

**Lists**:

Now, we will look at built in data structures in Python which are extremely important for building real applications.

Earlier we saw square brackets [ ] to define a list or a sequence of objects.

In between the brackets we can have object of any type, so we can have a list of strings

["a", "b", "c"]

And assign it to a variable,

letters = ["a", "b", "c"]

We can also have a list of numbers, Booleans or even a *list of lists!*

Here we have a matrix, which is a 2 dimensional list,

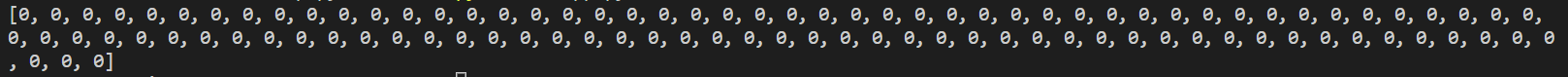
matrix = [[0,1], [2,3]]

Suppose we want to print a list of 100 zeroes, we can do something like this,

zeros = [0] \* 100

print(zeros)

O/P:



So using an *asterisk* ***\**** here, we ***repeated the items*** *in a list*.

***Concatenate*** *two lists(just use* ***+*** *operator)*:

letters = ["a", "b", "c"]

zeros = [0] \* 5

combined = zeros + letters

print(combined)

O/P: 

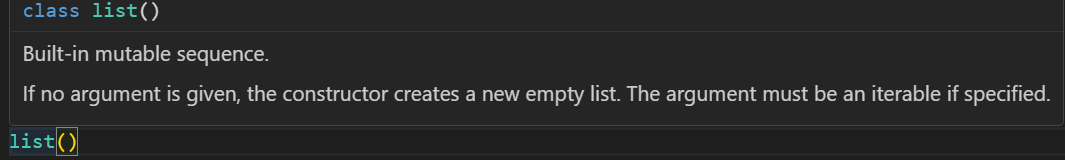
So every object in a list can be of a different type. *We can combine a list of numbers with strings and Booleans or even lists*.

*Example*:

You want to have a list of numbers 0,1,2…all the way to 20.

Solution:

We use a list function.



As we can see this function takes an iterable which will be converted to a list.

So we use range function here which returns a range object which is iterable, so we can iterate or loop over it.

numbers = list(range(20))

print(numbers)

O/P:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Since strings are also iterable, we can pass them to list function,

letters = list("Hello World")

print(letters)

O/P: ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']

Each letter in our original string is an item in this list.

Note: We can print the number of items in a list using *len* function.

print(len(letters))

O/P:11

**Accessing Items**:

Here we have a list of 4 items

letters = ["a", "b", "c", "d"]

We can use [ ] square brackets to access individual items from this list.

print(letters[0]) //a

letters[0] returns first item from this list.

To get first item from the end of list use [-1]

print(letters[-1]) //d

Using square bracket notation we can also modify items

letters = ["a", "b", "c", "d"]

letters[0] = "A"

print(letters) // ['A', 'b', 'c', 'd']

To get first 3 items from the list we can use *slicing*,

letters = ["a", "b", "c", "d"]

letters[0] = "A"

print(letters[0:3]) // ['A', 'b', 'c']

print(letters) // ['A', 'b', 'c', 'd']

Just like strings if you do not specify the first argument(*start index*) in 0:3, 0 will be assumed by default. With same logic if second argument(*end index*) in 0:3 is not specified then the expression will return a new list with all the items of original list.

print(letters[0:]) // ['A', 'b', 'c', 'd']

With No start index and end index,

print(letters[:]) // ['A', 'b', 'c', 'd']

We get a copy of the original list.

Similarly we can use *step* when we want to return every 2nd or 3rd element from the list.

numbers = list(range(20))

print(numbers[::2])

O/P:

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18], So we see that we get all the even numbers between 0-20.

We can also reverse the order of original list and return a new one,

numbers = list(range(10))

print(numbers[::-1]) //[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

**List Unpacking**:

There are times that you may want to get individual items on a list and assign them to different variables.

numbers = [1,2,3]

first = numbers[0]

second = numbers[1]

third = numbers[2]

Perhaps you want to use these variables in a few complex expressions in your code.

There is a more cleaner and elegant way to achieve the same result, that is what we call *list unpacking*.

So *we can unpack this list into multiple variables*.

numbers = [1, 2, 3]

first, second, third = numbers

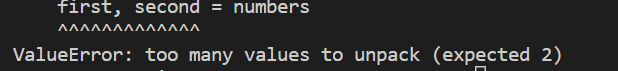
It is identical to above code.

Note: It is **important** that number of variables we have on the left side of assignment operator should be equal to number of elements we have in list.

numbers = [1, 2, 3]

first, second = numbers

O/P: Error



What if we have too many items in the list but we only care about first two,

We can unpack rest of the items in a variable called \*other,

numbers = [1, 2, 3, 4, 5, 6, 7, 8]

first, second, \*other = numbers

print(second) //2

print(other) // [3, 4, 5, 6, 7, 8]

other is list of all items after second item.

In this example we are seeing packing and unpacking both in action.

First we tried to *unpack* first two items in numbers list into first and second variable and then we are *packing* all the other items into *\*other* variable.

Note: We saw a similar syntax while learning variable number of arguments where we prefix our parameter with an asterisk.

def multiply(\*numbers):

    print(numbers)

multiply(1, 2, 3, 4) //(1,2,3,4)

This is exactly what is happening while packing items in a list.

In case we need only first and last item from the list, *we move \*other to middle*,

numbers = [1, 2, 3, 4, 5, 6, 7, 8]

first, \*other, last = numbers

print(last) //8

print(other) //[2, 3, 4, 5, 6, 7]

**Looping over list**:

In this lecture we will learn how to loop over lists,

We can use for loop,

letters = ["a", "b", "c"]

for letter in letters:

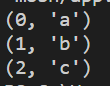
    print(letter) //a b c

What if we also want index of each items as well, for that we have a built in function called enumerate, we call it like this,

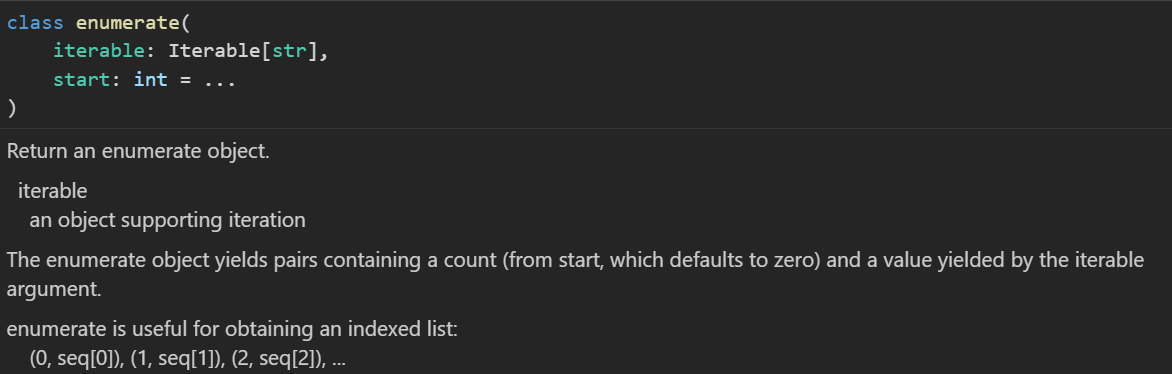
letters = ["a", "b", "c"]

for letter in enumerate(letters):

    print(letter)

O/P: 

enumerate function returns a *enumerate object* which is iterable.



In each iteration this enumerate object will give us a *tuple* like (0, 'a') (1, 'b') (2, 'c').

A *tuple* is like a list but it is read only, We cannot add new items to it.

*First item in these tuples is index and second is item at that index*.

To access only the index part of the tuples we can use [ ] bracket notation,

letters = ["a", "b", "c"]

for letter in enumerate(letters):

    print(letter[0]) // 0 1 2

To get both together,

letters = ["a", "b", "c"]

for letter in enumerate(letters):

    print(letter[0], letter[1])

O/P: 

We can do the same operation in a cleaner way *by unpacking both items in tuple into variables and then printing them*,

letters = ["a", "b", "c"]

for index, letter in enumerate(letters):

    print(index, letter)

O/P:  Here we unpack the tuple in for loop statement.

**Adding/Removing Items in list**:

In this lecture we will learn how to add new items to a list or remove existing items.

For adding items we have two options depending on where you want to add those,

🡪 To *add item at the end of list*, use append method.

letters = ["a", "b", "c"]

letters.append("D")

print(letters) //['a', 'b', 'c', 'D']

🡪 To *add item at a specific position*, use insert method(*takes index where you want to place item as its first argument*),

letters = ["a", "b", "c"]

letters.append("D")

letters.insert(0, "-")

print(letters) //['-', 'a', 'b', 'c', 'D']

🡪 To *remove item from the end of list*, use pop method.

letters = ["a", "b", "c"]

letters.append("D")

letters.insert(0, "-")

letters.pop()

print(letters) //['-', 'a', 'b', 'c'] will remove ‘D’ from the list.

If we specify index in the pop method, the item at that index will be removed.

letters.pop(0) //['a', 'b', 'c', 'D'] ‘-’ is removed from 0 index

Note: Sometimes we do not know the index of the element that we want removed, so we can use remove method for the same.

letters.remove("b") //['-', 'a', 'c', 'D'] ‘b’ is removed

remove method removes the *first occurrence* of the letter ‘b’

If you want to remove all ‘b’ from the list, we have to loop over this list and remove each one individually.

🡪 We can use **del** statement to delete a range of elements from the list, that is what makes it different from pop method.

letters = ["a", "b", "c"]

letters.append("D")

letters.insert(0, "-")

print(letters) //['-', 'a', 'b', 'c', 'D']

del letters[0:3]

print(letters) //['c', 'D']

🡪 To remove all items from the list use clear method.

print(letters) //['-', 'a', 'b', 'c', 'D']

letters.clear()

print(letters) //[]

**Finding Items**:

There are times when you want to find index of a given object in a list.

letters = ["a", "b", "c"]

Let us say we want to find the index of letter ‘a’ in our list.

print(letters.index("a")) //0

If you try to get the index of an item that does not exist in the list,

print(letters.index("d"))



Note: This behavior is different from other programming languages like c or JavaScript where we get -1 if the object is not in list.

So to avoid an error like this, we can write a conditional statement to check if this item exists in list,

if "d" in letters:

    print(letters.index("d"))

We do not get any error now.

🡪 To get number of occurrences of a given item in a list we can use count method.

print(letters.count("d")) //0

**Sorting Lists**:

Here we have a list of numbers that are not in any particular order.

numbers= [3, 51, 2, 8, 6]

For sorting this list we use sort method.

numbers = [3, 51, 2, 8, 6]

numbers.sort()

print(numbers) //[2, 3, 6, 8, 51]

Now our numbers are sorted in ascending order.

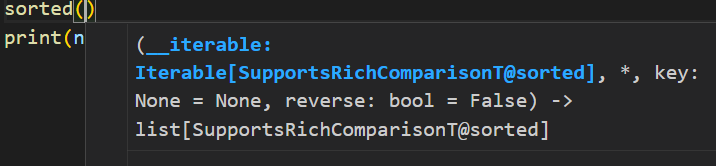
🡪 To sort the list in descending order , we use a keyword argument called reverse and set its value to *True*.

numbers = [3, 51, 2, 8, 6]

numbers.sort(reverse=True)

print(numbers) //[51, 8, 6, 3, 2]

Note: The sort method modifies the original list, but if you want to return a new sorted list use built in function **sorted**.



As you can see this function takes an iterable. So we can pass our *numbers* list.

numbers = [3, 51, 2, 8, 6]

print(sorted(numbers)) //[2, 3, 6, 8, 51]

print(numbers) //[3, 51, 2, 8, 6]

As you can see, original list is not modified. Similarly we can set reverse argument to True in order to reverse the sorting.

print(sorted(numbers, reverse=True)) //[51, 8, 6, 3, 2]

So, Sorting numbers and strings is pretty easy and straightforward but what if we are dealing with a list of *complex objects*.

For example , what if we have a list of tuples?

Suppose we are building an application for processing orders and we have a list of order items.

items =[

    ("Product1", 10),

    ("Product2", 9),

    ("Product3", 12)

]

Every item in this list is a tuple with two items, Product name and its price.

*How to sort this list of items as per the unit price*?

First we will define a function *sort\_item* that python will use for sorting lists,

def sort\_items(item):

item parameter is a *tuple* in our case and the function will return a value that will be used for sorting.

def sort\_items(item):

    return item[1] //1 means the ‘price’ in our tuples, 10,9,12

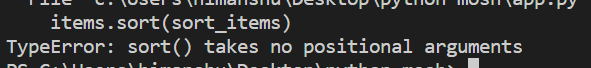
Since now we are dealing with a list of numbers we can easily sort that list.

Now we pass our ***function reference*** as the argument in sort function(*not calling function*)

items.sort(sort\_items)

print(items)

But we see an error,



sort() takes no positional arguments means we have to provide it a *keyword argument* instead.

items.sort(key=sort\_items)

print(items)

Now our list is sorted as per our provided key.

**Lambdas**:

We got our list sorted as per the price of each item but the way we implemented our solution using *sort\_items* function is a little ugly. In this lecture we will make it better looking.

If you have experience with any other programming language, you probably heard of the term *lambda expression* or *lambda function*.

**“***It is a simple one line anonymous function that we can pass to other functions***”**.

items.sort(key=sort\_items)

In this line we are passing a reference to our *sort*\_*items* function.

We can improve this code and make it cleaner by using a lambda expression or an anonymous function.

So we do not have to define it first and then pass it inside *sort*.

The syntax of writing *lambda* function is this,

lambda parameters : expression

using this syntax we can write sort\_items function,

def sort\_items(item):

    return item[1]

So in lambda we have,

items.sort(key=lambda item: item[1])

Now we do not need to write *def* to define function name, parentheses for parameters or a *return* statement.

So this is a shorter and cleaner way to define a function that we are going to use only once as an argument to another function.

items.sort(key=lambda item: item[1])

print(items) //[('Product2', 9), ('Product1', 10), ('Product3', 12)]

**Map functions**:

Over the next two lectures , we will use more example on how to use *lambdas* in our program.

items = [("Product1", 10), ("Product2", 9), ("Product3", 12)]

Here we have a list of items, Let us imagine we want to *transform this list into different shape*.

So currently each item in this list is a tuple of two items. Let us say we are only interested in the price of each item, so we want to transform this list into a list of prices(*numbers*).

Here is the *basic* way,

prices = []

for item in items:

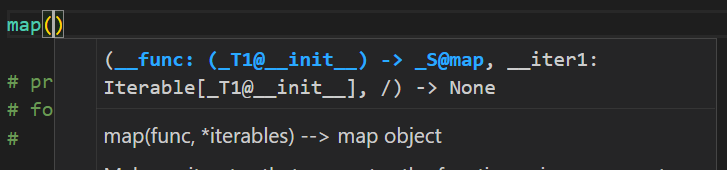
    prices.append(item[1])

print(prices) //[10, 9, 12]

we have transformed or *mapped* our list into different list.

Here is the *elegant* way,

Instead of loop we can use the *map* function,



This function takes two parameters, a function and one or more Iterables.

So as the first argument, we can pass a lambda function and as the second argument we can pass our list of items. This map function *will apply to the lambda function for each item on the list*.

map(lambda item: item[1], items)

*what is happening in background*?

This map function will iterate over our iterable *items* and it will *call* this *lambda* function *on each item of this iterable*.

Furthermore it will return a *map object*, which is another iterable.

x = map(lambda item: item[1], items)

print(x) //<map object at 0x000001E1B446AAD0>

Let us iterate over this,

x = map(lambda item: item[1], items)

for item in x:

    print(item) // 10 9 12

Alternatively we can convert this map object into a list object,

prices = list(map(lambda item: item[1], items))

print(prices) //[10, 9, 12]

**Filter Functions**:

Let us say we want to filter our list of items and only get items with price greater than or equal to 10.

The *Basic* way:

Define an empty list, iterate over our list of items, for each item get the price and if it matches our criteria we append it.

filtered = []

for item in items:

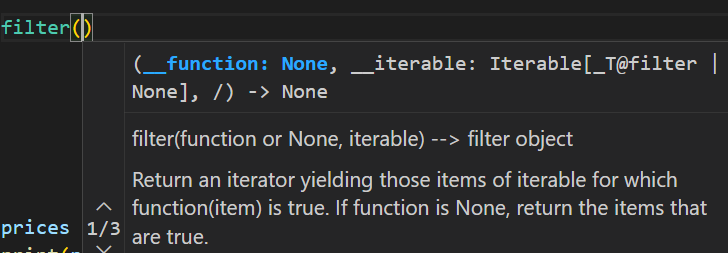
    if item[1] >= 10:

        filtered.append(item)

print(filtered) //[('Product1', 10), ('Product3', 12)]

The *Better* way:

For the purpose we use built in filter function.



Just like the map function, it also takes two parameters a function and a iterable.

So it will apply this function on each item of the iterable, if the item matches a criteria, it will return it.

filtered\_prices = list(filter(lambda item: item[1] >= 10, items))

print(filtered\_prices) // [('Product1', 10), ('Product3', 12)]

**List Comprehension**:

We have another useful technique in python called comprehension and here is the basic syntax,

🡪 **[***expression* for *item* in *items***]**

The for loop part is familiar where we are iterating over an iterable and then applying an *expression* on each item.

So if you want to get price for each item, we can write and expression like,

[item[1] for item in items]

This is what we call list comprehension and it produces the exact same result as

list(map(lambda item: item[1], items))

As we can see code is shorter and cleaner, we do not need parenthesis or colon.

Most Python community agrees that *list comprehension* is the preferred way to map and filter list as it is cleaner and efficient.

For filtering ,

print([item for item in items if item[1] >= 10])

O/P: [('Product1', 10), ('Product3', 12)]

**Zip Function**:

list1 = [1, 2, 3]

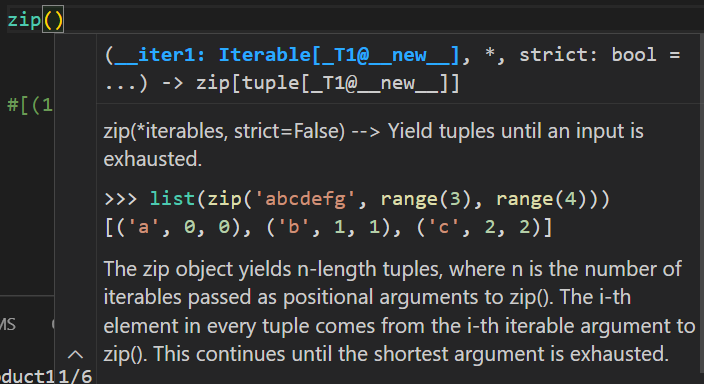
list2 = [10, 20, 30]

Here we have two list, Let us say we want to combine these two lists into a single list of tuples like this,

In the first tuple we are going to have the first element of each list. Similarly, in second tuple we should have second item of each list and so on…like this,

[(1, 10), (2, 20), (3, 30)]

We cannot use map or list comprehension here because they use a single list, for combining multiple lists we use built in zip function.



This function takes multiple Iterables and it will combine them in expected way.

print(zip(list1, list2))

// <zip object at 0x000001C0D294C140>

It returns a zip object which is also iterable and we can iterate over it.

Let us directly convert it into list,

print(list(zip(list1, list2))) // [(1, 10), (2, 20), (3, 30)]

There is a cool thing you can do with this function, since it takes n number of iterables,

print(list(zip("abc", list1, list2)))

O/P: [('a', 1, 10), ('b', 2, 20), ('c', 3, 30)]

Our string “abc” which is also an iterable is spread across multiple tuples in this list.

This is pretty powerful, If you want to achieve the same thing in other programming languages, you have to write several lines of code but in python we can do it in a single line.

**Stacks**:

In programming we have a common data structure called *stack* which *resembles a stack of items in the real world*.

Imagine you have a stack of books, the last book you put on top of your stack is the first book you can remove. This behavior is called…



This is a stack data structure and it is very common in real world applications.

Good example is our browser, *whenever we navigate to new website, our browser keeps our browsing session in a stack so when we click on back button it takes us to the previous website*.

*This is what happens*:

🡪Suppose these numbers represent each website we browsed and 3 is the current website.

[1, 2, 3]

🡪 When we press back button browser removes the item at the top of the stack and redirect us to the previous website,

[1, 2]

🡪 Now we press back button a couple more times and now we end up with an empty stack at this point the browser will disable back button.

[ ]

This is how a stack works, **Last in First Out**.

Now let us see, how to use a stack in python.

Basically we can use a list object as a stack,

browsing\_session = []

Now let us say the user navigates to website numbers one so we call *append* method and add the address of the current website,

browsing\_session.append(1)

User now navigates to website #2, #3 and so on…

browsing\_session.append(2)

browsing\_session.append(3)

Let us see what we have in browsing\_session stack now,

print(browsing\_session) // [1, 2, 3]

We have a list of 3 items, now when the user presses back button we should remove the last item in this list for that we use *pop* method,

browsing\_session.pop()

print(browsing\_session) //[1, 2]

Now we need to redirect user to website that is on the top of stack, so we use index of ***[****-1****]***.

print(f"redirect--> {browsing\_session[-1]}") //redirect--> 2

To check if stack is empty in order to disable back button,

if not browsing\_session:

    print('disable')

*Recap*:

Used append to add an item on top of stack,

Used pop method to remove an item from top of stack.

Check to see if stack is empty or not.

Index of -1 to get the item from top of stack.

*Recap code*:

browsing\_session = []

browsing\_session.append(1)

browsing\_session.pop()

if not browsing\_session:

    browsing\_session[-1]

**Queue**:

We have another very important data structure called *queue* which has FIFO structure.



It resembles a queue in real world, Let us say a queue of people outside an ATM machine. The first person in queue will be the first person to get out as well.

Technically you can use a List to represent a queue in python. So let us say we have a queue of three items.

[1, 2, 3]

*If you want to remove an item from a queue, you should remove the one at the beginning as opposed to one at the end*.

Note: If you are dealing with a large list, you might see some performance issues. Because every time we remove an item from the beginning of the list, all the other items need to be shifted to the left.

So if you have a list of 1000 items, when you remove one item 999 items need to be moved in memory.

In situations like this we need to use a ***deque*** object. For that import ***deque*** from collections module.

from collections import deque

We wrap an empty list with ***deque*** object, this object has similar methods that we have in list object.

queue = deque([])

queue.append(1)

queue.append(2)

queue.append(3)

Now to remove an item from the beginning of queue, we call *popleft* method(*it is not available in list object*).

queue.popleft()

print(queue) // deque([2, 3])

Similarly, For checking if the queue is empty we can use *not* operator,

if not queue:

    print("empty")

**Tuples**:

**“***A tuple is basically a read-only list, we can use it to contain a sequence of objects but we cannot modify this sequence like adding a new object or modify and remove an existing one***”**

point = (1, 2)

Here we have a *tuple* called *point*. We define this tuple inside a pair of parentheses.

Note: If we even remove parentheses, Python will still treat it as a tuple. Here key word is a *comma* ,

point = 1, 2

print(type(point)) // <class 'tuple'>

empty tuple 🡪( )

*concatenating a tuple*:

point = (1, 2) + (3, 4)

print(point) // (1, 2, 3, 4)

*Repeating a tuple*:

point = (1, 2) \* 3

print(point) // (1, 2, 1, 2, 1, 2)

*Convert a list and string to a tuple*:

point = tuple([1, 2])

print(point) //(1, 2)

point = tuple("Hello World")

print(point) // ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd')

*Access tuple items*:

point = (1, 2, 3)

print(point[0]) //1

print(point[0:2]) //(1, 2)

*Unpack* tuple:

point = (1, 2, 3)

x, y, z = point

print(z) //3

Since we know that tuples are immutable, that means we cannot mutate them, so if try to modify point[0] = 10, we get,



*Where we use tuple in real world*?

Suppose we have a sequence of objects that we do not want to accidently modify , there we use tuples to avoid accidental errors like this.

**Swapping Variables**:

x = 10

y = 11

print(x) //10

print(y) //11

temp = x    # copy the value of x to temp as backup

x = y       # overwrite x with y, copy the value of y to x

y = temp    # old value of x is stored in temp, so copy temp to y

print(x) //11

print(y) //10

This above code is how we do swapping of two variables in any programming language. But in Python we can do the same swap using only one line of code and no third variable.

x = 10

y = 11

print(x) //10

print(y) //11

x, y = y, x     #Just this one line

print(x) //11

print(y) //10

*What happens in background*?

A tuple is unpacking into another tuple. So using same strategy we can define multiple variables in the same line.

a,b = 1,2

“*Defining a tuple on right side and unpacking it in left*”

**Arrays**:

Lists are very useful but if we are dealing with a large sequence of numbers, we have a more efficient data type in python called *array*.

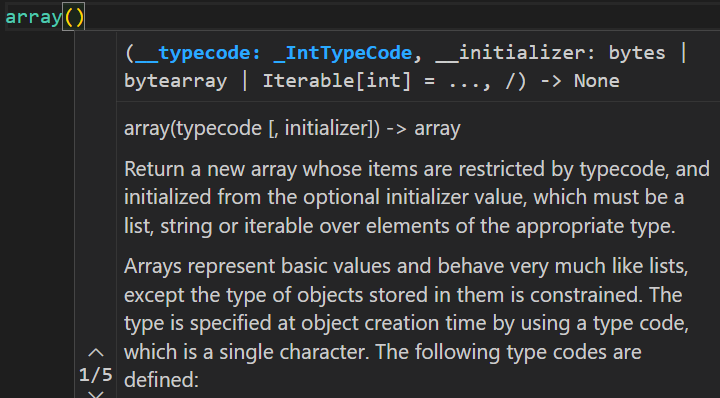
These arrays take less memory and perform a little bit faster. But you will see the difference only when dealing with a large list of numbers, say 10,000 or more.

For 90% of cases we will use lists but if we run our program and see some performance problems then we can think of replacing our list with an array.

To use an array we first need to import it from the array module.

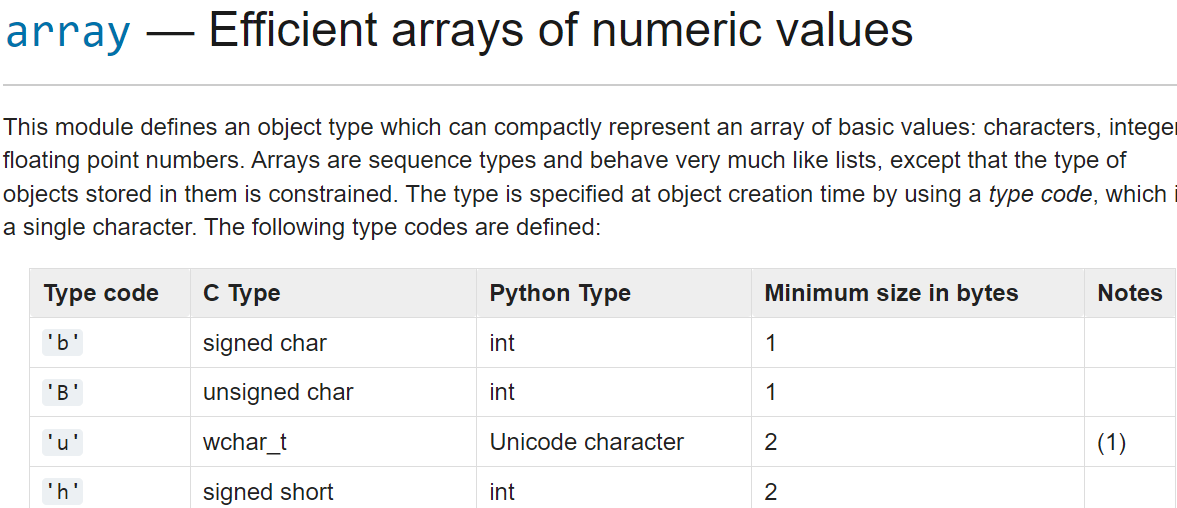
from array import array

Now we call array() and look at its definition.



Note: The first argument is *typecode*, which is a string that determines the type of object in your array.

Search ‘python 3 typecode’ in google and find all the latest typecode in python.



So it is a string of one character that determines the type of object in your list.

For example, if you are dealing with signed integer you should use lowercase ‘i’ .



numbers = array("i", [1, 2, 3])

As first argument use ‘i’ and as a second argument pass list of integers.

In this *numbers* object similar to lists, we have methods for adding, removing objects like append, insert(*to insert a value at a specific index*), pop etc. same as that of lists.

We can also access items using their index under [*square bracket notation*], however *unlike lists objects in array are* ***typed****. For example if we try to put a floating point number inside an array of integers, we will get an error*.

from array import array

numbers = array("i", [1, 2, 3])

numbers.append(4.2)

print(numbers) // **TypeError:** 'float' object cannot be interpreted as an integer

“*So every object in the array must be of same type which is determined at the time of creating the array using type code*”.

**Sets**:

In Python we have another very useful data structure called a ***set****, which is basically a collection with no duplicates*.

For example, here is a list with duplicate items,

numbers = [1, 1, 2, 3, 4, 4]

If you want to remove duplicates from this list we can convert it into a set.

numbers = [1, 1, 2, 3, 4, 4]

uniques = set(numbers)

print(uniques) // {1, 2, 3, 4}

Note: Curly braces { } are used to define a set, so here we can define a second set.

second = {1, 5}

Similar to list, we can add new items or remove existing items from a set. These below are basic ones.

second = {1, 5}

second.add(6) 🡪 Add a New items

print(second) //{1, 5, 6}

second.remove(5) 🡪 Remove an existing one

print(second) //{1, 6}

print(len(second)) //2 Find length of a set

But sets shine in the powerful mathematical operations that are supported by them.

For example, **|** 🡨this operator represents union of two sets,

numbers = [1, 1, 2, 3, 4]

first = set(numbers)

second = {1, 5}

print(first | second) // {1, 2, 3, 4, 5}

“*The union of these two sets has all the items that are either in first or second set*”.

🡪 & operator represents intersection of two sets,

numbers = [1, 1, 2, 3, 4]

first = set(numbers)

second = {1, 5}

print(first & second) // {1}

“*It return the items that are common in both sets*”.

🡪 **-** operator represents difference of two sets.

numbers = [1, 1, 2, 3, 4]

first = set(numbers) 🡪 🡪 {1, 2, 3, 4}

second = {1, 5}

print(first - second) // {2, 3, 4}

print(second - first)    //{5}

*It returns additional items that are not in other set*.

🡪 Finally we have ^ operator which represents symmetric sets,

numbers = [1, 1, 2, 3, 4]

first = set(numbers)

second = {1, 5}

print(first ^ second) // {2, 3, 4, 5}

***“****It return items that are either in first or second set but not in both****”***.

Note: One thing we need to know about sets is that unlike lists they are *unordered collection*. It means the *items that we have in a set are not in sequence so we cannot access them using an* ***index***.

second = {1, 5}

print(second[0]) //Error: TypeError: 'set' object is not subscriptable

So generally we use these above operations on a set or check to see existence of an item in a set.

if 1 in second:

    print("Yes") //Yes

So the definition of a set is…

***Set is an unordered collection of unique items***.

**Dictionary**:

In python we have a very powerful data structure called ***dictionary*** that is *basically a collection of* ***key*** *–* ***value*** *pairs*. We use it to map a key to a value.

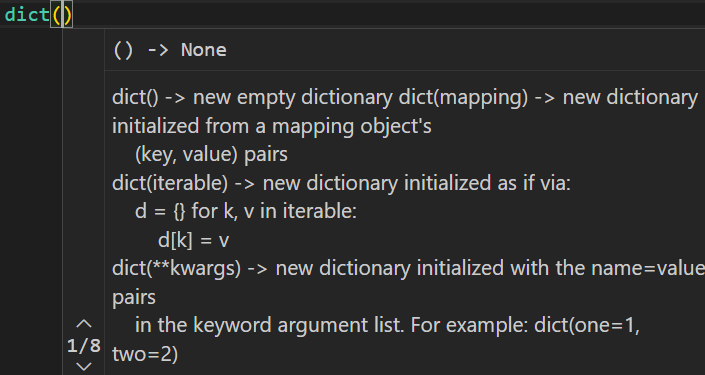
A real world example of this, is a phonebook where we map a person’s name to contact details. So we use a person’s name as a key and their contact information as value.

Example:

point = {"x": 1, "y": 2}

here I am using a string as the *key* and an integer as the *value*. *In Python we can only use immutable types for the keys*, so quite often we use strings and numbers. *But the value can be of any type*.

Note: Just like we have *list*(), *tuple*() and *set*() functions, we can also use built in ***dict***() function that we can use to create a dictionary.



As you can see that *dict* function takes one or more keyword arguments.

point = dict(x=1, y=2)

print(point) // {'x': 1, 'y': 2}

*How to access an item from dictionary*?

We can get the value associated with a key using *index*.

point = dict(x=1, y=2)

print(point["x"]) //1

Notice that index is the name of a key. Since dictionaries are a collection of key value pairs, we cannot access an item using numeric index, as we do with lists.

Other examples:

*Assigning new value to an existing key*:

point = dict(x=1, y=2)

point["x"] = 10

print(point) //{'x': 10, 'y': 2}

*Adding new key to a dictionary*:

point["z"] = 20

print(point) //{'x': 10, 'y': 2, 'z': 20}

Note: We will get an error if use an invalid key and our program crash,

print(point["a"])

KeyError: 'a'

So we can check the existence of a key in dictionary

if "a" in point:

    print(point["a"])

print("not exist") //not exist

You can also use get method to avoid crash and we get a *none* object.

print(point.get("a")) //none

*To delete an item*:

We use del,

del point["x"]

print(point) //{'y': 2, 'z': 20}

*To iterate over a dictionary*,

point = dict(x=1, y=2, z=3)

for key in point:

    print(key) //x y z

In each iteration our loop variable will hold the key of an item.

We can print simply the key or the value associated with key,

point = dict(x=1, y=2, z=3)

for key in point:

    print(key, point[key])



We can also unpack, key and value into different variables at the time of iteration. Observe we get the same result.

point = dict(x=1, y=2, z=3)

for key, value in point.items():

    print(key, value)



**Dictionary comprehensions**:

Look at this piece of code,

values = []

for x in range(5):

    values.append(x \* 2) //[0, 2, 4, 6, 8]

At the top we are defining an empty list, then we are iterating over the range object and in each iteration we are appending x\*2 into our list.

Whenever we have this pattern , we can either use map function or *list comprehension*.

values = [x \* 2 for x in range(5)]

print(values) //[0, 2, 4, 6, 8]

These comprehensions are not limited to lists, we can also use them with sets and dictionaries, *If we replace these square braces with curly braces, and we get a set*,

values = {x \* 2 for x in range(5)}

print(values) //{0, 2, 4, 6, 8}

*What is syntactical difference between a set and a dictionary*?

For both these data structures we use { }. In sets we just have values but in dictionaries we have key value pairs that are separated using a colon( **:** ).

So in dictionaries, we can even map numbers to strings,

{1:"a", 2:"b"}

*Dictionary comprehension*, we just have to introduce a key 🡪

values = {x: x \* 2 for x in range(5)}

print(values) // {0: 0, 1: 2, 2: 4, 3: 6, 4: 8}

Note: If try to create a tuple using the same logic as lists, set and dictionaries.

values = (x \* 2 for x in range(5))

print(values)

O/P: <generator object <genexpr> at 0x00000157FAE48380>

We get a generator object.

**Generator Expressions**:

Here we have,

values = [x \* 2 for x in range(10)]

for x in values:

    print(x) //0 2 4 6 8 10 12 14 16 18

This is ok but there are situations when we need to work with a really large dataset or perhaps an infinite stream of data.

In that case we should not store all those values in the memory because that is very memory inefficient. In situations like this, it is more efficient to use a *generator object*.

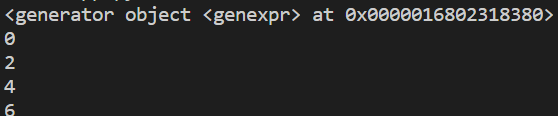
***“****Generator objects are iterable so we can iterate over them and with each iteration they generate or spit out a new value****”***. So unlike lists they do not store all the values in memory, they generate a new value with each iteration.

values = (x \* 2 for x in range(10))

print(values)

for x in values:

    print(x)



After replacing [ ] with ( ) and printing *values*, we get a generator object.

What is interesting is the *size of generator objects*,

from sys import getsizeof

values = (x \* 2 for x in range(1000))

print("gen:", getsizeof(values)) //208

208 bytes of memory.

We change range from 1000 to 100000,

values = (x \* 2 for x in range(100000))

print("gen:", getsizeof(values)) //208

Size remains same for generator object.

Instead of it , if we use a list comprehension object,

values = [x \* 2 for x in range(100000)]

print("list:", getsizeof(values)) //800984

So our list is taking over 800,000 bytes of memory.

So, Use a generator expression to for a generator object if you are dealing with a large or infinite stream of data*.* ***“****Just be aware that since generator objects do not store all items in memory, you would not be able to get the total number of items you are working with***”**.

values = (x \* 2 for x in range(100000))

print(len(values))

TypeError: object of type 'generator' has no len()

**Unpacking operator**:

Consider the below code

numbers = [1, 2, 3]

print(numbers) //[1, 2, 3]

We get a list of three items,

But what if we do not want to print a list , instead we want to get only the individual items like 1 2 3…

So we can use unpacking operator \* (just like *spread* operator **…** in JavaScript)

numbers = [1, 2, 3]

print(\*numbers) //1 2 3

The good thing about this **\*** unpacking operator is that, it can be applied over any iterable, it does not have to be a list. Like in below example you can apply it over range() object as well.

values = [\*range(5)]

print(values) // [0, 1, 2, 3, 4]

We can also unpack a string,

values = [\*"Hello World"]

print(values) // ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']

We can also add two lists using **\***,

first = [1, 2]

second = [3]

print([\*first, \*second]) // [1, 2, 3]

To unpack a dictionary we need two **\*\***,

first = {"x": 1}

second = {"x": 10, "y": 2}

print({\*\*first, \*\*second}) // {'x': 10, 'y': 2}

**Exercise**:

We have this text,

sentence = "This is a common interview question"

write a program to find the most repeated character in this text.

Solution:

from pprint import pprint

sentence = "This is a common interview question"

char\_frequency = {}

for char in sentence:

    if char in char\_frequency:

        char\_frequency[char] += 1

    else:

        char\_frequency[char] = 1

char\_frequency\_sorted = sorted(

    char\_frequency.items(), key=lambda kv: kv[1], reverse=True

)

print(char\_frequency\_sorted[0])

**Attempt it again**.

**Good read and Problems**:

🡪 What does yield keyword do (*related to generator object, for divide into array chunk problem*)?

<https://sentry.io/answers/python-yield-keyword/>

Write a function called *chunk* which divide a list into chunks and where we provide chunk size as argument:

[1, 2, 3, 4] => [[1, 2], [3, 4]]

<https://www.programiz.com/python-programming/examples/list-chunks>

🡪