Computer Programming in Python

Chapter 8

Strings, Lists, Dictionaries, and Sets

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - Used extensively in computer programs
  - Python provides many ways to examine and manipulate strings
    - Including the ability to examine the individual characters in a string
      - Consider a program that validates a password to ensure that it contains specific characters
      - Each character of the password needs to be visited and checked to determine if it meets one of the requirements.

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - The ***for-in*** loop can be used to walk the string one character at a time
    - It places a copy of the character in a variable that can be used in statements within the loop

```
temp = 'something'

for char in temp:
    print(char)
```

```
s
o
m
e
t
h
i
n
g
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

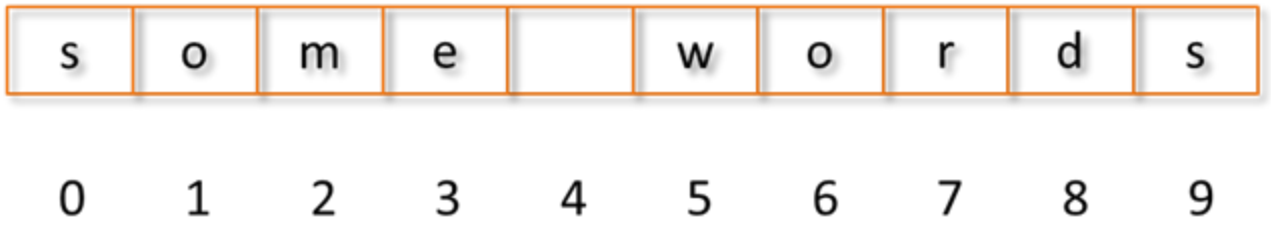
- Strings
  - The ***for-in*** loop copies the characters, so any changes to the character do not affect the original string

```
temp = 'the'

for char in temp:
    if char == 't':
        char = 's'
        print(char, end='')
    else:
        print(char, end='')

print('\n' + temp)
```

```
she
the
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - String characters can also be accessed using the ***index*** of the character
  - The index is the position in the string beginning at zero



String character indexes begin at zero

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - To access the character using the index, the index is placed in square brackets

```
my_string[index]
```

```
a_string = 'something'
print('Index zero is ', a_string[0])
```

```
Index zero is  s
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - Any valid index can be used

```
a_string = 'something'
print (a_string[0], a_string[3], a_string[7])
```

```
s e n
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - Negative indexes access character positions relative to the last character in the string
  - The index -1 is the last character in the string
    - Negative numbers work backward from there

```
b_string = 'negative'
print (b_string[-1], b_string[-4], b_string[-6])
```

```
e t g
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - An ***IndexError*** exception will be thrown if an index is out of range
  - The ***len*** function, which returns the length of the string, can be used as a way of controlling loops to prevent errors

initialized to to zero

incremented

```
temp = 'theater tickets'
index = 0

while index < len(temp):
    print(temp[index], end='')
    index = index + 1
```

```
theater tickets
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - Recall that strings in Python are immutable, and cannot be changed
  - The '+' operator will concatenate strings
    - This actually creates a new string and assigns it to the variable name for the original string
    - The original string can no longer be used because there is no longer a variable referencing it
    - Eventually, the Python interpreter will remove the original string from memory

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - When the concatenation occurs, a new string is created and Python assigns the variables name to the new string

```
city_string = 'New'
city_string = city_string + ' York'  New York

part1 = 'New'     New York
part2 = ' York'

part1 = part1 + part2
print(part1)
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Strings
  - A third string can also be created by concatenating two other strings

```
part1 = 'San'
part2 = ' Diego'

part3 = part1 + part2
print(part3)
```

```
San Diego
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- String Slicing
  - String ***slicing*** is used to select a portion of a string
    - Obtain a substring
  - There are optional start, end, and step specifiers
  - When the first specifier is omitted, Python uses zero as the start and the specifier as the end (or limit) which is not included in the slice

```
my_string[:end]
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- String Slicing
  - When two specifiers are used, the first is the start index and the second specifier indexes the end of the slice and is not included in the slice

```
my_string[start:end]
```

  - When three specifiers are used, the third is the step in the sequence

```
my_string[start:end:step]
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- String Slicing
  - Slicing example

```
sequence = '123456789'

first_four = sequence[:4]
print(first_four, end='')
print()

second_four = sequence[5:9]
print(second_four, end='')
print()

every_other = sequence[0:9:2]
print(every_other, end='')
```

```
1234
6789
13579
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- String Searching
  - Can use the ***in*** and ***not in*** operators to search strings
  - Example searching for the word “time” in the string with ***in***

```
phrase = 'A stitch in time saves nine.'
search_word = 'time'

if search_word in phrase:
    print('Found it.')
else:
    print('Not found.')
```

```
Found it.
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- String Searching
  - Reversed logic
  - Example searching using ***not in***

```
phrase = 'A stitch in time saves nine.'
search_word = 'time'

if search_word not in phrase:
    print('Not found.')
else:
    print('Found it.')
```

```
Found it.
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- String Testing Methods
  - Return true or false (Boolean), and test each character

Method	Description
isalnum()	True if the string contains only alphabetic letters or digits
isalpha()	True if the string contains only alphabetic letters
isdigit()	True if the string contains only numeric digits
islower()	True if the string contains only lower case alphabetic letters
isspace()	True if the string contains only white space characters
isupper()	True if the string contains only uppercase alphabetic letters

Chapter 8 Strings, Lists, Dictionaries, and Sets

- String Methods
  - Modification Methods include:
    - Conversion to upper and lower case, and various strip methods: lower(), upper(), lstrip(), rstrip(), and strip(*char*)
  - Search and Replace Methods include:
    - endswith(*substring*), find(*substring*), replace(*old, new*), and startswith(*substring*),

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Sequences of data that are mutable (can be changed), dynamic (can grow and shrink), and can be sliced
  - Can hold different types of data
  - Can be accessed using an index
    - Begin at zero

```
list = [ ]
```

declares an empty list

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Lists are initialized using the assignment operator and enclosing the members of the list in brackets

```
numbers = [5, 15, 25, 35]          # numbers
words = ['the', 'and', 'why']      # strings
mixed = ['first', 105, 15.6]       # strings and numbers
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists - Accessing List Elements
  - The first statement below assigns a list of numbers to num_list
  - Notice in the output that the first print statement displays the list surrounded by square brackets

```
num_list = [5, 15, 25, 35]
print(num_list)

for n in num_list:
    print(n, end=' ')

print()
print(num_list[2])
```

```
[5, 15, 25, 35]
5 15 25 35
25
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists - Accessing List Elements
  - The second set of statements use a for-in loop to access each element in the list
  - The last statement accesses a list element using an index

```
num_list = [5, 15, 25, 35]
print(num_list)

for n in num_list:
    print(n, end=' ')

print()
print(num_list[2])
```

```
[5, 15, 25, 35]
5 15 25 35
25
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists - Accessing List Elements
  - The len function can be used to control a loop
  - In this example, the loop counter *index* is incremented to control the loop, and is used as the index for accessing the list elements

```
word_list = ['one', 'two', 'three']
index = 0
while index < len(word_list):
    print(word_list[index])
    index = index + 1
```

```
one
two
three
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Built in functions and methods
    - Add elements
    - Insert elements
    - Remove elements
    - Change the order of the list
    - Find the minimum and maximum values in a list

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - To ***append*** an item to the end of a list
    - Include the name of the list, the dot operator, the append function, and the element to be added in parentheses

```
num_list = [5, 15, 25, 35]
num_list.append(45)
print(num_list)
```

```
[5, 15, 25, 35, 45]
```

Appends to the end of the list

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - To ***insert*** an item into a list
    - Include the name of the list, the dot operator, the insert function, the index where the element is to be inserted, and the element to be inserted

```
num_list = [5, 15, 25, 35]
num_list.insert(2, 45)
print(num_list)
```

```
[5, 15, 45, 25, 35]
```

Insert moves other elements toward the end of the list

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - To ***remove*** an item from a list
    - Include the name of the list, the dot operator, the remove function, and the element to be removed in parentheses
    - Elements beyond the element removed are shifted toward the front of the list

```
num_list = [5, 15, 25, 35]
num_list.remove(25)
print(num_list)
```

```
[5, 15, 35]
```

The element must be in the list or an exception is raised

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - To ***reverse*** a list
    - Include the name of the list, the dot operator, and the reverse function

```
num_list = [5, 15, 25, 35]
num_list.reverse()
print(num_list)
```

```
[35, 25, 15, 5]
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - To ***sort*** a list
    - Include the name of the list, the dot operator, and the reverse function

```
cities = ['Boston', 'Caldon', 'Albany']
cities.sort()
print(cities)
```

```
['Albany', 'Boston', 'Caldon']
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - To find the minimum or maximum value in a list, the list is passed to the ***min*** and ***max*** functions

```
numbers = [15, 3, 106, 27]
print(min(numbers))
print(max(numbers))
```

```
3
106
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Elements in a list can be changed using the index of the element
  - There is also an ***index()*** function that can be used to find the index for a specific element
    - But it will raise an exception if the element is not in the list
    - Determine first if the item is in the list using the '***in***' operator

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Determine first if the item is in the list using the '***in***' operator

```
numbers = [1, 2, 3, 4, 5]

if 3 in numbers:
    pos = numbers.index(3)
    numbers[pos] = 99

print(numbers)
```

```
[1, 2, 99, 4, 5]
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Lists can be ***concatenated*** using the '+' operator to combine two lists

```
list1 = ['a', 'c', 'e', 'g']
list2 = ['b', 'd', 'f', 'h']

list1 = list1 + list2
print(list1)
list1.sort()
print(list1)
```

```
['a', 'c', 'e', 'g', 'b', 'd', 'f', 'h']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Lists can be copied, but not using the assignment operator
    - Assigning one list to another would simply have both list names reference the same list

```
new_list = old_list      # referencing the same list
```

The assignment operator does not copy the list

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - To ***copy*** a list, define an empty list, and append each element in the first list to the new list
    - Can also concatenate the old list onto the new empty list

```
old_list = [12, 22, 32]
new_list = []

for element in old_list:
    new_list.append(element)
```

Copy the individual elements to copy a list

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - The ***Split*** method by default uses the space as a separator and returns a list of items in a string

```
time_string = 'hour minute second'
time_list = my_string.split()
print(time_list[1])
```

```
minute
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - A different separator can be specified for split
    - Including “/” when a date is being parsed

```
time_string = '10:23:59'
time_list = time_string.split(':')
print(time_list[1])
```

```
23
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Lists can be passed to functions

```
def main():
    num_list = [5, 15, 25, 35]
    print('The sum is :', get_sum(num_list))


def get_sum(in_list):
    vals = 0
    for num in in_list:
        vals = vals + num

    return vals

main()
```

```
The sum is : 80
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Functions can return lists

```
def main():
    my_list = get_list()

    print('The list is :', my_list)


def get_list():
    new_list = [1, 2, 3, 4, 5]

    return new_list

main()
```

```
The list is : [1, 2, 3, 4, 5]
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

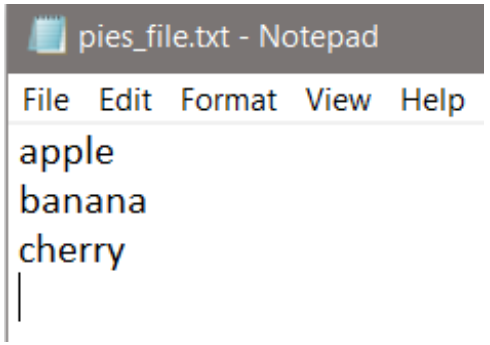
- Lists
  - Lists can be written to files with ***writelines****(list_name)*
    - But there are no line feeds with this method
    - To include line feeds, a loop is needed and the newline character needs to be added
      - A tab or a space could be added the same way and used as a delimiter when reading

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Lists
  - Adding line feeds when writing a list to a file

```
def main():
    pies = ['apple', 'banana', 'cherry']

    out_file = open('pies_file.txt', 'w')

    for pie_type in pies:
        out_file.write(pie_type + '\n')

    out_file.close()

main()
```

line feed



Chapter 8 Strings, Lists, Dictionaries, and Sets

- A line can be read into a list from a file

```
def main():
    input_file = open('pies_file.txt', 'r')

    pie_list = input_file.readlines()

    input_file.close()

    print(pie_list)

    count = 0
    while count < len(pie_list):
        pie_list[count] = pie_list[count].rstrip('\n')
        count = count + 1

    print(pie_list)

main()
```

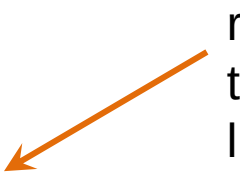
remove the line feed

```
['apple\n', 'banana\n', 'cherry\n']
['apple', 'banana', 'cherry']
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Reading into a List using Append

```
def main():
    input_file = open('pies_file.txt', 'r')

    pie_list = []

    for line in input_file:
        pie_list.append(line.rstrip('\n'))

    input_file.close()

    print(pie_list)

main()
```




```
['apple', 'banana', 'cherry']
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Two-dimensional Lists
  - A list of lists has rows and columns
    - Both indexes begin at zero

values[0][0]	values[0][1]	values[0][2]
values[1][0]	values[1][1]	values[1][2]
values[2][0]	values[2][1]	values[2][2]
values[3][0]	values[3][1]	values[3][2]

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Two-dimensional Lists
  - Access the elements in a nested loop
  - Consider:

'Amir' [0][0]	'Conner' [0][1]	'Darla' [0][2]
'ID 112' [1][0]	'ID 204' [1][1]	'ID 157' [1][2]
'15.75' [2][0]	'18.50' [2][1]	'28.30' [2][2]

```
emp_list = [['Amir', 'Conner', 'Darla'],
            ['ID 112', 'ID 204', 'ID 157'],
            ['15.75', '18.50', '28.30']]
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Two-dimensional List Access – Nested Loop

```
ROWS = 3
COLS = 3

emp_list = [['Amir', 'Conner', 'Darla'],
            ['ID 112', 'ID 204', 'ID 157'],
            ['15.75', '18.50', '28.30']]


for c in range(COLS):
    for r in range(ROWS):
        print(emp_list[r][c], end='\t')

    print()
```

```
Amir    ID 112  15.75
Conner  ID 204  18.50
Darla   ID 157  28.30
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Tuples
  - A ***tuple*** is a list that is immutable and cannot be changed
    - Process faster
    - Protects the data
    - Support all list operations and built-in functions
      - Except those that modify lists

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Tuples
  - To modify a tuple, it can be converted to a list, and then back to a tuple

```
my_tuple = tuple(my_list)        # convert list to tuple
my_list2 = list(my_tuple)        # convert tuple to list
```

Plotting List Data with matplotlib

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Plotting List Data
  - One package for plotting in Python is ***matplotlib***
    - Enables plotting line, bar, histogram, scatterplots, pie charts, and more using list data in an auto-scaling resizable window
    - Not part of the Python standard library, and must be installed separately using the Python ***pip*** installer

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Plotting List Data
  - Once matplotlib is installed, the module ***pyplot*** from the package is imported similar to the way that the math package is imported
  - Note the module name, dot operator, and package
    - Typically the module is imported "as" a shortened name to lessen the amount of typing each time it is accessed. Here it is imported as "plt"

```
import matplotlib.pyplot as plt
```

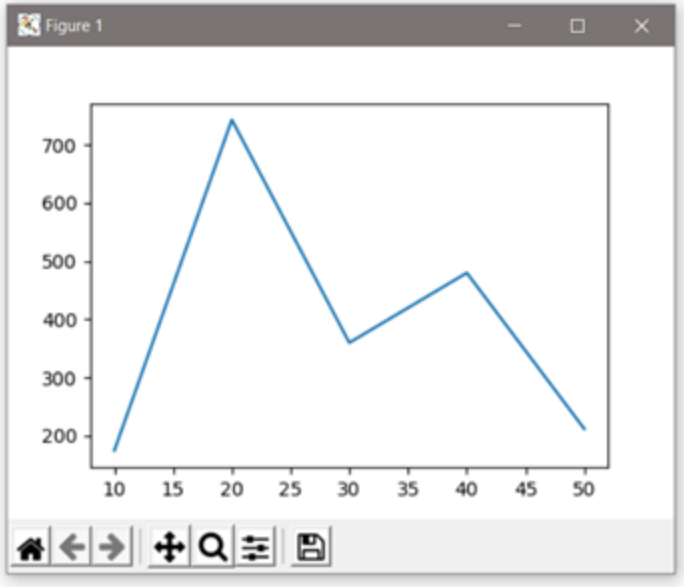
Chapter 8 Strings, Lists, Dictionaries, and Sets

- Using pyplot from matplotlib
  - Establish the number of data points using lists
  - The call to ***plot*** builds the graph in memory
  - The call to ***show*** actually displays the plot

```
import matplotlib.pyplot as plt

x_coords = [10, 20, 30, 40, 50]

y_coords = [175, 743, 360, 480, 212]

plt.plot(x_coords, y_coords)
plt.show()
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Using pyplot from matplotlib
  - The data is plotted



  - The features are automatically added in the lower left-hand corner including zooming in a rectangular shape, saving the image, and others

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Using pyplot from matplotlib
  - Many options for customizing charts is provided in the module
    - Axis labels, tick marks, data markers, the width of bars for bar charts, and slice labels for pie charts
      - Tic mark labels, axis labels, and a title for the chart add clarity

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Using pyplot from matplotlib
  - The options provide for a more informative chart

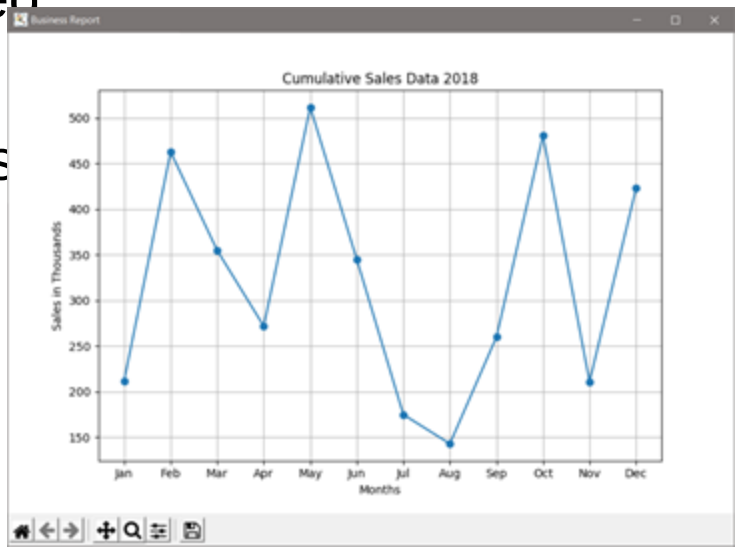
axis title

chart title

tic mark labels

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Using pyplot from matplotlib
  - Plotting two lines requires two plot functions
  - There is a legend option with labeling
  - Line styles can be assigned
  - Different markers can be used
  - A grid background
  - And a variety of other options



Chapter 8 Strings, Lists, Dictionaries, and Sets

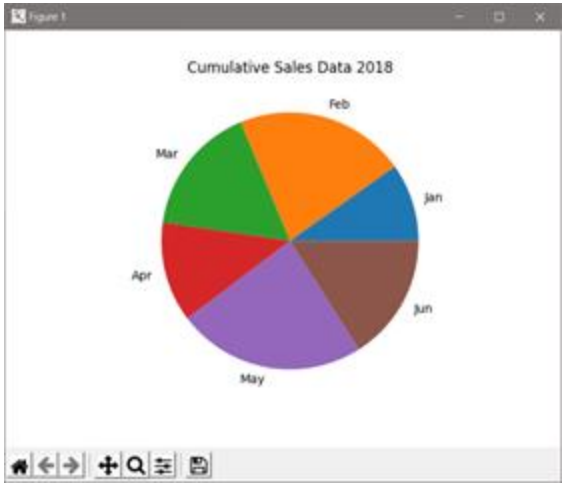
- Plotting bar charts

```
def chart_list(sales_list):
        x_coords = [0,10,20,30,40,50]
        y_coords = sales_list
        plt.xticks([0,10,20,30,40,50],
                   ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'])

        bar_width = 2
        plt.bar(x_coords, y_coords, bar_width)


        plt.title('Cumulative Sales Data 2018')
        plt.ylabel('Sales in Thousands')
        plt.xlabel('Months')

        plt.show()

main()
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Plotting Pie Charts

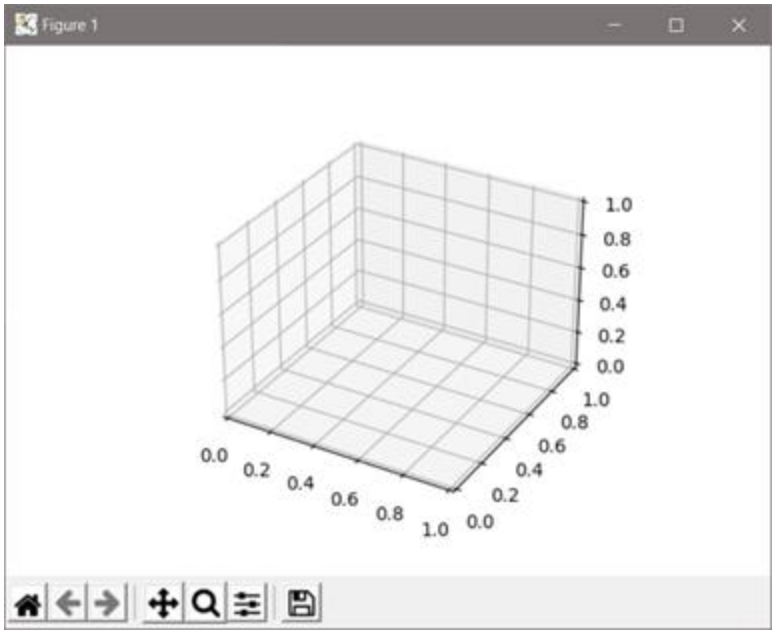
```
def main():
    sales = [212, 463, 355, 272, 512, 345]
    pie_list(sales)

def pie_list(sales_list):
        slice_labels = ['Jan', 'Feb', 'Mar',
                        'Apr', 'May', 'Jun']
        plt.pie(sales_list, labels = slice_labels)
        plt.title('Cumulative Sales Data 2018')
        plt.show()

main()
```



Chapter 8 Strings, Lists, Dictionaries, and Sets

- Plotting 3D

```
fig = plt.figure(figsize=(4,4))
ax = fig.add_subplot(111, projection='3d')

plt.show()
```



Dictionaries and Sets

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Containers that store and manage data are referred to as data structures which can be used to implement ***collections***
  - Collections are objects that store other objects as elements
  - A list is an example of a collection
  - A dictionary which stores elements as key/value pairs is a collection
  - Sets which contain no duplicates are collections
  - There are benefits and limitations with each collection type that should be considered when using them in a solution

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Dictionaries
  - A ***dictionary*** is an associative array container with a key and a value associated with the key
  - Consider a data set of student ID numbers and student names
  - A dictionary could store the ID number as the key, and the name would be the associated value for the key

Key (Student ID)	Value (Student Name)
10310	Allison Knox
11298	Amir Cumber
10452	Cody Garfield
12034	Layna Camron

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Dictionaries
  - A dictionary can be created by assigning key/value pairs to a dictionary name

```
students = {10310:'Alison Knox', 11298:'Amir Cumber',...}
```

  - Typically, a dictionary is created by declaring an empty dictionary and then adding key/value pairs

```
dictionary_name = {}

dictionary_name[key] = value
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Dictionaries
  - To add a key/value pair to a dictionary, the key is placed in the brackets and the value is assigned

```
students = {}

students[10310] = 'Allison Knox'
students[11298] = 'Amir Cumber'
students[10452] = 'Cody Garfield'
students[12034] = 'Layna Camron'
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Dictionaries
  - To access an element in a dictionary, the key is used
    - If the key does not exist, there is an error
    - Test to be sure the key exists in the dictionary

```
ID = int(input('Enter the student ID: '))

if ID in students:
    print(students[ID])
else:
    print('That ID is not valid')
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Dictionaries
  - There cannot be any duplicate keys in a dictionary
  - When assigning a value to a key in a dictionary, if the key exists, the value will be changed
  - If the key does not exist, the key/value pair will be added to the dictionary
  - For testing and debugging, the a dictionary can be passed to the print function

```
print(students)
```

```
{10310: 'Allison Knox', 11298: 'Amir Cumber', 10452: 'Cody Garfield',
12034: 'Layna Camron'}
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Dictionaries
  - To ***del*** statement is used to delete an element from a dictionary using the key
    - If the key does not exist, an error will result

```
if ID in students:
    del students[ID]
else:
    print('That ID is not valid')
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Dictionaries
  - The ***len()*** function returns the number of key/value pairs

```
print('There are ' + str(len(students)) + ' students')
```

  - The ***get()*** function can determine if an element exists in a dictionary, and provides for a default value if the key does not exist

```
stu_name = students.get(10310, 'Not found')
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - A ***set*** is a collection that cannot contain duplicates
  - Set operations include union, intersection, difference, and symmetric difference
  - Sets are optimized in memory for fast searching
  - A set can be declared and populated later or initialized when declared

```
set_name = set()

set_name = ([element, element, ...])
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The ***add()*** method is used to add an element
    - Again a set cannot have duplicates
  - A for-in loop accesses the elements

```
numset = set([1,2,3,])

numset.add(4)

for num in numset:
    print(num, end=':')          1:2:3:4:
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - There are two ways to remove an element
    - ***remove*** – causes an error if the element is not in the set
    - ***discard*** – does not cause an error

```
numset = set([1,2,3,4])

numset.remove(3)
numset.discard(2)

for num in numset:
    print(num, end=':')
```

```
1:4:
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The *in* and *not in* operators can be used to determine if an element is in a set

```
numset = set([1,2,3,4,5])

print(str(len(numset)))

search_value = 3

if search_value in numset:
    print('Found it')
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The ***union*** method returns a set of elements that is the union of both sets
    - All of the elements that appear in the sets without duplicates
    - The "|" operator (referred to as a pipe) can also be used

```
set1.union(set2)

set1 | set2
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The ***intersection*** method returns a set of elements that appear in both sets
    - The “&” operator (ampersand) can also be used.

```
set1.intersection(set2)

set1 & set2
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The ***difference*** method returns a set of elements that appear in set1 but do not appear in set2
    - The subtraction "-" operator can also be used.

```
set1.difference(set2)
set1 - set2
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The ***symmetric difference*** method returns a set of elements that do not appear in both sets
    - The "^" operator (caret symbol) can also be used.

```
set1.symetric_difference(set2)

set1 ^ set2
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The ***issubset*** method returns a Boolean value
    - True if set2 is a subset of set1
    - False otherwise
    - The comparison operator can also be used

```
set2.issubsset(set1)

set2 <= set1
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

- Sets
  - The ***issuperset*** method returns a Boolean value
    - True if set1 is a superset of set2
    - False otherwise
    - The comparison operator can also be used

```
set1.issuperset(set2)

set1 >= set2
```

Chapter 8 Strings, Lists, Dictionaries, and Sets

Chapter 8 Strings, Lists, Dictionaries, and Sets