

Data Structures

Lists

In Python, a list is a commonly used data structure that allows you to store and manipulate a collection of items. Lists are ordered, mutable, and can contain elements of different data types. Here's how you can work with lists in Python:

Creating Lists

To create a list, you can enclose a comma-separated sequence of values within square brackets. For example:

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

Lists can contain elements of different data types, including numbers, strings, other lists, or any other objects:

```
mixed_list = [1, "hello", 3.14, [7, 8, 9]]
```

Accessing List Elements

You can access individual elements of a list using indexing. Indexing starts at 0 for the first element. For example:

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

```
print(my_list[0]) # Output: 1
```

```
print(my_list[3]) # Output: 4
```

Slicing Lists

You can extract a portion of a list using slicing. Slicing allows you to specify a range of indices to create a new list containing the selected elements. The general syntax for slicing is **start:stop:step**, where **start** is inclusive, **stop** is exclusive, and **step** specifies the interval:

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

```
sublist = my_list[1:4] # Extract elements at index 1, 2, and 3
```

```
print(sublist)      # Output: [2, 3, 4]
```

You can omit **start**, **stop**, or **step** values to use default values:

- If **start** is omitted, it defaults to the beginning of the list.
- If **stop** is omitted, it defaults to the end of the list.
- If **step** is omitted, it defaults to 1.

Modifying Lists

Lists are mutable, which means you can change their contents. You can modify individual elements using indexing, append new elements, insert elements, remove elements, and more. Here are some examples:

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

```
# Modify an element
```

```
my_list[2] = 10
```

```
# Append an element
```

```
my_list.append(6)
```

```
# Insert an element at a specific index
```

```
my_list.insert(1, 7)
```

```
# Remove an element by value
```

```
my_list.remove(4)
```

```
# Remove an element by index
```

```
del my_list[0]
```

```
# Extend the list with another list
```

```
my_list.extend([8, 9])
```

```
# Pop an element from the end
```

```
popped_value = my_list.pop()
```

```
print(my_list)      # Output: [7, 2, 10, 5, 8, 9]
```

```
print(popped_value) # Output: 6
```

List Functions and Methods

Python provides several built-in functions and methods to work with lists. Some commonly used ones include:

- `len(list)`: Returns the number of elements in the list.
- `list.append(item)`: Appends an item to the end of the list.
- `list.insert(index, item)`: Inserts an item at a specific index.
- `list.remove(item)`: Removes the first occurrence of an item by value.
- `list.pop(index)`: Removes and returns an item at the specified index.
- `list.index(item)`: Returns the index of the first occurrence of an item by value.
- `list.count(item)`: Returns the number of occurrences of an item.
- `list.sort()`: Sorts the list in ascending order.
- `list.reverse()`: Reverses the order of elements in the list.
- `list.copy()`: Returns a shallow copy of the list.
- `list.clear()`: Removes all elements from the list.

Using Lists as Stacks and Queues

In Python, you can use lists to implement data structures like stacks and queues, which are essential for managing and processing data in a specific order. Here's how you can use lists to create both stack and queue data structures:

Using Lists as Stacks

A stack is a data structure that follows the Last-In-First-Out (LIFO) principle, meaning the last element added is the first one to be removed. You can use a list to implement a stack by using the `append` method to push elements onto the stack and the `pop` method to remove elements from the top of the stack.

Here's an example of using a list as a stack:

```
stack = []
```

```
# Push elements onto the stack
```

```
stack.append(1)
```

```
stack.append(2)
```

```
stack.append(3)
```

```
# Pop elements from the stack
```

```
top_element = stack.pop()
```

```
print(top_element) # Output: 3
```

```
top_element = stack.pop()  
print(top_element) # Output: 2
```

```
top_element = stack.pop()  
print(top_element) # Output: 1
```

Using Lists as Queues

A queue is a data structure that follows the First-In-First-Out (FIFO) principle, meaning the first element added is the first one to be removed. You can use a list to implement a queue by using the **append** method to enqueue elements at the end of the queue and the **pop** method with index 0 to dequeue elements from the front of the queue. However, dequeues are not very efficient with lists since removing an element from the front of a list is relatively slow due to shifting all other elements. For a more efficient queue, you can use the **collections.deque** data structure.

Here's an example of using a list as a queue:

```
queue = []
```

```
# Enqueue elements at the end of the queue
```

```
queue.append(1)
```

```
queue.append(2)
```

```
queue.append(3)
```

```
# Dequeue elements from the front of the queue
```

```
front_element = queue.pop(0)
```

```
print(front_element) # Output: 1
```

```
front_element = queue.pop(0)
```

```
print(front_element) # Output: 2
```

```
front_element = queue.pop(0)
```

```
print(front_element) # Output: 3
```

It's important to note that while using lists for both stacks and queues is possible, Python's **`collections.deque`** is more efficient for implementing queues because it's optimized for fast **`append`** and **`popleft`** operations, ensuring better performance for queue-like behavior. Here's how you can use **`deque`** as a queue:

```
from collections import deque
```

```
queue = deque()
```

```
# Enqueue elements at the end of the queue
```

```
queue.append(1)
```

```
queue.append(2)
```

```
queue.append(3)
```

```
# Dequeue elements from the front of the queue
```

```
front_element = queue.popleft()
```

```
print(front_element) # Output: 1
```

```
front_element = queue.popleft()
```

```
print(front_element) # Output: 2
```

```
front_element = queue.popleft()
```

```
print(front_element) # Output: 3
```

Using **`collections.deque`** for queues is a more efficient and recommended approach when working with large datasets or when you need fast insertions and removals from both ends of the queue.

List Comprehensions

List comprehensions are a concise and powerful way to create lists in Python. They provide a compact syntax for generating new lists by applying an expression to each item in an iterable, such as a list, tuple, or range. List comprehensions make your code more readable

and efficient when you need to create a new list based on an existing one or a sequence of values. Here's the general syntax for list comprehensions:

```
new_list = [expression for item in iterable if condition]
```

- **expression**: The operation to be performed on each item in the iterable. The result of this operation is added to the new list.
- **item**: A variable that represents each element in the iterable.
- **iterable**: The source of data from which the list is created.
- **condition (optional)**: An optional filter that can be used to include or exclude items from the new list. If the condition is not met, the item is skipped.

Here are some examples to illustrate list comprehensions:

Example 1: Generating a list of squares

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

```
squares = [x**2 for x in numbers]
```

```
print(squares) # Output: [1, 4, 9, 16, 25]
```

Example 2: Filtering even numbers

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
evens = [x for x in numbers if x % 2 == 0]
```

```
print(evens) # Output: [2, 4, 6, 8, 10]
```

Example 3: Creating a list of words longer than a certain length

```
words = ["apple", "banana", "cherry", "date", "elderberry"]
```

```
long_words = [word for word in words if len(word) > 5]
```

```
print(long_words) # Output: ['banana', 'cherry', 'elderberry']
```

Nested list comprehensions

Nested list comprehensions are a way to create lists of lists or perform operations on nested data structures using list comprehensions. You can use nested list comprehensions to work with multi-dimensional lists or perform operations on elements that are nested within other iterable structures. Here's the basic structure of a nested list comprehension:

```
[[expression for inner_item in inner_iterable] for outer_item in outer_iterable]
```

You can also use nested list comprehensions to create more complex structures:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
flattened = [num for row in matrix for num in row]
print(flattened) # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In this example, the outer comprehension iterates over the rows of the `matrix`, and the inner comprehension iterates over the numbers within each row.

Here are a few more examples to illustrate how nested list comprehensions work:

Example 1: Transposing a Matrix

You can use nested list comprehensions to transpose a matrix by swapping rows and columns:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposed = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
print(transposed)
# Output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Example 2: Filtering Nested Lists

You can use nested comprehensions to filter elements within nested lists based on a condition:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[num for num in row if num % 2 == 0] for row in matrix]
print(filtered)
# Output: [[2], [4, 6], [8]]
```

Example 3: Flattening a List of Lists

You can use nested comprehensions to flatten a list of lists:

```
list_of_lists = [[1, 2, 3], [4, 5], [6, 7, 8]]
flattened = [num for sublist in list_of_lists for num in sublist]
print(flattened) # Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

The `del` statement / Tuples and Sequences

The `del` statement in Python is used to delete objects or elements in different contexts. It can be used to delete variables, items from a list, elements from a dictionary, or attributes of an object. Here's how the `del` statement works in different scenarios:

Deleting Variables

You can use **del** to delete a variable, making it undefined and releasing the associated memory:

```
my_variable = 42  
del my_variable  
# Now, my_variable is undefined, and you can't use it.
```

Deleting Items from a List

You can use **del** to remove an item from a list by specifying the index of the item you want to delete:

```
my_list = [1, 2, 3, 4, 5]  
del my_list[2] # Deletes the element at index 2 (value 3)
```

Deleting Elements from a Dictionary

You can use **del** to remove a key-value pair from a dictionary:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}  
del my_dict['b'] # Removes the key 'b' and its associated value
```

Deleting Attributes of an Object

You can use **del** to remove an attribute from an object:

```
class MyClass:  
    def __init__(self):  
        self.my_attribute = 42  
  
my_object = MyClass()  
del my_object.my_attribute # Removes the attribute 'my_attribute'
```

Tuples and Sequences

Tuples are another built-in data structure in Python. Tuples are similar to lists, but they are immutable, meaning their elements cannot be changed after creation. Tuples are typically used to group related data together. They are defined using parentheses or without any delimiter, separated by commas:

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

```
another_tuple = 4, 5, 6
```

You can access tuple elements using indexing just like you do with lists:

```
print(my_tuple[0]) # Output: 1
```

Tuples can be used as keys in dictionaries, while lists cannot because they are mutable.

Tuples can also be used in functions to return multiple values:

```
def get_coordinates():
```

```
    return 10, 20
```

```
x, y = get_coordinates()
```

```
print(x, y) # Output: 10 20
```

Python also supports sequences, which are iterable objects like lists and tuples. Sequences include not only lists and tuples but also strings, ranges, and more. You can perform various operations on sequences, such as indexing, slicing, and iteration.

In summary, the **del** statement is used to delete objects or elements in Python, and tuples are a type of sequence that is similar to lists but immutable. Sequences are iterable objects that support common sequence operations.

Sets

In Python, a set is an unordered collection of unique elements. Sets are used to store a collection of items where each item is unique, and the order of the items doesn't matter. Sets are defined using curly braces {} or the **set()** constructor. Here's a basic introduction to sets in Python:

Creating Sets

You can create a set by enclosing a comma-separated sequence of values in curly braces {} or using the **set()** constructor. For example:

```
my_set = {1, 2, 3, 4, 5}
```

```
another_set = set([3, 4, 5, 6, 7])
```

Adding and Removing Elements

You can add elements to a set using the **add()** method, and you can remove elements using the **remove()** or **discard()** methods. The **discard()** method is used to remove an element if it exists, while the **remove()** method raises a `KeyError` if the element is not found:

```
my_set = {1, 2, 3}
```

```
my_set.add(4)
```

```
my_set.remove(3)  
my_set.discard(2)
```

Operations on Sets

Sets support a variety of operations, including union, intersection, difference, and more. You can perform these operations using methods or operators. Here are some examples:

- Union: Combines two sets to create a new set with all unique elements.

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
union_set = set1 | set2 # Using the '|' operator
```

```
# union_set is {1, 2, 3, 4, 5}
```

Intersection: Creates a new set with elements common to both sets.

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
intersection_set = set1 & set2 # Using the '&' operator
```

```
# intersection_set is {3}
```

Difference: Creates a new set with elements from the first set that are not in the second set.

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
difference_set = set1 - set2 # Using the '-' operator
```

```
# difference_set is {1, 2}
```

Subset: Checks if one set is a subset of another.

```
set1 = {1, 2, 3}
```

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

```
is_subset = set1.issubset(set2)
```

```
# is_subset is True
```

Frozensets

A frozenset is an immutable version of a set. It is created using the **frozenset()** constructor and can be used as a dictionary key because it is hashable, unlike regular sets. Frozensets do not support operations that modify the set, such as **add()** or **remove()**. Here's how to create a frozenset:

```
frozen_set = frozenset([1, 2, 3])
```

Sets are useful for various purposes, including eliminating duplicate elements, performing set operations, and solving mathematical problems. They provide an efficient way to work with collections of unique items in Python.

Looping Techniques

In Python, there are several looping techniques to iterate over sequences like lists, tuples, dictionaries, strings, and more. Here are some common looping techniques:

1. for Loop:

The **for** loop is a fundamental looping construct in Python. It allows you to iterate over items in a sequence, such as a list, tuple, string, or range. Here's the basic syntax:

for item in sequence:

```
    # Loop body
```

```
    # Code to process each item
```

Example:

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

for fruit in fruits:

```
    print(fruit)
```

2. enumerate() Function:

The **enumerate()** function allows you to loop over elements in a sequence while keeping track of their indices. This is useful when you need both the item and its index.

Example:

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

for index, fruit in enumerate(fruits):

```
    print(f"Index {index}: {fruit}")
```

3. zip() Function:

The **zip()** function combines two or more sequences into pairs. It's often used to iterate over multiple sequences in parallel.

Example:

```
names = ["Alice", "Bob", "Charlie"]
```

```
scores = [85, 92, 78]
```

for name, score in zip(names, scores):

```
print(f"{name}: {score}")
```

4. range() Function:

The `range()` function generates a sequence of numbers within a specified range. It's commonly used in `for` loops to iterate a specific number of times.

Example:

```
for i in range(5):
```

```
    print(i)
```

6. List Comprehensions:

List comprehensions are a concise way to create new lists by applying an expression to each item in an existing sequence. They are essentially `for` loops in a more compact form.

Example:

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

```
squared = [x**2 for x in numbers]
```

7. Iterating Over Dictionary:

You can iterate over the keys, values, or key-value pairs of a dictionary using the `for` loop and dictionary methods like `items()`, `keys()`, and `values()`.

Example:

```
my_dict = {"a": 1, "b": 2, "c": 3}
```

```
for key, value in my_dict.items():
```

```
    print(f"Key: {key}, Value: {value}")
```

Errors and Exceptions

In Python, errors and exceptions are mechanisms for handling unexpected or problematic situations that can occur during the execution of a program. Errors and exceptions help you identify and deal with issues that may prevent your program from running correctly. Python provides a wide range of built-in exceptions, and you can also create custom exceptions as needed.

Here's an overview of errors and exceptions in Python:

- Syntax Errors:
- Syntax errors are also known as parsing errors. They occur when the Python interpreter cannot understand the code due to incorrect syntax. Common causes include missing colons, indentation errors, and misspelled keywords.
- Example:

```
if x = 5: # SyntaxError: invalid syntax  
    print("Hello, world")
```

Runtime Errors:

- Runtime errors occur during program execution, and they can cause the program to terminate. These errors can be due to a variety of issues, such as division by zero, attempting to access an index that is out of range, or trying to use a variable before it's been defined.

Example:

```
x = 0
```

```
result = 10 / x # ZeroDivisionError: division by zero
```

Exception Handling:

- Python provides a way to handle runtime errors gracefully using the try-except block. You can enclose the code that might raise an exception within a **try** block and then specify how to handle the exception in the **except** block.

Example:

```
try:
```

```
    x = 10 / 0
```

```
except ZeroDivisionError:
```

```
    print("Error: Division by zero")
```

- Built-in Exceptions:
- Python has a wide range of built-in exception types for handling various error situations. Some common built-in exceptions include:
- **ZeroDivisionError**: Raised when attempting to divide by zero.
- **IndexError**: Raised when trying to access an index that is out of range.
- **ValueError**: Raised when a function receives an argument of the correct data type but an inappropriate value.
- **TypeError**: Raised when performing an operation on an inappropriate data type.
- Custom Exceptions:
- You can create your own custom exceptions by defining a new class that inherits from the **Exception** or a built-in exception class. Custom exceptions are helpful for handling specific application-related errors.
- Example:

```
class CustomError(Exception):
```

```
    pass
```

```
try:
```

```
raise CustomError("This is a custom exception")
except CustomError as e:
    print("Custom error:", e)
```

The **finally** Block:

- You can also use the **finally** block in conjunction with the **try** and **except** blocks. The code in the **finally** block will execute regardless of whether an exception is raised or not. It's often used for cleanup tasks, such as closing files or network connections.

Example:

```
try:
    f = open("file.txt", "r")
    data = f.read()
except FileNotFoundError:
    print("File not found")
finally:
    f.close() # This will always be executed
```

Handling errors and exceptions effectively is an essential part of writing robust Python code. It allows your program to gracefully handle unexpected situations and recover from errors without crashing.

Classes

In Python, classes are like blueprints for creating objects. They allow you to encapsulate data and behavior into a single unit. Here's a simple example of a class in Python:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f'{self.name} says woof!')

# Creating an instance of the Dog class
my_dog = Dog("Buddy", 3)
```

```
# Accessing attributes  
print(f"my_dog.name} is {my_dog.age} years old.")
```

```
# Calling a method
```

```
my_dog.bark()
```

In this example, **Dog** is a class with attributes (**name** and **age**) and a method (**bark**). The **__init__** method is a special method that gets called when an object is created. It's used to initialize the object's attributes.

You can create multiple instances of a class, each with its own set of attributes. Classes provide a way to structure and organize code in a more modular and reusable way.

Python Scopes and Namespaces

Python has a simple yet powerful way of handling scopes and namespaces. Here are the key points:

- **Namespace:**
 - A namespace is a mapping from names to objects.
 - Namespaces are used to avoid naming conflicts in a program.
 - Python implements namespaces through dictionaries.
- **Scope:**
 - A scope is a region of a program where a namespace can be directly accessed.
 - Python uses a LEGB (Local, Enclosing, Global, Built-in) rule to determine the scope of a variable.
- **Local Scope:**
 - Variables defined inside a function have a local scope and are only accessible within that function.
 - Example:

```
def example_function():
```

```
    local_variable = 10  
    print(local_variable)
```

```
example_function()
```

```
# This will print 10
```

Enclosing Scope:

- If a function is defined inside another function, the inner function can access variables from the outer (enclosing) function.
- Example:

```
def outer_function():
```

```
    outer_variable = 20
```

```
    def inner_function():
```

```
        print(outer_variable)
```

```
    inner_function()
```

```
outer_function()
```

```
# This will print 20
```

Global Scope:

- Variables defined at the top level of a module or script have a global scope and can be accessed throughout the module or script.
- Example:

```
global_variable = 30
```

```
def print_global():
```

```
    print(global_variable)
```

```
print_global()
```

```
# This will print 30
```

Built-in Scope:

- This scope contains names that are built into Python, such as `print()`, `len()`, etc.

Class and Instance Variables

In Python, class variables and instance variables are two types of variables associated with classes.

- **Class Variables:**

- Class variables are shared among all instances of a class.

- They are defined outside any method in the class and are typically placed at the top of the class definition.
- Example:

```
class Dog:
```

```
    species = "Canis familiaris"
```

```
def __init__(self, name, age):
```

```
    self.name = name
```

```
    self.age = age
```

```
# Accessing class variable
```

```
print(Dog.species) # Output: Canis familiaris
```

```
# Creating instances
```

```
dog1 = Dog("Buddy", 3)
```

```
dog2 = Dog("Max", 5)
```

```
# Class variable is shared among instances
```

```
print(dog1.species) # Output: Canis familiaris
```

```
print(dog2.species) # Output: Canis familiaris
```

Instance Variables:

- Instance variables are unique to each instance of a class.
- They are defined inside the `__init__` method using the `self` keyword.
- Example:

```
class Car:
```

```
def __init__(self, make, model, year):
```

```
    self.make = make
```

```
    self.model = model
```

```
    self.year = year
```

```
# Creating instances with unique instance variables
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2023)

print(car1.make, car1.model, car1.year) # Output: Toyota Camry 2022
print(car2.make, car2.model, car2.year) # Output: Honda Civic 2023
```