Chapter 3: Working with Data

Reading and writing files

You can read and write files using the built-in **`open`** function and the file object it returns. Here is an example of how to read a file:

```
# Open the file in read mode
with open('my_file.txt', 'r') as f:
  # Read the contents of the file into a variable
  file_contents = f.read()
  # Print the contents of the file
  print(file_contents)
```

In the example above, the **`open`** function is called with the file name and the mode 'r' (for read). The function returns a file object, which is stored in the **`f`** variable. The **`read`** method of the file object is then called to read the contents of the file into the **`file_contents`** variable. Finally, the contents of the file are printed using the **`print`** function.

Here is an example of how to write to a file:

```
# Open the file in write mode
with open('my_file.txt', 'w') as f:
  # Write some text to the file
  f.write("Hello, world!")
```

In the example above, the **`open`** function is called with the file name and the mode 'w' (for write). The function returns a file object, which is stored in the **`f`** variable. The **`write`** method of the file object is then called to write the string "Hello, world!" to the file.

Note that the **`with`** statement is used to open the file and automatically close it when the block of code is finished executing. This is a recommended practice to ensure that the file is properly closed and released after you are done with it.

Working with data structures

Lists

- Lists are ordered collections of items that can be of any data type (e.g. integers, strings, objects, etc.)
- Lists are defined using square brackets [] and items are separated by commas
- Lists are mutable, meaning you can change their contents by adding, removing, or modifying items

Here are some examples of how to work with lists:

```
# Define a list
```

```
my_list = [1, 2, 3, 4]

# Access an item in the list
item = my_list[2] # item is 3

# Modify an item in the list
my_list[3] = 5

# Add an item to the end of the list
my_list.append(6)

# Remove an item from the list
my_list.remove(4)
```

Tuples

- Tuples are similar to lists, but they are immutable, meaning you cannot modify their contents once they are created
- Tuples are defined using parentheses () and items are separated by commas

Here is an example of how to work with tuples:

```
# Define a tuple
my_tuple = (1, 2, 3)

# Access an item in the tuple
item = my_tuple[1] # item is 2

# Cannot modify items in a tuple
my_tuple[1] = 4 # this will raise a TypeError
```

Dictionaries

A dictionary is a collection of key-value pairs that is unordered, changeable, and does not allow duplicates. Dictionaries are also known as associative arrays or hash maps.

Here is an example of how to create a dictionary:

```
# Create an empty dictionary
my_dict = {}

# Add key-value pairs to the dictionary
my_dict['name'] = 'John'
my_dict['age'] = 30
my_dict['city'] = 'New York'

print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city': 'New York'}
```

In this example, we have created an empty dictionary called `my_dict` and then added three key-value pairs to it. The keys are `'name'`, `'age'`, and `'city'`, and the corresponding values are `'John'`, `30`, and `'New York'`, respectively.

You can also create a dictionary using the `dict()` function and a sequence of key-value pairs, like this:

```python
# Create a dictionary using the dict() function
my_dict = dict([('name', 'John'), ('age', 30), ('city', 'New York')])

print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city': 'New York'}
```

To access the values in a dictionary, you can use the square brackets notation and the key of the value you want to access, like this:

```python
# Access a value in the dictionary
print(my_dict['name'])  # Output: 'John'
print(my_dict['age'])   # Output: 30
print(my_dict['city'])  # Output: 'New York'
```

You can also use the `get()` method to access the values in a dictionary, which returns a default value if the key is not found in the dictionary:

```python
# Access a value in the dictionary using the get() method
print(my_dict.get('name'))  # Output: 'John'
print(my_dict.get('age'))   # Output: 30
print(my_dict.get('city'))  # Output: 'New York'

# Access a value with a key that does not exist in the dictionary
print(my_dict.get('country', 'United States'))  # Output: 'United States'
```

You can also update the values in a dictionary, add new key-value pairs, and delete key-value pairs using the assignment operator, the `update()` method, and the `del` statement, respectively.

Sets

Sets are unordered collections of unique items. Sets are defined using curly braces {} and items are separated by commas. Sets are mutable, meaning you can add or remove items. Here are some examples of how to work with sets:

```python
# Define a set
my_set = {1, 2, 3}

# Add an item to the set
my_set.add(4)

# Remove an item from the set
my_set.remove(2)
```

```
# Check if an item is in the set
if 3 in my_set:
  print("3 is in the set")
```

Pandas

Pandas is a popular Python library for working with data in the form of tabular data structures (i.e. dataframes). Here are some examples of how to use pandas to work with data:

Importing Pandas

To use pandas in your Python code, you will need to import it using the **import** statement:

```
import pandas as pd
```

Reading a CSV File

You can use pandas to read a CSV (Comma Separated Values) file into a dataframe using the **read_csv** function:

```
df = pd.read_csv('my_data.csv')
```

The **read_csv** function returns a dataframe object containing the data from the CSV file.

Accessing Data

Once you have a dataframe, you can access the data in it using the **[]** operator and the name of the column:

```
# Access the 'Name' column
names = df['Name']

# Access multiple columns
selected_columns = df[['Name', 'Age', 'Gender']]
```

Manipulating data (e.g. sorting, filtering, aggregating)

Sorting Data

You can use the **sorted** function to sort a list or tuple in ascending order:

```
# Define a list of integers
my_list = [5, 2, 7, 1, 3]

# Sort the list in ascending order
sorted_list = sorted(my_list)

# Define a tuple of strings
my_tuple = ('c', 'a', 'b')
```

```
# Sort the tuple in ascending order
sorted_tuple = sorted(my_tuple)
```

You can also use the **sort** method to sort a list in place:

```
# Define a list of integers
my_list = [5, 2, 7, 1, 3]

# Sort the list in ascending order
my_list.sort()
```

Filtering Data

You can use a list comprehension and a boolean condition to filter a list or tuple:

```
# Define a list of integers
my_list = [5, 2, 7, 1, 3]

# Filter the list to get only even numbers
filtered_list = [x for x in my_list if x % 2 == 0]

# Define a tuple of strings
my_tuple = ('c', 'a', 'b')

# Filter the tuple to get only strings of length 2
filtered_tuple = tuple(x for x in my_tuple if len(x) == 2)
```

Aggregating Data

You can use the **sum** function to get the sum of the items in a list or tuple:

```
# Define a list of integers
my_list = [5, 2, 7, 1, 3]

# Get the sum of the items in the list
total = sum(my_list)

# Define a tuple of integers
my_tuple = (5, 2, 7, 1, 3)

# Get the sum of the items in the tuple
total = sum(my_tuple)
```

You can also use other functions such as min, max, and len to get the minimum, maximum, and length of a list or tuple.