Variables and Data Types

Variables and Datatypes in Python

Variables in Python

A **variable** in Python is a container for storing data values. Unlike some other programming languages, Python does not require you to declare the type of a variable when you create it. Python automatically determines the type based on the value you assign.

Key Points:

1. Variables are created when you assign a value to them.
2. Variable names must start with a letter or an underscore (_) and can contain letters, numbers, and underscores.
3. Variable names are case-sensitive (e.g., `myVar` and `myvar` are different).

Examples:

```python
# Assigning values to variables
x = 5          # Integer
y = 3.14       # Float
name = "John"  # String
is_active = True  # Boolean

# Updating a variable
x = x + 10

# Multiple assignments
a, b, c = 1, 2.5, "Hello"
```

Data Types in Python

Python has several built-in data types, which can be broadly categorized into **numeric**, **sequence**, **mapping**, **set**, **boolean**, and **none** types.

1. Numeric Types

- **int**: Integer values.
- **float**: Decimal numbers.
- **complex**: Complex numbers with real and imaginary parts.

```python
x = 10          # int
y = 3.14        # float
z = 1 + 2j      # complex
```

2. Text Type

- **str**: Strings, enclosed in single (`'`) or double (`"`) quotes.

```python
text = "Hello, Python!"
```

3. Sequence Types

- **list**: Ordered, mutable (can be changed), allows duplicate elements.
- **tuple**: Ordered, immutable (cannot be changed), allows duplicate elements.
- **range**: Represents a sequence of numbers.

```python
my_list = [1, 2, 3, "Python"]
my_tuple = (1, 2, 3, "Python")
my_range = range(5)  # 0, 1, 2, 3, 4
```

4. Mapping Type

- **dict**: Key-value pairs, unordered (from Python 3.7, insertion-ordered).

```python
my_dict = {"name": "Alice", "age": 25}
```

5. Set Types

- **set**: Unordered, mutable, no duplicate elements.

- **frozenset**: Unordered, immutable, no duplicate elements.

```python
my_set = {1, 2, 3, 4}
frozen_set = frozenset([1, 2, 3, 4])
```

6. Boolean Type

- **bool**: Represents `True` or `False`.

```python
is_valid = True
is_error = False
```

7. None Type

- **NoneType**: Represents the absence of a value or null.

```python
result = None
```

Dynamic Typing

Python is dynamically typed, meaning the type of a variable can change at runtime:

```python
x = 10       # int
x = "Hello"  # str
```

Type Checking and Conversion

1. **Checking Data Types** Use the `type()` function to check a variable's data type.

```python
print(type(x))  # <class 'int'>
```

2. **Type Conversion** You can convert between data types using built-in functions like `int()`, `float()`, `str()`, `list()`, etc.

```python
x = "123"
y = int(x)  # Converts string to integer
```

Would you like to try some examples?

Strings, Slicing, Split

Strings in Python

A **string** in Python is a sequence of characters enclosed in single quotes (`'`) or double quotes (`"`).

Example:

```python
string1 = 'Hello'
string2 = "World"
```

String Slicing

String slicing allows you to access parts of a string using **indexes**. In Python:

- Indexes start at `0` for the first character.
- Negative indexes start at `-1` for the last character.

Syntax:

```python
```

```
string[start:end:step]
```

- **start**: The starting index (inclusive).
- **end**: The ending index (exclusive).
- **step**: The interval between characters.

Examples:

python

```python
text = "Python Programming"

# Accessing a single character
print(text[0])    # 'P'
print(text[-1])   # 'g'

# Slicing a substring
print(text[0:6])  # 'Python' (from index 0 to 5)
print(text[:6])   # 'Python' (start defaults to 0)
print(text[7:])   # 'Programming' (end defaults to the last character)
print(text[::2])  # 'Pto rgamn' (every second character)

# Reversing a string
print(text[::-1])  # 'gnimmargorP nohtyP'
```

String Methods: `split()`

The `split()` method breaks a string into a list of substrings based on a specified delimiter (default is space).

Syntax:

python

```python
string.split(separator, maxsplit)
```

- **separator**: The delimiter to split on (default is whitespace).

- **maxsplit**: The maximum number of splits (optional).

Examples:

```python
text = "Python is awesome"

# Default splitting (by whitespace)
words = text.split()
print(words)  # ['Python', 'is', 'awesome']

# Splitting with a specific delimiter
csv_data = "red,green,blue"
colors = csv_data.split(",")
print(colors)  # ['red', 'green', 'blue']

# Limiting the number of splits
limited_split = text.split(" ", 1)
print(limited_split)  # ['Python', 'is awesome']
```

Practice Examples:

1. Given `message = "Learn Python Step by Step"`, extract:
   - The word "Python".
   - The phrase "Step by Step".
2. Split the string `"one,two,three,four"` into a list of words.

Let me know if you'd like to try them or need more help!

Operators

Operators in Python

Operators are symbols or keywords used to perform operations on variables and values. Python supports several types of operators:

---

1. Arithmetic Operators

Used for mathematical operations.

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`5 - 3`	`2`
`*`	Multiplication	`5 * 3`	`15`
`/`	Division	`5 / 2`	`2.5`
`//`	Floor Division	`5 // 2`	`2`
`%`	Modulus (remainder)	`5 % 2`	`1`
`**`	Exponentiation	`2 ** 3`	`8`

---

2. Comparison (Relational) Operators

Compare two values and return a Boolean (`True` or `False`).

Operator	Description	Example	Result
`==`	Equal to	`5 == 3`	`False`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`5 > 3`	`True`
`<`	Less than	`5 < 3`	`False`
`>=`	Greater than or equal	`5 >= 3`	`True`
`<=`	Less than or equal	`5 <= 3`	`False`

---

3. Logical Operators

Used to combine conditional statements.

Operator	Description	Example	Result
`and`	Logical AND	`True and False`	`False`
`or`	Logical OR	`True or False`	`True`

Operator	Description	Example	Result
`not`	Logical NOT	`not True`	`False`

---

4. Assignment Operators

Used to assign or modify values of variables.

Operator	Description	Example	Equivalent To
`=`	Assign	`x = 5`	`x = 5`
`+=`	Add and assign	`x += 3`	`x = x + 3`
`-=`	Subtract and assign	`x -= 3`	`x = x - 3`
`*=`	Multiply and assign	`x *= 3`	`x = x * 3`
`/=`	Divide and assign	`x /= 3`	`x = x / 3`
`//=`	Floor divide and assign	`x //= 3`	`x = x // 3`
`%=`	Modulus and assign	`x %= 3`	`x = x % 3`
`**=`	Exponent and assign	`x **= 3`	`x = x ** 3`

---

5. Bitwise Operators

Used to perform operations at the binary level.

Operator	Description	Example	Result (Binary)	Result (Decimal)
`&`	AND	`5 & 3`	`0101 & 0011`	`1`
`` ` ``	`` ` ``	OR	`` `5 ``	3`` ` ``
`^`	XOR	`5 ^ 3`	`0101 ^ 0011`	`6`
`~`	NOT	`~5`	`~0101`	`-6`
`<<`	Left Shift	`5 << 1`	`0101 << 1`	`10`
`>>`	Right Shift	`5 >> 1`	`0101 >> 1`	`2`

---

6. Membership Operators

Used to check if a value is in a sequence (like a list, string, or tuple).

Operator	Description	Example	Result
`in`	Returns `True` if present	`'a' in 'apple'`	`True`
`not in`	Returns `True` if not present	`'x' not in 'apple'`	`True`

---

7. Identity Operators

Used to check if two variables reference the same object.

Operator	Description	Example	Result
`is`	Returns `True` if same object	`x is y`	`True`
`is not`	Returns `True` if not same	`x is not y`	`False`

Examples to Try:

1. Arithmetic: Calculate `(5 + 3) * 2 - 4 / 2`.
2. Comparison: Check if `10 > 5` and `5 == 5`.
3. Logical: Combine conditions using `and`, `or`, and `not`.
4. Membership: Check if `"Python"` is in the string `"Learn Python!"`.

Would you like help practicing any of these?

Conditionals

Conditionals in Python

Conditional statements in Python allow you to execute specific blocks of code based on whether a condition is `True` or `False`. These include `if`, `elif`, and `else`.

Syntax

```python
if condition:
    # Code block executed if condition is True
elif another_condition:
    # Code block executed if the above condition is False, but this one is True
else:
    # Code block executed if all conditions are False
```

Example 1: Basic Conditional

```python
x = 10

if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

Logical Conditions

Python supports conditions using operators like:

- **Comparison operators** (`==` , `!=` , `<` , `>` , `<=` , `>=`)
- **Logical operators** (`and` , `or` , `not`)
- **Membership operators** (`in` , `not in`)

Example:

```python
```

```python
x = 10
y = 20

if x < 15 and y > 15:
    print("Both conditions are True")
```

Nested Conditionals

You can nest `if` statements inside other `if` statements.

```python
x = 15

if x > 10:
    if x < 20:
        print("x is between 10 and 20")
    else:
        print("x is 20 or more")
```

Example 2: Checking Even or Odd

```python
number = 7

if number % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Example 3: Using `if-else` with Strings

```python
name = "Alice"

if "A" in name:
    print("Name contains the letter A")
else:
    print("Name does not contain the letter A")
```

Example 4: Ternary Conditional Operator

Python allows you to write a one-liner conditional using a ternary operator.

```python
x = 10
result = "Even" if x % 2 == 0 else "Odd"
print(result)  # Output: Even
```

Common Practice Problems:

1. Write a program to check if a number is positive, negative, or zero.
2. Check if a string is a palindrome (reads the same backward as forward).
3. Write a program to determine the largest of three numbers.

Would you like to work through an example together?

Iterators

Iterators in Python

An **iterator** is an object in Python that allows you to traverse through all the elements in a collection (like a list, tuple, or dictionary) one at a time. Iterators are part of Python's **iterable**

protocol, which consists of two methods:

1. `__iter__()`: Returns the iterator object itself.
2. `__next__()`: Returns the next value from the iterator. When there are no more values, it raises a `StopIteration` exception.

Iterable vs Iterator

- **Iterable**: Any object that can return an iterator. Examples: `list`, `tuple`, `dict`, `set`, `str`.
- **Iterator**: An object with a state so it can remember where it is during iteration.

Example:

```python
# Iterable
my_list = [1, 2, 3]

# Convert iterable to iterator
my_iterator = iter(my_list)

# Using next() to access elements
print(next(my_iterator))  # Output: 1
print(next(my_iterator))  # Output: 2
print(next(my_iterator))  # Output: 3
# print(next(my_iterator))  # Raises StopIteration
```

Using Iterators in Loops

You don't usually call `next()` directly. Instead, Python's `for` loop automatically handles the iteration for you.

Example:

```python
```

```python
my_list = [1, 2, 3]
for item in my_list:
    print(item)
```

Creating Custom Iterators

You can create custom iterator objects by defining the `__iter__()` and `__next__()` methods in a class.

Example: Custom Iterator

python

```python
class MyNumbers:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= self.end:
            num = self.current
            self.current += 1
            return num
        else:
            raise StopIteration

# Using the custom iterator
numbers = MyNumbers(1, 5)
for num in numbers:
    print(num)
```

Generators (Simpler Iterators)

Generators provide a simpler way to create iterators using the `yield` keyword instead of defining `__iter__()` and `__next__()`.

Example: Generator Function

```python
def my_numbers(start, end):
    while start <= end:
        yield start
        start += 1

# Using the generator
for num in my_numbers(1, 5):
    print(num)
```

Built-in Iterators

Python provides several built-in iterators:

1. `range`: Generates a sequence of numbers.

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

2. `enumerate`: Returns the index and value from an iterable.

   ```python
   fruits = ["apple", "banana", "cherry"]
   for index, fruit in enumerate(fruits):
       print(index, fruit)
   ```

3. `zip`: Combines two or more iterables.

   ```python
   names = ["Alice", "Bob"]
   scores = [85, 92]
   ```

```python
for name, score in zip(names, scores):
    print(f"{name} scored {score}")
```

4. `map` **and** `filter`: Apply a function to all items or filter based on a condition.

```python
nums = [1, 2, 3, 4]
squares = map(lambda x: x**2, nums)  # map creates an iterator
print(list(squares))  # [1, 4, 9, 16]
```

Practice Problems:

1. Create an iterator that generates the first `n` Fibonacci numbers.
2. Write a generator that yields all even numbers up to a given limit.
3. Use `zip` to iterate over two lists simultaneously.

Let me know which one you'd like to try or explore further!

Functions

Functions in Python

A **function** in Python is a block of reusable code designed to perform a specific task. Functions make programs modular, easier to read, and efficient to maintain.

Types of Functions in Python

1. **Built-in Functions**: Predefined functions like `print()`, `len()`, `input()`, etc.
2. **User-defined Functions**: Functions you define yourself.
3. **Anonymous Functions**: Functions without a name, created using the `lambda` keyword.

Defining a Function

You define a function using the `def` keyword, followed by the function name and parentheses. Inside the parentheses, you can define parameters (optional).

Syntax:

```python
def function_name(parameters):
    # Code block
    return value  # Optional
```

Example:

```python
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")  # Output: Hello, Alice!
```

Function Parameters

Functions can accept **arguments** or **parameters** to make them dynamic.

Types of Parameters:

1. **Positional Parameters**: Passed in order.

   ```python
   def add(a, b):
       return a + b

   print(add(5, 3))  # Output: 8
   ```

2. **Default Parameters**: Provide a default value if no argument is passed.

   ```python
   ```

```python
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()           # Output: Hello, Guest!
greet("Alice")    # Output: Hello, Alice!
```

3. **Keyword Arguments**: Specify arguments by name.

```python
def introduce(name, age):
    print(f"My name is {name} and I am {age} years old.")

introduce(age=30, name="John")
```

4. **Variable-Length Arguments**:
   - `*args`: For a variable number of positional arguments.

```python
def sum_numbers(*args):
    return sum(args)

print(sum_numbers(1, 2, 3, 4))  # Output: 10
```

   - `**kwargs`: For a variable number of keyword arguments.

```python
def print_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_details(name="Alice", age=25, country="USA")
```

Return Statement

The `return` statement is used to send a value back to the caller. If no `return` is used, the function returns `None` by default.

Example:

```python
def multiply(a, b):
    return a * b

result = multiply(3, 4)
print(result)  # Output: 12
```

Lambda Functions

A **lambda function** is an anonymous, single-line function.

Syntax:

```python
lambda arguments: expression
```

Example:

```python
square = lambda x: x ** 2
print(square(5))  # Output: 25

# Lambda with multiple arguments
add = lambda a, b: a + b
print(add(3, 4))  # Output: 7
```

Scope of Variables

1. **Local Scope**: Variables declared inside a function are local to that function.

2. **Global Scope**: Variables declared outside a function can be accessed globally.
3. **Nonlocal Scope**: Used inside nested functions to refer to variables in the enclosing function's scope.

Example:

```python
x = 10  # Global variable

def outer():
    x = 5  # Nonlocal variable

    def inner():
        nonlocal x
        x += 1
        print(x)  # Output: 6

    inner()
    print(x)  # Output: 6

outer()
print(x)  # Output: 10
```

Practice Problems

1. Write a function to calculate the factorial of a number.
2. Create a function that checks if a string is a palindrome.
3. Write a function that takes a list of numbers and returns their average.
4. Define a lambda function to find the maximum of two numbers.

Let me know which one you'd like to try or explore further!

Lists

Lists in Python

A **list** in Python is a collection of items that is **ordered**, **mutable**, and allows **duplicate elements**. Lists are one of the most commonly used data structures in Python.

Creating a List

Lists are created using square brackets `[]`.

Example:

```python
# Empty list
empty_list = []

# List with elements
fruits = ["apple", "banana", "cherry"]

# Mixed data types
mixed_list = [1, "hello", 3.14, True]
```

Accessing Elements

Indexing:

- Use an index to access a specific item.
- Index starts from `0` (negative indexing starts from `-1` for the last element).

```python
fruits = ["apple", "banana", "cherry"]
print(fruits[0])    # Output: apple
print(fruits[-1])   # Output: cherry
```

Slicing:

- Use slicing to get a subset of the list.

```python
print(fruits[1:3])  # Output: ['banana', 'cherry']
print(fruits[:2])   # Output: ['apple', 'banana']
print(fruits[::2])  # Output: ['apple', 'cherry']
```

Modifying Lists

Adding Elements:

1. `append()`: Add a single element to the end of the list.

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

2. `insert()`: Insert an element at a specific index.

   ```python
   fruits.insert(1, "grape")
   print(fruits)  # Output: ['apple', 'grape', 'banana', 'cherry']
   ```

3. `extend()`: Add multiple elements to the end of the list.

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

Removing Elements:

1. `remove()`: Remove the first occurrence of an element.

   ```python
   fruits.remove("banana")
   print(fruits)  # Output: ['apple', 'grape', 'cherry']
   ```

2. `pop()`: Remove and return the element at a specific index (default is the last element).

```python
fruits.pop()  # Removes the last item
print(fruits)  # Output: ['apple', 'grape']
```

3. `clear()`: Remove all elements.

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

Updating Elements:

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

Common List Methods

Method	Description
`len(list)`	Returns the number of elements in the list.
`sort()`	Sorts the list in ascending order.
`reverse()`	Reverses the order of the list.
`index(item)`	Returns the index of the first occurrence.
`count(item)`	Returns the number of occurrences of an item.

Examples:

```python
numbers = [3, 1, 4, 1, 5, 9]

# Get the length
print(len(numbers))  # Output: 6
```

```python
# Sort the list
numbers.sort()
print(numbers)  # Output: [1, 1, 3, 4, 5, 9]

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

# Find the index of an element
print(numbers.index(4))  # Output: 2

# Count occurrences of an element
print(numbers.count(1))  # Output: 2
```

Iterating Through a List

1. **Using a `for` loop:**

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

2. **Using `enumerate()`:**

   ```python
   for index, fruit in enumerate(fruits):
       print(index, fruit)
   ```

List Comprehensions

A concise way to create lists.

Syntax:

```python
[expression for item in iterable if condition]
```

Examples:

```python
# Create a list of squares
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]

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

Practice Problems

1. Create a list of numbers from 1 to 10 and calculate their squares.
2. Write a program to remove duplicates from a list.
3. Use list comprehension to filter out words shorter than 5 characters from a list of strings.

Let me know which one you'd like to try!

Tuples

Tuples in Python

A **tuple** in Python is an immutable (unchangeable) sequence used to store collections of items. Tuples are similar to lists, but once created, their values cannot be modified.

Characteristics of Tuples

- **Ordered**: Tuples maintain the order of elements.

- **Immutable**: You cannot change, add, or remove items once the tuple is created.
- **Allow duplicates**: Tuples can contain duplicate items.
- **Heterogeneous**: Can contain elements of different data types.

Creating a Tuple

Syntax:

Tuples are created by placing elements inside parentheses `()` separated by commas.

Examples:

```python
# Empty tuple
empty_tuple = ()

# Tuple with elements
fruits = ("apple", "banana", "cherry")

# Tuple without parentheses (optional)
numbers = 1, 2, 3

# Tuple with one element (requires a trailing comma)
single_element = (5,)
print(type(single_element))  # Output: <class 'tuple'>
```

Accessing Tuple Elements

Indexing:

- Use an index to access specific elements.
- Index starts from `0`.

```python
```

```python
fruits = ("apple", "banana", "cherry")
print(fruits[1])  # Output: banana
print(fruits[-1]) # Output: cherry
```

Slicing:

- Extract a subset of the tuple.

```python
print(fruits[0:2])  # Output: ('apple', 'banana')
print(fruits[::-1]) # Output: ('cherry', 'banana', 'apple')
```

Tuple Operations

1. **Concatenation**: Combine two tuples using the `+` operator.

   ```python
   tuple1 = (1, 2, 3)
   tuple2 = (4, 5, 6)
   result = tuple1 + tuple2
   print(result)  # Output: (1, 2, 3, 4, 5, 6)
   ```

2. **Repetition**: Repeat elements using the `*` operator.

   ```python
   repeated = (1, 2) * 3
   print(repeated)  # Output: (1, 2, 1, 2, 1, 2)
   ```

3. **Membership**: Check if an element exists in the tuple using `in` and `not in`.

   ```python
   print("apple" in fruits)  # Output: True
   print("pear" not in fruits)  # Output: True
   ```

Tuple Methods

Method	Description
`count(value)`	Returns the number of times a value appears.
`index(value)`	Returns the index of the first occurrence of a value.

Example:

```python
numbers = (1, 2, 3, 1, 2, 1)

# Count occurrences
print(numbers.count(1))  # Output: 3

# Find index
print(numbers.index(3))  # Output: 2
```

Immutability and Workarounds

Although tuples are immutable, you can:

1. **Convert to a list**: Modify the list, then convert it back to a tuple.

   ```python
   fruits = ("apple", "banana", "cherry")
   fruits_list = list(fruits)
   fruits_list.append("orange")
   fruits = tuple(fruits_list)
   print(fruits)  # Output: ('apple', 'banana', 'cherry', 'orange')
   ```

2. **Concatenate tuples**: Create a new tuple by combining existing ones.

   ```python
   fruits += ("orange",)
   ```

```python
print(fruits)  # Output: ('apple', 'banana', 'cherry', 'orange')
```

Unpacking Tuples

You can unpack the values of a tuple into variables.

```python
fruits = ("apple", "banana", "cherry")
a, b, c = fruits
print(a, b, c)  # Output: apple banana cherry

# Using * to unpack
numbers = (1, 2, 3, 4, 5)
a, *b, c = numbers
print(a, b, c)  # Output: 1 [2, 3, 4] 5
```

Nested Tuples

Tuples can contain other tuples.

```python
nested_tuple = ((1, 2), (3, 4), (5, 6))
print(nested_tuple[1])       # Output: (3, 4)
print(nested_tuple[1][0])    # Output: 3
```

Tuple Use Cases

1. **Storing fixed data**: Tuples are ideal for data that should not be modified (e.g., coordinates, RGB colors).
2. **Returning multiple values**: Functions can return multiple values as a tuple.

```python
def min_max(numbers):
    return min(numbers), max(numbers)

print(min_max([3, 1, 4, 1, 5]))  # Output: (1, 5)
```

Practice Problems

1. Write a function that takes a tuple of numbers and returns a new tuple containing only the even numbers.
2. Create a tuple of student names and unpack them into variables.
3. Write a program to find the second largest element in a tuple of numbers.

Let me know which one you'd like to try!

Sets

Sets in Python

A **set** is an unordered collection of unique items. Sets are commonly used for operations like union, intersection, and difference, and they ensure that no duplicate elements exist.

Characteristics of Sets

- **Unordered**: Sets do not maintain any order.
- **Unique Elements**: Sets automatically remove duplicates.
- **Mutable**: You can add or remove elements.
- **Heterogeneous**: Sets can store different data types.
- **Not Indexable**: You cannot access set elements using an index.

Creating a Set

Sets are defined using curly braces `{}` or the `set()` constructor.

Examples:

```python
# Empty set
empty_set = set()  # Note: {} creates an empty dictionary, not a set.

# Set with elements
fruits = {"apple", "banana", "cherry"}

# Duplicate elements are removed automatically
unique_numbers = {1, 2, 2, 3, 4, 4}
print(unique_numbers)  # Output: {1, 2, 3, 4}
```

Accessing Elements in a Set

Since sets are unordered and unindexed, you cannot access elements using an index.

Instead, you can iterate over the set:

```python
fruits = {"apple", "banana", "cherry"}
for fruit in fruits:
    print(fruit)
```

Adding and Removing Elements

Adding Elements:

1. `add()`: Add a single element.

   ```python
   ```

```python
fruits.add("orange")
print(fruits)  # Output: {'apple', 'banana', 'cherry', 'orange'}
```

2. `update()`: Add multiple elements.

```python
fruits.update(["kiwi", "grape"])
print(fruits)  # Output: {'apple', 'banana', 'cherry', 'kiwi', 'grape'}
```

Removing Elements:

1. `remove()`: Removes an element (raises `KeyError` if the element does not exist).

```python
fruits.remove("banana")
```

2. `discard()`: Removes an element (does not raise an error if the element does not exist).

```python
fruits.discard("pear")
```

3. `pop()`: Removes and returns an arbitrary element.

```python
item = fruits.pop()
print(item)
```

4. `clear()`: Removes all elements from the set.

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

Set Operations

1. **Union (`|` or `union()`)**: Combines all unique elements from both sets.

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

2. **Intersection (`&` or `intersection()`)**: Common elements in both sets.

```python
print(set1 & set2)  # Output: {3}
```

3. **Difference (`-` or `difference()`)**: Elements in one set but not the other.

```python
print(set1 - set2)  # Output: {1, 2}
```

4. **Symmetric Difference (`^` or `symmetric_difference()`)**: Elements in either set, but not both.

```python
print(set1 ^ set2)  # Output: {1, 2, 4, 5}
```

Set Methods

Method	Description
`add(element)`	Adds a single element to the set.
`remove(element)`	Removes an element (raises error if missing).
`discard(element)`	Removes an element (no error if missing).
`pop()`	Removes and returns an arbitrary element.
`clear()`	Removes all elements from the set.
`union(set)`	Returns a new set with elements from both.
`intersection(set)`	Returns a new set with common elements.

Method	Description
`difference(set)`	Returns elements unique to the first set.
`symmetric_difference(set)`	Returns elements in either set but not both.
`issubset(set)`	Checks if a set is a subset of another.
`issuperset(set)`	Checks if a set is a superset of another.
`isdisjoint(set)`	Checks if two sets have no elements in common.

Frozen Sets

A **frozen set** is an immutable version of a set. Once created, you cannot add or remove elements.

Example:

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

# Operations like union, intersection, etc., are allowed
print(frozen.union({4, 5}))  # Output: frozenset({1, 2, 3, 4, 5})
```

Practice Problems

1. Write a program to find the union, intersection, and difference of two sets.
2. Create a set of student names and check if a specific name exists in the set.
3. Use a set to remove duplicate elements from a list.

Let me know if you'd like to try any of these!

Dictionaries

Dictionaries in Python

A **dictionary** in Python is an unordered, mutable collection of key-value pairs. Each key must be unique and immutable, while values can be of any data type and duplicated.

Characteristics of Dictionaries

- **Key-Value Pairs**: Each element is stored as a pair of a key and its corresponding value.
- **Unordered**: Dictionaries do not maintain insertion order before Python 3.7. In Python 3.7+, dictionaries maintain insertion order as an implementation detail (now guaranteed in Python 3.8+).
- **Mutable**: You can add, update, or remove items.
- **Unique Keys**: Duplicate keys are not allowed; if a key is repeated, the last value overwrites the previous one.

Creating a Dictionary

Syntax:

Dictionaries are created using curly braces `{}` or the `dict()` constructor.

Examples:

```python
# Empty dictionary
empty_dict = {}

# Dictionary with elements
person = {"name": "Alice", "age": 25, "city": "New York"}

# Using dict constructor
data = dict(name="Bob", age=30, city="San Francisco")

# Mixed data types as values
mixed_dict = {"name": "Alice", "age": 25, "is_student": False, "grades": [90, 85, 88]}
```

Accessing Dictionary Elements

Using Keys:

Access values by their corresponding keys.

```python
print(person["name"])  # Output: Alice
print(person["age"])   # Output: 25
```

Using `get()`:

- Returns `None` or a default value if the key does not exist.

```python
print(person.get("city"))          # Output: New York
print(person.get("country", "N/A"))  # Output: N/A
```

Adding and Updating Elements

Adding New Key-Value Pairs:

```python
person["country"] = "USA"
print(person)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA'}
```

Updating Existing Keys:

```python
person["age"] = 26
print(person)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

Removing Elements

1. `pop()`: Removes a key and returns its value.

```python
age = person.pop("age")
print(age)  # Output: 26
print(person)  # Output: {'name': 'Alice', 'city': 'New York', 'country': 'USA'}
```

2. `del`: Deletes a key-value pair.

```python
del person["city"]
print(person)  # Output: {'name': 'Alice', 'country': 'USA'}
```

3. `popitem()`: Removes and returns the last inserted key-value pair (useful in Python 3.7+).

```python
last_item = person.popitem()
print(last_item)  # Output: ('country', 'USA')
```

4. `clear()`: Removes all items.

```python
person.clear()
print(person)  # Output: {}
```

Dictionary Methods

Method	Description
`keys()`	Returns a view of all keys in the dictionary.
`values()`	Returns a view of all values.
`items()`	Returns a view of all key-value pairs.

Method	Description
`update(dictionary)`	Updates the dictionary with another dictionary.
`get(key, default)`	Returns the value of a key or a default value.
`pop(key, default)`	Removes a key and returns its value or default.
`setdefault(key, value)`	Returns the value of a key; adds key if missing.

Examples:

```python
person = {"name": "Alice", "age": 25, "city": "New York"}

# View keys, values, and items
print(person.keys())    # Output: dict_keys(['name', 'age', 'city'])
print(person.values())  # Output: dict_values(['Alice', 25, 'New York'])
print(person.items())   # Output: dict_items([('name', 'Alice'), ('age', 25),
('city', 'New York')])

# Update dictionary
person.update({"age": 26, "country": "USA"})
print(person)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country':
'USA'}
```

Iterating Through a Dictionary

Keys:

```python
for key in person:
    print(key)
```

Values:

```python
for value in person.values():
```

```
    print(value)
```

Key-Value Pairs:

```python
for key, value in person.items():
    print(f"{key}: {value}")
```

Nested Dictionaries

Dictionaries can contain other dictionaries.

Example:

```python
students = {
    "student1": {"name": "Alice", "age": 25},
    "student2": {"name": "Bob", "age": 30},
}

print(students["student1"]["name"])  # Output: Alice
```

Dictionary Comprehensions

Syntax:

```python
{key_expression: value_expression for item in iterable if condition}
```

Example:

```python
```

```python
# Create a dictionary with squares of numbers
squares = {x: x**2 for x in range(5)}
print(squares)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Filter even squares
even_squares = {x: x**2 for x in range(5) if x % 2 == 0}
print(even_squares)  # Output: {0: 0, 2: 4, 4: 16}
```

Practice Problems

1. Create a dictionary to store student names as keys and their scores as values. Write a program to find the student with the highest score.
2. Write a program to merge two dictionaries into one.
3. Create a dictionary from two lists: one for keys and one for values.

Let me know if you'd like to try any of these!

Exception handling

Exception Handling in Python

Exception handling in Python allows you to handle runtime errors (exceptions) gracefully without crashing the program. Python uses the `try`, `except`, `else`, and `finally` blocks for handling exceptions.

Key Components

1. `try` **block**: Contains code that might raise an exception.
2. `except` **block**: Catches and handles the exception.
3. `else` **block**: Executes if no exception was raised in the `try` block.
4. `finally` **block**: Always executes, regardless of whether an exception occurred or not.

Basic Syntax

```python
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
else:
    # Code to run if no exception occurs
finally:
    # Code that always runs (clean-up code)
```

Example: Handling a Division by Zero Error

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
else:
    print("Division successful.")
finally:
    print("Execution completed.")
```

Output:

```vbnet
Error: Division by zero is not allowed.
Execution completed.
```

Common Exception Types

Here are some common built-in exceptions in Python:

- `ZeroDivisionError`: Raised when dividing by zero.
- `ValueError`: Raised when a function receives an argument of the right type but inappropriate value.
- `TypeError`: Raised when an operation is performed on an object of inappropriate type.
- `FileNotFoundError`: Raised when trying to open a file that doesn't exist.
- `IndexError`: Raised when trying to access an index that is out of range in a list.
- `KeyError`: Raised when trying to access a key that doesn't exist in a dictionary.
- `NameError`: Raised when a local or global name is not found.

Example: Handling Multiple Exceptions

You can handle different types of exceptions in separate `except` blocks.

python

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed.")
```

Output (if input is `0`):

vbnet

```
Error: Cannot divide by zero.
Execution completed.
```

Catching Multiple Exceptions in One Block

You can also catch multiple exceptions in a single `except` block by specifying a tuple of exception types.

```python
try:
    value = input("Enter a number: ")
    result = 10 / int(value)
except (ValueError, ZeroDivisionError) as e:
    print(f"Error occurred: {e}")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed.")
```

Raising Exceptions

You can raise exceptions manually using the `raise` keyword.

```python
def check_positive_number(num):
    if num < 0:
        raise ValueError("Number must be positive.")
    else:
        return num

try:
    check_positive_number(-5)
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```javascript
Error: Number must be positive.
```

Custom Exception Classes

You can define your own exception classes by subclassing the built-in `Exception` class.

```python
class NegativeNumberError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

def check_positive_number(num):
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")
    else:
        return num

try:
    check_positive_number(-5)
except NegativeNumberError as e:
    print(f"Custom Error: {e}")
```

Output:

```vbnet
Custom Error: Negative numbers are not allowed.
```

Best Practices for Exception Handling

1. **Catch Specific Exceptions**: Always catch specific exceptions rather than using a generic `except Exception:` block, which can hide errors.
2. **Keep `try` Block Short**: Only include the code that might raise an exception in the `try` block, to avoid catching unexpected exceptions.
3. **Use `finally` for Clean-Up**: The `finally` block is useful for closing files, releasing resources, or other clean-up actions, ensuring they are executed regardless of errors.

4. **Logging**: For debugging purposes, consider logging exceptions instead of printing them directly.

Practice Problems

1. Write a program that asks the user for a number and handles cases where the input is not a valid number or division by zero occurs.
2. Create a function that opens a file, reads its content, and handles cases where the file does not exist or cannot be read.
3. Write a custom exception class to handle situations where a user enters an invalid email format.

Let me know if you'd like to try any of these!

Object Oriented Programming

Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, rather than functions or logic. In Python, everything is an object, and OOP allows you to model real-world entities using classes and objects.

OOP in Python is based on the following key principles:

1. **Classes and Objects**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**
5. **Abstraction**

---

1. Classes and Objects

- **Class**: A blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects will have.
- **Object**: An instance of a class. Each object has its own unique attributes and behaviors.

Syntax for Creating a Class:

python

```python
class ClassName:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    def method(self):
        print("This is a method")
```

Example:

python

```python
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Car Info: {self.year} {self.brand} {self.model}")

# Creating an object (instance) of the Car class
my_car = Car("Toyota", "Corolla", 2020)

# Accessing the object's attributes and methods
print(my_car.brand)  # Output: Toyota
my_car.display_info()  # Output: Car Info: 2020 Toyota Corolla
```

---

2. Encapsulation

Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on that data within a single unit or class. It also refers to restricting access to some of an object's components, which is usually done by marking attributes as private.

- **Private Attributes**: In Python, private attributes are indicated by a double underscore (`__`) before the attribute name.
- **Public Methods**: These are methods that can be accessed by any object.

Example:

python

```python
class Person:
    def __init__(self, name, age):
        self.name = name        # public attribute
        self.__age = age        # private attribute

    def greet(self):
        print(f"Hello, my name is {self.name}.")

    def get_age(self):
        return self.__age  # Accessing the private attribute using a public method

    def set_age(self, age):
        if age > 0:
            self.__age = age  # Modifying the private attribute using a public
method
        else:
            print("Invalid age!")

# Creating an object of Person
person = Person("Alice", 30)
person.greet()

# Accessing private attribute via method
print(person.get_age())  # Output: 30

# Modifying private attribute via method
person.set_age(35)
print(person.get_age())  # Output: 35
```

3. Inheritance

Inheritance allows a class (called a **child class** or **subclass**) to inherit attributes and methods from another class (called a **parent class** or **superclass**). This allows for code reuse and extension of functionality.

Syntax for Inheritance:

```python
class ParentClass:
    def method1(self):
        print("This is a method from the parent class")

class ChildClass(ParentClass):
    def method2(self):
        print("This is a method from the child class")
```

Example:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")

class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows.")

# Creating objects of Dog and Cat
dog = Dog("Buddy")
cat = Cat("Whiskers")
```

```
# Calling the overridden methods
dog.speak()  # Output: Buddy barks.
cat.speak()  # Output: Whiskers meows.
```

- In the example above, both `Dog` and `Cat` inherit from the `Animal` class but override the `speak()` method to provide their own implementation.

---

4. Polymorphism

Polymorphism means "many shapes." It allows different classes to have methods with the same name, but each method can behave differently based on the object calling it.

Example:

python

```python
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

# Polymorphism in action
shapes = [Circle(5), Square(4)]

for shape in shapes:
    print(f"Area: {shape.area()}")
```

Output:

```makefile
Area: 78.5
Area: 16
```

- In the example above, both `Circle` and `Square` implement the `area()` method, but each class calculates the area differently.

---

5. Abstraction

Abstraction hides the complex implementation details and shows only the essential features of an object. In Python, abstraction can be implemented using abstract classes and methods.

- **Abstract Class**: A class that cannot be instantiated and typically contains abstract methods (methods that are declared but contain no implementation).
- **Abstract Method**: A method that is defined in an abstract class but is meant to be implemented by subclasses.

Example:

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Creating an object of Circle
```

```
circle = Circle(5)
print(f"Area of Circle: {circle.area()}")  # Output: 78.5

# shape = Shape()  # This will raise an error because Shape is abstract
```

- **Note**: You cannot create an instance of `Shape` directly, as it contains abstract methods. Subclasses like `Circle` must implement the abstract methods.

Summary of OOP Concepts

1. **Classes and Objects**: Define blueprints for objects.
2. **Encapsulation**: Bundle data and methods, and restrict access to internal components.
3. **Inheritance**: Create new classes based on existing ones, promoting code reuse.
4. **Polymorphism**: Allow methods with the same name to behave differently across classes.
5. **Abstraction**: Hide implementation details and only expose essential functionality.

Practice Problems

1. Create a `BankAccount` class with methods to deposit, withdraw, and check the balance. Implement inheritance to create a `SavingsAccount` class with interest.
2. Write a `Shape` class and implement subclasses like `Rectangle` and `Circle`. Add a method to calculate area.
3. Create a class hierarchy for `Vehicle`, `Car`, and `Truck`. Add attributes like speed and fuel capacity, and implement methods to accelerate and brake.

Let me know if you'd like to try any of these or need further clarification on any concept!