**Lists**

Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index ``maps to'' one of the elements.

List indices work the same way as string indices: Any integer expression can be used as an index.

If you try to read or write an element that does not exist, you get an IndexError.

If an index has a negative value, it counts backward from the end of the list. The in operator also works on lists.

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

In this section we will learn about:

1.) Creating lists

2.) Indexing and Slicing Lists

3.) Basic List Methods

4.) Nesting Lists

5.) Introduction to List Comprehensions

Lists are constructed with brackets [] and commas separating every element in the list.

*# heterogenous*

*# ordered collection*

*# mutable*

​

​

lst**=**[1,"ASha",67.4,[23,5]]

print(lst[1])

​

​

​

lst **=** [1,2,3,4,5]

lst.append(10)

​

lst[0]**=** 10

​

​

​

​

​

lst **=**list()

​

​

​

dir(lst)

lst **=** [] *#empty list*

​

​

ASha

​

st **=** "Asha"

lst **=** list(st)

​

print(lst)

​

tu **=** (1,2,3,4)

lst **=** list(tu)

print(lst)

​

print(type(lst))

​

​

['A', 's', 'h', 'a']

[1, 2, 3, 4]

<class 'list'>

lst **=** [1,2,3,4,5]

​

**for** item **in** lst:

print(item)

range(0,10)

​

**for** item **in** range(10):

print(item)

list(range(10))

print(list(range(10)))

1

2

3

4

5

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

my\_list

Out[2]:

[1, 2, 3, 'hello']

We just created a list of integers, but lists can actually hold different object types.

my\_list **=** ['A string',23,100.232,'o']

print(my\_list)

['A string', 23, 100.232, 'o']

Just like strings, the len() function will tell you how many items are in the sequence of the list.

len(my\_list)

Out[4]:

4

**Indexing and Slicing**

Indexing and slicing work just like in strings.

my\_list **=** ['one','two','three',4,5]

print(my\_list[0])

print(my\_list[**-**2])

​

print(my\_list)

one

4

['one', 'two', 'three', 4, 5]

*# Grab element at index 0*

my\_list[**-**1:**-**3:**-**1]

Out[5]:

[5, 4]

*# Grab index 1 and everything past it*

my\_list[1:]

*# my\_list[startindex:endindex:increment]*

Out[7]:

['two', 'three', 4, 5]

*# Grab everything UP TO index 3*

my\_list[:3:1]

Out[8]:

['one', 'two', 'three']

We can also use + to concatenate lists, just like we did for strings.

print(my\_list **+** ['new item'])

print(my\_list)

['one', 'two', 'three', 4, 5, 'new item']

['one', 'two', 'three', 4, 5]

Note: This doesn't actually change the original list!

my\_list **=**[1,2,3,45,6]

print(my\_list[3]**\***2)

my\_list**\***2

90

Out[11]:

[1, 2, 3, 45, 6, 1, 2, 3, 45, 6]

your\_list **=** [1,2,3,4]

your\_list **=** your\_list **+**['new list']

your\_list

Out[8]:

[1, 2, 3, 4, 'new list', 'new list']

You would have to reassign the list to make the change permanent.

*# Reassign*

my\_list **=** my\_list **+** ['add new item permanently']

my\_list

Out[4]:

[1, 2, 3, 45, 6, 'add new item permanently']

We can also use the \* for a duplication method similar to strings:

*# Make the list double*

my\_list **\*** 3

Out[37]:

['one',

'two',

'three',

4,

5,

'one',

'two',

'three',

4,

5,

'one',

'two',

'three',

4,

5]

*# Again doubling not permanent*

my\_list

Out[16]:

[1, 2, 3]

**Basic List Methods**

If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).

*# Create a new list*

​

​

list1 **=** [1,2,3]

dir(list1)

print(list1)

​

​

nxt **=**list1.append([10,20])

*# nxt = list1.append([10,2])*

print(list1)

print(nxt)

​

​

list1.extend([30,2,5])

print(list1)

​

*# print(dir(list1))*

[1, 2, 3]

[1, 2, 3, [10, 20]]

None

[1, 2, 3, [10, 20], 30, 2, 5]

​

Use the **append** method to permanently add an item to the end of a list:

*# Append*

list1 **=**[1,2,3,4]

*# nxt = list1.append(['append me!',2])*

nxt **=** list1.append(10)

*# print(dir(list1))*

*# print(help(list1.append(1)))*

print(nxt)

print(list1)

None

[1, 2, 3, 4, 10]

*# Show*

list1

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

*# Pop off the 0 indexed item*

​

x **=** list1.pop(1)

print(x)

print(list1)

2

[1, 3, [10, 20], 30, 2, 5]

*# Show*

list1

Out[113]:

[2, 3, 'append me!']

*# Assign the popped element, remember default popped index is -1*

popped\_item **=** list1.pop()

print(popped\_item)

print(list1)

[10, 2]

[1, 2, 3]

popped\_item

Out[115]:

'append me!'

*# Show remaining list*

list1

Out[116]:

[2, 3]

It should also be noted that lists indexing will return an error if there is no element at that index. For example:

list1[100]

**---------------------------------------------------------------------------**

**IndexError** Traceback (most recent call last)

**<ipython-input-117-af6d2015fa1f>** in <module>

**----> 1** list1**[100]**

**IndexError**: list index out of range

We can use the **sort** method and the **reverse** methods to also effect your lists:

new\_list **=** ['a','e','x','b','c']

print(new\_list)

​

lst **=** new\_list.sort() *#ascending*

print(new\_list)

print(lst)

​

lst **=** new\_list.sort(reverse**=True**) *#sorts in the desecnding order ,in ascending but does not return any value , the updation happens on the object itself*

*# lst = new\_list.sort() #ascending*

print(new\_list)

print(lst)

​

​

​

​

['a', 'e', 'x', 'b', 'c']

['a', 'b', 'c', 'e', 'x']

None

['x', 'e', 'c', 'b', 'a']

None

*#Show*

new\_list

Out[27]:

['x', 'e', 'c', 'b', 'a']

*# Use reverse to reverse order (this is permanent!)*

x **=** new\_list.reverse()

print(x)

print(new\_list)

None

['a', 'b', 'c', 'e', 'x']

new\_list

Out[29]:

['a', 'b', 'c', 'e', 'x']

*# Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)*

new\_list.sort(reverse **=** **True**)

​

new\_list

Out[24]:

['x', 'e', 'c', 'b', 'a']

new\_list.sort()

new\_list

Out[27]:

['a', 'b', 'c', 'e', 'x']

str **=** "Asha"

print(str)

lst **=** list(str)

print(lst)

lst.reverse()

print(lst)

Asha

['A', 's', 'h', 'a']

['a', 'h', 's', 'A']

​

*# Creating Lists using range() function*

lst **=** [1,2,3,4]

*# range(startvalu,endvalue,increment)*

*# xrange - python 2*

*# range(0,10,1)*

*# range(4,9,2)*

*# int(2.5)*

list(range(10))

Out[17]:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

*# Aliasing - giving new name to the existing list , lets see how to create new list by aliasing, both point to same object*

lst **=**[1,2,3,4,5]

lst1 **=** lst

​

*# lst[1] ->[4,20,8]*

*# lst1[1]->*

​

​

lst1

Out[12]:

[1, 2, 3, 4, 5]

lst

Out[13]:

[1, 2, 3, 4, 5]

lst[1]**=**30

lst

Out[15]:

[1, 30, 3, 4, 5]

lst1

Out[16]:

[1, 30, 3, 4, 5]

*# Cloning -- copying or cloning will create exact copy of an exisitng ojbect and way to create it is*

y **=** lst[:]

print(y)

print(lst)

​

*# lst = [4,20,8]*

*# y= [4,20,8]*

​

print(id(lst)) *#will print the address of the object*

print(id(y))

​

[1, 30, 3, 4, 5]

[1, 30, 3, 4, 5]

89881256

89691240

y[2]**=**30

y

Out[23]:

[1, 30, 30, 4, 5]

lst

Out[20]:

[1, 30, 3, 4, 5]

*# min and max functions*

*# sorting in both ascending and descending*

n1 **=** max(lst)

n1

Out[58]:

20

n1 **=** min(lst)

n1

Out[60]:

4

lst.sort()

lst

Out[62]:

[4, 8, 20]

lst.sort(reverse **=** **True**)

lst

Out[64]:

[20, 8, 4]

*# number of occurances*

lst **=** [1,2,3,4]

lst.append([4,6])

lst.append(5)

lst.append([70,50]) *#will append like list instead of its values*

lst

Out[9]:

[1, 2, 3, 4, [4, 6], 5, [70, 50]]

lst[4][1]

Out[12]:

6

lst **=** [2,3,4]

lst.extend([10,30,40]) *#add many values as list and it should be one argument only*

lst

Out[21]:

[2, 3, 4, 10, 30, 40]

lst**+=**[50,60]

lst **=** lst**+**[50,60]

lst

*# lst =[]*

*# lst*

Out[31]:

[2, 3, 4, 10, 30, 40, 50, 60, 50, 60]

n **=** lst.count(4)

n

Out[33]:

1

**Lists and strings**

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

s **=** 'spam spam'

t **=** list(s)

print(t)

​

​

['s', 'p', 'a', 'm', ' ', 's', 'p', 'a', 'm']

*#The list function breaks a string into individual letters.*

*#If you want to break a string into words, you can use the split method:*

​

​

s **=** 'pining for the fjords'

​

t **=** s.split()

print(t)

​

print(t[2])

​

*#Once you have used split to break the string into a list of tokens,*

*#you can use the index operator (square bracket) to look at a particular word in the list.*

​

['pining', 'for', 'the', 'fjords']

the

*#You can call split with an optional argument called a delimiter specifies which characters to use as word boundaries.*

*#The following example uses a hyphen as a delimiter:*

​

​

s **=** 'spam-spam-spam'

delimiter **=** '-'

s.split(delimiter)

​

​

​

Out[48]:

['spam', 'spam', 'spam']

*#join is the inverse of split. It takes a list of strings and concatenates the elements.*

*#join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:*

​

​

t **=** ['pining', 'for', 'the', 'fjords']

*# delimiter = '\*'*

delimiter **=** ' ' *#here the joining is happening the character specified in the delimeter will be used for joining*

delimiter.join(t)

​

*#In this case the delimiter is a space character,*

*#so join puts a space between words. To concatenate strings without spaces, you can use the empty string, '', as a delimiter.*

​

Out[39]:

'pining for the fjords'

**Parsing lines**

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the interesting lines'' and then parse the line to find some interesting part of the line. What if we wanted to print out the day of the week from those lines that start withFrom ''.

From [stephen.marquard@uct.ac.za](mailto:stephen.marquard@uct.ac.za) Sat Jan 5 09:14:16 2008 The split method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with ``From '' and then split those lines and then print out the third word in the line:

fhand = open('mbox-short.txt') for line in fhand: line = line.rstrip() if not line.startswith('From ') : continue words = line.split() print(words[2]) We also use the contracted form of the if statement where we put the continue on the same line as the if. This contracted form of the if functions the same as if the continue were on the next line and indented.

Objects and values If we execute these assignment statements:

a = 'banana' b = 'banana' We know that a and b both refer to a string, but we don't know whether they refer to the same string. There are two possible states:

In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the is operator.

*# a = 'banana'*

*# print(id(a))*

*# b = 'banana'*

*# print(id(b))*

*# print(a is b)*

​

​

​

​

​

​

​

*# print(id(a))*

*# print(id(b))*

*# #In this example, Python only created one string object, and both a and b refer to it.*

​

*# x = {"name":"Asha","id":"51329042"}*

*# print(id(x))*

*# y = {"name":"Asha","id":"51329042"}*

*# print(x)*

*# print(id(y))*

*# x is y*

​

​

a **=** [1, 2, 3]

b **=** [1, 2, 3]

​

​

print(id(b))

print(id(a))

print(a **is** b)

​

*# if a == b:*

*# print("True")*

​

*# a = b #if you voluntrality make b point to a then both the objects will have same address and a is b will be true*

*# print(id(b))*

*# print(id(a))*

*# a is b*

​

*#Even though tuple is immutable , but it can have mutable elements , hence the address will be different*

*# a = (1, 2, 3,(4,5),[7,8])*

*# b = (1, 2, 3,(4,5),[7,8])*

​

*# print(id(a))*

​

*# print(id(b))*

*# a is b*

​

3984680

3984440

False

*#But when you create two lists, you get two objects:*

​

*# a = [1, 2, 3]*

*# b = [1, 2, 3]*

​

*# print(id(b))*

*# print(id(a))*

*# a is b*

​

*#In this case we would say that the two lists are equivalent,*

*#because they have the same elements, but not identical, because they are not the same object.*

*#If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.*

​

​

​

*#Until now, we have been using ``object'' and ``value'' interchangeably,*

*#but it is more precise to say that an object has a value.*

*#If you execute a = [1,2,3], a refers to a list object whose value is a particular sequence of elements.*

*#If another list has the same elements, we would say it has the same value.*

​

146825864

128817032

Out[47]:

False

Aliasing If a refers to an object and you assign b = a, then both variables refer to the same object:

a **=** [1, 2, 3]

b **=** a

b **is** a

​

Out[7]:

True

The association of a variable with an object is called a reference. In this example, there are two references to the same object. An object with more than one reference has more than one name, so we say that the object is aliased. If the aliased object is mutable, changes made with one alias affect the other:

b[0] **=** 17

print(a)

​

[17, 2, 3]

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

a = 'banana' b = 'banana' It almost never makes a difference whether a and b refer to the same string or not.

**List arguments**

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete\_head removes the first element from a list:

**def** delete\_head(t):

print(id(t))

**del** t[0]

print(t)

​

​

letters **=** ['a', 'b', 'c']

print(id(letters))

delete\_head(letters)

print(letters)

​

*#The parameter t and the variable letters are aliases for the same object.*

​

​

100758912

100758912

['b', 'c']

['b', 'c']

*#It is important to distinguish between operations that modify lists and operations that create new lists.*

*#For example, the append method modifies a list, but the + operator creates a new list:*

​

​

t1 **=** [1, 2]

t2 **=** t1.append(3)

print(t1)

​

print(t2)

​

*# print(id(t1))*

*# print(id(t2))*

​

​

t3 **=** t1 **+** [3]

print(t3)

t2 **is** t3

​

​

[1, 2, 3]

None

130641312

1515093060

[1, 2, 3, 3]

Out[8]:

False

This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

**def** bad\_delete\_head(t):

print(id(t))

*# t = t[1:]*

*# t.append(10)*

t **=** [4,5,6]

print(id(t))*# WRONG!*

*# del t[0]*

print(t)

*# print(t1)*

*#The slice operator creates a new list and the assignment makes t refer to it,*

*#but none of that has any effect on the list that was passed as an argument.*

​

letters **=**['a','b','c']

print(id(letters))

*# rest = bad\_delete\_head(letters)*

bad\_delete\_head(letters)

print(letters)

​

​

89946912

89946912

89913784

[4, 5, 6]

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

*#An alternative is to write a function that creates and returns a new list.*

*#For example, tail returns all but the first element of a list:*

​

**def** tail(t):

**return** t[1:]

*#This function leaves the original list unmodified. Here's how it is used:*

​

letters **=** ['a', 'b', 'c']

rest **=** tail(letters)

print(rest)

print(letters)

​

['b', 'c']

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

**Nesting Lists**

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

*# Let's make three lists*

lst\_1**=**[1,2,3]

lst\_2**=**[4,5,6]

lst\_3**=**[7,8,9]

​

*# Make a list of lists to form a matrix*

matrix **=** [lst\_1,lst\_2,lst\_3]

matrix**=**[

[1,2,3],

[4,5,6],

[7,8,9]

]

*# Show*

matrix[2][2]

Out[8]:

9

We can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

*# Grab first item in matrix object*

matrix[0]

Out[9]:

[1, 2, 3]

*# Grab first item of the first item in the matrix object*

matrix[0][0]

Out[10]:

1

lst **=** [10,20,30,[80,90]]

lst[3]

Out[13]:

[80, 90]

lst[3][1]

Out[14]:

90

**Debugging**

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them: Don't forget that most list methods modify the argument and return None. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

word = word.strip() It is tempting to write list code like this:

t = t.sort() # WRONG! Because sort returns None, the next operation you perform with t is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at docs.python.org/lib/typesseq.html. The methods and operators that only apply to mutable sequences are documented at docs.python.org/lib/typesseq-mutable.html.

Pick an idiom and stick with it.

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use pop, remove, del, or even a slice assignment.

To add an element, you can use the append method or the + operator. But don't forget that these are right:

t.append(x) t = t + [x]

And these are wrong:

t.append([x]) # WRONG! t = t.append(x) # WRONG! t + [x] # WRONG! t = t + x # WRONG! Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

Make copies to avoid aliasing.

If you want to use a method like sort that modifies the argument, but you need to keep the original list as well, you can make a copy.

orig = t[:] t.sort() In this example you could also use the built-in function sorted, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using sorted as a variable name!

Lists, split, and files

When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the guardian pattern when it comes writing programs that read through a file and look for a ``needle in the haystack''.

Let's revisit our program that is looking for the day of the week on the from lines of our file:

From [stephen.marquard@uct.ac.za](mailto:stephen.marquard@uct.ac.za) Sat Jan 5 09:14:16 2008 Since we are breaking this line into words, we could dispense with the use of startswith and simply look at the first word of the line to determine if we are interested in the line at all. We can use continue to skip lines that don't have ``From'' as the first word as follows:

fhand = open('mbox-short.txt') for line in fhand: words = line.split() if words[0] != 'From' : continue print( words[2]) This looks much simpler and we don't even need to do the rstrip to remove the newline at the end of the file. But is it better?

python search8.py Sat Traceback (most recent call last): File "search8.py", line 5, in if words[0] != 'From' : continue IndexError: list index out of range It kind of works and we see the day from the first line (Sat) but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a print statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable words right before line five. We even add a prefix ``Debug:'' to the line so we can keep our regular output separate from our debug output.

for line in fhand: words = line.split() print ('Debug:', words) if words[0] != 'From' : continue print( words[2]) When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

Debug: ['X-DSPAM-Confidence:', '0.8475'] Debug: ['X-DSPAM-Probability:', '0.0000'] Debug: [] Traceback (most recent call last): File "search9.py", line 6, in if words[0] != 'From' : continue IndexError: list index out of range Each debug line is printing the list of words which we get when we split the line into words. When the program fails the list of words is empty []. If we open the file in a text editor and look at the file, at that point it looks as follows:

X-DSPAM-Result: Innocent X-DSPAM-Processed: Sat Jan 5 09:14:16 2008 X-DSPAM-Confidence: 0.8475 X-DSPAM-Probability: 0.0000

The error occurs when our program encounters a blank line! Of course there are zero words'' on a blank line. Why didn't we think of that when we were writing the code. When the code looks for the first word (word[0]) to check to see if it matchesFrom'', we get an ``index out of range'' error.

This of course is the perfect place to add some guardian code to avoid checking the first word if the first word is not there. There are many ways to protect this code, we will choose to check the number of words we have before we look at the first word:

fhand = open('mbox-short.txt') count = 0 for line in fhand: words = line.split()

# print ('Debug:', words)

if len(words) == 0 : continue

if words[0] != 'From' : continue

print( words[2])

First we commented out the debug print statement instead of removing it in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use continue to skip to the next line in the file.

We can think of the two continue statements as helping us refine the set of lines which are interesting'' to us and which we want to process some more. A line which has no words isuninteresting'' to us so we skip to the next line. A line which does not have ``From'' as its first word is uninteresting to us so we skip it.

The program as modified runs successfully so perhaps it is correct. Our guardian statement does make sure that the words[0] will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, ``What might go wrong?''.

employee **=** [

['John' , 38 ,'Sales'],

['Lisa',29,'Marketing'],

['Srujan', 33,'HR']

]

print((employee))

print(len(employee)) *#3 objects so len is 3*

**for** row **in** range(len(employee)):

**for** col **in** range(len(employee[row])):

print(employee[row][col])

print('-'**\***30)

print("Name: ",employee[1][0])

print("Age :",employee[1][1])

print("Department :",employee[1][2])

print('-'**\***30)

[['John', 38, 'Sales'], ['Lisa', 29, 'Marketing'], ['Srujan', 33, 'HR']]

3

John

38

Sales

Lisa

29

Marketing

Srujan

33

HR

------------------------------

Name: Lisa

Age : 29

Department : Marketing

------------------------------

employee **=** [

['John' , 38 ,'Sales'],

['Lisa',29,'Marketing'],

['Srujan', 33,'HR']

]

*# print(employee)*

​

**for** employees **in** employee:

print(employees)

​

**for** employees **in** employee:

print('-'**\***30)

print('Name:',employees[0])

print('Age',employees[1])

print('Department:',employees[2])

['John', 38, 'Sales']

['Lisa', 29, 'Marketing']

['Srujan', 33, 'HR']

------------------------------

Name: John

Age 38

Department: Sales

------------------------------

Name: Lisa

Age 29

Department: Marketing

------------------------------

Name: Srujan

Age 33

Department: HR

Type *Markdown* and LaTeX: 𝛼2α2

X **=** [[1,2,3],

[4,5,6],

[7,8,9]

]

Y**=** [[10,11,12],

[13,14,15],

[16,17,18]

]

​

*# result = [[0,0,0],*

*# [0,0,0],*

*# [0,0,0]*

*# ]*

​

**for** i **in** range(len(X)):

**for** j **in** range(len(X[0])):

result[i][j] **=** X[i][j] **+** Y[i][j]

**for** x **in** result:

print(x)

[11, 13, 15]

[17, 19, 21]

[23, 25, 27]

**for** j **in** Y:

print(j)

[10, 11, 12]

[13, 14, 15]

[16, 17, 18]

l1 **=** [1,2,3,4]

print(l1[1:3])

print(l1[**-**1:**-**3:**-**1])

print(l1[::**-**1])

print(l1[**-**1::**-**1])

[2, 3]

[4, 3]

[4, 3, 2, 1]

[4, 3, 2, 1]

print(l1[3:1])

print(l1[3:1:**-**1])

[]

[4, 3]

print(l1[2:1])

print(l1[2:1:**-**1])

print(l1)

[]

[3]

[1, 2, 3, 4]