Python Data Structures

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Overview of Python Data Structures

Data structures are a way of organizing and storing data so that they can be accessed and worked with efficiently. The choice of data structure can significantly impact the performance and clarity of your code. Python offers a variety of built-in data structures, each tailored to handle different types of data and operations.

Lists

Lists are one of the most versatile data structures in Python. They are ordered, mutable, and allow duplicate elements. Lists are particularly useful when you need to maintain a collection of items that can change over time.

Tuples

Tuples are similar to lists, but with one key difference: they are immutable. Once a tuple is created, its contents cannot be changed. This immutability makes tuples useful for data that should not be modified.

Dictionaries

Dictionaries are powerful data structures that store data in key-value pairs. They are unordered, mutable, and indexed by keys, which must be unique and immutable. Dictionaries are ideal for scenarios where you need to associate values with keys and perform fast lookups.

Sets

Sets are unordered collections of unique elements. They are mutable and do not allow duplicate values, making them useful for membership testing and eliminating duplicate entries.

Choosing the Right Data Structure

Choosing the appropriate data structure is crucial for optimizing performance and ensuring code readability. Here are some guidelines:

- **Use Lists:** When you need an ordered collection of items that may change over time.
- **Use Tuples:** When you need an immutable sequence of items.
- **Use Dictionaries:** When you need to associate values with unique keys and perform fast lookups.
- **Use Sets:** When you need a collection of unique items and set operations.

Creating and Accessing Lists

Lists in Python are one of the most versatile and frequently used data structures. They provide an ordered collection of items, which can be of mixed types, although typically, items are of the same type for consistency. Lists are mutable, meaning their contents can be changed after they are created, making them highly flexible for a wide range of applications.

Creating a List

Creating a list in Python is straightforward. Lists are defined using square brackets `[]`, and the items within the list are separated by commas.

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

In this example, `fruits` is a list containing three string elements. Lists can contain elements of different data types, such as integers, strings, and even other lists:

```
mixed_list = [1, 'apple', 3.14, [2, 4, 6]]
```

Here, `mixed_list` includes an integer, a string, a float, and another list.

Accessing List Elements

Elements in a list are accessed using their index, which starts at 0 for the first element. Negative indices can also be used to access elements from the end of the list, where -1 refers to the last element, -2 to the second last, and so on.

```
print(fruits[0])  # Output: apple
print(fruits[-1])  # Output: cherry
```

You can also access a range of elements using slicing. Slicing syntax is `list[start:end]`, where `start` is the index of the first element to include, and `end` is the index of the first element to exclude.

```
print(fruits[1:3])  # Output: ['banana', 'cherry']
```

Modifying Lists: Adding, Removing, and Changing Elements

Lists are mutable, meaning you can modify their content after they are created. Python provides several methods to add, remove, and change elements in a list.

Adding Elements

You can add elements to a list using the `append()` method to add a single element at the end, or the `extend()` method to add multiple elements.

```
fruits.append('orange')
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'orange']

fruits.extend(['mango', 'grape'])
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'orange', 'mango
    ', 'grape']
```

To add an element at a specific position, use the `insert()` method, which takes two arguments: the index at which to insert the element and the element itself.

```
fruits.insert(1, 'blueberry')
print(fruits)  # Output: ['apple', 'blueberry', 'banana', 'cherry', '
    orange', 'mango', 'grape']
```

Removing Elements

You can remove elements from a list using the `remove()` method to remove the first occurrence of a value, the `pop()` method to remove an element at a specific index (or the last element if no index is specified), or the `clear()` method to remove all elements.

```
fruits.remove('banana')
print(fruits)  # Output: ['apple', 'blueberry', 'cherry', 'orange', '
    mango', 'grape']

fruits.pop(2)
print(fruits)  # Output: ['apple', 'blueberry', 'orange', 'mango', '
    grape']

fruits.clear()
print(fruits)  # Output: []
```

Changing Elements

Since lists are mutable, you can change the value of elements by accessing them directly by their index.

```
fruits = ['apple', 'banana', 'cherry']
fruits[1] = 'blueberry'
print(fruits)  # Output: ['apple', 'blueberry', 'cherry']
```

List Comprehensions

List comprehensions provide a concise way to create lists. They consist of brackets containing an expression followed by a `for` clause, and can also include `if` clauses to conditionally include elements.

Basic List Comprehension

A simple example of a list comprehension that creates a list of squares of numbers from 0 to 9:

```
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List Comprehension with Conditionals

List comprehensions can also include conditional logic. For example, creating a list of even numbers from 0 to 9:

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8]
```

Nested List Comprehensions

List comprehensions can be nested to handle more complex structures. For example, creating a list of products of pairs of numbers:

```
products = [x * y for x in range(1, 4) for y in range(1, 4)]
print(products)  # Output: [1, 2, 3, 2, 4, 6, 3, 6, 9]
```

List comprehensions provide a powerful way to create and manipulate lists efficiently and with clear, concise code.

Common List Methods

Python provides a rich set of methods to work with lists, making it easy to perform common tasks. Here are some of the most frequently used list methods:

- `append(x)`: Adds an element `x` to the end of the list.
- `extend(iterable)`: Extends the list by appending elements from an iterable.
- `insert(i, x)`: Inserts an element `x` at position `i`.
- `remove(x)`: Removes the first occurrence of element `x`.
- `pop([i])`: Removes and returns the element at position `i`. If `i` is not specified, it removes and returns the last element.
- `clear()`: Removes all elements from the list.
- `index(x[, start[, end]])`: Returns the index of the first occurrence of element `x` within the optional range `start` to `end`.
- `count(x)`: Returns the number of occurrences of element `x`.
- `sort(key=None, reverse=False)`: Sorts the list in place. The `key` parameter can be a function that serves as a key for the sort comparison, and `reverse` specifies whether to sort in descending order.
- `reverse()`: Reverses the elements of the list in place.
- `copy()`: Returns a shallow copy of the list.

Examples of Common List Methods

Here are some practical examples of how these methods can be used:

```
# Example list
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]

# Append
numbers.append(3)
print(numbers)  # Output: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

# Count
print(numbers.count(1))  # Output: 2
```

```
# Index
print(numbers.index(5))  # Output: 4

# Sort
numbers.sort()
print(numbers)  # Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

# Reverse
numbers.reverse()
print(numbers)  # Output: [9, 6, 5, 5, 4, 3, 3, 2, 1, 1]

# Copy
numbers_copy = numbers.copy()
print(numbers_copy)  # Output: [9, 6, 5, 5, 4, 3, 3, 2, 1, 1]
```

Creating and Accessing Tuples

Tuples serve as an immutable sequence of values. They provide a simple and efficient way to group related data together, ensuring that once a tuple is created, its contents cannot be altered. This immutability makes tuples a reliable choice for certain types of data storage, where consistency and integrity are paramount.

Creating Tuples

Creating a tuple in Python is straightforward. Tuples are defined by enclosing a comma-separated sequence of elements within parentheses. Here's a simple example:

```
coordinates = (10.0, 20.0)
```

In this example, `coordinates` is a tuple containing two floating-point numbers. The parentheses `()` are used to define the tuple, and the values inside are separated by commas. Tuples can hold elements of various data types, such as integers, floats, strings, and even other tuples. Here's an example of a tuple containing different types of data:

```
person = ("Alice", 30, "Data Scientist", (2024, 6, 1))
```

In this case, `person` is a tuple that includes a string, an integer, another string, and another tuple representing a date. This demonstrates the flexibility of tuples in grouping together related but different types of data.

Accessing Tuple Elements

To access elements in a tuple, you use indexing, just like with lists. Tuple indices start at 0, and negative indexing is also supported. Here's how you can access elements in the `person` tuple:

```
name = person[0]        # "Alice"
age = person[1]         # 30
profession = person[2] # "Data Scientist"
date = person[3]        # (2024, 6, 1)
```

You can also use negative indexing to access elements from the end of the tuple:

```
last_element = person[-1] # (2024, 6, 1)
```

Tuple Unpacking

Tuple unpacking is a powerful feature in Python that allows you to assign the values of a tuple to multiple variables in a single statement. This can make your code more readable and concise. Here's an example of tuple unpacking:

```
coordinates = (10.0, 20.0)
x, y = coordinates
```

In this example, the values in the `coordinates` tuple are unpacked and assigned to the variables `x` and `y`. This is particularly useful when a function returns a tuple, and you want to assign the returned values to separate variables immediately.
Tuple unpacking also works with nested tuples. Consider the `person` tuple we defined earlier:

```
name, age, profession, (year, month, day) = person
```

In this case, not only are the top-level elements unpacked into `name`, `age`, and `profession`, but the nested tuple representing the date is also unpacked into `year`, `month`, and `day`.
Another common use of tuple unpacking is in looping constructs, such as when iterating over a list of tuples. Here's an example:

```
students = [("Alice", 30), ("Bob", 25), ("Charlie", 35)]
for name, age in students:
    print(f"{name} is {age} years old.")
```

In this loop, each tuple in the `students` list is unpacked into the `name` and `age` variables, making the code inside the loop cleaner and more intuitive.

Immutability and Its Implications

One of the defining characteristics of tuples is their immutability. Once a tuple is created, its contents cannot be modified. This immutability has several important implications:

Safety and Integrity

Since tuples cannot be altered, they are a safe way to ensure that the data remains consistent throughout the execution of a program. This is particularly useful for representing fixed collections of data, such as the coordinates of a point, the RGB values of a color, or configuration settings.

Hashability

Because tuples are immutable, they can be used as keys in dictionaries and stored in sets, which require their elements to be hashable. Lists, which are mutable, do not have this property.

Performance

Tuples can be more memory-efficient and faster to create than lists, especially for large sequences of data that do not need to be modified. This performance benefit can be significant in scenarios where large, immutable sequences of data are used frequently.

Predictability

Immutability ensures that functions and methods that receive tuples as arguments cannot inadvertently modify the data. This predictability makes it easier to reason about the behavior of your code and to avoid unintended side effects.
However, the immutability of tuples also means that if you need to change the data, you will need to create a new tuple with the desired modifications. This can involve copying and manipulating data, which may be less efficient than modifying a list in place.

When to Use Tuples

Tuples are best used in scenarios where the immutability, hashability, and efficiency they provide are advantageous. Here are some common use cases for tuples:

- **Fixed Collections of Items:** When you have a collection of items that should not change, such as the dimensions of a rectangle, the coordinates of a point, or configuration settings.
- **Function Returns:** When a function needs to return multiple values, using a tuple is a natural and efficient choice. This allows the caller to immediately unpack the returned values into separate variables.
- **Dictionary Keys:** When you need to use compound keys in a dictionary, tuples are an excellent choice because they are immutable and hashable.
- **Ensuring Data Integrity:** In scenarios where data integrity is critical, using tuples can prevent accidental modifications. For example, tuples can be used to store the days of the week, months of the year, or other constant collections.
- **Iterating with Enumerated Values:** The `enumerate` function returns tuples containing a count and a value, making it easy to loop through a sequence and keep track of the index.
- **Data Exchange Between Functions:** Tuples provide a simple way to group multiple pieces of data together and pass them between functions, ensuring that the grouped data remains intact and unmodified.

Creating and Accessing Dictionaries

Dictionaries are among the most powerful and flexible data structures in Python, offering a way to store and manage data in key-value pairs. This characteristic makes dictionaries particularly suitable for applications where data relationships are essential, such as databases, configuration files, and various data manipulation tasks.

Dictionaries in Python are created using curly braces {} and consist of key-value pairs, where each key is unique and immutable (strings, numbers, or tuples) and the value can be of any type. Here's how you can create a dictionary:

Creating a Dictionary

```
student = {
    'name': 'Alice',
    'age': 25,
    'course': 'Data Science'
}
```

In this example, `student` is a dictionary with three key-value pairs. The keys `'name'`, `'age'`, and `'course'` are strings, each associated with their respective values.

Accessing Values in a Dictionary

To access values in a dictionary, you use the keys. This is done using square brackets `[]`:

```
print(student['name'])  # Output: Alice
```

If you attempt to access a key that doesn't exist in the dictionary, Python will raise a `KeyError`. To avoid this, you can use the `get` method, which allows you to provide a default value if the key is not found:

```
print(student.get('grade', 'N/A'))  # Output: N/A
```

The `get` method is particularly useful for safely accessing dictionary values without risking a runtime error.

Adding, Modifying, and Removing Key-Value Pairs

Dictionaries are mutable, meaning you can add, modify, and remove key-value pairs after the dictionary has been created.

Adding Key-Value Pairs

To add a new key-value pair to a dictionary, you simply assign a value to a new key:

```
student['grade'] = 'A'
print(student)  # Output: {'name': 'Alice', 'age': 25, 'course': 'Data
    Science', 'grade': 'A'}
```

Modifying Key-Value Pairs

Modifying an existing key-value pair is done by reassigning a new value to the key:

```
student['age'] = 26
print(student)  # Output: {'name': 'Alice', 'age': 26, 'course': 'Data
    Science', 'grade': 'A'}
```

Removing Key-Value Pairs

To remove a key-value pair, you can use the `del` statement or the `pop` method. The `del` statement removes the key-value pair without returning the value, while `pop` removes the pair and returns the value:

```
del student['grade']
print(student)  # Output: {'name': 'Alice', 'age': 26, 'course': 'Data
    Science'}

age = student.pop('age')
print(age)      # Output: 26
print(student)  # Output: {'name': 'Alice', 'course': 'Data Science'}
```

The `pop` method is particularly useful when you need the value that was associated with the key before it was removed.

Clearing a Dictionary

If you need to remove all key-value pairs from a dictionary, you can use the `clear` method:

```
student.clear()
print(student)  # Output: {}
```

This effectively resets the dictionary to an empty state.

Dictionary Comprehensions

Dictionary comprehensions provide a concise way to create dictionaries. Similar to list comprehensions, dictionary comprehensions allow you to construct dictionaries in a single line of code using an iterable.

Basic Dictionary Comprehension

Here's an example of a simple dictionary comprehension that creates a dictionary of squares:

```
squares = {x: x*x for x in range(1, 6)}
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

In this example, the comprehension iterates over a range of numbers from 1 to 5, setting each number as a key and its square as the corresponding value.

Conditional Dictionary Comprehension

You can also include conditions in dictionary comprehensions. For example, to create a dictionary of squares only for even numbers:

```
even_squares = {x: x*x for x in range(1, 6) if x % 2 == 0}
print(even_squares)  # Output: {2: 4, 4: 16}
```

The condition `if x % 2 == 0` ensures that only even numbers are included in the resulting dictionary.

Complex Dictionary Comprehension

Dictionary comprehensions can also be used to transform or filter data from existing dictionaries. For instance, suppose you have a dictionary of student grades and you want to create a new dictionary with only students who passed:

```
grades = {'Alice': 85, 'Bob': 70, 'Charlie': 50, 'Diana': 90}
passing_grades = {student: grade for student, grade in grades.items()
    if grade >= 60}
print(passing_grades)  # Output: {'Alice': 85, 'Bob': 70, 'Diana': 90}
```

Here, the comprehension iterates over the `grades` dictionary, including only those students with grades 60 or above.

Common Dictionary Methods

Python dictionaries come with a variety of built-in methods that facilitate various operations. Understanding these methods will help you manage dictionaries more effectively.

`keys` Method

The `keys` method returns a view object that displays a list of all the keys in the dictionary:

```
keys = student.keys()
print(keys)  # Output: dict_keys(['name', 'age', 'course'])
```

`values` Method

The `values` method returns a view object that displays a list of all the values in the dictionary:

```
values = student.values()
print(values)  # Output: dict_values(['Alice', 26, 'Data Science'])
```

`items` Method

The `items` method returns a view object that displays a list of the dictionary's key-value pairs as tuples:

```
items = student.items()
print(items)  # Output: dict_items([('name', 'Alice'), ('age', 26), ('
    course', 'Data Science')])
```

`update` Method

The `update` method updates the dictionary with elements from another dictionary or an iterable of key-value pairs:

```
student.update({'grade': 'A', 'age': 27})
print(student)  # Output: {'name': 'Alice', 'age': 27, 'course': 'Data
    Science', 'grade': 'A'}
```

`setdefault` Method

The `setdefault` method returns the value of a key if it is in the dictionary; if not, it inserts the key with a specified default value:

```
course = student.setdefault('course', 'Mathematics')
print(course)   # Output: Data Science
level = student.setdefault('level', 'Undergraduate')
print(level)    # Output: Undergraduate
print(student)  # Output: {'name': 'Alice', 'age': 27, 'course': 'Data
    Science', 'grade': 'A', 'level': 'Undergraduate'}
```

`popitem` Method

The `popitem` method removes and returns the last key-value pair as a tuple:

```
last_item = student.popitem()
print(last_item)  # Output: ('level', 'Undergraduate')
print(student)    # Output: {'name': 'Alice', 'age': 27, 'course': '
    Data Science', 'grade': 'A'}
```

Creating and Accessing Sets

Sets in Python are a flexible data structure, designed to handle collections of unique elements. They are particularly useful when you need to ensure that each item in a collection is distinct, or when you need to perform common mathematical set operations such as union, intersection, and difference. Sets are mutable, meaning you can add or remove elements, although they only allow immutable (hashable) types to be included as elements.

Creating a Set

Creating a set in Python is straightforward. You can define a set using curly braces {} or the `set()` function. One of the key features of a set is that it automatically removes duplicate items, ensuring that all elements are unique.

```
# Using curly braces
fruits = {'apple', 'banana', 'cherry'}
print(fruits)  # Output: {'apple', 'banana', 'cherry'}

# Using the set() function
numbers = set([1, 2, 3, 4, 5])
print(numbers)  # Output: {1, 2, 3, 4, 5}
```

When using the `set()` function, you can pass any iterable (such as a list or a string) to create a set. The elements are then extracted and added to the set, with duplicates removed.

Accessing Set Elements

Unlike lists and tuples, sets are unordered, meaning there is no indexing or order to the elements. Therefore, you cannot access elements by index. However, you can iterate over the elements of a set using a `for` loop:

```
for fruit in fruits:
    print(fruit)
```

This code will print each element in the `fruits` set. Because sets are unordered, the output may vary each time you run the code.

Adding and Removing Elements

Sets are mutable, which means you can add or remove elements after the set has been created. Python provides methods to facilitate these modifications.

Adding Elements

To add a single element to a set, you use the `add()` method. If you want to add multiple elements, you can use the `update()` method:

```
# Adding a single element
fruits.add('orange')
print(fruits)  # Output: {'apple', 'banana', 'cherry', 'orange'}

# Adding multiple elements
fruits.update(['mango', 'grape'])
print(fruits)  # Output: {'apple', 'banana', 'cherry', 'orange', 'mango
    ', 'grape'}
```

The `add()` method is used for adding one item, while `update()` can take any iterable (like a list or another set) and add each of its elements to the set.

Removing Elements

To remove an element from a set, you can use the `remove()` or `discard()` methods. The difference between them is that `remove()` will raise a `KeyError` if the element is not found, whereas `discard()` will not.

```
# Removing an element
fruits.remove('banana')
print(fruits)  # Output: {'apple', 'cherry', 'orange', 'mango', 'grape
    '}

# Using discard
fruits.discard('apple')
print(fruits)  # Output: {'cherry', 'orange', 'mango', 'grape'}

# Using remove on a non-existent element raises an error
# fruits.remove('pineapple')  # Raises KeyError

# Using discard on a non-existent element does nothing
fruits.discard('pineapple')
print(fruits)  # Output: {'cherry', 'orange', 'mango', 'grape'}
```

For removing all elements from a set, you can use the `clear()` method:

```
fruits.clear()
print(fruits)  # Output: set()
```

Set Operations: Union, Intersection, Difference

Sets support several standard operations that are useful for mathematical and logical computations. These operations can be performed using methods or operators.

Union

The union of two sets includes all elements from both sets. This can be done using the `union()` method or the `|` operator.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using union() method
union_set = set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5}

# Using | operator
union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

Intersection

The intersection of two sets includes only the elements that are present in both sets. This can be done using the `intersection()` method or the `&` operator.

```
# Using intersection() method
intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {3}

# Using & operator
intersection_set = set1 & set2
print(intersection_set)  # Output: {3}
```

Difference

The difference between two sets includes elements that are in the first set but not in the second. This can be done using the `difference()` method or the `-` operator.

```
# Using difference() method
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2}

# Using - operator
difference_set = set1 - set2
print(difference_set)  # Output: {1, 2}
```

Symmetric Difference

The symmetric difference includes elements that are in either of the sets but not in both. This can be done using the `symmetric_difference()` method or the `^` operator.

```
# Using symmetric_difference() method
sym_diff_set = set1.symmetric_difference(set2)
print(sym_diff_set)  # Output: {1, 2, 4, 5}
```

```
# Using ^ operator
sym_diff_set = set1 ^ set2
print(sym_diff_set)  # Output: {1, 2, 4, 5}
```

Common Set Methods

Besides the set operations discussed above, Python sets provide several other useful methods that can enhance your ability to manage and manipulate sets.

isdisjoint()

The `isdisjoint()` method checks if two sets have no elements in common. It returns `True` if the sets are disjoint and `False` otherwise.

```
set3 = {1, 2}
set4 = {3, 4}

print(set3.isdisjoint(set4))  # Output: True
```

issubset()

The `issubset()` method checks if all elements of one set are present in another set. It returns `True` if the set is a subset and `False` otherwise.

```
set5 = {1, 2}
set6 = {1, 2, 3, 4}

print(set5.issubset(set6))  # Output: True
```

issuperset()

The `issuperset()` method checks if a set contains all elements of another set. It returns `True` if the set is a superset and `False` otherwise.

```
print(set6.issuperset(set5))  # Output: True
```

copy()

The `copy()` method creates a shallow copy of a set.

```
set7 = set5.copy()
print(set7)  # Output: {1, 2}
```

frozenset

While not a method, it's worth mentioning `frozenset`, which is an immutable version of a set. Once created, a `frozenset` cannot be modified. This can be useful when you need a set that should not change.

```
frozen_set = frozenset([1, 2, 3])
print(frozen_set)  # Output: frozenset({1, 2, 3})
```