Tuples and Dictionaries

10.1 Introduction to Tuples

Tuples are immutable sequences of Python objects. They are similar to lists, but they cannot be changed once they are created. This makes them useful for storing data that should not be modified, such as the coordinates of a point or the date of a holiday.

Tuples are created using parentheses. The following code creates a tuple containing the numbers 1, 2, and 3:

```
my_tuple = (1, 2, 3)
```

Tuples can contain any type of Python object, including other tuples. The following code creates a tuple containing a string, a number, and a list:

```
my_tuple = ("Hello", 2, [1, 2, 3])
```

The elements of a tuple can be accessed using the same syntax as lists. The following code prints the first element of my_tuple:

```
print(my_tuple[0])
```

Output:

```
Hello
```

10.2 Tuple Operations

Tuples support a number of operations, including:

- **Indexing:** Tuples can be indexed using the same syntax as lists. The following code prints the first element of my_tuple:

```
print(my_tuple[0])
```

Output:

```
Hello
```

- **Slicing:** Tuples can be sliced using the same syntax as lists. The following code prints the first two elements of my_tuple:

```
print(my_tuple[0:2])
```

Output:

```
('Hello', 2)
```

- **Concatenation:** Tuples can be concatenated using the + operator. The following code creates a new tuple that contains all of the elements of my_tuple1 and my_tuple2:

```
my_tuple3 = my_tuple1 + my_tuple2
```

- **Iteration:** Tuples can be iterated over using a for loop. The following code prints each element of my_tuple:

```
for element in my_tuple:
    print(element)
```

Output:

```
Hello
2
[1, 2, 3]
```

- **Membership:** The in operator can be used to test if an element is in a tuple. The following code prints True if the string "Hello" is in my_tuple:

```
if "Hello" in my_tuple:
    print("True")
```

Output:

```
True
```

- **Length:** The len() function can be used to find the length of a tuple. The following code prints the length of my_tuple:

```
print(len(my_tuple))
```

Output:

```
3
```

10.3 Tuple Methods and Built-in Functions

Tuples support a few methods and built-in functions that can be used to perform various operations on them:

1. count():

Counts the number of occurrences of a given element in the tuple.

Syntax:

```
tuple.count(element)
```

Example:

```
test_tuple = (1, 2, 3, 2, 4, 2)

print("Count of 2 in the tuple:", test_tuple.count(2))

# Output:
# Count of 2 in the tuple: 3
```

2. index():

Searches for the first occurrence of a given element in the tuple and returns its index.

Syntax:

```
tuple.index(element)
```

Example:

```
test_tuple = (1, 2, 3, 2, 4, 2)

print("Index of 2 in the tuple:", test_tuple.index(2))

# Output:
# Index of 2 in the tuple: 1
```

3. len():

Returns the length of the tuple, which is the number of elements it contains.

Syntax:

```
len(tuple)
```

Example:

```
test_tuple = (1, 2, 3, 4, 5)

print("Length of the tuple:", len(test_tuple))

# Output:
# Length of the tuple: 5
```

4. min():

Returns the smallest element in the tuple.

Syntax:

```
min(tuple)
```

Example:

```
test_tuple = (1, 2, 3, 4, 5)

print("Smallest element in the tuple:", min(test_tuple))

# Output:
# Smallest element in the tuple: 1
```

5. max():

Returns the largest element in the tuple.

Syntax:

```
max(tuple)
```

Example:

```
test_tuple = (1, 2, 3, 4, 5)

print("Largest element in the tuple:", max(test_tuple))

# Output:
# Largest element in the tuple: 5
```

6. sum():

Returns the sum of all elements in the tuple.

Syntax:

```
sum(tuple)
```

Example:

```
test_tuple = (1, 2, 3, 4, 5)

print("Sum of elements in the tuple:", sum(test_tuple))
```

```
# Output:
# Sum of elements in the tuple: 15
```

7. sorted():

Returns a new tuple containing the elements of the original tuple sorted in ascending order.

Syntax:

```
sorted(tuple)
```

Example:

```
test_tuple = (5, 2, 1, 4, 3)

print("Sorted tuple:", sorted(test_tuple))

# Output:
# Sorted tuple: (1, 2, 3, 4, 5)
```

10.4 Tuple Assignment

Tuple assignment allows you to assign multiple values to multiple variables simultaneously. This is done by placing the variables on the left-hand side of the assignment operator (=) and the tuple on the right-hand side. The number of variables on the left-hand side must match the number of elements in the tuple on the right-hand side.

Example:

```
x, y = (1, 2)

print("Value of x:", x)
print("Value of y:", y)

# Output:
# Value of x: 1
# Value of y: 2
```

Tuple assignment can also be used to swap the values of two variables:

```
x, y = y, x

print("Value of x:", x)
print("Value of y:", y)

# Output:
# Value of x: 2
# Value of y: 1
```

10.5 Nested Tuples

Nested tuples are tuples that contain other tuples as elements. They can be created using parentheses and commas, just like regular tuples. For example:

```
my_tuple = (1, 2, (3, 4), 5)
```

This tuple contains four elements:

- The integer 1
- The integer 2
- The tuple (3, 4)
- The integer 5

Nested tuples can be accessed using the same syntax as regular tuples. For example, to access the tuple (3, 4), you would use the following expression:

```
my_tuple[2]
```

This would return the tuple (3, 4).

Nested tuples can be used to represent complex data structures. For example, the following tuple represents a list of students and their grades:

```
students = (
    ('John', 'Doe', (90, 85, 95)),
    ('Jane', 'Smith', (85, 90, 95)),
    ('Michael', 'Jones', (95, 90, 85))
)
```

This tuple contains three elements:

- The tuple ('John', 'Doe', (90, 85, 95)) represents John Doe and his grades.
- The tuple ('Jane', 'Smith', (85, 90, 95)) represents Jane Smith and her grades.
- The tuple ('Michael', 'Jones', (95, 90, 85)) represents Michael Jones and his grades.

To access the first student's name, you would use the following expression:

```
students[0][0]
```

This would return the string 'John'.

10.6 Tuple Handling

Tuples can be handled in a variety of ways. Some of the most common operations include:

- **Accessing elements:** Elements of a tuple can be accessed using the same syntax as regular variables. For example, to access the first element of a tuple, you would use the following expression:

```
my_tuple[0]
```

- **Slicing:** Tuples can be sliced using the same syntax as regular lists. For example, to get a slice of the first three elements of a tuple, you would use the following expression:

```
my_tuple[0:3]
```

This would return a new tuple containing the elements ('John', 'Doe', 90).

- **Concatenation:** Tuples can be concatenated using the + operator. For example, to concatenate two tuples, you would use the following expression:

```
my_tuple1 + my_tuple2
```

This would return a new tuple containing all of the elements from both tuples.

- **Iteration:** Tuples can be iterated over using a for loop. For example, to iterate over a tuple, you would use the following code:

```
for element in my_tuple:
    print(element)
```

This would print each element of the tuple to the console.

- **Membership:** The in operator can be used to check if an element is present in a tuple. For example, to check if the integer 90 is present in a tuple, you would use the following expression:

```
90 in my_tuple
```

This would return True if the integer 90 is present in the tuple, and False otherwise.

10.7 Introduction to Dictionaries

A dictionary is a data structure that stores data in key-value pairs. Each key is associated with a value and is used to retrieve the value from the dictionary. Dictionaries are useful for storing data that can be accessed by a key, such as a name or an ID number.

Example:

```
# Create a dictionary
my_dict = {"name": "John Doe", "age": 30, "city": "New York"}

# Access the value associated with a key
print(my_dict["name"])  # John Doe
```

```
# Add a new key-value pair to the dictionary
my_dict["job"] = "Software Engineer"

# Update the value associated with a key
my_dict["age"] = 31

# Delete a key-value pair from the dictionary
del my_dict["city"]

# Print the dictionary
print(my_dict)  # {'name': 'John Doe', 'age': 31, 'job': 'Software
Engineer'}
```

10.8 Dictionaries are Mutable

Dictionaries are mutable, which means that they can be changed after they have been created. This is in contrast to tuples, which are immutable and cannot be changed once they have been created.

Example:

```
# Create a dictionary
my_dict = {"name": "John Doe", "age": 30, "city": "New York"}

# Add a new key-value pair to the dictionary
my_dict["job"] = "Software Engineer"

# Update the value associated with a key
my_dict["age"] = 31

# Delete a key-value pair from the dictionary
del my_dict["city"]

# Print the dictionary
print(my_dict)  # {'name': 'John Doe', 'age': 31, 'job': 'Software
Engineer'}
```

10.9 Dictionary Operations

Dictionaries offer a comprehensive range of operations to manipulate key-value pairs effectively. Some key operations in Python dictionaries include:

- **Accessing Elements:**

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
print(my_dict['age'])  # Output: 25
```

- **Adding Elements:**

```
my_dict = {'name': 'John', 'age': 25}
my_dict['city'] = 'New York'  # Add new key-value pair
```

```
print(my_dict)  # Output: {'name': 'John', 'age': 25, 'city': 'New
York'}
```

- **Updating Elements:**

```
my_dict = {'name': 'John', 'age': 25}
my_dict['age'] = 30  # Update existing key-value pair
print(my_dict)  # Output: {'name': 'John', 'age': 30}
```

- **Deleting Elements:**

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
del my_dict['city']  # Delete 'city' key-value pair
print(my_dict)  # Output: {'name': 'John', 'age': 25}
```

- **Checking Membership:**

```
my_dict = {'name': 'John', 'age': 25}
print('name' in my_dict)  # Output: True
print('city' in my_dict)  # Output: False
```

- **Length of a Dictionary:**

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
print(len(my_dict))  # Output: 3
```

- **Creating a Copy of a Dictionary:**

```
my_dict = {'name': 'John', 'age': 25}
new_dict = my_dict.copy()  # Creates a new dictionary with the
same data
print(new_dict)  # Output: {'name': 'John', 'age': 25}
```

- **Merging Dictionaries:**

```
dict1 = {'name': 'John', 'age': 25}
dict2 = {'city': 'New York', 'state': 'NY'}
merged_dict = {**dict1, **dict2}  # Merge two dictionaries
print(merged_dict)  # Output: {'name': 'John', 'age': 25, 'city':
'New York', 'state': 'NY'}
```

10.10 Traversing a Dictionary

Traversing a dictionary involves iterating through its key-value pairs to access and process the data. Here are a few ways to traverse a dictionary in Python:

- **Using a for loop with items():**

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

Output:

```
name: John
age: 25
city: New York
```

- **Using a for loop with keys():**

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for key in my_dict.keys():
    print(f"{key}: {my_dict[key]}")
```

Output:

```
name: John
age: 25
city: New York
```

- **Using a for loop with values():**

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for value in my_dict.values():
    print(value)
```

Output:

```
John
25
New York
```

- **Using the get() Method:** The get() method can be used to access dictionary values without raising a KeyError. It takes the key as the first argument and an optional default value as the second argument. If the key exists in the dictionary, it returns the corresponding value. Otherwise, it returns the default value.

```
my_dict = {'name': 'John', 'age': 25}
print(my_dict.get('name'))  # Output: John
print(my_dict.get('city', 'Unknown'))  # Output: Unknown
```

Traversing dictionaries allows you to access and manipulate data efficiently and is a fundamental technique for working with key-value data structures in Python.

Dictionary Methods and Built-in functions

Python provides a number of methods and built-in functions that can be used to manipulate dictionaries. Some of the most commonly used ones are:

- **dict.keys()**: Returns a list of all the keys in the dictionary.
- **dict.values()**: Returns a list of all the values in the dictionary.
- **dict.items()**: Returns a list of tuples, each of which contains a key-value pair from the dictionary.
- **dict.get(key, default=None)**: Returns the value associated with the specified key, or the default value if the key is not found.
- **dict.pop(key, default=None)**: Removes the specified key and its associated value from the dictionary, and returns the value. If the key is not found, the default value is returned.
- **dict.update(other)**: Updates the dictionary with the key-value pairs from the other dictionary. If a key is present in both dictionaries, the value from the other dictionary is used.

Manipulating Dictionaries

Dictionaries can be manipulated in a number of ways, including:

- **Adding key-value pairs**: This can be done using the `dict[key] = value` syntax. For example:

```
my_dict = {}
my_dict["name"] = "John Doe"
my_dict["age"] = 30
```

- **Removing key-value pairs**: This can be done using the `del dict[key]` syntax. For example:

```
my_dict = {"name": "John Doe", "age": 30}
del my_dict["age"]
```

- **Updating key-value pairs**: This can be done using the `dict[key] = new_value` syntax. For example:

```
my_dict = {"name": "John Doe", "age": 30}
my_dict["age"] = 31
```

- **Iterating over dictionaries**: This can be done using the `for key in dict` syntax. For example:

```
my_dict = {"name": "John Doe", "age": 30}
for key in my_dict:
    print(key, my_dict[key])
```

- **Checking if a key exists**: This can be done using the `in` operator. For example:

```
my_dict = {"name": "John Doe", "age": 30}
if "name" in my_dict:
    print("The name key exists in the dictionary.")
```