init

01 August 2024 12:35

What is __init__ keyword in Python?

- The __init__ method is known as a **constructor** in object-oriented programming (OOP) terminology.
- It is used to **initialize an object's state** when it is created.
- Automatically called when a new instance of a class is instantiated.
- Purpose:
- Assign values to object properties.
- Perform any initialization operations.

Example:

```
class BookShop:
    def __init__(self, title):
        self.title = title

    def book(self):
        print('The title of the book is', self.title)

b = BookShop('Sandman')
b.book()  # Output: The title of the book is Sandman
```

__init__ , What does _init_ method do?

02 August 2024 13:22

The __init__ method in Python is a **constructor**. When you create an object from a class, __init__ initializes its state (instance variables). It's automatically called during object creation, allowing you to set initial values and perform any necessary setup

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

p = Person("Alice")
print(p.name)  # Output: Alice
```

Self-Keyword

01 August 2024 12:37

- In Python, self refers to the instance of a class that is currently being used.
- It is used within instance methods to access and modify the object's attributes.
- When you call a method on an object, the object itself is automatically passed as the first argument (using self).

Example:

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

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

p = Person("Alice")
p.greet()  # Output: Hello, my name is Alice
```

lambda function

01 August 2024 13:25

What is a lambda function?

- A **lambda function** (also known as an anonymous function) is a small, one-line function defined using the lambda keyword.
- It can take any number of arguments but has only one expression.
- Useful for short, simple operations.

Example :

```
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Difference between lambda and normal function

01 August 2024 13:27

Lambda functions:

- Defined using **lambda**.
- Limited to a single expression.
- No statements (like assignments or loops).
- Often used for short, inline operations.

Normal functions:

- Defined using **def.**
- Can have multiple expressions and statements.
- Named and reusable.
- Suitable for more complex logic

Generators

01 August 2024 13:29

What are generators? When to use?

- **Generators** are functions that produce a sequence of values on-the-fly.
- They use the yield keyword instead of return.
- Ideal for large data sets or infinite sequences.
- Use when memory efficiency is crucial.

Example:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib_gen = fibonacci()
for _ in range(10):
    print(next(fib_gen))
```

Decorators

01 August 2024 13:45

What are decorators? When to use?

- **Decorators** modify or enhance functions/methods.
- Used for logging, authentication, memoization, etc.
- Example:

```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args: {args}, kwargs: {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned: {result}")
        return result
    return wrapper

@log_function_call
def add(a, b):
    return a + b

add(3, 5)  # Output: Calling add with args: (3, 5), add returned: 8
```

When to use dictionaries

- Use dictionaries when you need key-value pairs for efficient lookups.
- Ideal for storing data with associated labels (e.g., phone book).

Python as a compiled or interpreted language

01 August 2024 13:29

1. **Python as a compiled or interpreted language:**

- Python is **interpreted**.
- It compiles source code to an intermediate form (bytecode) and then interprets that bytecode.
- No separate compilation step.
- Easier to debug and more flexible.

lists and tuples in Python

01 August 2024 13:34

Lists:

- Mutable (can be modified after creation).
- Ordered collection.
- Use square brackets ([]).

Tuples:

- Immutable (cannot be changed after creation).
- Ordered collection.
- Use parentheses (())

lists and sets in Python

01 August 2024 13:36

Difference between lists and sets in Python:

- **Lists**:
- Ordered collection.
- Allows duplicates.
- Mutable.

Sets:

- Unordered collection of unique elements.
- No duplicates.
- Mutable (can add/remove elements).

Iterators

01 August 2024 13:47

What are iterators?

- **Iterators** allow sequential access to elements.
- Implement __iter__() and __next__() methods.
- Used in for loops and comprehensions.

Slicing

01 August 2024 13:48

- **Slicing** extracts a portion of a sequence (e.g., list, string).
- Syntax: sequence[start:stop:step].
- Example: my_list[1:5] gets elements from index 1 to 4.

Note:

- If start is omitted, it defaults to the beginning of the sequence.
- If stop is omitted, it defaults to the end of the sequence.
- Negative indices count from the end (e.g., -1 refers to the last element).

Long Answer :

- **Slicing** is a technique to extract a portion of a sequence (such as a list, string, or tuple) by specifying a range of indices.
- Syntax: sequence[start:stop:step]
- start: The index where the slice begins (inclusive).
- stop: The index where the slice ends (exclusive).
- step: Optional; specifies the step size (default is 1).

Example:

```
my_list = [10, 20, 30, 40, 50]
sliced_list = my_list[1:4]  # Elements from index 1 to 3 (20, 30, 40)
print(sliced_list)

my_string = "Hello, World!"
sliced_string = my_string[7:12]  # "World"
print(sliced_string)
```

mutable and immutable

01 August 2024 14:06

What is mutable and immutable?

mutable and **immutable** refer to the behaviour of objects with respect to modification after creation:

1. **Mutable**:
- Objects that can be modified after creation.
- Examples: **lists**, **dictionaries**, and **sets**.
- You can change their contents (add, remove, or modify elements).

2. **Immutable**:
- Objects that cannot be changed after creation.
- Examples: **tuples**, **strings**, and **integers**.
- Once created, their values remain fixed.

For instance:

- A list can be modified by appending elements or changing existing ones.
- A tuple, once created, cannot be altered (you can’t add or remove elements).

Python is single thread or multithread?

01 August 2024 14:10

Python is **single-threaded**, but it is **capable of multi-threading**. Let me explain:

- Python supports the creation and management of multiple threads, which means you can create and work with multiple threads within a Python program.
- However, there's a catch: **the Global Interpreter Lock (GIL)**. The GIL prevents multiple threads from running Python code at the same time in the (Python implementation (the most common one you'll encounter).
- Essentially, even though Python uses system threads, it can't fully utilize more than one of the available CPU cores due to the GIL.
- For I/O-bound tasks (such as reading files or making network requests), Python threads still work well. However, for CPU-bound tasks (like heavy computations), the GIL can cause limitations and prevent true parallel execution.

In summary, Python threads are there, but their effectiveness depends on the type of task you're dealing with!

GIL

01 August 2024 14:11

Short Answer:
GIL ensures thread safety but restricts performance in CPU-bound scenarios. Understanding its impact helps design efficient Python programs!

The **Global Interpreter Lock (GIL)** in Python is a mutex (or lock) that ensures only **one thread** can hold control of the Python interpreter at any given time

1. **Purpose of the GIL:**
- Python uses **reference counting** for memory management. Objects have a reference count that tracks how many references point to them.
- The GIL protects this reference count from race conditions, preventing simultaneous modifications by multiple threads.
- Without the GIL, memory leaks or incorrect memory releases could occur, leading to unexpected bugs.

2. **How the GIL Works:**

- The GIL is a **single lock** on the interpreter itself.
- Any Python bytecode execution requires acquiring this lock.
- This makes CPU-bound Python programs effectively **single-threaded**, even on multi-core processors.
- While it prevents deadlocks and simplifies memory management, it limits true parallelism.

3. **Impact on Performance:**
- For I/O-bound tasks (e.g., reading files), Python threads work well despite the GIL.
- However, CPU-bound tasks suffer due to the GIL, as only one thread can execute Python code at a time.
- To achieve true parallelism, consider using **multiprocessing** (which creates separate processes, each with its own interpreter).

What you don’t like about python?

02 August 2024 13:02

1. **Global Interpreter Lock (GIL)**:
   - The GIL limits true multithreading in Python.
   - CPU-bound tasks may not benefit from parallel execution due to the GIL.
   - However, for I/O-bound tasks, Python threads work well.
2. **Performance**:
   - Python can be slower than compiled languages like C++ for certain tasks.
   - Interpreted languages generally have some overhead compared to compiled ones.
3. **Whitespace Sensitivity**:
   - Python’s strict indentation rules can be frustrating for some developers.
   - Proper indentation is crucial for code correctness.
4. **Lack of Static Typing (in Python 2)**:
   - Python 2 didn’t have native type hints.
   - Python 3 introduced type annotations, but it’s still optional.
5. **Global State**:
   - Python allows global variables, which can lead to unexpected behavior.
   - Proper scoping and encapsulation are essential.
6. **Verbosity**:
   - Some find Python code verbose compared to more concise languages.
   - However, readability is a core Python principle.

I love Python for its readability, community, and extensive libraries. Python’s strengths often outweigh its limitations

List Comprehension?

02 August 2024 13:10

Short

list comprehension is a concise way to create a new list in Python based on the values of an existing list. It allows you to apply an expression to each element of the original list and generate a new list with minimal code.

Think of it as a shortcut for creating lists

Long

List comprehension in Python is an elegant and concise way to create new lists based on the values of an existing iterable (such as a list, tuple, or range). It allows you to generate a new list by applying an expression to each element of the original iterable.

Basic Syntax:

- A list comprehension consists of square brackets [...].
- Inside the brackets, you define an **expression** that determines what you want to include in the new list.
- You also specify an **iterable** (e.g., an existing list) from which you'll extract elements.

Ex:

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

Components of List Comprehension:

- **Expression**: The value you want to include in the new list (e.g., x ** 2 in the example above).
- **Element**: Each element from the iterable (e.g., x in the example).
- **Iterable**: The original sequence of elements (e.g., range(10) in the example).

Filtering with Conditions:

- You can add a condition (using an if statement) to filter elements.
- Ex:

```
# Create a list of even squares (only for even numbers)
even_squares = [x ** 2 for x in range(10) if x % 2 == 0]
print(even_squares)  # Output: [0, 4, 16, 36, 64]
```

Manipulating the Expression:

- You can modify the expression itself (e.g., convert to uppercase, change values, etc.).

  Ex

```
# Create a list of uppercase fruit names
fruits = ["apple", "banana", "cherry"]
uppercase_fruits = [x.upper() for x in fruits]
print(uppercase_fruits)  # Output: ['APPLE', 'BANANA', 'CHERRY']
```

Dunder methods

02 August 2024 13:19

Dunder methods, also known as **magic methods**, are special methods in Python that have names surrounded by double underscores (e.g., __init__, __str__, __add__). They allow you to customize how objects of a class behave in specific situations or interact with built-in Python functions and operators. Here are some common examples:

1. __init__(self, ...): Constructor method called during object creation.
2. __str__(self): Defines behavior for str() (human-readable representation).
3. __add__(self, other): Implements behavior for the + operator (addition).
4. __repr__(self): Defines the unambiguous representation of an object.
5. __len__(self): Returns the length of an object (used with len()).

dunder methods make Python classes more powerful and flexible!

Difference between array and numpy library

02 August 2024 13:25

1. **Python** array:
- Part of the Python standard library.
- Lightweight and basic.
- Good for simple arrays of data types (e.g., integers, floats).
- Efficient for I/O and basic storage.
- Doesn't provide advanced numerical operations.
2. **NumPy (**numpy.array**)**:
- External library (requires installation).
- Powerful N-dimensional array object.
- Used in scientific computing, linear algebra, and more.
- Efficient calculations with large data sets.
- Supports various data types and advanced functions.

In summary, if you need basic arrays, use Python's array. For more complex tasks and numerical work, NumPy is your go-to!