Practice3

August 31, 2025

A tuple is a fixed-length, immutable sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values.

[2]:
```
tup=1,'ankit',1.1
tup
```

[2]:
```
(1, 'ankit', 1.1)
```

[3]:
```
tup[1]
```

[3]:
```
'ankit'
```

[1]:
```
# If an object inside a tuple is mutable, such as a list, you can modify it␣
 ↪in-place
tup1=(1,1.2,'ankit',[3,4],{'key1':'value1','key2':'value2'},(5,6),{1,2,3})
tup1[3][0]=30
tup1[4]['key1']='new_value'
tup1[6].add(4)
tup1
```

[1]:
```
(1,
 1.2,
 'ankit',
 [30, 4],
 {'key1': 'new_value', 'key2': 'value2'},
 (5, 6),
 {1, 2, 3, 4})
```

[2]:
```
tup1[3].append(40)
tup1
```

[2]:
```
(1,
 1.2,
 'ankit',
 [30, 4, 40],
 {'key1': 'new_value', 'key2': 'value2'},
 (5, 6),
 {1, 2, 3, 4})
```

[4]:
```
# there is no order in set, so we cannot access it by index or change it by␣
 ↪index
tup1[6][1] = 10
tup1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 2
      1 # there is no order in set, so we cannot access it by index or change i␣
 ↪by index
----> 2 tup1[6][1] = 10
      3 tup1

TypeError: 'set' object does not support item assignment
```

[7]:
```
(1,2,3)+(4,5,6)+('ankit',)
```

[7]:
```
(1, 2, 3, 4, 5, 6, 'ankit')
```

[8]:
```
(1,2,3)+(4,5,6)+'ankit',
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 (1,2,3)+(4,5,6)+'ankit',

TypeError: can only concatenate tuple (not "str") to tuple
```

[11]:
```
# Note that the objects themselves are not copied, only the references to them.
((1,2,3)+(4,5,6)+('ankit',))*2
```

[11]:
```
(1, 2, 3, 4, 5, 6, 'ankit', 1, 2, 3, 4, 5, 6, 'ankit')
```

[12]:
```
# Unpacking Tuples
tup2=(1,2,3,'ankit',1.1)
a,b,c,d,e=tup2
a
```

[12]:
```
1
```

[13]:
```
a,b,c=1,2,3
b,c=c,b
(a,b,c)
```

[13]:
```
(1, 3, 2)
```

[14]:
```
a,b,*rest=1,2,3,4,5,6,7,8,9
rest
```

[14]: [3, 4, 5, 6, 7, 8, 9]

[15]:
```
a,b,*_=1,2,3,4,5,6,7,8,9
_
```

[15]: [3, 4, 5, 6, 7, 8, 9]

[16]:
```
(a,b,_)
```

[16]: (1, 2, [3, 4, 5, 6, 7, 8, 9])

[17]:
```
('ankit','kiioo',1.1,1,2,3,'kiioo','kiioo','ankit').count('kiioo')
```

[17]: 3

[18]:
```
[1,2,3].count(4)
```

[18]: 0

[19]:
```
[1,1,1,1,2,2,1,0,1].count(1)
```

[19]: 6

In contrast with tuples, lists are variable-length and their contents can be modified in-place. You can define them using square brackets [] or using the list type function.

[22]:
```
lst=list(range(1,11,2))
lst
```

[22]: [1, 3, 5, 7, 9]

[23]:
```
gen=range(1,11,2)
list(gen)
```

[23]: [1, 3, 5, 7, 9]

[25]:
```
type(gen)
```

[25]: range

[26]:
```
lst=['ankit','kiioo']
lst.insert(1,'summi')
lst
```

[26]: ['ankit', 'summi', 'kiioo']

insert is computationally expensive compared with append, because references to subsequent elements have to be shifted internally to make room for the new element.
If you need to insert elements at both the beginning and end of a sequence, you may wish to explore collections.deque, a double-ended queue, for this purpose

Checking whether a list contains a value is a lot slower than doing so with dicts and sets.

[28]:
```
x=[1,2,3]
x.extend([4,5])
x
```

[28]:
```
[1, 2, 3, 4, 5]
```

[29]:
```
x=[1,2,3]
y=x.append([4,5])
y
```

[30]:
```
x
```

[30]:
```
[1, 2, 3, [4, 5]]
```

Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over.
Using extend to append elements to an existing list, especially if you are building up a large list, is usually preferable.

You can sort a list in-place (without creating a new object) by calling its sort function

[31]:
```
lst=[4,2,3,1]
lst.sort()
lst
```

[31]:
```
[1, 2, 3, 4]
```

[32]:
```
lst=['ankit','kiioo','summi','soumi']
lst.sort(key=len)
lst
```

[32]:
```
['ankit', 'kiioo', 'summi', 'soumi']
```

sort(key=len): This sorts the list based on the length of each string (number of characters). Since all strings have the same length (5 characters), the sort() method will maintain the original order of elements that have the same key (this is called a “stable sort” behavior, but note: Python’s sort is stable only when multiple elements have the same key? Actually, the stability of sort is guaranteed only when sorting with the same key, but here the key is the same for all, so the original order is preserved).

However, it’s important to note that when all elements have the same key value (same length), the sort operation doesn’t need to rearrange the elements. So the list remains unchanged.

[33]:
```
b = ['saw', 'small', 'He', 'foxes', 'six']
b.sort(key=len)
b
```

[33]:
```
['He', 'saw', 'six', 'small', 'foxes']
```

[35]:
```
b.sort(key=len, reverse=True)
b
```

[35]:
```
['small', 'foxes', 'saw', 'six', 'He']
```

[36]:
```
seq=[1,2,3,4,5]
seq[2:5]=seq[2:5][::-1]
seq
```

[36]:
```
[1, 2, 5, 4, 3]
```

[38]:
```
seq=[1,2,3,4,5]
seq[::2]
```

[38]:
```
[1, 3, 5]
```

[39]:
```
seq
```

[39]:
```
[1, 2, 3, 4, 5]
```

[2]:
```
sorted('ankit', reverse=True)
```

[2]:
```
['t', 'n', 'k', 'i', 'a']
```

[5]:
```
sorted(['ankit','kiiooo'],key=len, reverse=True)
```

[5]:
```
['kiiooo', 'ankit']
```

[6]:
```
sorted('horse race')
```

[6]:
```
[' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

[8]:
```
list(zip(['a','b','c'],[1,2,3]))
```

[8]:
```
[('a', 1), ('b', 2), ('c', 3)]
```

[11]:
```
list(zip(*(['ankit','kiio'],['summi','soumi'])))
```

[11]:
```
[('ankit', 'summi'), ('kiio', 'soumi')]
```

[ ]:
```
list(reversed(range(1,11)))
#Keep in mind that reversed is a generator. so it does not create the reversed␣
 ↪sequence
# until materialized (e.g., with list or a for loop).
```

[ ]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

dictionary

A more common name for it is hash map or associative array. It is a flexibly sized collection of key-value pairs, where key and value are Python objects. One approach for creating one is to use curly braces {} and colons to separate keys and values

[14]:
```
d=dict(k='kiio')
```

[15]:
```
d
```

[15]:
```
{'k': 'kiio'}
```

[16]:
```
'k' in d
```

[16]:
```
True
```

[17]:
```
d.pop('k')
```

[17]:
```
'kiio'
```

[18]:
```
d
```

[18]:
```
{}
```

[19]:
```
d={}
d['key1']='value1'
d['key2']='value2'
d
```

[19]:
```
{'key1': 'value1', 'key2': 'value2'}
```

[20]:
```
d.keys()
```

[20]:
```
dict_keys(['key1', 'key2'])
```

[21]:
```
list(d.keys())
```

[21]:
```
['key1', 'key2']
```

[22]:
```
d.values()
```

[22]:
```
dict_values(['value1', 'value2'])
```

[23]:
```
list(d.values())
```

[23]:
```
['value1', 'value2']
```

[24]:
```
d.items()
```

[24]:
```
dict_items([('key1', 'value1'), ('key2', 'value2')])
```

[25]:
```
list(d.items())
```

[25]:
```
[('key1', 'value1'), ('key2', 'value2')]
```

[26]:
```
d1={'a':1,'b':2}
```

[27]:
```
# You can merge one dict into another using the update method:

d.update(d1)
print(d)

# The update method changes dicts in-place, so any existing keys in the data␣
 ↪passed to update will have their old values discarded
```

```
{'key1': 'value1', 'key2': 'value2', 'a': 1, 'b': 2}
```

[28]:
```
d
```

[28]:
```
{'key1': 'value1', 'key2': 'value2', 'a': 1, 'b': 2}
```

[29]:
```
d1
```

[29]:
```
{'a': 1, 'b': 2}
```

[30]:
```
dict(zip(range(5), reversed(range(5))))
```

[30]:
```
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

While the values of a dict can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is hashability. You can check whether an object is hashable (can be used as a key in a dict) with the hash function.

[31]:
```
# can be hashed
print(hash(231.1),hash(231),hash("ankit"),hash('k'),hash((1,2)))
```

```
230584300921356519 231 -8256921466915439470 -6945423361581673583
-3550055125485641917
```

[ ]:
```
print(hash([1,2]))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 1
----> 1 print(hash([1,2]))
      2 print(hash({1,2}))
      3 print(hash({1:'ankit',2:'arpit'}))

TypeError: unhashable type: 'list'
```

[33]:
```
print(hash({1,2}))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 1
----> 1 print(hash({1,2}))

TypeError: unhashable type: 'set'
```

[34]:
```
print(hash({1:'ankit',2:'arpit'}))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 1
----> 1 print(hash({1:'ankit',2:'arpit'}))

TypeError: unhashable type: 'dict'
```

[35]:
```
print(hash((1, 2, [2, 3])))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 1
----> 1 print(hash((1, 2, [2, 3])))

TypeError: unhashable type: 'list'
```

Set

A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the set function or via a set literal with curly braces:

[36]:
```
print({1,2}.union({3,4}),{1,2} | {3,4})
print({1,2}.intersection({1,3,4}),{1,2} & {1,3,4})
print({1,2,3}.difference({1,2}),{1,2,3} - {1,2})
print({1,2,3}.symmetric_difference({2,3,4}),{1,2,3} ^ {2,3,4})
```

```
{1, 2, 3, 4} {1, 2, 3, 4}
{1} {1}
{3} {3}
{1, 4} {1, 4}
```

[37]:
```
{1,2}.issubset({1,2,3}), {1,2} <= {1,2,3}
```

[37]:
```
(True, True)
```

[38]:
```
{1,2,3}.issuperset({1,2}), {1,2,3} >= {1,2}
```

[38]:
```
(True, True)
```

[39]:
```
{1,2}.isdisjoint({3,4}), {1,2}.isdisjoint({2,3})
```

[39]:
```
(True, False)
```

[40]:
```
#All of the logical set operations have in-place counterparts, which enable you␣
 ↪to replace the contents
#of the set on the left side of the operation with the result. For very large␣
 ↪sets, this may be more efficient:

a = {1, 2, 3}
b= {2, 3, 4}
a |= b
a
```

[40]:
```
{1, 2, 3, 4}
```

[41]:
```
b
```

[41]:
```
{2, 3, 4}
```

[42]:
```
a &= {2,3}
a
```

[42]:
```
{2, 3}
```

[43]:
```
a^={3,4}
a
```

[43]:
```
{2, 4}
```

List, Set, and Dict Comprehensions

It allows you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression. They take the basic form:

[expr for val in collection if condition]

[44]:
```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in strings if len(x) > 2]
```

[44]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

[45]:
```
{x:x.upper() for x in strings if len(x) > 2}
```

[45]:
```
{'bat': 'BAT', 'car': 'CAR', 'dove': 'DOVE', 'python': 'PYTHON'}
```

[46]:
```
{x.upper() for x in strings if len(x) > 2}
```

[46]:
```
{'BAT', 'CAR', 'DOVE', 'PYTHON'}
```

[47]:
```
{key: value for key,value in zip([1,2,3],['a','b','c']) if value != 'b'}
```

[47]:
```
{1: 'a', 3: 'c'}
```

[49]:
```
list(map(print,[1,2]))
```

```
1
2
```

[49]:
```
[None, None]
```

[48]:
```
list(map(lambda x: sorted(x),[[3,1,2],[2,3,1]]))
```

[48]:
```
[[1, 2, 3], [1, 2, 3]]
```

[50]:
```
# nested list comprehension:

nested_list=[[1,2,3],['a','b','c'],[(1,2,3),{1,2,3},{'a':1}]]
flattened=[x for ind in nested_list for x in ind]
print(flattened)
```

```
[1, 2, 3, 'a', 'b', 'c', (1, 2, 3), {1, 2, 3}, {'a': 1}]
```

[51]:
```
def my_fun(x,y,z=1.1):
    if z!=1.1:
        return x+y
    else:
        return x+z
# x,y are positional arguments and z is keyword argument
```

```
print(my_fun(1,2))
print(my_fun(1,2,1.2))
```

```
2.1
3
```

The main restriction on function arguments is that the keyword arguments must follow the positional arguments (if any). You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

Namespaces, Scope, and Local Functions:

An alternative and more descriptive name describing a variable scope in Python is a namespace.

Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed

[52]:
```
a=2
def my_fun():
    a=a+3
my_fun()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[52], line 4
      2 def my_fun():
      3     a=a+3
----> 4 my_fun()

Cell In[52], line 3, in my_fun()
      2 def my_fun():
----> 3     a=a+3

UnboundLocalError: local variable 'a' referenced before assignment
```

[53]:
```
a=2
def my_fun():
    global a
    a=a+3
my_fun()
```

[54]:
```
a
```

[54]: 5

I generally discourage use of the global keyword. Typically global variables are used to store some kind of state in a system. If you find yourself using a lot of them, it may indicate a need for

objectoriented programming (using classes).

[ ]:
```
def f():
 a = 5
 b = 6
 c = 7
 return a, b, c
a, b, c = f()
print(a,b,c)
```

```
5 6 7
```

[56]:
```
import re
i = re.search('a', 'ankit')
i
```

[56]:
```
<re.Match object; span=(0, 1), match='a'>
```

[57]:
```
'           ankit' .strip()
```

[57]:
```
'ankit'
```

[59]:
```
re.sub('a', 'A', 'ankit')
```

[59]:
```
'Ankit'
```

[60]:
```
re.sub('[aeiou]', '1', 'ankit')
```

[60]:
```
'1nk1t'
```

[61]:
```
re.sub('[%#@&]', '', 'an#ki@t%')
```

[61]:
```
'ankit'
```

Python has support for so-called anonymous or lambda functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the lambda keyword, which has no meaning other than "we are declaring an anonymous function"

One reason lambda functions are called anonymous functions is that , unlike functions declared with the def keyword, the function object itself is never given an explicit ___name___ attribute

[62]:
```
square=lambda x: x**2
square(5)
```

[62]:
```
25
```

[63]:
```
strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
strings.sort(key=lambda x: len(set(list(x))))
strings
```

[63]:
```
['aaaa', 'foo', 'abab', 'bar', 'card']
```

[64]:
```
# iterating over a dict yields the dict keys:

some_dict = {'a': 1, 'b': 2, 'c': 3}
for key in some_dict:
    print(key)
```

```
a
b
c
```

An iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as min, max, and sum, and type constructors like list and tuple:

[65]:
```
some_dict = {'a': 1, 'b': 2, 'c': 3}
print(list(iter(some_dict)))
```

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

A generator is a concise way to construct a new iterable object. Whereas normal functions execute and return a single result at a time, generators return a sequence of multiple results lazily, pausing after each one until the next one is requested. To create a generator, use the yield keyword instead of return in a function

[66]:
```
def squares(n=10):
 print('Generating squares from 1 to {0}'.format(n ** 2))
 for i in range(1, n + 1):
     yield i ** 2

gen = squares()

for x in gen:
    print(x, end=' ')
```

```
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Another even more concise way to make a generator is by using a generator expression. This is a generator analogue to list, dict, and set comprehensions; to create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets

[67]:
```
gen = (x ** 2 for x in range(100))
gen
```

[67]:
```
<generator object <genexpr> at 0x7cc8f6bd7990>
```

[68]:
```
next(gen)
```

[68]:
```
0
```

[69]:
```
next(gen)
```

[69]:
```
1
```

[70]:
```
next(gen)
```

[70]:
```
4
```

[71]:
```
sum(x ** 2 for x in range(100))
```

[71]:
```
328350
```

[72]:
```
dict((i, i **2) for i in range(5))
```

[72]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

[73]:
```
#cErrors and Exception Handling:

def attempt_float(x):
 try:
     return float(x)
 except (TypeError, ValueError):
     return x
```

[74]:
```
attempt_float('1.234')
```

[74]:
```
1.234
```

[76]:
```
attempt_float('async')
```

[76]:
```
'async'
```

[77]:
```
# you can have code that executes only if the try: block succeeds using else:

def type_check(x):
    try:
        print(float(x))
    except (TypeError,ValueError):
        print(x)
    else:
        print("hi")
```

```
    finally:
        print("bye")
print(type_check(3))
print(type_check("an"))
```

```
3.0
hi
bye
None
an
bye
None
```

[78]:
```
type_check(3)
```

```
3.0
hi
bye
```

[80]:
```
type_check("an")
```

```
an
bye
```

By default, the file is opened in read-only mode 'r'. We can then treat the file handle f like a list and iterate over the lines like so

There is also the 'x' file mode, which creates a writable file but fails if the file path already exists.

[86]:
```
# f.read(10) # reads 10 characters from the file

# Mode --> Description
# r --> Read-only mode
# w --> Write-only mode; creates a new file (erasing the data for any file with␣
 ↪the same name)
# x --> Write-only mode; creates a new file, but fails if the file path already␣
 ↪exists
# a --> Append to existing file (create the file if it does not already exist)
# r+ --> Read and write b Add to mode for binary files (i.e., 'rb' or 'wb')
# t -->Text mode for files (automatically decoding bytes to Unicode). This is␣
 ↪the default if not specified. Add t to other
# modes to use this (i.e., 'rt' or 'xt')
```

To write text to a file, you can use the file's write or writelines methods

[88]:
```
with open('tmp.txt', 'w') as handle:
    handle.writelines(x for x in open('sample.txt') if len(x) > 1)
with open('tmp.txt') as f:
    lines = f.readlines()
```

```
lines
```

[88]:
```
['111.11']
```

[ ]:
```

```