Python Documentation

Introduction to Python

What is Python?

Python is a high-level programming language first developed by Guido van Rossum, a Dutch programmer, and released in 1991. Today, it's maintained and developed by the Python Software Foundation.

Key Characteristics

1. **Human-Readable**: Python's syntax closely resembles the English language, making it intuitive to read and write.
2. **Object-Oriented**: Python supports object-oriented programming (OOP), allowing data and code to be organized into reusable structures called objects.
3. **Strong and Dynamic Typing**: While Python enforces strict type rules, it also allows variable types to change during runtime, providing both reliability and flexibility.

Common Applications

- Data Science
- Robotics
- Internet of Things (IoT)
- Artificial Intelligence (AI)
- Software Development

Why Learn Python?

1. **Beginner-Friendly**: Python's clear syntax and readability make it an excellent choice for those new to programming.
2. **Industry Popularity**: It's widely adopted in various tech sectors and consistently ranks as one of the most popular programming languages.
3. **Versatility**: Python supports a wide range of programming paradigms and can be used in numerous fields.
4. **Extensive Ecosystem**: A vast collection of libraries and frameworks supports Python, enabling rapid development across various domains.

Python Variables and Data Types

Python variables are objects used to store data values for computation. They function like containers, holding information that can be accessed and manipulated throughout your code.

Key points about Python variables:

- No explicit declaration required

- Created upon value assignment
- Data types are inferred automatically
- Act as references to memory locations

Assigning Values to Variables

In Python, you can assign values to variables using the = operator:

```
x = 5
name = "Alice"
pi = 3.14
```

Python Primitive Data Types

Python has several built-in data types. Here are the most common ones:

1. **Integers (int)**: Whole numbers that can be positive, negative, or zero.

```
age = 22
```

2. **Strings (str)**: Sequences of characters enclosed in single or double quotes. Can include letters, numbers, and special characters.

```
name = "Jack"
```

3. **Floats (float)**: Real numbers with decimal points.

```
height = 1.75
```

4. **Booleans (bool):** Represent truth values with two constant objects: True and False.

```
is_student = True
```

Python Operators

Python operators are special symbols that perform operations on variables and values. They are essential for building both simple and complex algorithms. These operators form the foundation of Python programming, allowing you to perform calculations, make decisions, and control the flow of your programs. There are several types of

operators in Python, with the four most common being arithmetic, assignment, comparison, and logical operators.

1. Arithmetic operators perform mathematical calculations.

Operator	Description	Example
+	Addition	5 + 3 = 8
-	Subtraction	5 - 3 = 2
*	Multiplication	5 * 3 = 15
/	Division	5 / 3 = 1.67
//	Floor Division	5 // 3 = 1
%	Modulus (remainder)	5 % 3 = 2
**	Exponentiation	5 ** 3 = 125

2. Assignment operators are used to assign values to variables. **Let's assume x = 35 for all examples.**

Operator	Description	Example	Equivalent to
=	Assign value	x = 35	x = 35
+=	Add and assign	x += 5	x = x + 5
-=	Subtract and assign	x -= 5	x = x - 5
*=	Multiply and assign	x *= 2	x = x * 2
/=	Divide and assign	x /= 5	x = x / 5

3. Comparison operators are used to compare values. They return a Boolean result (True or False).

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

4. Logical operators are used to combine conditional statements. They also return a Boolean value.

Operator	Description	Example
and	True if both statements are true	(5 > 3) and (5 < 10) (True)
or	True if at least one statement is true	(5 > 3) or (5 > 10) (True)
not	Inverts the result	not(5 > 10) (True)

Python Conditionals

Conditional statements in Python are used to control the flow of a program based on certain conditions. They allow the program to make decisions and execute different code blocks depending on whether specific conditions are true or false.

The if...elif...else Structure

One of the most common control structures in Python is the if...elif...else statement. This selection control structure determines the program's path based on the conditions met during execution. Here is an example of the basic 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 conditions are False
```

Importance of Indentation

Unlike many other programming languages that use curly braces to define code blocks, Python uses indentation. Proper indentation is crucial for the correct execution of your code.

```
x = 10

if x > 5:
    print("x is greater than 5")
    if x > 8:
        print("x is also greater than 8")
else:
    print("x is not greater than 5")
```

In this example, the indentation clearly shows which code belongs to each condition.

Expanded if...elif...else Example

The below code demonstrates how multiple conditions can be checked sequentially, with only the first true condition's code block being executed.

Key Points to Remember:

- Conditionals help create dynamic, responsive programs.
- The if statement can be used alone or with elif and else.

- You can have multiple elif statements but only one else.
- Indentation defines the scope of each code block.
- Conditions are evaluated in order from top to bottom.

```
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

Python Functions

A Python function is a reusable block of code that performs a specific task. Functions are designed to take input, process it, and optionally return a result. They allow you to organize your code into manageable, reusable pieces, enhancing readability and efficiency.

Why Use Python Functions?

Programmers use functions for several reasons:

1. **Code Decomposition**: Breaking down complex problems into smaller, manageable parts.
2. **Reusability**: Writing code once and using it multiple times with different inputs.
3. **Extensibility**: Easily adding new features or modifying existing ones without affecting the entire program.

Basic Syntax of a Python Function

- def keyword is used to define a function
- function_name is the name you give to your function
- Parameters are optional and enclosed in parentheses
- The function body is indented
- return statement is optional and used to send a result back to the caller

```
def function_name (parameter1, parameter2):
    # Function body
    # Code to be executed
    return result  # Optional
```

A Simple Function Example

Here's a basic example of a function that calculates the area of a rectangle:

```
def calculate_rectangle_area(length, width):
    area = length * width
    return area

# Using the function
rectangle1_area = calculate_rectangle_area(5, 3)
print(f"The area of the rectangle is: {rectangle1_area}")

rectangle2_area = calculate_rectangle_area(7, 4)
print(f"The area of another rectangle is: {rectangle2_area}")
```

Here is the output for the above function example:

```
The area of the rectangle is: 15
The area of another rectangle is: 28
```

In the above example:

- We define a function called calculate_rectangle_area that takes two parameters: length and width.
- The function calculates the area and returns the result.
- We call the function twice with different inputs, demonstrating reusability.
- The function encapsulates the area calculation logic, making our code cleaner and more organized.

Iteration in Python

In programming, iteration refers to the process of repeating a set of instructions or a block of code multiple times. Iterative control structures are used to execute a block of code repeatedly until a specific condition is met or changes.

Iteration is essential for tasks that require repetitive actions. For example, if your goal is to flip a coin until you get three heads in a row, you would continue flipping the coin (repeating the action) until this condition is met.

Python provides two main types of iterative structures: for loops and while loops.

For Loop

A for loop is used to iterate over a sequence (such as a list, tuple, string, or range) or other iterable objects.

```python
# Printing numbers from 1 to 5
for number in range(1, 6):
    print(number)
```

Here is the output for the above for loop example:

```
1
2
3
4
5
```

While Loop

A while loop repeats a block of code as long as a given condition is true.

```python
# Flipping a coin until we get 3 heads in a row
import random

heads_count = 0
flips = 0

while heads_count < 3:
    flip = random.choice(['heads', 'tails'])
    flips += 1

    if flip == 'heads':
        heads_count += 1
        print(f"Flip {flips}: Heads")
    else:
        heads_count = 0
        print(f"Flip {flips}: Tails")

print(f"It took {flips} flips to get 3 heads in a row.")
```

Here is the output for the above while loop example. Note that the output will vary because of randomness:

```
Flip 1: Tails
Flip 2: Heads
Flip 3: Heads
Flip 4: Tails
Flip 5: Heads
Flip 6: Heads
Flip 7: Heads
It took 7 flips to get 3 heads in a row.
```

In the above example, the while loop continues until we get three heads in a row. The number of iterations (coin flips) is not predetermined and depends on the random outcomes.

Both for and while loops offer powerful ways to implement iteration in Python, allowing you to automate repetitive tasks efficiently.

Python Data Structures: Lists and Dictionaries

Python Lists

Python lists are versatile data structures used to store multiple items in a single variable. They are ordered collections of elements, accessible by index starting from 0.

List Properties:

- Ordered: Elements maintain their order
- Mutable: Can be modified after creation
- Allow duplicates: Can contain multiple identical items
- Indexed: Elements are accessed by their position (index)

```
fruits = ["apple", "banana", "cherry", "apple"]
print(fruits)
print(fruits[1])  # Accessing by index
fruits[1] = "blueberry"  # Modifying an element
print(fruits)
```

Here is the output for the above example:

```
 ['apple', 'banana', 'cherry', 'apple']
banana
['apple', 'blueberry', 'cherry', 'apple']
```

Common List Operations:

```
fruits.append("orange")  # Add an item
fruits.remove("cherry")  # Remove an item
print(len(fruits))  # Get the length of the list
```

Python Dictionaries

Python dictionaries store data in key-value pairs. Unlike lists, dictionaries use keys (which can be strings, numbers, or tuples) to access values, rather than numerical indices.

Dictionary Properties:

- Ordered (as of Python 3.7+): Keys maintain their insertion order
- Mutable: Can be modified after creation
- No duplicates: Each key must be unique
- Key-based access: Values are accessed by their associated keys

Dictionary Example:

```
student = {
    "name": "John Doe",
    "age": 20,
    "major": "Computer Science",
    "gpa": 3.8
}

print(student)
print(student["major"])  # Accessing by key
student["age"] = 21  # Modifying a value
student["year"] = "Junior"  # Adding a new key-value pair
print(student)
```

Here is the output for the above example:

```
{'name': 'John Doe', 'age': 20, 'major': 'Computer Science', 'gpa': 3.8}
Computer Science
{'name': 'John Doe', 'age': 21, 'major': 'Computer Science', 'gpa': 3.8, 'year': 'Junior'}
```

Common Dictionary Operations:

```
print(student.keys())  # Get all keys
print(student.values())  # Get all values
print(student.items())  # Get all key-value pairs
student.pop("gpa")  # Remove a key-value pair
```

Both lists and dictionaries are fundamental data structures in Python, each with its own use cases. Lists are great for ordered collections of items, while dictionaries excel at storing and retrieving data with meaningful keys.