Mastering Python: Essential Concepts for Developers

▸ Table of Contents

Introduction

In the world of programming, countless languages are available, each with its unique strengths and uses. Among these, Python has gained remarkable popularity as a versatile and beginner-friendly language, known for its simplicity, readability, and wide range of applications. Whether you're a seasoned developer or a beginner taking your first steps into programming, Python offers an excellent starting point. Whether you're building a web app, analyzing data, or diving into artificial intelligence, Python provides an amazing foundation.

What is Python?

- **Python** is a high-level, interpreted programming language created by **Guido van Rossum** and first released in **1991**. Its design philosophy emphasizes code readability and simplicity, making it ideal for beginners.
- Python supports multiple programming paradigms, including **object-oriented**, **imperative**, and **functional** programming approaches.
- It comes with an extensive **standard library**, and its **third-party ecosystem** (with packages like NumPy, Pandas, and TensorFlow) provides tools for everything from data analysis to web development.
- Python is dynamically typed and garbage-collected, which makes coding faster and more flexible.

Why Choose Python?

- **Readable Syntax:** Python's syntax is clear, concise, and often described as "English-like." This makes the code easy to read, write, and maintain, making it an ideal first language for beginners.

- **Interpreted Language:** Python is an interpreted language, meaning that the Python interpreter executes the code line by line, rather than compiling it into machine code first. This makes Python great for rapid prototyping and quick testing.
- **Open-Source and Community-Driven:** Python is open-source, which means it's free to use and modify. It also has a large, active community that continuously contributes to the language's development and resources, ensuring that Python stays up-to-date with industry needs.
- **Extensive Library Support:** Python provides an impressive array of built-in libraries, allowing you to accomplish a wide variety of tasks with minimal code. Popular libraries include:
  - **NumPy** (for numerical computing)
  - **Pandas** (for data manipulation and analysis)
  - **Matplotlib** (for data visualization)
  - **TensorFlow** and **PyTorch** (for machine learning)
  - **Flask** and **Django** (for web development)
- **Versatile Applications:** Python can be used in virtually every field of software development:
  - **Web Development:** Build dynamic websites and web applications using frameworks like Flask and Django.
  - **Data Science & Machine Learning:** Python is a popular choice for data analysis, machine learning, and artificial intelligence projects.
  - **Automation & Scripting:** Python excels at automating repetitive tasks, from file handling to system administration.
  - **Game Development, Robotics, and more:** Python's simplicity and versatility make it a strong candidate for a wide range of applications.

Getting Started with Python

To start coding in Python, you'll need to set up a development environment. Here's a step-by-step guide to help you get started:

1. **Installing Python:**

- Visit the official Python website: python.org.
- Download the latest version of Python for your operating system (Windows, macOS, or Linux).
- Follow the installation instructions:

2. **Choosing an Integrated Development Environment (IDE):**

While you can write Python code using a simple text editor, an **IDE (Integrated Development Environment)** can help by providing features like syntax highlighting, debugging, and code completion. Popular Python IDEs include:

- **PyCharm**: A powerful IDE with excellent support for Python.
- **Visual Studio Code (VS Code)**: A lightweight, flexible IDE with great Python support through extensions.
- **IDLE**: Python's default IDE, which is simple but works well for beginners.

You can choose any of these based on your personal preference and the complexity of your projects.

3. **Writing Your First Python Program:**

Let's write your very first Python program. Follow these steps:

- Open your IDE

- Create a new Python file. Make sure to save the file with the `.py` extension (for example, `hello_world.py`).
- In the file, type the following code:

```
print("Hello, World!")
```

This is a simple Python statement that prints the message "Hello, World!" to the console.

- Save the file.

4. **Running Your Python Program:** Open a terminal or command prompt, navigate to the directory where the file is located using the `cd` command, and type the following command:

```
python hello_world.py
```

If everything is set up correctly, you will see the output `Hello, World!` printed on the terminal.

Congratulations! You've just executed your first Python program. 🎉

Modules in Python

In Python, **modules** are essentially code libraries containing functions, variables, and classes that you can use in your Python programs without having to rewrite them. This makes your code more **organized**, **efficient**, and **reusable**.

Modules can be classified into two types:

- Built-in modules
- External modules

Built-in Modules

These modules are pre-installed in python, you don't need to install them. You can just simply import and use them into your code. Some examples of built-in modules are:

- `random` (for random number generation)
- `datetime` (for working with dates and times)
- `math` (for mathematical operations)

Using a built-in module:

To use a built-in module, you need to import it at the beginning of your python script.
Here's an example of using the `math` module:

```
import math result = math.sqrt(9) print(result) # Output: 3.0
```

In this example:

- We import the `math` module.

- Then, we use the `sqrt()` function to calculate the square root of 9.

External Modules

These modules are third-party modules or libraries that are not included in the Python standard library. These need to be installed separately using a package manager such as `pip` or `conda`. Some examples of external modules are:

- `pandas` (for data manipulation and analysis)
- `numpy` (for numerical computations)
- `tensorflow` (for machine learning)

Using an external module

To use an external module, you first need to install it using a package manager. Open your terminal or command prompt and run the following command:

```
pip install pandas
```

After installation, you can import and use the module in your Python script. Here's an example of using `pandas` to read a JSON file:

```
import pandas # Reading data from a json file df = pandas.read_json('data.json') # Make
sure the file exists in your project directory print(df)
```

In this example:

- We install and import the `pandas` library.
- We use the `read_json()` function to read a JSON file (`data.json`), which should be in your project directory for this code to work.

Let's take another example.

Open your terminal or command prompt and run the following command:

```
pip install numpy
```

After installation, you can import and use the module in your Python script as follows:

```
import numpy # Using numpy to create an array arr = numpy.array([1, 2, 3, 4]) print(arr)
# Output: [1 2 3 4]
```

Similarly you can use other modules as well and go through their documentations for usage instructions.

Comments in Python

If you want to write your code just to explain a block of code or to avoid the execution of a specific part of code while testing, then python comments comes in handy.

Single-line comment

To write a single-line comment just add a `#` at the start of the line.

```
# This is a single-line comment print("Hello World!")
```

The line starting with `#` is ignored by the Python interpreter. It's only there for understanding.

Multi-Line Comment

In Python, there are two ways to write multi-line comments:

1. **Using `#` on each line**:

You can write comments over multiple lines by starting each line with the `#` symbol.

```
# This is a comment # that spans multiple # lines print("Hello, World!")
```

2. **Using a multi-line string** (triple quotes):

While multi-line strings (triple quotes: `"""` or `'''`) are typically used for docstrings (documentation strings), they can also be used to create multi-line comments, as Python will ignore them if they're not assigned to a variable or used in any other way.

```
""" This is a multi-line comment using a string. Python will ignore this string. """ prin
t("Hello, World!")
```

Best Practices for Comments

- **Use comments to explain "why" rather than "what"**:
  Use comments to explain the **reasoning** behind your code, especially for more complex sections.

```
# Using a while loop because we need to repeat the task until the condition is met while
counter < 10: counter += 1
```

- **Avoid over-commenting**:

Don't add comments for every single line of code. Your code itself should be as readable as possible.

```
# Bad practice counter = 10 # Set counter to 10 result = counter * 2 # Multiply counter
by 2
```

Instead, just write clear code:

```
counter = 10 result = counter * 2 # This line is already clear
```

Python `print()` Function

In Python, the print() function is used to display output on the console. It is one of the most commonly used functions in Python for debugging and outputting information.

For example:

```
print("Hello World!") # Output: Hello World!
```

In this example, the `print()` function displays the string `"Hello, World!"` on the console.

Syntax

```
print(object(s), sep=separator, end=end, file=file, flush=flush)
```

Let's break down the parameters of the `print()` function:

1. **object(s):**

The object(s) to be printed. You can pass any number of objects (strings, integers, lists, etc.), and they will be converted to strings before being printed.

For example:

```
print("Hello", 42) # Output: Hello 42
```

2. **sep='separator':**

Specifies how to separate the objects, if there is more than one. The default separator is a space `' '`. You can change this to any string. (This is optional)

For example:

```
print("Hello", "World", sep=", ") # Output: Hello, World
```

3. **end='end':**

Specifies what to print at the end. By default, it is a newline character (`\n`), meaning the next output will be printed on a new line. You can change this to any string. (This is optional)

For example:

```
print("Hello", "World", end="!") # Output: Hello World!
```

4. **file:**

An object with a `write()` method (such as a file object). The default is `sys.stdout`, meaning output will be printed to the console. (This is optional)

For example:

```
# writing to a file with open("output.txt", "w") as file: print("Hello, World!",
file=file)
```

5. **flush:**

A Boolean value, specifying if the output is flushed (True) or buffered (False). Default is False. (This is optional)

For example:

```
print("Hello", flush=True)
```

This will flush the output buffer, meaning that it will be printed to the console immediately.

Now, let's take a look at the example using `sep` and `end` parameters.

```
print("Hello", "World", sep=", ", end="!\n") # Output: Hello, World!
```

In this example:

- The `sep` parameter places a comma and a space between `"Hello"` and `"World"`.
- The `end` parameter changes the default newline to an exclamation mark followed by a newline.

Python Escape Sequence Characters

To insert characters that cannot be used directly in a string, you use an escape character. An example of a such character is a double quote inside a string that is surrounded by double quotes.

```
print("This will give "error")
```

Escape sequence characters are represented with a backslash (\) followed by a specific character or combination of characters.

```
print("This will run without any \" error") #output: This will run without any " error
```

Common Escape Sequences in Python

1. `\\` - **Backslash:** The backslash itself is an escape character, so to print a literal backslash, you need to escape it with another backslash.

```
print("This is a backslash: \\") #output: This is a backslash: \
```

2. `\'` - **Single Quote:** This allows you to insert a single quote inside a string that is enclosed by single quotes.

```
print('This is a \'single quote\'') #output: This is a 'single quote'
```

3. `\"` - **Double Quote:** Similar to the single quote escape sequence, this allows you to insert double quotes inside a string that is surrounded by double quotes.

```
print("This is a \"double quote\"") #output: This is a "double quote"
```

4. `\n` - **Newline:** Inserts a newline, causing the text after it to appear on a new line.

```
print("Line 1\nLine 2") #output: #Line 1 #Line 2
```

5. `\t` - **Tab:** Inserts a tab space, which is often used for indentation.

```
print("First\tSecond") #output: First Second
```

6. `\\b` - **Backspace:** Removes the character immediately before it.

```
print("Hello\bWorld") #output: HellWorld
```

7. `\r` - **Carriage Return:** Moves the cursor back to the beginning of the line.

```
print("Hello\rWorld") # Output: World # The \r escape sequence moves the cursor to the be
ginning of the line, so the text "World" overwrites the text "Hello."
```

Variables in Python

In Python, **variables** are used to store data values. They act as containers that hold data, which can be of various types such as integers, strings, lists, etc.

For example:

```
name = "John" # name is a variable that stores the value "John" print(name) # The
variable can be used in functions, print statements and more. #output: John
```

Variable naming rules

When naming variables in Python, there are a few important rules and guidelines to follow:

- Variable names must start with a letter or underscore.

```
_name = "Alice" # Valid name1 = "Bob" # Valid
```

- They can contain letters, digits, and underscores.

```
name_1 = "Charlie" # Valid age_30 = 30 # Valid
```

- Variable names are case-sensitive (`a` and `A` are considered different variables).

```
a = 5 A = 19 sum = a + A print("The sum is:", sum) # Output: The sum is: 24
```

- There are specific keywords in Python (such as `if`, `else`, `for`, `while`, etc.) that cannot be used as variable names because they have special meanings in the language.

```
# Invalid variable names # if = 10 # SyntaxError: cannot assign to keyword # for = 5 #
SyntaxError: cannot assign to keyword
```

Variable Assignment and Types

Variables in Python can store values of different data types. Python automatically determines the type of a variable based on the assigned value. This is known as **dynamic typing**.

Example:

```
x = 10 # x is an integer y = "Hello" # y is a string z = [1, 2, 3] # z is a list
```

Python allows you to assign new values of different types to variables without any issue:

```
x = "Now I'm a string" # x was initially an integer, but now it's a string
```

Multiple Variable Assignment

You can also assign values to multiple variables in a single line as follows:

```
x, y, z = 10, "Hello", [1, 2, 3] print(x) # Output: 10 print(y) # Output: Hello print(z)
# Output: [1, 2, 3]
```

Best Practices for Naming Variables

- **Be descriptive**: Choose meaningful names that describe the purpose of the variable. For example, use `age` instead of `a`, and `first_name` instead of `fn`.
- **Use underscores for readability**: If your variable name has more than one word, use underscores to separate them. This follows the **PEP 8** style guide.

```
user_age = 30 first_name = "John"
```

Data Types in Python

Data types specifies the type of value that a variable can hold. Python is a dynamically-typed language, meaning that you don't need to declare a variable's type explicitly; Python infers the type based on the assigned value.

To determine the type of a variable, you can use the built-in `type()` function.

Python's Built-in Data Types

Python provides the following built-in data types:

- **Text type:** `str`
  The **string** (`str`) data type is used for textual data. You can create a string by enclosing characters within single or double quotes.

```
a = "Hello World!" print(type(a)) #output: <class 'str'>
```

- **Numeric type:** `int`, `float`, `complex`

Python provides three types of numeric values:

`int`: Integer numbers (positive or negative).

`float`: Floating-point numbers (decimal numbers).

`complex`: Complex numbers, with a real and imaginary part.

```
a = 5 #int b = 5.5 #float c = 5j #complex print(type(a)) # Output: <class 'int'>
print(type(b)) # Output: <class 'float'> print(type(c)) # Output: <class 'complex'>
```

- **Boolean type:** `bool`

The **boolean** (`bool`) data type is used to represent one of two possible values: `True` or `False`. It is commonly used in conditions and loops.

```
a = True print(type(a)) #output: <class 'bool'>
```

- **Sequence type:** `list`, `tuple`, `range`

Python has three built-in sequence types:

`list` : Ordered, mutable collections of items (can hold different data types).

`tuple` : Ordered, immutable collections of items.

`range` : Represents a sequence of numbers, often used in for-loops.

We will explore these data types in much more detail as we progress through the notes.

```
a = ["one", "two", "three"] #list b = ("one", "two", "three") #tuple c = range(5) #range
print(type(a)) # Output: <class 'list'> print(type(b)) # Output: <class 'tuple'>
print(type(c)) # Output: <class 'range'>
```

- **Set type:** `set` , `frozenset`

Sets are unordered collections of unique items.

`set` : Mutable set.

`frozenset` : Immutable set.

```
a = {"one", "two", "three"} #set b = frozenset({"one", "two", "three"}) #frozenset
print(type(a)) # Output: <class 'set'> print(type(b)) # Output: <class 'frozenset'>
```

- **Mapping type:** `dict`

A **dictionary** (`dict`) is an unordered collection of key-value pairs. The keys must be unique, and the values can be any data type.

```
a = {"name" : "John", "age" : 25} print(type(a)) # Output: <class 'dict'>
```

- **None type:** `NoneType`

The `None` type represents the absence of a value or a null value. It is commonly used to signify that a variable has no value assigned.

```
a = None print(type(a)) # Output: <class 'NoneType'>
```

Summary of Data Types in Python

Data Type	Description	Example
`str`	String type for textual data	`"Hello World!"`
`int`	Integer type for whole numbers	`5`
`float`	Float type for decimal numbers	`5.5`
`complex`	Complex number type (real + imaginary part)	`5j`
`bool`	Boolean type, represents True or False	`True` or `False`
`list`	Mutable sequence of items	`["one", "two", "three"]`
`tuple`	Immutable sequence of items	`("one", "two", "three")`
`range`	Sequence of numbers (usually used in loops)	`range(5)`
`set`	Unordered collection of unique items	`{"one", "two", "three"}`
`frozenset`	Immutable set	`frozenset({"one", "two", "three"})`
`dict`	Key-value pair collection	`{"name": "John", "age": 25}`
`NoneType`	Represents no value (None)	`None`

Typecasting in Python

The conversion of one data type into the other data type, to perform the required operation, is known as typecasting or type conversion.

There are two types of typecasting in Python:

- Explicit Type Conversion
- Implicit Type Conversion

Explicit Type Conversion

Manually conversion of one data type into another data type, is known as explicit type conversion. This is typically done when you need to perform operations that require a specific data type.

It can be done using Python's built-in type conversion functions such as:

- `int()` — converts to integer
- `float()` — converts to float
- `str()` — converts to string
- `bool()` — converts to boolean, and so on.

For example:

```
x = 5 print(type(x)) #output: <class 'int'> y = str(x) # Explicitly converts integer to
string print(type(y)) #output: <class 'str'>
```

Implicit Type Conversion

Implicit type conversion, happens automatically by the Python interpreter when it needs to perform operations with different data types.

Python automatically converts the lower data type into a higher data type to avoid data loss and ensure the operation is performed without errors.

For example:

```
x = 3 print(type(x)) #output: <class 'int'> y = 7.1 print(type(y)) #output: <class
'float'> z = x + y print(z) print(type(z)) #output: <class 'float'> # Python
automatically converts z to float as it is a float addition
```

In this example, the integer `x` is automatically converted to a float before the addition operation, resulting in a float.

Input Function in Python

In python, the `input()` function is used to take input from the user.

Syntax for `input()` function

```
variable = input(prompt)
```

- `prompt` (optional): A string that is displayed to the user before taking input. If not provided, the function will simply wait for user input without displaying any prompt.
- `variable` : The variable where the user's input is stored.

For example:

```
name = input("Enter your name: ") print("Hello "+ name) # Example output: # Enter your
name: John # Hello John
```

- The `input()` function prompts the user with "Enter your name: ".
- The user types something, and that input is stored as a string in the `name` variable.
- The `print()` function then outputs the message "Hello [name]" to the console.

Note: `input()` returns a string:

Even if the user enters a number or other types of data, it will be treated as a string. If you need the input to be a different type (like an integer or float), you'll need to convert it explicitly using functions like `int()` or `float()`.

For example:

```
age = input("Enter your age: ") # Input is stored as string by default age = int(age) #
Convert string to integer print("Your age is:", age) # Example output: # Enter your age:
25 # Your age is: 25
```

Strings in Python

A string is a sequence of characters enclosed between single or double quotation marks. Strings are one of the most common data types in Python and can contain letters, numbers, symbols, or even spaces.

Creating Strings

```
name = "John Doe" print(name) message = "Hello, World!" print(message)
```

If you want to print a statement with quotation marks in between the strings. Then you can use single quotes for convenience. For example:

```
print('He said, "I love Python programming"!')
```

Multiline Strings

You can create multiline strings by using triple single or triple double quotes. For example:

```
string = '''Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua.''' print(string) string1 = """Lorem
ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua.""" print(string1)
```

Accessing Characters in a String

String is like an array of characters in python. You can access characters of string by using its index which starts from 0.

```
message = "Hello, World!" print(message[0]) #output: H print(message[1]) #output: e
print(message[5]) #output: ,
```

Length of a String

You can use the `len()` function to find the length of a string.

```
message = "Hello, World!" messageLength = len(message) print(messageLength) #Output: 13
```

String Slicing

String is like an array of characters, so you can extract a part of a string using slicing.

```
message = "Hello, World!" # Extract characters from index 0 to index 5 (including 0 but
not 5) print(message[0:5]) # Output: "Hello" # Omitting start index (defaults to 0)
(including 0 but not 5) print(message[:5]) # Output: "Hello" # Extract characters from
index 3 to the end print(message[3:]) # Output: "lo, World!" # Slicing using negative
index print(message[0:-3]) # Output: "Hello, Wor" # Python interprets above code as
print(message[0:len(message)-3]) print(message[-1:]) # Output: "!" # Python interprets
above code as print(message[len(message)-1:])
```

Looping Through a String

Strings are iterable, so you can loop through each character in the string.

```
message = "Hello, World!" for i in message: print(i)
```

String Methods

Python provides several built-in methods for working with strings:

- `upper()` : The `upper()` method converts a string to upper case.

```
message = "Hello, World!" print(message.upper()) #Output: HELLO, WORLD!
```

- `lower()` : The `lower()` method converts a string to lower case.

```
message = "Hello, World!" print(message.lower()) #Output: hello, world!
```

- `capitalize()` : The `capitalize()` method converts the first character of the string to uppercase and the rest other characters of the string to lowercase.

```
message = "hello, World!" print(message.capitalize()) #Output: Hello, world!
```

- `split()` : The `split()` method splits the given string and returns the separated strings as list items.

```
message = "Hello, World!" print(message.split(" ")) #Output: ['Hello,', 'World!']
```

- `strip()` : The `strip()` method removes white spaces at the beginning and end of the string.

```
message = " Hello, World! " print(message.strip()) #Output: Hello, World!
```

- `rstrip()` : The `rstrip()` removes any trailing characters.

```
message = " Hello, World! " print(message.rstrip("!")) #Output: Hello, World
```

- `replace()` : The `replace()` method replaces all occurrences of a string with another string.

```
message = "Hello, World!" print(message.replace("Hello", "Hi")) #Output: Hi, World!
```

- `center()` : The `center()` method aligns the string to the center as per the parameters given by the user.

```
message = "Hello, World!" print(message.center(50)) # Output: " Hello, World! "
```

You can also provide padding character.

```
message = "Hello, World!" print(message.center(50,"!")) #Output:!!!!!!!!!!!!!!!!!Hello,
World!!!!!!!!!!!!!!!!!!!!
```

- `count()` : The `count()` method returns the number of times the given value has occurred within the given string.

```
message = "Hello, World!" print(message.count("l")) #Output: 3
```

- `title()` : The `title()` method capitalizes first letter of each word within the string.

```
message = "Python is a programming language" print(message.title()) #Output: Python Is A
Programming Language
```

- `startswith()` : The `startswith()` method checks if the string starts with a given value. It gives result in `true` or `false`.

```
message = "Hello, World!" print(message.startswith("Hello")) #Output: True
```

- `endswith()` : The `endswith()` method checks if the string ends with a given value. It gives result in `true` or `false`.

```
message = "Hello, World!" print(message.endswith("!")) #Output: True
```

- `find()` : The `find()` method searches for the first occurrence of the given value and returns the index where it is present. If given value is absent from the string then return `-1`.

```
message = "Hello, World!" print(message.find("W")) #Output: 7 print(message.find("a"))
#Output: -1
```

- `index()` : The `index()` method searches for the first occurrence of the given value and returns the index where it is present. If given value is absent from the string then raise an exception.

```
message = "Hello, World!" print(message.index("W")) #Output: 7 print(message.index("a"))
#Output: ValueError: substring not found
```

This method is somewhat similar to the `find()` method. The major difference being that `index()` raises an exception if value is absent whereas `find()` **return -1.**

- `isalnum()` : The `isalnum()` method returns `True` if the string consists only of A-Z, a-z, 0-9. If any other characters or punctuations are present, then it returns `False`.

```
message = "Hello, World!" print(message.isalnum()) #Output: False message2 = "HelloWorld"
print(message2.isalnum()) #Output: True
```

- `isalpha()` : The `isalpha()` method returns `True` if the string consists only of A-Z, a-z. If any other characters or punctuations or numbers(0-9) are present, then it returns `False`.

```
message = "Hello, World!" print(message.isalpha()) #Output: False message2 = "HelloWorld"
print(message2.isalpha()) #Output: True
```

- `isupper()` : The `isupper()` method returns `True` if all the characters in the string are upper case, else it returns `False.`

```
message = "Hello, World!" print(message.isupper()) #Output: False
```

- `islower()` : The `islower()` method returns `True` if all the characters in the string are lower case, else it returns `False`.

```
message = "hello, world!" print(message.islower()) #Output: True
```

- `isprintable()` : The `isprintable()` method returns `True` if all the values within the given string are printable (i.e., not whitespace, control characters, etc.), if not, then return `False`.

```
message = "hello, world!" print(message.isprintable()) #Output: True
```

- `isspace()` : The `isspace()` method returns `True` if the string consists white spaces, else returns `False`.

```
message = " " print(message.isspace()) #Output: True
```

- `istitle()` : The `istitle()` method returns `True` if the first letter of each word of the string is capitalized, else it returns `False`.

```
message = "World Health Organization" print(message.istitle()) #Output: True
```

- `swapcase()` : The `swapcase()` method changes the character casing of the string. Upper case are converted to lower case and lower case to upper case.

```
message = "Hello, World!" print(message.swapcase()) #Output: hELLO, wORLD!
```

Conditional Statements

In Python, conditional statements are used to make decisions in the code based on certain conditions. They allow us to execute different blocks of code depending on whether a specific condition evaluates to True or False.

Types of Conditional Statements in Python

Python has following conditional statements:

`if-else` statement:

If the condition in the `if` statement evaluates to true, the code inside the `if` block will be executed; otherwise, the code within the `else` block will be executed.

Syntax:

```
if condition: # Code to execute if condition is True else: # Code to execute if condition
is False
```

Example:

```
num = int(input("Enter a number: ")) if num < 0: print("Number is negative.") else:
print("Number is positive.")
```

`elif` statement:

The `elif` statement (short for "else if") is used when there are multiple conditions to evaluate. It is placed between `if` and `else`.

Syntax:

```
if condition1: # Code to execute if condition1 is True elif condition2: # Code to execute
if condition2 is True else: # Code to execute if all above conditions are False
```

Example:

```
num = int(input("Enter a number: ")) if num < 0: print("Number is negative.") elif num ==
0: print("Number is zero.") else: print("Number is positive.")
```

Nested `if` statements:

`if` statements can be nested within other `if`, `elif`, or `else` blocks to evaluate additional conditions.

Syntax:

```
if condition1: # Code to execute if condition1 is True if condition2: # Code to execute
if condition2 is True elif condition3: # Code to execute if condition3 is True else: #
Code to execute if all inner conditions are False else: # Code to execute if condition1
is False
```

Example:

```
num = int(input("Enter a number: ")) if num < 0: print("Number is negative.") elif num >
0: if num <= 50: print("Number is between 1 and 50.") elif num > 50 and num <= 70:
print("Number is between 51 and 70.") else: print("Number is greater than 70.") else:
print("Number is zero.")
```

Key Points About Conditional Statements

1. **Indentation Matters**:
    - Python uses indentation (spaces or tabs) to show which code belongs to a condition.
    - Make sure all lines in a block have the same indentation.
2. **Only One `else`**:
    - You can use many `if` and `elif` statements, but only one `else` is allowed.
3. **Comparison and Logical Operators**:
    - Conditions use operators like:
        - `==` (equal), `!=` (not equal), `<` (less than), `>` (greater than).
        - `and`, `or`, `not` to combine conditions.
4. **Order of Checks**:
    - Python checks conditions one by one. If a condition is true, the following conditions are skipped.

Match Case Statements

The `match` statement is a feature introduced in Python 3.10, enabling a clean and powerful way to perform pattern matching. It is similar to a switch-case statement found in other programming languages but more versatile.

The `match` statement evaluates the value of a variable (or expression) against multiple patterns defined in `case` blocks. When a match is found, the corresponding block of code is executed. If no match is found, a default case (`case _`) can be used.

Syntax:

```
match variable: case pattern1: # Code to execute if pattern1 matches case pattern2: # Code to execute if pattern2 matches ... case _: # Code to execute if no patterns match (default case)
```

Example:

```
x = 4 # x is the variable to match match x: case 0: print("x is zero") # This will execute if x == 0 case 4: print("x is 4") # This will execute if x == 4 case _ if x < 10: print("x is less than 10") # This will execute for x < 10 case _: print("No match found") # Default case # Output: x is 4
```

Advantages of Match Case

- **Readable Code**:

  It is more readable and concise compared to multiple `if-elif-else` statements.
- **Flexible Matching**:

  It supports both value and condition-based matching, making it highly versatile.
- **Default Case Handling**:

  The `_` wildcard provides a straightforward way to handle unmatched cases.

Loops in Python

Loops are used to execute a block of code repeatedly. For example, If you have to print the numbers from 1 to 5, you can do it very easily. But, when printing the numbers from 1 to 1000, writing so much code could be hectic. This can be done easily using loops.

The `for` loop

This loop iterates over sequences such as strings, lists, tuples, sets and dictionaries.

Example 1: iterating over a string

```
name = 'Python' for i in name: print(i)
```

Output:

```
P y t h o n
```

Example 2: iterating over a list

```
colors = ["Red", "Green", "Blue", "Yellow"] for color in colors: print(color)
```

Output:

```
Red Green Blue Yellow
```

You can also use `for` loop inside a `for` loop. For example:

```
colors = ["Red", "Green", "Blue", "Yellow"] for color in colors: print(color) for i in
color: print(i)
```

Using `range()` in Loops

If you want to use `for` loop for a specific number of times, then you can use the `range()` function.

Syntax:

```
range(start, stop, step)
```

Example 1:

```
for n in range(5): print(n)
```

Output:

```
0 1 2 3 4
```

Here loop starts from 0 by default and increments at each iteration.

Example 2:

```
for n in range(5, 10): print(n)
```

Output:

```
5 6 7 8 9
```

Here loop starts from 5 and increments at each iteration upto (n-1) i.e. (10-1).

Example 3:

```
for n in range(5, 10, 2): #5 is start parameter, 10 is stop parameter and 2 is step
parameter. print(n)
```

Output:

```
5 7
```

The `while` loop

The `while` loop executes a block of code as long as a specified condition is `True`.
For example:

```
count = 1 while count < 5: print(count) count = count + 1
```

Output:

```
1 2 3 4
```

Infinite Loop

If the condition is never made `False`, the loop will continue endlessly.

Example of breaking an infinite loop:

```
while True: number = int(input("Enter a positive number: ")) print(number) if number <=
0: break
```

The `break` Statement

The `break` statement enables a program to skip over a part of the code. When the program encounters the `break` statement, it stops the loop it's currently running in.

For example:

```
for i in range(10): if i == 5: break print(i)
```

Output:

```
0 1 2 3 4
```

The `continue` Statement

The `continue` statement skips the remaining part of a loop and moves on to the next iteration of the loop.

For example:

```
for i in range(10): if i == 5: print("Skipping iteration") continue print(i)
```

Output of the above code:

```
0 1 2 3 4 Skipping iteration 6 7 8 9
```

`Do-While` Loop Equivalent

Python does not have a built-in `do-while` loop, but similar behavior can be achieved with a `while` loop.

For example:

```
while True: number = int(input("Enter a positive number: ")) print(number) if number <=
0: break
```

Functions in Python

A function is a block of reusable code that performs a specific task.

Functions help in organizing code, making it easier to maintain. They allow you to break down a complex problem into smaller, more manageable tasks, and you can call these functions whenever you need to execute the associated code.

Types of Functions

There are two types of functions:

1. **Built-in functions**: Predefined in Python, such as `len()`, `max()`, `print()`, `sum()`, etc.
2. **User-defined functions:** Functions created by the user to perform custom tasks.

Creating a Function

The `def` keyword is used to define a function in Python.

Syntax:

```
def function_name(parameters): # Code block
```

For example:

```
def greet(): print("Hello!")
```

Calling a function

You can call a function by giving the function name, followed by parameters (if any) in the parenthesis.

For example:

```
def add(a, b): print(a + b) add(5, 3) #output: 8
```

If you want to declare a function and don't want to give any code statements at that time, then you can use the `pass` statement to avoid any errors.

For example:

```
def add(a, b): #This gives a syntax error
```

You can use the `pass` statement to avoid this as follows:

```
def add(a, b): pass
```

Function Arguments

Functions can accept inputs, called **arguments**. There are four types of arguments that you can provide in a function:

- Default Arguments
- Keyword Arguments
- Required Arguments
- Variable length Arguments

Default arguments:

You can provide a default value for parameters while creating a function. This way if you don't provide any value at the time of function call then the function will take the default values.

For example:

```
def greet(name="Mark"): print("Hello", name) greet("John") # output: Hello John greet() #
output: Hello Mark
```

Keyword arguments:

You can provide arguments in key = value syntax. This way, the interpreter identifies the arguments based on their parameter names, so the order in which the arguments are passed becomes unimportant.

For example:

```
def greet(firstName, lastName): print("Hello", firstName, lastName) greet(lastName = "Phi
llips", firstName="Mark") # output: Hello Mark Phillips
```

Required arguments:

In case you don't pass the arguments with a key = value syntax, then it is necessary to pass the arguments in the correct positional order and the number of arguments passed should match with actual function definition.

For example:

```
def greet(firstName, lastName): print("Hello", firstName, lastName) greet("Mark", "Philli
ps") # output: Hello Mark Phillips
```

Variable-length arguments:

Sometimes you may need to pass more arguments than those defined in the actual function. This can be done using variable-length arguments.

You can achieve this in the following two ways:

Arbitrary Arguments:

While creating a function, pass a `*` (args) before the parameter name. The function accesses the arguments by processing them in the form of tuple.

For example:

```
def greet(*names): print("Hello,", names[0], names[1]) greet("Mark", "John") #output: Hel
lo, Mark John
```

Keyword Arbitrary Arguments:

While creating a function, pass a `**` (kwargs) before the parameter name. The function accesses the arguments by processing them in the form of dictionary.

```
def greet(**names): print("Hello,", names["firstName"], names["lastName"])
greet(firstName = "John", lastName = "Doe") #output: Hello, John Doe
```

Return Statement

The `return` statement is used to return a value back to the caller.

For example:

```
def add(a, b): return a + b result = add(5, 3) print(result) # Output: 8
```

Why Use Functions?

- Functions avoid rewriting the same code and makes your code reusable.
- Break large programs into smaller, manageable pieces.
- Functions clearly define their purpose, so it improves readability.
- Debug and update specific parts without affecting the rest.

Lists in Python

Lists are collection of data items in a specific order. In a list you can store multiple items in a single variable.

Lists are mutable, meaning their contents can be modified after creation.

Creating Lists

Lists are created using square brackets `[]`, with items separated by commas.

A list can contain items of different data types.

For example:

```
# List of integers list1 = [1,2,36,3,15] # List of strings list2 = ["Red", "Yellow",
"Blue"] # List of mixed data types list3 = [1, "John",12, 5.3] print(list1) # [1, 2, 36,
3, 15] print(list2) # ['Red', 'Yellow', 'Blue'] print(list3) # [1, 'John', 12, 5.3]
```

Length of List

You can find length of list (number of items in a list) using `len()` function.

For example:

```
list1 = [1,2,36,3,15] lengthOfList = len(list1) print(lengthOfList) # 5
```

Accessing List Items

You can access list items/elements using indexing. Each element has its unique index.

Indexing starts from 0 for the first element, 1 for the second element, and so on.

For example:

```
fruits = ["Orange", "Apple", "Banana"] print(fruits[0]) # Orange print(fruits[1]) # Apple
print(fruits[2]) # Banana
```

You can also access elements from the end of the list (-1 for the last element, -2 for the second-to-last element, and so on), this is called negative indexing.

For example:

```
fruits = ["Orange", "Apple", "Banana"] print(fruits[-1]) # Banana print(fruits[-2]) #
Apple print(fruits[-3]) # Orange # for understanding, you can consider this as
fruits[len(fruits)-3]
```

Check if an item exists in the list

You can check whether an item is present in the list or not, using the `in` keyword.

Example 1:

```
fruits = ["Orange", "Apple", "Banana"] if "Orange" in fruits: print("Orange is in the
list.") else: print("Orange is not in the list.") # Orange is in the list.
```

Example 2:

```
numbers = [1, 57, 13] if 7 in numbers: print("7 is in the list.") else: print("7 is not
in the list.") # 7 is not in the list.
```

Slicing Lists

You can access a range of list items by giving start, end and jump(skip) parameters.

Syntax:

```
listName[start : end : jumpIndex]
```

Note: jump Index is optional.

Example 1:

```
# Printing elements within a particular range numbers = [1, 57, 13, 6, 18, 54] #using pos
itive indexes(this will print the items starting from index 2 and ending at index 4 i.e.
(n-1)) print(numbers[2:5]) # [13, 6, 18] #using negative indexes(this will print the item
s starting from index -5 and ending at index -3 i.e. (-2-1)) print(numbers[-5:-2]) # [57,
13, 6]
```

Example 2:

When no end index is provided, the interpreter prints all the values till the end.

```
# Printing all elements from a given index till the end numbers = [1, 57, 13, 6, 18, 54]
#using positive indexes print(numbers[2:]) # [13, 6, 18, 54] #using negative indexes prin
t(numbers[-5:]) # [57, 13, 6, 18, 54]
```

Example 3:

When no start index is provided, the interpreter prints all the values from start up to the end index provided.

```
# Printing all elements from start to a given index numbers = [1, 57, 13, 6, 18, 54] #usi
ng positive indexes print(numbers[:4]) # [1, 57, 13, 6] #using negative indexes print(num
bers[:-2]) # [1, 57, 13, 6]
```

Example 4:

You can print alternate values by giving jump index.

```
# Printing alternate values numbers = [1, 57, 13, 6, 18, 54] #using positive indexes(here
start and end indexes are not given and 2 is jump index. print(numbers[::2]) # [1, 13, 1
8] #using negative indexes(here start index is -2, end index is not given and 2 is jump i
ndex. print(numbers[-2::2]) # [18]
```

List Comprehension

You can create new lists from other iterables like lists, tuples, dictionaries, sets, using list comprehensions.

Syntax:

```
List = [Expression(item) for item in Iterable if Condition]
```

Here,

- **Expression**: It's the item which is being iterated.
- **Iterable**: It can be list, tuples, dictionaries, sets, etc.
- **Condition**: Condition checks if the item should be added to the new list or not.

Example 1:

```
names = ["John", "Mark", "Bruno", "Dany"] # Print list of names in which name contains
"a" namesWith_a = [item for item in names if "a" in item] print(namesWith_a) # ['Mark',
'Dany']
```

Example 2:

```
names = ["John", "Mark", "Bruno", "Dany", "Jay"] # Print list of names in which length of
name is less than 4. nameLength = [item for item in names if (len(item) < 4)]
print(nameLength) # ['Jay']
```

List Methods

`list.sort()`

This method sorts the list in ascending order or alphabetically. The original list is updated.

Example 1:

```
fruits = ["banana", "apple", "orange", "plum"] fruits.sort() print(fruits) numbers = [14,
2,7,3,2,1,8,1,10,18,5,3] numbers.sort() print(numbers)
```

Output:

```
['apple', 'banana', 'orange', 'plum'] [1, 1, 2, 2, 3, 3, 5, 7, 8, 10, 14, 18]
```

To print the list in descending order, you have to give `reverse=True` as a parameter in the sort method.

For example:

```
fruits = ["banana", "apple", "orange", "plum"] fruits.sort(reverse=True) print(fruits) nu
mbers = [14,2,7,3,2,1,8,1,10,18,5,3] numbers.sort(reverse=True) print(numbers)
```

Output:

```
['plum', 'orange', 'banana', 'apple'] [18, 14, 10, 8, 7, 5, 3, 3, 2, 2, 1, 1]
```

The `reverse` parameter is set to `False` by default.

Note: Do not mistake the `reverse` parameter with the `reverse` method.

`reverse()`

This method reverses the order of the list.

Example:

```
fruits = ["banana", "apple", "orange", "plum"] fruits.reverse() print(fruits) numbers =
[14,2,7,3,2,1,8,1,10,18,5,3] numbers.reverse() print(numbers)
```

Output:

```
['plum', 'orange', 'apple', 'banana'] [3, 5, 18, 10, 1, 8, 1, 2, 3, 7, 2, 14]
```

`index()`

This method returns the index of the first occurrence of a specified value.

Example:

```
fruits = ["banana", "apple", "orange", "plum"] print(fruits.index("orange")) # 2 numbers
= [14,2,7,3,2,1,8,1,10,18,5,3] print(numbers.index(8)) # 6
```

`count()`

This method returns the number of occurrences of a specified value.

Example:

```
fruits = ["banana", "apple", "orange", "plum", "orange"] print(fruits.count("orange")) #
2 numbers = [14,2,7,3,2,1,8,1,10,18,5,3] print(numbers.count(2)) # 2
```

`copy()`

This method returns the copy of the list. This does not modify the original list.

Example:

```
fruits = ["banana", "apple", "orange", "plum", "orange"] copiedList = fruits.copy() print
(fruits) print(copiedList)
```

Output:

```
['banana', 'apple', 'orange', 'plum', 'orange'] ['banana', 'apple', 'orange', 'plum', 'or
ange']
```

`append()`

This method appends items to the end of the existing list.

Example:

```
fruits = ["banana", "apple", "orange", "plum", "orange"] fruits.append("grapes") print(fr
uits)
```

Output:

```
['banana', 'apple', 'orange', 'plum', 'orange', 'grapes']
```

`insert()`

This method inserts an item at the given index. You have to specify index and the item to be inserted within the `insert()` method.

Example:

```
fruits = ["banana", "apple", "orange", "plum", "orange"] fruits.insert(3, "grapes") #inse
rts item at index 3 print(fruits)
```

Output:

```
['banana', 'apple', 'orange', 'grapes', 'plum', 'orange']
```

`extend()`

This method adds an entire list or any other collection datatype (set, tuple, dictionary) to the existing list.

Example:

```
fruits = ["banana", "apple", "orange", "plum", "orange"] vegetables = ["potato", "tomat
o", "spinach"] fruits.extend(vegetables) print(fruits)
```

Output:

```
['banana', 'apple', 'orange', 'plum', 'orange', 'potato', 'tomato', 'spinach']
```

Concatenating Two Lists

You can combine two or more lists using the `+` operator.

Example:

```
fruits = ["banana", "apple", "orange", "plum", "orange"] vegetables = ["potato", "tomat
o", "spinach"] print(fruits + vegetables)
```

Output:

```
['banana', 'apple', 'orange', 'plum', 'orange', 'potato', 'tomato', 'spinach']
```

Tuples in Python

Tuples are ordered collection of data items. In tuples, you can store multiple items in a single variable.

Tuples are immutable i.e. you can not change them after creation.

Creating Tuples

Tuples are defined using round brackets `()` and items are separated by commas.

A tuple can contain items of different data types.

For example:

```
tuple1 = (1,2,36,3,15) tuple2 = ("Red", "Yellow", "Blue") tuple3 = (1, "John",12, 5.3)
print(tuple1) # (1, 2, 36, 3, 15) print(tuple2) # ('Red', 'Yellow', 'Blue') print(tuple3)
# (1, 'John', 12, 5.3)
```

Single-Item Tuples

To create a tuple with one item, add a **comma** after the item. Without a comma, Python will treat it as a integer type.

For example:

```
tuple1 = (1) # This is an integer. print(type(tuple1)) # <class 'int'> tuple2 = (1,) #
This is a tuple. print(type(tuple2)) # <class 'tuple'>
```

Length of Tuple

You can find length of tuple (number of items in a tuple) using `len()` function.

For example:

```
tuple1 = (1,2,36,3,15) lengthOfTuple = len(tuple1) print(lengthOfTuple) # 5
```

Accessing Tuple Items

You can access tuple items/elements using indexing. Each element has its unique index.

Indexing starts from 0 for the first element, 1 for the second element, and so on.

For example:

```
fruits = ("Orange", "Apple", "Banana") print(fruits[0]) # Orange print(fruits[1]) # Apple
print(fruits[2]) # Banana
```

You can also access elements from the end of the tuple (-1 for the last element, -2 for the second-to-last element, and so on), this is called **negative indexing**.

For example:

```
fruits = ("Orange", "Apple", "Banana") print(fruits[-1]) # Banana print(fruits[-2]) #
Apple print(fruits[-3]) # Orange # for understanding, you can consider this as
fruits[len(fruits)-3]
```

Check if an item is present in the tuple?

You can check whether an element is present in the tuple or not, using the `in` keyword.

Example 1:

```
fruits = ("Orange", "Apple", "Banana") if "Orange" in fruits: print("Orange is in the
tuple.") else: print("Orange is not in the tuple.") #Output: Orange is in the tuple.
```

Example 2:

```
numbers = (1, 57, 13) if 7 in numbers: print("7 is in the tuple.") else: print("7 is not
in the tuple.") # Output: 7 is not in the tuple.
```

Slicing Tuples

You can get a range of tuple items by giving start, end and jump(skip) parameters.

Syntax:

```
tupleName[start : end : jumpIndex]
```

Note: jump Index is optional.

Example 1:

```
# Printing elements within a particular range numbers = (1, 57, 13, 6, 18, 54) # using po
sitive indexes(this will print the items starting from index 2 and ending at index 4 i.e.
(5-1)) print(numbers[2:5]) # using negative indexes(this will print the items starting fr
om index -5 and ending at index -3 i.e. (-2-1)) print(numbers[-5:-2])
```

Output:

```
(13, 6, 18) (57, 13, 6)
```

Example 2:

When no end index is provided, the interpreter prints all the values till the end.

```
# Printing all elements from a given index to till the end numbers = (1, 57, 13, 6, 18, 5
4) # using positive indexes print(numbers[2:]) # using negative indexes print(numbers[-
5:])
```

Output:

```
(13, 6, 18, 54) (57, 13, 6, 18, 54)
```

Example 3:

When no start index is provided, the interpreter prints all the values from start up to the end index provided.

```
# Printing all elements from start to a given index numbers = (1, 57, 13, 6, 18, 54) #usi
ng positive indexes print(numbers[:4]) #using negative indexes print(numbers[:-2])
```

Output:

```
(1, 57, 13, 6) (1, 57, 13, 6)
```

Example 4:

You can print alternate values by giving jump index.

```
# Printing alternate values numbers = (1, 57, 13, 6, 18, 54) # using positive indexes(her
e start and end indexes are not given and 2 is jump index.) print(numbers[::2]) # using n
egative indexes(here start index is -2, end index is not given and 2 is jump index.) prin
t(numbers[-2::2])
```

Output:

```
(1, 13, 18) (18)
```

Manipulating Tuples

Tuples are **immutable**, so items cannot be added, removed, or changed. However, you can convert a tuple to a **list**, modify the list, and convert it back to a tuple.

For example:

```
fruits = ("Apple", "Orange", "Plum", "Banana") fruits_list = list(fruits) # Convert to li
st fruits_list.append("Guava") # Modify the list fruits = tuple(fruits_list) # Convert ba
ck to tuple print(fruits) # Output: ('Apple', 'Orange', 'Plum', 'Banana', 'Guava')
```

Concatenating Tuples

You can join two tuples using the `+` operator.

For example:

```
fruits1 = ("Apple", "Orange", "Plum") fruits2 = ("Banana", "Grapes") fruits = fruits1+ fr
uits2 print(fruits)
```

Output:

```
('Apple', 'Orange', 'Plum', 'Banana', 'Grapes')
```

Tuple Methods

Tuple has following built-in methods:

`count()`

This method returns the number of times an element appears in a tuple.

Syntax:

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

For example:

```
tuple1 = (1, 57, 3, 6, 18, 3, 3) count_3 = tuple1.count(3) print(count_3) # Output: 3
```

`index()`

This method returns the first occurrence of the given element from the tuple.

Note: This method raises a `ValueError` if the element is not found in the tuple.

For example:

```
tuple1 = (1, 57, 3, 6, 18, 54, 3) numberIndex = tuple1.index(3) print('Index of 3 in tupl
e1 is:', numberIndex) # Output: Index of 3 in tuple1 is: 2
```

You can specify a **start index** for the search. For example:

```
tuple1 = (1, 57, 13, 6, 18, 54, 13) numberIndex = tuple1.index(13, 3) # Start search at
index 3 print(numberIndex) # Output: 6
```

String Formatting in Python

Python provides multiple ways to format strings, making it easier to create dynamic and readable text.

Using the `format()` Method

The `format()` method allows you to dynamically insert values into strings using placeholders (`{}`).

For example:

```
name = "Shefali" country = "India" text = "Hey my name is {} and I am from {}"
print(text.format(name, country))
```

Output:

```
Hey my name is Shefali and I am from India
```

Changing Order of Parameters:

If you change the order of the given parameters, for example:

```
name = "Shefali" country = "India" text = "Hey my name is {} and I am from {}"
print(text.format(country, name))
```

Output:

```
Hey my name is India and I am from Shefali
```

To avoid the above case you can give the variables numbers. For example:

```
name = "Shefali" country = "India" text = "Hey my name is {1} and I am from {0}"
print(text.format(country, name))
```

Output:

```
Hey my name is Shefali and I am from India
```

Named Placeholders

You can use named arguments in the `format()` method for clarity.

For example:

```
txt = "For only {price:.2f} dollars!" print(txt.format(price = 49))
```

Output:

```
For only 49.00 dollars!
```

Here, `{price:.2f}` formats the value to 2 decimal places.

Using f-Strings

Introduced in Python 3.6, **f-strings** (formatted string literals) are prefixed with `f` and allow embedding expressions directly within the string.

Example 1:

```
name = "Shefali" country = "India" text = f"Hey my name is {name} and I am from {countr
y}" print(text)
```

Output:

```
Hey my name is Shefali and I am from India
```

Example 2: f-strings can evaluate expressions inline.

```
price = 49 txt = f"For only {price:.2f} dollars!" print(txt)
```

Output:

```
For only 49.00 dollars!
```

Example 3: f-strings in a single statement

```
print(f"{2 * 30})" # output: 60
```

Docstrings in python

Docstrings are **string literals** used to describe a module, function, class, or method. They help document the purpose and behaviour of the code and are typically placed right after the declaration.

For example:

```
def square(n): '''Takes in a number n, returns the square of n''' return n**2
```

Here, the docstring:

```
'''Takes in a number n, returns the square of n'''
```

is a description of the function.

Accessing Docstrings

You can access a function's docstring using its `__doc__` attribute.

For example:

```
def square(n): '''Takes in a number n, returns the square of n''' return n**2 print(squar
e.__doc__)
```

Output:

```
Takes in a number n, returns the square of n
```

Python Comments vs Docstrings

Python Comments

- Used to explain **what** the code does.
- Written with a `#` and ignored by the Python interpreter.
- Meant for developers and do not appear in the output.

Example:

```
# Takes in a number n, returns the square of n def square(n): return n**2
```

Python Docstrings

- Used to explain **what** the code or a function is intended for.
- Written as a string literal after a function, class, or module definition.
- Can be accessed at runtime using the `__doc__` attribute.

Example:

```
def square(n): '''Takes in a number n, returns the square of n''' return n**2 print(square.__doc__)
```

Output:

```
Takes in a number n, returns the square of n
```

PEP 8 and The Zen of Python

What is PEP 8?

PEP 8 is a style guide for writing Python code. It outlines conventions and best practices that ensure:

1. **Consistency**: Code looks uniform, making it easier to read and understand.
2. **Readability**: It emphasizes writing clean and readable code.
3. **Community Standards**: Encourages practices accepted and expected by the Python community.

Purpose of PEP 8

- To help Python developers write code that's more **understandable**.
- To ensure that Python codebases have a **consistent style**, regardless of who wrote the code.
- To make it easier for newcomers to read and contribute to existing Python projects.

Key PEP 8 Guidelines

1. **Indentation**:
   - Use **4 spaces per indentation level**.
   - Never mix tabs and spaces.

   ```
   def example_function(): for i in range(10): print(i) # 4 spaces used for indentation
   ```

2. **Line Length**:
   - Limit lines to a **maximum of 79 characters**.
   - For longer blocks of text (like comments), limit them to **72 characters**.

   ```
   # This is a very long comment. To ensure readability, limit each line # to 72 characters, even for comments, as demonstrated here.
   ```

3. **Blank Lines**:
   - Use blank lines to separate functions, classes, and blocks of code.

```
class Example: def method_one(self): pass def method_two(self): pass
```

4. **Imports**:
   - Place all imports at the top of the file.
   - Group imports in the following order:
     1. Standard library imports.
     2. Third-party library imports.
     3. Local application/library-specific imports.

```
# Example: import os import sys import numpy as np from myproject import mymodule
```

5. **Whitespace**:
   - Avoid extra spaces in expressions or statements.

```
# Correct: x = 1 + 2 y = (a, b) # Incorrect: x = 1 + 2 y = ( a, b )
```

6. **Naming Conventions**:
   - Variable and function names: `snake_case`
   - Class names: `PascalCase`
   - Constants: `UPPER_CASE`

```
# Example: class MyClass: def my_method(self): my_variable = 42
```

7. **Comments**:
   - Comments should be concise and relevant.
   - Use complete sentences when explaining complex code.

```
# Correct: # This function calculates the square of a number. def square(n): return n
** 2 # Incorrect: # function calc square def square(n): return n ** 2
```

8. **Docstrings**:
   - Use triple quotes for docstrings, and include a description for classes, functions, and modules.

```
def add(a, b): """Return the sum of a and b.""" return a + b
```

The Zen of Python

The Zen of Python is a collection of guiding principles for Python's design, articulated as a set of aphorisms by Tim Peters. These principles prioritize simplicity, readability, and practicality.

Key Aphorisms

- **Explicit is better than implicit**: Code should be straightforward and transparent.
- **Simple is better than complex**: Prefer simplicity when solving problems.
- **Complex is better than complicated**: If complexity is necessary, avoid unnecessary complications.
- **Flat is better than nested**: Minimize levels of nesting in code for better readability.
- **Readability counts**: Prioritize clarity and readability in code design.
- **Errors should never pass silently (unless explicitly silenced)**: Handle exceptions appropriately.
- **There should be one-- and preferably only one --obvious way to do it**: Encourage standard solutions over creative but unclear ones.
- **Namespaces are one honking great idea -- let's do more of those!**: Namespaces prevent conflicts and keep code organized.

Easter Egg: Viewing The Zen of Python

You can view the Zen of Python directly in Python by typing:

```
import this
```

Output:

```
The Zen of Python, by Tim Peters Explicit is better than implicit. Simple is better than
complex. Complex is better than complicated. Flat is better than nested. Sparse is better
than dense. Readability counts. Special cases aren't special enough to break the rules. A
lthough practicality beats purity. Errors should never pass silently. Unless explicitly s
ilenced. In the face of ambiguity, refuse the temptation to guess. There should be one--
and preferably only one --obvious way to do it. Although that way may not be obvious at f
irst unless you're Dutch. Now is better than never. Although never is often better than *
right* now. If the implementation is hard to explain, it's a bad idea. If the implementat
ion is easy to explain, it may be a good idea. Namespaces are one honking great idea -- l
et's do more of those!
```

Python Recursive Function

A **recursive function** is a function that calls itself in order to solve a problem. Each recursive call reduces the problem to a smaller subproblem, eventually leading to a base case, which stops the recursion.

Example:

```
def factorial(num): # Base case if num == 1 or num == 0: return 1 # Recursive case else:
return num * factorial(num - 1) num = 5 print("Number:", num) print("Factorial:", factori
al(num))
```

Output:

```
Number: 5 Factorial: 120
```

Explanation for the above code:

```
num * factorial(num - 1) 5 * factorial(5-1) 5 * factorial(4) This will again call the
function factorial(num) for factorial(4) 5 * 4 * factorial(4-1) 5 * 4 * factorial(3) 5 *
4 * 3 * factorial(2) 5 * 4 * 3 * 2 * factorial(1) 5 * 4 * 3 * 2 * 1 120
```

Sets in Python

Sets are unordered collection of data items. In a set, you can store multiple items in a single variable.

Sets are immutable, meaning you cannot change items of the set once created.

Sets do not contain duplicate items.

Creating Sets

Sets are defined using curly brackets `{}` and items are separated by commas.

Example:

```
data = {"John", 11, 2, False, 3.9, 11} print(data)
```

Output:

```
{False, 2, 3.9, 11, 'John'}
```

Here you can see that the items of set occur in random order and hence they cannot be accessed using index numbers. Also sets do not allow duplicate values.

Creating Empty Set

```
data = {} print(type(data)) # <class 'dict'>
```

In the above example, using empty curly brackets creates a dictionary not set. To create a empty set use `set()` as follows:

```
data = set() print(type(data)) # <class 'set'>
```

Accessing Set Items

Using `for` loop

Since indexing is not supported for sets, you can access items of set using a `for` loop.

Example:

```
data = {"John", 11, 2, False, 3.9, 11} for item in data: print(item)
```

Output:

```
False 2 3.9 11 John
```

Check if an item exists in the set

You can check if an item exists in the set or not, using `in` keyword.

Example:

```
info = {"Carla", 19, False, 5.9} if "Carla" in info: print("Carla is present.") else:
print("Carla is absent.") # Carla is present.
```

Joining Sets

`union()` and `update()`

The `union()` and `update()` methods are used to print all items that are present in two sets.

The `union()` method returns a new set whereas `update()` method adds item into the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Tokyo", "Seoul", "Kabul", "Ma
drid"} cities3 = cities.union(cities2) print(cities3)
```

Output:

```
{'Tokyo', 'Madrid', 'Kabul', 'Seoul', 'Berlin', 'Delhi'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Tokyo", "Seoul", "Kabul", "Ma
drid"} cities.update(cities2) print(cities)
```

Output:

```
{'Berlin', 'Madrid', 'Tokyo', 'Delhi', 'Kabul', 'Seoul'}
```

`intersection()` and `intersection_update()`

The `intersection()` and `intersection_update()` methods are used to print items that are common in both of the sets.

The `intersection()` method returns a new set whereas `intersection_update()` method updates the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Tokyo", "Seoul", "Kabul", "Ma
drid"} cities3 = cities.intersection(cities2) print(cities3)
```

Output:

```
{'Madrid', 'Tokyo'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Tokyo", "Seoul", "Kabul", "Ma
drid"} cities.intersection_update(cities2) print(cities)
```

Output:

```
{'Tokyo', 'Madrid'}
```

`symmetric_difference()` and `symmetric_difference_update()`

The `symmetric_difference()` and `symmetric_difference_update()` methods are used to print items that are not similar in both of the sets.

The `symmetric_difference()` method returns a new set whereas `symmetric_difference_update()` method updates the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Tokyo", "Seoul", "Kabul", "Ma
drid"} cities3 = cities.symmetric_difference(cities2) print(cities3)
```

Output:

```
{'Seoul', 'Kabul', 'Berlin', 'Delhi'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Tokyo", "Seoul", "Kabul", "Ma
drid"} cities.symmetric_difference_update(cities2) print(cities)
```

Output:

```
{'Kabul', 'Delhi', 'Berlin', 'Seoul'}
```

`difference()` and `difference_update()`

The `difference()` and `difference_update()` methods are used to print items that are only present in the original set and not in both of the sets.

The `difference()` method returns a new set whereas `difference_update()` method updates the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Seoul", "Kabul", "Delhi"} cit
ies3 = cities.difference(cities2) print(cities3)
```

Output:

```
{'Tokyo', 'Madrid', 'Berlin'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Seoul", "Kabul", "Delhi"} pri
nt(cities.difference(cities2))
```

Output:

```
{'Tokyo', 'Berlin', 'Madrid'}
```

Set Methods

Python provides following built-in methods for set manipulation:

`isdisjoint()`

The `isdisjoint()` method checks if items of a given set are present in another set. This method returns `False` if items are present, else it returns `True`.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Tokyo", "Seoul", "Kabul", "Ma
drid"} print(cities.isdisjoint(cities2)) # False
```

`issuperset()`

The `issuperset()` method checks if all the items of a particular set are present in the original set. It returns `True` if all the items are present, else it returns `False`.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Seoul", "Kabul"} print(citie
s.issuperset(cities2)) #False cities3 = {"Seoul", "Madrid","Kabul"} print(cities.issupers
et(cities3)) #False
```

`issubset()`

The `issubset()` method checks if all the items of the original set are present in the particular set. It returns `True` if all the items are present, else it returns `False`.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Delhi", "Madrid"} print(citie
s2.issubset(cities)) # True
```

`add()`

The `add()` method is used to add a single item to the set.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities.add("Helsinki") print(cities) #
{'Tokyo', 'Helsinki', 'Madrid', 'Berlin', 'Delhi'}
```

`update()`

If you want to add more than one item, simply create another set or any other iterable object(list, tuple, dictionary), and use the `update()` method to add it into the existing set.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities2 = {"Helsinki", "Warsaw", "Seoul"}
cities.update(cities2) print(cities) # {'Seoul', 'Berlin', 'Delhi', 'Tokyo', 'Warsaw', 'H
elsinki', 'Madrid'}
```

`remove()/discard()`

The `remove()` and `discard()` methods are used to remove items from the set.

Example :

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities.remove("Tokyo") print(cities) #
{'Delhi', 'Berlin', 'Madrid'}
```

The main difference between `remove()` and `discard()` is that, if you try to remove an item which is not present in the set, then `remove()` raises an error, whereas `discard()` does not raise any error.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities.remove("Seoul") print(cities) # Ke
yError: 'Seoul'
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities.discard("Seoul") print(cities) #
{'Madrid', 'Delhi', 'Berlin', 'Tokyo'}
```

`pop()`

The `pop()` method is used to remove the last item of the set but the catch is that you don't know which item gets popped as sets are unordered.

However, you can access the popped item if you assign the `pop()` method to a variable.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} item = cities.pop() print(cities) print(i
tem)
```

Output:

```
{'Tokyo', 'Delhi', 'Berlin'} Madrid
```

`del`

`del` is not a method, rather it is a keyword which is used to delete the set entirely.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} del cities print(cities) # NameError: nam
e 'cities' is not defined
```

You get an error because the entire set has been deleted and there is no variable called cities which contains a set.

Now, if you don't want to delete the entire set and just want to delete all items within that set then you can use the `clear()` method.

`clear()`

The `clear()` method is used to clear all items in the set and prints an empty set.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"} cities.clear() print(cities) # set()
```

Dictionaries

Dictionaries are ordered collection of data items. They store multiple items in a single variable.

Creating Dictionaries

Dictionary items are key-value pairs that are separated by commas and enclosed within curly brackets `{}`.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} print(info)
```

Output:

```
{'name': 'John', 'profession': 'Developer', 'age': 21}
```

Accessing Dictionary items

You can access dictionary items in the following ways:

Accessing single values

Values in a dictionary can be accessed using keys. You can access dictionary values by mentioning keys either in square brackets `[]` or by using `get()` method.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} print(info['name']) print(inf
o.get('name'))
```

Output:

```
John John
```

The difference between `[]` and `get()` is that when you try to access the key which is not present in the dictionary then `[]` gives an error while `get()` gives `None`.

For example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} print(info['experience']) #
KeyError: 'experience' print(info.get('experience')) #None
```

Accessing multiple values

You can access all the values of a dictionary using `values()` method.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} print(info.values())
```

Output:

```
dict_values(['John', 'Developer', 21])
```

Accessing keys

You can access all the keys of a dictionary using `keys()` method.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} print(info.keys())
```

Output:

```
dict_keys(['name', 'profession', 'age'])
```

Accessing key-value pairs

You can access all the key-value pairs of a dictionary using `items()` method.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} print(info.items())
```

Output:

```
dict_items([('name', 'John'), ('profession', 'Developer'), ('age', 21)])
```

Dictionary Methods

Python provides the following built-in methods for dictionary manipulation:

`update()`

The `update()` method is used to update the value of the key provided to it. If the item already exists in the dictionary then it will update the item, otherwise it creates a new key-value pair.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} print(info) info.update({'ag
e':32}) info.update({'lastName':"Wick"}) print(info)
```

Output:

```
{'name': 'John', 'profession': 'Developer', 'age': 21} {'name': 'John', 'profession': 'De
veloper', 'age': 32, 'lastName': 'Wick'}
```

`clear()`

The `clear()` method is used to remove all the items from the list.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} info.clear() print(info)
```

Output:

```
{}
```

`pop()`

The `pop()` method is used to remove that key-value pair whose key is passed as a parameter.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} info.pop('profession') print(i
nfo)
```

Output:

```
{'name': 'John', 'age': 21}
```

`popitem()`

The `popitem()` method is used to remove the last key-value pair from the dictionary.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} info.popitem() print(info)
```

Output:

```
{'name': 'John', 'profession': 'Developer'}
```

`del`

The `del` keyword is used to remove a dictionary item.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} del info['profession'] print(i
nfo)
```

Output:

```
{'name': 'John', 'age': 21}
```

If key is not provided, then the `del` keyword will delete the entire dictionary.

Example:

```
info = {'name':'John', 'profession':'Developer', 'age':21} del info print(info)
```

Output:

```
NameError: name 'info' is not defined
```

Merge Dictionaries

In the latest update of python (Python 3.9), you can use `|` operator to merge two dictionaries.

```
info1 = {'name': 'John', 'age': 21} info2 = {'profession': 'Developer', 'city': 'New
York'} merged = info1 | info2 print(merged)
```

Output:

```
{'name': 'John', 'age': 21, 'profession': 'Developer', 'city': 'New York'}
```

Python - `else` in Loops

In Python, the `else` keyword can be used with both `for` and `while` loops. This special usage allows the execution of a block of code after the loop completes all iterations. However, the `else` block is **not executed if the loop is terminated prematurely using a `break` statement**.

Syntax:

```
for variable in sequence: # Code to execute for each iteration else: # Code to execute af
ter the loop finishes
```

The `else` block is executed only when the loop successfully completes its iterations. If a `break` statement exits the loop, the `else` block will be skipped.

Using `else` with a `for` Loop

Example:

```
for x in range(5): print(f"Iteration {x + 1} in the loop") else: print("Loop completed su
ccessfully.") print("Outside the loop")
```

Output:

```
Iteration 1 in the loop Iteration 2 in the loop Iteration 3 in the loop Iteration 4 in th
e loop Iteration 5 in the loop Loop completed successfully. Outside the loop
```

Using `else` with `break`

If a `break` is used to exit the loop, the `else` block will not execute.
For example:

```
for x in range(5): if x == 3: print(f"Breaking the loop at iteration {x + 1}") break
print(f"Iteration {x + 1} in the loop") else: print("Loop completed successfully.")
```

Output:

```
Iteration 1 in the loop Iteration 2 in the loop Iteration 3 in the loop Breaking the loop
at iteration 4
```

Using `else` with a `while` Loop

The same behavior applies to `while` loops. If the loop completes naturally, the `else` block will execute. If the loop is interrupted with a `break`, the `else` block will not execute.

```
i = 0 while i < 5: print(f"Iteration {i + 1} in the while loop") i += 1 if i == 3: break
else: print("While loop completed successfully.")
```

Output:

```
Iteration 1 in the while loop Iteration 2 in the while loop Iteration 3 in the while loop
```

Special Scenarios

Empty Loops:

If the loop body does not execute (e.g., an empty sequence or a `while` loop condition is `False` initially), the `else` block will still execute.

```
for x in []: print("This will not execute.") else: print("Loop completed successfully.")
```

Output:

```
Loop completed successfully.
```

Nested Loops:

The `else` block applies only to the loop it is associated with. In nested loops, it executes when the specific loop completes successfully.

```
for i in range(2): for j in range(2): if i == j: print(f"Breaking inner loop at i={i}, j=
{j}") break else: print(f"Outer loop iteration {i} completed without breaking.")
```

Output:

```
Breaking inner loop at i=0, j=0 Outer loop iteration 1 completed without breaking.
```

Python Exception Handling

Exception handling involves managing unexpected errors during program execution. It ensures the program can gracefully handle unforeseen situations without crashing, maintaining smooth functionality.

What are Exceptions?

An exception is an error that occurs during program execution. Python has several built-in exceptions to handle errors like invalid input, type mismatches, or index out of bounds.

If an exception is not handled, the program terminates with a traceback. Exception handling prevents this by providing alternative code execution paths.

Using `try...except`

The `try...except` blocks are used to handle errors and exceptions. The code in `try` block runs when there is no error. If the `try` block catches the error, then the `except` block is executed.

Syntax:

```
try: # Code that may raise an exception except: # Code to handle the exception
```

Example:

```
a = input("Enter a number: ") print(f"Multiplication table of {a}:") try: for i in
range(1, 11): print(f"{int(a)} x {i} = {int(a) * i}") except: print("Invalid input!")
print("End of program.")
```

Output (Valid Input):

```
Enter a number: 5 Multiplication table of 5: 5 X 1 = 5 5 X 2 = 10 5 X 3 = 15 5 X 4 = 20 5
X 5 = 25 5 X 6 = 30 5 X 7 = 35 5 X 8 = 40 5 X 9 = 45 5 X 10 = 50 End of program.
```

Output (Invalid Input):

```
Enter a number: 3.2 Multiplication table of 3.2: Invalid Input! End of program.
```

Handling Multiple Exceptions

You can handle specific exceptions separately by using multiple `except` blocks.

```
try: num = int(input("Enter an integer: ")) a = [6, 3] print(a[num]) except ValueError:
print("The input is not an integer.") except IndexError: print("Index is out of range.")
```

The `finally` Clause

The `finally` block is executed **no matter what**—whether an exception is raised or not. It is often used for cleanup tasks like closing files or releasing resources.

Syntax:

```
try: # Code that may raise an exception except: # Code to handle exceptions finally: # Co
de that always executes
```

Example:

```
try: num = int(input("Enter an integer: ")) except ValueError: print("Invalid input.") el
se: print("Input accepted.") finally: print("This block always executes.")
```

Output (Valid Input):

```
Enter an integer: 10 Input accepted. This block always executes.
```

Output (Invalid Input):

```
Enter an integer: abc Invalid input. This block always executes.
```

Raising Exceptions

You can explicitly raise exceptions by using the `raise` keyword. This is useful for enforcing specific conditions in your code.

For example:

```
value = int(input("Enter a value between 5 and 9: ")) if value < 5 or value > 9: raise Va
lueError("Value must be between 5 and 9.") else: print("Valid input!")
```

Output (Valid Input):

```
Enter a value between 5 and 9: 7 Valid input!
```

Output (Invalid Input):

```
Enter a value between 5 and 9: 3 ValueError: Value must be between 5 and 9.
```

Custom Exceptions

You can define custom exceptions by creating a class that inherits from Python's built-in `Exception` class. This is useful when handling application-specific errors.

Defining a Custom Exception:

```
class CustomError(Exception): pass try: raise CustomError("This is a custom exception.")
except CustomError as e: print(e)
```

Shorthand If-Else Statements

You can use conditional expressions (shorthand if-else) to simplify simple conditions into one line.

Syntax:

```
result = value_if_true if condition else value_if_false
```

Example:

```
a = 10 b = 20 print("A is greater") if a > b else print("B is greater")
```

Output:

```
B is greater
```

Multiple Conditions in One Line

You can nest multiple conditions in a single line.

Example:

```
a = 330 b = 330 print("A") if a > b else print("=") if a == b else print("B")
```

Output:

```
=
```

Comparison with Standard `if-else`

The shorthand is equivalent to:

```
if condition: result = value_if_true else: result = value_if_false
```

Example:

```
x = 15 y = 10 # Shorthand result = "X is greater" if x > y else "Y is greater" # Equivale
nt Full Form if x > y: result = "X is greater" else: result = "Y is greater" print(resul
t)
```

When to Use Shorthand If-Else

- **Use it when**:
  - The condition and expressions are simple.
  - You're assigning a value or printing a message.
- **Avoid it when**:
  - The logic is complex or spans multiple lines.

Python's `enumerate` Function

The `enumerate` function is a built-in function in Python that simplifies looping through sequences (such as a lists, tuples, or strings) by providing both the **index** and the **value** of each element in the sequence. This makes it an efficient and clean way to access both the element and its position during iteration.

Basic Usage

The `enumerate` function returns a sequence of tuples, where each tuple contains:

1. The **index** of the element.
2. The **value** of the element.

Syntax:

```
enumerate(iterable, start=0)
```

- `iterable`: The sequence you want to iterate over (e.g., list, tuple, string).
- `start` *(optional)*: Specifies the starting index (default is `0`).

Example:

```
fruits = ['apple', 'banana', 'mango'] for index, fruit in enumerate(fruits): print(index,
fruit)
```

Output:

```
0 apple 1 banana 2 mango
```

Customizing the Start Index

By default, the enumerate function starts the index at 0, but you can change the starting index by passing a value for the `start` parameter.

For example:

```
fruits = ['apple', 'banana', 'mango'] for index, fruit in enumerate(fruits, start=1): pri
nt(index, fruit)
```

Output:

```
1 apple 2 banana 3 mango
```

Common Use Cases

1. Numbered Output

When you need to print a sequence with numbered items:

```
fruits = ['apple', 'banana', 'mango'] for index, fruit in enumerate(fruits, start=1): pri
nt(f'{index}: {fruit}')
```

Output:

```
1: apple 2: banana 3: mango
```

2. Enumerating Tuples

The `enumerate` function works seamlessly with tuples.

```
colors = ('red', 'green', 'blue') for index, color in enumerate(colors): print(index, col
or)
```

Output:

```
0 red 1 green 2 blue
```

3. Enumerating Strings

Strings are iterable, so you can use `enumerate` to loop over their characters.

```
s = 'hello' for index, char in enumerate(s): print(index, char)
```

Output:

```
0 h 1 e 2 l 3 l 4 o
```

Virtual Environment

A virtual environment is a tool that creates isolated Python environments on your system. It allows you to manage dependencies for multiple projects independently, avoiding conflicts between them. For example, you might need different versions of the same library for two separate projects, and virtual environments ensure they don't interfere with each other.

Creating a Virtual Environment

You can create a virtual environment using Python's built-in `venv` module.
Steps:

1. Create the Environment:

```
python -m venv myenv
```

   This creates a directory named `myenv` containing the virtual environment.

2. Activate the Virtual Environment:

- Linux/macOS:

```
source myenv/bin/activate
```

- Windows (Command Prompt):

```
myenv\Scripts\activate.bat
```

- Windows (PowerShell):

```
myenv\Scripts\activate.ps1
```

3. Deactivate the Environment:

```
deactivate
```

Installing Dependencies

Once the virtual environment is activated, any packages installed via `pip` will only apply to that environment. These packages won't affect the global Python environment or other virtual environments.

```
pip install <package-name>
```

The `requirements.txt` file

A `requirements.txt` file helps manage and share project dependencies.

Generate `requirements.txt`

To create a `requirements.txt` file, you can use the `pip freeze` command, which outputs a list of installed packages and their versions.

For example:

```
pip freeze > requirements.txt
```

Install Packages from `requirements.txt`

To install the packages listed in the `requirements.txt` file, you can use the `pip install` command with the `-r` flag:

```
pip install -r requirements.txt
```

This ensures all necessary dependencies are installed when setting up the project on a new system.

Importing Modules in Python

Importing in Python allows you to reuse code from another module or library in your current script. This allows you to use the functions and variables defined in the module in your current script.

Using `import`

```
import math
```

Once a module is imported, you can then use any of its functions and variables by using the dot notation.

For example:

```
import math result = math.sqrt(9) print(result) # Output: 3.0
```

Using `from`

You can import specific functions or variables from a module using `from`.

For example:

```
from math import sqrt result = sqrt(9) print(result) # Output: 3.0
```

You can also import multiple functions or variables at once by separating them with a comma:

```
from math import sqrt, pi result = sqrt(9) print(result) # Output: 3.0 print(pi) # Outpu
t: 3.141592653589793
```

Importing All Items

The `*` wildcard imports everything from a module. However, this is discouraged as it can cause conflicts between similarly named items.

```
from math import * result = sqrt(9) print(result) # Output: 3.0 print(pi) # Output: 3.141592653589793
```

Using `as`

You can rename imported modules using the `as` keyword. This can be useful if you want to use a shorter or more descriptive name for a module, or if you want to avoid naming conflicts with other modules or variables in your code.

```
import math as m print(m.sqrt(9)) # Output: 3.0
```

Exploring Module Contents

Python has a built-in function called `dir` that you can use to view all the functions and variables defined in a module. This can be helpful for exploring and understanding the contents of a new module.

```
import math print(dir(math))
```

This lists available functions, constants, and attributes in the `math` module.

`if "__name__ == "__main__"` in Python

The `if __name__ == "__main__"` idiom helps distinguish between when a script is run directly versus when it is imported as a module.

Here,

- `__name__`: A special variable that contains the name of the module.
- If the script is run directly, `__name__` is set to `"__main__"`.
- If the script is imported, `__name__` is set to the module's name.

Example:

```
def main(): print("This script is being run directly.") if __name__ == "__main__": main()
```

When Run Directly:

```
python script.py
```

Output:

```
This script is being run directly.
```

When Imported:

```
import script
```

Output:

```
No output
```

Why Use It?

The idiom ensures certain code only executes when the script is run directly, not when imported.

For example:

- A script might contain reusable functions or classes that shouldn't execute upon import.
- Ensures clear separation of reusable logic and executable script code.

Local and Global Variables

Local Variables

A **local variable** is declared inside a function and is only accessible within that function. It is created when the function is called and is destroyed once the function finishes execution.

Example:

```
def my_function(): y = 5 # Local variable print(y) my_function() print(y) # This will cau
se an error because 'y' is not accessible outside the function.
```

Output:

```
5 NameError: name 'y' is not defined
```

Global Variables

A **global variable** is declared outside of all functions and can be accessed from any function within the code.

Example:

```
x = 10 # Global variable def my_function(): print(x) # Accessing the global variable insi
de the function my_function() print(x) # Accessing the global variable outside the functi
on
```

Output:

```
10 10
```

The `global` Keyword

The `global` keyword is used inside a function to indicate that a variable is defined in the global scope, allowing the function to modify its value.

Example:

```
x = 10 # Global variable def my_function(): global x x = 5 # Modifying the global variabl
e print(x) my_function() print(x) # The value of 'x' is now changed globally
```

Output:

```
5 5
```

Global Variables Without `global`

If you attempt to modify a global variable inside a function without using the `global` keyword, Python will treat it as a local variable, leading to an error.

Example:

```
x = 10 def my_function(): x = x + 5 # This will cause an error because 'x' is treated as
a local variable print(x) my_function()
```

Error:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

Good Practices

1. **Limit the use of global variables**: Overusing them can make your code harder to debug and maintain.
2. **Prefer passing arguments**: Instead of modifying global variables, pass variables as arguments to functions.
3. **Encapsulation**: Keep variables localized to where they are needed to avoid unintended side effects.

Handling Files in Python

Opening a File

The `open()` function is used to open files in Python. It takes two arguments:

1. **File name**: The name of the file to open.
2. **Mode**: Specifies how the file should be opened (e.g., for reading, writing, or appending).

Example:

```
f = open('myfile.txt', 'r') # Opens the file in read mode
```

File Open Modes

Mode	Description
`'r'`	Open for reading (default). Raises an error if the file does not exist.
`'w'`	Open for writing. Creates a new file or overwrites an existing file.
`'a'`	Open for appending. Creates a new file if it doesn't exist, and appends to the end of the file.
`'x'`	Open for exclusive creation. Raises an error if the file already exists.
`'t'`	Text mode (default).
`'b'`	Binary mode (used for non-text files like images, videos, etc.).

Reading from a File

1. **Using** `read()`:
   Reads the entire content of a file as a string.

   ```
   with open('myfile.txt', 'r') as f: content = f.read() print(content)
   ```

2. **Using** `readline()`:
   Reads a single line from the file.

   ```
   with open('myfile.txt', 'r') as f: line = f.readline() print(line)
   ```

3. Using `readlines()`:
   Reads all lines from the file and returns a list of strings.

```
with open('myfile.txt', 'r') as f: lines = f.readlines() print(lines)
```

Writing to a File

1. Using `write()`:
   Writes a string to a file. Overwrites the file if it exists.

```
with open('myfile.txt', 'w') as f: f.write('Hello, world!')
```

2. Using `writelines()`:
   Writes multiple lines (provided as a list or iterable) to a file.

```
lines = ['Line 1\n', 'Line 2\n', 'Line 3\n'] with open('myfile.txt', 'w') as f: f.wri
telines(lines)
```

Appending to a File

To add content to the end of an existing file, use append mode (`'a'`).

Example:

```
with open('myfile.txt', 'a') as f: f.write('This will be appended.\n')
```

Closing a File

Always close a file after working with it to release resources. Use `close()` or the `with` statement for automatic closure.

Manual closure:

```
f = open('myfile.txt', 'r') # Do something f.close()
```

Automatic closure using `with`:

```
with open('myfile.txt', 'r') as f: content = f.read()
```

File Position: `seek()` and `tell()`

1. `seek(offset)`:
   Moves the file's position to a specific byte.

   Example:

   ```
   with open('myfile.txt', 'r') as f: f.seek(5) # Move to the 5th byte data = f.read() p
   rint(data)
   ```

2. `tell()`:
   Returns the current position of the file pointer.

   ```
   with open('myfile.txt', 'r') as f: print(f.tell()) # Prints the current position
   ```

Truncating a File: `truncate()`

Shortens a file to a specified length (in bytes). If no length is provided, it truncates the file from the current position.

Example:

```
with open('myfile.txt', 'w') as f: f.write('Hello World!') f.truncate(5) # Keeps only the
first 5 bytes with open('myfile.txt', 'r') as f: print(f.read()) # Output: Hello
```

Best Practices

1. **Use `with` for safe file handling**: Automatically closes the file even in case of exceptions.
2. **Use appropriate modes**: Choose the correct mode to avoid unintentional data loss.
3. **Validate file paths**: Check for file existence and permissions before performing operations.

Map, Filter, and Reduce in Python

In Python, `map`, `filter`, and `reduce` are powerful higher-order functions that apply other functions to sequences or iterables. They allow concise and functional-style processing of data.

1. `map()`

The `map()` function applies a specified function to each item in an iterable (e.g., list, tuple) and returns a new iterable with the results.

Syntax:

```
map(function, iterable)
```

- **function**: A function to apply to each element.
- **iterable**: The sequence of elements to process.

Example:

```
# List of numbers numbers = [1, 2, 3, 4, 5] # Double each number using `map` doubled = ma
p(lambda x: x * 2, numbers) print(list(doubled)) # Output: [2, 4, 6, 8, 10]
```

2. `filter()`

The `filter()` function filters elements from an iterable based on a predicate (a function that returns `True` or `False`). It returns an iterable containing only the elements that satisfy the condition.

Syntax:

```
filter(predicate, iterable)
```

- **predicate**: A function that evaluates to `True` or `False`.
- **iterable**: The sequence of elements to filter.

Example:

```
# List of numbers numbers = [1, 2, 3, 4, 5] # Get only even numbers evens = filter(lambda
x: x % 2 == 0, numbers) print(list(evens)) # Output: [2, 4]
```

Key Notes:

- If the predicate function returns `False`, the element is excluded.
- Useful for filtering based on conditions or rules.

3. `reduce()`

The `reduce()` function reduces a sequence of elements into a single value by repeatedly applying a specified function. It is part of the `functools` module.

Syntax:

```
from functools import reduce reduce(function, iterable[, initializer])
```

- **function**: A function that takes two arguments and reduces them to one.
- **iterable**: The sequence of elements to process.
- **initializer** *(optional)*: A starting value for the reduction.

Example:

```
from functools import reduce # List of numbers numbers = [1, 2, 3, 4, 5] # Calculate the
sum of the numbers total = reduce(lambda x, y: x + y, numbers) print(total) # Output: 15
```

Key Notes:

- The function is applied cumulatively:
  - First step: `1 + 2 = 3`
  - Second step: `3 + 3 = 6`
  - Third step: `6 + 4 = 10`
  - Fourth step: `10 + 5 = 15`
- Adding an **initializer** can specify a starting point:

  ```
  result = reduce(lambda x, y: x + y, numbers, 10) print(result) # Output: 25 (10 + 15)
  ```

Example Combining All Three

```
from functools import reduce # List of numbers numbers = [1, 2, 3, 4, 5] # Step 1: Double
each number doubled = map(lambda x: x * 2, numbers) # Step 2: Filter out numbers greater
than 5 filtered = filter(lambda x: x > 5, doubled) # Step 3: Sum up the filtered numbers
result = reduce(lambda x, y: x + y, filtered) print(result) # Output: 18 (6 + 8 + 10)
```

Lambda Functions in Python

A **lambda function** in Python is a concise, **anonymous function** defined using the `lambda` keyword. These functions are often used for simple operations, especially when passing them as arguments to higher-order functions like `map()`, `filter()`, and `reduce()`.

Syntax:

```
lambda arguments: expression
```

- **Arguments**: Can be zero, one, or multiple.
- **Expression**: A single expression whose result is returned when the lambda function is called.

Key Features

1. **Anonymous**: Unlike regular functions, lambda functions are not bound to a name.
2. **Single Expression**: Lambda functions can only contain a single expression, which is automatically returned.
3. **Short-lived**: Primarily used in places where small, simple functions are needed temporarily.

Examples

Basic Lambda Function:

Equivalent to a function that doubles a number:

```
# Regular function def double(x): return x * 2 # Lambda function double_lambda = lambda
x: x * 2 print(double_lambda(5)) # Output: 10
```

Lambda Function with Multiple Arguments:

Calculate the product of two numbers:

```
# Regular function def multiply(x, y): return x * y # Lambda function multiply_lambda = l
ambda x, y: x * y print(multiply_lambda(3, 4)) # Output: 12
```

Lambda with No Arguments:

Return a fixed value:

```
constant_lambda = lambda: 42 print(constant_lambda()) # Output: 42
```

Using Lambda with Higher-Order Functions

1. `map()` : Applies a function to each element of an iterable.

   ```
   numbers = [1, 2, 3, 4] squared = map(lambda x: x**2, numbers) print(list(squared)) #
   Output: [1, 4, 9, 16]
   ```

2. `filter()` : Filters elements of an iterable based on a condition.

   ```
   numbers = [1, 2, 3, 4, 5, 6] even_numbers = filter(lambda x: x % 2 == 0, numbers) pri
   nt(list(even_numbers)) # Output: [2, 4, 6]
   ```

3. `reduce()` : Reduces an iterable to a single value (requires `functools`).

   ```
   from functools import reduce numbers = [1, 2, 3, 4] product = reduce(lambda x, y: x *
   y, numbers) print(product) # Output: 24
   ```

Lambda vs Regular Functions

Aspect	Lambda Function	Regular Function
Definition	Defined using `lambda`.	Defined using `def`.
Name	Anonymous (no name).	Always has a name.
Complexity	Limited to a single expression.	Can include multiple statements and logic.
Use Case	Short-lived, simple operations.	General-purpose, reusable logic.
Readability	Less readable for complex logic.	More readable and maintainable.

`is` vs `==` in Python

In Python, `is` and `==` are both comparison operators that can be used to check if two values are equal.

Key Differences Between `is` and `==`

Feature	`is`	`==`
Purpose	Checks **identity** (same memory location).	Checks **equality** (same value).
Returns	`True` if objects are the same instance.	`True` if objects have the same value.
Use Case	Verify if two references point to the same object.	Compare data for equality.
Example with Lists	`a is b` returns `False` for two separate but equal lists.	`a == b` returns `True` for lists with the same contents.

Example 1: Immutable Objects

```
# Immutable integers a = 1000 b = 1000 print(a == b) # True (values are the same) print(a
is b) # False (different memory locations for large integers) # Small integers (Python ca
ches -5 to 256) x = 100 y = 100 print(x == y) # True print(x is y) # True (both point to
the same cached object) # Strings s1 = "hello" s2 = "hello" print(s1 == s2) # True print
(s1 is s2) # True (string interning reuses memory)
```

Example 2: Mutable Objects

```
# Mutable lists a = [1, 2, 3] b = [1, 2, 3] print(a == b) # True (values are the same) pr
int(a is b) # False (different objects in memory) # Same object reference c = a print(a i
s c) # True (both refer to the same object)
```

Example 3: `None` Comparisons

The `is` operator is often used to compare with `None` because `None` is a singleton in Python.

```
a = None print(a is None) # True print(a == None) # True, but using `is` is preferred sty
listically.
```

When to Use `is` vs `==`

- Use `is`:
  - To check if two variables refer to the **same object**.
  - Comparing with `None`: `if obj is None:`
  - Verifying identity of singletons or cached objects.
- Use `==`:
  - To check if the **values** of two variables are equal.
  - Suitable for most comparisons involving data structures (lists, strings, dictionaries, etc.).

Introduction to Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that revolves around **objects** and **classes**. In OOP, classes define the structure and behavior of objects, which are instances of those classes. This approach helps model real-world problems more effectively by using principles such as encapsulation, inheritance, polymorphism, and abstraction.

Key Concepts of OOP

1. **Class**: A blueprint for creating objects. It defines attributes (data) and methods (functions) that objects created from the class will have.
2. **Object**: An instance of a class, containing specific data and the ability to perform actions via methods.
3. **Encapsulation**: The bundling of data and methods that operate on that data within a single unit (class). It restricts direct access to some of an object's components and provides controlled access through methods.
4. **Inheritance**: The ability of a new class (child class) to inherit properties and methods from an existing class (parent class), promoting code reuse.
5. **Polymorphism**: The ability to define methods in a base class and override them in derived classes, allowing the same method name to behave differently based on the object it is called on.

Python Classes and Objects

Defining a Class

A class in Python is defined using the `class` keyword. It can contain attributes and methods that define the behavior and characteristics of objects created from the class.

```
class Person: name = "John" age = 30
```

Creating an Object

An object is an instance of a class. Objects can be created by calling the class.

```
person1 = Person() # Creates an object of the Person class
```

Accessing Class Properties

You can access attributes and methods of an object using the dot (`.`) operator.

```
print(person1.name) # Output: John print(person1.age) # Output: 30
```

The `self` Parameter

In Python, the `self` parameter is a reference to the current instance of the class. It is used within the class to access attributes and methods.

```
class Person: def greet(self): print("Hello, I am", self.name) person1 = Person() person
1.name = "Alice" person1.greet() # Output: Hello, I am Alice
```

Constructors

A **constructor** is a special method used to initialize objects. In Python, the constructor method is `__init__`.

Types of Constructors

1. **Default Constructor**: Takes no parameters except `self`.

```
class Person: def __init__(self): self.name = "Default Name" person1 = Person() print(per
son1.name) # Output: Default Name
```

2. **Parameterized Constructor**: Accepts arguments to initialize attributes.

```
class Person: def __init__(self, name, age): self.name = name self.age = age person1 = Pe
rson("Alice", 25) print(person1.name, person1.age) # Output: Alice 25
```

Inheritance

Inheritance allows a class (child class) to inherit properties and methods from another class (parent class). This promotes code reuse and enhances modularity.

Example

```
class Animal: def speak(self): print("Animal speaks") class Dog(Animal): def speak(self):
print("Dog barks") dog = Dog() dog.speak() # Output: Dog barks
```

Types of Inheritance

1. **Single Inheritance**: A child class inherits from one parent class.
2. **Multiple Inheritance**: A child class inherits from more than one parent class.
3. **Multilevel Inheritance**: A class inherits from another derived class, creating a chain.
4. **Hierarchical Inheritance**: Multiple classes inherit from a single parent class.
5. **Hybrid Inheritance**: Combines two or more types of inheritance.

Polymorphism

Polymorphism allows methods to behave differently based on the object calling them, even if they have the same name.

Example:

```
class Cat: def speak(self): print("Cat meows") class Dog: def speak(self): print("Dog bar
ks") def animal_sound(animal): animal.speak() dog = Dog() cat = Cat() animal_sound(dog) #
Output: Dog barks animal_sound(cat) # Output: Cat meows
```

Encapsulation

Encapsulation involves restricting access to some of an object's attributes and methods, typically by making them private, and providing public methods to access or modify them.

Example:

```
class BankAccount: def __init__(self, balance): self.__balance = balance # Private attrib
ute def deposit(self, amount): self.__balance += amount def get_balance(self): return sel
f.__balance account = BankAccount(1000) account.deposit(500) print(account.get_balance())
# Output: 1500
```

Access Modifiers

Python provides three main types of access modifiers to manage the visibility of class attributes and methods:

1. **Public**: Accessible from anywhere (default).
2. **Private**: Accessible only within the class. Indicated by `__`.
3. **Protected**: Meant to be accessed within the class and its subclasses. Indicated by `_`.

Example: Public Access

```
class Student: def __init__(self, age, name): self.age = age # Public variable self.name
= name # Public variable obj = Student(21, "John") print(obj.age) # Output: 21 print(obj.
name) # Output: John
```

Example: Private Access

```
class Student: def __init__(self, age, name): self.__age = age # Private variable obj = S
tudent(21, "John") # This will raise an error print(obj.__age) # AttributeError
```

Example: Protected Access

```
class Student: def __init__(self): self._name = "John" # Protected variable class Subject
(Student): pass obj = Student() print(obj._name) # Output: John
```

Static Methods

Static methods are bound to the class rather than an instance. They don't have access to the instance (`self`) or class (`cls`) data.

Example:

```
class Math: @staticmethod def add(a, b): return a + b result = Math.add(1, 2) print(resul
t) # Output: 3
```

Instance vs. Class Variables

Class Variables

Class variables are shared among all instances of a class.

```
class MyClass: class_variable = 0 def __init__(self): MyClass.class_variable += 1 def pri
nt_class_variable(self): print(MyClass.class_variable) obj1 = MyClass() obj2 = MyClass()
obj1.print_class_variable() # Output: 2
```

Instance Variables

Instance variables are specific to each instance of a class.

```
class MyClass: def __init__(self, name): self.name = name def print_name(self): print(sel
f.name) obj1 = MyClass("John") obj2 = MyClass("Jane") obj1.print_name() # Output: John ob
j2.print_name() # Output: Jane
```

Decorators

In Python, decorators are functions that modify the behavior of other functions or methods without changing their source code.

Example: Logging Decorator

```
def log(func): def wrapper(*args, **kwargs): print(f"Calling {func.__name__} with {arg
s}") return func(*args, **kwargs) return wrapper @log def add(a, b): return a + b print(a
dd(2, 3)) # Logs the function call and result
```

Python Class Methods

In Python, **class methods** are a special type of method defined within a class, designed to operate on the class itself rather than its instances. These methods allow you to define behavior at the class level, offering powerful tools to enhance code modularity, clarity, and maintainability.

What Are Python Class Methods?

A **class method** is a method that belongs to the class rather than to any single instance of the class. Unlike regular instance methods, class methods can modify class state or perform actions related to the class as a whole.

They are defined using the `@classmethod` decorator, and their first parameter is always `cls`, a reference to the class itself.

Example:

```
class Example: @classmethod def describe_class(cls): return f"This is the {cls.__name__}
class."
```

Here, `describe_class` operates on the class itself, not on any individual instance of `Example`.

Why Use Python Class Methods?

Class methods are particularly useful in several scenarios:

1. **Factory Methods**: When you need to create objects in a specific way or with complex initialization logic.
2. **Alternative Constructors**: To allow multiple ways of creating objects from diverse data formats or default configurations.
3. **Class-level Utilities**: For tasks that logically belong to the class but do not involve individual instance data.

Defining and Using Class Methods

To create a class method, use the `@classmethod` decorator and ensure the first parameter is `cls`. For example:

```
class ExampleClass: @classmethod def factory_method(cls, param1, param2): return cls(para
m1 + param2)
```

Here, the `factory_method` creates an instance of `ExampleClass` by combining `param1` and `param2` before passing them to the constructor.

Class Methods as Alternative Constructors

In object-oriented programming, **constructors** initialize new instances of a class. While the default constructor (defined by the `__init__` method) is sufficient for most cases, alternative constructors allow for more flexible and intuitive object creation.

What Are Alternative Constructors?

Alternative constructors use class methods to create objects in ways not directly supported by the default constructor. They can process inputs like strings, dictionaries, or tuples to initialize objects.

Example 1: Creating an Object from a String

Consider a `Person` class:

```
class Person: def __init__(self, name, age): self.name = name self.age = age @classmethod
def from_string(cls, person_str): name, age = person_str.split(',') return cls(name.strip
(), int(age.strip()))
```

This `from_string` method parses a string into a `Person` instance:

```
person = Person.from_string("Alice, 30") print(person.name) # Output: Alice print(person.
age) # Output: 30
```

Example 2: Creating an Object with Default Values

You might want to initialize a `Rectangle` object as a square:

```
class Rectangle: def __init__(self, width, height): self.width = width self.height = heig
ht @classmethod def square(cls, size): return cls(size, size)
```

Here's how to use the alternative constructor:

```
square = Rectangle.square(10) print(square.width, square.height) # Output: 10, 10
```

This separates the logic for creating a square from the default rectangle constructor, making the code more intuitive.

More Examples

1. **Creating Objects from Dictionaries**

```
class Config: def __init__(self, host, port, debug): self.host = host self.port = port se
lf.debug = debug @classmethod def from_dict(cls, config_dict): return cls( config_dict.ge
t('host', 'localhost'), config_dict.get('port', 8000), config_dict.get('debug', False) )
```

This allows you to create a `Config` object directly from a dictionary:

```
config = Config.from_dict({'host': '127.0.0.1', 'port': 8080}) print(config.host) # Outpu
t: 127.0.0.1
```

2. **Handling Multiple Formats**

```
class Date: def __init__(self, year, month, day): self.year = year self.month = month sel
f.day = day @classmethod def from_iso(cls, iso_str): year, month, day = map(int, iso_str.
split('-')) return cls(year, month, day) @classmethod def from_us_format(cls, us_date): m
onth, day, year = map(int, us_date.split('/')) return cls(year, month, day)
```

With this, you can create a `Date` object using different formats:

```
date1 = Date.from_iso("2024-11-19") date2 = Date.from_us_format("11/19/2024")
```

The `dir()` Method in Python

The `dir()` function provides a list of all attributes and methods (including special methods) of an object.

Example:

```
x = [1, 2, 3] # Use dir() to explore the list object print(dir(x))
```

Output:

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc_
_', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__has
h__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__l
en__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '_
_subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Here, `dir(x)` lists all the attributes and methods available for the `x` list object.

The `__dict__` Attribute in Python

The `__dict__` attribute gives us a dictionary of an object's attributes.

Example:

```
class Car: def __init__(self, brand, model): self.brand = brand self.model = model # Crea
te an object car = Car("Toyota", "Corolla") # Access the __dict__ attribute print(car.__d
ict__)
```

Output:

```
{'brand': 'Toyota', 'model': 'Corolla'}
```

This example shows the `__dict__` attribute, which returns a dictionary representation of the object's attributes (`brand` and `model`).

The `help()` Method in Python

The `help()` function provides documentation for an object, such as its attributes and methods.

Example:

```
# Use help() to get information about the str class help(str)
```

Output:

```
Help on class str in module builtins: class str(object) | str(object='') -> str | str(byt
es_or_buffer[, encoding[, errors]]) -> str | | Create a new string object from the given
object.
```

This example shows how `help(str)` provides detailed documentation about the `str` class in Python.

The `super()` Function in Python

The `super()` function is used to call methods from the parent class in a class that inherits from another.

Example:

```
class Animal: def speak(self): print("Animal speaks") class Dog(Animal): def speak(self):
super().speak() # Calls the speak method from Animal print("Dog barks") # Create an objec
t of Dog dog = Dog() # Call the speak method dog.speak()
```

Output:

```
Animal speaks Dog barks
```

In this example, the `Dog` class inherits from `Animal` and calls the `speak` method of the parent class (`Animal`) using `super()`.

The Magic/Dunder Methods in Python

Magic methods (or **dunder methods**) are special methods that allow you to customize an object's behavior. These methods are called automatically when specific operations are performed on objects.

Example 1: `__init__` (Constructor)

```
class Book: def __init__(self, title, author): self.title = title self.author = author #
Create a Book object book = Book("1984", "George Orwell") # Print the book details print
(book.title, book.author)
```

Output:

```
1984 George Orwell
```

The `__init__` method initializes the `Book` object with `title` and `author`.

Example 2: `__str__` Method

```
class Book: def __init__(self, title, author): self.title = title self.author = author de
f __str__(self): return f"'{self.title}' by {self.author}" # Create a Book object book =
Book("1984", "George Orwell") # Print the book (uses __str__ method) print(book)
```

Output:

```
'1984' by George Orwell
```

The `__str__` method provides a human-readable string representation of the `Book` object when printed.

Example 3: `__len__` Method

```
class Box: def __init__(self, items): self.items = items def __len__(self): return len(se
lf.items) # Create a Box object box = Box([1, 2, 3]) # Get the length of the box (uses __
len__ method) print(len(box))
```

Output:

```
3
```

The `__len__` method enables the use of `len()` to get the number of items in the `Box` object.

Example 4: `__add__` Method

```
class Point: def __init__(self, x, y): self.x = x self.y = y def __add__(self, other): re
turn Point(self.x + other.x, self.y + other.y) def __str__(self): return f"({self.x}, {se
lf.y})" # Create two Point objects p1 = Point(2, 3) p2 = Point(4, 5) # Add the points (us
es __add__ method) p3 = p1 + p2 print(p3)
```

Output:

```
(6, 8)
```

The `__add__` method allows us to use the `+` operator to add two `Point` objects together.

Generators in Python

Generators are a special type of function in Python that enable the creation of iterable sequences of values, generated one at a time as needed. Unlike lists or tuples, which store all elements in memory, generators produce values **on-the-fly**, making them particularly useful for working with large datasets or sequences where memory efficiency is crucial.

How Generators Work

Generators use the `yield` statement instead of `return`. When a generator function is called, it does not execute its body immediately. Instead, it returns a **generator object**. Each call to the generator retrieves the next value and resumes execution from where it last yielded.

Creating a Generator

A generator function is defined just like a regular function, but it uses `yield` to return values one by one.

Example:

```
def my_generator(): for i in range(5): yield i # Yield one value at a time gen = my_generator() # Create the generator object print(next(gen)) # Output: 0 print(next(gen)) # Output: 1 print(next(gen)) # Output: 2
```

Explanation:

1. The `my_generator` function generates numbers from 0 to 4.
2. Each call to `next(gen)` resumes the generator's execution, returning the next value and pausing at `yield`.

If you call `next(gen)` after all values are exhausted, it raises a `StopIteration` exception.

Using Generators in Loops

Generators integrate seamlessly with `for` loops, which handle `StopIteration` automatically.

Example:

```
def my_generator(): for i in range(5): yield i for value in my_generator(): print(value) # Output: # 0 # 1 # 2 # 3 # 4
```

Key Point:

The generator does not store all values in memory—it generates each value only when needed.

Generator Expressions

You can create a generator object using a compact generator expression, similar to a list comprehension but with parentheses instead of square brackets.

Example:

```
gen = (x**2 for x in range(5)) for value in gen: print(value) # Output: # 0 # 1 # 4 # 9 # 16
```

Key Difference from List Comprehension:

- List comprehension `[x**2 for x in range(5)]` creates and stores the entire list in memory.
- Generator expression `(x**2 for x in range(5))` creates values on demand, saving memory.

The Walrus Operator in Python

Introduced in Python 3.8, the **Walrus Operator** (`:=`) is a feature that allows you to assign a value to a variable as part of an expression. This feature can make code more concise and reduce repetition, especially in loops and conditional statements.

What Does the Walrus Operator Do?

The Walrus Operator assigns a value to a variable and evaluates it in the same expression. This eliminates the need for separate lines for assignment and evaluation.

Syntax:

```
variable := expression
```

Common Use Cases

1. In While Loops

You can use the Walrus Operator to assign a value within the loop's condition, avoiding redundant calculations.

Example:

```
numbers = [1, 2, 3, 4, 5] while (n := len(numbers)) > 0: print(numbers.pop()) # Removes a
nd prints the last element
```

Explanation:

- `(n := len(numbers))` assigns the length of `numbers` to `n` .
- The condition checks if `n > 0` to continue looping.

2. In If Statements

The Walrus Operator can evaluate and assign a value within an `if` condition.