Lists-Chapter 8

Each element in a list can be a different type such as strings, integers, floats, or even other lists.

my\_list = [‘hello’, -4.2, 5] CLI

The user creates a new list

my\_list = [‘hello’, -4.2, 5] Python interpreter

interpreter creates a new object for each list element

Diagram

Description automatically generated

my\_list holds references to objects in the list

Old person list: You have a list of the oldest ages, and you want user input to select how old is nth oldest person. you want to print out the results with “the nth oldest person is {oldest\_person}, based on an int input from the user.

Problem: for numbers ending in 2 or 3, the suffix is nd and rd, respectively.

Also, what if the list is 100 or 1,000 numbers long?

Start with:

oldest\_people = [] #if you want to create a list of random numbers from i to j you will assign oldest\_people to a new variable:

def num\_list():

num\_ls = []

for i in range(105,120)

num\_ls.append(i)

return num\_ls

print(num\_list())

#creates list [105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119]

\*if you wanted to create random list of ages from 105 -120 with repeats and then sort from highest to lowest

import random

num\_list = []

for i in range(35):

#if you write num\_list.append(105,120), it won't work.

#.append only with one argument

#but if you write num\_list.append(random.randint(1,100)), it will work.

num\_list.append(random.randint(105, 120))

num\_list.sort(reverse =True) #will list from biggest to smallest

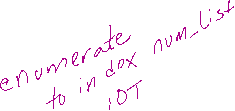
print(num\_list)

#Oldest person List:

#user inputs a number to id age from list

nth\_person = int(input('Enter Number: '))

len\_person = len(num\_list())



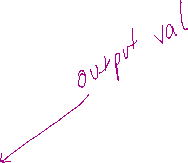
# Check if nth\_person ends with a 2 or 3



for i, val in enumerate(num\_list()):



if nth\_person % 10 == 2:



print(f'The {nth\_person}nd oldest person was {val} years old.')

elif nth\_person % 10 == 3:

print(f'The {nth\_person}rd oldest person lived was {val} years old')



elif (nth\_person == 1) or (nth\_person > 3):

print(f'The {nth\_person}th oldest person lived was {val} years old ')

else:

print(f'There are only {len\_person} people in this list')

* when input is 21
* The 21th oldest person lived was 106 years old



**List Indices**

animals = ['cat', 'dog', 'bird', 'raptor']

i = 3

print(animals[i])

* **raptor**

print(animals[i-1])

* **bird**

**Ways to create a list:**

# A list with 3 integers

numbers = [1, 2, 5]

# empty list

my\_list = []

# my\_list = list(‘123’)

print(my\_list)

# list with mixed data types

my\_list = [1, "Hello", 3.4]

#Concatenate

my\_list = [1,2] + [3] + ['ducky']

print(my\_list)

* [1, 2, 3, 'ducky']

**Modify a List**

Change in Place:

my\_list = list('123456')

my\_list[5] = 'ducky'

print(my\_list)

* ['1', '2', '3', '4', '5', 'ducky']

my\_list = "Stafford"

new\_list = ([\*my\_list])

print(new\_list)

* ['S', 't', 'a', 'f', 'f', 'o', 'r', 'd']

my\_list = list('123456')

my\_list[len(my\_list):] = [9]



#notice the (my\_list): indicates everything after len(my\_list)

#this in contrast to Change in Place method

print(my\_list)

* ['1', '2', '3', '4', '5', '6', 9]

**Delete from list using: del my\_list[i]**

my\_list = list('123456')

del my\_list[0]

print(my\_list)

* ['2', '3', '4', '5', '6']

**Change in Place Considerations:**

my\_teams = ['Raptors', 'Heat', 'Nets']

your\_teams = my\_teams # Create a shared reference to same list, so if

# you change one you change the other

my\_teams[1] = 'Lakers' #this would change both my\_teams & your\_teams

instead to create an original variable use:

your\_teams = my\_teams[:]

Careful:

colors = ['red', 'green', 'blue']

colors[1] = 'yellow' *# Changes list element*

fav\_color = colors[2] #doesn’t change list element

However, this is okay:

user\_values = [3, 6, 7]

user\_values[1] = user\_values[1] + 1

print(user\_values) => [3, 7, 7]

**Adding to an index:**



**user\_values = [2, 6, 8]**



**user\_values[2] = user\_values[2] + 1**

**user\_values[1] = user\_values[1] + 2**



**[2, 8, 9]**



Append, Extend, Insert

**Append method -** Add an item to the end of list.

**my\_list = [5, 8]**

**my\_list.append(16)**

* [5, 8, 16]

my\_list = list('123456')

my\_list.append(list('789'))

print(my\_list)

print(my\_list[6])

print(my\_list[6][1]) #an index of an index

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

**Extend method -** Add all items in [x] to list.

**my\_list = [5, 8]**

**my\_list.extend([4, 12])**

* [5, 8, 4, 12]

my\_string = "Stafford"

new\_string = '123'

a\_list = []

b\_list = []

a\_list.extend(my\_string)

b\_list.extend(new\_string)

print(a\_list)

print(b\_list)

* ['S', 't', 'a', 'f', 'f', 'o', 'r', 'd']
* ['1', '2', '3']

**Insert method -** Insert x into list before position i.

my\_list = [5, 8]

my\_list.insert(1, 1.7)

* [5, 1.7, 8]

Remove, list.pop(), list.pop(i)

**.remove method**

my\_list = [5, 8, 14]

my\_list.remove(8)

* [5, 14]

**.pop() method -** Remove and return last item in list.

my\_list = [5, 8, 14]

val = my\_list.pop()

* [5, 8] #14 removed

**.pop(i) method -** Remove and return item at position i in list.

my\_list = [5, 8, 14]  
val = my\_list.pop(0)

* [8, 14]

**Modifying elements: Sort/Reverse**

**.sort method -** Sort the items of list in-place.

my\_list = [14, 5, 8]

my\_list.sort()

* [5, 8, 14]

[**sorted()**](https://docs.python.org/3/library/functions.html#sorted) **method -** It returns a new sorted list

sorted([5, 2, 3, 1, 4])

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

You can use sorted() with a dictionary list

sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})

print(sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'}))

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

result = sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})

print(\*result)

* 'A' 'B' 'B' 'D' 'E'

sorted("This is a test string from Andrew".split(), key=str.lower)

If you wanted the reverse:

original\_string = "This is a test string from Andrew".split()

sorted\_output = sorted(original\_string, key=str.lower) #key=str.lower is an argument of sorted(). It sorts by lowercase, but outputs in the original case

print(sorted\_output)

reversed\_output = reversed(sorted\_output) # Reverse the output

print(\*reversed\_output) # Print reversed output

* ['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
* This test string is from Andrew a

**list.index(x) -** Return index of first item in list with value x.

my\_list = [5, 8, 14]

print(my\_list.index(14))

* 2 #indicates the indexed location

**list.count(x) -** Count the number of times value x is in list.

my\_list = [5, 8, 5, 5, 14]

print(my\_list.count(5))

* 3 #prints the number of occurrences in list

**Split -** Split a string into a list where each word is a list item:

user\_input = input()

short\_names = user\_input.split()

txt = "welcome to the jungle"

x = txt.split()

print(x)

* ['welcome', 'to', 'the', 'jungle']

**string.split (separator, maxsplit)**

separator- Specifies the separator to use when splitting the string. default is any whitespace

maxsplit - Optional. Specifies how many splits to do (-1 is default, all occurrences)

txt = "apple#banana#cherry#orange"

x = txt.split("#")

print(x)

* ['apple', 'banana', 'cherry', 'orange']

**Similar approach:**

my\_list = "Stafford"

new\_list = ([\*my\_list])

print(new\_list)

* ['S', 't', 'a', 'f', 'f', 'o', 'r', 'd']

**Using map( ) objects -** pass in a function and an iterable, and map() will create an object. This object contains the output you would get from running each iterable element through the supplied function.

#map() objects pass in a function as an argument and an iterable object #as an argument.

txns = [1.09, 23.56, 57.84, 4.56, 6.78]

TAX\_RATE = .08

def get\_price\_with\_tax(txn):

    return txn \* (1 + TAX\_RATE)

#if you want a rounded tax rate, you can use the following line:

    #return round(txn \* (1 + TAX\_RATE), 2)

final\_prices = map(get\_price\_with\_tax, txns)

list(final\_prices)

[1.1772000000000002, 25.4448, 62.467200000000005, 4.9248, 7.322400000000001]

vs. (with rounded)

[1.17, 25.7, 62.39, 4.91, 7.32]

List Comprehensions: One main benefit of using a list comprehension in Python is that it’s a single tool that you can use in many different situations. In addition to standard list creation, list comprehensions can also be used for mapping and filtering. includes 3 features:

**List Comprehension Example:**

squares = [i \* i for i in range(10)]

squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

1. expression: i\*\*2, or 2%10
2. members: i is the member
3. iterable: is a list, set, sequence, generator, or any other object that can return its elements one at a time. In this example, range(10)

**More about List Comprehensions:**

List comprehensions are also more **declarative** than loops, which means they’re easier to read and understand. Loops require you to focus on how the list is created. You have to manually create an empty list, loop over the elements, and add each of them to the end of the list. With a list comprehension in Python, you can instead focus on what you want to go in the list and trust that Python will take care of how the list construction takes place.

Using conditionals with List Comprehensions:

sentence = 'the rocket came back from mars'

vowels = [i for i in sentence if i in 'aeiou']

vowels

['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']

new\_list = [expression for member in iterable (if conditional)]

**Nested Lists -** Since a list can contain any type of object as an element, and a list is itself an object, a list can contain another list as an element. Such embedding of a list inside another list is known as ***list nesting***.

my\_list = [[5, 13], [50, 75, 100]]

or

my\_list = [

[10, 0, 55],

[0, 4, 16]

]

*# Write to some elements*

my\_list[0][0] = 33

my\_list[1][1] = 77

my\_list[1][2] = 99

Problem:

Print the two-dimensional list mult\_table by row and column.

On each line, each character is separated by a space. Hint: Use nested loops.

Sample output with input: '1 2 3,2 4 6,3 6 9':

1 | 2 | 3

2 | 4 | 6

3 | 6 | 9

# print(str(cell) + (" | " if not row.index(cell) == (len(row) - 1) else ""), end="")

for row in mult\_table:

for col in row[:-1]:

# print character separated by spaces, with a pipe after each one

print(str(col) + ' | ', end='')

# print the last character without a pipe

print(row[-1])

This code first converts the string the user inputs into a two-dimensional list (a list of

lists). Then, it loops through each row and column of the list and prints them out with

spaces between each number, placing a pipe "|" after each one. Finally, it prints the last

character in each row without a pipe.

user\_input= input()

lines = user\_input.split(',')

# This line uses a construct called a list comprehension, introduced elsewhere,

# to convert the input string into a two-dimensional list.

# Ex: 1 2, 2 4 is converted to [ [1, 2], [2, 4] ]

mult\_table = [[int(num) for num in line.split()] for line in lines]

for row in mult\_table:

for cell in row:

print(str(cell) + (" | " if not row.index(cell) == (len(row) - 1) else ""), end="")

print("")

**List Slicing** - read multiple elements from a list, creating a new list that contains only the desired elements.

my\_list[start:end] - Get a list from start to end (minus 1).

my\_list = [5, 10, 20]

print(my\_list[0:2])

Step Slicing: Step slicing allows us to take every nth element from a given list. The syntax for step slicing looks like this: my\_list[start:end:step].

my\_list = [13, 14, 15, 16, 17, 18, 19]

new\_list = my\_list[0:6:2]

print(new\_list)

* [13, 15, 17] starts at 13

In our example, the start is 0 (the first element in the list), the end is 6 (the seventh element in the list) and the step is 2 (which means we want to take every second element in the list). This is why the final output for new\_list is [13, 15, 17].

Doesn’t print out 19 because 19 is the 7th element of the list.

**Loops Modifying Lists -** When a program iterates over a list while modifying the elements, such as changing some elements' values or moving elements' positions.

my\_list = [3.2, 5.0, 16.5, 12.25]

for i in range(len(my\_list)):

    my\_list[ i ] += 5

print(my\_list)

* [8.2, 10.0, 21.5, 17.25]

\*adds 5 to each element of the list as it iterates through

**List comprehensions:** - A programmer modifies every element of a list in the same way, such as adding 10 to every element. The Python language provides a convenient construct, known as ***list comprehension***, that iterates over a list, modifies each element, and returns a new list of the modified elements.

[A list comprehension is always surrounded by brackets]

A list comprehension has three components:

1. An *expression component* to evaluate for each element in the iterable object.
2. A *loop variable component* to bind to the current iteration element.
3. An *iterable object component* to iterate over (list, string, tuple, enumerate, etc).

my\_list = [10, 20, 30]

list\_plus\_5 = [(i + 5) for i in my\_list]

print(f'New list contains: {list\_plus\_5}')

my\_list = [50, 23, -4]

my\_list\_minus10 = [(i - 10) for i in my\_list]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Num | Description | For loop | Equivalent list comprehension | Output of both programs |
| 1 | Add 10 to every element. | my\_list = [5, 20, 50]  for i in range(len(my\_list)):  my\_list[ i ] += 10  print(my\_list) | my\_list = [5, 20, 50]  my\_list = [(i+10) for i in my\_list]  print(my\_list) | [15, 30, 60] |
| 2 | Convert every element to a string. | my\_list = [5, 20, 50]  for i in range(len(my\_list)):  my\_list[ i ] = str(my\_list[ i ])  print(my\_list) | my\_list = [5, 20, 50]  my\_list = [str(i) for i in my\_list]  print(my\_list) | ['5', '20', '50'] |
| 3 | Convert user input to a list of integers. | inp = input('Enter numbers:')  my\_list = []  for i in inp.split():  my\_list.append(int(i))  print(my\_list) | inp = input('Enter numbers:')  my\_list = [int(i) for i in inp.split()]  print(my\_list) | Enter numbers: 7 9 3 [7, 9, 3] |
| 4 | Find the sum of each row in a two-dimensional list. | my\_list = [[5, 10, 15], [2, 3, 16], [100]]  sum\_list = []  for row in my\_list:  sum\_list.append(sum(row))  print(sum\_list) | my\_list = [[5, 10, 15], [2, 3, 16], [100]]  sum\_list = [sum(row) for row in my\_list]  print(sum\_list) | [30, 21, 100] |
| 5 | Find the sum of the row with the smallest sum in a two-dimensional table. | my\_list = [[5, 10, 15], [2, 3, 16], [100]]  sum\_list = []  for row in my\_list:  sum\_list.append(sum(row))  min\_row = min(sum\_list)  print(min\_row) | my\_list = [[5, 10, 15], [2, 3, 16], [100]]  min\_row = min([sum(row) for row in my\_list])  print(min\_row) | 21 |

**Command Line for Python**

python myprog.py myfile1.txt

For a command line of python myprog.py myfile1.txt, argv has the contents ['myprog.py', 'myfile1.txt'].

* Whitespace separates arguments.
* User text is stored in sys.argv list.

import sys

name = sys.argv[1]

age = int(sys.argv[2])

print(f'Hello {name}.')

print(f'{age} is a great age.\n')

> python myprog.py Tricia 12

Hello Tricia.

12 is a great age.

> python myprog.py Aisha 30

Hello Aisha.

30 is a great age.

> python myprog.py Franco

Traceback (most recent call last):

File "myprog.py", line 4, in <module>

age = sys.argv[2]

IndexError: list index out of range

> python myprog.exe Tricia 12

Hello Tricia. 12 is a great age.

> python myprog.py Franco

Usage: python myprog.py name age

> python myprog.py Alan 70 pizza

Usage: python myprog.py name age

import sys

if len(sys.argv) != 3:

print('Usage: python myprog.py name age\n')

sys.exit(1) # Exit the program, indicating an error with 1.

name = sys.argv[1]

age = int(sys.argv[2])

print(f'Hello {name}. ')

print(f'{age} is a great age.\n')

Command-line arguments can become quite complicated for large programs with many options. There are entire modules of the standard library dedicated to aiding a programmer develop sophisticated argument parsing strategies. The reader is encouraged to explore modules such as argparse and getopt.

* [argparse:](http://docs.python.org/3/library/argparse.html#module-argparse)Parser for command-line options, arguments, and sub-commands
* [getopt:](http://docs.python.org/3/library/getopt.html)C-style parser for command-line options

**Dictionaries -** A dictionary is another type of container object that is different from sequences such as strings, tuples, and lists.

A Python dictionary associates keys with values.

dictionary comprehension

# **Iterating over a dictionary**

go back and review 8.14 - this seems important

Figure 8.15.1: Nested dictionaries.