias",

"year" : 2007

}

myfamily = {

"child1" : child1,

"child2" : child2

}

print(myfamily)

# {'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}}

**Or**

myfamily = {

"child1" : {

"name" : "Emil",

"year" : 2004

},

"child2" : {

"name" : "Tobias",

"year" : 2007

}}

**Dict Methods**:

clear(), copy(), get(key), items(), keys(), values(), pop(key), popitem(), update({key:value}),

fromkeys(key, value)- method returns a dictionary with the specified keys and the specified value

x = ('key1', 'key2', 'key3')

y = 0

thisdict = dict.fromkeys(x, y)

print(thisdict)

# ['key1': 0, 'key2': 0, 'key3': 0]

If value is not specified, default value None

is taken

x = ('key1', 'key2', 'key3')

thisdict = dict.fromkeys(x)

print(thisdict)

# {'key1': None, 'key2': None, 'key3': None}

**Python if, elif, else**

a= 2

b= 300

1. **If** a> b**:**

print(“a > b”)

**else:**

print(“a is not > b”)

1. if b > a:

print("b is greater than a")

elif a == b:

print("a and b are equal")

else:

print("a is greater than b")

1. Short hand if:

if a > b: print("a is greater than b")

1. Short hand if .. else: (Ternary Operators, or Conditional Expressions.)

print("A") if a > b else print("B") #B

print("A") if a > b else print("=") if a == b else print("B")

1. Nested if

x = 41

if x > 10:

print("Above ten,")

if x > 20:

print("and also above 20!")

else:

print("but not above 20.")

1. Pass statement

if b > a:

Pass

**Looping (while, for)**

1. While loop

i = 1

while i < 6:

print(i)

i += 1

1. Break statement: (break the loop)

i = 1

while i < 6:

print(i)

if i == 3:

break

i += 1

o/p:

1

2

3

1. Continue statement: (stop the current iteration, and continue with the next)

i = 0

while i < 6:

i += 1

if (i == 3):

continue

print(i)

o/p: 1

2

4

5

6

1. else statement

i = 1

while i < 6:

print(i)

i += 1

else: ## once i=>6 else statement executes

print("i is no longer less than 6")

Same for **for loop.**

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

1. fruits = ["apple", "banana", "cherry"]

for x in fruits:

print(x)

1. for x in "banana":

print(x) #each letter in string

1. For and break , continue, else, pass
2. For and range()

for x in range(6):

print(x) #prints 0 1 2 3 4 5

1. for x in range(2, 6):

print(x) #prints 2 3 4 5

1. for x in range(2, 30, 3):

print(x) #increment sequence3 starting from 2 till 29

# prints 2 5 8 11 14 17 20 23 26 29

1. Nested loop

adj = ["red", "big", "tasty"]

fruits = ["apple", "banana", "cherry"]

for x in adj:

for y in fruits:

print(x, y)

**Python Function**

1. Creating a function using def keyword.

def my\_function():

print("Hello from a function")

1. Calling the function:

myfunction()

1. Passing arguments :

def my\_function(fname):

print(fname + " Refsnes")

my\_function(“emil”)

1. No.of arguments :

def my\_function(fname, lname):

print(fname + " " + lname)

my\_function("Emil", "Refsnes")

1. Arbitrary arguments: **\*args** - used when we don't know how many arguments will be passed to function

def my\_function(\*kids):

print("The youngest child is " + kids[2])

my\_function("Emil", "Tobias", "Linus")

# function will receive a *tuple* of arguments

1. Keyword arguments: You can also send arguments with the *key* = *value* syntax. This way the order of the arguments does not matter.

def my\_function(child3, child2, child1):

print("The youngest child is " + child3) # prints Emil

my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

1. Arbitrary Keyword Arguments: \*\*kwargs   
   (use \*\* if we dont know how many arg key args passes)

def my\_function(\*\*kid):

print("His last name is " + kid["lname"])

my\_function(fname = "Tobias", lname = "long")

# prints His last name is long

1. Default parameter value:

If we call the function without argument, it uses the default value:

def my\_function(country = "Norway"):

print("I am from " + country)

my\_function("Sweden")

my\_function("India")

my\_function() #here “Norway” is used.

1. Passing lists as arguments
2. **Return** statements

def my\_function(x):

return 5 \* x

print(my\_function(3))

1. **Pass** statement

def my\_function():

Pass

1. **Recursion**: Python also accepts function recursion, which means a defined function can call itself.

**Python Lambda**

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax: lambda *arguments* : *expression*

Ex:

1. x = lambda a : a + 10

print(x(5)) # prints 15

1. x = lambda a, b : a \* b

print(x(5, 6)) #prints 30

1. x = lambda a, b, c : a + b + c

print(x(5, 6, 2)) # prints 13

Use lambda functions when an anonymous function is required for a short period of time.

1. Why use lambda function?

def myfunc(n):

return lambda a : a \* n

mydoubler = myfunc(2) # this doubles the number

mytripler = myfunc(3) # this doubles the number

print(mydoubler(11)) # prints 22

print(mytripler(12)) # prints 36

**Python Arrays**

Python does not have built-in support for Arrays, but [Python Lists](https://www.w3schools.com/python/python_lists.asp) can be used instead.

**Python Classes**

1. Create a class using class keyword

class MyClass:

x = 5

1. Create object

p1= MyClass();

print(p1.x)

1. The \_\_init\_\_(arguments) function (it is a built in function)

All classes have a function called \_\_init\_\_(), which is always executed when the class is being initiated.

Use the \_\_init\_\_() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def myfunc(self):

print("Hello my name is " + self.name)

p1 = Person("John", 36)

p1.myfunc() # prints Hello my name is John

Imp: The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

1. Modify Object properties:

p1.age = 45

1. Delete Object properties:

del p1.age #this delete age property from object p1

1. Delete objects

del p1

**Python Inheritance**

Parent class (base class), child class (inherited class)

class Person: # create parent class

def \_\_init\_\_(self, fname, lname):

self.firstname = fname

self.lastname = lname

def printname(self):

print(self.firstname, self.lastname)

class Student(Person): # create child class of “Person” parent

Pass

x = Student("Mike", "Olsen") # declare object of student

x.printname()

2. Add \_\_init(..) in child class

class Student(Person):

def \_\_init\_\_(self, fname, lname):

#add properties etc.

Imp: When you add the \_\_init\_\_() function, the child class will no longer inherit the parent's \_\_init\_\_() function.

To keep the inheritance of the parent's \_\_init\_\_() function, add a call to the parent's \_\_init\_\_() function:

class Student(Person):

def \_\_init\_\_(self, fname, lname):

Person.\_\_init\_\_(self, fname, lname)

3. super() function :

This will make the child class inherit all the methods and properties from its parent.

class Student(Person):

def \_\_init\_\_(self, fname, lname):

super().\_\_init\_\_(fname, lname)

4. Add properties to child class

class Student(Person):

def \_\_init\_\_(self, fname, lname):

super().\_\_init\_\_(fname, lname)

Self.gradyear = 2002

x = Student("Mike", "Olsen", 2019)

5. Add Methods to child class

class Student(Person):

def \_\_init\_\_(self, fname, lname, year):

super().\_\_init\_\_(fname, lname)

self.graduationyear = year

def welcome(self):

print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

**Python Iterators**

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods \_\_iter\_\_() and \_\_next\_\_()

## Iterator vs Iterable:

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.

All these objects have a iter() method which is used to get an iterator.

Ex:

mytuple = ("apple", "banana", "cherry")

myit = iter(mytuple)

print(next(myit))

print(next(myit))

print(next(myit))

# prints apple banana cherry

Create an Iterator:

o create an object/class as an iterator you have to implement the methods \_\_iter\_\_() and \_\_next\_\_() to your object.

1. The \_\_iter\_\_() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
2. The \_\_next\_\_() method also allows you to do operations, and must return the next item in the sequence.

Ex:

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

x = self.a

self.a += 1

return x

myclass = MyNumbers()

myiter = iter(myclass)

print(next(myiter)) #prints 1

print(next(myiter)) #prints 2

print(next(myiter)) #prints 3

1. StopIteration statement :

To prevent the iteration to go on forever, we can use the StopIteration statement.

Ex:

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

If self.a < 10”

x = self.a

self.a += 1

return x

Else:

raise StopIteration

myclass = MyNumbers()

myiter = iter(myclass)

For x in myiter:

Print x # this loops till a is 9.

**Python Modules**

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

1. To create a module just save the code you want in a file with the file extension .py

Ex: mymodule.py file

def greeting(name):

print("Hello, " + name)

1. Use this module: (use **import**)

import mymodule

mymodule.greeting("Jonathan")

1. The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc)

Ex: mymodule.py

person1 = {

"name": "John",

"age": 36,

"country": "Norway"

}

Use:

import mymodule

a = mymodule.person1["age"]

print(a) # prints 36

1. Rename a module (use **as** keyword while importing)

import mymodule **as** mx

a = mx.person1["age"]

print(a)

1. Built-in modules

There are several built-in modules in Python.

Ex: platform

import platform

x = platform.system()

print(x) # prints windows

8. **dir()** function : is a built-in function to list all the function names (or variable names) in a module

Ex:

import platform

x = dir(platform)

print(x)

9. Import only parts **from** Module

from mymodule import person1

print (person1["age"])

**Python Date**

A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

import datetime

x = datetime.datetime.now()

print(x)

print(x.year)

print(x.strftime("%A")) # prints weekday

Creating Date objects:

x = datetime.datetime(2020, 5, 17)

Formatting method for date: strftime(format)

print(x.strftime("%B"))# prints Month

**Python Math**

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

Ex: min(), max(), abs(), pow(),

x = min(5, 10, 25)

y = max(5, 10, 25)

z = abs(-7.25)

w = pow(4, 3)

Python has also a built-in module called math, which extends the list of mathematical functions.

Ex:

import math

x = math.sqrt(64) # returns 8

y = math.ceil(1.4) # returns 2

z = math.floor(1.4) # returns 1

w = math.pi # constant value 3.14