The Python Programming Language Book

By : Khawar Nehal

Date : 6 March 2024
Urdu version : 7 March 2024

پائتھون پروگرامنگ لینگویج بک
منجانب: خاور نہال

تاریخ: 6 مارچ 2024

An introduction to Python and an overview of programming languages.

Introduction to Python:

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

Key Features of Python:

1. **Simple and Readable Syntax**: Python's syntax is designed to be intuitive and easy to read, which makes it an excellent language for beginners and experts alike.

2. **Interpreted Language**: Python is an interpreted language, meaning that code written in Python is executed line by line without the need for compilation. This makes the development process faster and more interactive.

3. **Dynamic Typing**: Python uses dynamic typing, which means you don't need to specify variable types explicitly. Variable types are inferred at runtime, making Python code more flexible and concise.

4. **Rich Standard Library**: Python comes with a vast standard library that provides modules and packages for various tasks, such as file I/O, networking, web development, and more. This rich ecosystem reduces the need for external dependencies.

5. **Cross-Platform**: Python is cross-platform, meaning it runs on multiple operating systems, including Windows, macOS, and Linux. This portability makes it easy to write code that can be run on different platforms without modification.

6. **Object-Oriented**: Python supports object-oriented programming (OOP) paradigms, allowing you to create classes and objects, encapsulate data, and implement inheritance and polymorphism.

7. **Community and Ecosystem**: Python has a large and active community of developers who contribute to its ecosystem by creating libraries, frameworks, and tools. This vibrant community ensures that Python remains relevant and up-to-date.

Common Use Cases for Python:

- **Web Development**: Frameworks like Django and Flask are popular choices for building web applications. - **Data Science and Machine Learning**: Python has become the de facto language for data science and machine learning due to libraries like NumPy, Pandas, and TensorFlow. - **Scripting and Automation**: Python's simplicity and versatility make it well-suited for writing scripts to automate tasks. - **Game Development**: Python is used in game development, often for prototyping and scripting within game engines. - **Desktop GUI Applications**: Libraries like Tkinter and PyQt allow developers to create desktop GUI applications using Python.

Now, let's move on to the overview of programming languages.

Overview of Programming Languages:

Programming languages are tools used to instruct computers to perform specific tasks. There are thousands of programming languages, each with its own syntax, semantics, and use cases. Here are some common categories of programming languages:

1. **Low-Level Languages**: These languages are close to machine code and provide little abstraction from the hardware. Examples include Assembly language and machine code.

2. **High-Level Languages**: High-level languages provide more abstraction from the hardware and are closer to human language. Examples include Python, Java, C++, and Ruby.

3. **Interpreted Languages**: In interpreted languages, code is executed line by line without the need for compilation. Examples include Python, JavaScript, and Ruby.

4. **Compiled Languages**: In compiled languages, code is translated into machine code before execution. Examples include C, C++, and Rust.

5. **Scripting Languages**: Scripting languages are often used for automation and rapid prototyping. Examples include Python, Perl, and Bash.

6. **Functional Languages**: Functional languages emphasize the use of functions as the primary building blocks of programs. Examples include Haskell, Lisp, and Erlang.

7. **Object-Oriented Languages**: Object-oriented languages organize code around objects and classes. Examples include Java, Python, and C++.

8. **Procedural Languages**: Procedural languages organize code around procedures or routines. Examples include C, Pascal, and BASIC.

Each programming language has its strengths and weaknesses, and the choice of language depends on factors such as the nature of the project, performance requirements, and personal preference. Learning multiple languages can broaden your understanding of programming concepts and make you a more versatile developer.

کا تعارف اور پروگرامنگ زبانوں کا ایک جائزہ۔ Python
:ازگر کا تعارف
ایک اعلیٰ سطحی، تشریح شدہ پروگرامنگ زبان ہے جو اپنی سادگی اور پڑھنے کی اہلیت کے لیے مشہور ہے۔ اسے Python کا ڈیزائن فلسفہ اس کے نمایاں Python نے بنایا تھا اور اسے پہلی بار 1991 میں ریلیز کیا گیا تھا۔ Guido van Rossum وائٹ اسپیس کے قابل ذکر استعمال کے ساتھ کوڈ پڑھنے کی اہلیت پر زور دیتا ہے۔
:کی اہم خصوصیات Python
1. سادہ اور پڑھنے کے قابل نحو: ازگر کے نحو کو بدیہی اور پڑھنے میں آسان بنانے کے لیے ڈیزائن کیا گیا ہے، جو اسے ابتدائی اور ماہرین دونوں کے لیے ایک بہترین زبان بناتا ہے۔
2. ترجمانی شدہ زبان: Python ایک تشریح شدہ زبان ہے، مطلب یہ ہے کہ Python میں لکھا ہوا کوڈ تالیف کی ضرورت کے بغیر لائن بہ لائن عمل میں آتا ہے۔ یہ ترقی کے عمل کو تیز تر اور زیادہ انٹرایکٹو بناتا ہے۔
3. ڈائنامک ٹائپنگ: Python ڈائنامک ٹائپنگ کا استعمال کرتا ہے، جس کا مطلب ہے کہ آپ کو متغیر کی قسمیں واضح طور پر بتانے کی ضرورت نہیں ہے۔ رن ٹائم پر متغیر اقسام کا اندازہ لگایا جاتا ہے، جس سے ازگر کوڈ زیادہ لچکدار اور جامع ہوتا ہے۔
4. امیر معیاری لائبریری: پائتھون ایک وسیع معیاری لائبریری کے ساتھ آتا ہے جو مختلف کاموں، جیسے فائل I/O، نیٹ ورکنگ، ویب ڈویلپمنٹ، اور بہت کچھ کے لیے ماڈیولز اور پیکجز فراہم کرتا ہے۔ یہ بھرپور ماحولیاتی نظام بیرونی انحصار کی ضرورت کو کم کرتا ہے۔
5. کراس پلیٹ فارم: ازگر کراس پلیٹ فارم ہے، یعنی یہ ونڈوز، میک او ایس اور لینکس سمیت متعدد آپریٹنگ سسٹمز پر چلتا ہے۔ یہ پورٹیبلٹی کوڈ لکھنا آسان بناتی ہے جسے بغیر کسی ترمیم کے مختلف پلیٹ فارمز پر چلایا جا سکتا ہے۔
6. پیراڈیمز کو سپورٹ کرتا ہے، جس سے آپ کو کلاسز اور (OOP) آبجیکٹ اورینٹڈ: پائتھون آبجیکٹ اورینٹڈ پروگرامنگ آبجیکٹ بنانے، ڈیٹا کو انکیپسلیٹ کرنے، اور وراثت اور پولیمورفزم کو نافذ کرنے کی اجازت ملتی ہے۔
7. کمیونٹی اور ایکو سسٹم: Python کے پاس ڈیولپرز کی ایک بڑی اور فعال کمیونٹی ہے جو لائبریری، فریم ورک اور ٹولز بنا متعلقہ اور Python کر اس کے ماحولیاتی نظام میں اپنا حصہ ڈالتی ہے۔ یہ متحرک کمیونٹی اس بات کو یقینی بناتی ہے کہ اپ ٹو ڈیٹ رہے۔
:کے لیے عام استعمال کے معاملات Python
- ویب ڈویلپمنٹ: Django اور Flask جیسے فریم ورک ویب ایپلیکیشنز بنانے کے لیے مقبول انتخاب ہیں۔ - ڈیٹا سائنس اور مشین لرننگ: Python جیسی لائبریریوں کی وجہ سے TensorFlow، Pandas، اور NumPy کے لیے اصل زبان بن گئی ہے۔ - اسکرپٹ اور آٹومیشن: ازگر کی سادگی اور استعداد اسے کاموں کو خودکار کرنے کے لیے گیم ڈویلپمنٹ میں استعمال ہوتا ہے، Python :اسکرپٹ لکھنے کے لیے اچھی طرح سے موزوں بناتی ہے۔ - گیم ڈویلپمنٹ جیسی PyQt اور Tkinter :ایپلی کیشنز GUI اکثر گیم انجنوں میں پروٹو ٹائپنگ اور اسکرپٹنگ کے لیے۔ - ڈیسک ٹاپ ایپلی کیشنز بنانے کی اجازت دیتی ہیں۔ GUI کا استعمال کرتے ہوئے ڈیسک ٹاپ Python لائبریریاں ڈیولپرز کو
اب، پروگرامنگ زبانوں کے جائزہ کی طرف چلتے ہیں۔
:پروگرامنگ زبانوں کا جائزہ
پروگرامنگ زبانیں وہ ٹولز ہیں جو کمپیوٹر کو مخصوص کاموں کو انجام دینے کی ہدایت دینے کے لیے استعمال ہوتے ہیں۔ پروگرامنگ کی ہزاروں زبانیں ہیں، جن میں سے ہر ایک کا اپنا نحو، الفاظ اور استعمال کے معاملات ہیں۔ یہاں پروگرامنگ :زبانوں کے کچھ عام زمرے ہیں
1. کم سطحی زبانیں: یہ زبانیں مشین کوڈ کے قریب ہیں اور ہارڈ ویئر سے تھوڑا سا خلاصہ فراہم کرتی ہیں۔ مثالوں میں اسمبلی کی زبان اور مشین کوڈ شامل ہیں۔
2. اعلیٰ سطحی زبانیں: اعلیٰ سطحی زبانیں ہارڈ ویئر سے زیادہ خلاصہ فراہم کرتی ہیں اور انسانی زبان کے قریب ہوتی ہیں۔ مثالوں میں Python، Java، C++ اور Ruby شامل ہیں۔
3. تشریح شدہ زبانیں: ترجمانی شدہ زبانوں میں، کوڈ کو تالیف کی ضرورت کے بغیر لائن بہ لائن عمل میں لایا جاتا ہے۔ مثالوں میں Python، JavaScript اور Ruby شامل ہیں۔
4. مرتب شدہ زبانیں: مرتب شدہ زبانوں میں، کوڈ کو عملدرآمد سے پہلے مشین کوڈ میں ترجمہ کیا جاتا ہے۔ مثالوں میں C، C++، اور Rust شامل ہیں۔
5. سکرپٹ کی زبانیں: سکرپٹ کی زبانیں اکثر آٹومیشن اور تیز رفتار پروٹو ٹائپنگ کے لیے استعمال ہوتی ہیں۔ مثالوں میں Python، Perl، اور Bash شامل ہیں۔
6. فنکشنل لینگویجز: فنکشنل لینگویجز پروگراموں کے بنیادی بلڈنگ بلاکس کے طور پر فنکشن کے استعمال پر زور دیتی ہیں۔ مثالوں میں ہاسکل، لسپ، اور ایرلنگ شامل ہیں۔
7. آبجیکٹ پر مبنی زبانیں: آبجیکٹ پر مبنی زبانیں اشیاء اور کلاسوں کے ارد گرد کوڈ کو منظم کرتی ہیں۔ مثالوں میں Java، Python، اور C++ شامل ہیں۔

8. طریقہ کار کی زبانیں: طریقہ کار کی زبانیں طریقہ کار یا معمولات کے ارد گرد کوڈ کو منظم کرتی ہیں۔ مثالوں میں C، BASIC اور Pascal شامل ہیں۔

ہر پروگرامنگ زبان کی اپنی خوبیاں اور کمزوریاں ہوتی ہیں، اور زبان کا انتخاب پروجیکٹ کی نوعیت، کارکردگی کے تقاضوں اور ذاتی ترجیح جیسے عوامل پر منحصر ہوتا ہے۔ متعدد زبانیں سیکھنا پروگرامنگ کے تصورات کے بارے میں آپ کی سمجھ کو وسیع کر سکتا ہے اور آپ کو زیادہ ورسٹائل ڈویلپر بنا سکتا ہے۔

Introduction to Python and its features

Python is a high-level programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

Features of Python:

1. **Simple and Readable Syntax**: Python code is easy to read and write, making it suitable for beginners and experienced programmers alike. Its syntax is clean and uncluttered, which enhances code readability and reduces the cost of program maintenance.

2. **Interpreted and Interactive**: Python is an interpreted language, meaning that code execution happens line by line, making debugging and testing easier. It also supports an interactive mode, allowing users to test snippets of code quickly without the need for compiling.

3. **Dynamic Typing**: Python uses dynamic typing, which means you don't need to specify variable types explicitly. Variables can change types as needed during execution, providing flexibility but also requiring careful attention to variable types during development.

4. **High-level Language**: Python abstracts many complex details away from the programmer, providing built-in high-level data structures and a vast standard library. This allows developers to focus more on solving problems rather than worrying about memory management or low-level details.

5. **Cross-platform**: Python is available on various platforms such as Windows, macOS, and Linux, making it highly portable. Python code written on one platform can easily run on another platform with minimal modifications.

6. **Extensive Standard Library**: Python comes with a comprehensive standard library that provides support for various tasks such as string operations, file I/O, networking, and more. This reduces the need for external libraries for many common programming tasks.

7. **Object-Oriented**: Python supports object-oriented programming paradigms, allowing developers to create and use objects to model real-world entities. This approach promotes code reusability, modularity, and scalability.

8. **Large and Active Community**: Python has a large and vibrant community of developers, which contributes to its extensive documentation, numerous third-party libraries, and active support forums. This community aspect fosters collaboration and helps developers solve problems more efficiently.

9. **Open Source**: Python is open-source, meaning that its source code is freely available and can be modified and distributed by anyone. This encourages innovation and collaboration within the Python community.

10. **Versatile and Scalable**: Python is versatile and can be used for various purposes such as web development, data analysis, artificial intelligence, machine learning, scientific computing, and more. Its scalability allows it to be used for small scripts to large-scale applications.

These features make Python a popular choice for a wide range of applications, from simple scripting tasks to complex software development projects. Its simplicity, readability, and extensive ecosystem make it an excellent language for both beginners and experienced developers.

ازگر اور اس کی خصوصیات کا تعارف
Python ایک اعلیٰ سطحی پروگرامنگ زبان ہے جو اپنی سادگی اور پڑھنے کی اہلیت کے لیے مشہور ہے۔ اسے Guido van Rossum نے بنایا تھا اور اسے پہلی بار 1991 میں ریلیز کیا گیا تھا۔ Python کا ڈیزائن فلسفہ اس کے نمایاں وائٹ اسپیس کے قابل ذکر استعمال کے ساتھ کوڈ پڑھنے کی اہلیت پر زور دیتا ہے۔
ازگر کی خصوصیات:
1. سادہ اور پڑھنے کے قابل نحو: ازگر کوڈ پڑھنے اور لکھنے میں آسان ہے، جو اسے ابتدائی اور تجربہ کار پروگرامرز کے لیے یکساں موزوں بناتا ہے۔ اس کا نحو صاف اور بے ترتیب ہے، جو کوڈ کی پڑھنے کی اہلیت کو بڑھاتا ہے اور پروگرام کی دیکھ بھال کی لاگت کو کم کرتا ہے۔
2. تشریح شدہ اور انٹرایکٹو: Python ایک تشریح شدہ زبان ہے، مطلب یہ ہے کہ کوڈ پر عمل درآمد لائن بہ لائن ہوتا ہے، ڈیبگنگ اور ٹیسٹنگ کو آسان بناتا ہے۔ یہ ایک انٹرایکٹو موڈ کو بھی سپورٹ کرتا ہے، جس سے صارفین کوڈ کے ٹکڑوں کو مرتب کرنے کی ضرورت کے بغیر تیزی سے جانچ سکتے ہیں۔
3. ڈائنامک ٹائپنگ: Python ڈائنامک ٹائپنگ کا استعمال کرتا ہے، جس کا مطلب ہے کہ آپ کو متغیر کی قسمیں واضح طور پر بتانے کی ضرورت نہیں ہے۔ متغیرات عملدرآمد کے دوران ضرورت کے مطابق اقسام کو تبدیل کر سکتے ہیں، لچک فراہم کرتے ہیں لیکن ترقی کے دوران متغیر اقسام پر بھی احتیاط کی ضرورت ہوتی ہے۔
4. اعلیٰ سطحی زبان: Python ایک اعلیٰ سطحی ڈیٹا پروگرامر سے دور بہت سی پیچیدہ تفصیلات کا خلاصہ کرتا ہے، بلٹ میں اعلیٰ سطحی ڈھانچے اور ایک وسیع معیاری لائبریری فراہم کرتا ہے۔ یہ ڈویلپرز کو میموری مینجمنٹ یا کم سطح کی تفصیلات کے بارے میں فکر کرنے کی بجائے مسائل کو حل کرنے پر زیادہ توجہ دینے کی اجازت دیتا ہے۔
5. کراس پلیٹ فارم: ازگر مختلف پلیٹ فارمز جیسے کہ ونڈوز، میک او ایس اور لینکس پر دستیاب ہے، جو اسے انتہائی کوڈ کم سے کم ترمیم کے ساتھ آسانی سے دوسرے پلیٹ فارم پر چل Python پورٹیبل بناتا ہے۔ ایک پلیٹ فارم پر لکھا ہوا سکتا ہے۔
6. وسیع معیاری لائبریری: Python ایک جامع معیاری لائبریری کے ساتھ آتا ہے جو مختلف کاموں جیسے کہ سٹرنگ آپریشنز، فائل I/O، نیٹ ورکنگ اور بہت کچھ کے لیے معاونت فراہم کرتا ہے۔ یہ بہت سے عام پروگرامنگ کاموں کے لیے بیرونی لائبریریوں کی ضرورت کو کم کر دیتا ہے۔
7. آبجیکٹ اورینٹڈ: ازگر آبجیکٹ اورینٹڈ پروگرامنگ پیراڈائمز کو سپورٹ کرتا ہے، جس سے ڈویلپرز کو حقیقی دنیا کے اداروں کو ماڈل بنانے کے لیے آبجیکٹ بنانے اور استعمال کرنے کی اجازت ملتی ہے۔ یہ نقطہ نظر کوڈ کو دوبارہ استعمال کرنے، ماڈیولرٹی، اور اسکیل ایبلٹی کو فروغ دیتا ہے۔
8. بڑی اور فعال کمیونٹی: Python میں ڈویلپرز کی ایک بڑی اور متحرک کمیونٹی ہے، جو اس کی وسیع دستاویزات، متعدد تھرڈ پارٹی لائبریریوں، اور فعال سپورٹ فورمز میں حصہ ڈالتی ہے۔ کمیونٹی کا یہ پہلو تعاون کو فروغ دیتا ہے اور ڈیولپرز کو مسائل کو زیادہ موثر طریقے سے حل کرنے میں مدد کرتا ہے۔
9. اوپن سورس: ازگر اوپن سورس ہے، یعنی اس کا سورس کوڈ آزادانہ طور پر دستیاب ہے اور اسے کوئی بھی تبدیل اور تقسیم کر سکتا ہے۔ یہ کمیونٹی کے اندر جدت اور تعاون کی حوصلہ افزائی کرتا ہے۔ Python
10. ورسٹائل اور اسکیل ایبل: Python ورسٹائل ہے اور اسے مختلف مقاصد کے لیے استعمال کیا جا سکتا ہے جیسے کہ ویب ڈویلپمنٹ، ڈیٹا کا تجزیہ، مصنوعی ذہانت، مشین لرننگ، سائنسی کمپیوٹنگ، اور بہت کچھ۔ اس کی توسیع پذیری اسے چھوٹے اسکرپٹ سے لے کر بڑے پیمانے پر ایپلی کیشنز کے لیے استعمال کرنے کی اجازت دیتی ہے۔
Python یہ خصوصیات کے کاموں کے لیے ایک مقبول انتخاب بناتی ہیں، سادہ سکرپٹ سے لے کر پیچیدہ سافٹ ویئر ڈویلپمنٹ پروجیکٹس تک۔ اس کی سادگی، پڑھنے کی اہلیت، اور وسیع ماحولیاتی نظام اسے ابتدائی اور تجربہ کار ڈویلپرز دونوں کے لیے ایک بہترین زبان بناتا ہے۔

Setting up Python environment (interpreter, IDE)

Introduction to Python:

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python emphasizes code readability and productivity, making it a popular choice for beginners and experienced developers alike. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Setting up Python environment:

To start coding in Python, you'll need to set up your development environment, which typically involves installing Python interpreter and choosing an Integrated Development Environment (IDE) or a text editor for writing and running your code.

1. **Installing Python Interpreter:**
    1. Visit the official Python website at https://www.python.org/.
    2. Download the latest version of Python for your operating system (Windows, macOS, or Linux).
    3. Follow the installation instructions provided by the installer.
2. **Choosing an IDE or Text Editor:**
    1. IDEs (Integrated Development Environments) provide a comprehensive set of tools for coding, debugging, and managing projects. Some popular Python IDEs include:
        1. PyCharm
        2. Visual Studio Code (with Python extension)
        3. Spyder
        4. IDLE (Python's built-in IDE)
    2. Text editors offer simplicity and flexibility. Some widely used text editors for Python development are:
        1. Sublime Text
        2. Atom
        3. VS Code (can be used as a simple text editor without additional extensions)
        4. Notepad++
    3. Choose an IDE or text editor based on your preferences and requirements. Many developers prefer VS Code due to its versatility and extensive community support.

3. **Configuring the Environment:**

    1. Once you have installed Python and chosen your IDE or text editor, ensure they are properly configured.
    2. IDEs often require minimal setup, but you may need to configure Python interpreter paths if multiple versions of Python are installed on your system.
    3. Text editors usually require manual configuration for syntax highlighting, linting, and other features. Install relevant extensions or plugins for Python development.

4. **Testing Your Setup:**

    1. After setting up your environment, it's a good idea to test it by writing a simple Python script and running it.
    2. Open your chosen IDE or text editor, create a new Python file, write some Python code (e.g., `print(“Hello, Python!”)`), and save the file with a `.py` extension.
    3. Run the script either from within the IDE/editor or through the terminal/command prompt by navigating to the directory containing the script and executing `python filename.py`.

Once you have completed these steps, you're ready to start coding in Python! You can explore Python's vast ecosystem of libraries and frameworks to build a wide range of applications, from web development and data analysis to artificial intelligence and machine learning.

(IDE ،ترجمان) ازگر کا ماحول ترتیب دینا
:ازگر کا تعارف
ایک اعلیٰ سطحی، تشریح شدہ پروگرامنگ زبان ہے جو اپنی سادگی اور پڑھنے کی اہلیت کے لیے مشہور ہے۔ اسے Python کوڈ پڑھنے کی اہلیت اور پیداواری Python نے بنایا تھا اور پہلی بار 1991 میں ریلیز کیا گیا تھا۔ Guido van Rossum صلاحیت پر زور دیتا ہے، جس سے یہ ابتدائی اور تجربہ کار ڈویلپرز کے لیے ایک مقبول انتخاب ہے۔ یہ متعدد پروگرامنگ پیراڈیمز کی حمایت کرتا ہے، بشمول پروسیجرل، آبجیکٹ پر مبنی، اور فنکشنل پروگرامنگ۔
:ازگر کا ماحول ترتیب دینا
میں کوڈنگ شروع کرنے کے لیے، آپ کو اپنے ترقیاتی ماحول کو ترتیب دینے کی ضرورت ہوگی، جس میں عام طور Python یا ٹیکسٹ (IDE) انٹرپریٹر کو انسٹال کرنا اور اپنے کوڈ کو لکھنے اور چلانے کے لیے ایک مربوط ترقیاتی ماحول Python پر ایڈیٹر کا انتخاب کرنا شامل ہے۔
:انٹرپریٹر انسٹال کرنا 1. Python
پر دیکھیں۔ https://www.python.org/ کی آفیشل ویب سائٹ 1. Python
کا تازہ ترین ورژن ڈاؤن لوڈ کریں۔ Python کے لیے (Linux یا ،Windows, macOS) اپنے آپریٹنگ سسٹم 2.
انسٹالر کی طرف سے فراہم کردہ تنصیب کی ہدایات پر عمل کریں۔ 3.
:یا ٹیکسٹ ایڈیٹر کا انتخاب IDE ایک 2.
کوڈنگ، ڈیبگنگ، اور پراجیکٹس کے انتظام کے لیے ٹولز کا (Integrated Development Environments) IDEs 1.
:میں شامل ہیں Python IDEs ایک جامع سیٹ فراہم کرتے ہیں۔ کچھ مشہور
پائی چارم 1.
بصری اسٹوڈیو کوڈ (ازگر کی توسیع کے ساتھ) 2.
سپائیڈر 3.
(IDE ازگر کا بلٹ ان) IDLE 4.
ٹیکسٹ ایڈیٹرز سادگی اور لچک پیش کرتے ہیں۔ ازگر کی ترقی کے لیے کچھ بڑے پیمانے پر استعمال ہونے والے 2.
:ٹیکسٹ ایڈیٹرز یہ ہیں
شاندار متن 1.
ایٹم 2.
کوڈ (اضافی ایکسٹینشن کے بغیر ایک سادہ ٹیکسٹ ایڈیٹر کے طور پر استعمال کیا جا سکتا ہے) VS 3.
++نوٹ پیڈ 4.
کوڈ کو اس کی VS یا ٹیکسٹ ایڈیٹر کا انتخاب کریں۔ بہت سے ڈویلپر IDE اپنی ترجیحات اور ضروریات کی بنیاد پر 3.
استعداد اور وسیع کمیونٹی سپورٹ کی وجہ سے ترجیح دیتے ہیں۔
:ماحول کو ترتیب دینا 3.
یا ٹیکسٹ ایڈیٹر منتخب کر لیا، تو یقینی بنائیں کہ وہ IDE انسٹال کر لیا اور اپنا Python ایک بار جب آپ نے 1.
مناسب طریقے سے کنفیگر ہیں۔
کے متعدد ورژن انسٹال Python کو اکثر کم سے کم سیٹ اپ کی ضرورت ہوتی ہے، لیکن اگر آپ کے سسٹم پر IDEs 2.
انٹرپریٹر پاتھ کو کنفیگر کرنے کی ضرورت پڑ سکتی ہے۔ Python ہیں تو آپ کو
ٹیکسٹ ایڈیٹرز کو عام طور پر نحو کو نمایاں کرنے، لنٹنگ اور دیگر خصوصیات کے لیے دستی ترتیب کی ضرورت ہوتی 3.
کی ترقی کے لیے متعلقہ ایکسٹینشنز یا پلگ ان انسٹال کریں۔ Python ہے۔
:اپنے سیٹ اپ کی جانچ کرنا 4.
اسکرپٹ لکھ کر اور اسے چلا کر اس کی جانچ کرنا اچھا Python اپنے ماحول کو ترتیب دینے کے بعد، ایک سادہ 1.
خیال ہے۔
،جیسے) کوڈ لکھیں Python فائل بنائیں، کچھ Python یا ٹیکسٹ ایڈیٹر کھولیں، ایک نئی IDE اپنا منتخب کردہ 2.
ایکسٹینشن کے ساتھ محفوظ کریں۔ `.py` اور فائل کو ،(`print("Hello, Python!")`
کے اندر سے یا ٹرمینل/کمانڈ پرامپٹ کے ذریعے اسکرپٹ پر مشتمل ڈائریکٹری میں جا IDE/editor اسکرپٹ کو یا تو 3.
کو چلا کر چلائیں۔ `python filename.py` کر اور
میں کوڈنگ شروع کرنے کے لیے تیار ہیں! آپ ویب Python ایک بار جب آپ ان مراحل کو مکمل کر لیتے ہیں، تو آپ
ڈویلپمنٹ اور ڈیٹا کے تجزیہ سے لے کر مصنوعی ذہانت اور مشین لرننگ تک ایپلی کیشنز کی ایک وسیع رینج بنانے کے لیے
کے وسیع ماحولیاتی نظام کو تلاش کر سکتے ہیں۔ Python لائبریریوں اور فریم ورک کے

How to write and run your first Python program

How to write and run your first Python program:

Introduction to Python:

Python is a high-level, interpreted programming language known for its simplicity and readability. It is widely used for various purposes such as web development, data analysis, artificial intelligence, scientific computing, and more. Python emphasizes code readability and its syntax allows programmers to express concepts in fewer lines of code compared to other programming languages.

Writing and Running Your First Python Program:

To write and run a Python program, you'll need a text editor and a Python interpreter installed on your computer. Here's how you can do it:

Step 1: Install Python:

If you don't have Python installed on your computer, you can download and install it from the official Python website: [Python.org](https://www.python.org/downloads/).

Step 2: Write Your Python Program:

Open a text editor (such as Notepad, Sublime Text, Visual Studio Code, etc.) and type the following code:

My First Python Program

print(“Hello, World!”)

This simple program will print “Hello, World!” to the console when executed.

Step 3: Save Your Python Program:

Save the file with a `.py` extension, for example, `first_program.py`. Choose a location where you can easily access it.

Step 4: Run Your Python Program:

Open a terminal or command prompt on your computer. Navigate to the directory where you saved your Python file using the `cd` command.

Once you're in the correct directory, type the following command to run your Python program:

python first_program.py

Replace `first_program.py` with the name of your Python file if it's different.

After running the command, you should see the output “Hello, World!” printed to the console.

Congratulations! You've successfully written and executed your first Python program. From here, you can explore Python's vast ecosystem and learn more about its features and capabilities.

پروگرام کیسے لکھیں اور چلائیں۔ Python اپنا پہلا

:پروگرام کیسے لکھیں اور چلائیں Python اپنا پہلا

:ازگر کا تعارف

ایک اعلیٰ سطحی، تشریح شدہ پروگرامنگ زبان ہے جو اپنی سادگی اور پڑھنے کی اہلیت کے لیے مشہور ہے۔ یہ Python وسیع پیمانے پر مختلف مقاصد کے لیے استعمال ہوتا ہے جیسے کہ ویب ڈویلپمنٹ، ڈیٹا کا تجزیہ، مصنوعی ذہانت، سائنسی کوڈ پڑھنے کی اہلیت پر زور دیتا ہے اور اس کا نحو پروگرامرز کو دیگر پروگرامنگ زبانوں Python کمپیوٹنگ، اور بہت کچھ۔ کے مقابلے کوڈ کی کم لائنوں میں تصورات کا اظہار کرنے کی اجازت دیتا ہے۔

:اپنا پہلا ازگر پروگرام لکھنا اور چلانا

انٹرپریٹر انسٹال کرنے Python پروگرام لکھنے اور چلانے کے لیے، آپ کو اپنے کمپیوٹر پر ایک ٹیکسٹ ایڈیٹر اور Python :کی ضرورت ہوگی۔ یہاں یہ ہے کہ آپ اسے کیسے کر سکتے ہیں

:مرحلہ 1: ازگر انسٹال کریں

کی آفیشل ویب سائٹ سے ڈاؤن لوڈ اور انسٹال کر Python انسٹال نہیں ہے، تو آپ اسے Python اگر آپ کے کمپیوٹر پر ۔[Python.org](https://www.python.org/downloads/) :سکتے ہیں

:پروگرام لکھیں Python مرحلہ 2: اپنا

:ٹیکسٹ ایڈیٹر کھولیں (جیسے نوٹ پیڈ، سبلائم ٹیکسٹ، ویژول اسٹوڈیو کوڈ وغیرہ) اور درج ذیل کوڈ ٹائپ کریں

میرا پہلا ازگر پروگرام #

پرنٹ ("ہیلو، ورلڈ!")

یہ سادہ پروگرام پرنٹ کرے گا "ہیلو، ورلڈ!" جب پھانسی دی جاتی ہے تو کنسول پر۔

:پروگرام محفوظ کریں Python مرحلہ 3: اپنا

۔ ایک ایسی جگہ کا انتخاب `first_program.py`، ایکسٹینشن کے ساتھ محفوظ کریں، مثال کے طور پر `.py` فائل کو کریں جہاں آپ آسانی سے اس تک رسائی حاصل کر سکیں۔

:پروگرام چلائیں Python مرحلہ 4: اپنا

کمانڈ کا استعمال کرتے ہوئے اپنی 'cd' اپنے کمپیوٹر پر ٹرمینل یا کمانڈ پرامپٹ کھولیں۔ ڈائرکٹری پر جائیں جہاں آپ نے فائل کو محفوظ کیا تھا۔ Python

:پروگرام چلانے کے لیے درج ذیل کمانڈ کو ٹائپ کریں Python ایک بار جب آپ صحیح ڈائرکٹری میں ہیں، تو اپنا

```
python first_program.py
```

کو تبدیل کریں۔ `first_program.py` فائل کے نام سے Python اگر یہ مختلف ہے تو اپنی

کمانڈ چلانے کے بعد، آپ کو "ہیلو، ورلڈ!" آؤٹ پٹ دیکھنا چاہیے۔ کنسول پر پرنٹ کیا گیا۔

کے Python پروگرام لکھا اور اس پر عمل کیا ہے۔ یہاں سے، آپ Python مبارک ہو! آپ نے کامیابی کے ساتھ اپنا پہلا وسیع ماحولیاتی نظام کو دریافت کر سکتے ہیں اور اس کی خصوصیات اور صلاحیتوں کے بارے میں مزید جان سکتے ہیں۔

An introduction to Python's basic syntax and variables.

An introduction to Python's basic syntax and variables.

Basic Syntax

Python syntax is relatively straightforward and uses indentation to define code blocks. Here's a simple example of a Python script that prints “Hello, World!”:

print(“Hello, World!”)

Variables

Variables are used to store data in Python. You can think of variables as containers that hold values. Unlike some other programming languages, Python is dynamically typed, meaning you don't need to declare the data type of a variable before assigning a value to it.

Variable Naming Rules

- Variable names can contain letters, numbers, and underscores. - Variable names must start with a letter or an underscore. - Variable names are case-sensitive. - Variable names cannot be reserved keywords like `print`, `if`, `else`, etc.

Examples of Variables

String Variables

name = “Alice”

Integer Variables

age = 25

Float Variables

height = 5.9

Boolean Variables

is_student = True

Multiple Assignments

You can also assign multiple variables in a single line:

a, b, c = 1, 2, 3

Comments

Comments are used to explain your code and are ignored by the Python interpreter. You can use comments to make your code more readable.

This is a comment

Conclusion

Python's simple syntax and powerful features make it an excellent choice for beginners and experienced programmers alike. In this introduction, we covered basic syntax, variable declaration, and comments. As you continue learning Python, you'll explore more advanced topics and learn how to use Python to build various applications.

ازگر کے بنیادی نحو اور متغیرات کا تعارف۔
ازگر کے بنیادی نحو اور متغیرات کا تعارف۔
بنیادی نحو
Python نحو نسبتاً سیدھا ہے اور کوڈ بلاکس کی وضاحت کے لیے انڈینٹیشن کا استعمال کرتا ہے۔ یہاں ایک Python
:پرنٹ کرتی ہے "Hello, World!" اسکرپٹ کی ایک سادہ مثال ہے جو
پرنٹ ("ہیلو، ورلڈ!")
متغیرات
میں ڈیٹا ذخیرہ کرنے کے لیے استعمال کیا جاتا ہے۔ آپ متغیرات کو کنٹینرز کے طور پر سوچ سکتے Python متغیرات کو
کو متحرک طور پر ٹائپ کیا جاتا ہے، یعنی آپ Python ،ہیں جو اقدار رکھتے ہیں۔ کچھ دیگر پروگرامنگ زبانوں کے برعکس
کو کسی متغیر کے ڈیٹا کی قسم کا اعلان کرنے کی ضرورت نہیں ہے اس کی کوئی قدر تفویض کرنے سے پہلے۔
متغیر نام کے قواعد
متغیر ناموں میں حروف، اعداد اور انڈر سکور شامل ہو سکتے ہیں۔ - متغیر ناموں کا آغاز ایک حرف یا انڈر سکور سے ہونا -
چاہیے۔ - متغیر نام کیس کے لحاظ سے حساس ہیں۔ - متغیر ناموں کو مطلوبہ الفاظ محفوظ نہیں کیا جا سکتا جیسے `پرنٹ`،
`اگر`، `اور`، وغیرہ۔
متغیرات کی مثالیں۔
سٹرنگ متغیرات
"نام = "ایلس
عددی متغیرات
عمر = 25
فلوٹ متغیرات
اونچائی = 5.9
بولین متغیرات
سچ ہے۔ = is_student
متعدد اسائنمنٹس
:آپ ایک لائن میں متعدد متغیرات بھی تفویض کر سکتے ہیں
a، b، c = 1، 2، 3
تبصرے
ترجمان کے ذریعے نظر انداز کیے جاتے ہیں۔ Python تبصرے آپ کے کوڈ کی وضاحت کے لیے استعمال کیے جاتے ہیں اور
آپ اپنے کوڈ کو مزید پڑھنے کے قابل بنانے کے لیے تبصرے استعمال کر سکتے ہیں۔
یہ ایک تبصرہ ہے۔ #
نتیجہ
کی سادہ ترکیب اور طاقتور خصوصیات اسے ابتدائی اور تجربہ کار پروگرامرز کے لیے ایک بہترین انتخاب بناتی ہیں۔ Python
سیکھنا جاری رکھیں گے، Python اس تعارف میں، ہم نے بنیادی نحو، متغیر اعلامیہ، اور تبصروں کا احاطہ کیا۔ جیسا کہ آپ
کو مختلف ایپلی کیشنز بنانے کے لیے استعمال Python آپ مزید جدید موضوعات کو دریافت کریں گے اور سیکھیں گے کہ
کرنا ہے۔

Numeric data types: int, float, complex

1. **int**: Integers are whole numbers, positive or negative, without any decimal point. For example, `5`, `-3`, `1000`, etc. Integers have unlimited precision in Python 3.

x = 5

y = -10

2. **float**: Floats represent real numbers and are written with a decimal point dividing the integer and fractional parts. For example, `3.14`, `2.71828`, `-0.5`, etc.

pi = 3.14

euler = 2.71828

3. **complex**: Complex numbers are specified as `<real_part>+<imaginary_part>j`, where `j` represents the square root of -1 (also known as the imaginary unit). For example, `3 + 4j`, `-2.5 - 1j`, etc.

z = 3 + 4j

w = -2.5 - 1j

Python provides built-in functions to convert between these numeric types:

- `int()`: Converts a number or string to an integer. - `float()`: Converts a number or string to a floating-point number. - `complex()`: Converts a number or string to a complex number.

x = int(5.7) # x would be 5

y = float(“3.14”) # y would be 3.14

z = complex(2, -3) # z would be 2 - 3j

These data types can be used in arithmetic operations and mathematical functions according to their respective properties.

فلوٹ، پیچیدہ ،int :عددی ڈیٹا کی اقسام
1. int: عدد مکمل اعداد ہیں، مثبت یا منفی، بغیر کسی اعشاریہ کے۔ مثال کے طور پر، `5`، `-3`، `1000` وغیرہ۔ Python میں انٹیجرز لامحدود درستگی رکھتے ہیں۔ 3
x = 5
y = -10
2. فلوٹ: فلوٹس حقیقی اعداد کی نمائندگی کرتے ہیں اور انٹیجر اور فریکشنل حصوں کو تقسیم کرتے ہوئے اعشاریہ کے ساتھ لکھا جاتا ہے۔ مثال کے طور پر، `3.14`, `2.71828`, `-0.5` وغیرہ۔
pi = 3.14
یولر = 2.71828
3. پیچیدہ: پیچیدہ اعداد کو `<real_part>+<imaginary_part>j` کے طور پر متعین کیا جاتا ہے، جہاں `j` -1 کے مربع ہے (جسے خیالی اکائی بھی کہا جاتا ہے)۔ مثال کے طور پر، `3 + 4j`، `-2.5 - 1j` وغیرہ۔
z = 3 + 4j
w = -2.5 - 1j
Python ان عددی اقسام کے درمیان تبدیل کرنے کے لیے بلٹ ان فنکشن فراہم کرتا ہے:
- `int()`: عدد یا سٹرنگ کو عدد میں تبدیل کرتا ہے۔ `float()`: ایک نمبر یا سٹرنگ کو فلوٹنگ پوائنٹ نمبر میں تبدیل کرتا ہے۔ - `complex()`: ایک عدد یا سٹرنگ کو پیچیدہ نمبر میں تبدیل کرتا ہے۔
x = int(5.7) # x 5 ہوگا۔
y = float("3.14") # y 3.14 ہوگا۔
z = (2, -3) # z 2 - 3j پیچیدہ ہوگا۔
ان ڈیٹا کی اقسام کو ان کی متعلقہ خصوصیات کے مطابق ریاضی کی کارروائیوں اور ریاضی کے افعال میں استعمال کیا جا سکتا ہے۔

Strings and string manipulation

Strings and string manipulation are fundamental concepts in Python programming.

Strings in Python

In Python, strings are sequences of characters enclosed within either single quotes (' '), double quotes (“ ”), or triple quotes (' ' or “”“ ”“”). Here's a simple example:

my_string = “Hello, World!”

Accessing Characters in a String

You can access characters in a string using indexing and slicing:

my_string = “Hello, World!”

print(my_string[0]) # Output: H

print(my_string[7]) # Output: W

print(my_string[0:5]) # Output: Hello

String Concatenation

You can concatenate strings using the `+` operator:

str1 = “Hello”

str2 = “World”

result = str1 + “, ” + str2

print(result) # Output: Hello, World

String Methods

Python provides many built-in methods for string manipulation. Some common methods include:

- `len()`: Returns the length of the string. - `lower()`: Converts all characters in the string to lowercase. - `upper()`: Converts all characters in the string to uppercase. - `strip()`: Removes leading and trailing whitespace. - `split()`: Splits the string into a list of substrings based on a delimiter. - `join()`: Joins elements of an iterable (e.g., a list) into a string using the specified delimiter.

Here's how you can use some of these methods:

my_string = “ Hello, World! ”

print(len(my_string)) # Output: 17

print(my_string.lower()) # Output: hello, world!

print(my_string.strip()) # Output: Hello, World!

print(my_string.split(',')) # Output: [' Hello', ' World! ']

String Formatting

Python supports multiple ways to format strings, including the `format()` method and f-strings (formatted string literals):

name = “Alice”

age = 30

print(“My name is {} and I am {} years old.”.format(name, age))

Output: My name is Alice and I am 30 years old.

print(f“My name is {name} and I am {age} years old.”)

Output: My name is Alice and I am 30 years old.

These are some of the basics of strings and string manipulation in Python. They are incredibly versatile and essential for various programming tasks.

سٹرنگز اور سٹرنگ ہیرا پھیری
پروگرامنگ میں بنیادی تصورات ہیں۔ Python سٹرنگز اور سٹرنگ ہیرا پھیری
میں سٹرنگز Python
میں، سٹرنگز حروف کی ترتیب ہیں جو کسی ایک کوٹس ('')، ڈبل کوٹس ("")، یا ٹرپل کوٹس (''' یا """") میں بند Python
:ہیں۔ یہاں ایک سادہ مثال ہے
"!ہیلو، ورلڈ" = my_string
ایک تار میں حروف تک رسائی
:آپ انڈیکسنگ اور سلائسنگ کا استعمال کرتے ہوئے سٹرنگ میں حروف تک رسائی حاصل کر سکتے ہیں
"!ہیلو، ورلڈ" = my_string
H :آؤٹ پٹ # (print(my_string[0]
W :آؤٹ پٹ # (print(my_string[7]
آؤٹ پٹ: ہیلو # (print(my_string[0:5]
سٹرنگ کنکٹنیشن
:آپ `+` آپریٹر کا استعمال کرتے ہوئے تاروں کو جوڑ سکتے ہیں
"ہیلو" = str1
"دنیا" = str2
نتیجہ = str1 + "," + str2
پرنٹ (نتیجہ) # آؤٹ پٹ: ہیلو، ورلڈ
سٹرنگ کے طریقے
:سٹرنگ ہیرا پھیری کے لیے بہت سے بلٹ ان طریقے فراہم کرتا ہے۔ کچھ عام طریقوں میں شامل ہیں Python
- سٹرنگ کے تمام حروف کو چھوٹے حروف میں تبدیل کرتا ہے۔ :`lower()` - تار کی لمبائی لوٹاتا ہے۔ :`len()` -
آگے اور پیچھے والی خالی جگہ کو ہٹاتا :`strip()` - سٹرنگ کے تمام حروف کو بڑے حروف میں تبدیل کرتا ہے۔ :`upper()`
مخصوص :`join()` - ڈلیمیٹر کی بنیاد پر سٹرنگ کو ذیلی اسٹرنگ کی فہرست میں تقسیم کرتا ہے۔ :`split()` - ہے۔
ڈیلیمیٹر کا استعمال کرتے ہوئے ایک سٹرنگ میں دوبارہ قابل (مثلاً فہرست) کے عناصر کو جوڑتا ہے۔
:یہاں یہ ہے کہ آپ ان طریقوں میں سے کچھ کیسے استعمال کرسکتے ہیں
" !ہیلو، دنیا" = my_string
آؤٹ پٹ: 17 # (print(len(my_string))
!آؤٹ پٹ: ہیلو، دنیا # (print(my_string.lower())
!آؤٹ پٹ: ہیلو، ورلڈ # (print(my_string.strip())
آؤٹ پٹ: ['ہیلو'، 'دنیا! '] # (print(my_string.split(',')
سٹرنگ فارمیٹنگ
فارمیٹ) f-strings طریقہ اور `format()` سٹرنگز کو فارمیٹ کرنے کے متعدد طریقوں کی حمایت کرتا ہے، بشمول Python
:(شدہ سٹرنگ لٹریلز
"نام = "ایلس
عمر = 30
پرنٹ ("میرا نام {} ہے اور میں {} سال کا ہوں۔" فارمیٹ (نام، عمر))
آؤٹ پٹ: میرا نام ایلس ہے اور میری عمر 30 سال ہے۔ #
("ہے اور میری عمر {عمر} سال ہے۔ {name} میرا نام"f) پرنٹ
آؤٹ پٹ: میرا نام ایلس ہے اور میری عمر 30 سال ہے۔ #
میں سٹرنگز اور سٹرنگ ہیرا پھیری کی کچھ بنیادی باتیں ہیں۔ وہ ناقابل یقین حد تک ورسٹائل اور پروگرامنگ کے Python یہ
مختلف کاموں کے لیے ضروری ہیں۔

Boolean data type and logical operators

In Python, the Boolean data type represents truth values, which are either True or False. Boolean values are commonly used in conditional statements and expressions to control the flow of a program.

Here's a brief overview of Boolean data type and logical operators in Python:

1. **Boolean Data Type**: In Python, the Boolean data type has two possible values: True and False. These are keywords in Python and must be capitalized.

x = True

y = False

2. **Logical Operators**: Python provides three main logical operators for working with Boolean values: `and`, `or`, and `not`.

- `and`: Returns True if both operands are True. - `or`: Returns True if at least one of the operands is True. - `not`: Returns the opposite Boolean value of the operand.

and operator

print(True and True) # Output: True

print(True and False) # Output: False

or operator

print(True or False) # Output: True

print(False or False) # Output: False

not operator

print(not True) # Output: False

print(not False) # Output: True

3. **Comparison Operators**: Comparison operators (`<`, `>`, `⇐`, `>=`, `==`, `!=`) are often used to compare values and produce Boolean results.

x = 5

y = 10

print(x < y) # Output: True

print(x == y) # Output: False

4. **Boolean Context**: In Python, any object can be tested for truth value. Certain values such as empty sequences (`'', [], {}`), numbers equal to 0, and `None` are considered False in Boolean context. Everything else is considered True.

```
print(bool(0)) # Output: False

print(bool(10)) # Output: True

print(bool([])) # Output: False

print(bool([1,2])) # Output: True
```

Understanding Boolean data type and logical operators is fundamental for writing conditional statements and controlling the flow of your Python programs.

سٹرنگز اور سٹرنگ ہیرا پھیری
پروگرامنگ میں بنیادی تصورات ہیں۔ Python سٹرنگز اور سٹرنگ ہیرا پھیری
میں سٹرنگز Python
میں، سٹرنگز حروف کی ترتیب ہیں جو کسی ایک کوٹس ('')، ڈبل کوٹس ("")، یا ٹرپل کوٹس (''' یا """") میں بند Python
:ہیں۔ یہاں ایک سادہ مثال ہے
"!ہیلو، ورلڈ" = my_string
ایک تار میں حروف تک رسائی
:آپ انڈیکسنگ اور سلائسنگ کا استعمال کرتے ہوئے سٹرنگ میں حروف تک رسائی حاصل کر سکتے ہیں
"!ہیلو، ورلڈ" = my_string
H :آؤٹ پٹ # (print(my_string[0]
W :آؤٹ پٹ # (print(my_string[7]
آؤٹ پٹ: ہیلو # (print(my_string[0:5]
سٹرنگ کنکٹنیشن
:آپ `+` آپریٹر کا استعمال کرتے ہوئے تاروں کو جوڑ سکتے ہیں
"ہیلو" = str1
"دنیا" = str2
نتیجہ = str1 + "," + str2
پرنٹ (نتیجہ) # آؤٹ پٹ: ہیلو، ورلڈ
سٹرنگ کے طریقے
:سٹرنگ ہیرا پھیری کے لیے بہت سے بلٹ ان طریقے فراہم کرتا ہے۔ کچھ عام طریقوں میں شامل ہیں Python
- سٹرنگ کے تمام حروف کو چھوٹے حروف میں تبدیل کرتا ہے۔ :`lower()` - تار کی لمبائی لوٹاتا ہے۔ :`len()` -
آگے اور پیچھے والی خالی جگہ کو ہٹاتا :`strip()` - سٹرنگ کے تمام حروف کو بڑے حروف میں تبدیل کرتا ہے۔ :`upper()`
مخصوص :`join()` - ڈلیمیٹر کی بنیاد پر سٹرنگ کو ذیلی اسٹرنگ کی فہرست میں تقسیم کرتا ہے۔ :`split()` - ہے۔
ڈیلیمیٹر کا استعمال کرتے ہوئے ایک سٹرنگ میں دوبارہ قابل (مثلاً فہرست) کے عناصر کو جوڑتا ہے۔
:یہاں یہ ہے کہ آپ ان طریقوں میں سے کچھ کیسے استعمال کرسکتے ہیں
" !ہیلو، دنیا" = my_string
آؤٹ پٹ: 17 # (print(len(my_string))
!آؤٹ پٹ: ہیلو، دنیا # (print(my_string.lower())
!آؤٹ پٹ: ہیلو، ورلڈ # (print(my_string.strip())
آؤٹ پٹ: ['ہیلو'، 'دنیا! '] # (print(my_string.split(',')
سٹرنگ فارمیٹنگ
فارمیٹ) f-strings طریقہ اور `format()` سٹرنگز کو فارمیٹ کرنے کے متعدد طریقوں کی حمایت کرتا ہے، بشمول Python
:(شدہ سٹرنگ لٹریلز
"نام = "ایلس
عمر = 30
پرنٹ ("میرا نام {} ہے اور میں {} سال کا ہوں۔" فارمیٹ (نام، عمر))
آؤٹ پٹ: میرا نام ایلس ہے اور میری عمر 30 سال ہے۔ #
("ہے اور میری عمر {عمر} سال ہے۔ {name} میرا نام"f) پرنٹ
آؤٹ پٹ: میرا نام ایلس ہے اور میری عمر 30 سال ہے۔ #
میں سٹرنگز اور سٹرنگ ہیرا پھیری کی کچھ بنیادی باتیں ہیں۔ وہ ناقابل یقین حد تک ورسٹائل اور پروگرامنگ کے Python یہ
مختلف کاموں کے لیے ضروری ہیں۔

Basic arithmetic, comparison, and assignment operators

In Python, there are several basic arithmetic, comparison, and assignment operators that you can use.

Here's a brief overview of each:

Arithmetic Operators:

1. Addition: `+`
2. Subtraction: `-`
3. Multiplication: `*`
4. Division: `/`
5. Floor Division (integer division): `//`
6. Modulus (remainder): `%`
7. Exponentiation: `**`

Comparison Operators:

1. Equal to: `==`
2. Not equal to: `!=`
3. Greater than: `>`
4. Less than: `<`
5. Greater than or equal to: `>=`
6. Less than or equal to: `⇐`

Assignment Operators:

1. Assignment: `=`
2. Addition assignment: `+=`
3. Subtraction assignment: `-=`
4. Multiplication assignment: `*=`
5. Division assignment: `/=`
6. Floor division assignment: `//=`
7. Modulus assignment: `%=`
8. Exponentiation assignment: `**=`

Examples:

```
# Arithmetic Operators
a = 10
b = 3
print(a + b) # Addition
print(a - b) # Subtraction
print(a * b) # Multiplication
print(a / b) # Division
print(a // b) # Floor Division
print(a % b) # Modulus
print(a ** b) # Exponentiation
# Comparison Operators
x = 5
y = 10
print(x == y) # Equal to
print(x != y) # Not equal to
print(x > y) # Greater than
print(x < y) # Less than
print(x >= y) # Greater than or equal to
print(x ⇐ y) # Less than or equal to
# Assignment Operators
c = 5
c += 3 # Equivalent to c = c + 3
print©
d = 10
d -= 2 # Equivalent to d = d - 2
print(d)
e = 3
e *= 4 # Equivalent to e = e * 4
print(e)
```

```
f = 20
f /= 5 # Equivalent to f = f / 5
print(f)
g = 11
g *= 3 # Equivalent to g = g * 3
print(g)
h = 13
h %= 5 # Equivalent to h = h % 5
print(h)
i = 2
i **= 3 # Equivalent to i = i ** 3
print(i)
```

These operators are fundamental in Python programming and are used extensively in various programming tasks.

بولین ڈیٹا کی قسم اور منطقی آپریٹرز
میں، بولین ڈیٹا کی قسم سچائی کی قدروں کی نمائندگی کرتی ہے، جو کہ یا تو سچ ہیں یا غلط۔ بولین اقدار عام Python طور پر پروگرام کے بہاؤ کو کنٹرول کرنے کے لیے مشروط بیانات اور اظہار میں استعمال ہوتی ہیں۔
:میں بولین ڈیٹا کی قسم اور منطقی آپریٹرز کا ایک مختصر جائزہ یہ ہے Python
1. Boolean Data Type: Python میں، Boolean data type میں ہیں کی دو ممکنہ قدریں: True اور False۔ یہ Python میں کلیدی الفاظ ہیں اور ان کا بڑا ہونا ضروری ہے۔
x = سچ
y = غلط
2. منطقی آپریٹرز: Python بولین اقدار کے ساتھ کام کرنے کے لیے تین اہم منطقی آپریٹرز فراہم کرتا ہے: `اور`، `یا`، اور `نہیں`۔
- اور`: درست لوٹاتا ہے اگر دونوں کام صحیح ہیں۔ - `یا`: درست لوٹاتا ہے اگر کاموں میں سے کم از کم ایک صحیح ہے۔` - اوپرینڈ کی مخالف بولین قدر لوٹاتا ہے۔ :`not`
اور آپریٹر #
پرنٹ (سچ اور سچ) # آؤٹ پٹ: سچ
پرنٹ (سچ اور غلط) # آؤٹ پٹ: غلط
یا آپریٹر #
پرنٹ (سچ یا غلط) # آؤٹ پٹ: سچ
پرنٹ (غلط یا غلط) # آؤٹ پٹ: غلط
آپریٹر نہیں۔ #
پرنٹ (سچ نہیں) # آؤٹ پٹ: غلط
پرنٹ (غلط نہیں) # آؤٹ پٹ: سچ ہے۔
3. موازنہ آپریٹرز: موازنہ آپریٹرز (`<`, `>`, `⇐`, `=<`, `==`, `=!`) اکثر اقدار کا موازنہ کرنے اور بولین نتائج پیدا کرنے کے لیے استعمال ہوتے ہیں۔
x = 5
y = 10
پرنٹ (x < y) # آؤٹ پٹ: سچ ہے۔
پرنٹ (x == y) # آؤٹ پٹ: غلط
4. بولین سیاق و سباق: ازگر میں، کسی بھی چیز کو سچائی کی قدر کے لیے جانچا جا سکتا ہے۔ کچھ قدریں جیسے خالی ترتیب (`""`، []، {}`)، نمبر 0 کے برابر، اور "کوئی نہیں" کو بولین سیاق و سباق میں غلط سمجھا جاتا ہے۔ باقی سب کچھ سچ سمجھا جاتا ہے۔
پرنٹ(بول(0)) # آؤٹ پٹ: غلط
پرنٹ(بول(10)) # آؤٹ پٹ: سچ ہے۔
پرنٹ(بول([])) # آؤٹ پٹ: غلط
print(bool([1,2])) # آؤٹ پٹ: سچ ہے۔
پروگراموں کے بہاؤ کو کنٹرول Python بولین ڈیٹا کی قسم اور منطقی آپریٹرز کو سمجھنا مشروط بیانات لکھنے اور آپ کے کرنے کے لیے بنیادی ہے۔

Type conversion and type casting

In Python, type conversion refers to the process of changing an object from one data type to another. This can be done implicitly by Python itself, or explicitly through casting. Here's an overview of both concepts:

Type Conversion:

Type conversion happens automatically in Python in certain situations. For example, when you perform operations between different types, Python will automatically convert them as needed. Here's an example:

num_int = 123 # Integer

num_float = 1.23 # Float

result = num_int + num_float # Result will be float

print(result) # Output: 124.23

In the above example, `num_int` is an integer and `num_float` is a float. When we add them together, Python converts `num_int` to a float before performing the addition.

Type Casting:

Type casting is the explicit conversion of a data type to another data type. Python provides several built-in functions for type casting, such as `int()`, `float()`, `str()`, etc. Here are some examples:

Convert float to int

float_num = 3.14

int_num = int(float_num)

print(int_num) # Output: 3

Convert int to string

num = 123

str_num = str(num)

print(str_num) # Output: “123”

Convert string to int

str_num = “456”

int_num = int(str_num)

print(int_num) # Output: 456

Explicit Conversion vs Implicit Conversion:

- **Explicit Conversion (Type Casting):** This is done using the built-in functions like `int()`, `float()`, etc. It gives you more control over the conversion process and allows you to handle conversion errors more gracefully.

- **Implicit Conversion:** This happens automatically in Python. While it's convenient, you may lose control over the conversion process, and it's not always obvious when and how the conversion occurs.

In summary, type conversion and type casting are essential concepts in Python that allow you to work with different data types effectively and manipulate data as needed.

بولین ڈیٹا کی قسم اور منطقی آپریٹرز
میں، بولین ڈیٹا کی قسم سچائی کی قدروں کی نمائندگی کرتی ہے، جو کہ یا تو سچ ہیں یا غلط۔ بولین اقدار عام Python طور پر پروگرام کے بہاؤ کو کنٹرول کرنے کے لیے مشروط بیانات اور اظہار میں استعمال ہوتی ہیں۔
:میں بولین ڈیٹا کی قسم اور منطقی آپریٹرز کا ایک مختصر جائزہ یہ ہے Python
1. Boolean Data Type: Python میں، Boolean data type میں ہیں کی دو ممکنہ قدریں: True اور False۔ یہ Python میں کلیدی الفاظ ہیں اور ان کا بڑا ہونا ضروری ہے۔
x = سچ
y = غلط
2. منطقی آپریٹرز: Python بولین اقدار کے ساتھ کام کرنے کے لیے تین اہم منطقی آپریٹرز فراہم کرتا ہے: `اور`، `یا`، اور `نہیں`۔
- اور`: درست لوٹاتا ہے اگر دونوں کام صحیح ہیں۔ - `یا`: درست لوٹاتا ہے اگر کاموں میں سے کم از کم ایک صحیح ہے۔` - اوپرینڈ کی مخالف بولین قدر لوٹاتا ہے۔ :`not`
اور آپریٹر #
پرنٹ (سچ اور سچ) # آؤٹ پٹ: سچ
پرنٹ (سچ اور غلط) # آؤٹ پٹ: غلط
یا آپریٹر #
پرنٹ (سچ یا غلط) # آؤٹ پٹ: سچ
پرنٹ (غلط یا غلط) # آؤٹ پٹ: غلط
آپریٹر نہیں۔ #
پرنٹ (سچ نہیں) # آؤٹ پٹ: غلط
پرنٹ (غلط نہیں) # آؤٹ پٹ: سچ ہے۔
3. موازنہ آپریٹرز: موازنہ آپریٹرز (`<`, `>`, `⇐`, `=<`, `==`, `=!`) اکثر اقدار کا موازنہ کرنے اور بولین نتائج پیدا کرنے کے لیے استعمال ہوتے ہیں۔
x = 5
y = 10
پرنٹ (x < y) # آؤٹ پٹ: سچ ہے۔
پرنٹ (x == y) # آؤٹ پٹ: غلط
4. بولین سیاق و سباق: ازگر میں، کسی بھی چیز کو سچائی کی قدر کے لیے جانچا جا سکتا ہے۔ کچھ قدریں جیسے خالی ترتیب (`""`، []، {}`)، نمبر 0 کے برابر، اور "کوئی نہیں" کو بولین سیاق و سباق میں غلط سمجھا جاتا ہے۔ باقی سب کچھ سچ سمجھا جاتا ہے۔
پرنٹ(بول(0)) # آؤٹ پٹ: غلط
پرنٹ(بول(10)) # آؤٹ پٹ: سچ ہے۔
پرنٹ(بول([])) # آؤٹ پٹ: غلط
print(bool([1,2])) # آؤٹ پٹ: سچ ہے۔
پروگراموں کے بہاؤ کو کنٹرول Python بولین ڈیٹا کی قسم اور منطقی آپریٹرز کو سمجھنا مشروط بیانات لکھنے اور آپ کے کرنے کے لیے بنیادی ہے۔

Conditional statements: if, elif, else

In Python, conditional statements are used to execute different blocks of code based on whether a certain condition evaluates to `True` or `False`. The basic structure includes `if`, `elif` (short for “else if”), and `else`.

Here's a basic syntax example:

x = 10

if x > 10:

```
  print("x is greater than 10")
```

elif x == 10:

```
  print("x is equal to 10")
```

else:

```
  print("x is less than 10")
```

In this example:

- The `if` statement checks if `x` is greater than 10. If it's true, it executes the corresponding block of code. - The `elif` statement checks if `x` is equal to 10. If the previous condition (`x > 10`) was false and this condition is true, it executes the corresponding block of code. - The `else` statement catches anything that didn't satisfy the previous conditions and executes its corresponding block of code. It doesn't have a condition because it's the “catch-all” block.

You can have multiple `elif` statements, and `else` is optional.

Here's another example with multiple conditions:

x = 5

if x > 10:

```
  print("x is greater than 10")
```

elif x == 10:

```
  print("x is equal to 10")
```

elif x > 5:

```
  print("x is greater than 5")
```

else:

```
  print("x is 5 or less")
```

In this case, since `x` is less than 10, it moves to the next condition and checks if `x` is equal to 10. Since it's not, it moves to the next condition and checks if `x` is greater than 5, which is false. Therefore, the `else` block is executed.

مشروط بیانات: if، elif، else
میں، مشروط بیانات کو کوڈ کے مختلف بلاکس پر عمل درآمد کرنے کے لیے استعمال کیا جاتا ہے اس بنیاد پر کہ آیا Python اور ،(اور اگر" کے لئے مختصر") `if`، `elif` میں جانچتی ہے۔ بنیادی ڈھانچے میں `False` یا `True` کوئی خاص شرط شامل ہیں۔ `else`
:یہاں ایک بنیادی نحوی مثال ہے
```
x = 10
:x > 10 اگر
("سے بڑا ہے 10 x") پرنٹ
:ایلیف ایکس == 10
("برابر ہے 10 x") پرنٹ
:دوسری
("سے کم ہے 10 x") پرنٹ
```
:اس مثال میں
سے زیادہ ہے۔ اگر یہ سچ ہے، تو یہ کوڈ کے متعلقہ بلاک پر عمل کرتا ہے۔ - 10 `x` سٹیٹمنٹ چیک کرتا ہے کہ آیا `if` - غلط تھی اور یہ شرط درست ہے، (`x > 10`) کے برابر ہے۔ اگر پچھلی شرط 10 `x` `ایلیف` سٹیٹمنٹ چیک کرتا ہے کہ آیا تو یہ کوڈ کے متعلقہ بلاک پر عمل کرتا ہے۔ - 'اور' بیان کسی بھی چیز کو پکڑتا ہے جو پچھلی شرائط کو پورا نہیں کرتا ہے اور کوڈ کے اس سے متعلقہ بلاک پر عمل درآمد کرتا ہے۔ اس کی کوئی شرط نہیں ہے کیونکہ یہ "کیچ آل" بلاک ہے۔
آپ کے پاس متعدد `ایلیف` بیانات ہو سکتے ہیں، اور `اور` اختیاری ہے۔
:متعدد شرائط کے ساتھ ایک اور مثال یہ ہے
```
x = 5
:x > 10 اگر
("سے بڑا ہے 10 x") پرنٹ
:ایلیف ایکس == 10
("برابر ہے 10 x") پرنٹ
elif x > 5:
("سے بڑا ہے 5 x") پرنٹ
:دوسری
("یا اس سے کم ہے 5 x") پرنٹ
```
کے برابر ہے۔ 10 `x` سے کم ہے، یہ اگلی حالت میں چلا جاتا ہے اور چیک کرتا ہے کہ آیا 10 `x` اس صورت میں، چونکہ سے بڑا ہے۔ ، جو غلط ہے. لہذا، 'اور' بلاک کو 5 `x` چونکہ یہ نہیں ہے، یہ اگلی شرط پر جاتا ہے اور چیک کرتا ہے کہ آیا پھانسی دی جاتی ہے۔

Looping constructs: while loop, for loop

In Python, looping constructs like `while` loops and `for` loops are used to execute a block of code repeatedly. Here's an explanation of each:

While Loop:

A `while` loop repeatedly executes a block of code as long as a specified condition is true. It continues to execute the block until the condition becomes false.

Syntax:

```
python

while condition:    # Code block to be executed repeatedly
```

Example:

```
python

count = 0while count < 5:    print(count)    count += 1
```

Output:

```
01234
```

For Loop:

A `for` loop iterates over a sequence (such as a list, tuple, or string) and executes a block of code for each element in the sequence.

Syntax:

```
python

for element in sequence:    # Code block to be executed for each element
```

Example:

```
python

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

Output:

```
applebananacherry
```

Range Function:

The `range()` function is commonly used with `for` loops to generate a sequence of numbers. It generates a sequence of numbers from a starting value up to (but not including) an ending value.

Syntax:

```
python

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

Example:

```
python

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

Output:

```
01234
```

Break and Continue Statements:

- `break`: Terminates the loop prematurely, skipping the remaining iterations.
- `continue`: Skips the rest of the current iteration and proceeds to the next iteration.

Example:

```
python

for i in range(10):    if i == 3:        continue    print(i)    if i == 7:
break
```

Output:

```
0124567
```

Else Clause in Loops:

Both `for` and `while` loops in Python can have an `else` clause, which is executed when the loop terminates normally (i.e., not by a `break` statement).

Example:

```
python

for i in range(5):    print(i)else:    print("Loop finished normally")
```

Output:

```
vbnet
```

```
01234Loop finished normally
```

Conclusion:

Loops are essential constructs in Python for executing repetitive tasks. Whether you're using a `while` loop to iterate based on a condition or a `for` loop to iterate over a sequence, mastering these looping constructs is crucial for writing efficient and concise code in Python.

لوپنگ کنسٹرکٹس: جبکہ لوپ، لوپ کے لیے
کو کوڈ کے بلاک کو بار بار چلانے کے لیے استعمال کیا for loops اور while loops میں، لوپنگ کنسٹرکٹس جیسے Python
:جاتا ہے۔ یہاں ہر ایک کی وضاحت ہے
:جبکہ لوپ
بار بار کوڈ کے بلاک پر عمل درآمد کرتا ہے جب تک کہ ایک مخصوص حالت درست ہو۔ یہ اس وقت تک A while loop
بلاک پر عمل درآمد جاری رکھتا ہے جب تک کہ شرط غلط نہ ہو جائے۔
:نحو
ازگر
جبکہ شرط: # کوڈ بلاک کو بار بار عمل میں لایا جائے۔
:مثال
ازگر
شمار = 0 جبکہ شمار < 5: پرنٹ (شمار) شمار += 1
:آؤٹ پٹ
01234
:لوپ کے لیے
پر اعادہ کرتا ہے اور ترتیب میں ہر عنصر کے لیے کوڈ کے (string یا ،tuple ،جیسے کہ فہرست) ایک ترتیب A for loop
ایک بلاک پر عمل کرتا ہے۔
:نحو
ازگر
ترتیب میں عنصر کے لیے: # کوڈ بلاک ہر عنصر کے لیے عمل میں لایا جائے گا۔
:مثال
ازگر
پھل = ["سیب"، "کیلا"، "چیری"] پھلوں میں پھل کے لیے: پرنٹ (پھل)
:آؤٹ پٹ
applebananacherry
:رینج فنکشن
رینج () فنکشن عام طور پر نمبروں کی ترتیب پیدا کرنے کے لیے لوپس کے ساتھ استعمال ہوتا ہے۔ یہ ایک ابتدائی قدر سے
لے کر اختتامی قدر تک (لیکن اس میں شامل نہیں) نمبروں کی ترتیب تیار کرتا ہے۔
:نحو
ازگر
حد (شروع، روک[، قدم])
:مثال
ازگر
(i) کے لیے (5): پرنٹ i رینج میں
:آؤٹ پٹ
01234
:بیانات توڑیں اور جاری رکھیں
وقفہ: لوپ کو وقت سے پہلے ختم کرتا ہے، بقیہ تکرار کو چھوڑ دیتا ہے۔ •
جاری رکھیں: باقی موجودہ تکرار کو چھوڑ دیتا ہے اور اگلی تکرار پر جاتا ہے۔ •
:مثال
ازگر
وقفہ :7 == i اگر (i) پرنٹ جاری رکھیں :3 == i کے لیے (10): اگر i رینج میں
:آؤٹ پٹ
0124567
:لوپس میں دوسری شق
میں لوپ کے لیے اور دوران دونوں میں ایک اور شق ہو سکتی ہے، جو اس وقت عمل میں آتی ہے جب لوپ عام Python
طور پر ختم ہو جائے (یعنی بریک سٹیٹمنٹ کے ذریعے نہیں)۔
:مثال
ازگر
اور: پرنٹ ("لوپ عام طور پر ختم ہوا") (i) رینج (5) کے لیے: پرنٹ

:آؤٹ پٹ
vbnet
لوپ عام طور پر ختم ہوا۔01234
:نتیجہ
بار بار کاموں کو انجام دینے کے لیے لوپس ازگر میں ضروری تعمیرات ہیں۔ چاہے آپ کسی حالت کی بنیاد پر اعادہ کرنے کے لیے تھوڑی دیر کا لوپ استعمال کر رہے ہوں یا کسی ترتیب پر اعادہ کرنے کے لیے لوپ کا استعمال کر رہے ہوں، پائتھون میں موثر اور جامع کوڈ لکھنے کے لیے ان لوپنگ کنسٹرکٹس میں مہارت حاصل کرنا بہت ضروری ہے۔

Understanding lists and tuples

Lists and tuples are two fundamental data structures in Python that allow you to store collections of items. While they share some similarities, they also have distinct differences in terms of mutability and use cases.

Lists:

- Lists are ordered collections of items, which means the items have a specific order and can be accessed by their index.
- Lists are mutable, meaning you can change, add, or remove elements after the list is created.
- Lists are defined using square brackets `[]`.
- Lists can contain elements of different data types, including numbers, strings, other lists, tuples, etc.
- Lists are commonly used when you need a collection of items that may change over time or when you need to perform operations like appending, inserting, or removing elements.

Example of a list:

```
python

my_list = [1, 2, 3, 'hello', True]
```

Tuples:

- Tuples are ordered collections of items, similar to lists, but they are immutable, meaning once created, you cannot change, add, or remove elements.
- Tuples are defined using parentheses `()`.
- Tuples can contain elements of different data types, similar to lists.
- Tuples are commonly used when you want to represent a collection of items that should not change, such as coordinates, database records, or function arguments.

Example of a tuple:

```
python

my_tuple = (1, 2, 3, 'hello', True)
```

Key Differences:

1. **Mutability**: Lists are mutable, while tuples are immutable. This means you can modify a list after it's created, but you cannot modify a tuple.

2. **Syntax**: Lists are defined using square brackets `[]`, while tuples are defined using parentheses `()`.
3. **Performance**: Tuples are generally faster and more memory-efficient than lists because of their immutability. However, this difference may be negligible for small collections.
4. **Use Cases**: Lists are typically used when you need a collection of items that may change over time, while tuples are used for fixed collections of items that should not change.

When to Use Each:

- Use lists when you need a mutable collection of items, such as when you're building dynamic lists or need to modify the contents frequently.
- Use tuples when you need an immutable collection of items, such as when you're representing fixed data structures or passing arguments to functions that should not be modified.

Understanding the differences between lists and tuples and knowing when to use each is essential for writing efficient and maintainable Python code.

لوپنگ کنسٹرکٹس: جبکہ لوپ، لوپ کے لیے
کو کوڈ کے بلاک کو بار بار چلانے کے لیے استعمال کیا for loops اور while loops میں، لوپنگ کنسٹرکٹس جیسے Python
:جاتا ہے۔ یہاں ہر ایک کی وضاحت ہے
:جبکہ لوپ
بار بار کوڈ کے بلاک پر عمل درآمد کرتا ہے جب تک کہ ایک مخصوص حالت درست ہو۔ یہ اس وقت تک A while loop
بلاک پر عمل درآمد جاری رکھتا ہے جب تک کہ شرط غلط نہ ہو جائے۔
:نحو
ازگر
جبکہ شرط: # کوڈ بلاک کو بار بار عمل میں لایا جائے۔
:مثال
ازگر
شمار = 0 جبکہ شمار < 5: پرنٹ (شمار) شمار += 1
:آؤٹ پٹ
01234
:لوپ کے لیے
پر اعادہ کرتا ہے اور ترتیب میں ہر عنصر کے لیے کوڈ کے (string یا ،tuple ،جیسے کہ فہرست) ایک ترتیب A for loop
ایک بلاک پر عمل کرتا ہے۔
:نحو
ازگر
ترتیب میں عنصر کے لیے: # کوڈ بلاک ہر عنصر کے لیے عمل میں لایا جائے گا۔
:مثال
ازگر
پھل = ["سیب"، "کیلا"، "چیری"] پھلوں میں پھل کے لیے: پرنٹ (پھل)
:آؤٹ پٹ
applebananacherry
:رینج فنکشن
رینج () فنکشن عام طور پر نمبروں کی ترتیب پیدا کرنے کے لیے لوپس کے ساتھ استعمال ہوتا ہے۔ یہ ایک ابتدائی قدر سے
لے کر اختتامی قدر تک (لیکن اس میں شامل نہیں) نمبروں کی ترتیب تیار کرتا ہے۔
:نحو
ازگر
حد (شروع، روک[، قدم])
:مثال
ازگر
(i) کے لیے (5): پرنٹ i رینج میں
:آؤٹ پٹ
01234
:بیانات توڑیں اور جاری رکھیں
وقفہ: لوپ کو وقت سے پہلے ختم کرتا ہے، بقیہ تکرار کو چھوڑ دیتا ہے۔ •
جاری رکھیں: باقی موجودہ تکرار کو چھوڑ دیتا ہے اور اگلی تکرار پر جاتا ہے۔ •
:مثال
ازگر
وقفہ :7 == i اگر (i) پرنٹ جاری رکھیں :3 == i کے لیے (10): اگر i رینج میں
:آؤٹ پٹ
0124567
:لوپس میں دوسری شق
میں لوپ کے لیے اور دوران دونوں میں ایک اور شق ہو سکتی ہے، جو اس وقت عمل میں آتی ہے جب لوپ عام Python
طور پر ختم ہو جائے (یعنی بریک سٹیٹمنٹ کے ذریعے نہیں)۔
:مثال
ازگر
اور: پرنٹ ("لوپ عام طور پر ختم ہوا") (i) رینج (5) کے لیے: پرنٹ

:آؤٹ پٹ
vbnet
لوپ عام طور پر ختم ہوا۔01234
:نتیجہ
بار بار کاموں کو انجام دینے کے لیے لوپس ازگر میں ضروری تعمیرات ہیں۔ چاہے آپ کسی حالت کی بنیاد پر اعادہ کرنے کے لیے تھوڑی دیر کا لوپ استعمال کر رہے ہوں یا کسی ترتیب پر اعادہ کرنے کے لیے لوپ کا استعمال کر رہے ہوں، پائتھون میں موثر اور جامع کوڈ لکھنے کے لیے ان لوپنگ کنسٹرکٹس میں مہارت حاصل کرنا بہت ضروری ہے۔

Lists: creation, indexing, slicing, appending, and extending

The basics of lists in Python, covering their creation, indexing, slicing, appending, and extending:

1. Creation:

You can create a list by enclosing comma-separated values within square brackets `[]`.

```
python

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

2. Indexing:

Lists in Python are zero-indexed, meaning the first element has an index of 0, the second element has an index of 1, and so on. You can access elements of a list using square brackets `[]` and the index of the element you want to access.

```
python

print(my_list[0])  # Output: 1print(my_list[2])  # Output: 3
```

3. Slicing:

You can also extract a sublist (slice) from a list using the slicing notation `start:stop:step`. This notation allows you to specify a starting index, an ending index (exclusive), and an optional step size.

```
python

print(my_list[1:4])   # Output: [2, 3, 4]print(my_list[:3])    # Output: [1, 2,
3]print(my_list[::2])   # Output: [1, 3, 5]
```

4. Appending:

You can add elements to the end of a list using the `append()` method.

```
python

my_list.append(6)print(my_list)  # Output: [1, 2, 3, 4, 5, 6]
```

5. Extending:

You can also append elements from another iterable (e.g., another list) to the end of a list using the `extend()` method.

```
python

another_list = [7, 8, 9]my_list.extend(another_list)print(my_list)  # Output: [1,
2, 3, 4, 5, 6, 7, 8, 9]
```

Conclusion:

Lists are versatile data structures in Python that allow you to store collections of items. Understanding how to create, index, slice, append, and extend lists is fundamental for working with them effectively in Python. These basic operations provide you with the necessary tools to manipulate lists and perform various tasks efficiently.

فہرستیں: تخلیق، اشاریہ سازی، سلائسنگ، منسلک، اور توسیع
میں فہرستوں کی بنیادی باتیں، ان کی تخلیق، اشاریہ سازی، سلائسنگ، شامل کرنا اور توسیع کرنا Python:
1. تخلیق:
آپ مربع بریکٹ [] میں کوما سے الگ کردہ اقدار کو بند کر کے ایک فہرست بنا سکتے ہیں۔
ازگر
میری_لسٹ = [1، 2، 3، 4، 5]
2. اشاریہ بندی:
میں فہرستیں زیرو انڈیکس ہیں، یعنی پہلے عنصر کا انڈیکس 0 ہے، دوسرے عنصر کا انڈیکس 1 ہے، وغیرہ۔ آپ Python مربع بریکٹ [] اور جس عنصر تک آپ رسائی حاصل کرنا چاہتے ہیں اس کے اشاریہ کا استعمال کرتے ہوئے فہرست کے عناصر تک رسائی حاصل کر سکتے ہیں۔
ازگر
آؤٹ پٹ: 3 # ([2]my_list)printآؤٹ پٹ: 1 # ([0]my_list)print
3. سلائسنگ:
آپ سلائسنگ نوٹیشن start:stop:step کا استعمال کرتے ہوئے فہرست سے سب لسٹ (سلائس) بھی نکال سکتے ہیں۔ یہ اشارے آپ کو ایک ابتدائی اشاریہ، ایک اختتامی اشاریہ (خصوصی)، اور ایک اختیاری قدم کا سائز بتانے کی اجازت دیتا ہے۔
ازگر
آؤٹ پٹ: # ([::2]my_list)printآؤٹ پٹ: [1, 2, 3] # ([:3]my_list)printآؤٹ پٹ: [2, 3, 4] # ([1:4]my_list)print
[1، 3، 5]
4. شامل کرنا:
آپ ضمیمہ () طریقہ استعمال کرکے فہرست کے آخر میں عناصر شامل کرسکتے ہیں۔
ازگر
آؤٹ پٹ: [1, 2, 3, 4, 5, 6] # (6)my_list.appendprint(my_list)
5. توسیع کرنا:
آپ توسیع () طریقہ استعمال کرتے ہوئے فہرست کے آخر میں کسی اور قابل تکرار (مثلاً ایک اور فہرست) سے عناصر کو بھی شامل کر سکتے ہیں۔
ازگر
آؤٹ پٹ: [1, 2, 3, 4, 5, 6, 7, 8, 9] # [9 ,8 ,7 = دوسری_لسٹ my_list.extend(ather_list)print(my_list)
نتیجہ:
میں ورسٹائل ڈیٹا ڈھانچے ہیں جو آپ کو اشیاء کے مجموعے کو ذخیرہ کرنے کی اجازت دیتی ہیں۔ Python فہرستیں میں ان کے ساتھ مؤثر طریقے سے کام کرنے کے لیے فہرستیں بنانے، انڈیکس کرنے، سلائس کرنے، شامل کرنے اور Python بڑھانے کے طریقے کو سمجھنا بنیادی ہے۔ یہ بنیادی کارروائیاں آپ کو فہرستوں میں ہیرا پھیری کرنے اور مختلف کاموں کو مؤثر طریقے سے انجام دینے کے لیے ضروری ٹولز فراہم کرتی ہیں۔

Accessing elements in a list

Accessing elements in a list involves retrieving specific items from the list using their indices or using various methods provided by Python. Let's explore different ways to access elements in a list:

1. Accessing by Index:

You can access elements in a list using their index, which is the position of the element in the list. Remember that indexing in Python starts from 0.

```
python

my_list = [10, 20, 30, 40, 50]print(my_list[0])  # Output: 10print(my_list[2])  #
Output: 30print(my_list[-1]) # Output: 50 (negative index starts from the end)
```

2. Slicing:

Slicing allows you to retrieve a subset of elements from the list by specifying a range of indices.

```
python

print(my_list[1:3])  # Output: [20, 30] (elements from index 1 to
2)print(my_list[:3])   # Output: [10, 20, 30] (elements from beginning to index
2)print(my_list[2:])   # Output: [30, 40, 50] (elements from index 2 to
end)print(my_list[:-1])  # Output: [10, 20, 30, 40] (elements from beginning to
second last)
```

3. Looping Through Elements:

You can iterate over each element in the list using a loop, such as a `for` loop.

```
python

for item in my_list:    print(item)
```

4. List Comprehensions:

List comprehensions provide a concise way to create lists. You can also use them to filter elements based on certain conditions.

```
python

# Create a new list containing squared elements of the original listsquared_list =
[x ** 2 for x in my_list]print(squared_list)
```

5. Using Index Method:

You can use the `index()` method to find the index of a specific element in the list.

```
python
```

```
index = my_list.index(30)print(index)  # Output: 2
```

Conclusion:

These are some common methods for accessing elements in a list in Python. Whether you need to retrieve specific elements by index, extract a subset of elements using slicing, iterate over the list, or find the index of a particular element, Python provides versatile tools to work with lists effectively.

فہرست میں عناصر تک رسائی
فہرست میں عناصر تک رسائی میں فہرست سے مخصوص آئٹمز کو ان کے اشاریوں کا استعمال کرتے ہوئے یا ازگر کے فراہم کردہ مختلف طریقوں کو استعمال کرنا شامل ہے۔ آئیے فہرست میں عناصر تک رسائی کے مختلف طریقے تلاش کریں:
1. انڈیکس کے ذریعے رسائی:
آپ فہرست میں عناصر تک رسائی حاصل کر سکتے ہیں ان کے انڈیکس کا استعمال کرتے ہوئے، جو فہرست میں عنصر کی پوزیشن ہے۔ Python میں انڈیکسنگ 0 سے شروع ہوتی ہے۔ یاد رکھیں کہ
ازگر
my_list = [10, 20, 30, 40, 50]print(my_list[0]) # آؤٹ پٹ: 10 print(my_list[2]) # آؤٹ پٹ: 50 (منفی انڈیکس سے شروع ہوتا ہے ختم شد) print(my_list[-1]) # آؤٹ پٹ: 30
2. سلائسنگ:
سلائسنگ آپ کو فہرست سے عناصر کے ذیلی سیٹ کو دوبارہ حاصل کرنے کی اجازت دیتی ہے جس سے انڈیکس کی ایک رینج کی وضاحت ہوتی ہے۔
ازگر
print(my_list[1:3]) # پرنٹ (انڈیکس 1 سے 2 تک عناصر) آؤٹ پٹ: [20, 30] print(my_list[:3]) # پرنٹ (شروع سے انڈیکس 2 تک عناصر) آؤٹ پٹ: [10, 20, 30] print(my_list[2:]) # پرنٹ (عناصر انڈیکس 2 سے آخر تک) آؤٹ پٹ: [30, 40, 50] print(my_list[:-1]) # پرنٹ (عناصر شروع سے دوسرے آخری تک) آؤٹ پٹ: [10, 20, 30, 40]
3. عناصر کے ذریعے لوپنگ:
آپ لوپ کا استعمال کرتے ہوئے فہرست میں ہر عنصر پر اعادہ کر سکتے ہیں، جیسے کہ لوپ کے لیے۔
ازگر
میں آئٹم کے لیے: پرنٹ (آئٹم) my_list
4. فہم کی فہرست:
فہرست کی فہمیاں فہرستیں بنانے کا ایک جامع طریقہ فراہم کرتی ہیں۔ آپ انہیں بعض شرائط کی بنیاد پر عناصر کو فلٹر کرنے کے لیے بھی استعمال کر سکتے ہیں۔
ازگر
squared_list = [x ** 2 for x in my_list] # ایک نئی فہرست بنائیں جس میں اصل لسٹ اسکوائرڈ_لسٹ کے مربع عناصر ہوں print(squared_list)
5. انڈیکس کا طریقہ استعمال کرنا:
آپ فہرست میں کسی مخصوص عنصر کا اشاریہ تلاش کرنے کے لیے index() طریقہ استعمال کر سکتے ہیں۔
ازگر
index = my_list.index(30)print(index) # آؤٹ پٹ: 2
نتیجہ:
Python میں ایک فہرست میں عناصر تک رسائی کے لیے یہ کچھ عام طریقے ہیں۔ چاہے آپ کو انڈیکس کے ذریعے مخصوص عناصر کو بازیافت کرنے کی ضرورت ہو، سلائسنگ کا استعمال کرتے ہوئے عناصر کے ذیلی سیٹ کو نکالنا ہو، فہرستوں کے ساتھ مؤثر طریقے سے کام کرنے Python ،فہرست پر اعادہ کرنا ہو، یا کسی خاص عنصر کا اشاریہ تلاش کرنا ہو کے لیے ورسٹائل ٹولز فراہم کرتا ہے۔

List manipulation

List manipulation involves various operations that allow you to modify, add, remove, or reorder elements in a list. Here are some common list manipulation techniques in Python:

1. Adding Elements:

Append:

You can add a single element to the end of a list using the `append()` method.

```
python

my_list = [1, 2, 3]my_list.append(4)print(my_list)  # Output: [1, 2, 3, 4]
```

Extend:

You can add multiple elements from another iterable (such as another list) to the end of a list using the `extend()` method.

```
python

another_list = [5, 6, 7]my_list.extend(another_list)print(my_list)  # Output: [1,
2, 3, 4, 5, 6, 7]
```

2. Removing Elements:

Remove:

You can remove a specific element from the list using the `remove()` method.

```
python

my_list.remove(3)print(my_list)  # Output: [1, 2, 4, 5, 6, 7]
```

Pop:

You can remove and return the element at a specific index using the `pop()` method.

```
python

popped_element = my_list.pop(2)print(my_list)        # Output: [1, 2, 5, 6,
7]print(popped_element) # Output: 4
```

3. Inserting Elements:

You can insert an element at a specific index using the `insert()` method.

```
python

my_list.insert(2, 3)print(my_list)  # Output: [1, 2, 3, 5, 6, 7]
```

4. Reordering Elements:

Reverse:

You can reverse the order of elements in a list using the `reverse()` method.

```
python

my_list.reverse()print(my_list)  # Output: [7, 6, 5, 3, 2, 1]
```

Sort:

You can sort the elements of a list in ascending order using the `sort()` method.

```
python

my_list.sort()print(my_list)  # Output: [1, 2, 3, 5, 6, 7]
```

5. Copying Lists:

Shallow Copy:

You can create a shallow copy of a list using the `copy()` method or the slice notation.

```
python

copy_of_list = my_list.copy()
```

Deep Copy:

For nested lists or complex objects within a list, you can create a deep copy using the `copy.deepcopy()` function.

```
python

import copydeep_copy_of_list = copy.deepcopy(my_list)
```

Conclusion:

These are some common techniques for manipulating lists in Python. Whether you need to add, remove, insert, reorder, or copy elements in a list, Python provides versatile methods and functions to perform these operations efficiently.

فہرست میں ہیرا پھیری
فہرست کی ہیرا پھیری میں مختلف آپریشنز شامل ہوتے ہیں جو آپ کو فہرست میں عناصر کو تبدیل کرنے، شامل کرنے،
:میں فہرست میں ہیرا پھیری کی کچھ عام تکنیکیں یہ ہیں Python ہٹانے یا دوبارہ ترتیب دینے کی اجازت دیتے ہیں۔
:عناصر کو شامل کرنا .1
:شامل کریں
طریقہ استعمال کرکے فہرست کے آخر میں ایک عنصر شامل کرسکتے ہیں۔ ()append آپ
ازگر
آؤٹ پٹ: [1, 2, 3, 4] # (my_list)print(my_list.append(4)my_list = [1, 2, 3]
:توسیع
آپ توسیع () طریقہ کا استعمال کرتے ہوئے ایک فہرست کے آخر میں ایک اور قابل تکرار (جیسے دوسری فہرست) سے متعدد عناصر شامل کرسکتے ہیں۔
ازگر
آؤٹ پٹ: [1, 2, 3, 4, 5, 6, 7] # (my_list)print(my_list.extend(another_list)دوسری_لسٹ = [5, 6, 7]
:عناصر کو ہٹانا .2
:دور
آپ ریمو() طریقہ استعمال کرکے فہرست سے مخصوص عنصر کو ہٹا سکتے ہیں۔
ازگر
آؤٹ پٹ: [1, 2, 4, 5, 6, 7] # (my_list)print(my_list.remove(3)
:پاپ
طریقہ استعمال کرتے ہوئے ایک مخصوص انڈیکس میں عنصر کو ہٹا اور واپس کر سکتے ہیں۔ ()pop آپ
ازگر
آؤٹ # (popped_element)print(آؤٹ پٹ: [1, 2, 5, 6, 7] # (my_list)print(popped_element = my_list.pop(2)
پٹ: 4
:عناصر داخل کرنا .3
طریقہ استعمال کرتے ہوئے ایک مخصوص انڈیکس میں ایک عنصر داخل کر سکتے ہیں۔ ()insert آپ
ازگر
آؤٹ پٹ: [1, 2, 3, 5, 6, 7] # (my_list)print(my_list.insert(2, 3)
:عناصر کو دوبارہ ترتیب دینا .4
:معکوس
آپ ریورس() طریقہ استعمال کرکے فہرست میں عناصر کی ترتیب کو ریورس کرسکتے ہیں۔
ازگر
آؤٹ پٹ: [7, 6, 5, 3, 2, 1] # (my_list)print(my_list.reverse()
:ترتیب دیں
طریقہ استعمال کرکے فہرست کے عناصر کو صعودی ترتیب میں ترتیب دے سکتے ہیں۔ ()sort آپ
ازگر
آؤٹ پٹ: [1, 2, 3, 5, 6, 7] # (my_list)print(my_list.sort()
:فہرستیں کاپی کرنا .5
:اتلی کاپی
آپ کاپی() طریقہ یا سلائس نوٹیشن کا استعمال کرتے ہوئے فہرست کی اتلی کاپی بنا سکتے ہیں۔
ازگر
copy_of_list = my_list.copy()
:گہری کاپی
فنکشن کا استعمال کرتے ہوئے ایک گہری کاپی ()copy.deepcopy نیسٹڈ لسٹ یا لسٹ کے اندر پیچیدہ اشیاء کے لیے، آپ بنا سکتے ہیں۔
ازگر
درآمد کریں copydeep_copy_of_list = copy.deepcopy (my_list)
:نتیجہ
میں فہرستوں میں ہیرا پھیری کے لیے یہ کچھ عام تکنیکیں ہیں۔ چاہے آپ کو فہرست میں عناصر کو شامل کرنے، Python ان کارروائیوں کو مؤثر طریقے سے انجام دینے Python ،ہٹانے، داخل کرنے، دوبارہ ترتیب دینے یا کاپی کرنے کی ضرورت ہو کے لیے ورسٹائل طریقے اور افعال فراہم کرتا ہے۔

Tuple immutability

In Python, tuples are immutable, meaning once they are created, their content cannot be changed. Let's understand what immutability means in the context of tuples:

1. Creating a Tuple:

You create a tuple by enclosing comma-separated values within parentheses `()`.

```
python

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

2. Immutability:

Once a tuple is created, you cannot modify its elements, add new elements, or remove existing elements. Attempts to modify a tuple will result in an error.

```
python

my_tuple[0] = 4  # Error: 'tuple' object does not support item assignment
```

3. Why Tuples are Immutable:

Tuples are designed to be immutable for several reasons:

- **Performance**: Immutable objects are more efficient in terms of memory and performance because they can be cached and optimized by the interpreter.
- **Hashability**: Tuples are hashable, meaning they can be used as keys in dictionaries or elements in sets. Immutability ensures that the hash value of a tuple remains constant, making it suitable for these use cases.
- **Safety**: Immutable objects prevent accidental modification of data, reducing the risk of unintended side effects in your code.

4. When to Use Tuples:

Tuples are commonly used when you have a fixed collection of items that should not change over time. Some common use cases include:

- Representing fixed data structures, such as coordinates, RGB colors, or database records.
- Returning multiple values from a function.
- Passing arguments to functions that should not be modified.
- Using as keys in dictionaries or elements in sets.

Conclusion:

Understanding the immutability of tuples is essential for writing reliable and efficient Python code. While tuples cannot be modified after creation, their immutability provides benefits such as improved performance, hashability, and safety. Use tuples when you need a fixed collection of items that should not change over time, and use lists when you need a mutable collection of items that can be modified dynamically.

ٹیپل ناقابل تغیر
ناقابل تغیر ہوتے ہیں، یعنی ایک بار جب وہ بن جاتے ہیں، تو ان کے مواد کو تبدیل نہیں کیا جا سکتا۔ tuples ،میں Python
:آئیے سمجھتے ہیں کہ ٹیپلز کے تناظر میں ناقابل تغیر کا کیا مطلب ہے
:ایک ٹوپل بنانا .1
آپ قوسین () کے اندر کوما سے الگ کردہ اقدار کو بند کر کے ٹیپل بناتے ہیں۔
ازگر
my_tuple = (1، 2، 3)
:تغیر پذیری .2
ایک بار ٹیپل بن جانے کے بعد، آپ اس کے عناصر میں ترمیم نہیں کر سکتے، نئے عناصر شامل نہیں کر سکتے، یا موجودہ عناصر کو ہٹا نہیں سکتے۔ ٹیپل میں ترمیم کرنے کی کوششوں کے نتیجے میں ایک خرابی ہوگی۔
ازگر
آبجیکٹ آئٹم اسائنمنٹ کو سپورٹ نہیں کرتا 'tuple' :خرابی # 4 = [0]my_tuple
:کیوں ٹیپلز ناقابل تغیر ہیں .3
:کئی وجوہات کی بناء پر ناقابل تغیر ہونے کے لیے ڈیزائن کیے گئے ہیں Tuples
• کارکردگی: ناقابل تغیر اشیاء میموری اور کارکردگی کے لحاظ سے زیادہ کارآمد ہیں کیونکہ انہیں مترجم کے ذریعے کیش اور بہتر بنایا جا سکتا ہے۔
• ہیش ایبلٹی: ٹیپلز ہیش ایبل ہیں، یعنی وہ لغت میں کلیدوں یا سیٹوں میں عناصر کے طور پر استعمال ہو سکتے ہیں۔ تغیر پذیری اس بات کو یقینی بناتی ہے کہ ٹیوپل کی ہیش ویلیو مستقل رہے، جو اسے استعمال کے ان معاملات کے لیے موزوں بناتی ہے۔
• سیفٹی: غیر تبدیل شدہ اشیاء ڈیٹا کی حادثاتی ترمیم کو روکتی ہیں، آپ کے کوڈ میں غیر ارادی ضمنی اثرات کے خطرے کو کم کرتی ہیں۔
:ٹیپلز کب استعمال کریں .4
عام طور پر اس وقت استعمال ہوتے ہیں جب آپ کے پاس آئٹمز کا ایک مقررہ مجموعہ ہوتا ہے جو وقت کے ساتھ Tuples
:تبدیل نہیں ہونا چاہیے۔ کچھ عام استعمال کے معاملات میں شامل ہیں
• فکسڈ ڈیٹا ڈھانچے کی نمائندگی کرنا، جیسے کوآرڈینیٹ، آر جی بی رنگ، یا ڈیٹا بیس ریکارڈ۔
• ایک فنکشن سے متعدد اقدار کو لوٹانا۔
• فنکشنز کو دلائل دینا جن میں ترمیم نہیں کی جانی چاہیے۔
• لغت میں کلیدوں یا سیٹوں میں عناصر کے طور پر استعمال کرنا۔
:نتیجہ
کی تغیر پذیری کو سمجھنا ضروری ہے۔ اگرچہ ٹیپلز کو تخلیق کے tuples کوڈ لکھنے کے لیے Python قابل اعتماد اور موثر بعد تبدیل نہیں کیا جا سکتا ہے، لیکن ان کی عدم تبدیلی فوائد فراہم کرتی ہے جیسے بہتر کارکردگی، ہیش ایبلٹی، اور حفاظت۔ جب آپ کو آئٹمز کے ایک مقررہ مجموعے کی ضرورت ہو جو وقت کے ساتھ تبدیل نہ ہوں تو ٹیپلز کا استعمال کریں، اور جب آپ کو ایسی اشیاء کے متغیر مجموعہ کی ضرورت ہو جس میں متحرک طور پر ترمیم کی جا سکتی ہو تو فہرستیں استعمال کریں۔

Sets: creation, operations, and methods

Sets in Python are unordered collections of unique elements. They are useful for storing and performing operations on distinct items. Here's how you can work with sets, including creation, operations, and methods:

1. Creating Sets:

You can create a set by enclosing comma-separated values within curly braces `{}`.

```
python

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

Alternatively, you can use the `set()` constructor by passing an iterable such as a list or a tuple.

```
python

my_set = set([1, 2, 3, 4])
```

2. Set Operations:

Union (`|`):

Combines elements from two sets, removing duplicates.

```
python

set1 = {1, 2, 3}set2 = {3, 4, 5}union_set = set1 | set2print(union_set)  # Output: {1, 2, 3, 4, 5}
```

Intersection (`&`):

Returns elements that are common to both sets.

```
python

intersection_set = set1 & set2print(intersection_set)  # Output: {3}
```

Difference (`-`):

Returns elements in the first set that are not in the second set.

```
python

difference_set = set1 - set2print(difference_set)  # Output: {1, 2}
```

Symmetric Difference (`^`):

Returns elements that are in either of the sets, but not both.

```
python

symmetric_difference_set = set1 ^ set2print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

3. Set Methods:

Add:

Adds a single element to the set.

```
python

my_set.add(5)
```

Remove:

Removes a specified element from the set. Raises an error if the element is not present.

```
python

my_set.remove(3)
```

Discard:

Removes a specified element from the set, but does not raise an error if the element is not present.

```
python

my_set.discard(3)
```

Clear:

Removes all elements from the set, making it empty.

```
python

my_set.clear()
```

Length:

Returns the number of elements in the set.

```
python

length = len(my_set)
```

Conclusion:

Sets in Python provide a convenient way to work with unique collections of elements and perform various operations such as union, intersection, difference, and more. Understanding

how to create sets, perform set operations, and use set methods is essential for working with sets effectively in Python.

سیٹ: تخلیق، آپریشن، اور طریقے
میں سیٹ منفرد عناصر کے غیر ترتیب شدہ مجموعہ ہیں۔ وہ الگ الگ اشیاء کو ذخیرہ کرنے اور آپریشن کرنے کے Python
:لیے مفید ہیں۔ یہاں ہے کہ آپ سیٹ کے ساتھ کیسے کام کر سکتے ہیں، بشمول تخلیق، آپریشن اور طریقے
:سیٹ بنانا .1
آپ گھوبگھرالی منحنی خطوط وحدانی {} کے اندر کوما سے الگ کردہ اقدار کو بند کر کے ایک سیٹ بنا سکتے ہیں۔
ازگر
my_set = {1, 2, 3, 4}
متبادل کے طور پر، آپ سیٹ() کنسٹرکٹر کو دوبارہ استعمال کر سکتے ہیں جیسے کہ فہرست یا ٹوپل۔
ازگر
سیٹ ([1, 2, 3, 4]) = my_set
:سیٹ آپریشنز .2
:(|) یونین
دو سیٹوں سے عناصر کو یکجا کرتا ہے، ڈپلیکیٹس کو ہٹاتا ہے۔
ازگر
آؤٹ پٹ: {1, 2, 3, 4, 5} # set2print(union_set) = set1 | set2 = {3, 4, 5}union_set = set1 = {1, 2, 3}
:(&) چوراہا
عناصر لوٹاتا ہے جو دونوں سیٹوں میں مشترک ہیں۔
ازگر
آؤٹ پٹ: {3} # intersection_set = set1 & set2print(intersection_set)
:(-) فرق
پہلے سیٹ میں ایسے عناصر لوٹاتا ہے جو دوسرے سیٹ میں نہیں ہیں۔
ازگر
فرق_سیٹ = سیٹ 1 - سیٹ2 پرنٹ (فرق_سیٹ) # آؤٹ پٹ: {1، 2}
:(^) ہم آہنگی فرق
وہ عناصر لوٹاتا ہے جو سیٹوں میں سے کسی ایک میں ہیں، لیکن دونوں میں نہیں۔
ازگر
آؤٹ پٹ: {1, 2, 4, 5} # symmetric_difference_set = set1 ^ set2print(symmetric_difference_set)
:طریقے طے کریں .3
:شامل کریں
سیٹ میں ایک عنصر شامل کرتا ہے۔
ازگر
my_set.add(5)
:دور
سیٹ سے ایک مخصوص عنصر کو ہٹاتا ہے۔ اگر عنصر موجود نہ ہو تو ایک خرابی پیدا کرتا ہے۔
ازگر
my_set.remove(3)
:رد کریں
سیٹ سے ایک مخصوص عنصر کو ہٹاتا ہے، لیکن اگر عنصر موجود نہیں ہے تو کوئی خرابی پیدا نہیں کرتا ہے۔
ازگر
my_set.discard(3)
:صاف کریں
سیٹ سے تمام عناصر کو ہٹاتا ہے، اسے خالی بناتا ہے۔
ازگر
my_set.clear()
:لمبائی
سیٹ میں عناصر کی تعداد لوٹاتا ہے۔
ازگر
لمبائی = لین (میرا_سیٹ)
:نتیجہ

Python میں سیٹ عناصر کے منفرد مجموعوں کے ساتھ کام کرنے اور مختلف آپریشنز جیسے کہ یونین، انٹرسیکشن، فرق، Python میں سیٹوں کے ساتھ مؤثر طریقے سے کام کرنے کے لیے Python وغیرہ کو انجام دینے کا ایک آسان طریقہ فراہم کرتے ہیں۔ سیٹ بنانے، سیٹ آپریشن کرنے، اور سیٹ طریقے استعمال کرنے کے طریقے کو سمجھنا ضروری ہے۔

Using list comprehensions

List comprehensions are a concise and powerful way to create lists in Python. They allow you to generate new lists by applying an expression to each element of an existing iterable (such as a list, tuple, or range) and optionally filtering the elements based on a condition. Here's how you can use list comprehensions:

Basic List Comprehension Syntax:

The basic syntax of a list comprehension consists of square brackets `[]`, containing an expression followed by a `for` clause.

```
python

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

Example 1: Creating a List of Squares:

Let's create a list of squares of numbers from 0 to 9 using a list comprehension.

```
python

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

Example 2: Filtering Even Numbers:

You can also include an optional `if` clause to filter elements based on a condition.

```
python

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

Example 3: Flattening a Nested List:

List comprehensions can also be used to flatten nested lists.

```
python

nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]flattened_list = [x for sublist in nested_list for x in sublist]print(flattened_list)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Example 4: Creating a List of Tuples:

You can generate lists of tuples using list comprehensions.

```
python

coordinates = [(x, y) for x in range(3) for y in range(2)]print(coordinates)  # Output: [(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

Example 5: Using Function in List Comprehension:

You can apply a function to elements in a list comprehension.

```
python

words = ['hello', 'world', 'python']upper_case_words = [word.upper() for word in
words]print(upper_case_words)  # Output: ['HELLO', 'WORLD', 'PYTHON']
```

Conclusion:

List comprehensions provide a concise and readable way to create lists in Python, making your code more expressive and efficient. By mastering list comprehensions, you can simplify your code and perform complex operations with ease.

فہرست کی تفہیم کا استعمال کرتے ہوئے
میں فہرستیں بنانے کا ایک مختصر اور طاقتور طریقہ ہے۔ وہ آپ کو موجودہ اٹیبل کے ہر عنصر (جیسے Python فہرست فہم
فہرست، ٹوپل، یا رینج) پر ایک اظہار لگا کر اور اختیاری طور پر کسی شرط کی بنیاد پر عناصر کو فلٹر کرکے نئی فہرستیں
:بنانے کی اجازت دیتے ہیں۔ یہاں یہ ہے کہ آپ فہرست کی تفہیم کو کس طرح استعمال کرسکتے ہیں
:بنیادی فہرست فہم نحو
a for clause فہرست کی فہم کا بنیادی نحو مربع بریکٹ [] پر مشتمل ہوتا ہے، جس میں ایک اظہار ہوتا ہے جس کے بعد
ہوتا ہے۔
ازگر
[اعادہ میں آئٹم کے لیے اظہار] = new_list
:مثال 1: مربعوں کی فہرست بنانا
آئیے فہرست کی تفہیم کا استعمال کرتے ہوئے 0 سے 9 تک نمبروں کے مربعوں کی فہرست بنائیں۔
ازگر
پرنٹ (مربع) # آؤٹ پٹ: [0، 1، 4، 9، 16، 25، 36، 49، 64، 81] [رینج (10) x in برائے x**2] = مربع
:مثال 2: یکساں نمبروں کو فلٹر کرنا
آپ شرط کی بنیاد پر عناصر کو فلٹر کرنے کے لیے ایک اختیاری اگر شق بھی شامل کر سکتے ہیں۔
ازگر
آؤٹ پٹ: [0, 2, 4, 6, 8] # (even_numbers) پرنٹ[print(even_numbers = [x for x in range(10) if x % 2 == 0
:مثال 3: نیسٹڈ لسٹ کو ہموار کرنا
فہرست کی تفہیم کو نیسٹڈ لسٹوں کو ہموار کرنے کے لیے بھی استعمال کیا جا سکتا ہے۔
ازگر
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]flatened_list = [x for sublist in nested_list in x for
sublist]print(flatened_list) # آؤٹ پٹ: [1, 2، 3، 4، 5، 6، 7، 8، 9]
:مثال 4: ٹیپلز کی فہرست بنانا
کی فہرستیں تیار کر سکتے ہیں۔ tuples آپ فہرست کی تفہیم کا استعمال کرتے ہوئے
ازگر
پرنٹ (کوآرڈینیٹس) # آؤٹ پٹ: [(0, 0), (0, 1), ([کے لیے (2) y کے لیے (3) رینج میں x رینج میں (x, y)] = کوآرڈینیٹ
1, 0), (1، 1)، (2، 0)، (2، 1)]
:مثال 5: فہرست فہم میں فنکشن کا استعمال
آپ فہرست کی تفہیم میں عناصر پر فنکشن لاگو کرسکتے ہیں۔
ازگر
الفاظ میں لفظ کے ()upper_case_words = [word.upper 'python'] ،'ہیلو'، 'دنیا'] = الفاظ
['HELLO', 'WORLD', 'PYTHON'] :آؤٹ پٹ # (upper_case_words)پرنٹ[لیے
:نتیجہ
میں فہرستیں بنانے کے لیے ایک جامع اور پڑھنے کے قابل طریقہ فراہم کرتی ہے، جس سے آپ Python فہرست کی تفہیم
کے کوڈ کو مزید تاثراتی اور موثر بنایا جاتا ہے۔ فہرست کی فہموں میں مہارت حاصل کر کے، آپ اپنے کوڈ کو آسان بنا سکتے
ہیں اور آسانی کے ساتھ پیچیدہ کام انجام دے سکتے ہیں۔

Dictionaries: creation, accessing elements, dictionary methods

Dictionaries in Python are unordered collections of key-value pairs, allowing you to store and retrieve data using keys instead of numerical indices. Here's an overview of working with dictionaries, including creation, accessing elements, and common dictionary methods:

Creation:

Dictionaries are created using curly braces `{}` and key-value pairs separated by colons `:`.

Example:

```
python

# Empty dictionaryempty_dict = {} # Dictionary with initial valuesperson = {"name":
"John", "age": 30, "city": "New York"}
```

Accessing Elements:

You can access elements in a dictionary by using square brackets `[]` and providing the key of the element you want to access.

Example:

```
python

person = {"name": "John", "age": 30, "city": "New York"} # Accessing
elementsprint(person["name"])  # Output: Johnprint(person["age"])   # Output: 30
```

Dictionary Methods:

Python dictionaries have several built-in methods for performing common operations:

1. `get(key[, default])`: Returns the value associated with the specified key. If the key is not found, it returns the specified default value (or `None` if not specified).

   ```
   python

   1. person = {"name": "John", "age": 30}print(person.get("name"))     # Output:
      Johnprint(person.get("city", "NA"))  # Output: NA
   ```

2. `keys()`: Returns a view object containing the keys of the dictionary.

   ```
   python

   3. person = {"name": "John", "age": 30}print(person.keys())  # Output:
      dict_keys(['name', 'age'])
   ```

4. `values()`: Returns a view object containing the values of the dictionary.

   ```
   python
   ```

```
5. person = {"name": "John", "age": 30}print(person.values())  # Output:
   dict_values(['John', 30])
```

6. `items()`: Returns a view object containing the key-value pairs of the dictionary as tuples.

   ```
   python
   ```

```
7. person = {"name": "John", "age": 30}print(person.items())  # Output:
   dict_items([('name', 'John'), ('age', 30)])
```

8. `update()`: Updates the dictionary with the key-value pairs from another dictionary or iterable.

   ```
   python
   ```

```
9. person = {"name": "John", "age": 30}person.update({"city": "New York",
   "country": "USA"})print(person)  # Output: {'name': 'John', 'age': 30,
   'city': 'New York', 'country': 'USA'}
```

10. `pop(key[, default])`: Removes and returns the value associated with the specified key. If the key is not found, it returns the specified default value (or raises a `KeyError` if not specified).

    ```
    python
    ```

```
1. person = {"name": "John", "age": 30}print(person.pop("age"))  # Output: 30
```

Conclusion:

Dictionaries in Python are versatile data structures that allow you to store and retrieve data using keys. By understanding how to create dictionaries, access elements, and use dictionary methods, you can efficiently work with dictionaries in your Python code.

لغت: تخلیق، عناصر تک رسائی، لغت کے طریقے

میں ڈکشنریاں کلیدی قدر کے جوڑوں کے غیر ترتیب شدہ مجموعے ہیں، جو آپ کو عددی اشاریہ کے بجائے کیز کا Python استعمال کرتے ہوئے ڈیٹا کو ذخیرہ کرنے اور بازیافت کرنے کی اجازت دیتی ہیں۔ یہاں لغات کے ساتھ کام کرنے کا ایک جائزہ ہے، بشمول تخلیق، عناصر تک رسائی، اور عام لغت کے طریقے:

تخلیق:

لغتیں گھوبگھرالی منحنی خطوط وحدانی {} اور کلیدی قدر کے جوڑوں کو کالون کے ذریعہ الگ کرکے بنائی جاتی ہیں :۔

مثال:

ازگر

empty_dict = {} # خالی ڈکشنری ابتدائی ویلیو پرسن کے ساتھ ڈکشنری = {"نام": "جان"، "عمر": 30، "شہر": "نیو یارک"}

عناصر تک رسائی:

آپ مربع بریکٹ [] کا استعمال کرکے اور جس عنصر تک آپ رسائی حاصل کرنا چاہتے ہیں اس کی کلید فراہم کرکے آپ لغت میں عناصر تک رسائی حاصل کرسکتے ہیں۔

مثال:

ازگر

person = {"نام": "جان"، "عمر": 30، "شہر": "نیو یارک"} # رسائی عناصر پرنٹ (person["name"]) # آؤٹ پٹ: John
print(person["age"]) # آؤٹ پٹ: 30

لغت کے طریقے:

عام کاموں کو انجام دینے کے لیے ازگر کی لغات میں کئی بلٹ ان طریقے ہیں:

1. get(key[, default]): مخصوص کلید سے وابستہ قدر لوٹاتا ہے۔ اگر کلید نہیں ملتی ہے، تو یہ مخصوص ڈیفالٹ قدر (یا کوئی نہیں اگر متعین نہ ہو) لوٹاتا ہے۔

ازگر

print(person.get("name")) # آؤٹ پٹ: John
شخص = {"نام": "جان"، "عمر": 30} پرنٹ (person.get("city", "NA")) # آؤٹ پٹ : N / A

2. کیز(): لغت کی کلیدوں پر مشتمل ویو آبجیکٹ لوٹاتا ہے۔

ازگر

شخص = {"نام": "جان"، "عمر": 30} پرنٹ (person.keys()) # آؤٹ پٹ: dict_keys(['name', 'age'])

3. اقدار(): لغت کی قدروں پر مشتمل ویو آبجیکٹ لوٹاتا ہے۔

ازگر

شخص = {"نام": "جان"، "عمر": 30} پرنٹ (person.values()) # آؤٹ پٹ: dict_values(['John', 30])

4. آئٹمز(): ایک ویو آبجیکٹ لوٹاتا ہے جس میں لغت کے کلیدی قدر کے جوڑے بطور ٹیپلز ہوتے ہیں۔

ازگر

شخص = {"نام": "جان"، "عمر": 30} پرنٹ (person.items()) # آؤٹ پٹ: dict_items([('نام'، 'جان')، ('عمر'، 30)])

5. اپ ڈیٹ(): کسی اور لغت سے کلیدی قدر کے جوڑوں کے ساتھ لغت کو اپ ڈیٹ کرتا ہے یا دوبارہ قابل استعمال۔

ازگر

شخص = {"نام": "جان"، "عمر": 30} person.update({"city": "New York", "country": "USA"})print(person) # آؤٹ پٹ: {'نام': 'جان'، 'عمر': 30، 'شہر': 'نیویارک'، 'ملک': 'USA'}

6. pop(key[, default]): مخصوص کلید سے وابستہ قدر کو ہٹاتا اور واپس کرتا ہے۔ اگر کلید نہیں ملتی ہے، تو یہ KeyError کو بڑھاتا ہے یا اگر متعین نہ ہو تو) مخصوص ڈیفالٹ قدر لوٹاتا ہے۔

ازگر

شخص = {"نام": "جان"، "عمر": 30} پرنٹ (person.pop("عمر")) # آؤٹ پٹ: 30

نتیجہ:

میں ڈکشنریاں ورسٹائل ڈیٹا سٹرکچر ہیں جو آپ کو کیز کا استعمال کرتے ہوئے ڈیٹا کو اسٹور اور بازیافت کرنے کی Python اجازت دیتی ہیں۔ لغت بنانے، عناصر تک رسائی اور لغت کے طریقے استعمال کرنے کے طریقے کو سمجھ کر، آپ اپنے کوڈ میں لغات کے ساتھ مؤثر طریقے سے کام کر سکتے ہیں۔ Python

Nested data structures

Nested data structures in Python refer to data structures that can contain other data structures as elements. Common examples include lists of lists, dictionaries of dictionaries, lists of dictionaries, dictionaries of lists, and combinations thereof. They allow you to represent hierarchical or structured data in a flexible and organized way. Here are some examples of nested data structures:

1. List of Lists:

```
python

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

2. List of Dictionaries:

```
python

people = [    {"name": "John", "age": 30},    {"name": "Alice", "age": 25},
{"name": "Bob", "age": 35}]
```

3. Dictionary of Dictionaries:

```
python

users = {    "john_doe": {"name": "John Doe", "age": 30},    "alice_smith":
{"name": "Alice Smith", "age": 25},    "bob_jones": {"name": "Bob Jones", "age":
35}}
```

4. Dictionary of Lists:

```
python

grades = {    "math": [90, 85, 95],    "science": [80, 75, 85],    "history": [85,
90, 88]}
```

5. List of Tuples:

```
python

students = [    ("John", 30),    ("Alice", 25),    ("Bob", 35)]
```

Accessing Elements:

You can access elements in nested data structures using multiple index or key lookups, depending on the structure.

```
python

# Accessing elements in a list of listsprint(matrix[0][1])  # Output: 2 # Accessing
elements in a list of dictionariesprint(people[1]["name"])  # Output: Alice #
```

```
Accessing elements in a dictionary of dictionariesprint(users["john_doe"]["age"])
# Output: 30 # Accessing elements in a dictionary of listsprint(grades["math"][1])
# Output: 85
```

Iterating Over Nested Data Structures:

You can use nested loops to iterate over nested data structures, or use list comprehensions or generator expressions for more concise code.

```
python

# Iterating over a list of listsfor row in matrix:    for element in row:
print(element) # Using list comprehension to flatten a list of
listsflattened_matrix = [element for row in matrix for element in row]
```

Conclusion:

Nested data structures in Python are powerful tools for representing complex data relationships and hierarchies. By understanding how to create, access, and iterate over nested data structures, you can effectively work with structured data in your Python programs.

نیسٹڈ ڈیٹا ڈھانچے
پائتھون میں نیسٹڈ ڈیٹا سٹرکچرز ڈیٹا سٹرکچرز کا حوالہ دیتے ہیں جن میں دیگر ڈیٹا سٹرکچر بطور عناصر شامل ہو سکتے ہیں۔ عام مثالوں میں فہرستوں کی فہرستیں، لغات کی لغات، لغات کی فہرستیں، فہرستوں کی لغات، اور ان کے مجموعے شامل ہیں۔ وہ آپ کو لچکدار اور منظم طریقے سے درجہ بندی یا ساختی ڈیٹا کی نمائندگی کرنے کی اجازت دیتے ہیں۔ نیسٹڈ ڈیٹا ڈھانچے کی کچھ مثالیں یہ ہیں:

1. فہرستوں کی فہرست:
ازگر
میٹرکس = [[1، 2، 3]، [4، 5، 6]، [7، 8، 9]]
2. لغت کی فہرست:
ازگر
لوگ = [{"نام": "جان"، "عمر": 30}، {"نام": "ایلس"، "عمر": 25}، {"نام": "باب"، "عمر": 35}]
3. لغت کی لغت:
ازگر
user = { "john_doe": {"name": "John Doe", "عمر": 30}, "alice_smith": {"name": "Alice Smith", "عمر": 25}, "bob_jones": { "نام": "باب جونز"، "عمر": 35" } }
4. فہرستوں کی لغت:
ازگر
درجات = { "ریاضی": [90, 85, 95]، "سائنس": [80, 75, 85]، "تاریخ": [85, 90, 88]}
5. ٹیپلز کی فہرست:
ازگر
طلباء = [("جان"، 30)، ("ایلس"، 25)، ("باب، 35)]
عناصر تک رسائی:
آپ ساخت کے لحاظ سے متعدد انڈیکس یا کلیدی تلاش کا استعمال کرتے ہوئے نیسٹڈ ڈیٹا ڈھانچے میں عناصر تک رسائی حاصل کر سکتے ہیں۔
ازگر
لسٹ پرنٹ (میٹرکس[0][1]) کی فہرست میں عناصر تک رسائی # آؤٹ پٹ: 2 # ڈکشنری پرنٹ کی فہرست میں عناصر # ["john_doe"]صارفین) آؤٹ پٹ: ایلس # ڈکشنری میں عناصر تک رسائی ڈکشنری پرنٹ # (["name"]لوگ[1]) تک رسائی آؤٹ پٹ: 30 # فہرست پرنٹ (گریڈز["ریاضی"][1]) کی لغت میں عناصر تک رسائی # آؤٹ پٹ: 85 # (["age"]
نیسٹڈ ڈیٹا سٹرکچر پر تکرار کرنا:
آپ نیسٹڈ ڈیٹا ڈھانچے پر اعادہ کرنے کے لیے نیسٹڈ لوپس استعمال کر سکتے ہیں، یا مزید جامع کوڈ کے لیے فہرست فہم یا جنریٹر ایکسپریشنز کا استعمال کر سکتے ہیں۔
ازگر
میٹرکس میں قطار کے لیے فہرستوں کی فہرست پر اعادہ کرنا: قطار میں عنصر کے لیے: پرنٹ(عنصر) # فہرستوں کی # فہرست کو فلیٹ کرنے کے لیے فلیٹ_میٹرکس = [صفر میں عنصر کے لیے میٹرکس میں قطار کے لیے عنصر]
نتیجہ:
میں نیسٹڈ ڈیٹا ڈھانچے پیچیدہ ڈیٹا تعلقات اور درجہ بندی کی نمائندگی کرنے کے لیے طاقتور ٹولز ہیں۔ نیسٹڈ ڈیٹا Python پروگراموں میں سٹرکچرڈ ڈیٹا Python سٹرکچرز کو بنانے، ان تک رسائی اور اعادہ کرنے کے طریقے کو سمجھ کر، آپ اپنے کے ساتھ مؤثر طریقے سے کام کر سکتے ہیں۔

Defining and calling functions

In Python, functions are blocks of reusable code that perform a specific task. They allow you to break down your code into smaller, more manageable pieces, improving readability, reusability, and maintainability. Here's how you define and call functions in Python:

Defining Functions:

You can define a function using the `def` keyword followed by the function name and parameters (if any). The function body is indented below the function definition. Here's the general syntax:

```
python

def function_name(parameter1, parameter2, ...):    # Function body    # Perform
some tasks    return result  # optional
```

- `def`: Keyword used to define a function.
- `function_name`: Name of the function.
- `parameters`: Input values passed to the function (optional).
- `return`: Statement used to return a value from the function (optional).

Example:

```
python

def greet(name):    return f"Hello, {name}!" def add(x, y):    return x + y
```

Calling Functions:

To call a function, you simply use the function name followed by parentheses `()`, optionally passing any required arguments inside the parentheses. If the function returns a value, you can assign it to a variable or use it directly.

```
python

result = function_name(argument1, argument2, ...)
```

Example:

```
python

greeting = greet("Alice")print(greeting)  # Output: Hello, Alice! sum_result =
add(5, 3)print(sum_result)  # Output: 8
```

Function without Return Value:

Not all functions need to return a value. Some functions may perform tasks without producing an explicit result. In such cases, you can omit the `return` statement, or explicitly return `None`.

```
python

def say_hello():    print("Hello!") say_hello()  # Output: Hello!
```

Function with Default Parameters:

You can provide default values for function parameters. These default values are used when the function is called without specifying the corresponding arguments.

```
python

def greet(name="Guest"):    return f"Hello, {name}!" print(greet())        #
Output: Hello, Guest!print(greet("Alice")) # Output: Hello, Alice!
```

Function with Keyword Arguments:

You can also pass arguments to functions using keyword arguments. This allows you to specify arguments by their parameter names, regardless of their order.

```
python

def greet(greeting, name):    return f"{greeting},
{name}!" print(greet(name="Alice", greeting="Good morning"))  # Output: Good
morning, Alice!
```

These are the basics of defining and calling functions in Python. Functions are fundamental building blocks in Python programming, enabling code reuse and organization.

افعال کی تعریف اور کال کرنا
میں، فنکشنز دوبارہ قابل استعمال کوڈ کے بلاکس ہیں جو ایک مخصوص کام انجام دیتے ہیں۔ وہ آپ کو اپنے کوڈ کو Python چھوٹے، زیادہ قابل انتظام ٹکڑوں میں تقسیم کرنے کی اجازت دیتے ہیں، پڑھنے کی اہلیت، دوبارہ پریوست، اور برقرار رکھنے :میں فنکشنز کی وضاحت اور کال کیسے کرتے ہیں Python کی صلاحیت کو بہتر بناتے ہیں۔ یہ ہے کہ آپ
:افعال کی تعریف
کلیدی لفظ کا استعمال کرتے ہوئے فنکشن کی وضاحت کر سکتے ہیں جس کے بعد فنکشن کا نام اور پیرامیٹرز (اگر def آپ :کوئی ہیں)۔ فنکشن باڈی فنکشن ڈیفینیشن کے نیچے انڈینٹڈ ہے۔ یہاں عام نحو ہے
ازگر
فنکشن باڈی # کچھ کام انجام دیں نتیجہ واپسی # :(...اختیاری # def function_name(parameter1, parameter2,
کلیدی لفظ کسی فنکشن کی وضاحت کے لیے استعمال ہوتا ہے۔ :def •
فنکشن_نام: فنکشن کا نام۔
پیرامیٹرز: ان پٹ ویلیوز فنکشن کو دی گئیں (اختیاری)۔ •
واپسی: بیان کا استعمال فنکشن (اختیاری) سے قدر واپس کرنے کے لیے ہوتا ہے۔ •
:مثال
ازگر
def greet(name): واپس f"Hello, {name}!" def add(x, y): واپس کریں x + y
:کالنگ کے افعال
فنکشن کو کال کرنے کے لیے، آپ صرف فنکشن کا نام استعمال کرتے ہیں جس کے بعد قوسین ()، اختیاری طور پر قوسین کے اندر کسی بھی مطلوبہ دلائل کو پاس کرتے ہیں۔ اگر فنکشن ایک قدر واپس کرتا ہے، تو آپ اسے متغیر کو تفویض کرسکتے ہیں یا اسے براہ راست استعمال کرسکتے ہیں۔
ازگر
نتیجہ = فنکشن_نام (دلیل 1، دلیل 2، ...)
:مثال
ازگر
greeting = greet("Alice")print(greeting) # آؤٹ پٹ: ہیلو، ایلس !sum_result = add(5, 3)print(sum_result) # آؤٹ پٹ: 8
:ریٹرن ویلیو کے بغیر فنکشن
تمام افعال کو قدر واپس کرنے کی ضرورت نہیں ہے۔ کچھ افعال واضح نتیجہ پیدا کیے بغیر کام انجام دے سکتے ہیں۔ ایسی صورتوں میں، آپ واپسی کے بیان کو چھوڑ سکتے ہیں، یا واضح طور پر کوئی نہیں واپس کر سکتے ہیں۔
ازگر
def say_hello(): print("Hello!") say_hello() # آؤٹ پٹ: ہیلو!
:ڈیفالٹ پیرامیٹرز کے ساتھ فنکشن
آپ فنکشن پیرامیٹرز کے لیے ڈیفالٹ ویلیوز فراہم کر سکتے ہیں۔ یہ ڈیفالٹ قدریں استعمال ہوتی ہیں جب فنکشن کو متعلقہ دلائل کی وضاحت کیے بغیر کال کیا جاتا ہے۔
ازگر
def greet(name="Guest"): واپس f"Hello, {name}!" پرنٹ(گریٹ(("ایلس")) پرنٹ(گریٹ()) # آؤٹ پٹ: ہیلو، گیسٹ! !# آؤٹ پٹ: ہیلو، ایلس
:مطلوبہ الفاظ کے دلائل کے ساتھ فنکشن
آپ کلیدی الفاظ کے دلائل کا استعمال کرتے ہوئے فنکشنز کو دلائل بھی دے سکتے ہیں۔ یہ آپ کو ان کے پیرامیٹر کے ناموں کے ذریعہ دلائل کی وضاحت کرنے کی اجازت دیتا ہے، قطع نظر ان کے آرڈر کے۔
ازگر
def greet(سلام، نام): واپس f"{greeting}, {name}!" پرنٹ(گریٹ(نام="ایلس"، گریٹنگ="گڈ مارننگ")) # آؤٹ پٹ: گڈ !مارننگ، ایلس
پروگرامنگ میں بنیادی تعمیراتی Python میں فنکشن کی وضاحت اور کال کرنے کی بنیادی باتیں ہیں۔ فنکشنز Python یہ بلاکس ہیں، کوڈ کے دوبارہ استعمال اور تنظیم کو فعال کرتے ہیں۔

Function parameters and arguments

In Python, functions can have parameters, which are input values passed to the function, and return values, which are the values returned by the function after it has completed its execution. Here's how you can work with parameters and return values in Python functions:

Parameters:

Parameters are defined within the parentheses following the function name. When calling the function, you pass arguments to these parameters. There are different types of parameters:

1. **Positional Parameters**: These are parameters that are matched by position.

```
python

def greet(name):    print(f"Hello, {name}!") greet("Alice")  # Output: Hello,
Alice!
```

2. **Keyword Parameters**: These are parameters specified by their names.

```
python

def greet(name, greeting):    print(f"{greeting}, {name}!") greet(name="Alice",
greeting="Good morning")  # Output: Good morning, Alice!
```

3. **Default Parameters**: These are parameters that have default values assigned to them.

```
python

def greet(name="Guest", greeting="Hello"):    print(f"{greeting}, {name}!") greet()
# Output: Hello, Guest!greet("Bob")             # Output: Hello, Bob!
greet(greeting="Hi")     # Output: Hi, Guest!
```

Return Values:

Functions in Python can return values using the `return` statement. You can return single or multiple values from a function. If no `return` statement is specified, the function returns `None` by default.

```
python

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

You can return multiple values by separating them with commas. When multiple values are returned, they are returned as a tuple.

```
python

def square_and_cube(x):    return x**2, x**3 result =
square_and_cube(3)print(result)  # Output: (9, 27)
```

You can unpack the returned tuple directly into variables:

```
python

square, cube = square_and_cube(3)print(square)  # Output: 9print(cube)    # Output:
27
```

Returning Early:

You can use the `return` statement to exit a function early if a condition is met. This can be useful for error handling or optimization.

```
python

def divide(x, y):    if y == 0:        return "Error: Division by zero"    return x
/ y result = divide(10, 2)  # Output: 5.0
```

These are the basics of working with parameters and return values in Python functions. They allow you to create flexible and reusable code by passing data into functions and returning results back to the caller.

فنکشن پیرامیٹرز اور دلائل
میں، فنکشنز کے پیرامیٹرز ہو سکتے ہیں، جو کہ فنکشن کو دی گئی ان پٹ ویلیوز ہیں، اور ریٹرن ویلیوز ہیں، جو کہ Python
فنکشنز میں پیرامیٹرز اور واپسی Python فنکشن کے مکمل ہونے کے بعد واپس کی جانے والی اقدار ہیں۔ یہاں یہ ہے کہ آپ
:کی اقدار کے ساتھ کیسے کام کر سکتے ہیں
:پیرامیٹرز
پیرامیٹرز کو فنکشن کے نام کے بعد قوسین میں بیان کیا گیا ہے۔ فنکشن کو کال کرتے وقت، آپ ان پیرامیٹرز پر دلائل دیتے
:ہیں۔ پیرامیٹرز کی مختلف اقسام ہیں
پوزیشنی پیرامیٹرز: یہ وہ پیرامیٹرز ہیں جو پوزیشن سے مماثل ہیں۔ .1
ازگر
!آؤٹ پٹ: ہیلو، ایلس # def greet(name): print(f"Hello, {name}!") greet("Alice")
مطلوبہ الفاظ کے پیرامیٹرز: یہ ان کے ناموں سے متعین کردہ پیرامیٹرز ہیں۔ .2
ازگر
def greet(name, greeting): print(f"{greeting}, {name}!") greet(name="Alice", greeting="Good
!آؤٹ پٹ: گڈ مارننگ، ایلس # ("morning
پہلے سے طے شدہ پیرامیٹرز: یہ وہ پیرامیٹرز ہیں جن کے لیے پہلے سے طے شدہ قدریں تفویض کی گئی ہیں۔ .3
ازگر
def greet(name="Guest", greeting="Hello"): print(f"{greeting}, {name}!") greet() # آؤٹ پٹ: Hello,
Guest!greet("Bob") # آؤٹ پٹ: ہیلو , Bob!greet(greeting="Hi") # آؤٹ پٹ: ہیلو، مہمان!
:واپسی کی قیمتیں
میں افعال ریٹرن سٹیٹمنٹ کا استعمال کرتے ہوئے قدریں واپس کر سکتے ہیں۔ آپ فنکشن سے سنگل یا ایک سے Python
زیادہ ویلیو واپس کر سکتے ہیں۔ اگر کوئی واپسی کا بیان متعین نہیں ہے تو، فنکشن ڈیفالٹ کے طور پر کوئی نہیں لوٹاتا ہے۔
ازگر
def add(x, y): واپس کریں x + y نتیجہ = add(3, 5)print(نتائج) # آؤٹ پٹ: 8
آپ متعدد اقدار کو کوما سے الگ کر کے واپس کر سکتے ہیں۔ جب ایک سے زیادہ قدریں واپس کی جاتی ہیں، تو وہ ایک
ٹیپل کے طور پر لوٹائی جاتی ہیں۔
ازگر
def square_and_cube(x): واپسی x**2, x**3 نتیجہ = مربع_اور_کیوب(3)پرنٹ(نتیجہ) # آؤٹ پٹ: (9، 27)
:آپ واپس آنے والے ٹیپل کو براہ راست متغیر میں کھول سکتے ہیں
ازگر
مربع، کیوب = مربع_اور_کیوب (3) پرنٹ (مربع) # آؤٹ پٹ: 9 پرنٹ (کیوب) # آؤٹ پٹ: 27
:جلد واپسی
اگر کوئی شرط پوری ہو جاتی ہے تو آپ کسی فنکشن سے جلد باہر نکلنے کے لیے ریٹرن اسٹیٹمنٹ استعمال کر سکتے ہیں۔
یہ غلطی سے نمٹنے یا اصلاح کے لیے مفید ہو سکتا ہے۔
ازگر
def divide(x, y): اگر y == 0: واپسی "Error: Division by zero" واپس کریں x / y نتیجہ = divide(10, 2) # آؤٹ
پٹ: 5.0
فنکشنز میں پیرامیٹرز اور ریٹرن ویلیو کے ساتھ کام کرنے کی بنیادی باتیں ہیں۔ وہ آپ کو ڈیٹا کو فنکشنز میں Python یہ
منتقل کرکے اور کال کرنے والے کو نتائج واپس کرکے لچکدار اور دوبارہ قابل استعمال کوڈ بنانے کی اجازت دیتے ہیں۔

Return statements and returning values

In Python, the `return` statement is used to exit a function and optionally return a value back to the caller. It allows functions to communicate information to the calling code by passing data back. Here's how you can use return statements and return values in Python functions:

Syntax of the Return Statement:

The general syntax of the `return` statement is:

```
python

def function_name(parameters):    # Function body    # Perform some operations
return value  # optional
```

- `def`: Keyword used to define a function.
- `function_name`: Name of the function.
- `parameters`: Input values passed to the function (optional).
- `return`: Statement used to return a value from the function (optional).

Example 1: Returning a Single Value:

You can use the `return` statement to return a single value from a function.

```
python

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

Example 2: Returning Multiple Values:

You can return multiple values from a function by separating them with commas. In Python, multiple values are returned as a tuple.

```
python

def calculate(x, y):    addition = x + y    subtraction = x - y    return addition,
subtraction result = calculate(10, 5)print(result)  # Output: (15, 5)
```

Example 3: Returning Early:

You can use the `return` statement to exit a function early if a condition is met.

```
python

def divide(x, y):    if y == 0:        return "Error: Division by zero"    return x
/ y result = divide(10, 2)  # Output: 5.0
```

Conclusion:

Return statements and returning values are essential concepts in Python functions. They allow functions to produce output and communicate results back to the caller. By understanding how to use return statements effectively, you can create functions that perform specific tasks and return meaningful results, making your code more modular and reusable.

بیانات اور واپسی قدریں واپس کریں۔
میں، واپسی کا بیان کسی فنکشن سے باہر نکلنے اور اختیاری طور پر کالر کو ایک قدر واپس کرنے کے لیے استعمال Python کیا جاتا ہے۔ یہ فنکشنز کو ڈیٹا بیک پاس کرکے کالنگ کوڈ تک معلومات پہنچانے کی اجازت دیتا ہے۔ یہاں یہ ہے کہ آپ :فنکشنز میں واپسی کے بیانات اور واپسی کی قدروں کو کس طرح استعمال کرسکتے ہیں Python
:واپسی کے بیان کا نحو
:واپسی کے بیان کا عمومی نحو یہ ہے
ازگر
فنکشن باڈی # کچھ آپریشنز انجام دیں واپسی ویلیو # اختیاری۔ # :(پیرامیٹر)def function_name
کلیدی لفظ کسی فنکشن کی وضاحت کے لیے استعمال ہوتا ہے۔ :def •
فنکشن_نام: فنکشن کا نام۔
پیرامیٹرز: ان پٹ ویلیوز فنکشن کو دی گئیں (اختیاری)۔ •
واپسی: بیان کا استعمال فنکشن (اختیاری) سے قدر واپس کرنے کے لیے ہوتا ہے۔ •
:مثال 1: ایک واحد قدر واپس کرنا
آپ کسی فنکشن سے کسی ایک ویلیو کو واپس کرنے کے لیے واپسی کا بیان استعمال کر سکتے ہیں۔
ازگر
آؤٹ پٹ: 8 # (نتائج)print(5 ,3)add = نتیجہ x + y واپس کریں :def add(x, y)
:مثال 2: متعدد قدریں واپس کرنا
آپ ایک فنکشن سے متعدد اقدار کو کوما سے الگ کر کے واپس کر سکتے ہیں۔ ازگر میں، ایک سے زیادہ قدریں بطور ٹیوپل لوٹائی جاتی ہیں۔
ازگر
واپسی کا اضافہ، گھٹاؤ نتیجہ = کیلکولیٹ(10, 5)پرنٹ(نتیجہ) # آؤٹ x - y = گھٹاؤ x + y = اضافہ :def calculate(x, y)
پٹ: (15, 5)
:مثال 3: جلد واپس آنا
اگر کوئی شرط پوری ہو جاتی ہے تو آپ کسی فنکشن سے جلد باہر نکلنے کے لیے ریٹرن اسٹیٹمنٹ استعمال کر سکتے ہیں۔
ازگر
آؤٹ # (2 ,10)divide = نتیجہ x / y واپس کریں "Error: Division by zero" واپسی :0 == y اگر :def divide(x, y)
پٹ: 5.0
:نتیجہ
واپسی کے بیانات اور واپسی قدریں ازگر کے افعال میں ضروری تصورات ہیں۔ وہ فنکشنز کو آؤٹ پٹ پیدا کرنے اور کال کرنے والے کو نتائج واپس کرنے کی اجازت دیتے ہیں۔ واپسی کے بیانات کو مؤثر طریقے سے استعمال کرنے کے طریقے کو سمجھ کر، آپ ایسے فنکشنز بنا سکتے ہیں جو مخصوص کام انجام دیتے ہیں اور بامعنی نتائج دیتے ہیں، جس سے آپ کا کوڈ مزید ماڈیولر اور دوبارہ قابل استعمال ہوتا ہے۔

Scope of variables: global vs local

In Python, the scope of a variable refers to the region of code where the variable is accessible. Understanding variable scope is crucial for writing clear and bug-free code. Python follows a set of rules to determine the scope of variables:

1. Local Scope:

Variables defined inside a function have local scope, meaning they are only accessible within that function.

```
python

def my_function():    x = 10  # Local variable    print(x) my_function()  # Output:
10print(x)       # Error: NameError: name 'x' is not defined
```

2. Enclosing Scope (Closure):

If a function is defined inside another function, the inner function has access to variables in the outer (enclosing) function's scope.

```
python

def outer_function():    y = 20    def inner_function():        print(y)  #
Accessing y from outer_function's scope    inner_function()  outer_function()  #
Output: 20
```

3. Global Scope:

Variables defined outside of any function or class have global scope, meaning they are accessible throughout the entire module.

```
python

global_var = 30  # Global variable def my_function():
print(global_var) my_function()  # Output: 30
```

4. Built-in Scope:

Python comes with a set of built-in functions and objects that are always available. These are in the built-in scope.

```
python

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

5. LEGB Rule:

Python follows the LEGB rule to determine the scope of variables:

- **Local (L)**: Inside the current function.

- **Enclosing (E)**: Inside any enclosing function (if present).
- **Global (G)**: At the top level of the module.
- **Built-in (B)**: In the built-in namespace.

Modifying Global Variables Inside Functions:

To modify a global variable from within a function, you can use the `global` keyword to indicate that you're referring to a global variable.

```
python

global_var = 30  # Global variable def modify_global():    global global_var
global_var = 40 print(global_var)  # Output: 30modify_global()print(global_var)  #
Output: 40
```

Conclusion:

Understanding variable scope in Python is essential for writing maintainable and bug-free code. By following the rules of scope, you can avoid unexpected behavior and make your code more predictable and readable.

متغیرات کا دائرہ: عالمی بمقابلہ مقامی
میں، متغیر کا دائرہ کوڈ کے اس خطے سے مراد ہے جہاں متغیر قابل رسائی ہے۔ واضح اور بگ فری کوڈ لکھنے کے Python
متغیرات کے دائرہ کار کا تعین کرنے کے لیے قواعد کے ایک سیٹ Python لیے متغیر دائرہ کار کو سمجھنا بہت ضروری ہے۔
:پر عمل کرتا ہے
:مقامی دائرہ کار .1
کسی فنکشن کے اندر متعین کردہ متغیرات کی مقامی گنجائش ہوتی ہے، یعنی وہ صرف اس فنکشن کے اندر ہی قابل رسائی
ہوتے ہیں۔
ازگر
:خرابی # print(x)آؤٹ پٹ: 10 # my_function() (x) مقامی متغیر پرنٹ # def my_function(): x = 10
کی وضاحت نہیں کی گئی ہے 'x' نام :NameError
:دائرہ کار بند کرنا .2
اگر کسی فنکشن کو کسی دوسرے فنکشن کے اندر بیان کیا جاتا ہے، تو اندرونی فنکشن کو بیرونی (انکلوزنگ) فنکشن کے
دائرہ کار میں متغیرات تک رسائی حاصل ہوتی ہے۔
ازگر
def outer_function(): y = 20 def inner_function(): print(y) # outer_function کے دائرہ inner_function()
تک رسائی y آؤٹ پٹ: 20 سے # outer_function()
:عالمی دائرہ کار .3
کسی بھی فنکشن یا کلاس سے باہر متعین کردہ متغیرات کا عالمی دائرہ کار ہے، یعنی وہ پورے ماڈیول میں قابل رسائی
ہیں۔
ازگر
آؤٹ پٹ: 30 # global_var = 30 عالمی متغیر # def my_function(): print(global_var) my_function()
:اندرونی دائرہ کار .4
بلٹ ان فنکشنز اور اشیاء کے ایک سیٹ کے ساتھ آتا ہے جو ہمیشہ دستیاب رہتے ہیں۔ یہ بلٹ ان دائرہ کار میں Python
ہیں۔
ازگر
!پرنٹ ("ہیلو، ورلڈ!") # آؤٹ پٹ: ہیلو، ورلڈ
:اصول 5. LEGB
:اصول کی پیروی کرتا ہے LEGB متغیرات کے دائرہ کار کا تعین کرنے کے لیے Python
موجودہ فنکشن کے اندر۔ :(L) مقامی •
کسی بھی انکلوژنگ فنکشن کے اندر (اگر موجود ہو)۔ :(E) انکلوزنگ
ماڈیول کے اوپری سطح پر۔ :(G) گلوبل •
بلٹ ان نیم اسپیس میں۔ :(B) بلٹ ان
:افعال کے اندر عالمی متغیرات میں ترمیم کرنا
کسی فنکشن کے اندر سے عالمی متغیر کو تبدیل کرنے کے لیے، آپ عالمی کلیدی لفظ کا استعمال اس بات کی نشاندہی
کرنے کے لیے کر سکتے ہیں کہ آپ عالمی متغیر کا حوالہ دے رہے ہیں۔
ازگر
global_var = 30 # عالمی متغیر def modify_global(): global global_var global_var = 40 print(global_var)
آؤٹ پٹ: 40 # print(global_var)modify_global()آؤٹ پٹ: 30 #
:نتیجہ
پائیتھون میں متغیر دائرہ کار کو سمجھنا قابل برقرار اور بگ فری کوڈ لکھنے کے لیے ضروری ہے۔ دائرہ کار کے اصولوں پر
عمل کر کے، آپ غیر متوقع رویے سے بچ سکتے ہیں اور اپنے کوڈ کو مزید قابل قیاس اور پڑھنے کے قابل بنا سکتے ہیں۔

Understanding objects and classes

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of classes. Python is an object-oriented programming language that supports OOP features such as classes, objects, inheritance, polymorphism, and encapsulation. Here's an introduction to objects and classes in Python:

Classes:

A class is a blueprint for creating objects. It defines the attributes (data) and methods (functions) that the objects will have. Classes are used to encapsulate related data and behavior into a single unit.

Syntax:

```
python

class ClassName:    # Attributes and methods are defined here    pass
```

Example:

```
python

class Person:    def __init__(self, name, age):        self.name = name
self.age = age     def greet(self):        return f"Hello, my name is {self.name}
and I am {self.age} years old." # Creating an object (instance) of the Person
classperson1 = Person("Alice", 30) # Accessing attributes and calling
methodsprint(person1.name)    # Output: Aliceprint(person1.age)     # Output:
30print(person1.greet()) # Output: Hello, my name is Alice and I am 30 years old.
```

Objects:

An object is an instance of a class. It represents a specific instance of the class, with its own unique attributes and methods. Objects are created using the class constructor.

Example:

```
python

# Creating multiple instances (objects) of the Person classperson1 =
Person("Alice", 30)person2 = Person("Bob", 25) # Accessing attributes and calling
methods for each objectprint(person1.name)    # Output: Aliceprint(person2.name)
# Output: Bobprint(person1.greet()) # Output: Hello, my name is Alice and I am 30
years old.print(person2.greet()) # Output: Hello, my name is Bob and I am 25 years
old.
```

Understanding Attributes and Methods:

- **Attributes**: Attributes are variables that belong to objects. They store data that defines the object's state.

- **Methods**: Methods are functions that belong to objects. They define the behavior of the object and can operate on the object's attributes.

Conclusion:

In Python, classes are used to create objects, which encapsulate related data and behavior. Understanding how to define classes, create objects, access attributes, and call methods is fundamental to working with object-oriented programming in Python.

اشیاء اور طبقات کو سمجھنا
ایک پروگرامنگ پیراڈائم ہے جو "آبجیکٹ" کے تصور پر مبنی ہے جو کلاسز کی مثالیں (OOP) آبجیکٹ اورینٹڈ پروگرامنگ خصوصیات جیسے کہ کلاسز، آبجیکٹ، وراثت، OOP ایک آبجیکٹ پر مبنی پروگرامنگ زبان ہے جو Python ہیں۔
:میں اشیاء اور کلاسوں کا تعارف یہ ہے Python پولیمورفزم، اور انکیپسولیشن کو سپورٹ کرتی ہے۔
:کلاسز
ایک کلاس اشیاء بنانے کے لیے ایک بلیو پرنٹ ہے۔ یہ ان صفات (ڈیٹا) اور طریقوں (فنکشنز) کی وضاحت کرتا ہے جو اشیاء کے پاس ہوں گے۔ کلاسز کا استعمال متعلقہ ڈیٹا اور رویے کو ایک اکائی میں سمیٹنے کے لیے کیا جاتا ہے۔
:نحو
ازگر
صفات اور طریقے یہاں بیان کیے گئے ہیں پاس # :ClassName #
:مثال
ازگر
ہیلو،"f واپسی :(def __init__(self, name, age): self.name = name self.age = age def greet(self: کلاس شخص = classperson1 ہوں سالوں کا." # فرد کی ایک چیز (مثال) بنانا {self.age} ہے اور میں {self.name} میرا نام آؤٹ پٹ: ایلیس # (person1.name) خصوصیات تک رسائی اور کال کرنے کے طریقے پرنٹ # Person("Alice", 30)
آؤٹ پٹ: ہیلو، میرا نام ایلس ہے اور میری عمر 30 سال # (()print(person1.greetآؤٹ پٹ: 30 # (person1.age)پرنٹ ہے۔
:اشیاء
ایک شے کلاس کی ایک مثال ہے۔ یہ کلاس کی ایک مخصوص مثال کی نمائندگی کرتا ہے، اس کی اپنی منفرد صفات اور طریقوں کے ساتھ۔ کلاس کنسٹرکٹر کا استعمال کرتے ہوئے آبجیکٹ بنائے جاتے ہیں۔
:مثال
ازگر
فرد کی ایک سے زیادہ مثالیں (آبجیکٹ) بنانا کلاس پرسن1 = شخص("ایلس"، 30)پرسن2 = شخص("باب"، 25) # ہر # کے لیے صفات تک رسائی اور کال کرنے کے طریقے # آؤٹ پٹ: ایلیس (person1.name) آبجیکٹ پرنٹ
آؤٹ پٹ: ہیلو، میرا نام ایلس ہے اور میری عمر 30 # (()print(person1.greet :Bobآؤٹ پٹ # (person2) .name)پرنٹ
آؤٹ پٹ: ہیلو، میرا نام باب ہے اور میں ہوں 25 سال کی عمر۔ # (()person2.greetسال ہے۔ پرنٹ
:صفات اور طریقوں کو سمجھنا
• صفات: اوصاف متغیرات ہیں جو اشیاء سے تعلق رکھتے ہیں۔ وہ ڈیٹا کو ذخیرہ کرتے ہیں جو آبجیکٹ کی حالت کی وضاحت کرتا ہے۔
• طریقے: طریقے وہ افعال ہیں جو اشیاء سے تعلق رکھتے ہیں۔ وہ آبجیکٹ کے رویے کی وضاحت کرتے ہیں اور آبجیکٹ کی صفات پر کام کر سکتے ہیں۔
:نتیجہ
میں، کلاسیں اشیاء بنانے کے لیے استعمال کی جاتی ہیں، جو متعلقہ ڈیٹا اور رویے کو سمیٹتی ہیں۔ یہ سمجھنا کہ Python کلاسوں کی وضاحت کیسے کی جاتی ہے، اشیاء کی تخلیق کیسے کی جاتی ہے، صفات تک رسائی حاصل کی جاتی ہے، اور میں آبجیکٹ پر مبنی پروگرامنگ کے ساتھ کام کرنے کے لیے بنیادی ہیں۔ Python کال کے طریقے

Class attributes and methods

In Python, classes are blueprints for creating objects. They can contain both attributes (variables) and methods (functions). Class attributes are variables that are shared by all instances of the class, while class methods are functions that are associated with the class rather than instances of the class. Let's explore class attributes and methods in Python:

Class Attributes:

Class attributes are defined inside the class block but outside any method. They are shared by all instances of the class.

Example:

```
python

class MyClass:    class_attribute = "I am a class attribute" # Accessing class attribute using class name or instanceprint(MyClass.class_attribute)  # Output: I am a class attribute obj1 = MyClass()print(obj1.class_attribute)     # Output: I am a class attribute obj2 = MyClass()print(obj2.class_attribute)     # Output: I am a class attribute
```

Class Methods:

Class methods are defined using the `@classmethod` decorator. They take the class itself (usually named `cls`) as the first parameter, allowing them to access or modify class attributes.

Example:

```
python

class MyClass:    class_attribute = "I am a class attribute"     @classmethod def class_method(cls):        print("Class method called")        print("Accessing class attribute:", cls.class_attribute) # Calling class method using class name or instanceMyClass.class_method()# Output:# Class method called# Accessing class attribute: I am a class attribute obj = MyClass()obj.class_method()# Output:# Class method called# Accessing class attribute: I am a class attribute
```

Conclusion:

Class attributes and methods are essential features of object-oriented programming in Python. Class attributes are shared among all instances of the class, while class methods can access and manipulate these attributes. Understanding how to use class attributes and methods allows you to create more flexible and organized code structures.

Instance attributes and methods

In Python, instance attributes and methods are associated with individual instances of a class. Instance attributes are variables specific to each instance, while instance methods are functions that can access and manipulate these instance attributes. Let's delve into instance attributes and methods:

Instance Attributes:

Instance attributes are defined inside the constructor method `__init__()` using the `self` keyword. They are unique to each instance of the class.

Example:

```
python

class MyClass:    def __init__(self, name):        self.name = name  # Instance attribute # Creating instances and accessing instance attributesobj1 = MyClass("Object 1")print(obj1.name)  # Output: Object 1 obj2 = MyClass("Object 2")print(obj2.name)  # Output: Object 2
```

Instance Methods:

Instance methods are functions defined within a class that operate on instance attributes. They take `self` as the first parameter, which refers to the instance itself.

Example:

```
python

class MyClass:    def __init__(self, name):        self.name = name  # Instance attribute     def display_name(self):        print("Name:", self.name) # Creating instances and calling instance methodsobj1 = MyClass("Object 1")obj1.display_name() # Output: Name: Object 1 obj2 = MyClass("Object 2")obj2.display_name()  # Output: Name: Object 2
```

Modifying Instance Attributes:

Instance attributes can be modified directly using dot notation or within instance methods.

Example:

```
python

class MyClass:    def __init__(self, name):        self.name = name  # Instance attribute     def change_name(self, new_name):        self.name = new_name # Modifying instance attributesobj = MyClass("Old Name")print(obj.name)  # Output: Old Name obj.change_name("New Name")print(obj.name)  # Output: New Name
```

Conclusion:

Instance attributes and methods are fundamental concepts in object-oriented programming. Instance attributes store unique data for each instance, while instance methods operate on this data. Understanding how to define and use instance attributes and methods allows you to create classes that encapsulate data and behavior effectively.

طبقاتی صفات اور طریقے
میں، کلاسیں اشیاء بنانے کے لیے بلیو پرنٹس ہیں۔ ان میں صفات (متغیر) اور طریقے (فنکشن) دونوں شامل ہو Python سکتے ہیں۔ کلاس انتساب متغیرات ہیں جو کلاس کی تمام مثالوں کے ذریعہ مشترکہ ہیں، جبکہ کلاس کے طریقے ایسے میں کلاس کے اوصاف اور طریقوں کو Python فنکشن ہیں جو کلاس کی مثالوں کے بجائے کلاس سے وابستہ ہیں۔ آئیے :دریافت کریں
:کلاس کی خصوصیات
کلاس کی خصوصیات کلاس بلاک کے اندر لیکن کسی بھی طریقہ سے باہر بیان کی جاتی ہیں۔ وہ کلاس کے تمام واقعات کے ذریعہ مشترکہ ہیں۔
:مثال
ازگر
میں ایک کلاس انتساب ہوں" # کلاس کے نام یا مثال کے پرنٹ" = MyClass: class_attribute کلاس
کا استعمال کرتے ہوئے کلاس انتساب تک رسائی حاصل کرنا # آؤٹ پٹ: میں ایک کلاس (MyClass.class_attribute)
= a class attribute obj2 آؤٹ پٹ: میں ہوں # (obj1.class_attribute)print(MyClass() = obj1 وصف ہوں
آؤٹ پٹ: میں ایک کلاس وصف ہوں # (obj2.class_attribute)print(MyClass()
:کلاس کے طریقے
ڈیکوریٹر کا استعمال کرتے ہوئے کی گئی ہے۔ وہ کلاس کو ہی پہلے پیرامیٹر @classmethod کلاس کے طریقوں کی وضاحت لیتے ہیں، جس سے وہ کلاس کی صفات تک رسائی یا ترمیم کرسکتے ہیں۔ (کا نام دیا جاتا ہے cls عام طور پر) کے طور پر
:مثال
ازگر
کلاس MyClass: class_attribute = "میں ایک کلاس وصف ہوں" @classmethod def class_method(cls):
کلاس کا نام یا مثال استعمال کرتے #(print("class attribute:", cls.class_attribute) print("Class method called")
آؤٹ پٹ:# کلاس کا طریقہ کہلاتا ہے# کلاس کی خصوصیت: میں ایک #()class_method. ہوئے کلاس کا طریقہ کال کرنا
آؤٹ پٹ:# کلاس میتھڈ جس کو کہا جاتا ہے# تک رسائی #()obj.class_method() = obj کلاس انتساب ہوں
کلاس انتساب: میں ایک کلاس وصف ہوں
:نتیجہ
میں آبجیکٹ پر مبنی پروگرامنگ کی ضروری خصوصیات ہیں۔ کلاس کی تمام مثالوں میں Python کلاس اوصاف اور طریقے کلاس کی خصوصیات کا اشتراک کیا جاتا ہے، جبکہ کلاس کے طریقے ان صفات تک رسائی اور جوڑ توڑ کر سکتے ہیں۔ طبقاتی صفات اور طریقوں کو استعمال کرنے کے طریقے کو سمجھنا آپ کو زیادہ لچکدار اور منظم کوڈ ڈھانچے بنانے کی اجازت دیتا ہے۔

مثال کے اوصاف اور طریقے
میں، مثال کے اوصاف اور طریقے کلاس کی انفرادی مثالوں سے وابستہ ہیں۔ مثال کے اوصاف ہر مثال کے لیے Python مخصوص متغیر ہوتے ہیں، جب کہ مثال کے طریقے ایسے فنکشن ہوتے ہیں جو ان مثال کے اوصاف تک رسائی اور جوڑ توڑ کر سکتے ہیں۔ آئیے مثال کے اوصاف اور طریقوں پر غور کریں:
:مثال کے اوصاف
کے اندر خود کلیدی لفظ کا استعمال کرتے ہوئے بیان کیا گیا ہے۔ وہ کلاس ()__init__ مثال کے اوصاف کو کنسٹرکٹر طریقہ کی ہر مثال کے لیے منفرد ہیں۔
:مثال
ازگر
مثال کی خصوصیت # مثالیں بنانا اور انسٹانس تک # MyClass: def __init__(self, name): self.name = name کلاس
= obj2 آؤٹ پٹ: آبجیکٹ 1 # (obj1.name)print(MyClass("Object 1") = obj1 attributesobj رسائی حاصل کرنا
آؤٹ پٹ: آبجیکٹ 2 # (obj2.name)پرنٹ ("آبجیکٹ 2 ")MyClass(
:مثال کے طریقے
مثال کے طریقے ایک کلاس کے اندر بیان کردہ افعال ہیں جو مثال کے اوصاف پر کام کرتے ہیں۔ وہ خود کو پہلے پیرامیٹر کے طور پر لیتے ہیں، جس سے مراد خود مثال ہے۔
:مثال
ازگر

کلاس MyClass: def __init__(self, name): self.name = name # instance attribute def display_name(self): print("Name:", self.name) # مثالیں بنانا اور مثال کے طور پر کال کرنا obj1.display_name()(آؤٹ پٹ: نام:
آؤٹ پٹ: نام: آبجیکٹ 2 obj2 = MyClass("Object 2")obj2.display_name() # آبجیکٹ 1
مثال کے اوصاف میں ترمیم کرنا:
مثال کے اوصاف کو براہ راست ڈاٹ اشارے کا استعمال کرتے ہوئے یا مثال کے طریقوں میں تبدیل کیا جاسکتا ہے۔
مثال:
ازگر
کلاس MyClass: def __init__(self, name): self.name = name # مثال کی خصوصیت def change_name(self, new_name): self.name = new_name # مثال کے طور پر ترمیم کرنا attributesobj = MyClass("Old Name")print(obj.name) # پرانا نام آؤٹ پٹ: obj.change_name("نیا نام") پرنٹ (obj.name) # نیا نام آؤٹ پٹ:
نتیجہ:
مثال کے اوصاف اور طریقے آبجیکٹ پر مبنی پروگرامنگ میں بنیادی تصورات ہیں۔ مثال کے اوصاف ہر مثال کے لیے منفرد ڈیٹا اسٹور کرتے ہیں، جبکہ مثال کے طریقے اس ڈیٹا پر کام کرتے ہیں۔ مثال کے اوصاف اور طریقوں کی وضاحت اور استعمال کے طریقہ کو سمجھنا آپ کو ایسی کلاسیں بنانے کی اجازت دیتا ہے جو ڈیٹا اور رویے کو مؤثر طریقے سے سمیٹتے ہیں۔

Inheritance and polymorphism basics

In Python, inheritance and polymorphism are core concepts of object-oriented programming (OOP). They allow you to create classes that inherit attributes and methods from other classes and enable code to work with objects of different types interchangeably. Let's explore the basics of inheritance and polymorphism in Python:

Inheritance:

Inheritance is the process by which a new class (subclass) is created from an existing class (superclass). The subclass inherits attributes and methods from its superclass, allowing for code reuse and hierarchical organization.

Syntax:

```
python

class BaseClass:    # Base class attributes and methods class SubClass(BaseClass):
# Subclass attributes and methods
```

Example:

```
python

class Animal:    def speak(self):        print("Animal speaks") class Dog(Animal):
def bark(self):        print("Dog barks") # Dog class inherits from Animal class
```

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables code to work with objects of various types without knowing their specific class.

Method Overriding:

Polymorphism is often achieved through method overriding, where a method in a subclass has the same name as a method in its superclass. The method in the subclass overrides the behavior of the superclass method.

```
python

class Animal:    def speak(self):        print("Animal speaks") class Dog(Animal):
def speak(self):        print("Dog barks") # Method speak() in Dog overrides the
speak() method in Animal
```

Example of Polymorphic Behavior:

```
python
```

```
def make_speak(animal):    animal.speak() # Polymorphic behavior: make_speak can accept different types of animalsmake_speak(Animal())  # Output: "Animal speaks"make_speak(Dog())     # Output: "Dog barks"
```

Conclusion:

Inheritance and polymorphism are powerful features of Python's object-oriented programming paradigm. They allow for code reuse, modularity, and flexibility in designing and implementing software systems. By understanding how to use inheritance and polymorphism, you can create more modular, maintainable, and extensible code.

وراثت اور پولیمورفزم کی بنیادی باتیں
کے بنیادی تصورات ہیں۔ وہ آپ کو ایسی کلاسیں بنانے (OOP) ازگر میں، وراثت اور پولیمورفزم آبجیکٹ اورینٹڈ پروگرامنگ کی اجازت دیتے ہیں جو دوسری کلاسوں سے وراثت میں اوصاف اور طریقے حاصل کرتے ہیں اور کوڈ کو مختلف اقسام کی اشیاء کے ساتھ ایک دوسرے کے ساتھ کام کرنے کے قابل بناتے ہیں۔ آئیے ازگر میں وراثت اور پولیمورفزم کی بنیادی باتیں دریافت کریں:

وراثت:
وراثت وہ عمل ہے جس کے ذریعے موجودہ کلاس (سپر کلاس) سے ایک نئی کلاس (سب کلاس) بنائی جاتی ہے۔ ذیلی طبقے کو اپنے سپر کلاس سے صفات اور طریقے وراثت میں ملتے ہیں، جو کوڈ کو دوبارہ استعمال کرنے اور درجہ بندی کی تنظیم کی اجازت دیتا ہے۔

نحو:
ازگر
ذیلی کلاس کی خصوصیات اور # :(SubClass(BaseClass کلاس بیس کلاس: # بیس کلاس کی خصوصیات اور طریقے کلاس طریقے

مثال:
ازگر
کلاس اینیمل: ڈیف اسپیک(سیلف): پرنٹ ("جانور بولتا ہے") کلاس ڈاگ (جانور): ڈیف بارک (خود): پرنٹ ("کتے کی چھال") # کتے کی کلاس جانوروں کی کلاس سے وراثت میں ملتی ہے

پولیمورفزم:
پولیمورفزم مختلف طبقات کی اشیاء کو ایک مشترکہ سپر کلاس کی اشیاء کے طور پر علاج کرنے کی اجازت دیتا ہے۔ یہ کوڈ کو ان کی مخصوص کلاس کو جانے بغیر مختلف اقسام کی اشیاء کے ساتھ کام کرنے کے قابل بناتا ہے۔

طریقہ اوور رائیڈنگ:
پولیمورفزم اکثر طریقہ اوور رائیڈنگ کے ذریعے حاصل کیا جاتا ہے، جہاں ذیلی طبقے میں ایک طریقہ وہی نام رکھتا ہے جو اس کے سپر کلاس میں ایک طریقہ ہوتا ہے۔ ذیلی طبقے میں طریقہ سپر کلاس طریقہ کے رویے کو اوور رائیڈ کرتا ہے۔
ازگر
کلاس Animal: def speak(self): print("Animal speaks") class Dog(Animal): def speak(self): print("dog barks") # Method speak() in Dog Animal میں speak() طریقہ کو اوور رائیڈ کرتا ہے

پولیمورفک سلوک کی مثال:
ازگر
def make_speak(animal): animal.speak() # پولیمورفک برتاؤ: make_speak جانوروں کی مختلف اقسام کو قبول کر سکتا ہے make_speak(Animal()) # جانور بولتا ہے: "آؤٹ پٹ make_speak(Dog()) # کتے کے بھونکتے: "آؤٹ پٹ
ہیں"

نتیجہ:
وراثت اور پولیمورفزم ازگر کے آبجیکٹ پر مبنی پروگرامنگ پیراڈائم کی طاقتور خصوصیات ہیں۔ وہ کوڈ کے دوبارہ استعمال، ماڈیولریٹی، اور سافٹ ویئر سسٹمز کو ڈیزائن کرنے اور لاگو کرنے میں لچک کی اجازت دیتے ہیں۔ وراثت اور پولیمورفزم کو استعمال کرنے کے طریقہ کو سمجھ کر، آپ مزید ماڈیولر، برقرار رکھنے کے قابل، اور قابل توسیع کوڈ بنا سکتے ہیں۔

Understanding modules and importing them

In Python, a module is a file containing Python code, typically containing functions, classes, and variables, that can be imported and used in other Python scripts or modules. Here's an overview of modules and how to import them:

Creating a Module:

To create a module, simply create a Python file with a `.py` extension and define your functions, classes, and variables in it.

Example (my_module.py):

```
python

# Define a functiondef greet(name):    return f"Hello, {name}!" # Define a
variablemy_variable = 123
```

Importing Modules:

You can import modules into other Python scripts or modules using the `import` statement.

Example:

```
python

# Import the entire moduleimport my_module # Use the functions and variables
defined in the moduleprint(my_module.greet("Alice"))  # Output: Hello, Alice!
print(my_module.my_variable)      # Output: 123
```

Importing Specific Items:

You can also import specific functions, classes, or variables from a module using the `from` keyword.

Example:

```
python

# Import specific items from the modulefrom my_module import greet, my_variable #
Use the imported items directly without module prefixprint(greet("Bob"))     #
Output: Hello, Bob!print(my_variable)       # Output: 123
```

Aliasing Modules:

You can alias modules or items from modules using the `as` keyword.

Example:

```
python
```

```
# Import module with an aliasimport my_module as mm # Use the alias to access
module itemsprint(mm.greet("Charlie"))  # Output: Hello, Charlie!
```

Importing All Items:

You can import all items from a module using the `*` wildcard.

Example:

```
python

# Import all items from the modulefrom my_module import * # Use imported items
directly without module prefixprint(greet("David"))      # Output: Hello, David!
print(my_variable)         # Output: 123
```

Module Search Path:

When you import a module, Python searches for it in directories listed in the `sys.path` variable. By default, it includes the current directory and the directories specified in the `PYTHONPATH` environment variable.

Conclusion:

Understanding modules and how to import them is fundamental to organizing and reusing code in Python. By encapsulating related functionality in modules and importing them into your scripts, you can write cleaner and more modular code.

ماڈیولز کو سمجھنا اور انہیں درآمد کرنا
کوڈ ہوتا ہے، عام طور پر فنکشنز، کلاسز اور متغیرات پر مشتمل Python میں، ایک ماڈیول ایک فائل ہے جس میں Python اسکرپٹس یا ماڈیولز میں درآمد اور استعمال کیا جا سکتا ہے۔ یہاں ماڈیولز کا ایک جائزہ اور Python ہوتا ہے، جسے دیگر انہیں درآمد کرنے کا طریقہ ہے:
ایک ماڈیول بنانا:
فائل بنائیں اور اس میں اپنے فنکشنز، کلاسز اور Python ایکسٹینشن کے ساتھ ایک py. ایک ماڈیول بنانے کے لیے، صرف متغیرات کی وضاحت کریں۔
مثال (my_module.py):
ازگر
متغیر کی_متغیر کی وضاحت کریں = 123 # f"Hello, {name}!" فنکشن ڈیف گریٹ (نام) کی وضاحت کریں: واپسی #
ماڈیولز درآمد کرنا:
آپ امپورٹ سٹیٹمنٹ کا استعمال کرتے ہوئے ماڈیولز کو دیگر ازگر اسکرپٹس یا ماڈیولز میں درآمد کر سکتے ہیں۔
مثال:
ازگر
میں بیان کردہ (my_module.greet("Alice"))ماڈیول پرنٹ # import my_module پورے ماڈیول کو درآمد کریں #
آؤٹ پٹ: 123 # (print(my_module.my_variable!فنکشنز اور متغیرات کا استعمال کریں # آؤٹ پٹ: ہیلو، ایلس
مخصوص اشیاء کی درآمد:
کی ورڈ کا استعمال کرتے ہوئے ماڈیول سے مخصوص فنکشنز، کلاسز یا متغیرات بھی درآمد کر سکتے ہیں۔ from آپ
مثال:
ازگر
سے # امپورٹڈ آئٹمز کو ماڈیول my_module import greet, my_variable ماڈیول سے مخصوص آئٹمز درآمد کریں #
آؤٹ # (print(my_variable!Hello, Bob :آؤٹ پٹ # (greet("Bob")) پریفکس پرنٹ کے بغیر براہ‌راست استعمال کریں
پٹ: 123
عرفی ماڈیولز:
آپ بطور کلیدی لفظ استعمال کرتے ہوئے ماڈیولز سے ماڈیولز یا آئٹمز کو عرفی نام دے سکتے ہیں۔
مثال:
ازگر
کے طور پر # ماڈیول آئٹمز پرنٹ تک رسائی کے لیے عرف کا mm کو aliasimport my_module ماڈیول امپورٹ کریں #
!آؤٹ پٹ: ہیلو، چارلی # (mm.greet("Charlie")) استعمال کریں
تمام اشیاء درآمد کرنا:
آپ * وائلڈ کارڈ کا استعمال کرتے ہوئے ماڈیول سے تمام اشیاء درآمد کرسکتے ہیں۔
مثال:
ازگر
امپورٹ سے * # ماڈیول پریفکس پرنٹ کے بغیر درآمد شدہ آئٹمز کو براہ my_module ماڈیول سے تمام آئٹمز درآمد کریں #
آؤٹ پٹ: 123 # (print(my_variable) :آؤٹ پٹ # (greet("David"))است استعمال کریں
ماڈیول تلاش کاراستہ:
متغیر میں درج ڈائریکٹریوں میں تلاش کرتا ہے۔ پہلے سے طے sys.path اسے Python جب آپ ماڈیول درآمد کرتے ہیں، تو ماحولیاتی متغیر میں مخصوص ڈائریکٹریز شامل ہیں۔ PYTHONPATH شدہ طور پر، اس میں موجودہ ڈائرکٹری اور
نتیجہ:
میں کوڈ کو ترتیب دینے اور دوبارہ استعمال کرنے کے لیے بنیادی Python ماڈیولز کو سمجھنا اور انہیں درآمد کرنے کا طریقہ ہے۔ ماڈیولز میں متعلقہ فعالیت کو سمیٹ کر اور انہیں اپنی اسکرپٹ میں درآمد کرکے، آپ صاف ستھرا اور زیادہ ماڈیولر کوڈ لکھ سکتے ہیں۔

Creating and using packages

In Python, a package is a directory that contains one or more modules and an optional special file named `__init__.py`. Packages are used to organize and distribute Python code into reusable and modular components. Here's a step-by-step guide on how to create and use packages in Python:

Creating a Package:

1. **Create a Directory**: Create a directory to serve as the root directory of your package.
2. **Add Modules**: Inside the package directory, add one or more Python files (modules) containing your code. Each module should represent a logical component of your package.
3. **Optional: `__init__.py` File**: You can include an empty file named `__init__.py` (or with some initialization code if needed) inside the package directory. This file is necessary to treat the directory as a package.

Here's an example directory structure for a simple package named `my_package`:

```
markdown

my_package/|├── __init__.py├── module1.py└── module2.py
```

Using the Package:

Once you've created your package, you can use it in your Python code by importing modules from the package.

Example:

Suppose `module1.py` contains the following code:

```
python

# module1.pydef greet():    print("Hello from module1")
```

And `module2.py` contains:

```
python

# module2.pydef farewell():    print("Goodbye from module2")
```

You can use these modules in your Python code as follows:

```
python

# main.pyfrom my_package import module1, module2 module1.greet()   # Output: Hello
from module1module2.farewell()  # Output: Goodbye from module2
```

Installing the Package:

If you want to distribute your package for others to use, you can create a distribution package using tools like `setuptools` and `pip`. This involves creating a `setup.py` file to define metadata about your package and using `setuptools` to create a distribution package.

Conclusion:

Creating and using packages in Python allows you to organize and distribute your code into reusable and modular components, promoting code reusability and maintainability. By following the steps outlined above, you can create your own packages and leverage them in your Python projects.

پیکجز بنانا اور استعمال کرنا
نامی ایک اختیاری خصوصی __init__.py میں، ایک پیکیج ایک ڈائریکٹری ہے جس میں ایک یا زیادہ ماڈیولز اور Python کوڈ کو دوبارہ قابل استعمال اور ماڈیولر اجزاء میں ترتیب دینے اور تقسیم کرنے کے لیے Python فائل ہوتی ہے۔ پیکجز میں پیکیجز بنانے اور استعمال کرنے کے طریقہ کے بارے میں یہاں ایک مرحلہ وار گائیڈ Python استعمال کیے جاتے ہیں۔ ہے:
ایک پیکج بنانا:
1. ایک ڈائرکٹری بنائیں: اپنے پیکیج کی روٹ ڈائرکٹری کے طور پر کام کرنے کے لیے ایک ڈائرکٹری بنائیں۔
2. فائلیں (ماڈیولز) شامل کریں۔ Python ماڈیولز شامل کریں: پیکیج ڈائرکٹری کے اندر، اپنے کوڈ پر مشتمل ایک یا زیادہ ہر ماڈیول کو آپ کے پیکیج کے منطقی جزو کی نمائندگی کرنی چاہیے۔
3. اختیاری: __init__.py فائل: آپ پیکج ڈائرکٹری کے اندر __init__.py نام کی ایک (یا کچھ ابتدائی کوڈ کے ساتھ) خالی فائل شامل کر سکتے ہیں۔ یہ فائل ڈائریکٹری کو پیکج کے طور پر علاج کرنے کے لیے ضروری ہے۔
نامی ایک سادہ پیکیج کے لیے ڈائرکٹری ڈھانچہ کی ایک مثال یہ ہے۔ my_package
نشان لگانا
my_package/ │ ├── __init__.py ├── module1.py └── module2.py
پیکیج کا استعمال کرتے ہوئے:
کوڈ میں پیکیج سے ماڈیول درآمد کر کے استعمال کر Python ایک بار جب آپ اپنا پیکج بنا لیتے ہیں، تو آپ اسے اپنے سکتے ہیں۔
مثال:
درج ذیل کوڈ پر مشتمل ہے module1.py فرض کریں کہ:
ازگر
module1.pydef greet(): print("Hello from module1")
پر مشتمل ہے module2.py اور:
ازگر
module2.pydef farewell(): print("module2 سے الوداع")
کوڈ میں درج ذیل استعمال کر سکتے ہیں Python آپ ان ماڈیولز کو اپنے:
ازگر
main.pyfrom my_package import module1, module2 module1.greet() # آؤٹ پٹ: Hello from module1module2.farewell() # آؤٹ پٹ: module2 سے الوداع
پیکیج کی تنصیب:
اگر آپ اپنے پیکج کو دوسروں کے استعمال کے لیے تقسیم کرنا چاہتے ہیں، تو آپ سیٹ اپ ٹولز اور پائپ جیسے ٹولز کا استعمال کرکے ڈسٹری بیوشن پیکج بنا سکتے ہیں۔ اس میں آپ کے پیکج کے بارے میں میٹا ڈیٹا کی وضاحت کے لیے ایک فائل بنانا اور ڈسٹری بیوشن پیکج بنانے کے لیے سیٹ اپ ٹولز کا استعمال شامل ہے۔ setup.py
نتیجہ:
میں پیکجز بنانا اور استعمال کرنا آپ کو اپنے کوڈ کو دوبارہ قابل استعمال اور ماڈیولر اجزاء میں ترتیب دینے اور Python تقسیم کرنے کی اجازت دیتا ہے، کوڈ کو دوبارہ استعمال کرنے اور برقرار رکھنے کی صلاحیت کو فروغ دیتا ہے۔ اوپر بیان کیے گئے اقدامات پر عمل کرتے ہوئے، آپ خود اپنے پیکجز بنا سکتے ہیں اور اپنے ازگر کے پروجیکٹس میں ان کا فائدہ اٹھا سکتے ہیں۔

Exploring the Python Standard Library

Exploring the Python Standard Library can be a rich and rewarding experience for Python developers. The Python Standard Library is a collection of modules and packages that provide a wide range of functionality for various tasks, ranging from file I/O and networking to data manipulation and web development. Here's an overview of some key modules and packages in the Python Standard Library:

1. **os**: Provides functions for interacting with the operating system, including file and directory manipulation, environment variables, and process management.
2. **sys**: Contains functions and variables related to the Python interpreter and system-specific parameters and functions.
3. **datetime**: Offers classes for working with dates, times, and time deltas, allowing for parsing, formatting, and arithmetic operations on dates and times.
4. **json**: Enables encoding and decoding JSON data, facilitating interoperability between Python and other programming languages and systems.
5. **math**: Provides mathematical functions and constants for performing various mathematical operations, such as trigonometry, logarithms, and rounding.
6. **random**: Offers functions for generating pseudo-random numbers and performing random selections and shuffling.
7. **re**: Supports regular expressions for pattern matching and text manipulation, allowing for powerful string processing capabilities.
8. **collections**: Contains specialized container datatypes beyond the built-in data structures like lists, tuples, and dictionaries, including namedtuple, deque, Counter, and defaultdict.
9. **urllib**: Facilitates working with URLs, allowing for sending HTTP requests, handling responses, and parsing URLs.
10. **socket**: Provides low-level networking interfaces for creating network sockets, sending and receiving data over networks, and building network servers and clients.
11. **argparse**: Simplifies the process of parsing command-line arguments and options, making it easy to create command-line interfaces for Python programs.
12. **logging**: Offers a flexible and powerful logging framework for generating log messages at various severity levels, configuring logging behavior, and routing log messages to different destinations.

These are just a few examples of the many modules and packages available in the Python Standard Library. Exploring the Standard Library documentation and experimenting with

different modules and functions can help you discover the breadth and depth of functionality provided by Python's built-in tools, saving you time and effort in your development projects.

Importing modules and packages

In Python, modules and packages provide a way to organize and reuse code. Modules are individual Python files containing functions, classes, and variables, while packages are directories containing multiple modules. Let's explore how to import modules and packages in Python:

Importing Modules:

You can import modules using the `import` statement followed by the module name. Once imported, you can access functions, classes, and variables defined in the module using dot notation (`.`).

Syntax:

```
python

import module_name
```

Example:

```
python

import math print(math.pi)  # Accessing variable 'pi' from math
moduleprint(math.sqrt(25))  # Accessing function 'sqrt' from math module
```

Importing with Alias:

You can import a module with an alias to provide a shorter or more convenient name.

Syntax:

```
python

import module_name as alias
```

Example:

```
python

import math as m print(m.pi)  # Accessing variable 'pi' from math module using
alias 'm'
```

Importing Specific Items:

You can import specific functions, classes, or variables from a module instead of importing the entire module.

Syntax:

```
python

from module_name import item_name1, item_name2, ...
```

Example:

```
python

from math import pi, sqrt print(pi)        # Accessing variable 'pi'
directlyprint(sqrt(25))  # Accessing function 'sqrt' directly
```

Importing Everything:

You can import all items from a module into the current namespace, but this is generally discouraged as it can lead to namespace pollution and conflicts.

Syntax:

```
python

from module_name import *
```

Example:

```
python

from math import * print(pi)        # Accessing variable 'pi'
directlyprint(sqrt(25))  # Accessing function 'sqrt' directly
```

Importing Packages:

Packages are directories containing multiple modules. You can import modules from packages using dot notation (`.`).

Syntax:

```
python

import package_name.module_name
```

Example:

```
python

import urllib.request response =
urllib.request.urlopen('https://www.example.com')html = response.read()
```

Conclusion:

Importing modules and packages allows you to reuse code and organize your Python projects effectively. By understanding how to import modules, aliasing, importing specific items, and importing from packages, you can write cleaner, more modular, and more maintainable Python code.

ازگر کی معیاری لائبریری کی تلاش
کے ڈویلپرز کے لیے ایک بھرپور اور فائدہ مند تجربہ ہو سکتا ہے۔ Python اسٹینڈرڈ لائبریری کو دریافت کرنا Python اور نیٹ ورکنگ سے لے کر ڈیٹا ہیرا پھیری I/O ماڈیولز اور پیکجز کا ایک مجموعہ ہے جو فائل Python Standard Library اسٹینڈرڈ لائبریری میں Python اور ویب ڈویلپمنٹ تک مختلف کاموں کے لیے فعالیت کی ایک وسیع رینج فراہم کرتا ہے۔
:کچھ کلیدی ماڈیولز اور پیکجز کا ایک جائزہ یہ ہے
1. os: آپریٹنگ سسٹم کے ساتھ بات چیت کے لیے افعال فراہم کرتا ہے، بشمول فائل اور ڈائرکٹری میں ہیرا پھیری، ماحولیاتی متغیرات، اور عمل کا انتظام۔
2. sys: Python انٹرپریٹر اور سسٹم کے مخصوص پیرامیٹرز اور فنکشنز سے متعلق افعال اور متغیرات پر مشتمل ہے۔
3. تاریخ کا وقت: تاریخوں، اوقات اور وقت کے ڈیلٹا کے ساتھ کام کرنے کے لیے کلاسز پیش کرتا ہے، تاریخوں اور اوقات پر تجزیہ، فارمیٹنگ، اور ریاضی کی کارروائیوں کی اجازت دیتا ہے۔
4. json: JSON ڈیٹا کو انکوڈنگ اور ڈی کوڈنگ کو قابل بناتا ہے Python اور دیگر پروگرامنگ زبانوں اور سسٹمز کے درمیان انٹرآپریبلٹی کو سہولت فراہم کرتا ہے۔
5. ریاضی: ریاضی کے مختلف کاموں کو انجام دینے کے لیے ریاضی کے افعال اور مستقلات فراہم کرتا ہے، جیسے مثلثیات، لوگارتھمز، اور راؤنڈنگ۔
6. بے ترتیب: چھدم بے ترتیب نمبر پیدا کرنے اور بے ترتیب انتخاب کرنے اور شفل کرنے کے فنکشنز پیش کرتا ہے۔
7. re: پیٹرن کی مماثلت اور متن کی ہیرا پھیری کے لیے باقاعدہ اظہار کی حمایت کرتا ہے، طاقتور سٹرنگ پروسیسنگ کی صلاحیتوں کی اجازت دیتا ہے۔
8. مجموعے: بلٹ ان ڈیٹا ڈھانچے جیسے فہرستوں، ٹوپلز، اور لغات سے پرے خصوصی کنٹینر ڈیٹا ٹائپس پر مشتمل ہے، بشمول defaultdict۔ اور ،Counter ،deque ،nametuple
9. urllib: URLs درخواستیں بھیجنے، جوابات کو سنبھالنے، اور HTTP کے ساتھ کام کرنے میں سہولت فراہم کرتا ہے، URLs کو پارس کرنے کی اجازت دیتا ہے۔
10. ساکٹ: نیٹ ورک ساکٹ بنانے، نیٹ ورکس پر ڈیٹا بھیجنے اور وصول کرنے، اور نیٹ ورک سرورز اور کلائنٹس بنانے کے لیے کم سطح کے نیٹ ورکنگ انٹرفیس فراہم کرتا ہے۔
11. argparse: پروگراموں Python کمانڈ لائن آرگیومینٹس اور آپشنز کو پارس کرنے کے عمل کو آسان بناتا ہے، جس سے کے لیے کمانڈ لائن انٹرفیس بنانا آسان ہو جاتا ہے۔
12. لاگنگ: مختلف شدت کی سطحوں پر لاگ پیغامات پیدا کرنے، لاگنگ کے رویے کو ترتیب دینے، اور لاگ پیغامات کو مختلف منزلوں تک پہنچانے کے لیے ایک لچکدار اور طاقتور لاگنگ فریم ورک پیش کرتا ہے۔
سٹینڈرڈ لائبریری میں دستیاب بہت سے ماڈیولز اور پیکجز کی چند مثالیں ہیں۔ معیاری لائبریری کی دستاویزات Python یہ کے بلٹ ان ٹولز کے ذریعے فراہم Python کو دریافت کرنے اور مختلف ماڈیولز اور فنکشنز کے ساتھ تجربہ کرنے سے آپ کو کردہ فعالیت کی وسعت اور گہرائی کو دریافت کرنے میں مدد مل سکتی ہے، جس سے آپ کے ترقیاتی منصوبوں میں آپ کا وقت اور محنت کی بچت ہوتی ہے۔

ماڈیولز اور پیکجز کی درآمد
Python میں، ماڈیولز اور پیکجز کوڈ کو منظم اور دوبارہ استعمال کرنے کا طریقہ فراہم کرتے ہیں۔ ماڈیولز انفرادی Python فائلیں ہیں جن میں فنکشنز، کلاسز اور متغیرات شامل ہیں، جبکہ پیکجز ایک سے زیادہ ماڈیولز پر مشتمل ڈائریکٹریز ہیں۔
:میں ماڈیولز اور پیکجز کیسے درآمد کریں Python آئیے دریافت کرتے ہیں کہ
:ماڈیولز درآمد کرنا
آپ امپورٹ سٹیٹمنٹ کے بعد ماڈیول کا نام استعمال کر کے ماڈیولز درآمد کر سکتے ہیں۔ ایک بار درآمد کرنے کے بعد، آپ ڈاٹ نوٹیشن (.) کا استعمال کرتے ہوئے ماڈیول میں بیان کردہ فنکشنز، کلاسز اور متغیر تک رسائی حاصل کر سکتے ہیں۔
:نحو
ازگر
ماڈیول_نام درآمد کریں۔
:مثال
ازگر
تک رسائی حاصل کرنا # 'pi' سے متغیر ((math.sqrt(25) درآمد کریں # ریاضی کے ماڈیول پرنٹ (math.pi) ریاضی پرنٹ تک رسائی 'sqrt' ریاضی کے ماڈیول سے فنکشن
:عرف کے ساتھ درآمد کرنا
چھوٹا یا زیادہ آسان نام فراہم کرنے کے لیے آپ عرف کے ساتھ ماڈیول درآمد کر سکتے ہیں۔
:نحو
ازگر

ماڈیول_نام کو عرف کے طور پر درآمد کریں۔
:مثال
ازگر
تک 'pi' کا استعمال کرتے ہوئے ریاضی کے ماڈیول سے متغیر 'm' کے بطور ریاضی درآمد کریں # عرف m print(m.pi)
رسائی حاصل کرنا
:مخصوص اشیاء کی درآمد
آپ پورے ماڈیول کو درآمد کرنے کے بجائے کسی ماڈیول سے مخصوص فنکشنز، کلاسز یا متغیرات درآمد کر سکتے ہیں۔
:نحو
ازگر
module_name سے import item_name1, item_name2, ...
:مثال
ازگر
تک براہ راست 'sqrt' فنکشن # ((sqrt(25)تک رسائی براہ راست پرنٹ 'pi' متغیر # sqrt print(pi) ،سے math import pi
رسائی
:ہر چیز درآمد کرنا
آپ ماڈیول سے تمام اشیاء کو موجودہ نام کی جگہ میں درآمد کر سکتے ہیں، لیکن عام طور پر اس کی حوصلہ شکنی کی جاتی ہے کیونکہ یہ نام کی جگہ کی آلودگی اور تنازعات کا باعث بن سکتا ہے۔
:نحو
ازگر
* ماڈیول_نام درآمد سے
:مثال
ازگر
تک براہ راست رسائی 'sqrt' فنکشن # ((sqrt(25)تک رسائی براہ راست پرنٹ 'pi' متغیر # (pi)ریاضی کی درآمد سے * پرنٹ
:پیکجز درآمد کرنا
پیکیجز ایک سے زیادہ ماڈیولز پر مشتمل ڈائریکٹریز ہیں۔ آپ ڈاٹ نوٹیشن (.) کا استعمال کرتے ہوئے پیکجوں سے ماڈیول درآمد کر سکتے ہیں۔
:نحو
ازگر
پیکیج_نام. ماڈیول_نام درآمد کریں۔
:مثال
ازگر
urllib.request جواب درآمد کریں = urllib.request.urlopen('https://www.example.com')html = response.read()
:نتیجہ
پروجیکٹس کو مؤثر طریقے سے منظم کرنے Python ماڈیولز اور پیکجز کی درآمد آپ کو کوڈ کو دوبارہ استعمال کرنے اور اپنے کی اجازت دیتا ہے۔ ماڈیولز درآمد کرنے کے طریقے کو سمجھ کر، عرفیت، مخصوص اشیاء کو درآمد کرنا، اور پیکجوں سے درآمد کوڈ لکھ سکتے ہیں۔ Python کرنا، آپ کلینر، زیادہ ماڈیولر، اور زیادہ برقرار رکھنے کے قابل

Understanding exceptions and errors

In Python, exceptions and errors are types of events that occur during program execution that disrupt the normal flow of the program. Understanding exceptions and errors is essential for writing robust and reliable code. Let's delve into the concepts of exceptions and errors in Python:

Exceptions:

An exception is an event that occurs during the execution of a program, which disrupts the normal flow of the program's instructions. When an exception occurs, the Python interpreter raises an exception object, which can then be handled by the program. Exceptions can occur for various reasons, such as invalid input, file not found, division by zero, etc.

Errors:

Errors in Python refer to situations where the interpreter is unable to execute the code due to a violation of the language syntax or rules. Unlike exceptions, errors are not typically recoverable, and they often result in the termination of the program. Common types of errors in Python include syntax errors, runtime errors, and semantic errors.

Handling Exceptions:

Python provides a mechanism for handling exceptions using `try-except` blocks. With `try-except` blocks, you can catch exceptions that occur during the execution of your code and handle them gracefully, preventing the program from crashing. Additionally, you can use `finally` blocks for cleanup tasks that should be executed regardless of whether an exception occurred.

Example:

```
python

try:    x = 10 / 0  # Division by zero raises ZeroDivisionErrorexcept
ZeroDivisionError:    print("Error: Division by zero occurred")
```

Common Built-in Exceptions:

Python comes with a set of built-in exception classes that represent common error conditions. Some of the most commonly encountered built-in exceptions include `SyntaxError`, `NameError`, `TypeError`, `ValueError`, `FileNotFoundError`, `ZeroDivisionError`, etc.

Handling Errors:

While some errors are caught by the Python interpreter, others may go unnoticed unless explicitly handled by your code. It's important to anticipate potential errors and handle them appropriately to ensure that your program behaves as expected.

Conclusion:

Understanding exceptions and errors is crucial for writing robust and reliable Python code. By handling exceptions gracefully and anticipating potential error conditions, you can create programs that are more resilient and user-friendly.

مستثنیات اور غلطیوں کو سمجھنا

میں، مستثنیات اور غلطیاں ایسے واقعات ہیں جو پروگرام کے عمل کے دوران پیش آتے ہیں جو پروگرام کے معمول Python کے بہاؤ میں خلل ڈالتے ہیں۔ مستثنیات اور غلطیوں کو سمجھنا مضبوط اور قابل اعتماد کوڈ لکھنے کے لیے ضروری ہے۔ آئیے ازگر میں استثنیٰ اور غلطیوں کے تصورات پر غور کریں:

مستثنیات:

ایک استثناء ایک ایسا واقعہ ہے جو پروگرام کے عمل کے دوران ہوتا ہے، جو پروگرام کی ہدایات کے عام بہاؤ میں خلل ڈالتا مترجم ایک استثنائی چیز اٹھاتا ہے، جسے پروگرام کے ذریعے سنبھالا جا سکتا Python ہے۔ جب کوئی استثناء واقع ہوتا ہے ہے۔ مستثنیات مختلف وجوہات کی بناء پر ہوسکتی ہیں، جیسے غلط ان پٹ، فائل نہیں ملی، صفر سے تقسیم، وغیرہ۔

غلطیاں:

میں خرابیاں ان حالات کا حوالہ دیتی ہیں جہاں مترجم زبان کے نحو یا قواعد کی خلاف ورزی کی وجہ سے کوڈ پر Python عمل درآمد کرنے سے قاصر ہے۔ مستثنیات کے برعکس، غلطیاں عام طور پر قابل بازیافت نہیں ہوتی ہیں، اور ان کے نتیجے میں عام قسم کی غلطیوں میں نحو کی غلطیاں، رن ٹائم کی غلطیاں، اور Python میں اکثر پروگرام ختم ہو جاتا ہے۔ سیمنٹک غلطیاں شامل ہیں۔

ہینڈلنگ مستثنیات:

ازگر ایک طریقہ کار فراہم کرتا ہے استثنیٰ کو ہینڈل کرنے کے لیے ٹرائی سوائے بلاکس کا استعمال کرتے ہوئے۔ ٹرائی سوائے بلاکس کے ساتھ، آپ اپنے کوڈ پر عمل درآمد کے دوران ہونے والے استثناء کو پکڑ سکتے ہیں اور پروگرام کو کریش ہونے سے روکتے ہوئے انہیں احسن طریقے سے سنبھال سکتے ہیں۔ مزید برآں، آپ کلین اپ کے کاموں کے لیے آخر میں بلاکس کا استعمال کر سکتے ہیں جن پر عمل درآمد کیا جانا چاہیے چاہے کوئی استثنا ہوا ہو۔

مثال:

ازگر

```
x = 10 / 0 # کوشش کریں: ZeroDivisionError صفر کے حساب سے تقسیم ZeroDivisionError کو بڑھاتی ہے سوائے
print("Error: Division by zero واقع ہوئی")
```

عام بلٹ ان مستثنیات:

بلٹ ان استثنائی کلاسوں کے ایک سیٹ کے ساتھ آتا ہے جو عام خرابی کے حالات کی نمائندگی کرتا ہے۔ کچھ Python عام طور پر سامنے آنے والی بلٹ ان مستثنیات میں SyntaxError، NameError، TypeError، ValueError، FileNotFoundError، ZeroDivisionError، وغیرہ شامل ہیں۔

ہینڈلنگ کی خرابیاں:

اگرچہ کچھ غلطیاں Python کے ترجمان کے ذریعہ پکڑی جاتی ہیں، جب تک کہ آپ کے کوڈ کے ذریعہ واضح طور پر ہینڈل نہ کیا جائے تو دیگر کا دھیان نہیں دیا جاسکتا ہے۔ ممکنہ غلطیوں کا اندازہ لگانا اور انہیں مناسب طریقے سے سنبھالنا ضروری ہے تاکہ یہ یقینی بنایا جا سکے کہ آپ کا پروگرام توقع کے مطابق برتاؤ کرتا ہے۔

نتیجہ:

کوڈ لکھنے کے لیے بہت ضروری ہے۔ مستثنیات کو احسن Python مستثنیات اور غلطیوں کو سمجھنا مضبوط اور قابل اعتماد طریقے سے سنبھال کر اور ممکنہ خرابی کے حالات کا اندازہ لگا کر، آپ ایسے پروگرام بنا سکتے ہیں جو زیادہ لچکدار اور صارف دوست ہوں۔

Using try-except blocks for error handling

Using `try-except` blocks in Python allows you to handle exceptions gracefully, preventing your program from crashing when errors occur. Here's a guide on how to use `try-except` blocks for error handling:

Basic Syntax:

```
python

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

- The `try` block contains the code that you want to execute, which may raise an exception.
- If an exception of type `ExceptionType` (or its subclass) occurs within the `try` block, the program jumps to the `except` block.
- Inside the `except` block, you can handle the exception gracefully, log an error message, or take other appropriate actions.

Example:

```
python

try:    x = 10 / 0  # Division by zero raises ZeroDivisionErrorexcept ZeroDivisionError:    print("Error: Division by zero occurred")
```

Handling Multiple Exceptions:

You can handle multiple exceptions by specifying multiple `except` blocks or using a single `except` block with a tuple of exception types.

Example:

```
python

try:    # Code that may raise exceptions    # ...except (ValueError, TypeError):    # Handle ValueError or TypeError    # ...except ZeroDivisionError:    # Handle ZeroDivisionError    # ...
```

Using `except` without Specifying Exception Type:

If you don't specify an exception type in the `except` block, it will catch all exceptions. However, this is generally discouraged because it can make debugging more difficult.

Example:

```
python
```

```
try:    # Code that may raise exceptions    # ...except:    # Handle all exceptions
# ...
```

`finally` Block:

You can also use a `finally` block, which is executed regardless of whether an exception occurred or not. It's commonly used for cleanup tasks, such as closing files or releasing resources.

Example:

```
python

try:    # Code that may raise exceptions    # ...finally:    # Cleanup code
# ...
```

Conclusion:

Using `try-except` blocks in Python allows you to gracefully handle errors and exceptions, making your code more robust and reliable. By handling exceptions appropriately, you can prevent your program from crashing and provide better user experience.

غلطی سے نمٹنے کے لیے بلاکس کے علاوہ ٹرائی کا استعمال کرنا
میں بلاکس کو چھوڑ کر ٹرائی کا استعمال آپ کو مستثنیات کو احسن طریقے سے ہینڈل کرنے کی اجازت دیتا ہے، Python
جو آپ کے پروگرام کو خراب ہونے سے روکتا ہے۔ غلطی سے نمٹنے کے لیے بلاکس کو چھوڑ کر استعمال کرنے کا طریقہ یہاں
:ایک گائیڈ ہے
:بنیادی نحو
ازگر
... # کوشش کریں: # کوڈ جو ایک استثناء کو بڑھا سکتا ہے # ... استثناء کی قسم: # استثنا کو ہینڈل کرنے کے لیے کوڈ
ٹرائی بلاک میں وہ کوڈ ہوتا ہے جسے آپ عمل میں لانا چاہتے ہیں، جو ایک استثناء کو بڑھا سکتا ہے۔ •
کی ایک استثناء ہوتی ہے، تو پروگرام سوائے بلاک (یا اس کے ذیلی طبقے) ExceptionType اگر ٹرائی بلاک کے اندر •
پر چھلانگ لگا دیتا ہے۔
سوائے بلاک کے اندر، آپ استثنیٰ کو خوش اسلوبی سے ہینڈل کر سکتے ہیں، غلطی کے پیغام کو لاگ کر سکتے ہیں، •
یا دیگر مناسب اقدامات کر سکتے ہیں۔
:مثال
ازگر
ZeroDivisionError: کو بڑھاتی ہے سوائے ZeroDivisionError صفر کے حساب سے تقسیم # 0 / 10 = x :کوشش کریں
("واقع ہوئی print("Error: Division by zero
:متعدد مستثنیات کو سنبھالنا
آپ بلاکس کے علاوہ ایک سے زیادہ کی وضاحت کرکے یا ایک استثنائی اقسام کے ساتھ ایک سوائے بلاک کا استعمال کرکے
متعدد استثناء کو سنبھال سکتے ہیں۔
:مثال
ازگر
یا ValueError ہینڈل # :(ValueError, TypeError) کوشش کریں: # کوڈ جو مستثنیات کو بڑھا سکتا ہے # ... سوائے
... # ZeroDivisionError ہینڈل # :ZeroDivisionError سوائے ... # TypeError
:استثناء کی قسم کی وضاحت کیے بغیر استعمال کرنا
اگر آپ سوائے بلاک میں ایک استثنائی قسم کی وضاحت نہیں کرتے ہیں، تو یہ تمام مستثنیات کو پکڑ لے گا۔ تاہم، عام طور
پر اس کی حوصلہ شکنی کی جاتی ہے کیونکہ یہ ڈیبگنگ کو مزید مشکل بنا سکتا ہے۔
:مثال
ازگر
... # کوشش کریں: # کوڈ جو مستثنیات کو بڑھا سکتا ہے # ... سوائے: # تمام استثناء کو ہینڈل کریں
:آخر میں بلاک کریں
آپ ایک حتمی بلاک بھی استعمال کر سکتے ہیں، جس پر عمل درآمد کیا جاتا ہے قطع نظر اس کے کہ کوئی استثناء واقع ہوا
ہے یا نہیں۔ یہ عام طور پر صفائی کے کاموں کے لیے استعمال ہوتا ہے، جیسے فائلوں کو بند کرنا یا وسائل جاری کرنا۔
:مثال
ازگر
... # آزمائیں: # کوڈ جو مستثنیات کو بڑھا سکتا ہے # ... آخر میں: # کلین اپ کوڈ
:نتیجہ
میں بلاکس کو چھوڑ کر ٹرائی کا استعمال آپ کو غلطیوں اور مستثنیات کو احسن طریقے سے سنبھالنے کی اجازت Python
دیتا ہے، جس سے آپ کا کوڈ زیادہ مضبوط اور قابل اعتماد ہوتا ہے۔ مستثنیات کو مناسب طریقے سے سنبھال کر، آپ اپنے
پروگرام کو کریش ہونے سے روک سکتے ہیں اور صارف کا بہتر تجربہ فراہم کر سکتے ہیں۔

Raising exceptions

In Python, you can raise exceptions explicitly using the `raise` statement. This allows you to signal that an error condition has occurred within your code. When you raise an exception, you can provide an exception type and an optional error message to provide context about the error. Here's how you can raise exceptions in Python:

Syntax:

```
python

raise ExceptionType("Error message")
```

- `ExceptionType`: The type of exception to raise. This can be any built-in or user-defined exception class.
- `"Error message"`: An optional error message providing context about the exception.

Example:

```
python

def divide(x, y):    if y == 0:        raise ZeroDivisionError("Cannot divide by zero")    return x / y try:    result = divide(10, 0)except ZeroDivisionError as e: print("Error:", e)
```

In this example:

- The `divide` function checks if the divisor `y` is zero. If it is, it raises a `ZeroDivisionError` with the error message "Cannot divide by zero".
- In the `try-except` block, we call the `divide` function with arguments `10` and `0`. Since the divisor is zero, the function raises a `ZeroDivisionError`.
- The `except` block catches the `ZeroDivisionError` exception and prints the error message associated with it.

Custom Exceptions:

You can also define your own custom exception classes by subclassing built-in exception classes or the `Exception` class. This allows you to create more specific error types for your applications.

```
python

class MyError(Exception):    pass raise MyError("An error occurred")
```

Using `assert`:

You can also use the `assert` statement to raise an `AssertionError` if a condition is not met. This is commonly used for debugging and ensuring that certain conditions are true during program execution.

```
python

x = 10assert x > 0, "x must be positive"
```

If the condition `x > 0` is not met, the `assert` statement raises an `AssertionError` with the specified error message.

Conclusion:

Raising exceptions in Python allows you to signal error conditions and handle them gracefully in your code. Whether you're using built-in exceptions or defining custom ones, raising exceptions provides a way to communicate errors and ensure that your programs behave predictably.

مستثنیات کو بڑھانا
بیان کا استعمال کرتے ہوئے واضح طور پر مستثنیات کو بڑھا سکتے ہیں۔ یہ آپ کو یہ اشارہ کرنے raise میں، آپ Python کی اجازت دیتا ہے کہ آپ کے کوڈ میں ایک خرابی کی حالت واقع ہوئی ہے۔ جب آپ استثنیٰ اٹھاتے ہیں، تو آپ استثنیٰ کی قسم اور غلطی کے بارے میں سیاق و سباق فراہم کرنے کے لیے ایک اختیاری غلطی کا پیغام فراہم کر سکتے ہیں۔ یہاں یہ ہے :میں مستثنیات کیسے بڑھا سکتے ہیں Python کہ آپ
:نحو
ازگر
اٹھائیں ("خرابی کا پیغام") ExceptionType
استثنیٰ کی قسم: استثنیٰ کی قسم جس کو بڑھانا ہے۔ یہ کوئی بلٹ ان یا صارف کی وضاحت شدہ استثنائی کلاس ہو سکتی ہے۔
خرابی کا پیغام": ایک اختیاری غلطی کا پیغام جو استثنا کے بارے میں سیاق و سباق فراہم کرتا ہے۔" •
:مثال
ازگر
کوشش کریں: x / y واپس کریں ("Cannot divide by zero")raise ZeroDivisionError def divide(x, y): if y == 0:
e: print("Error:", e) کے بطور ZeroDivisionError سوائے(10, 0)divide = نتیجہ
:اس مثال میں
صفر ہے۔ اگر یہ ہے تو، یہ غلطی کے پیغام کے ساتھ ایک y تقسیم کا فنکشن چیک کرتا ہے کہ آیا تقسیم •
اٹھاتا ہے "صفر سے تقسیم نہیں کیا جا سکتا"۔ ZeroDivisionError
ٹرائی کے علاوہ بلاک میں، ہم ڈیوائیڈ فنکشن کو آرگیومینٹس 10 اور 0 کے ساتھ کہتے ہیں۔ چونکہ ڈیوائزر صفر ہے، اس • پیدا کرتا ہے۔ ZeroDivisionError لیے فنکشن ایک
استثناء کو پکڑتا ہے اور اس سے وابستہ ایرر میسج پرنٹ کرتا ہے۔ ZeroDivisionError سوائے بلاک •
:حسب ضرورت مستثنیات
کلاس کو ذیلی کلاس کرکے اپنی مرضی کے استثنائی کلاسوں کی بھی وضاحت Exception آپ بلٹ ان ایکسپیشن کلاسز یا کرسکتے ہیں۔ یہ آپ کو اپنی ایپلیکیشنز کے لیے مزید مخصوص خرابی کی قسمیں بنانے کی اجازت دیتا ہے۔
ازگر
("ایک غلطی ہو گئی")raise MyError پاس :(استثنیٰ)MyError کلاس
:دعوی کا استعمال کرتے ہوئے
اسٹیٹمنٹ بھی استعمال کر سکتے assert کو بڑھانے کے لیے AssertionError اگر کوئی شرط پوری نہیں ہوتی ہے تو آپ ہیں۔ یہ عام طور پر ڈیبگنگ اور اس بات کو یقینی بنانے کے لیے استعمال کیا جاتا ہے کہ پروگرام کے عمل کے دوران کچھ شرائط درست ہیں۔
ازگر
"مثبت ہونا چاہیے x"، x > 0 اصرار x = 10
پیدا کرتا ہے۔ AssertionError پوری نہیں ہوتی ہے، تو دعویٰ کا بیان مخصوص غلطی کے پیغام کے ساتھ x > 0 اگر شرط
:نتیجہ
میں مستثنیات کو بڑھانا آپ کو غلطی کے حالات کا اشارہ کرنے اور اپنے کوڈ میں ان کو احسن طریقے سے Python سنبھالنے کی اجازت دیتا ہے۔ چاہے آپ بلٹ ان مستثنیات استعمال کر رہے ہوں یا حسب ضرورت کی وضاحت کر رہے ہوں، مستثنیات کو بڑھانا غلطیوں کو بات چیت کرنے اور اس بات کو یقینی بنانے کا ایک طریقہ فراہم کرتا ہے کہ آپ کے پروگرام پیش گوئی کے مطابق برتاؤ کریں۔

Debugging techniques and tools

Debugging is the process of finding and fixing errors, or bugs, in your code. It's an essential skill for any programmer. Here are some techniques and tools for debugging in Python:

Techniques:

1. **Print Statements**: Inserting print statements at various points in your code to output the values of variables or to indicate the execution flow.
2. **Debugger**: Using a debugger to step through your code line by line, inspect variables, and track the execution flow.
3. **Logging**: Using the `logging` module to log messages at different levels (e.g., debug, info, warning, error) to help diagnose issues.
4. **Code Review**: Having someone else review your code to spot errors or provide suggestions for improvement.
5. **Rubber Duck Debugging**: Explaining your code and problem-solving process to an inanimate object or colleague, often leading to insights and solutions.

Tools:

1. **pdb**: The Python Debugger (pdb) is a built-in interactive debugger that allows you to step through your code, set breakpoints, inspect variables, and more.
2. **breakpoint()**: The `breakpoint()` function (introduced in Python 3.7) serves as a convenient way to set breakpoints in your code without needing to import the `pdb` module explicitly.
3. **IDEs**: Integrated Development Environments (IDEs) like PyCharm, Visual Studio Code, and Spyder provide built-in debugging tools with features such as breakpoints, variable inspection, and stepping through code.
4. **Online Debugging Tools**: Websites like Python Tutor (pythontutor.com) offer online environments where you can visualize the execution of your code step by step.
5. **Logging Module**: Python's built-in `logging` module allows you to log messages to various destinations (e.g., console, file) with different levels of severity for debugging purposes.
6. **Third-Party Debuggers**: Tools like `pdb++`, `ipdb`, and `pdbpp` provide enhancements and additional features on top of the standard Python debugger.

Best Practices:

1. **Start Small**: Isolate the problem by reducing the scope of your code or focusing on a specific function or module.

2. **Reproduce the Issue**: Identify the steps or conditions that trigger the problem and try to reproduce it consistently.
3. **Use Version Control**: Use version control systems like Git to track changes in your code and revert to previous versions if needed.
4. **Document Changes**: Keep track of changes you make while debugging and document the steps you take to resolve issues for future reference.
5. **Test Driven Development (TDD)**: Write tests for your code before implementing new features or making changes, which can help identify issues early and guide your debugging efforts.

By combining these techniques and tools, you can effectively debug your Python code and resolve issues efficiently. Remember that debugging is an iterative process, and don't hesitate to seek help from colleagues or online communities if you're stuck on a problem.

ڈیبگنگ کی تکنیک اور ٹولز
ڈیبگنگ آپ کے کوڈ میں غلطیوں، یا کیڑوں کو تلاش کرنے اور ٹھیک کرنے کا عمل ہے۔ یہ کسی بھی پروگرامر کے لیے ایک ضروری مہارت ہے۔ Python میں ڈیبگ کرنے کے لیے کچھ تکنیکیں اور ٹولز یہ ہیں:
تکنیک:
1. پرنٹ اسٹیٹمنٹس: اپنے کوڈ میں مختلف پوائنٹس پر پرنٹ اسٹیٹمنٹ داخل کرنا متغیرات کی قدروں کو آؤٹ پٹ کرنے کے لیے یا ایگزیکیوشن فلو کی نشاندہی کرنے کے لیے۔
2. ڈیبگر: ایک ڈیبگر کا استعمال کرتے ہوئے آپ کے کوڈ لائن کے ذریعے لائن کے ذریعے قدم، متغیرات کا معائنہ کریں، اور عملدرآمد کے بہاؤ کو ٹریک کریں۔
3. لاگنگ: مختلف سطحوں پر پیغامات کو لاگ کرنے کے لیے لاگنگ ماڈیول کا استعمال کرتے ہوئے (مثلاً، ڈیبگ، معلومات، انتباہ، غلطی) مسائل کی تشخیص میں مدد کے لیے۔
4. کوڈ کا جائزہ: غلطیوں کی نشاندہی کرنے یا بہتری کے لیے تجاویز فراہم کرنے کے لیے کسی اور سے آپ کے کوڈ کا جائزہ لینا۔
5. ربڑ ڈک ڈیبگنگ: کسی بے جان چیز یا ساتھی کو اپنے کوڈ اور مسئلہ حل کرنے کے عمل کی وضاحت کرنا، جو اکثر بصیرت اور حل کی طرف لے جاتا ہے۔
اوزار:
1. pdb: Python Debugger (pdb) ایک بلٹ ان انٹرایکٹو ڈیبگر ہے جو آپ کو اپنے کوڈ میں قدم رکھنے، بریک پوائنٹس سیٹ کرنے، متغیرات کا معائنہ کرنے اور مزید بہت کچھ کرنے کی اجازت دیتا ہے۔
2. breakpoint(): breakpoint() فنکشن (Python 3.7 میں متعارف کرایا گیا ہے) pdb ماڈیول کو واضح طور پر درآمد کرنے کی ضرورت کے بغیر آپ کے کوڈ میں بریک پوائنٹس سیٹ کرنے کا ایک آسان طریقہ ہے۔
3. IDEs: انٹیگریٹڈ ڈویلپمنٹ انوائرمنٹ (IDEs) جیسے PyCharm، Visual Studio Code، اور Spyder بلٹ ان ڈیبگنگ ٹولز فراہم کرتے ہیں جیسے کہ بریک پوائنٹس، متغیر معائنہ، اور کوڈ کے ذریعے قدم بڑھانا۔
4. آن لائن ڈیبگنگ ٹولز: Python Tutor (pythontutor.com) جیسی ویب سائٹس آن لائن ماحول پیش کرتی ہیں جہاں آپ قدم بہ قدم اپنے کوڈ کے نفاذ کا تصور کر سکتے ہیں۔
5. لاگنگ ماڈیول: ازگر کا بلٹ ان لاگنگ ماڈیول آپ کو مختلف مقامات پر پیغامات لاگ کرنے کی اجازت دیتا ہے (مثلاً کنسول، فائل) ڈیبگنگ کے مقاصد کے لیے شدت کی مختلف سطحوں کے ساتھ۔
6. تھرڈ پارٹی ڈیبگرز: ٹولز جیسے pdb++، ipdb، اور pdbpp معیاری Python ڈیبگر کے اوپر اضافہ اور اضافی خصوصیات فراہم کرتے ہیں۔
بہترین طریقوں:
1. چھوٹا شروع کریں: اپنے کوڈ کے دائرہ کار کو کم کرکے یا کسی مخصوص فنکشن یا ماڈیول پر توجہ مرکوز کرکے مسئلہ کو الگ کریں۔
2. مسئلہ کو دوبارہ پیش کریں: ان اقدامات یا حالات کی نشاندہی کریں جو مسئلہ کو متحرک کرتے ہیں اور اسے مستقل طور پر دوبارہ پیش کرنے کی کوشش کرتے ہیں۔
3. ورژن کنٹرول Git جیسے ورژن کنٹرول کا استعمال کریں: اپنے کوڈ میں ہونے والی تبدیلیوں کو ٹریک کرنے کے لیے سسٹم کا استعمال کریں اور اگر ضرورت ہو تو پچھلے ورژن پر واپس جائیں۔
4. دستاویزی تبدیلیاں: ڈیبگنگ کے دوران آپ جو تبدیلیاں کرتے ہیں ان کا ٹریک رکھیں اور مستقبل کے حوالے کے لیے مسائل کو حل کرنے کے لیے کیے جانے والے اقدامات کو دستاویز کریں۔
5. ٹیسٹ ڈرائیون ڈیولپمنٹ (TDD): ٹیسٹ نئی خصوصیات کو لاگو کرنے یا تبدیلیاں کرنے سے پہلے اپنے کوڈ کے لیے لکھیں، جو مسائل کی جلد شناخت کرنے اور آپ کی ڈیبگنگ کوششوں کی رہنمائی میں مدد کر سکتے ہیں۔
کوڈ کو مؤثر طریقے سے ڈیبگ کر سکتے ہیں اور مسائل کو مؤثر Python ان تکنیکوں اور ٹولز کو یکجا کر کے، آپ اپنے طریقے سے حل کر سکتے ہیں۔ یاد رکھیں کہ ڈیبگنگ ایک تکراری عمل ہے، اور اگر آپ کسی مسئلے میں پھنس گئے ہیں تو ساتھیوں یا آن لائن کمیونٹیز سے مدد لینے میں ہچکچاہٹ محسوس نہ کریں۔

Best practices for writing clean and debuggable code

Writing clean and debuggable code is crucial for improving code quality, readability, and maintainability. Here are some best practices to follow:

1. Use Meaningful Names:

- Choose descriptive and meaningful names for variables, functions, classes, and modules.
- Use consistent naming conventions (e.g., camelCase, snake_case) to improve readability.

2. Write Modular Code:

- Break down your code into smaller, reusable functions and classes.
- Each function or class should have a single responsibility (the Single Responsibility Principle).

3. Follow PEP 8 Guidelines:

- Adhere to the Python Enhancement Proposal 8 (PEP 8) guidelines for coding style and conventions.
- Use tools like `flake8` or IDE plugins to automatically check for PEP 8 compliance.

4. Write Docstrings:

- Document your code with clear and informative docstrings to explain its purpose, usage, and parameters.
- Follow the numpydoc or Google-style docstring conventions for consistency.

5. Handle Errors Gracefully:

- Use try-except blocks to catch and handle exceptions gracefully.
- Provide informative error messages and logging to aid in debugging.

6. Avoid Magic Numbers and Strings:

- Replace magic numbers and strings with named constants or variables to improve readability and maintainability.
- Define constants at the top of your module or in a separate configuration file.

7. Write Unit Tests:

- Write automated unit tests to verify the correctness of your code.
- Test edge cases and boundary conditions to ensure robustness.

8. Follow the DRY Principle:

- Don't Repeat Yourself (DRY): Avoid duplicating code by extracting common functionality into functions, classes, or modules.
- Encapsulate repeated patterns into reusable abstractions.

9. Keep Functions and Classes Small:

- Aim for concise and focused functions and classes.
- Break up long functions or classes into smaller, more manageable pieces.

10. Use Version Control:

- Use a version control system like Git to track changes to your code and collaborate with others.
- Commit your changes frequently and write informative commit messages.

11. Comment Thoughtfully:

- Write clear and concise comments to explain complex logic or algorithms.
- Avoid unnecessary or redundant comments that merely restate the code.

12. Continuously Refactor:

- Refactor your code regularly to improve its structure, readability, and performance.
- Take the time to review and improve existing code as part of your development process.

Conclusion:

Following these best practices can help you write clean, readable, and debuggable code that is easier to maintain and understand. By adhering to coding standards, documenting your code, handling errors gracefully, and practicing good software engineering principles, you can improve the quality and reliability of your Python projects.

صاف اور ڈیبگ ایبل کوڈ لکھنے کے بہترین طریقے

صاف اور ڈیبگ ایبل کوڈ لکھنا کوڈ کے معیار، پڑھنے کی اہلیت، اور برقرار رکھنے کی صلاحیت کو بہتر بنانے کے لیے بہت ضروری ہے۔ پیروی کرنے کے لیے یہاں کچھ بہترین طریقے ہیں:

1. معنی خیز نام استعمال کریں:

متغیرات، فنکشنز، کلاسز اور ماڈیولز کے لیے وضاحتی اور معنی خیز ناموں کا انتخاب کریں۔
• پڑھنے کی اہلیت کو بہتر بنانے کے لیے مستقل نام کے کنونشنز (جیسے، کیمل کیس، سانپ_کیس) استعمال کریں۔

2. ماڈیولر کوڈ لکھیں:

• اپنے کوڈ کو چھوٹے، دوبارہ قابل استعمال فنکشنز اور کلاسز میں تقسیم کریں۔
• ہر فنکشن یا کلاس کی ایک ذمہ داری ہونی چاہیے (سنگل ذمہ داری کا اصول)۔

3. PEP 8 کے رہنما خطوط پر عمل کریں:

• Python Enhancement Proposal 8 (PEP 8) کوڈنگ کے انداز اور کنونشنز کے لیے کے رہنما خطوط پر عمل کریں۔
• PEP 8 پلگ ان جیسے ٹولز کا استعمال کریں۔ IDE یا flake8 کی تعمیل کی خود بخود جانچ کرنے کے لیے

4. Docstrings لکھیں:

• اپنے کوڈ کو اس کے مقصد، استعمال اور پیرامیٹرز کی وضاحت کرنے کے لیے واضح اور معلوماتی دستاویز کے ساتھ دستاویز کریں۔
• docstring طرز کے Google یا numpydoc مستقل مزاجی کے لیے کنونشنز پر عمل کریں۔

5. غلطیوں کو احسن طریقے سے ہینڈل کریں:

• مستثنیات کو پکڑنے اور ہینڈل کرنے کے لیے بلاکس کے علاوہ ٹرائی کا استعمال کریں۔
• ڈیبگنگ میں مدد کے لیے معلوماتی غلطی کے پیغامات اور لاگنگ فراہم کریں۔

6. جادوئی نمبروں اور تاروں سے بچیں:

• میجک نمبرز اور سٹرنگز کو نام مستقل یا متغیر سے بدلیں تاکہ پڑھنے کی اہلیت اور برقراری کو بہتر بنایا جا سکے۔
• اپنے ماڈیول کے اوپری حصے میں یا الگ کنفیگریشن فائل میں مستقل کی وضاحت کریں۔

7. یونٹ ٹیسٹ لکھیں:

• اپنے کوڈ کی درستگی کی تصدیق کے لیے خودکار یونٹ ٹیسٹ لکھیں۔
• مضبوطی کو یقینی بنانے کے لیے ایج کیسز اور باؤنڈری کنڈیشنز کی جانچ کریں۔

8. DRY اصول پر عمل کریں:

• اپنے آپ کو نہ دہرائیں (DRY): فنکشنز، کلاسز، یا ماڈیولز میں عام فعالیت کو نکال کر کوڈ کو ڈپلیکیٹ کرنے سے گریز کریں۔
• دہرائے گئے نمونوں کو دوبارہ قابل استعمال تجریدوں میں سمیٹیں۔

9. فنکشنز اور کلاسز کو چھوٹا رکھیں:

• مختصر اور مرکوز فنکشنز اور کلاسز کا مقصد۔
• طویل فنکشنز یا کلاسز کو چھوٹے، زیادہ قابل انتظام ٹکڑوں میں تقسیم کریں۔

10. ورژن کنٹرول کا استعمال کریں:

• جیسا ورژن کنٹرول Git اپنے کوڈ میں ہونے والی تبدیلیوں کو ٹریک کرنے اور دوسروں کے ساتھ تعاون کرنے کے لیے سسٹم استعمال کریں۔
• اپنی تبدیلیاں کثرت سے کریں اور معلوماتی کمٹ میسیج لکھیں۔

11. سوچ سمجھ کر تبصرہ کریں:

پیچیدہ منطق یا الگورتھم کی وضاحت کے لیے واضح اور جامع تبصرے لکھیں۔
غیر ضروری یا بے کار تبصروں سے گریز کریں جو صرف کوڈ کو دوبارہ بیان کرتے ہیں۔

12. مسلسل ریفیکٹر:

• اپنے کوڈ کی ساخت، پڑھنے کی اہلیت اور کارکردگی کو بہتر بنانے کے لیے باقاعدگی سے ریفیکٹر کریں۔
• اپنے ترقیاتی عمل کے حصے کے طور پر موجودہ کوڈ کا جائزہ لینے اور اسے بہتر بنانے کے لیے وقت نکالیں۔

نتیجہ:

ان بہترین طریقوں پر عمل کرنے سے آپ کو صاف، پڑھنے کے قابل، اور ڈیبگ ایبل کوڈ لکھنے میں مدد مل سکتی ہے جسے برقرار رکھنے اور سمجھنا آسان ہے۔ کوڈنگ کے معیارات پر عمل کرکے، اپنے کوڈ کو دستاویزی شکل دے کر، غلطیوں کو خوش اسلوبی سے سنبھال کر، اور سافٹ ویئر انجینئرنگ کے اچھے اصولوں پر عمل کرکے، آپ اپنے Python پروجیکٹس کے معیار اور وشوسنییتا کو بہتر بنا سکتے ہیں۔

Tensorflow examples

1. Hello, TensorFlow!

This is a simple example to ensure that TensorFlow is installed correctly. It prints “Hello, TensorFlow!” to the console.

```
import tensorflow as tf

# Create a TensorFlow constant hello = tf.constant('Hello, TensorFlow!')

# Start a TensorFlow session with tf.Session() as session:

# Run the session and print the output

  print(session.run(hello))
```

2. Linear Regression:

Linear regression is a fundamental machine learning task. Here's a simple example using TensorFlow:

```
import tensorflow as tf import numpy as np

# Generate random data x = np.random.rand(100).astype(np.float32) y = 2 * x + 1

# Create TensorFlow variables for the model parameters W =
tf.Variable(tf.random.normal([1])) b = tf.Variable(tf.zeros([1]))

# Define the linear regression model y_pred = W * x + b

# Define the loss function (Mean Squared Error) loss = tf.reduce_mean(tf.square(y_pred - y))

# Create an optimizer (e.g., Gradient Descent) optimizer =
tf.train.GradientDescentOptimizer(0.1) train_op = optimizer.minimize(loss)

# Initialize the variables init = tf.global_variables_initializer()

# Create a TensorFlow session and train the model with tf.Session() as session:

  session.run(init)

  for step in range(1000):

      session.run(train_op)

  # Print the learned parameters

  learned_W, learned_b = session.run([W, b])

  print(f'Learned W: {learned_W[0]}, Learned b: {learned_b[0]}')
```

3. Image Classification with Convolutional Neural Network (CNN):

This example demonstrates image classification using a CNN with TensorFlow and the MNIST dataset.

```
import tensorflow as tf from tensorflow.keras import datasets, layers, models import matplotlib.pyplot as plt

# Load the MNIST dataset (train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()

# Preprocess the data train_images, test_images = train_images / 255.0, test_images / 255.0

# Build a simple CNN model model = models.Sequential([

  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),

  layers.MaxPooling2D((2, 2)),

  layers.Flatten(),

  layers.Dense(128, activation='relu'),

  layers.Dense(10)

])

# Compile the model model.compile(optimizer='adam',

        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),

        metrics=['accuracy'])

# Train the model model.fit(train_images, train_labels, epochs=5)

# Evaluate the model test_loss, test_acc = model.evaluate(test_images, test_labels) print(f“Test accuracy: {test_acc}”)

# Make predictions predictions = model.predict(test_images)
```

These examples cover a range of TensorFlow use cases, from simple “Hello, TensorFlow!” to more complex tasks like linear regression and image classification with convolutional neural networks. TensorFlow is a powerful library with extensive documentation and resources for further exploration and learning.

صاف اور ڈیبگ ایبل کوڈ لکھنے کے بہترین طریقے

صاف اور ڈیبگ ایبل کوڈ لکھنا کوڈ کے معیار، پڑھنے کی اہلیت، اور برقرار رکھنے کی صلاحیت کو بہتر بنانے کے لیے بہت ضروری ہے۔ پیروی کرنے کے لیے یہاں کچھ بہترین طریقے ہیں:

1. معنی خیز نام استعمال کریں:

متغیرات، فنکشنز، کلاسز اور ماڈیولز کے لیے وضاحتی اور معنی خیز ناموں کا انتخاب کریں۔

• پڑھنے کی اہلیت کو بہتر بنانے کے لیے مستقل نام کے کنونشنز (جیسے، کیمل کیس، سانپ_کیس) استعمال کریں۔

2. ماڈیولر کوڈ لکھیں:

• اپنے کوڈ کو چھوٹے، دوبارہ قابل استعمال فنکشنز اور کلاسز میں تقسیم کریں۔

• ہر فنکشن یا کلاس کی ایک ذمہ داری ہونی چاہیے (سنگل ذمہ داری کا اصول)۔

3. PEP 8 کے رہنما خطوط پر عمل کریں:

• Python Enhancement Proposal 8 (PEP 8) کے رہنما خطوط پر عمل کوڈنگ کے انداز اور کنونشنز کے لیے کریں۔

• PEP 8 کی تعمیل کی خود بخود جانچ کرنے کے لیے IDE یا flake8 جیسے ٹولز کا استعمال کریں۔ پلگ ان

4. Docstrings لکھیں:

• اپنے کوڈ کو اس کے مقصد، استعمال اور پیرامیٹرز کی وضاحت کرنے کے لیے واضح اور معلوماتی دستاویز کے ساتھ دستاویز کریں۔

• docstring کنونشنز پر عمل کریں۔ Google طرز کے numpydoc یا مستقل مزاجی کے لیے

5. غلطیوں کو احسن طریقے سے ہینڈل کریں:

• مستثنیات کو پکڑنے اور ہینڈل کرنے کے لیے بلاکس کے علاوہ ٹرائی کا استعمال کریں۔

• ڈیبگنگ میں مدد کے لیے معلوماتی غلطی کے پیغامات اور لاگنگ فراہم کریں۔

6. جادوئی نمبروں اور تاروں سے بچیں:

• میجک نمبرز اور سٹرنگز کو نام مستقل یا متغیر سے بدلیں تاکہ پڑھنے کی اہلیت اور برقراری کو بہتر بنایا جا سکے۔

• اپنے ماڈیول کے اوپری حصے میں یا الگ کنفیگریشن فائل میں مستقل کی وضاحت کریں۔

7. یونٹ ٹیسٹ لکھیں:

• اپنے کوڈ کی درستگی کی تصدیق کے لیے خودکار یونٹ ٹیسٹ لکھیں۔

• مضبوطی کو یقینی بنانے کے لیے ایج کیسز اور باؤنڈری کنڈیشنز کی جانچ کریں۔

8. DRY اصول پر عمل کریں:

• فنکشنز، کلاسز، یا ماڈیولز میں عام فعالیت کو نکال کر کوڈ کو ڈپلیکیٹ کرنے سے گریز (DRY): اپنے آپ کو نہ دہرائیں کریں۔

• دہرائے گئے نمونوں کو دوبارہ قابل استعمال تجریدوں میں سمیٹیں۔

9. فنکشنز اور کلاسز کو چھوٹا رکھیں:

• مختصر اور مرکوز فنکشنز اور کلاسز کا مقصد۔

• طویل فنکشنز یا کلاسز کو چھوٹے، زیادہ قابل انتظام ٹکڑوں میں تقسیم کریں۔

10. ورژن کنٹرول کا استعمال کریں:

• جیسا ورژن کنٹرول Git اپنے کوڈ میں ہونے والی تبدیلیوں کو ٹریک کرنے اور دوسروں کے ساتھ تعاون کرنے کے لیے سسٹم استعمال کریں۔

• اپنی تبدیلیاں کثرت سے کریں اور معلوماتی کمٹ میسیج لکھیں۔

11. سوچ سمجھ کر تبصرہ کریں:

پیچیدہ منطق یا الگورتھم کی وضاحت کے لیے واضح اور جامع تبصرے لکھیں۔

غیر ضروری یا بے کار تبصروں سے گریز کریں جو صرف کوڈ کو دوبارہ بیان کرتے ہیں۔

12. مسلسل ریفیکٹر:

• اپنے کوڈ کی ساخت، پڑھنے کی اہلیت اور کارکردگی کو بہتر بنانے کے لیے باقاعدگی سے ریفیکٹر کریں۔

• اپنے ترقیاتی عمل کے حصے کے طور پر موجودہ کوڈ کا جائزہ لینے اور اسے بہتر بنانے کے لیے وقت نکالیں۔

نتیجہ:

ان بہترین طریقوں پر عمل کرنے سے آپ کو صاف، پڑھنے کے قابل، اور ڈیبگ ایبل کوڈ لکھنے میں مدد مل سکتی ہے جسے برقرار رکھنے اور سمجھنا آسان ہے۔ کوڈنگ کے معیارات پر عمل کرکے، اپنے کوڈ کو دستاویزی شکل دے کر، غلطیوں کو خوش اسلوبی سے سنبھال کر، اور سافٹ ویئر انجینئرنگ کے اچھے اصولوں پر عمل کرکے، آپ اپنے Python پروجیکٹس کے معیار اور وشوسنییتا کو بہتر بنا سکتے ہیں۔

Light GBM

LightGBM (Light Gradient Boosting Machine) is an open-source, distributed, high-performance gradient boosting framework that is specifically designed for efficient and scalable machine learning tasks. It is written in C++ but provides Python interfaces for ease of use. LightGBM is known for its speed and efficiency, making it a popular choice for various machine learning applications, including classification, regression, and ranking tasks.

1. **Installation**:

You can install LightGBM using pip:

pip install lightgbm

2. **Importing LightGBM**:

Once installed, you can import LightGBM in your Python script or Jupyter Notebook:

import lightgbm as lgb

3. **Data Preparation**:

Before you can use LightGBM, you need to prepare your data. LightGBM works with tabular data, and it expects the data to be in a format that is compatible with the `Dataset` object provided by the library.

1. Load and preprocess your dataset using libraries like Pandas and NumPy.
2. Split your data into training and testing sets.

4. **Creating a Dataset**:

To efficiently use LightGBM, you need to create a `Dataset` object from your data. This object is optimized for training and prediction.

train_data = lgb.Dataset(data=X_train, label=y_train) test_data = lgb.Dataset(data=X_test, label=y_test, reference=train_data)

5. **Setting Parameters**:

LightGBM has a wide range of parameters that control the training process and the model's behavior. Some important parameters include:

- `objective`: Specifies the learning task (e.g., “regression,” “binary,” or “multiclass”).

- `num_leaves`: Number of leaves in each tree. It controls the complexity of the model.

- `learning_rate`: Step size for updates during training.

- `max_depth`: Maximum depth of the tree.

- `num_boost_round`: Number of boosting iterations (trees).

- `metric`: Evaluation metric for model performance.

You can set these parameters in a dictionary and pass it to the training process.

6. **Training the Model**:

To train the LightGBM model, you use the `train` method, passing in your training data and the parameter dictionary:

num_round = 100 bst = lgb.train(params, train_data, num_round, valid_sets=[test_data], early_stopping_rounds=10)

The `early_stopping_rounds` parameter allows the training to stop early if the evaluation metric doesn't improve for a specified number of rounds on the validation set.

7. **Making Predictions**:

After training, you can use the trained model to make predictions on new data:

predictions = bst.predict(X_new_data)

8. **Model Evaluation**:

Evaluate the model's performance using various metrics. You can access the model's feature importance and even plot the trees in the model to gain insights into its decision-making process.

9. **Hyperparameter Tuning**:

It's common to perform hyperparameter tuning to find the best set of parameters for your specific problem. Techniques like grid search or random search can be used.

10. **Deployment**:

Once you're satisfied with the model's performance, you can deploy it for real-world predictions, either in a production environment or in your applications.

LightGBM's efficiency and speed make it an excellent choice for large datasets and computationally intensive tasks. However, it's essential to understand its parameters and the data you are working with to achieve optimal results.

ہلکا جی بی ایم

LightGBM (Light Gradient Boosting Machine) ایک کھلا ذریعہ، تقسیم شدہ، اعلیٰ کارکردگی والا گریڈینٹ بوسٹنگ فریم ورک ہے جو خاص طور پر موثر اور قابل توسیع مشین سیکھنے کے کاموں کے لیے ڈیزائن کیا گیا ہے۔ یہ C++ اپنی رفتار اور کارکردگی LightGBM انٹرفیس فراہم کرتا ہے۔ Python میں لکھا گیا ہے لیکن استعمال میں آسانی کے لیے کے لیے جانا جاتا ہے، جو اسے مختلف مشین لرننگ ایپلی کیشنز، بشمول درجہ بندی، رجعت، اور درجہ بندی کے کاموں کے لیے ایک مقبول انتخاب بناتا ہے۔

1. تنصیب:

آپ پائپ کا استعمال کرتے ہوئے لائٹ جی بی ایم انسٹال کر سکتے ہیں:

pip install lightgbm

2. لائٹ جی بی ایم درآمد کرنا:

درآمد کر سکتے ہیں LightGBM میں Jupyter Notebook اسکرپٹ یا Python انسٹال ہونے کے بعد، آپ اپنی:

لائٹ جی بی ایم ایل جی بی کے بطور درآمد کریں۔

3. ڈیٹا کی تیاری:

استعمال کر سکیں، آپ کو اپنا ڈیٹا تیار کرنا ہوگا۔ لائٹ جی بی ایم ٹیبلر ڈیٹا کے ساتھ کام LightGBM اس سے پہلے کہ آپ کرتا ہے، اور یہ توقع کرتا ہے کہ ڈیٹا اس فارمیٹ میں ہوگا جو لائبریری کے ذریعہ فراہم کردہ 'ڈیٹا سیٹ' آبجیکٹ کے ساتھ مطابقت رکھتا ہے۔

1. جیسی لائبریریوں کا استعمال کرتے ہوئے اپنے ڈیٹاسیٹ کو لوڈ اور پری پروسیس کریں۔ NumPy پانڈاس اور

2. اپنے ڈیٹا کو ٹریننگ اور ٹیسٹنگ سیٹس میں تقسیم کریں۔

4. ڈیٹا سیٹ بنانا:

آبجیکٹ بنانا ہوگا۔ یہ `Dataset` کو مؤثر طریقے سے استعمال کرنے کے لیے، آپ کو اپنے ڈیٹا سے ایک LightGBM اعتراض تربیت اور پیشین گوئی کے لیے موزوں ہے۔

```
train_data = lgb.Dataset(data=X_train, label=y_train) test_data = lgb.Dataset(data=X_test, label=y_test, reference=train_data)
```

5. پیرامیٹرز کی ترتیب:

میں پیرامیٹرز کی ایک وسیع رینج ہے جو تربیت کے عمل اور ماڈل کے رویے کو کنٹرول کرتی ہے۔ کچھ اہم LightGBM پیرامیٹرز میں شامل ہیں:

- `مقصد`: سیکھنے کے کام کی وضاحت کرتا ہے (مثال کے طور پر، "رجعت،" "بائنری،" یا "ملٹی کلاس")۔`

- `num_leaves`: ہر درخت میں پتوں کی تعداد۔ یہ ماڈل کی پیچیدگی کو کنٹرول کرتا ہے۔

- `learning_rate`: ٹریننگ کے دوران اپ ڈیٹس کے لیے قدم کا سائز۔

- `max_depth`: درخت کی زیادہ سے زیادہ گہرائی۔

- `num_boost_round`: بڑھانے والی تکرار کی تعداد (درخت)۔

میٹرک`: ماڈل کی کارکردگی کا اندازہ میٹرک۔` -

آپ ان پیرامیٹرز کو ایک لغت میں سیٹ کر سکتے ہیں اور اسے تربیتی عمل میں منتقل کر سکتے ہیں۔

6. ماڈل کی تربیت:

ماڈل کو تربیت دینے کے لیے، آپ `ٹرین` کا طریقہ استعمال کرتے ہیں، اپنے تربیتی ڈیٹا اور پیرامیٹر کی لغت LightGBM میں پاس کرتے ہوئے:

num_round = 100 bst = lgb.train(params, train_data, num_round, valid_sets=[test_data], early_stopping_rounds=10)

پیرامیٹر تربیت کو جلد روکنے کی اجازت دیتا ہے اگر تشخیصی میٹرک توثیق سیٹ پر راؤنڈز 'early_stopping_rounds' کی ایک مخصوص تعداد کے لیے بہتر نہیں ہوتا ہے۔

7. پیشین گوئیاں کرنا:

تربیت کے بعد، آپ نئے ڈیٹا پر پیشین گوئیاں کرنے کے لیے تربیت یافتہ ماڈل استعمال کر سکتے ہیں:

پیشین گوئیاں = bst.predict(X_new_data)

8. ماڈل کی تشخیص:

مختلف میٹرکس کا استعمال کرتے ہوئے ماڈل کی کارکردگی کا اندازہ لگائیں۔ آپ ماڈل کی خصوصیت کی اہمیت تک رسائی حاصل کر سکتے ہیں اور اس کے فیصلہ سازی کے عمل کے بارے میں بصیرت حاصل کرنے کے لیے ماڈل میں درختوں کو بھی پلاٹ کر سکتے ہیں۔

9. ہائپر پیرامیٹر ٹیوننگ:

اپنے مخصوص مسئلے کے لیے پیرامیٹرز کا بہترین سیٹ تلاش کرنے کے لیے ہائپر پیرامیٹر ٹیوننگ کرنا عام ہے۔ گرڈ سرچ یا بے ترتیب تلاش جیسی تکنیکوں کا استعمال کیا جا سکتا ہے۔

10. تعیناتی:

ایک بار جب آپ ماڈل کی کارکردگی سے مطمئن ہو جائیں، آپ اسے حقیقی دنیا کی پیشین گوئیوں کے لیے، یا تو پیداواری ماحول میں یا اپنی ایپلی کیشنز میں تعینات کر سکتے ہیں۔

لائٹ جی بی ایم کی کارکردگی اور رفتار اسے بڑے ڈیٹا سیٹس اور کمپیوٹیشنل طور پر گہرے کاموں کے لیے بہترین انتخاب بناتی ہے۔ تاہم، اس کے پیرامیٹرز اور ڈیٹا کو سمجھنا ضروری ہے جس کے ساتھ آپ بہترین نتائج حاصل کرنے کے لیے کام کر رہے ہیں۔

OpenAI Gym

OpenAI Gym is an open-source toolkit designed for developing and comparing reinforcement learning (RL) algorithms. It provides a wide range of environments for building, training, and evaluating RL agents. OpenAI Gym is an essential tool for researchers, students, and practitioners interested in developing and testing RL algorithms.

1. **Installation**:

You can install OpenAI Gym using pip:

pip install gym

2. **Core Concepts**:

OpenAI Gym introduces several fundamental concepts:

- **Environment**: An environment is a task or problem that an RL agent interacts with. Environments in Gym are defined as Python classes and encapsulate the dynamics and rules of the task. For example, classic environments like CartPole, MountainCar, and Atari games are available.

- **Agent**: The RL agent is the learner that interacts with the environment. It takes actions and receives feedback in the form of rewards.

- **Observation Space**: The observation space is the set of all possible states that the agent can perceive from the environment. It can be continuous or discrete, depending on the problem.

- **Action Space**: The action space is the set of all possible actions that the agent can take in the environment. It can be continuous or discrete as well.

- **Reward**: The reward is a numerical value that the agent receives after taking an action in the environment. The goal of the agent is to maximize its cumulative reward over time.

- **Episode**: An episode is a single run or interaction between the agent and the environment, starting from the initial state and continuing until a terminal state is reached.

3. **Getting Started**:

To use OpenAI Gym, you start by creating an environment:

import gym

env = gym.make('CartPole-v1')

4. **Interacting with the Environment**:

You can interact with the environment using a simple loop. In each step, the agent selects an action, and the environment responds with the next state, a reward, and information about the episode's termination:

```
observation = env.reset()
for t in range(max_timesteps):
  action = agent.select_action(observation)
  observation, reward, done, info = env.step(action)
  if done:
     break
```

5. **Custom Environments**:

OpenAI Gym allows you to create custom environments. You need to define an environment class that adheres to the Gym's API. This feature is useful for developing RL environments for specific research or application purposes.

6. **Evaluation and Training**:

OpenAI Gym provides an interface for evaluating and training RL agents. Researchers can use Gym environments to test and benchmark different reinforcement learning algorithms. Several RL libraries, such as TensorFlow, PyTorch, and Stable Baselines, offer integration with Gym to facilitate agent training.

7. **Variety of Environments**:

OpenAI Gym includes a broad range of environments, from classic control problems like CartPole and MountainCar to complex environments like Atari games, robotic control, and 2D and 3D simulations.

8. **Community and Extensions**:

OpenAI Gym has a vibrant community, and it's common to find extensions, custom environments, and wrappers for the toolkit. Some popular extensions include Gym Retro for retro game emulation and Gym-Snake for playing Snake using RL agents.

9. **Visualization**:

Gym allows you to render environments to visualize the agent's behavior. This can be useful for debugging and understanding the learning process.

10. **Baselines and Leaderboards**:

OpenAI Gym provides a collection of benchmark problems, or baselines, to compare the performance of your RL agents. The Gym website also maintains leaderboards for different environments, allowing you to see how well various algorithms perform.

OpenAI Gym serves as a fundamental tool for developing and experimenting with reinforcement learning algorithms, making it easier to understand, test, and compare the performance of different approaches in various environments. It has played a significant role in advancing the field of RL and enabling its applications in various domains.

اوپن اے آئی جم

الگورتھم تیار کرنے اور موازنہ کرنے کے لیے (RL) جم ایک اوپن سورس ٹول کٹ ہے جسے ریانفورسمنٹ لرننگ OpenAI OpenAI ڈیزائن کیا گیا ہے۔ یہ آر ایل ایجنٹس کی تعمیر، تربیت اور جانچ کے لیے وسیع پیمانے پر ماحول فراہم کرتا ہے۔ الگورتھم کو تیار کرنے اور جانچنے میں دلچسپی رکھنے والے پریکٹیشنرز کے لیے ایک ضروری ٹول RL جم محققین، طلباء، اور ہے۔

1. تنصیب:

جم انسٹال کر سکتے ہیں OpenAI آپ پائپ کا استعمال کرتے ہوئے:

پائپ انسٹال جم

2. بنیادی تصورات:

جم نے کئی بنیادی تصورات متعارف کرائے ہیں OpenAI:

- ماحولیات: ماحول ایک کام یا مسئلہ ہے جس کے ساتھ ایک Python ایجنٹ تعامل کرتا ہے۔ جم میں ماحولیات کو RL کلاسز کے طور پر بیان کیا گیا ہے اور یہ کام کی حرکیات اور قواعد کو سمیٹتے ہیں۔ مثال کے طور پر CartPole، MountainCar، اور Atari گیمز جیسے کلاسک ماحول دستیاب ہیں۔

- ایجنٹ: ایجنٹ سیکھنے والا ہے جو ماحول کے ساتھ تعامل کرتا ہے۔ یہ اقدامات کرتا ہے اور انعامات کی شکل میں RL ایجنٹ رائے حاصل کرتا ہے۔

- مشاہدے کی جگہ: مشاہدے کی جگہ تمام ممکنہ حالتوں کا مجموعہ ہے جسے ایجنٹ ماحول سے محسوس کر سکتا ہے۔ مسئلہ کے لحاظ سے یہ مسلسل یا مجرد ہو سکتا ہے۔

- ایکشن اسپیس: ایکشن اسپیس تمام ممکنہ کارروائیوں کا مجموعہ ہے جو ایجنٹ ماحول میں کر سکتا ہے۔ یہ مسلسل یا مجرد بھی ہو سکتا ہے۔

- انعام: انعام ایک عددی قدر ہے جو ایجنٹ کو ماحول میں کارروائی کرنے کے بعد حاصل ہوتا ہے۔ ایجنٹ کا مقصد وقت کے ساتھ اس کے مجموعی انعام کو زیادہ سے زیادہ کرنا ہے۔

- ایپی سوڈ: ایک واقعہ ایجنٹ اور ماحول کے درمیان ایک ہی دوڑ یا تعامل ہے، جو ابتدائی حالت سے شروع ہوتا ہے اور ٹرمینل حالت تک پہنچنے تک جاری رہتا ہے۔

3. شروع کرنا:

جم استعمال کرنے کے لیے، آپ ایک ماحول بنا کر شروع کریں OpenAI:

جم درآمد کریں

```
env = gym.make('CartPole-v1')
```

4. ماحولیات کے ساتھ تعامل:

آپ ایک سادہ لوپ کا استعمال کرتے ہوئے ماحول کے ساتھ تعامل کر سکتے ہیں۔ ہر مرحلے میں، ایجنٹ ایک عمل کا انتخاب کرتا ہے، اور ماحول اگلی حالت، انعام، اور ایپی سوڈ کے خاتمے کے بارے میں معلومات کے ساتھ جواب دیتا ہے:

```
مشاہدہ = env.reset()

(max_timesteps) کے لیے t رینج میں:
```

(مشاہدہ) agent.select_action = ایکشن

(کارروائی)env.step = مشاہدہ، انعام، مکمل، معلومات

:اگر ہو جائے

توڑنا

:حسب ضرورت ماحول .5

جم آپ کو اپنی مرضی کے مطابق ماحول بنانے کی اجازت دیتا ہے۔ آپ کو ماحولیات کی کلاس کی وضاحت کرنے OpenAI ماحول RL پر عمل پیرا ہو۔ یہ خصوصیت مخصوص تحقیق یا درخواست کے مقاصد کے لیے API کی ضرورت ہے جو جم کے تیار کرنے کے لیے مفید ہے۔

:تشخیص اور تربیت .6

ایجنٹوں کی جانچ اور تربیت کے لیے ایک انٹرفیس فراہم کرتا ہے۔ محققین مختلف کمک سیکھنے کے RL جم OpenAI لائبریریاں، جیسے RL الگورتھم کو جانچنے اور بینچ مارک کرنے کے لیے جم کے ماحول کا استعمال کر سکتے ہیں۔ کئی ایجنٹ کی تربیت کو آسان بنانے کے لیے جم کے ساتھ انضمام کی ،Stable Baselines اور ،PyTorch ،TensorFlow پیشکش کرتی ہیں۔

:مختلف قسم کے ماحول .7

اور CartPole جم میں ماحول کی ایک وسیع رینج شامل ہے، جس میں کلاسک کنٹرول کے مسائل جیسے OpenAI سمولیشنز شامل ہیں۔ Dاور 3 Dگیمز، روبوٹک کنٹرول، اور 2 Atari سے لے کر پیچیدہ ماحول جیسے MountainCar

:کمیونٹی اور ایکسٹینشنز .8

جم میں ایک متحرک کمیونٹی ہے، اور ٹول کٹ کے لیے ایکسٹینشنز، حسب ضرورت ماحول اور ریپرز تلاش کرنا OpenAI عام بات ہے۔ کچھ مشہور ایکسٹینشنز میں ریٹرو گیم ایمولیشن کے لیے جم ریٹرو اور آر ایل ایجنٹس کا استعمال کرتے ہوئے سانپ کھیلنے کے لیے جم سانپ شامل ہیں۔

:تصور .9

جم آپ کو ایجنٹ کے رویے کو دیکھنے کے لیے ماحول فراہم کرنے کی اجازت دیتا ہے۔ یہ ڈیبگ کرنے اور سیکھنے کے عمل کو سمجھنے کے لیے مفید ہو سکتا ہے۔

:بیس لائنز اور لیڈر بورڈز .10

ایجنٹس کی کارکردگی کا موازنہ کرنے کے لیے بینچ مارک مسائل، یا بنیادی خطوط کا مجموعہ فراہم RL جم آپ کے OpenAI کرتا ہے۔ جم ویب سائٹ مختلف ماحول کے لیے لیڈر بورڈز کو بھی برقرار رکھتی ہے، جس سے آپ یہ دیکھ سکتے ہیں کہ مختلف الگورتھم کتنی اچھی کارکردگی کا مظاہرہ کرتے ہیں۔

اوپن اے آئی جم کمک سیکھنے کے الگورتھم کو تیار کرنے اور ان کے ساتھ تجربہ کرنے کے لیے ایک بنیادی ٹول کے طور پر کام کرتا ہے، جس سے مختلف ماحول میں مختلف طریقوں کی کارکردگی کو سمجھنا، جانچنا اور موازنہ کرنا آسان ہو جاتا کے میدان کو آگے بڑھانے اور مختلف ڈومینز میں اس کی ایپلی کیشنز کو فعال کرنے میں اہم کردار ادا کیا RL ہے۔ اس نے ہے۔

XGBoost

XGBoost, short for Extreme Gradient Boosting, is a powerful and efficient machine learning algorithm that falls under the gradient boosting framework. It is widely used for supervised learning tasks such as classification, regression, and ranking. Developed by Tianqi Chen, XGBoost is known for its exceptional predictive performance and speed. Below is a detailed explanation of XGBoost:

Gradient Boosting:

- XGBoost is a gradient boosting algorithm. Gradient boosting is an ensemble learning technique that builds a strong predictive model by combining the predictions of multiple weaker models. It operates by iteratively improving the model's predictive power through the addition of decision trees.

Key Features of XGBoost:

1. **Regularization**:

- XGBoost integrates L1 (Lasso) and L2 (Ridge) regularization terms into the objective function. This helps in controlling overfitting and improving the model's generalization ability.

2. **Gradient Descent Optimization**:

- XGBoost uses a gradient descent optimization technique to minimize the loss function. It updates the model's parameters in a way that reduces the loss function's value, improving the model's accuracy.

3. **Handling Missing Values**:

- XGBoost has a built-in mechanism to handle missing data. It can automatically learn how to handle missing values during training, making it a valuable tool when working with real-world data that often contains missing values.

4. **Parallel and Distributed Computing**:

- XGBoost is designed for efficiency and can utilize parallel and distributed computing to speed up training. It can take advantage of multi-core processors and distributed computing clusters, making it suitable for large datasets.

5. **Tree Pruning**:

- XGBoost performs “pruning” on decision trees to remove splits that don't contribute significantly to improving model performance. This reduces the complexity of the model and prevents overfitting.

6. **Cross-Validation Support**:

- XGBoost provides built-in support for cross-validation, which is crucial for model evaluation and hyperparameter tuning.

7. **Flexibility**:

- It can be used for various types of supervised learning tasks, including classification, regression, ranking, and more. XGBoost can also be used as a component within a broader ensemble learning strategy.

8. **Regular and Sparse Data**:

- XGBoost is compatible with both regular (dense) and sparse data, making it suitable for a wide range of applications, including text mining and recommendation systems.

Components of XGBoost:

1. **Objective Function**: XGBoost supports different objective functions for classification, regression, and ranking tasks. Examples include “reg:linear” for regression, “binary:logistic” for binary classification, and “rank:pairwise” for ranking.

2. **Decision Trees (Weak Learners)**: XGBoost employs decision trees as its base learners, specifically gradient boosted decision trees (GBDTs). These trees are added sequentially to improve the model's predictions.

3. **Boosting Rounds**: Training XGBoost involves specifying the number of boosting rounds (trees) and controlling their depth, learning rate, and other hyperparameters.

4. **Loss Function**: The loss function quantifies the error between the model's predictions and the true values, guiding the optimization process. Common loss functions include mean squared error for regression and log loss for classification.

Usage:

1. **Training**:

- To train an XGBoost model, you provide a dataset with input features and corresponding labels. XGBoost iteratively builds decision trees, minimizing the loss function with respect to the training data.

2. **Prediction**:

- Once trained, you can use the model to make predictions on new, unseen data.

3. **Hyperparameter Tuning**:

- Tuning hyperparameters is an important step in using XGBoost effectively. Common hyperparameters to tune include the learning rate, tree depth, and regularization terms.

4. **Model Evaluation**:

- To assess the model's performance, you can use metrics such as mean squared error (MSE) for regression and accuracy or AUC-ROC for classification tasks.

XGBoost has gained popularity in data science competitions (e.g., Kaggle) and industry applications due to its remarkable predictive performance and versatility. It is an essential tool in the machine learning toolkit, providing state-of-the-art results across a wide range of data science problems.

XGBoost

کے لیے مختصر، ایک طاقتور اور موثر مشین لرننگ الگورتھم ہے جو XGBoost، Extreme Gradient Boosting گریڈینٹ بوسٹنگ فریم ورک کے تحت آتا ہے۔ یہ وسیع پیمانے پر زیر نگرانی سیکھنے کے کاموں جیسے درجہ بندی، رجعت، اپنی غیر معمولی پیشن XGBoost ،کی طرف سے تیار کردہ Tianqi Chen اور درجہ بندی کے لیے استعمال ہوتا ہے۔ :کی تفصیلی وضاحت ہے XGBoost گوئی کی کارکردگی اور رفتار کے لیے جانا جاتا ہے۔ ذیل میں

:گریڈینٹ بڑھانا

ایک گریڈینٹ بوسٹنگ الگورتھم ہے۔ گریڈیئنٹ بوسٹنگ سیکھنے کی ایک ایسی تکنیک ہے جو متعدد کمزور XGBoost - ماڈلز کی پیشین گوئیوں کو ملا کر ایک مضبوط پیش گوئی کرنے والا ماڈل بناتی ہے۔ یہ فیصلہ کن درختوں کے اضافے کے ذریعے ماڈل کی پیشین گوئی کی طاقت کو تکراری طور پر بہتر بنا کر کام کرتا ہے۔

:کی اہم خصوصیات XGBoost

:ریگولرائزیشن .1

ریگولرائزیشن کی شرائط کو معروضی فنکشن میں ضم کرتا ہے۔ اس سے اوور (Ridge) L2 اور (Lasso) L1 XGBoost - فٹنگ کو کنٹرول کرنے اور ماڈل کی عمومی صلاحیت کو بہتر بنانے میں مدد ملتی ہے۔

:تدریجی نزول کی اصلاح .2

نقصان کے فنکشن کو کم سے کم کرنے کے لیے گریڈینٹ ڈیسنٹ آپٹیمائزیشن تکنیک کا استعمال کرتا ہے۔ یہ XGBoost - ماڈل کے پیرامیٹرز کو اس طرح اپ ڈیٹ کرتا ہے جس سے نقصان کے فنکشن کی قدر میں کمی آتی ہے، ماڈل کی درستگی میں بہتری آتی ہے۔

:گمشدہ اقدار کو سنبھالنا .3

میں گمشدہ ڈیٹا کو سنبھالنے کے لیے بلٹ ان میکانزم ہے۔ یہ خود بخود سیکھ سکتا ہے کہ تربیت کے دوران XGBoost - گمشدہ اقدار کو کس طرح سنبھالنا ہے، یہ حقیقی دنیا کے ڈیٹا کے ساتھ کام کرتے وقت اسے ایک قیمتی ٹول بناتا ہے جس میں اکثر گمشدہ اقدار ہوتے ہیں۔

:متوازی اور تقسیم شدہ کمپیوٹنگ .4

کو کارکردگی کے لیے ڈیزائن کیا گیا ہے اور تربیت کو تیز کرنے کے لیے متوازی اور تقسیم شدہ کمپیوٹنگ کا XGBoost - استعمال کر سکتا ہے۔ یہ ملٹی کور پروسیسرز اور تقسیم شدہ کمپیوٹنگ کلسٹرز کا فائدہ اٹھا سکتا ہے، جو اسے بڑے ڈیٹا سیٹس کے لیے موزوں بناتا ہے۔

:درختوں کی کٹائی .5

ان تقسیموں کو دور کرنے کے لیے فیصلہ کن درختوں پر "کاٹنا" کرتا ہے جو ماڈل کی کارکردگی کو بہتر بنانے XGBoost - میں اہم کردار ادا نہیں کرتے ہیں۔ یہ ماڈل کی پیچیدگی کو کم کرتا ہے اور زیادہ فٹنگ کو روکتا ہے۔

:کراس توثیق سپورٹ .6

کراس توثیق کے لیے بلٹ ان سپورٹ فراہم کرتا ہے، جو ماڈل کی تشخیص اور ہائپر پیرامیٹر ٹیوننگ کے لیے اہم XGBoost - ہے۔

:لچک .7

اسے مختلف قسم کے زیر نگرانی سیکھنے کے کاموں کے لیے استعمال کیا جا سکتا ہے، بشمول درجہ بندی، رجعت، درجہ - کو ایک وسیع تر جوڑ سیکھنے کی حکمت عملی کے اندر ایک جزو کے طور پر بھی XGBoost بندی، اور بہت کچھ۔ استعمال کیا جا سکتا ہے۔

8. ریگولر اور اسپارس ڈیٹا:

ریگولر (گھنے) اور ویرل ڈیٹا دونوں کے ساتھ مطابقت رکھتا ہے، جو اسے ٹیکسٹ مائننگ اور سفارشی نظاموں XGBoost - سمیت ایپلی کیشنز کی ایک وسیع رینج کے لیے موزوں بناتا ہے۔

XGBoost کے اجزاء:

1. مقصدی فنکشن: XGBoost درجہ بندی، رجعت، اور درجہ بندی کے کاموں کے لیے مختلف معروضی افعال کی حمایت کرتا ہے۔ مثالوں میں رجعت کے لیے "reg:linear"، بائنری درجہ بندی کے لیے "binary:logistic" اور درجہ بندی کے لیے "rank:pairwise" شامل ہیں۔

2. فیصلہ کن درخت (کمزور سیکھنے والے): XGBoost فیصلہ کرنے والے درختوں کو اپنے بنیادی سیکھنے والوں کے طور پر استعمال کرتا ہے، خاص طور پر گریڈیئنٹ بوسٹڈ ڈیسیسن ٹری (GBDTs) کے لیے ان کی پیشین گوئیوں کو بہتر بنانے کے لیے۔ ماڈل درختوں کو ترتیب وار شامل کیا جاتا ہے۔

.

4. نقصان کا فنکشن: نقصان کا فنکشن ماڈل کی پیشین گوئیوں اور حقیقی اقدار کے درمیان غلطی کو درست کرتا ہے، اصلاح کے عمل کی رہنمائی کرتا ہے۔ عام نقصان کے فنکشنز میں ریگریشن کے لیے اوسط مربع غلطی اور درجہ بندی کے لیے لاگ نقصان شامل ہیں۔

استعمال:

1. تربیت:

ماڈل کو تربیت دینے کے لیے، آپ ان پٹ خصوصیات اور متعلقہ لیبلز کے ساتھ ڈیٹا سیٹ فراہم کرتے ہیں۔ XGBoost - تکراری طور پر فیصلہ سازی کے درخت بناتا ہے، تربیتی ڈیٹا کے حوالے سے نقصان کو کم سے کم کرتا ہے۔ XGBoost

2. پیشین گوئی:

ایک بار تربیت حاصل کرنے کے بعد، آپ نئے، غیر دیکھے ڈیٹا پر پیشین گوئیاں کرنے کے لیے ماڈل کا استعمال کر سکتے - ہیں۔

3. ہائپر پیرامیٹر ٹیوننگ:

کو مؤثر طریقے سے استعمال کرنے کے لیے ہائپر پیرامیٹر کو ٹیون کرنا ایک اہم قدم ہے۔ ٹیون کرنے کے لیے XGBoost - عام ہائپرپیرامیٹر میں سیکھنے کی شرح، درخت کی گہرائی، اور ریگولرائزیشن کی شرائط شامل ہیں۔

4. ماڈل کی تشخیص:

ماڈل کی کارکردگی کا اندازہ لگانے کے لیے، آپ درجہ بندی کے کاموں کے لیے میٹرکس جیسے کہ اوسط مربع غلطی - استعمال کر سکتے ہیں۔ AUC-ROC یا رجعت اور درستگی کے لیے (MSE)

اور انڈسٹری ایپلی کیشنز میں اپنی شاندار پیشن گوئی کی (Kaggle، مثال کے طور پر) نے ڈیٹا سائنس مقابلوں XGBoost کارکردگی اور استعداد کی وجہ سے مقبولیت حاصل کی ہے۔ یہ مشین لرننگ ٹول کٹ میں ایک ضروری ٹول ہے، جو ڈیٹا سائنس کے مسائل کی ایک وسیع رینج میں جدید ترین نتائج فراہم کرتا ہے۔

Hugging Face Transformers

Hugging Face Transformers is an open-source Python library and ecosystem that provides easy access to a wide variety of pre-trained natural language processing (NLP) models, particularly transformer models. It simplifies the process of working with state-of-the-art NLP models for tasks such as text classification, named entity recognition, language generation, question-answering, and more. Here's a detailed explanation of Hugging Face Transformers:

Installation:

You can install Hugging Face Transformers using pip:

pip install transformers

Key Features:

1. **Pre-trained Models**:

- Hugging Face Transformers provides access to a vast library of pre-trained models, including popular architectures like BERT, GPT, RoBERTa, and many others. These models are pre-trained on large corpora of text data and can be fine-tuned for specific NLP tasks.

2. **State-of-the-Art Models**:

- The library offers cutting-edge models and architectures that have achieved top performance in various NLP benchmarks and competitions. This allows practitioners to leverage the latest advancements in the field with ease.

3. **Model Hub**:

- Hugging Face maintains a model hub (https://huggingface.co/models) where you can discover, share, and download pre-trained models and model checkpoints. It hosts a wide range of models contributed by the community.

4. **Model Fine-Tuning**:

- You can fine-tune pre-trained models on your specific NLP tasks with minimal effort. Fine-tuning involves training the model on your dataset to adapt it to a particular task, like sentiment analysis or text generation.

5. **Pipeline API**:

- Transformers provides a high-level API called the “pipeline” for common NLP tasks. It abstracts away the complexities of model loading, tokenization, and inference, making it easy to use pre-trained models for tasks like text classification, named entity recognition, text generation, translation, and more.

6. **Tokenization**:

- The library includes tokenizers for various languages and pre-trained models. It helps you convert text into input that the models can understand and handle efficiently.

7. **Interoperability**:

- Hugging Face Transformers is compatible with PyTorch and TensorFlow, allowing users to work with their preferred deep learning framework.

8. **Community Contributions**:

- The library has a large and active community of contributors, which means continuous updates, enhancements, and the addition of new models. It's also open-source, making it accessible for collaboration and extension.

9. **Inference and Deployment**:

- You can use Hugging Face Transformers to deploy models in production systems, serving predictions over APIs or integrating them into various applications, including chatbots, content generation, and more.

Usage:

1. **Model Loading**:

- You can load pre-trained models from the Hugging Face model hub by specifying the model name or path. For example:

from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(“bert-base-uncased”)

2. **Tokenization**:

- Use the model's associated tokenizer to convert input text into tokens and prepare it for model input:

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(“bert-base-uncased”) inputs = tokenizer(“Hello, how are you?”, return_tensors=“pt”)

3. **Inference**:

- Pass tokenized input to the model for inference:

outputs = model(**inputs)

4. **Fine-Tuning**:

- To fine-tune a pre-trained model for a specific task, load the model, set up the training loop, and fine-tune on your dataset. Transformers provides utilities for fine-tuning.

5. **Pipeline API**:

- Use the pipeline API for quick and easy access to various NLP tasks:

from transformers import pipeline

nlp = pipeline(“sentiment-analysis”)

result = nlp(“I love this product!”)

Community and Ecosystem:

Hugging Face Transformers is part of a broader ecosystem that includes additional libraries like Transformers Tokenizers, Accelerated Inference API, and more. This ecosystem is designed to facilitate NLP research and applications, making it easier for researchers and practitioners to work with state-of-the-art models and create innovative NLP solutions.

گلے لگانا چہرہ ٹرانسفارمرز

لائبریری اور ایکو سسٹم ہے جو پہلے سے تربیت یافتہ Python ایک اوپن سورس Hugging Face Transformers ماڈلز، خاص طور پر ٹرانسفارمر ماڈلز کی وسیع اقسام تک آسان رسائی فراہم کرتا ہے۔ یہ (NLP) قدرتی لینگویج پروسیسنگ ماڈلز کے ساتھ کام کرنے کے عمل کو آسان بناتا ہے جیسے کہ ٹیکسٹ کی درجہ بندی، نام کی ہستی کی NLP جدید ترین :شناخت، زبان کی تخلیق، سوال کا جواب دینا، اور بہت کچھ۔ ہگنگ فیس ٹرانسفارمرز کی تفصیلی وضاحت یہ ہے

:تنصیب

:آپ پائپ کا استعمال کرتے ہوئے ہیگنگ فیس ٹرانسفارمرز انسٹال کر سکتے ہیں

پائپ انسٹال ٹرانسفارمرز

:اہم خصوصیات

:پہلے سے تربیت یافتہ ماڈلز .1

پہلے سے تربیت یافتہ ماڈلز کی ایک وسیع لائبریری تک رسائی فراہم کرتا ہے، Hugging Face Transformers -
اور بہت سے دوسرے جیسے مشہور فن تعمیرات۔ یہ ماڈل ٹیکسٹ ڈیٹا کے بڑے کارپورا ،BERT، GPT، RoBERTa بشمول پر پہلے سے تربیت یافتہ ہیں اور مخصوص این ایل پی کاموں کے لیے ٹھیک بنائے جا سکتے ہیں۔

:جدید ترین ماڈلز .2

بینچ مارکس اور مقابلوں میں اعلیٰ NLP لائبریری جدید ترین ماڈلز اور فن تعمیرات پیش کرتی ہے جنہوں نے مختلف -
کارکردگی حاصل کی ہے۔ یہ پریکٹیشنرز کو آسانی کے ساتھ میدان میں تازہ ترین پیشرفت کا فائدہ اٹھانے کی اجازت دیتا
ہے۔

:ماڈل حب .3

کو برقرار رکھتا ہے جہاں آپ پہلے سے تربیت (https://huggingface.co/models) ایک ماڈل ہب Hugging Face -
یافتہ ماڈلز اور ماڈل چیک پوائنٹس کو دریافت، اشتراک اور ڈاؤن لوڈ کر سکتے ہیں۔ یہ کمیونٹی کی طرف سے تعاون کردہ
ماڈلز کی ایک وسیع رینج کی میزبانی کرتا ہے۔

:ماڈل فائن ٹیوننگ .4

کاموں پر پہلے سے تربیت یافتہ ماڈلز کو ٹھیک کر سکتے ہیں۔ فائن NLP آپ کم سے کم کوشش کے ساتھ اپنے مخصوص -
ٹیوننگ میں آپ کے ڈیٹاسیٹ پر ماڈل کو تربیت دینا شامل ہے تاکہ وہ اسے کسی خاص کام کے مطابق ڈھال سکے، جیسے
جذبات کا تجزیہ یا ٹیکسٹ جنریشن۔

:API پائپ لائن .5

فراہم کرتے ہیں جسے "پائپ لائن" کہا جاتا ہے۔ یہ ماڈل API کاموں کے لیے ایک اعلیٰ سطحی NLP ٹرانسفارمرز عام -
لوڈنگ، ٹوکنائزیشن، اور انفرنس کی پیچیدگیوں کو دور کرتا ہے، جس سے ٹیکسٹ کی درجہ بندی، نام کی ہستی کی شناخت،
ٹیکسٹ جنریشن، ترجمہ وغیرہ جیسے کاموں کے لیے پہلے سے تربیت یافتہ ماڈلز کو استعمال کرنا آسان ہو جاتا ہے۔

:ٹوکنائزیشن .6

لائبریری میں مختلف زبانوں اور پہلے سے تربیت یافتہ ماڈلز کے ٹوکنائزر شامل ہیں۔ یہ آپ کو متن کو ان پٹ میں تبدیل -
کرنے میں مدد کرتا ہے جسے ماڈل سمجھ سکتے ہیں اور مؤثر طریقے سے سنبھال سکتے ہیں۔

:انٹرآپریبلٹی .7

کے ساتھ مطابقت رکھتا ہے، جو صارفین کو اپنے TensorFlow اور PyTorch Hugging Face Transformers -
ترجیحی ڈیپ لرننگ فریم ورک کے ساتھ کام کرنے کی اجازت دیتا ہے۔

8. کمیونٹی کے تعاون:

لائبریری میں شراکت داروں کی ایک بڑی اور فعال کمیونٹی ہے، جس کا مطلب ہے مسلسل اپ ڈیٹس، اضافہ، اور نئے ماڈلز -
کا اضافہ۔ یہ اوپن سورس بھی ہے، جو اسے تعاون اور توسیع کے لیے قابل رسائی بناتا ہے۔

9. تخمینہ اور تعیناتی:

پر پیشین گوئیاں پیش کرنے یا APIs ،کو پروڈکشن سسٹمز میں ماڈلز کی تعیناتی Hugging Face Transformers آپ -
انہیں مختلف ایپلی کیشنز بشمول چیٹ بوٹس، مواد کی تیاری، اور مزید میں ضم کرنے کے لیے استعمال کر سکتے ہیں۔

استعمال:

1. ماڈل لوڈنگ:

آپ ماڈل کا نام یا راستہ بتا کر ہیگنگ فیس ماڈل ہب سے پہلے سے تربیت یافتہ ماڈل لوڈ کر سکتے ہیں۔ مثال کے طور پر: -
درآمد کریں۔ AutoModelForSequenceClassification ٹرانسفارمرز سے

ماڈل = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased")

2. ٹوکنائزیشن:

ان پٹ ٹیکسٹ کو ٹوکن میں تبدیل کرنے اور اسے ماڈل ان پٹ کے لیے تیار کرنے کے لیے ماڈل کے منسلک ٹوکنائزر کا -
استعمال کریں:

ٹرانسفارمرز سے آٹو ٹوکنائزر درآمد کریں۔

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased") inputs = tokenizer("ہیلو، آپ
کیسے ہیں؟"، return_tensors="pt")

3. اندازہ:

اندازہ لگانے کے لیے ٹوکنائزڈ ان پٹ ماڈل کو دیں: -

آؤٹ پٹ = ماڈل (** ان پٹ)

4. فائن ٹیوننگ:

کسی مخصوص کام کے لیے پہلے سے تربیت یافتہ ماڈل کو ٹھیک کرنے کے لیے، ماڈل کو لوڈ کریں، ٹریننگ لوپ سیٹ اپ -
کریں، اور اپنے ڈیٹا سیٹ پر فائن ٹیون کریں۔ ٹرانسفارمرز فائن ٹیوننگ کے لیے افادیت فراہم کرتے ہیں۔

5. پائپ لائن API:

کا استعمال کریں API کے مختلف کاموں تک فوری اور آسان رسائی کے لیے پائپ لائن NLP -

ٹرانسفارمرز سے پائپ لائن درآمد کریں۔

پائپ لائن ("جذبات کا تجزیہ") = nlp

نتیجہ = nlp ("مجھے یہ پروڈکٹ پسند ہے!")

کمیونٹی اور ماحولیاتی نظام:

ایک وسیع تر ماحولیاتی نظام کا حصہ ہے جس میں اضافی لائبریریاں شامل ہیں جیسے Hugging Face Transformers تحقیق اور NLP اور بہت کچھ۔ یہ ماحولیاتی نظام ،Accelerated Inference API، Transformers Tokenizers ایپلی کیشنز کو سہولت فراہم کرنے کے لیے ڈیزائن کیا گیا ہے، جس سے محققین اور پریکٹیشنرز کے لیے جدید ترین ماڈلز کے حل تیار کرنا آسان ہو جاتا ہے۔ NLP ساتھ کام کرنا اور جدید

CatBoost

CatBoost is a high-performance, open-source gradient boosting library designed for machine learning tasks, particularly in the field of tabular data, where it excels at handling categorical features. Developed by Yandex, CatBoost stands out for its effectiveness in producing accurate models with minimal hyperparameter tuning. Here's a detailed explanation of CatBoost:

Installation:

You can install CatBoost using pip:

pip install catboost

Key Features:

1. **Categorical Feature Handling**:

- CatBoost is optimized for handling categorical features out of the box. It can efficiently process categorical data without the need for extensive preprocessing like one-hot encoding or label encoding. This simplifies the feature engineering process and reduces the risk of data leakage.

2. **Gradient Boosting Algorithm**:

- CatBoost is based on the gradient boosting framework, which combines multiple decision trees to build a strong predictive model. It optimizes the boosting process by adapting to the distribution of the target variable.

3. **Robust to Overfitting**:

- CatBoost incorporates regularization techniques that make it robust to overfitting. It has built-in mechanisms to control tree complexity and ensemble size, reducing the risk of producing overly complex models.

4. **Efficient Training**:

- CatBoost is designed for efficient training and can take advantage of multiple CPU cores for parallel processing. It also provides support for training on GPU, which significantly accelerates training time, making it suitable for large datasets.

5. **Automated Hyperparameter Tuning**:

- CatBoost offers an efficient method for hyperparameter tuning. It includes a built-in hyperparameter optimization tool called CatBoost AutoTune, which can help you find the best set of hyperparameters with minimal manual intervention.

6. **Metrics and Evaluation**:

- CatBoost supports a wide range of evaluation metrics for both classification and regression tasks. This includes common metrics like accuracy, F1-score, and log loss for classification, as well as RMSE and MAE for regression.

7. **Support for Ranking Tasks**:

- CatBoost can be used for ranking problems, such as recommendation systems, by optimizing ranking-specific metrics like NDCG (Normalized Discounted Cumulative Gain).

8. **Interpretable Models**:

- CatBoost provides tools for model interpretation, allowing you to analyze feature importance and visualize decision boundaries. This helps you gain insights into the model's behavior.

9. **Cross-Validation Support**:

- CatBoost includes functions for cross-validation, which is essential for model evaluation and assessment of its generalization performance.

10. **Integration with Popular Libraries**:

- CatBoost can be integrated with popular Python libraries like Pandas, Scikit-Learn, and XGBoost, making it compatible with your existing data analysis and machine learning workflows.

Usage:

1. **Data Preparation**:

- Load and preprocess your data. CatBoost's efficient handling of categorical features means that you don't need to perform extensive encoding of these features.

2. **Model Training**:

- Initialize a CatBoost model, set the hyperparameters (e.g., learning rate, depth, and iterations), and train the model on your data.

from catboost import CatBoostClassifier

model = CatBoostClassifier(iterations=1000, learning_rate=0.1) model.fit(X_train, y_train)

3. **Prediction**:

- Once trained, you can use the model to make predictions on new data:

y_pred = model.predict(X_test)

4. **Evaluation**:

- Evaluate the model's performance using appropriate metrics and visualizations.

5. **Hyperparameter Tuning**:

- You can use CatBoost AutoTune or traditional hyperparameter tuning techniques to optimize your model.

CatBoost is a powerful library for gradient boosting that stands out for its efficient handling of categorical features, robustness against overfitting, and ease of use. It's well-suited for a wide range of machine learning tasks, especially when dealing with tabular data.

کیٹ بوسٹ

ایک اعلی کارکردگی، اوپن سورس گریڈینٹ کو فروغ دینے والی لائبریری ہے جسے مشین لرننگ کے کاموں کے CatBoost لیے ڈیزائن کیا گیا ہے، خاص طور پر ٹیبلر ڈیٹا کے شعبے میں، جہاں یہ واضح خصوصیات کو سنبھالنے میں سبقت لے جاتا کم سے کم ہائپر پیرامیٹر ٹیوننگ کے ساتھ درست ماڈل تیار کرنے میں CatBoost ،کی طرف سے تیار کردہ Yandex ہے۔ :کی ایک تفصیلی وضاحت ہے CatBoost اپنی تاثیر کے لیے نمایاں ہے۔ یہاں

:تنصیب

:انسٹال کرسکتے ہیں CatBoost آپ پائپ کا استعمال کرکے

```
pip install catboost
```

:اہم خصوصیات

:دوٹوک فیچر ہینڈلنگ .1

کیٹ بوسٹ کو باکس سے باہر کی خصوصیات کو ہینڈل کرنے کے لیے بہتر بنایا گیا ہے۔ یہ ون ہاٹ انکوڈنگ یا لیبل - انکوڈنگ جیسے وسیع پری پروسیسنگ کی ضرورت کے بغیر کلیدی ڈیٹا پر موثر طریقے سے کارروائی کر سکتا ہے۔ یہ فیچر انجینئرنگ کے عمل کو آسان بناتا ہے اور ڈیٹا لیک ہونے کا خطرہ کم کرتا ہے۔

:گریڈینٹ بوسٹنگ الگورتھم .2

گریڈینٹ بوسٹنگ فریم ورک پر مبنی ہے، جو ایک مضبوط پیشن گوئی ماڈل بنانے کے لیے متعدد فیصلہ کن CatBoost - درختوں کو یکجا کرتا ہے۔ یہ ہدف متغیر کی تقسیم کے مطابق ڈھال کر فروغ دینے کے عمل کو بہتر بناتا ہے۔

:اوور فٹنگ کے لیے مضبوط .3

کیٹ بوسٹ میں ریگولرائزیشن کی تکنیکیں شامل ہیں جو اسے زیادہ فٹنگ کے لیے مضبوط بناتی ہیں۔ اس میں درختوں - کی پیچیدگی اور جوڑ کے سائز کو کنٹرول کرنے کے لیے بلٹ ان میکانزم ہیں، جس سے ضرورت سے زیادہ پیچیدہ ماڈل تیار کرنے کا خطرہ کم ہوتا ہے۔

:موثر تربیت .4

کور کا فائدہ اٹھا سکتا CPU کو موثر تربیت کے لیے ڈیزائن کیا گیا ہے اور متوازی پروسیسنگ کے لیے متعدد CatBoost - پر تربیت کے لیے بھی معاونت فراہم کرتا ہے، جو تربیت کے وقت کو نمایاں طور پر تیز کرتا ہے، جو اسے بڑے GPU ہے۔ یہ ڈیٹا سیٹس کے لیے موزوں بناتا ہے۔

:خودکار ہائپر پیرامیٹر ٹیوننگ .5

کیٹ بوسٹ ہائپر پیرامیٹر ٹیوننگ کے لیے ایک موثر طریقہ پیش کرتا ہے۔ اس میں بلٹ ان ہائپر پیرامیٹر آپٹیمائزیشن ٹول - کہا جاتا ہے، جو آپ کو کم سے کم دستی مداخلت کے ساتھ ہائپر پیرامیٹر کا CatBoost AutoTune شامل ہے جسے بہترین سیٹ تلاش کرنے میں مدد کر سکتا ہے۔

:میٹرکس اور تشخیص .6

درجہ بندی اور رجعت دونوں کاموں کے لیے وسیع پیمانے پر تشخیصی میٹرکس کی حمایت کرتا ہے۔ اس میں CatBoost - شامل MAE اور RMSE اسکور، اور درجہ بندی کے لیے لاگ نقصان، نیز رجعت کے لیے-F1 ،عام میٹرکس جیسے درستگی ہیں۔

:درجہ بندی کے کاموں کے لیے معاونت .7

کیٹ بوسٹ کو درجہ بندی کے مسائل کے لیے استعمال کیا جا سکتا ہے، جیسے کہ سفارشی نظام، درجہ بندی کے لیے - کو بہتر بنا کر۔ (نارملائزڈ ڈسکاؤنٹڈ کمولیٹیو گین) NDCG مخصوص میٹرکس جیسے

:قابل تشریح ماڈلز .8

کیٹ بوسٹ ماڈل کی تشریح کے لیے ٹولز فراہم کرتا ہے، جس سے آپ کو خصوصیت کی اہمیت کا تجزیہ کرنے اور فیصلے - کی حدود کا تصور کرنے کی اجازت ملتی ہے۔ اس سے آپ کو ماڈل کے رویے کے بارے میں بصیرت حاصل کرنے میں مدد ملتی ہے۔

:کراس توثیق سپورٹ .9

میں کراس توثیق کے فنکشنز شامل ہیں، جو ماڈل کی تشخیص اور اس کی عمومی کارکردگی کی تشخیص کے CatBoost - لیے ضروری ہے۔

:مقبول لائبریریوں کے ساتھ انضمام .10

لائبریریوں کے ساتھ مربوط کیا جا سکتا Python جیسی مشہور XGBoost اور ،Scikit-Learn ،کو پنڈاس CatBoost - ہے، جو اسے آپ کے موجودہ ڈیٹا کے تجزیہ اور مشین لرننگ ورک فلو کے ساتھ ہم آہنگ بناتا ہے۔

:استعمال

:ڈیٹا کی تیاری .1

کی واضح خصوصیات کی موثر ہینڈلنگ کا مطلب ہے کہ آپ کو ان CatBoost اپنے ڈیٹا کو لوڈ اور پری پروسیس کریں۔ - خصوصیات کی وسیع انکوڈنگ کرنے کی ضرورت نہیں ہے۔

:ماڈل ٹریننگ .2

کیٹ بوسٹ ماڈل شروع کریں، ہائپر پیرامیٹرس (جیسے سیکھنے کی شرح، گہرائی، اور تکرار) سیٹ کریں، اور ماڈل کو اپنے - ڈیٹا پر تربیت دیں۔

سے CatBoostClassifier درآمد catboost

ماڈل = CatBoostClassifier(iterations=1000, learning_rate=0.1) model.fit(X_train, y_train)

:پیشین گوئی .3

:ایک بار تربیت حاصل کرنے کے بعد، آپ نئے ڈیٹا پر پیشین گوئیاں کرنے کے لیے ماڈل کا استعمال کر سکتے ہیں -

y_pred = model.predict(X_test)

:تشخیص .4

مناسب میٹرکس اور تصورات کا استعمال کرتے ہوئے ماڈل کی کارکردگی کا اندازہ کریں۔ -

:ہائپر پیرامیٹر ٹیوننگ .5

یا روایتی ہائپر پیرامیٹر ٹیوننگ تکنیک استعمال کر سکتے CatBoost AutoTune آپ اپنے ماڈل کو بہتر بنانے کے لیے - ہیں۔

گریڈیئنٹ بوسٹنگ کے لیے ایک طاقتور لائبریری ہے جو اس کی واضح خصوصیات، زیادہ فٹنگ کے خلاف CatBoost مضبوطی، اور استعمال میں آسانی کے لیے نمایاں ہے۔ یہ مشین لرننگ کے کاموں کی ایک وسیع رینج کے لیے موزوں ہے، خاص طور پر جب ٹیبلر ڈیٹا سے نمٹنا ہو۔

PyTorch

PyTorch is an open-source deep learning framework that has gained immense popularity in the machine learning and artificial intelligence communities. It is known for its flexibility, dynamic computation graph, and extensive support for neural networks.

Tensors:

At the core of PyTorch are tensors, which are multi-dimensional arrays similar to NumPy arrays. Tensors are fundamental data structures used for representing and manipulating data in PyTorch.

Key Features:

1. **Dynamic Computational Graph**:

PyTorch uses a dynamic computational graph, also known as a define-by-run approach. This means that the graph is built on the fly as operations are executed, enabling more flexibility in defining and modifying models during runtime. This is in contrast to static computational graphs used in frameworks like TensorFlow.

2. **Automatic Differentiation**:

One of PyTorch's standout features is its automatic differentiation system. The `autograd` module tracks operations performed on tensors and automatically computes gradients. This is crucial for training deep learning models using gradient-based optimization techniques.

3. **Neural Networks and Deep Learning**:

PyTorch provides a comprehensive set of tools for building and training neural networks. It includes modules for defining layers, activation functions, loss functions, and optimizers.

4. **GPU Acceleration**:

PyTorch fully supports GPU acceleration, making it possible to train deep learning models on CUDA-enabled GPUs. This speeds up training significantly and is essential for working with large models and datasets.

5. **Model Building**:

PyTorch makes it easy to define custom neural network architectures by subclassing the `nn.Module` class. This flexibility allows you to create complex models and experiment with various architectural designs.

6. **Data Loading and Transformation**:

PyTorch provides tools for efficiently loading and preprocessing data using the `DataLoader` and `Dataset` classes. This is crucial for handling large datasets and setting up data pipelines for training.

7. **Community and Ecosystem**:

PyTorch has a thriving community of researchers, developers, and users. It is widely adopted in the academic and research communities, and many state-of-the-art research papers release code and models implemented in PyTorch. Additionally, there is a rich ecosystem of libraries and tools built on top of PyTorch, such as fastai, transformers, and detectron2.

8. **Visualization and Debugging**:

PyTorch integrates with popular visualization libraries like TensorBoard for monitoring training and visualization of model performance. It also provides utilities for debugging, such as custom gradient computation and a debugger.

9. **ONNX Integration**:

PyTorch has native support for the Open Neural Network Exchange (ONNX) format, allowing users to export models to a format that can be used in other deep learning frameworks.

Usage:

1. **Installation**:

Install PyTorch using pip, specifying the version and CUDA support according to your system's configuration.

2. **Tensors and Operations**:

Start by creating tensors and performing mathematical operations on them, similar to how you would use NumPy.

3. **Model Definition**:

Define your neural network model by creating a custom class that inherits from `nn.Module`. This class should include the model architecture and forward pass method.

4. **Loss and Optimizer**:

Specify the loss function and optimizer for training. PyTorch provides a variety of loss functions and optimization algorithms to choose from.

5. **Data Loading**:

Prepare your data and create data loaders to efficiently load and preprocess data for training.

6. **Training Loop**:

Implement the training loop, which includes forward and backward passes, gradient updates, and model evaluation.

7. **Visualization and Debugging**:

Use visualization tools like TensorBoard or PyTorch's built-in functionalities for monitoring and debugging.

8. **Inference**:

Once your model is trained, you can use it for inference on new data by passing it through the trained model.

PyTorch's flexibility, dynamic computation graph, and strong support for research make it a favored choice for both academic and industrial applications in the deep learning and machine learning fields. Its vibrant community and ecosystem continue to drive innovation and development in the field of AI.

پائی ٹارچ

ایک اوپن سورس ڈیپ لرننگ فریم ورک ہے جس نے مشین لرننگ اور مصنوعی ذہانت کی کمیونٹیز میں بے پناہ PyTorch مقبولیت حاصل کی ہے۔ یہ اپنی لچک، متحرک کمپیوٹیشن گراف، اور نیورل نیٹ ورکس کے لیے وسیع تعاون کے لیے جانا جاتا ہے۔

:ٹینسر

کی طرح کثیر جہتی صفیں ہیں۔ ٹینسر بنیادی ڈیٹا ڈھانچے NumPy arrays ہیں، جو tensors کے مرکز میں PyTorch میں ڈیٹا کی نمائندگی اور ہیرا پھیری کے لیے استعمال ہوتے ہیں۔ PyTorch ہیں جو

:اہم خصوصیات

:متحرک کمپیوٹیشنل گراف .1

ایک متحرک کمپیوٹیشنل گراف استعمال کرتا ہے، جسے ڈیفائن بائی رن اپروچ بھی کہا جاتا ہے۔ اس کا مطلب یہ PyTorch ہے کہ گراف کو فلائی پر بنایا گیا ہے جیسا کہ آپریشنز انجام پاتے ہیں، رن ٹائم کے دوران ماڈلز کی وضاحت اور ان میں ترمیم جیسے فریم ورک میں استعمال ہونے والے جامد کمپیوٹیشنل گراف TensorFlow کرنے میں مزید لچک پیدا کرتے ہیں۔ یہ کے برعکس ہے۔

:خودکار تفریق .2

کی نمایاں خصوصیات میں سے ایک اس کا خودکار تفریق کا نظام ہے۔ 'آٹو گراڈ' ماڈیول ٹینسر پر کیے گئے آپریشنز PyTorch کو ٹریک کرتا ہے اور خود بخود گریڈیئنٹس کی گنتی کرتا ہے۔ یہ تدریجی بنیاد پر اصلاح کی تکنیکوں کا استعمال کرتے ہوئے گہری سیکھنے کے ماڈلز کی تربیت کے لیے اہم ہے۔

:اعصابی نیٹ ورکس اور گہری تعلیم .3

نیورل نیٹ ورکس کی تعمیر اور تربیت کے لیے ٹولز کا ایک جامع سیٹ فراہم کرتا ہے۔ اس میں پرتوں کی PyTorch وضاحت کے لیے ماڈیولز، ایکٹیویشن کے افعال، نقصان کے افعال، اور اصلاح کار شامل ہیں۔

:ایکسلریشن GPU .4

پر ڈیپ لرننگ GPUs سے چلنے والے CUDA ایکسلریشن کو مکمل طور پر سپورٹ کرتا ہے، جس سے PyTorch GPU ماڈلز کی تربیت ممکن ہو جاتی ہے۔ یہ تربیت کو نمایاں طور پر تیز کرتا ہے اور بڑے ماڈلز اور ڈیٹا سیٹس کے ساتھ کام کرنے کے لیے ضروری ہے۔

:ماڈل بلڈنگ .5

کلاس کو ذیلی کلاس کرکے حسب ضرورت نیورل نیٹ ورک آرکیٹیکچرز کی وضاحت کرنا آسان 'PyTorch 'nn.Module بناتا ہے۔ یہ لچک آپ کو پیچیدہ ماڈل بنانے اور مختلف آرکیٹیکچرل ڈیزائن کے ساتھ تجربہ کرنے کی اجازت دیتی ہے۔

:ڈیٹا لوڈنگ اور تبدیلی .6

کلاسز کا استعمال کرتے ہوئے ڈیٹا کو مؤثر طریقے سے لوڈ کرنے اور پری 'Dataset' اور 'PyTorch 'DataLoader پروسیسنگ کے لیے ٹولز فراہم کرتا ہے۔ یہ بڑے ڈیٹا سیٹس کو سنبھالنے اور ٹریننگ کے لیے ڈیٹا پائپ لائنز ترتیب دینے کے لیے بہت ضروری ہے۔

:کمیونٹی اور ماحولیاتی نظام .7

میں محققین، ڈویلپرز اور صارفین کی ایک فروغ پزیر کمیونٹی ہے۔ اسے علمی اور تحقیقی برادریوں میں بڑے PyTorch میں نافذ کردہ کوڈ اور ماڈلز جاری کرتے ہیں۔ PyTorch پیمانے پر اپنایا جاتا ہے، اور بہت سے جدید ترین تحقیقی مقالے

کے اوپر بنے ہوئے لائبریریوں اور ٹولز کا ایک بھرپور ماحولیاتی نظام موجود ہے، جیسے فاسٹائی، PyTorch ،مزید برآں ٹرانسفارمرز، اور ڈیٹیکٹرون2۔

:تصور اور ڈیبگنگ .8

جیسی مشہور ویژولائزیشن TensorBoard ٹریننگ کی نگرانی اور ماڈل کی کارکردگی کے تصور کے لیے PyTorch لائبریریوں کے ساتھ مربوط ہے۔ یہ ڈیبگنگ کے لیے یوٹیلیٹیز بھی فراہم کرتا ہے، جیسے کسٹم گریڈینٹ کمپیوٹیشن اور ڈیبگر۔

:انٹیگریشن ONNX .9

فارمیٹ کے لیے مقامی حمایت حاصل ہے، جس سے صارفین کو (ONNX) کو اوپن نیورل نیٹ ورک ایکسچینج PyTorch ایسے فارمیٹ میں ماڈلز ایکسپورٹ کرنے کی اجازت ملتی ہے جو دوسرے ڈیپ لرننگ فریم ورک میں استعمال کیے جاسکتے ہیں۔

:استعمال

:تنصیب .1

کا استعمال کرتے ہوئے pip سپورٹ کی وضاحت کرتے ہوئے CUDA اپنے سسٹم کی کنفیگریشن کے مطابق ورژن اور انسٹال کریں۔ PyTorch

:ٹینسر اور آپریشنز .2

استعمال کریں گے۔ NumPy ٹینسر بنا کر شروع کریں اور ان پر ریاضی کے عمل کو انجام دیں، جیسا کہ آپ

:ماڈل کی تعریف .3

سے وراثت میں ملتی `nn.Module` ایک حسب ضرورت کلاس بنا کر اپنے نیورل نیٹ ورک ماڈل کی وضاحت کریں جو ہے۔ اس کلاس میں ماڈل آرکیٹیکچر اور فارورڈ پاس کا طریقہ شامل ہونا چاہیے۔

:نقصان اور اصلاح کرنے والا .4

مختلف قسم کے نقصان کے افعال اور PyTorch تربیت کے لیے نقصان کی تقریب اور اصلاح کار کی وضاحت کریں۔ آپٹیمائزیشن الگورتھم فراہم کرتا ہے جس میں سے انتخاب کیا جاسکتا ہے۔

:ڈیٹا لوڈنگ .5

اپنا ڈیٹا تیار کریں اور ڈیٹا لوڈرز بنائیں تاکہ ٹریننگ کے لیے ڈیٹا کو مؤثر طریقے سے لوڈ اور پری پروسیس کیا جا سکے۔

:ٹریننگ لوپ .6

ٹریننگ لوپ کو لاگو کریں، جس میں فارورڈ اور بیکورڈ پاسز، گریڈینٹ اپ ڈیٹس، اور ماڈل کی تشخیص شامل ہے۔

:تصور اور ڈیبگنگ .7

کی بلٹ ان فنکشنلٹیز استعمال PyTorch یا TensorBoard مانیٹرنگ اور ڈیبگنگ کے لیے ویژولائزیشن ٹولز جیسے کریں۔

:اندازہ .8

ایک بار جب آپ کا ماڈل تربیت یافتہ ہو جاتا ہے، تو آپ اسے تربیت یافتہ ماڈل سے گزر کر نئے ڈیٹا کا اندازہ لگانے کے لیے استعمال کر سکتے ہیں۔

کی لچک، متحرک کمپیوٹیشن گراف، اور تحقیق کے لیے مضبوط تعاون اسے گہری سیکھنے اور مشین لرننگ کے PyTorch شعبوں میں علمی اور صنعتی ایپلی کیشنز دونوں کے لیے ایک پسندیدہ انتخاب بناتا ہے۔ اس کی متحرک کمیونٹی اور کے میدان میں جدت اور ترقی کو آگے بڑھا رہا ہے۔ AI ماحولیاتی نظام

GLM PyTorch

Generalized Linear Models (GLMs) in PyTorch are a class of models that extend traditional linear regression to handle a wide range of data types and modeling scenarios. PyTorch is a popular deep learning framework, but it also provides support for traditional machine learning tasks, including GLMs. Here's a detailed explanation of GLMs in PyTorch:

Generalized Linear Models (GLMs):

- Generalized Linear Models are a class of statistical models used for regression and classification tasks. They extend linear regression by accommodating a variety of probability distributions for the response variable and link functions to model non-linear relationships. GLMs are particularly useful when dealing with non-Gaussian and non-continuous data.

Key Components of GLMs:

1. **Linear Predictor**:

- The linear predictor in GLMs is similar to the linear regression model. It represents the relationship between the independent variables (predictors) and the expected value of the dependent variable. However, it is usually transformed using a link function.

2. **Link Function**:

- The link function is a non-linear function that connects the linear predictor to the expected value of the response variable. It ensures that the predicted values are within the appropriate range for the distribution of the response variable. Common link functions include the logit function for logistic regression, the log function for Poisson regression, and the identity function for linear regression.

3. **Probability Distribution**:

- GLMs allow you to specify the probability distribution of the response variable. Depending on the nature of your data, you can choose distributions such as Gaussian, Binomial, Poisson, and more.

4. **Loss Function**:

- The loss function in a GLM quantifies the difference between the predicted values and the actual observations. It is specific to the chosen probability distribution and link function.

PyTorch Implementation:

PyTorch provides a versatile framework for implementing GLMs due to its ability to define custom loss functions, models, and optimization algorithms. Here's how to implement a GLM in PyTorch:

1. **Data Preparation**:

- Prepare your data, including the independent variables (predictors) and the dependent variable (response).

2. **Model Definition**:

- Define your GLM model. This typically involves defining a custom model class that inherits from `nn.Module`. In this class, you specify the linear predictor, link function, and any additional components required for your specific GLM.

3. **Loss Function**:

- Create a custom loss function that represents the appropriate loss for your GLM. The loss function should take the model's predictions and the true values as input and compute the loss based on the chosen probability distribution and link function.

4. **Optimization**:

- Choose an optimization algorithm (e.g., stochastic gradient descent, Adam) and train the GLM using your custom loss function. This process involves iteratively updating the model's parameters to minimize the loss.

Usage Examples:

- Here are some common use cases for GLMs in PyTorch:

1. **Linear Regression**: You can implement simple linear regression by using Gaussian distribution and the identity link function.

2. **Logistic Regression**: For binary classification, you can use the logistic (logit) link function and the Binomial distribution.

3. **Poisson Regression**: When dealing with count data, Poisson regression is used with the log link function.

4. **Ordinal Regression**: Ordinal regression models, such as proportional odds models, can be implemented by customizing the link function and the loss function.

Conclusion:

GLMs in PyTorch provide a flexible and customizable framework for implementing a wide range of regression and classification models. With the ability to define custom loss functions and models, PyTorch is well-suited for handling non-Gaussian data and specialized modeling scenarios where traditional linear regression may not suffice.

جی ایل ایم پائی ٹارچ

ماڈلز کی ایک کلاس ہیں جو ڈیٹا کی اقسام اور ماڈلنگ کے منظرناموں کی (GLMs) میں جنرلائزڈ لائنر ماڈلز PyTorch ایک مقبول ڈیپ لرننگ فریم ورک PyTorch ایک وسیع رینج کو سنبھالنے کے لیے روایتی لکیری رجعت کو بڑھاتے ہیں۔ کی GLMs میں PyTorch ۔GLMs ہے، لیکن یہ مشین لرننگ کے روایتی کاموں کے لیے بھی مدد فراہم کرتا ہے، بشمول :تفصیلی وضاحت یہ ہے

:(GLMs) عام لکیری ماڈلز

جنرلائزڈ لائنر ماڈل شماریاتی ماڈلز کی ایک کلاس ہیں جو رجعت اور درجہ بندی کے کاموں کے لیے استعمال ہوتے ہیں۔ وہ - ردعمل متغیر کے لیے مختلف قسم کے امکانی تقسیموں کو ایڈجسٹ کرکے لکیری رجعت کو بڑھاتے ہیں اور غیر لکیری خاص طور پر اس وقت کارآمد ہوتے ہیں جب غیر گاوسی اور غیر GLMs تعلقات کے ماڈل کے لیے افعال کو جوڑتے ہیں۔ مسلسل ڈیٹا سے نمٹتے ہیں۔

:کے اہم اجزاء GLMs

:لکیری پیشن گوئی کرنے والا .1

میں لکیری پیشن گوئی لکیری ریگریشن ماڈل کی طرح ہے۔ یہ آزاد متغیر (پیش گوئی کرنے والے) اور منحصر متغیر GLMs - کی متوقع قدر کے درمیان تعلق کی نمائندگی کرتا ہے۔ تاہم، یہ عام طور پر ایک لنک فنکشن کا استعمال کرتے ہوئے تبدیل ہوتا ہے۔

:لنک فنکشن .2

لنک فنکشن ایک غیر لکیری فنکشن ہے جو لکیری پیشن گو کو جوابی متغیر کی متوقع قدر سے جوڑتا ہے۔ یہ اس بات کو - یقینی بناتا ہے کہ پیش گوئی کی گئی قدریں جوابی متغیر کی تقسیم کے لیے مناسب حد کے اندر ہوں۔ عام لنک فنکشنز میں لاجسٹک ریگریشن کے لیے لاگٹ فنکشن، پوسن ریگریشن کے لیے لاگ فنکشن، اور لکیری ریگریشن کے لیے شناختی فنکشن شامل ہیں۔

:امکانی تقسیم .3

آپ کو جوابی متغیر کی امکانی تقسیم کی وضاحت کرنے کی اجازت دیتے ہیں۔ آپ کے ڈیٹا کی نوعیت پر منحصر GLMs - وغیرہ۔ ،Gaussian، Binomial، Poisson، ہے، آپ تقسیم کا انتخاب کر سکتے ہیں جیسے کہ

:نقصان کا فنکشن .4

میں نقصان کا فنکشن پیش گوئی شدہ اقدار اور حقیقی مشاہدات کے درمیان فرق کو درست کرتا ہے۔ یہ منتخب کردہ GLM - امکانی تقسیم اور لنک فنکشن کے لیے مخصوص ہے۔

:کا نفاذ PyTorch

اپنی مرضی کے نقصان کے افعال، ماڈلز، اور اصلاحی الگورتھم کی وضاحت کرنے کی صلاحیت کی وجہ سے PyTorch :کو لاگو کرنے کا طریقہ یہاں ہے GLM میں PyTorch کو لاگو کرنے کے لیے ایک ورسٹائل فریم ورک فراہم کرتا ہے۔ GLMs

:ڈیٹا کی تیاری .1

اپنا ڈیٹا تیار کریں، بشمول آزاد متغیر (پیش گوئی کرنے والے) اور منحصر متغیر (جواب)۔ -

:ماڈل کی تعریف .2

ماڈل کی وضاحت کریں۔ اس میں عام طور پر ایک حسب ضرورت ماڈل کلاس کی وضاحت شامل ہوتی ہے جو GLM اپنے - GLM سے وراثت میں ملتی ہے۔ اس کلاس میں، آپ لکیری پیشن گوئی، لنک فنکشن، اور آپ کے مخصوص `nn.Module` کے لیے درکار کوئی بھی اضافی اجزاء بتاتے ہیں۔

3. نقصان کا فنکشن:

- GLM کے لیے مناسب نقصان کی نمائندگی کرے۔ نقصان کے فنکشن بنائیں جو آپ کے فنکشن کو ماڈل کی پیشین گوئیوں اور حقیقی قدروں کو بطور ان پٹ لینا چاہیے اور منتخب کردہ امکانی تقسیم اور لنک فنکشن کی بنیاد پر نقصان کا حساب لگانا چاہیے۔

4. اصلاح:

- ایک اصلاحی الگورتھم کا انتخاب کریں (مثال کے طور پر، اسٹاکسٹک گریڈینٹ ڈیسنٹ، ایڈم) اور اپنی مرضی کے نقصان کے کو تربیت دیں۔ اس عمل میں نقصان کو کم کرنے کے لیے ماڈل کے پیرامیٹرز کو بار بار GLM فنکشن کا استعمال کرتے ہوئے اپ ڈیٹ کرنا شامل ہے۔

استعمال کی مثالیں:

- PyTorch میں GLMs کے استعمال کے کچھ عام معاملات یہ ہیں:

1. لکیری رجعت: آپ Gaussian نافذ کو رجعت لکیری ڈسٹری بیوشن اور شناختی لنک فنکشن کا استعمال کر کے سادہ کر سکتے ہیں۔

2. لاجسٹک ریگریشن: بائنری درجہ بندی کے لیے، آپ لاجسٹک (لاجٹ) لنک فنکشن اور بائنومیل ڈسٹری بیوشن استعمال کر سکتے ہیں۔

3. پوائسن ریگریشن: کاؤنٹ ڈیٹا کے ساتھ کام کرتے وقت، پوسن ریگریشن لاگ لنک فنکشن کے ساتھ استعمال کیا جاتا ہے۔

4. آرڈینل ریگریشن: آرڈینل ریگریشن ماڈلز، جیسے متناسب مشکلات کے ماڈل، کو لنک فنکشن اور نقصان کے فنکشن کو اپنی مرضی کے مطابق بنا کر لاگو کیا جا سکتا ہے۔

نتیجہ:

PyTorch میں GLMs حسب ضرورت نقصان کے افعال اور ماڈلز کی وضاحت کرنے کی صلاحیت کے ساتھ ،ضرورت فریم ورک فراہم کرتے ہیں۔ حسب ریگریشن اور درجہ بندی کے ماڈلز کی ایک وسیع رینج کو نافذ کرنے کے لیے ایک لچکدار اور غیر گاوسی ڈیٹا اور خصوصی ماڈلنگ کے منظرناموں کو سنبھالنے کے لیے موزوں ہے جہاں روایتی لکیری رجعت PyTorch کافی نہیں ہو سکتی۔

Pyro

Pyro is a probabilistic programming library for Python that combines the best of deep learning frameworks, such as PyTorch, with the capabilities of probabilistic programming. It is developed by Uber AI Labs and provides a powerful framework for Bayesian modeling, probabilistic machine learning, and inference. Here's a detailed explanation of Pyro:

Probabilistic Programming:

- Probabilistic programming is a paradigm that allows you to specify and manipulate probabilistic models using code. In this approach, you can define random variables, probabilistic dependencies, and infer quantities of interest by leveraging probabilistic programming languages like Pyro.

Key Features:

1. **Deep Integration with PyTorch**:

- Pyro is built on top of PyTorch, one of the leading deep learning libraries. This integration allows you to leverage PyTorch's powerful features, including automatic differentiation, GPU support, and a wide range of neural network capabilities.

2. **Flexibility and Expressiveness**:

- Pyro provides a high level of expressiveness, allowing you to build complex probabilistic models with ease. You can define random variables, condition on observations, and specify intricate dependencies using Pyro's concise syntax.

3. **Stochastic Functions**:

- Pyro treats probabilistic programs as stochastic functions. These stochastic functions combine deterministic and stochastic operations, enabling you to model complex real-world phenomena.

4. **Automatic Differentiation**:

- Pyro performs automatic differentiation to calculate gradients efficiently. This is crucial for performing probabilistic inference, including methods like variational inference and Markov Chain Monte Carlo (MCMC).

5. **Inference Algorithms**:

- Pyro supports various probabilistic inference algorithms, including variational inference, Hamiltonian Monte Carlo (HMC), and No-U-Turn Sampler (NUTS). These methods help you estimate posterior distributions and make probabilistic predictions.

6. **Scalability**:

- Pyro is designed to scale from small-scale problems to large, complex models. It is well-suited for both research and production use cases.

7. **Bayesian Modeling**:

- You can use Pyro to create Bayesian models, which allow you to reason about uncertainty in your data, make predictions, and perform Bayesian parameter estimation.

8. **Model Validation**:

- Pyro provides tools for model validation and model comparison. You can assess the quality of your models and perform hypothesis testing.

9. **Probabilistic Deep Learning**:

- Pyro is particularly well-suited for probabilistic deep learning. It allows you to build deep generative models and perform uncertainty quantification in deep neural networks.

10. **Community and Ecosystem**:

- Pyro has an active community and a growing ecosystem of libraries and resources. You can find tutorials, examples, and extensions to enhance your Pyro workflow.

Usage:

Using Pyro typically involves the following steps:

1. **Define a Probabilistic Model**:

- Start by defining a probabilistic model in Pyro. You can specify random variables, probability distributions, and relationships between variables. Pyro uses PyTorch's tensor operations to model these dependencies.

2. **Inference**:

- Perform inference on your probabilistic model to estimate the posterior distribution over latent variables. Pyro provides various inference algorithms for this purpose.

3. **Predictions and Uncertainty**:

- Once you have an estimated posterior, you can make predictions and quantify uncertainty in your model. This is particularly useful for applications where uncertainty is critical, such as Bayesian decision-making and uncertainty-aware machine learning.

Applications:

Pyro is applicable in a wide range of fields and applications, including:

- Bayesian modeling and probabilistic graphical models.

- Probabilistic deep learning for uncertainty estimation in neural networks.

- Natural language processing, speech recognition, and computer vision.

- Reinforcement learning and robotics.

- Causal inference and counterfactual reasoning.

In summary, Pyro is a powerful and flexible probabilistic programming library for Python that seamlessly integrates with PyTorch. It allows you to build complex probabilistic models, perform Bayesian inference, and make predictions while taking into account the uncertainty inherent in real-world data.

پائرو

کے لیے ایک امکانی پروگرامنگ لائبریری ہے جو ممکنہ پروگرامنگ کی صلاحیتوں کے ساتھ گہرے سیکھنے Pyro Python ماڈلنگ، Bayesian نے تیار کیا ہے اور Uber AI Labs کو یکجا کرتی ہے۔ اسے PyTorch کے بہترین فریم ورک، جیسے :امکانی مشین لرننگ، اور اندازہ کے لیے ایک طاقتور فریم ورک فراہم کرتا ہے۔ یہاں پائرو کی تفصیلی وضاحت ہے

:امکانی پروگرامنگ

امکانی پروگرامنگ ایک ایسا نمونہ ہے جو آپ کو کوڈ کا استعمال کرتے ہوئے امکانی ماڈلز کی وضاحت اور ہیرا پھیری کرنے - جیسی امکانی پروگرامنگ زبانوں کا فائدہ اٹھا کر بے ترتیب متغیرات، Pyro کی اجازت دیتا ہے۔ اس نقطہ نظر میں، آپ امکانی انحصار، اور دلچسپی کی مقدار کا اندازہ لگا سکتے ہیں۔

:اہم خصوصیات

:کے ساتھ گہرا انضمام PyTorch 1.

کے اوپر بنایا گیا ہے، جو کہ معروف گہری سیکھنے والی لائبریریوں میں سے ایک ہے۔ یہ انضمام آپ PyTorch کو Pyro - سپورٹ، اور نیورل GPU ،کی طاقتور خصوصیات سے فائدہ اٹھانے کی اجازت دیتا ہے، بشمول خودکار تفریق PyTorch کو نیٹ ورک کی صلاحیتوں کی ایک وسیع رینج۔

:لچک اور اظہار خیال .2

پائرو ایک اعلی سطحی اظہار فراہم کرتا ہے، جس سے آپ آسانی کے ساتھ پیچیدہ امکانی ماڈلز بنا سکتے ہیں۔ آپ بے - کے جامع نحو کا استعمال کرتے ہوئے پیچیدہ Pyro ترتیب متغیرات کی وضاحت کر سکتے ہیں، مشاہدات کی حالت، اور انحصار کی وضاحت کر سکتے ہیں۔

:اسٹاکسٹک افعال .3

پائرو احتمالی پروگراموں کو اسٹاکسٹک افعال کے طور پر مانتا ہے۔ یہ سٹاکسٹک فنکشن ڈیٹرمنسٹک اور سٹاکسٹک آپریشنز - کو یکجا کرتے ہیں، جو آپ کو پیچیدہ حقیقی دنیا کے مظاہر کو ماڈل بنانے کے قابل بناتے ہیں۔

:خودکار تفریق .4

میلان کو مؤثر طریقے سے شمار کرنے کے لیے خودکار تفریق کرتا ہے۔ یہ امکانی تخمینہ لگانے کے لیے بہت اہم ہے، Pyro - ۔(MCMC) بشمول تغیراتی تخمینہ اور مارکوف چین مونٹی کارلو

:انفرنس الگورتھم .5

اور ،(HMC) پائرو مختلف امکانی تخمینہ الگورتھم کی حمایت کرتا ہے، بشمول تغیراتی تخمینہ، ہیملٹونین مونٹی کارلو - ۔ یہ طریقے آپ کو بعد کی تقسیم کا اندازہ لگانے اور امکانی پیشین گوئیاں کرنے میں مدد(No-U-Turn Sampler (NUTS کرتے ہیں۔

:توسیع پذیری .6

پائرو کو چھوٹے پیمانے کے مسائل سے بڑے، پیچیدہ ماڈلز تک پیمانہ کرنے کے لیے ڈیزائن کیا گیا ہے۔ یہ تحقیق اور پیداوار - کے استعمال کے معاملات دونوں کے لیے موزوں ہے۔

:ماڈلنگ Bayesian .7

کا استعمال کر سکتے ہیں، جو آپ کو اپنے ڈیٹا میں غیر یقینی صورتحال کے Pyro ماڈلز بنانے کے لیے Bayesian آپ - پیرامیٹر کا تخمینہ لگانے کی اجازت دیتا ہے۔ Bayesian بارے میں استدلال کرنے، پیشین گوئیاں کرنے، اور

8. ماڈل کی توثیق:

- پائرو ماڈل کی توثیق اور ماڈل کے موازنہ کے لیے ٹولز فراہم کرتا ہے۔ آپ اپنے ماڈلز کے معیار کا اندازہ لگا سکتے ہیں اور مفروضے کی جانچ کر سکتے ہیں۔

9. امکانی گہری تعلیم:

- خاص طور پر ممکنہ گہری سیکھنے کے لیے موزوں ہے۔ یہ آپ کو گہرے جنریٹو ماڈلز بنانے اور گہرے نیورل نیٹ Pyro ورکس میں غیر یقینی کی مقدار درست کرنے کی اجازت دیتا ہے۔

10. کمیونٹی اور ماحولیاتی نظام:

- ورک فلو کو Pyro پائرو کی ایک فعال کمیونٹی ہے اور لائبریریوں اور وسائل کا بڑھتا ہوا ماحولیاتی نظام ہے۔ آپ اپنے بڑھانے کے لیے سبق، مثالیں اور ایکسٹینشنز تلاش کر سکتے ہیں۔

استعمال:

Pyro کے استعمال میں عام طور پر درج ذیل اقدامات شامل ہیں:

1. ایک امکانی ماڈل کی وضاحت کریں:

- پائرو میں امکانی ماڈل کی وضاحت کرکے شروع کریں۔ آپ بے ترتیب متغیرات، امکانی تقسیم، اور متغیر کے درمیان تعلقات کے ٹینسر آپریشنز کا استعمال کرتا ہے۔ PyTorch کی وضاحت کر سکتے ہیں۔ پائرو ان انحصاروں کو ماڈل کرنے کے لیے

2. اندازہ:

- پوشیدہ متغیرات پر بعد کی تقسیم کا اندازہ لگانے کے لیے اپنے امکانی ماڈل پر تخمینہ لگائیں۔ پائرو اس مقصد کے لیے مختلف انفرنس الگورتھم فراہم کرتا ہے۔

3. پیشین گوئیاں اور غیر یقینی صورتحال:

- ایک بار جب آپ کے پاس تخمینہ لگنے کے بعد، آپ پیشین گوئیاں کر سکتے ہیں اور اپنے ماڈل میں غیر یقینی صورتحال کا اندازہ لگا سکتے ہیں۔ یہ خاص طور پر ان ایپلی کیشنز کے لیے مفید ہے جہاں غیر یقینی صورتحال بہت اہم ہے، جیسے کہ فیصلہ سازی اور غیر یقینی صورتحال سے آگاہ مشین لرننگ۔ Bayesian

درخواستیں:

Pyro فیلڈز اور ایپلی کیشنز کی ایک وسیع رینج میں لاگو ہوتا ہے، بشمول:

- Bayesian ماڈلنگ اور امکانی گرافیکل ماڈل۔

- عصبی نیٹ ورکس میں غیر یقینی صورتحال کے تخمینے کے لیے ممکنہ گہرائی سے سیکھنا۔

- قدرتی زبان کی پروسیسنگ، تقریر کی شناخت، اور کمپیوٹر ویژن۔

- کمک سیکھنے اور روبوٹکس۔

- وجہ کا اندازہ اور جوابی استدلال۔

خلاصہ یہ کہ Pyro Python کے لیے ایک طاقتور اور لچکدار امکانی پروگرامنگ لائبریری ہے جو بغیر کسی رکاوٹ کے PyTorch کے ساتھ مربوط ہوتی ہے۔ یہ آپ کو حقیقی دنیا کے اعداد و شمار میں موجود غیر یقینی صورتحال کو مدنظر رکھتے ہوئے پیچیدہ امکانی ماڈلز بنانے، بایسیئن اندازہ لگانے اور پیشین گوئیاں کرنے کی اجازت دیتا ہے۔

NeRF

NeRF (Neural Radiance Fields) is a novel approach in computer graphics and computer vision for synthesizing novel views of complex 3D scenes by modeling the volumetric scene as a continuous 3D function. NeRF has gained popularity for its ability to create highly detailed and photorealistic 3D reconstructions of scenes and objects from 2D images. While NeRF is traditionally implemented in deep learning frameworks like PyTorch and TensorFlow, it is important to note that the core concept is based on the original research paper, “NeRF: Representing Scenes as Neural Radiance Fields,” by Ben Mildenhall, Pratul P. Srinivasan, et al.

Key Concepts:

1. **Scene Representation**:

- NeRF represents a 3D scene as a continuous function that maps 3D coordinates to radiance values. This function is commonly referred to as the “NeRF model.” It takes a set of 3D coordinates as input and predicts the color (radiance) and opacity (visibility) at those locations. This continuous scene representation enables the synthesis of novel views and can be used for 3D reconstruction.

2. **View Synthesis**:

- NeRF excels at view synthesis, where it can generate novel images of a scene from any viewpoint by querying the NeRF model. This process involves rendering a 2D image from the 3D scene representation, considering factors like camera poses, view directions, and lighting conditions. The rendered image is composited to create a novel view of the scene.

3. **Training Data**:

- NeRF requires a set of input images taken from different viewpoints and corresponding camera parameters, including camera poses and intrinsics. These images are used for training the NeRF model. Depth information or 3D point clouds can be used to help establish the 3D scene structure.

4. **Loss Functions**:

- NeRF is trained using loss functions that encourage the model to predict radiance values that match the observed images and to generate consistent 3D scene representations. Common loss functions include silhouette consistency, rendering loss, and positional loss.

Implementation in Python:

The implementation of NeRF in Python using deep learning frameworks like PyTorch or TensorFlow typically involves the following steps:

1. **Data Collection**:

- Collect a set of images and corresponding camera parameters that capture different viewpoints of the scene.

2. **Data Preprocessing**:

- Extract camera poses and intrinsics from the images and preprocess the data, such as normalizing the images and converting pixel coordinates to ray directions.

3. **Model Architecture**:

- Define the NeRF model, which typically consists of a neural network. This network takes 3D coordinates as input and predicts the radiance and opacity for each location.

4. **Training**:

- Train the NeRF model using the collected data and appropriate loss functions. Optimization methods like stochastic gradient descent (SGD) or Adam are commonly used.

5. **View Synthesis**:

- Once the model is trained, you can use it to synthesize novel views of the 3D scene by rendering images from different viewpoints.

6. **Evaluation**:

- Evaluate the quality of the synthesized views and the fidelity of the 3D scene reconstruction. This can involve quantitative metrics and visual inspection.

Applications:

NeRF has a wide range of applications, including:

- 3D scene reconstruction from 2D images.

- Creating photorealistic 3D models and environments for computer graphics and virtual reality.

- Novel view synthesis for augmented reality and virtual reality applications.

- Photogrammetry and 3D mapping.

NeRF and its variants have pushed the boundaries of 3D scene representation and view synthesis, enabling the creation of highly realistic 3D environments and models from 2D images. Its combination of neural networks and continuous scene representations has paved the way for exciting advancements in computer graphics and computer vision.

این آر ایف

کمپیوٹر گرافکس اور کمپیوٹر وژن میں ایک نیا نقطہ نظر ہے جو کہ ایک مسلسل (Neural Radiance Fields) NeRF نے NeRF مناظر کے نئے خیالات کی ترکیب کرتا ہے۔ Dفنکشن کے طور پر والیومیٹرک سین کو ماڈل کر کے پیچیدہ 3D 3 تعمیر نو کی صلاحیت کے لیے مقبولیت حاصل Dامیجز سے مناظر اور اشیاء کی انتہائی تفصیلی اور فوٹو ریئلسٹک 3 2D روایتی طور پر پائی ٹارچ اور ٹینسر فلو جیسے گہرے سیکھنے کے فریم ورک میں لاگو کیا جاتا ہے، NeRF کی ہے۔ اگرچہ کے NeRF: Neural Radiance Fields" ،لیکن یہ نوٹ کرنا ضروری ہے کہ بنیادی تصور اصل تحقیقی مقالے پر مبنی ہے رحمہ اللہ تعالی ET ، طور پر مناظر کی نمائندگی کرنا،" بین ملڈن ہال، پراتول پی سری نواسن۔

:بنیادی خیال

:منظر کی نمائندگی .1

کوآرڈینیٹس کو تابناک اقدار کے نقشے Dمنظر کو ایک مسلسل فنکشن کے طور پر پیش کرتا ہے جو 3 Dایک 3 NeRF - کوآرڈینیٹس کا ایک سیٹ لیتا ہے Dماڈل" کہا جاتا ہے۔ یہ ان پٹ کے طور پر 3 NeRF" بناتا ہے۔ اس فنکشن کو عام طور پر اور ان مقامات پر رنگ (چمک) اور دھندلاپن (مرئیت) کی پیش گوئی کرتا ہے۔ یہ مسلسل منظر کی نمائندگی ناول کے خیالات تعمیر نو کے لیے استعمال کیا جا سکتا ہے۔ Dکی ترکیب کو قابل بناتی ہے اور اسے 3

:ترکیب دیکھیں .2

ماڈل سے استفسار کر کے کسی بھی نقطہ نظر سے NeRF منظر کی ترکیب میں سبقت لے جاتا ہے، جہاں یہ NeRF - تصویر پیش کرنا، کیمرہ پوز، دیکھنے کی Dمنظر کی نمائندگی سے 2 Dمنظر کی نئی تصاویر بنا سکتا ہے۔ اس عمل میں 3 سمت اور روشنی کے حالات جیسے عوامل پر غور کرنا شامل ہے۔ پیش کردہ تصویر کو منظر کا ایک نیا منظر بنانے کے لیے مرکب کیا گیا ہے۔

:تربیتی ڈیٹا .3

کو مختلف نقطہ نظر اور متعلقہ کیمرے کے پیرامیٹرز سے لی گئی ان پٹ امیجز کا ایک سیٹ درکار ہوتا ہے، بشمول NeRF - منظر کے ڈھانچے کو قائم Dماڈل کی تربیت کے لیے استعمال ہوتی ہیں۔ 3 NRF کیمرہ پوز اور اندرونی چیزیں۔ یہ تصاویر پوائنٹ کلاؤڈز کا استعمال کیا جا سکتا ہے۔ Dکرنے میں مدد کے لیے گہرائی کی معلومات یا 3

:نقصان کے افعال .4

کو نقصان کے فنکشنز کا استعمال کرتے ہوئے تربیت دی جاتی ہے جو ماڈل کو تابناک اقدار کی پیش گوئی کرنے کی NeRF - منظر کی مستقل نمائندگی پیدا کرتی ہیں۔ عام نقصان کے Dترغیب دیتی ہے جو مشاہدہ شدہ تصاویر سے ملتی ہیں اور 3 افعال میں سلائیٹ کی مستقل مزاجی، رینڈرنگ نقصان، اور پوزیشنی نقصان شامل ہیں۔

:ازگر میں نفاذ

کے نفاذ میں عام NeRF میں Python جیسے ڈیپ لرننگ فریم ورک کا استعمال کرتے ہوئے TensorFlow یا PyTorch :طور پر درج ذیل اقدامات شامل ہوتے ہیں

:ڈیٹا اکٹھا کرنا .1

تصاویر اور متعلقہ کیمرہ پیرامیٹرز کا ایک سیٹ جمع کریں جو منظر کے مختلف نقطہ نظر کو گرفت میں لیتے ہیں۔ -

:ڈیٹا پری پروسیسنگ .2

تصاویر سے کیمرہ پوز اور اندرونی چیزیں نکالیں اور ڈیٹا کو پہلے سے پروسیس کریں، جیسے امیجز کو نارمل کرنا اور پکسل - کوآرڈینیٹ کو شعاعوں کی سمتوں میں تبدیل کرنا۔

3. ماڈل آرکیٹیکچر:

- NeRF 3 ماڈل کی وضاحت کریں، جو عام طور پر نیورل نیٹ ورک پر مشتمل ہوتا ہے۔ یہ نیٹ ورک ان پٹ کے طور پر 3D کوآرڈینیٹ لیتا ہے اور ہر مقام کے لیے چمک اور دھندلاپن کی پیش گوئی کرتا ہے۔

4. تربیت:

- NeRF ماڈل کو تربیت دیں۔ سٹاکاسٹک گریڈینٹ ڈیسنٹ (SGD) یا ایڈم جیسے اصلاح کے طریقے عام طور پر استعمال ہوتے ہیں۔ جمع کردہ ڈیٹا اور مناسب نقصان کے افعال کا استعمال کرتے ہوئے

5. ترکیب دیکھیں:

- ایک بار ماڈل کو تربیت دینے کے بعد، آپ اسے مختلف نقطہ نظر سے تصاویر پیش کرکے 3D منظر کے نئے خیالات کی ترکیب کے لیے استعمال کر سکتے ہیں۔

6. تشخیص:

- 3D منظر کی تعمیر نو کی مخلصی کا اندازہ کریں۔ اس میں مقداری میٹرکس اور بصری ترکیب شدہ نظاروں کے معیار اور معائنہ شامل ہو سکتا ہے۔

درخواستیں:

NeRF کے پاس ایپلی کیشنز کی ایک وسیع رینج ہے، بشمول:

- 2D امیجز سے 3D منظر کی تعمیر نو۔

- کمپیوٹر گرافکس اور ورچوئل رئیلٹی کے لیے فوٹو ریئلسٹک 3D ماڈلز اور ماحول بنانا۔

- بڑھا ہوا حقیقت اور ورچوئل رئیلٹی ایپلی کیشنز کے لیے ناول ویو کی ترکیب۔

- 3D فوٹوگرامیٹری اور میپنگ۔

NeRF 3D منظر کی نمائندگی اور نظارے کی ترکیب کی حدود کو آگے بڑھایا ہے، جس سے 2D امیجز سے انتہائی حقیقت پسندانہ 3D ماحول اور ماڈلز کی تخلیق ممکن ہے۔ اس کے عصبی نیٹ ورکس اور مسلسل نمائندگی کے امتزاج نے کمپیوٹر گرافکس اور کمپیوٹر ویژن میں دلچسپ ترقی کی راہ ہموار کی ہے۔

StyleGAN

StyleGAN (Style Generative Adversarial Network) is a powerful deep learning model used for generating high-quality and highly customizable images, particularly in the domain of generative adversarial networks (GANs). It was developed by NVIDIA as an extension of the original GAN framework to create images with remarkable detail, control over content and style, and the ability to generate entirely new and realistic images.

Key Features:

1. **High-Quality Image Generation**:

- StyleGAN is known for its ability to generate high-resolution images with impressive levels of detail and realism. This makes it suitable for various applications, including art, image manipulation, and more.

2. **Style Control**:

- StyleGAN allows for fine-grained control over the generated images. Users can manipulate the “style” of the generated images, which includes factors like color, texture, and structure. This level of control is achieved through the disentanglement of the latent space.

3. **Progressive Growing**:

- StyleGAN employs a progressive growing approach during training. It starts with low-resolution images and gradually increases the resolution, ensuring stable training and improved quality.

4. **Noise Injection**:

- The model uses noise injection to introduce stochasticity into the generation process. This adds variety to the generated images, making them more realistic.

5. **Latent Space Manipulation**:

- Users can manipulate the latent space to achieve various visual effects, such as morphing between images, changing facial expressions, or generating entirely new images based on the characteristics of existing images.

6. **Transfer Learning**:

- StyleGAN models can be fine-tuned on specific datasets or used for transfer learning. This means you can adapt pretrained models for specific image generation tasks.

7. **Community and Ecosystem**:

- StyleGAN has a vibrant community and ecosystem, with many resources, pretrained models, and tools available to users. It is supported by popular deep learning frameworks like TensorFlow and PyTorch.

Usage:

Using StyleGAN involves several steps:

1. **Data Collection**:

- Collect a dataset of images that you want your StyleGAN model to learn from. The dataset should be representative of the type of images you want to generate.

2. **Model Training**:

- Train a StyleGAN model on your dataset. Training a StyleGAN model is computationally intensive and typically requires access to powerful GPUs. You can start with a pretrained model and fine-tune it on your dataset or train a model from scratch.

3. **Latent Space Exploration**:

- Explore the latent space by manipulating the model's latent vectors. This allows you to control various aspects of the generated images, such as style, pose, and content.

4. **Image Generation**:

- Use the trained StyleGAN model to generate new images based on the manipulated latent vectors. You can generate individual images, morph between images, or create entire galleries of diverse images.

5. **Model Evaluation**:

- Evaluate the quality and diversity of the generated images using visual inspection and metrics like Inception Score or Fréchet Inception Distance (FID).

Applications:

StyleGAN has found applications in various domains, including:

- Art and creative projects, generating novel and artistic images.

- Image-to-image translation and style transfer.

- Deepfakes and face manipulation.

- Data augmentation in computer vision and machine learning.

- Creating high-quality synthetic data for training machine learning models.

StyleGAN and its variants have had a profound impact on the field of generative models, enabling the generation of realistic images with an unprecedented level of control and quality. However, it's important to use such technology responsibly and ethically, as it also has the potential to be used for malicious purposes, like deepfake generation.

GAN اسٹائل

ایک طاقتور ڈیپ لرننگ ماڈل ہے جو اعلیٰ معیار اور StyleGAN (Style Generative Adversarial Network) کے (GANs) انتہائی حسب ضرورت تصاویر بنانے کے لیے استعمال کیا جاتا ہے، خاص طور پر جنریٹو ایڈورسریل نیٹ ورکس فریم ورک کی توسیع کے طور پر تیار کیا ہے تاکہ قابل ذکر تفصیل، مواد اور انداز GAN نے اصل NVIDIA ڈومین میں۔ اسے پر کنٹرول، اور بالکل نئی اور حقیقت پسندانہ تصاویر بنانے کی صلاحیت کے ساتھ تصاویر بنائیں۔

:اہم خصوصیات

:اعلیٰ معیار کی تصویر کی تخلیق .1

- StyleGAN تفصیل اور حقیقت پسندی کی متاثر کن سطحوں کے ساتھ اعلی ریزولیوشن امیجز بنانے کی صلاحیت کے لیے جانا جاتا ہے۔ یہ اسے مختلف ایپلی کیشنز کے لیے موزوں بناتا ہے، بشمول آرٹ، تصویری ہیرا پھیری اور بہت کچھ۔

:انداز کنٹرول .2

- StyleGAN تیار کردہ امیجز پر عمدہ کنٹرول کی اجازت دیتا ہے۔ صارف تیار کردہ تصاویر کے "اسٹائل" کو جوڑ سکتے ہیں، جس میں رنگ، ساخت اور ساخت جیسے عوامل شامل ہیں۔ کنٹرول کی یہ سطح اویکت جگہ کو منقطع کرنے کے ذریعے حاصل کی جاتی ہے۔

:ترقی پسند نشوونما .3

- اسٹائل گین تربیت کے دوران ایک ترقی پسند بڑھتے ہوئے نقطہ نظر کو استعمال کرتا ہے۔ یہ کم ریزولیوشن امیجز سے شروع ہوتا ہے اور آہستہ آہستہ ریزولوشن کو بڑھاتا ہے، مستحکم تربیت اور بہتر معیار کو یقینی بناتا ہے۔

:شور انجکشن .4

- ماڈل جنریشن کے عمل میں سٹاکسٹیٹی کو متعارف کرانے کے لیے شور انجیکشن کا استعمال کرتا ہے۔ یہ تخلیق کردہ تصاویر میں مختلف قسم کا اضافہ کرتا ہے، انہیں مزید حقیقت پسندانہ بناتا ہے۔

:اویکت خلائی ہیرا پھیری .5

- صارفین مختلف بصری اثرات حاصل کرنے کے لیے اویکت جگہ کو جوڑ سکتے ہیں، جیسے کہ تصویروں کے درمیان مورف کرنا، چہرے کے تاثرات کو تبدیل کرنا، یا موجودہ تصاویر کی خصوصیات کی بنیاد پر بالکل نئی تصاویر بنانا۔

:ٹرانسفر لرننگ .6

- StyleGAN ماڈلز کو مخصوص ڈیٹاسیٹس پر ٹھیک بنایا جا سکتا ہے یا ٹرانسفر لرننگ کے لیے استعمال کیا جا سکتا ہے۔ اس کا مطلب ہے کہ آپ مخصوص امیج جنریشن کے کاموں کے لیے پہلے سے تربیت یافتہ ماڈلز کو اپنا سکتے ہیں۔

:کمیونٹی اور ماحولیاتی نظام .7

- StyleGAN کے پاس ایک متحرک کمیونٹی اور ماحولیاتی نظام ہے، جس میں بہت سے وسائل، پہلے سے تربیت یافتہ ماڈلز، اور ٹولز صارفین کے لیے دستیاب ہیں۔ اس کی حمایت مقبول ڈیپ لرننگ فریم ورک جیسے TensorFlow اور PyTorch سے ہوتی ہے۔

:استعمال

:StyleGAN استعمال کرنے میں کئی مراحل شامل ہیں

:ڈیٹا اکٹھا کرنا .1

ماڈل کو سیکھنا چاہتے ہیں۔ ڈیٹاسیٹ اس قسم کی تصاویر StyleGAN تصاویر کا ڈیٹاسیٹ جمع کریں جن سے آپ اپنے -
کا نمائندہ ہونا چاہیے جو آپ بنانا چاہتے ہیں۔

2. ماڈل ٹریننگ:

ماڈل کی تربیت کمپیوٹیشنل طور پر بہت StyleGAN ماڈل کو تربیت دیں۔ ایک StyleGAN اپنے ڈیٹاسیٹ پر ایک -
تک رسائی کی ضرورت ہوتی ہے۔ آپ پہلے سے تربیت یافتہ ماڈل کے ساتھ GPUs زیادہ ہوتی ہے اور عام طور پر طاقتور
شروع کر سکتے ہیں اور اسے اپنے ڈیٹاسیٹ پر ٹھیک کر سکتے ہیں یا کسی ماڈل کو شروع سے تربیت دے سکتے ہیں۔

3. اویکت خلائی ریسرچ:

ماڈل کے اویکت ویکٹرز کو جوڑ کر پوشیدہ جگہ کو دریافت کریں۔ یہ آپ کو تیار کردہ تصاویر کے مختلف پہلوؤں جیسے کہ -
انداز، پوز اور مواد کو کنٹرول کرنے کی اجازت دیتا ہے۔

4. تصویر کی تخلیق:

ماڈل کا استعمال کریں۔ آپ StyleGAN ہیرا پھیری والے اویکت ویکٹر کی بنیاد پر نئی تصاویر بنانے کے لیے تربیت یافتہ -
انفرادی تصاویر بنا سکتے ہیں، تصاویر کے درمیان شکل بنا سکتے ہیں، یا متنوع تصاویر کی پوری گیلریاں بنا سکتے ہیں۔

5. ماڈل کی تشخیص:

کا استعمال کرتے (FID) Fréchet Inception Distance یا Inception Score بصری معائنہ اور میٹرکس جیسے -
ہوئے تیار کردہ تصاویر کے معیار اور تنوع کا اندازہ کریں۔

درخواستیں:

StyleGAN کو مختلف ڈومینز میں ایپلی کیشنز ملی ہیں، بشمول:

- آرٹ اور تخلیقی منصوبے، ناول اور فنکارانہ امیجز بنانا۔

- تصویر سے تصویر کا ترجمہ اور انداز کی منتقلی۔

- ڈیپ فیکس اور چہرے کی ہیرا پھیری۔

- کمپیوٹر ویژن اور مشین لرننگ میں ڈیٹا میں اضافہ۔

- مشین لرننگ ماڈلز کی تربیت کے لیے اعلیٰ معیار کا مصنوعی ڈیٹا بنانا۔

اور اس کی مختلف حالتوں نے جنریٹو ماڈلز کے میدان پر گہرا اثر ڈالا ہے، جس سے کنٹرول اور معیار کی بے StyleGAN
مثال سطح کے ساتھ حقیقت پسندانہ تصاویر کی تخلیق کو قابل بنایا گیا ہے۔ تاہم، اس طرح کی ٹیکنالوجی کو ذمہ داری اور
اخلاقی طور پر استعمال کرنا ضروری ہے، کیونکہ اس میں ڈیپ فیک جنریشن جیسے بدنیتی پر مبنی مقاصد کے لیے استعمال
کیے جانے کی صلاحیت بھی ہے۔

Overview of SQLAlchemy

SQLAlchemy is a popular Python SQL toolkit and Object-Relational Mapping (ORM) library, providing a powerful and flexible way to work with databases in Python applications. It allows developers to interact with databases using Python objects, making database operations more intuitive and Pythonic. Here's an overview of SQLAlchemy:

Key Features:

1. **ORM (Object-Relational Mapping)**: SQLAlchemy provides a high-level ORM that allows developers to map database tables to Python classes and manipulate data using object-oriented programming techniques.
2. **Database Abstraction**: SQLAlchemy supports multiple database engines (e.g., PostgreSQL, MySQL, SQLite) through its dialect system, allowing developers to write database-agnostic code.
3. **SQL Expression Language**: SQLAlchemy includes a SQL Expression Language that allows developers to construct SQL queries using Python expressions, providing a more Pythonic way to write database queries.
4. **Session Management**: SQLAlchemy's `Session` object provides a unit of work pattern for managing transactions and database interactions, ensuring data consistency and integrity.
5. **Query API**: SQLAlchemy provides a powerful query API for constructing database queries using Pythonic syntax, supporting filtering, ordering, grouping, and aggregation operations.
6. **Schema Definition**: SQLAlchemy allows developers to define database schemas using Python classes and declarative mapping, making it easy to define and maintain database structures.

Components:

1. **Core**: The SQLAlchemy Core provides the foundation for interacting with databases using SQL expressions and database-agnostic constructs. It includes features such as database reflection, schema definition, and SQL expression generation.
2. **ORM**: The SQLAlchemy ORM builds on top of the Core to provide an object-oriented interface for interacting with databases. It allows developers to map Python classes to database tables and perform CRUD operations using Python objects.
3. **Engine**: The Engine component provides a connection pool and a transactional system for executing SQL commands and managing database connections.

4. **Session**: The Session component provides a high-level interface for managing transactions and interactions with the database. It tracks changes to objects and applies them to the database using the unit of work pattern.

Example:

```
python

from sqlalchemy import create_engine, Column, Integer, Stringfrom
sqlalchemy.ext.declarative import declarative_basefrom sqlalchemy.orm import
sessionmaker # Define the database connectionengine =
create_engine('sqlite:///example.db') # Define the base class for declarative
mappingBase = declarative_base() # Define the model classclass User(Base):
__tablename__ = 'users'    id = Column(Integer, primary_key=True)    name =
Column(String)    age = Column(Integer) # Create the database
schemaBase.metadata.create_all(engine) # Create a sessionSession =
sessionmaker(bind=engine)session = Session() # Insert a new usernew_user =
User(name='Alice', age=30)session.add(new_user)session.commit() # Query usersusers
= session.query(User).all()for user in users:    print(user.name, user.age)
```

Conclusion:

SQLAlchemy is a powerful and flexible library for working with databases in Python applications. Whether you're building web applications, data analysis tools, or other types of software, SQLAlchemy provides a robust set of tools for interacting with databases and managing data effectively.

کا جائزہ SQLAlchemy

لائبریری ہے، جو Object-Relational Mapping (ORM) ٹول کٹ اور Python SQL ایک مشہور SQLAlchemy ایپلی کیشنز میں ڈیٹا بیس کے ساتھ کام کرنے کا ایک طاقتور اور لچکدار طریقہ فراہم کرتی ہے۔ یہ ڈیولپرز کو Python اشیاء کا استعمال کرتے ہوئے ڈیٹا بیس کے ساتھ بات چیت کرنے کی اجازت دیتا ہے، ڈیٹا بیس کے آپریشنز کو زیادہ Python :کا ایک جائزہ ہے SQLAlchemy بناتا ہے۔ یہاں Pythonic بدیہی اور

:اہم خصوصیات

1. ORM (Object-Relational Mapping): SQLAlchemy ایک اعلیٰ سطحی ORM فراہم کرتا ہے جو ڈویلپرز کلاسز کے لیے نقشہ بنانے اور آبجیکٹ اورینٹڈ پروگرامنگ تکنیک کا استعمال کرتے ہوئے ڈیٹا Python کو ڈیٹا بیس ٹیبلز کو میں ہیرا پھیری کرنے کی اجازت دیتا ہے۔

2. ڈیٹا بیس خلاصہ: SQLAlchemy اپنے بولی نظام کے ذریعے متعدد ڈیٹا بیس انجنوں (جیسے، PostgreSQL، MySQL، SQLite) کو سپورٹ کرتا ہے، جس سے ڈویلپرز کو ڈیٹا بیس-ایگنوسٹک کوڈ لکھنے کی اجازت ملتی ہے۔

3. ایس کیو ایل ایکسپریشن لینگویج: ایس کیو ایل کیمی میں ایک ایس کیو ایل ایکسپریشن لینگویج شامل ہے جو ڈیولپرز کو پائتھون ایکسپریشنز کا استعمال کرتے ہوئے ایس کیو ایل استفسارات بنانے کی اجازت دیتی ہے، جو ڈیٹا بیس کے سوالات کو لکھنے کے لیے مزید پائتھونک طریقہ فراہم کرتی ہے۔

4. سیشن مینجمنٹ: SQLAlchemy کا سیشن آبجیکٹ لین دین اور ڈیٹا بیس کے تعاملات کو منظم کرنے، ڈیٹا کی مستقل مزاجی اور سالمیت کو یقینی بنانے کے لیے ورک پیٹرن کی اکائی فراہم کرتا ہے۔

5. Query API: SQLAlchemy Pythonic syntax کا استعمال کرتے ہوئے ڈیٹا بیس کے استفسارات کی تعمیر فراہم کرتا ہے، فلٹرنگ، آرڈرنگ، گروپنگ، اور ایگریگیشن آپریشنز کو سپورٹ کرتا ہے۔ API کے لیے ایک طاقتور استفسار

6. اسکیما کی تعریف: SQLAlchemy کلاسز اور ڈیکلیرٹیو میپنگ کا استعمال کرتے ہوئے ڈیٹا Python ڈویلپرز کو بیس اسکیموں کی وضاحت کرنے کی اجازت دیتا ہے، جس سے ڈیٹا بیس کے ڈھانچے کی وضاحت اور اسے برقرار رکھنا آسان ہو جاتا ہے۔

:اجزاء

1. کور: SQLAlchemy Core SQL تعمیرات کا استعمال کرتے ہوئے ڈیٹا بیس کے agnostic-اظہار اور ڈیٹا بیس ساتھ بات چیت کے لیے بنیاد فراہم کرتا ہے۔ اس میں ڈیٹا بیس کی عکاسی، اسکیما کی تعریف، اور ایس کیو ایل ایکسپریشن جنریشن جیسی خصوصیات شامل ہیں۔

2. ORM: SQLAlchemy ORM ڈیٹا بیس کے ساتھ تعامل کے لیے ایک آبجیکٹ پر مبنی انٹرفیس فراہم کرنے کے اشیاء کا استعمال Python کلاسز کو ڈیٹا بیس ٹیبلز پر نقشہ بنانے اور Python لیے کور کے اوپر بناتا ہے۔ یہ ڈویلپرز کو آپریشنز کرنے کی اجازت دیتا ہے۔ CRUD کرتے ہوئے

3. انجن: انجن کا جزو ایک کنکشن پول اور ایس کیو ایل کمانڈز پر عمل درآمد کرنے اور ڈیٹا بیس کنکشن کے انتظام کے لیے ٹرانزیکشنل سسٹم فراہم کرتا ہے۔

4. سیشن: سیشن کا جزو ڈیٹا بیس کے ساتھ لین دین اور تعاملات کے انتظام کے لیے ایک اعلیٰ سطحی انٹرفیس فراہم کرتا ہے۔ یہ اشیاء میں ہونے والی تبدیلیوں کو ٹریک کرتا ہے اور ورک پیٹرن کی اکائی کا استعمال کرتے ہوئے انہیں ڈیٹا بیس پر لاگو کرتا ہے۔

:مثال

ازگر

```
sqlalchemy درآمد سے create_engine، عدد، کالم، Stringfrom sqlalchemy.ext.declarative import
declarative_basefrom sqlalchemy.orm درآمد سیشن میکر # ڈیٹا بیس کنکشن انجن کی وضاحت کریں =
create_engine('sqlite:///example.b' class=declaring کے لیے declarative_base() # ماڈل کلاس کلاس
__tablename__ = 'users' id = Column(Integer, prime_key=True) name :یوزر(بیس) کی وضاحت کریں
schemaBase.metadata.create_all( ڈیٹا بیس بنائیں # (Column(Integer) age = Column(String) =
ایک سیشن بنائیں سیشن = سیشن میکر(بائنڈ=انجین) سیشن = سیشن () # ایک نیا صارف نیا صارف داخل کریں # (انجن
صارف صارفین سے سوال کریں # ()session.commit()session.add(new_user)= صارف (نام = 'ایلس'، عمر = 30)
print(user.name, user.age) :صارفین کے لیے()session.query(User).all() =
```

نتیجہ:

ایپلی کیشنز میں ڈیٹا بیس کے ساتھ کام کرنے کے لیے ایک طاقتور اور لچکدار لائبریری ہے۔ SQLAlchemy Python ڈیٹا بیس کے SQLAlchemy ،چاہے آپ ویب ایپلیکیشنز، ڈیٹا اینالیسس ٹولز، یا سافٹ ویئر کی دیگر اقسام بنا رہے ہوں ساتھ بات چیت کرنے اور ڈیٹا کو مؤثر طریقے سے منظم کرنے کے لیے ٹولز کا ایک مضبوط سیٹ فراہم کرتا ہے۔

ORM concepts

Object-Relational Mapping (ORM) is a programming technique that allows developers to interact with relational databases using object-oriented programming (OOP) concepts. ORM frameworks, such as SQLAlchemy in Python, provide a way to map database tables to classes, and rows in those tables to objects in the programming language. Here are some key ORM concepts:

1. Entity:

An entity represents a real-world object or concept that can be stored in a database. In ORM, entities are typically mapped to database tables.

2. Class-Table Mapping:

ORM frameworks map each class in the application to a table in the database. Attributes of the class correspond to columns in the table, and instances of the class represent rows in the table.

3. Object-Relational Mapping:

ORM frameworks provide mechanisms for converting between objects in the programming language and rows in the database, allowing developers to interact with the database using objects and methods instead of raw SQL queries.

4. Relationships:

ORM frameworks support relationships between entities, allowing developers to model complex data structures and associations between objects. Common types of relationships include one-to-one, one-to-many, and many-to-many relationships.

5. Lazy Loading:

ORM frameworks often support lazy loading, a technique where related objects are loaded from the database only when they are accessed by the application, improving performance by minimizing unnecessary database queries.

6. Unit of Work:

ORM frameworks typically provide a "unit of work" pattern for managing transactions and database interactions. Changes to objects are tracked by the framework and committed to the database in a single transaction.

7. Query Language:

ORM frameworks often provide a high-level query language or API for constructing database queries using object-oriented syntax. This allows developers to perform CRUD (Create, Read, Update, Delete) operations using familiar programming constructs.

8. Declarative Mapping:

Some ORM frameworks support declarative mapping, where database mappings are defined using class and attribute decorators or configuration files, rather than explicitly writing SQL queries.

Example (using SQLAlchemy):

```
python

from sqlalchemy import Column, Integer, String, ForeignKeyfrom sqlalchemy.orm
import relationshipfrom sqlalchemy.ext.declarative import declarative_base Base =
declarative_base() class Author(Base):    __tablename__ = 'authors'    id =
Column(Integer, primary_key=True)    name = Column(String)    books =
relationship("Book", back_populates="author") class Book(Base):    __tablename__ =
'books'    id = Column(Integer, primary_key=True)    title = Column(String)
author_id = Column(Integer, ForeignKey('authors.id'))    author =
relationship("Author", back_populates="books") # Usagenew_author =
Author(name='John Smith')new_book = Book(title='Python Programming',
author=new_author)
```

In this example, `Author` and `Book` classes are mapped to database tables. The `books` attribute of the `Author` class represents a one-to-many relationship with the `Book` class. The `relationship` function defines the relationship between the classes.

Conclusion:

ORM frameworks provide a powerful way to interact with relational databases using object-oriented programming concepts. By abstracting away the details of database interaction, ORM frameworks make it easier to develop and maintain database-driven applications. However, understanding the underlying concepts is essential for effective use of ORM frameworks.

تصورات ORM

تصورات کا (OOP) ایک پروگرامنگ تکنیک ہے جو ڈویلپرز کو آبجیکٹ اورینٹڈ پروگرامنگ (ORM) آبجیکٹ-ریلیشنل میپنگ فریم ورک، جیسے کہ ORM استعمال کرتے ہوئے رشتہ دار ڈیٹا بیس کے ساتھ بات چیت کرنے کی اجازت دیتی ہے۔ کلاسوں میں ڈیٹا بیس ٹیبلز کا نقشہ بنانے کا طریقہ فراہم کرتے ہیں، اور ان ٹیبلز میں ،SQLAlchemy میں Python تصورات ہیں ORM قطاریں پروگرامنگ لینگویج میں موجود اشیاء کے لیے فراہم کرتے ہیں۔ یہاں کچھ اہم:

1. ہستی:

ORM ایک ہستی ایک حقیقی دنیا کی چیز یا تصور کی نمائندگی کرتی ہے جسے ڈیٹا بیس میں محفوظ کیا جاسکتا ہے۔ میں، اداروں کو عام طور پر ڈیٹا بیس ٹیبلز پر نقش کیا جاتا ہے۔

2. کلاس ٹیبل میپنگ:

فریم ورک ایپلی کیشن میں ہر کلاس کو ڈیٹا بیس میں ایک ٹیبل پر نقشہ بناتا ہے۔ کلاس کے اوصاف ٹیبل کے کالموں ORM سے مطابقت رکھتے ہیں، اور کلاس کی مثالیں ٹیبل میں قطاروں کی نمائندگی کرتی ہیں۔

3. آبجیکٹ-ریلیشنل میپنگ:

فریم ورک پروگرامنگ لینگویج میں موجود اشیاء اور ڈیٹا بیس میں قطاروں کے درمیان تبدیل کرنے کا طریقہ کار فراہم ORM سوالات کی بجائے اشیاء اور طریقوں کا استعمال کرتے ہوئے ڈیٹا بیس کے ساتھ SQL کرتے ہیں، جس سے ڈویلپرز کو خام بات چیت کرنے کی اجازت دیتی ہے۔

4. تعلقات:

فریم ورک اداروں کے درمیان تعلقات کی حمایت کرتے ہیں، جس سے ڈویلپرز کو ڈیٹا کے پیچیدہ ڈھانچے اور اشیاء کے ORM درمیان ایسوسی ایشن کو ماڈل بنانے کی اجازت ملتی ہے۔ تعلقات کی عام اقسام میں ایک سے ایک، ایک سے کئی اور کئی سے کئی تعلقات شامل ہیں۔

5. سست لوڈنگ:

فریم ورک اکثر سست لوڈنگ کی حمایت کرتے ہیں، ایک ایسی تکنیک جہاں متعلقہ اشیاء کو ڈیٹا بیس سے صرف ORM اس وقت لوڈ کیا جاتا ہے جب ان تک ایپلیکیشن کے ذریعے رسائی حاصل کی جاتی ہے، ڈیٹا بیس کے غیر ضروری سوالات کو کم سے کم کرکے کارکردگی کو بہتر بنایا جاتا ہے۔

6. کام کی اکائی:

فریم ورک عام طور پر لین دین اور ڈیٹا بیس کے تعاملات کے انتظام کے لیے "کام کی اکائی" کا نمونہ فراہم کرتے ہیں۔ ORM اشیاء میں ہونے والی تبدیلیوں کو فریم ورک کے ذریعے ٹریک کیا جاتا ہے اور ایک ہی لین دین میں ڈیٹا بیس کے لیے پابند کیا جاتا ہے۔

7. استفسار کی زبان:

فریم ورک اکثر آبجیکٹ پر مبنی نحو کا استعمال کرتے ہوئے ڈیٹا بیس کے استفسارات کی تعمیر کے لیے ایک اعلیٰ ORM CRUD فراہم کرتے ہیں۔ یہ ڈویلپرز کو واقف پروگرامنگ کنسٹرکٹس کا استعمال کرتے ہوئے API سطحی استفسار کی زبان یا آپریشنز انجام دینے کی اجازت دیتا ہے۔ (تخلیق، پڑھیں، اپ ڈیٹ، ڈیلیٹ)

8. اعلانیہ نقشہ سازی:

استفسارات SQL فریم ورک اعلانیہ نقشہ سازی کی حمایت کرتے ہیں، جہاں ڈیٹا بیس میپنگ کو واضح طور پر ORM کچھ لکھنے کے بجائے کلاس اور انتساب ڈیکوریٹرز یا کنفیگریشن فائلوں کا استعمال کرتے ہوئے بیان کیا جاتا ہے۔

مثال (SQLAlchemy کا استعمال کرتے ہوئے):

ازگر

سے درآمد تعلقات کالم(اسٹرنگ) ForeignKeyfrom sqlalchemy.orm ،سے درآمد کالم، عدد، سٹرنگ sqlalchemy انٹیجر،)کالم = id 'کتابوں' = __tablename__ :کتابیں = رشتہ("کتاب"، بیک_پاپولٹس="مصنف") کلاس بک (بیس) مصنف = (('authors.id'ForeignKey , کالم(انٹیجر) = id_عنوان = کالم(اسٹرنگ) مصنف (True=بنیادی_کی رشتہ("Author", back_populates="books") # Usagenew_author = Author(name='John Smith')new_book = Book(title='Python Programming', author=new_author)

اس مثال میں، مصنف اور کتاب کی کلاسز کو ڈیٹا بیس ٹیبلز پر میپ کیا جاتا ہے۔ مصنف طبقے کی کتابوں کی خصوصیت کتاب کی کلاس کے ساتھ ایک سے کئی تعلق کی نمائندگی کرتی ہے۔ رشتہ کا فنکشن کلاسوں کے درمیان تعلق کی وضاحت کرتا ہے۔

نتیجہ:

فریم ورک آبجیکٹ پر مبنی پروگرامنگ تصورات کا استعمال کرتے ہوئے متعلقہ ڈیٹا بیس کے ساتھ بات چیت کرنے کا ORM فریم ورک ڈیٹا بیس سے چلنے ORM ،ایک طاقتور طریقہ فراہم کرتے ہیں۔ ڈیٹا بیس کے تعامل کی تفصیلات کو ختم کرکے فریم ورک کے مؤثر استعمال کے لیے بنیادی تصورات ORM ،والی ایپلی کیشنز کو تیار کرنا اور برقرار رکھنا آسان بناتا ہے۔ تاہم کو سمجھنا ضروری ہے۔

CRUD operations

SQLAlchemy is a powerful Python library for working with databases. It provides an Object-Relational Mapping (ORM) layer that allows you to interact with databases using Python objects, abstracting away the complexities of SQL queries. With SQLAlchemy, you can perform CRUD operations (Create, Read, Update, Delete) easily. Let's go through each operation:

1. Create (C)

To create a new record in a database using SQLAlchemy, you typically follow these steps:

```
python

from sqlalchemy import create_engine, Column, Integer, Stringfrom
sqlalchemy.ext.declarative import declarative_basefrom sqlalchemy.orm import
sessionmaker # Create an engineengine = create_engine('sqlite:///example.db',
echo=True) # Create a base classBase = declarative_base() # Define a modelclass
User(Base):    __tablename__ = 'users'    id = Column(Integer, primary_key=True)
name = Column(String)    age = Column(Integer) # Create tables in the
databaseBase.metadata.create_all(engine) # Create a sessionSession =
sessionmaker(bind=engine)session = Session() # Create a new usernew_user =
User(name='John', age=30)session.add(new_user)session.commit()
```

2. Read (R)

To retrieve records from a database:

```
python

# Retrieve all usersusers = session.query(User).all()for user in users:
print(user.name, user.age) # Retrieve a specific user by IDuser =
session.query(User).filter_by(id=1).first()print(user.name, user.age)
```

3. Update (U)

To update an existing record:

```
python

# Update user's ageuser = session.query(User).filter_by(id=1).first()user.age =
31session.commit()
```

4. Delete (D)

To delete a record:

```
python

# Delete a useruser =
session.query(User).filter_by(id=1).first()session.delete(user)session.commit()
```

These are the basic CRUD operations with SQLAlchemy. They allow you to perform essential database interactions with ease using Python objects and methods provided by SQLAlchemy.

Introduction to Flask framework

Flask is a lightweight and flexible web framework for Python, designed to make it easy to build web applications quickly and with minimal boilerplate code. It's often referred to as a "micro-framework" because it provides the essentials for web development without imposing strict architectural patterns or dependencies. Here's an introduction to Flask:

Key Features:

1. **Lightweight**: Flask is designed to be lightweight and minimalistic, allowing developers to get started quickly without unnecessary complexity.
2. **Modular**: Flask is built around the concept of "extensions," which are reusable components that add functionality to the framework. This modular design allows developers to pick and choose the features they need for their applications.
3. **Routing**: Flask uses a simple and intuitive routing system that maps URL patterns to Python functions (view functions), making it easy to define the behavior of different routes in the application.
4. **Template Engine**: Flask comes with a built-in template engine called Jinja2, which allows developers to generate HTML dynamically by combining Python code with template syntax.
5. **HTTP Request Handling**: Flask provides a request and response object for handling HTTP requests and responses, making it easy to access request data (e.g., form data, query parameters) and generate responses (e.g., HTML, JSON).
6. **Development Server**: Flask includes a built-in development server that makes it easy to run and test applications locally during development.
7. **Extension Ecosystem**: Flask has a rich ecosystem of extensions that provide additional functionality for tasks such as authentication, database integration, form validation, and more.

Example:

Here's a simple "Hello, World!" example using Flask:

```
python

from flask import Flask app = Flask(__name__) @app.route('/')def hello_world():
return 'Hello, World!' if __name__ == '__main__':    app.run(debug=True)
```

In this example:

- We import the `Flask` class from the `flask` module and create a new instance of the `Flask` class, passing `__name__` as the name of the application.
- We define a route using the `@app.route` decorator, which binds the URL pattern `'/'` to the `hello_world()` function.
- The `hello_world()` function returns the string `'Hello, World!'`, which will be displayed in the browser when accessing the root URL.
- Finally, we use the `run()` method to start the Flask development server.

Conclusion:

Flask is a versatile and user-friendly web framework for Python, suitable for building a wide range of web applications, from simple prototypes to large-scale projects. Its simplicity, flexibility, and extensive documentation make it a popular choice among developers for web development in Python.

آپریشنز CRUD

ڈیٹا بیس کے ساتھ کام کرنے کے لیے ایک طاقتور ازگر کی لائبریری ہے۔ یہ ایک آبجیکٹ-ریلیشنل میپنگ SQLAlchemy آبجیکٹ کا استعمال کرتے ہوئے ڈیٹا بیس کے ساتھ تعامل کرنے کی اجازت Python پرت فراہم کرتا ہے جو آپ کو (ORM) CRUD کے ساتھ، آپ آسانی سے SQLAlchemy دیتا ہے، ایس کیو ایل کے سوالات کی پیچیدگیوں کو دور کرتا ہے۔ :آپریشنز (تخلیق، پڑھیں، اپ ڈیٹ، حذف) کر سکتے ہیں۔ آئیے ہر آپریشن سے گزرتے ہیں

1. تخلیق کریں (C)

کا استعمال کرتے ہوئے ڈیٹا بیس میں نیا ریکارڈ بنانے کے لیے، آپ عام طور پر ان مراحل پر عمل کرتے SQLAlchemy :ہیں

ازگر

sqlalchemy سے مرآمد create_engine، انٹیجر، کالم، Stringfrom sqlalchemy.ext.declarative import = مرآمد سیشن میکر # ایک انجن انجن بنائیں declarative_basefrom sqlalchemy.orm ماڈلکلاس یوزر(بیس) # ()declarative_base بیس کریٹ کلاس = ایکو، ('create_engine('sqlite:///example.db __tablename__ = 'users' id = Column(Integer, prime_key=True) name = :کی وضاحت کریں Column(String) age = Column(Integer) # database.metadata.create_all(میں ٹیبلز بنائیں انجن) ایک # سیشن بنائیں سیشن = سیشن میکر(بائنڈ=انجن) سیشن = سیشن () # ایک نیا صارف نیا صارف بنائیں = ()session.add(new_user)session.commitصارف(نام='جان'، عمر=30)

2. پڑھیں (ر)

:ڈیٹا بیس سے ریکارڈ بازیافت کرنے کے لیے

ازگر

print(user.name, :صارفین میں صارف کے لیے()session.query(User).all = تمام صارفین کو بازیافت کریں # user.age) # IDuser = session.query(User) کے ذریعے ایک مخصوص صارف کو بازیافت کریں۔ filter_by(id=1) .first()print(user.name, user.age)

3. اپ ڈیٹ (U)

:موجودہ ریکارڈ کو اپ ڈیٹ کرنے کے لیے

ازگر

()session.query(User).filter_by(id=1).first()user.age = 31session.commit = صارف کی عمر صارف # کو اپ ڈیٹ کریں۔

4. حذف کریں (D)

:ریکارڈ کو حذف کرنے کے لیے

ازگر

= صارف کو حذف کریں # session.query(User).filter_by(id=1).first()session.delete(user)session.commit()

کی طرف سے SQLAlchemy اشیاء اور Python آپریشنز ہیں۔ وہ آپ کو CRUD کے ساتھ بنیادی SQLAlchemy یہ فراہم کردہ طریقوں کا استعمال کرتے ہوئے آسانی کے ساتھ ضروری ڈیٹا بیس کے تعاملات انجام دینے کی اجازت دیتے ہیں۔

فلاسک فریم ورک کا تعارف

کے لیے ایک ہلکا پھلکا اور لچکدار ویب فریم ورک ہے، جو ویب ایپلیکیشنز کو تیزی سے اور کم سے کم Python فلاسک بوائلر پلیٹ کوڈ کے ساتھ بنانا آسان بنانے کے لیے ڈیزائن کیا گیا ہے۔ اسے اکثر "مائیکرو فریم ورک" کہا جاتا ہے کیونکہ یہ سخت آرکیٹیکچرل پیٹرن یا انحصار کو مسلط کیے بغیر ویب ڈویلپمنٹ کے لیے ضروری چیزیں فراہم کرتا ہے۔ یہاں فلاسک کا :تعارف ہے

:اہم خصوصیات

1. ہلکا پھلکا: فلاسک کو ہلکا پھلکا اور کم سے کم بنانے کے لیے ڈیزائن کیا گیا ہے، جس سے ڈویلپرز کو غیر ضروری پیچیدگی کے بغیر تیزی سے کام شروع کرنے کا موقع ملتا ہے۔

2. ماڈیولر: فلاسک "ایکسٹینشنز" کے تصور کے گرد بنایا گیا ہے، جو دوبارہ قابل استعمال اجزاء ہیں جو فریم ورک میں فعالیت کا اضافہ کرتے ہیں۔ یہ ماڈیولر ڈیزائن ڈویلپرز کو اپنی ایپلی کیشنز کے لیے درکار خصوصیات کو چننے اور منتخب کرنے کی اجازت دیتا ہے۔

3. فنکشنز Python روٹنگ: فلاسک ایک سادہ اور بدیہی روٹنگ سسٹم کا استعمال کرتا ہے جو یو آر ایل پیٹرن کو (فنکشنز دیکھیں) سے نقشہ بناتا ہے، جس سے ایپلیکیشن میں مختلف روٹس کے رویے کی وضاحت کرنا آسان ہوجاتا ہے۔

4. ٹیمپلیٹ انجن: فلاسک ایک بلٹ ان ٹیمپلیٹ انجن کے ساتھ آتا ہے جسے Jinja2 کہتے ہیں، جو ڈیولپرز کو HTML کوڈ کو ٹیمپلیٹ نحو کے ساتھ ملا کر متحرک طور پر Python بنانے کی اجازت دیتا ہے۔

5. HTTP درخواست ہینڈلنگ: فلاسک HTTP درخواستوں اور جوابات کو سنبھالنے کے لیے ایک درخواست اور جوابی HTTP چیز فراہم کرتا ہے، جس سے درخواست کے ڈیٹا تک رسائی آسان ہو جاتی ہے (جیسے، فارم ڈیٹا، استفسار کے پیرامیٹرز) اور پیدا کرنا۔ (HTML، JSON ،جیسے) جوابات

6. ڈویلپمنٹ سرور: فلاسک میں ایک بلٹ ان ڈویلپمنٹ سرور شامل ہے جو ترقی کے دوران مقامی طور پر ایپلی کیشنز کو چلانے اور جانچنا آسان بناتا ہے۔

7. ایکسٹینشن ایکو سسٹم: فلاسک میں ایکسٹینشنز کا ایک بھرپور ماحولیاتی نظام ہے جو کہ توثیق، ڈیٹا بیس انٹیگریشن، فارم کی توثیق اور مزید کاموں کے لیے اضافی فعالیت فراہم کرتا ہے۔

:مثال

:یہاں ایک سادہ "ہیلو، ورلڈ!" فلاسک کا استعمال کرتے ہوئے مثال

ازگر

فلاسک درآمد کرنے سے Flask app = Flask(__name__)@app.route('/')def hello_world(): 'ہیلو، ورلڈ!' واپس app.run(debug=True) :'__main__' == __name__ اگر

:اس مثال میں

- ہم فلاسک کلاس کو فلاسک ماڈیول سے درآمد کرتے ہیں اور فلاسک کلاس کی ایک نئی مثال بناتے ہیں، درخواست کو پاس کرتے ہیں۔ __name__ کے نام کے طور پر
- ہم @app.route ڈیکوریٹر کا استعمال کرتے ہوئے ایک روٹ کی وضاحت کرتے ہیں، جو پیٹرن '/' کو URL فنکشن سے منسلک کرتا ہے۔ ()hello_world
- hello_world() فنکشن 'Hello, World!' سٹرنگ واپس کرتا ہے، جو روٹ URL تک رسائی کے وقت براؤزر میں ظاہر ہوگا۔

• طریقہ استعمال کرتے ہیں۔ ()run آخر میں، ہم فلاسک ڈویلپمنٹ سرور کو شروع کرنے کے لیے

نتیجہ:

کے لیے ایک ورسٹائل اور صارف دوست ویب فریم ورک ہے، جو سادہ پروٹو ٹائپ سے لے کر بڑے پیمانے Python فلاسک
پر پروجیکٹس تک ویب ایپلیکیشنز کی ایک وسیع رینج بنانے کے لیے موزوں ہے۔ اس کی سادگی، لچک، اور وسیع
میں ویب ڈویلپمنٹ کے لیے ڈویلپرز کے درمیان ایک مقبول انتخاب بناتی ہیں۔ Python دستاویزات اسے

Creating web applications

Creating web applications with Flask involves several steps, including setting up a Flask project, defining routes, handling requests and responses, and integrating with templates and databases. Here's a step-by-step guide to creating a simple web application with Flask:

1. Install Flask:

If you haven't already installed Flask, you can do so using pip:

```
pip install Flask
```

2. Create a Flask App:

Create a new Python file for your Flask application. This will serve as the entry point for your web application.

```
python

from flask import Flask app = Flask(__name__) @app.route('/')def hello_world():
return 'Hello, World!' if __name__ == '__main__':    app.run(debug=True)
```

3. Define Routes:

Define routes using the `@app.route` decorator to map URL patterns to view functions. View functions are responsible for generating responses to incoming requests.

```
python

@app.route('/')def hello_world():    return 'Hello, World!' @app.route('/about')def
about():    return 'This is the about page'
```

4. Run the Flask App:

Run the Flask app using the `run()` method. This starts the development server, allowing you to access the application in a web browser.

```
python

if __name__ == '__main__':    app.run(debug=True)
```

5. Access the Application:

Open a web browser and navigate to `http://localhost:5000` to access the home page of your Flask application. You can also access other routes defined in your application by appending the corresponding URL path (e.g., `http://localhost:5000/about`).

6. Templates:

Create HTML templates for rendering dynamic content in your Flask application. Use the `render_template()` function to render templates and pass data to them.

```
python

from flask import render_template @app.route('/greet/<name>')def greet(name):
return render_template('greet.html', name=name)
```

7. Forms:

Handle form submissions in your Flask application using the `request` object. Access form data using `request.form` and perform necessary processing or validation.

```
python

from flask import request @app.route('/submit', methods=['POST'])def submit():
name = request.form['name']    return f'Hello, {name}!'
```

8. Static Files:

Serve static files such as CSS, JavaScript, and images by placing them in a directory named `static` in your Flask project directory.

9. Database Integration:

Integrate Flask with databases using extensions such as Flask-SQLAlchemy, Flask-MongoEngine, or Flask-PyMongo. Define database models, perform CRUD operations, and interact with the database within your Flask application.

10. Deployment:

Deploy your Flask application to a production server using a web server such as Gunicorn or uWSGI. Consider deploying to platforms such as Heroku, AWS, or Google Cloud Platform for scalability and reliability.

Conclusion:

Flask provides a simple yet powerful framework for building web applications in Python. By following the steps outlined above, you can create a basic Flask application and gradually add features such as templates, forms, and database integration to build more complex and dynamic web applications.

ویب ایپلیکیشنز بنانا

فلاسک کے ساتھ ویب ایپلیکیشنز بنانے میں کئی مراحل شامل ہیں، بشمول فلاسک پروجیکٹ کو ترتیب دینا،راستوں کی Flask وضاحت کرنا، درخواستوں اور جوابات کو سنبھالنا، اور ٹیمپلیٹس اور ڈیٹا بیس کے ساتھ انضمام کرنا۔ یہاں :ایک سادہ ویب ایپلیکیشن بنانے کے لیے مرحلہ وار گائیڈ ہے

:فلاسک انسٹال کریں .1

:کا استعمال کر کے ایسا کر سکتے ہیں pip اگر آپ نے پہلے سے فلاسک انسٹال نہیں کیا ہے، تو آپ

پائپ انسٹال فلاسک

:ایک فلاسک ایپ بنائیں .2

فائل بنائیں۔ یہ آپ کی ویب ایپلیکیشن کے لیے داخلے کے مقام کے طور Python اپنی فلاسک ایپلیکیشن کے لیے ایک نئی پر کام کرے گا۔

ازگر

واپس 'ہیلو، ورلڈ!' :()def hello_world('/')@app.route(__name__)Flask app = Flask فلاسک مرآمد کرنے سے (debug=True)app.run: '__main__' == __name__ اگر

:راستوں کی وضاحت کریں .3

ڈیکوریٹر کا استعمال کرتے ہوئے راستوں کی @app.route پیٹرن کا نقشہ بنانے کے لیے URL فنکشنز دیکھنے کے لیے وضاحت کریں۔ دیکھنے کے افعال آنے والی درخواستوں کے جوابات پیدا کرنے کے ذمہ دار ہیں۔

ازگر

یہ صفحہ کے بارے' :()def about('/about')@app.route @ 'واپسی 'ہیلو، ورلڈ :()def hello_world('/')@app.route میں ہے' واپس کریں

:فلاسک ایپ چلائیں .4

طریقہ استعمال کرکے فلاسک ایپ چلائیں۔ اس سے ڈویلپمنٹ سرور شروع ہوتا ہے، جس سے آپ ویب براؤزر میں run() ایپلیکیشن تک رسائی حاصل کر سکتے ہیں۔

ازگر

(debug=True)app.run: '__main__' == __name__ اگر

:درخواست تک رسائی حاصل کریں .5

پر جائیں۔ http://localhost:5000 ایک ویب براؤزر کھولیں اور اپنی فلاسک ایپلیکیشن کے ہوم پیج تک رسائی کے لیے کو شامل کرکے اپنی درخواست میں بیان کردہ (http://localhost:5000/about ،مثال کے طور پر) پاتھ URL آپ متعلقہ دیگرراستوں تک بھی رسائی حاصل کرسکتے ہیں۔

:ٹیمپلیٹس .6

ٹیمپلیٹس بنائیں۔ ٹیمپلیٹس کو رینڈر کرنے اور ان تک HTML اپنی فلاسک ایپلیکیشن میں متحرک مواد پیش کرنے کے لیے فنکشن کا استعمال کریں۔ ()render_template ڈیٹا منتقل کرنے کے لیے

ازگر

واپسی :(def greet(name)@app.route('/greet/<name>') render_template فلاسک سے رآمد کریں
render_template('greet.html', name=name)

:فارم .7

کا request.form درخواست آبجیکٹ کا استعمال کرتے ہوئے اپنی فلاسک ایپلیکیشن میں فارم جمع کرانے کو ہینڈل کریں۔ استعمال کرتے ہوئے فارم ڈیٹا تک رسائی حاصل کریں اور ضروری پروسیسنگ یا توثیق کریں۔

ازگر

@app.route('/submit', methods=['POST'])def submit(): name = فلاسک رآمد کی درخواست سے
f'Hello, {name}!' واپسی request.form['name']

:جامد فائلیں .8

نام کی ڈائریکٹری میں static اور تصاویر کو اپنی فلاسک پروجیکٹ ڈائرکٹری میں ،JavaScript ،CSS جامد فائلوں جیسے رکھ کر پیش کریں۔

:ڈیٹا بیس انٹیگریشن .9

جیسے ایکسٹینشنز کا استعمال کرتے ہوئے Flask-PyMongo یا ،Flask-MongoEngine ،Flask-SQLAlchemy آپریشنز کریں، اور اپنی فلاسک CRUD ،کو ڈیٹا بیس کے ساتھ ضم کریں۔ ڈیٹا بیس ماڈلز کی وضاحت کریں Flask ایپلیکیشن کے اندر موجود ڈیٹا بیس کے ساتھ تعامل کریں۔

:تعیناتی .10

کا استعمال کرتے ہوئے اپنی فلاسک ایپلیکیشن کو پروڈکشن سرور پر تعینات uWSGI یا Gunicorn ویب سرور جیسے جیسے پلیٹ فارمز پر تعینات Google Cloud Platform یا ،AWS ،کریں۔ توسیع پذیری اور بھروسے کے لیے ہیروکو کرنے پر غور کریں۔

:نتیجہ

میں ویب ایپلیکیشنز بنانے کے لیے ایک سادہ لیکن طاقتور فریم ورک فراہم کرتا ہے۔ اوپر بیان کیے گئے Python فلاسک اقدامات پر عمل کرتے ہوئے، آپ ایک بنیادی فلاسک ایپلیکیشن بنا سکتے ہیں اور آہستہ آہستہ مزید پیچیدہ اور متحرک ویب ایپلیکیشنز بنانے کے لیے ٹیمپلیٹس، فارمز، اور ڈیٹا بیس انٹیگریشن جیسی خصوصیات شامل کر سکتے ہیں۔

Routing and views

Routing and views are fundamental concepts in Flask for defining URL patterns and handling requests. In Flask, routes are defined using the `@app.route` decorator, and views are Python functions associated with specific routes. Here's how routing and views work in Flask:

1. Define Routes:

Routes are defined using the `@app.route` decorator, which binds a URL pattern to a view function. The URL pattern specifies the path at which the view function will be invoked.

```
python

from flask import Flask app = Flask(__name__) @app.route('/')def home():    return
'Home Page' @app.route('/about')def about():    return 'About Page'
```

In this example, the `/` route maps to the `home()` view function, and the `/about` route maps to the `about()` view function.

2. Dynamic Routes:

Flask supports dynamic routes, allowing for variable components in the URL pattern. Dynamic components are specified using `<variable_name>` syntax in the route pattern.

```
python

@app.route('/user/<username>')def profile(username):    return f'Profile Page:
{username}'
```

In this example, the `/user/<username>` route maps to the `profile()` view function, and the value of the `username` variable is extracted from the URL and passed as an argument to the view function.

3. URL Building:

Flask provides the `url_for()` function for generating URLs based on route names and arguments. This allows you to create links between different parts of your application without hardcoding URLs.

```
python

from flask import url_for @app.route('/')def home():    return f'<a
href="{url_for("about")}">About</a>'
```

4. HTTP Methods:

Views can handle different HTTP methods (GET, POST, etc.) by specifying the methods parameter in the `@app.route` decorator.

```
python
```

```
@app.route('/login', methods=['GET', 'POST'])def login():    if request.method ==
'POST':        # Handle login form submission        pass    else:        # Display
login form        pass
```

5. Error Handling:

Flask allows you to define error handlers to handle specific HTTP error codes or exceptions that occur during request processing.

```
python

@app.errorhandler(404)def page_not_found(e):    return 'Page Not Found', 404
```

6. Redirects:

Flask provides the `redirect()` function for redirecting users to a different URL within the application.

```
python

from flask import redirect @app.route('/old-url')def old_url():    return
redirect(url_for('new_url'))
```

Conclusion:

Routing and views are essential components of Flask applications for defining URL patterns and handling requests. By defining routes and associating them with view functions, you can create a navigation structure and define the behavior of different parts of your application. Understanding routing and views is key to building Flask applications effectively.

روٹنگ اور نظارے۔

روٹنگ اور ویوز فلاسک میں یو آر ایل پیٹرن کی وضاحت اور درخواستوں کو سنبھالنے کے لیے بنیادی تصورات ہیں۔ فلاسک ڈیکوریٹر کا استعمال کرتے ہوئے راستوں کی وضاحت کی جاتی ہے، اور ویوز مخصوص راستوں سے @app.route ،میں فنکشنز ہیں۔ یہ ہے کہ فلاسک میں روٹنگ اور ویوز کیسے کام کرتے ہیں Python وابستہ:

1. راستوں کی وضاحت کریں:

ڈیکوریٹر کا استعمال کرتے ہوئے کی جاتی ہے، جو یو آر ایل پیٹرن کو ویو فنکشن سے @app.route راستوں کی وضاحت پیٹرن اس راستے کی وضاحت کرتا ہے جس پر ویو فنکشن کو طلب کیا جائے گا۔ URL منسلک کرتا ہے۔

ازگر

Flask app = Flask(__name__)@app.route('/')def home(): 'واپس 'ہوم پیج فلاسک درآمد کرنے سے
@app.route('/about')def about(): 'واپس 'صفحہ کے بارے میں

ویو فنکشن کے about() اس مثال میں، گھر () ویو فنکشن کے لیے / روٹ کے نقشے، اور / کے بارے میں روٹ کے نقشے لیے۔

2. متحرک راستے:

فلاسک یو آر ایل پیٹرن میں متغیر اجزاء کی اجازت دیتے ہوئے متحرک راستوں کو سپورٹ کرتا ہے۔ روٹ پیٹرن میں نحو کا استعمال کرتے ہوئے متحرک اجزاء کی وضاحت کی گئی ہے۔ <variable_name>

ازگر

@app.route('/user/<username>')def profile(username): واپس f'Profile Page: {username}'

روٹ کا نقشہ پروفائل() ویو فنکشن پر ہوتا ہے، اور یوزر نیم متغیر کی قدر یو آر ایل /user/<username> ،اس مثال میں سے نکالی جاتی ہے اور ویو فنکشن کے لیے دلیل کے طور پر پاس کی جاتی ہے۔

3. یو آر ایل کی تعمیر:

کو URLs فنکشن فراہم کرتا ہے۔ یہ آپ کو url_for() فلاسک روٹ کے ناموں اور دلائل کی بنیاد پر یو آر ایل بنانے کے لیے ہارڈ کوڈنگ کے بغیر اپنی درخواست کے مختلف حصوں کے درمیان لنکس بنانے کی اجازت دیتا ہے۔

ازگر

flask import url_for @app.route('/')def home(): واپس f'کے بارے میں'

4. HTTP طریقے:

(وغیرہ GET، POST، HTTP ڈیکوریٹر میں طریقوں کے پیرامیٹر کی وضاحت کرکے ویوز مختلف @app.route کو سنبھال سکتے ہیں۔

ازگر

@app.route('/login', methods=['GET', 'POST'])def login(): if request.method == 'POST': # لاگ ان فارم جمع کرانے کا پاس ہینڈل کریں اور: # لاگ ان فارم پاس دکھائیں

5. خرابی سے نمٹنے:

ایرر کوڈز یا استثنیٰ کو ہینڈل کرنے کے لیے ایرر ہینڈلرز کی وضاحت کرنے کی اجازت دیتا HTTP فلاسک آپ کو مخصوص ہے جو درخواست کی کارروائی کے دوران پائے جاتے ہیں۔

ازگر

@app.errorhandler(404)def page_not_found(e): 'Page Not Found'، 404 واپس کریں

6. ری ڈائریکٹ:

فلاسک صارفین کو ایپلی کیشن کے اندر ایک مختلف یو آر ایل پر ری ڈائریکٹ کرنے کے لیے ری ڈائریکٹ() فنکشن فراہم کرتا ہے۔

ازگر

@app.route('/old-url')def old_url(): ری ڈائریکٹ(url_for('new_url')) فلاسک امپورٹ ری ڈائریکٹ سے

نتیجہ:

یو آر ایل پیٹرن کی وضاحت اور درخواستوں کو سنبھالنے کے لیے روٹنگ اور ویوز فلاسک ایپلی کیشنز کے ضروری اجزاء ہیں۔ راستوں کی وضاحت کرکے اور انہیں ویو فنکشنز سے جوڑ کر، آپ نیویگیشن ڈھانچہ بنا سکتے ہیں اور اپنی ایپلیکیشن کے مختلف حصوں کے رویے کی وضاحت کر سکتے ہیں۔ فلاسک ایپلی کیشنز کو مؤثر طریقے سے بنانے کے لیے روٹنگ اور ویوز کو سمجھنا کلید ہے۔

Anaconda distribution

Anaconda is a popular open-source distribution of the Python and R programming languages for scientific computing, data science, and machine learning tasks. It aims to simplify package management and deployment by providing a comprehensive ecosystem of tools and libraries for data analysis, numerical computing, visualization, and machine learning.

Here are some key features and components of Anaconda:

1. Conda Package Manager:

Anaconda includes the `conda` package manager, which simplifies the installation and management of software packages and environments. Conda allows users to easily install, update, and remove packages, as well as create and manage isolated environments with specific package dependencies.

2. Package Ecosystem:

Anaconda provides a vast collection of pre-built packages for scientific computing, data analysis, machine learning, and more. These packages include popular libraries such as NumPy, Pandas, Matplotlib, SciPy, scikit-learn, TensorFlow, and PyTorch, among others.

3. Integrated Development Environment (IDE):

Anaconda includes the Anaconda Navigator, a graphical user interface (GUI) that allows users to manage environments, install packages, and launch applications. Additionally, Anaconda can be integrated with popular code editors and IDEs such as Jupyter Notebook, JupyterLab, Spyder, and VS Code, providing a seamless development experience.

4. Cross-Platform Support:

Anaconda is available for Windows, macOS, and Linux operating systems, making it accessible to a wide range of users across different platforms.

5. Data Science Libraries and Tools:

Anaconda is tailored for data science and provides a comprehensive set of tools and libraries for data manipulation, analysis, visualization, and machine learning. This includes support for data formats such as CSV, JSON, Excel, and HDF5, as well as tools for statistical analysis and visualization.

6. Community Support and Resources:

Anaconda has a large and active community of users and developers who contribute to its development, provide support, and share resources such as tutorials, documentation, and code examples.

Overall, Anaconda is a powerful platform that streamlines the setup and configuration of Python environments for scientific computing and data analysis. It has become a popular choice among data scientists, researchers, and developers for its ease of use, comprehensive package ecosystem, and robust features for data-driven workflows.

اناکونڈا کی تقسیم

پروگرامنگ زبانوں کی ایک R اور Python اناکونڈا سائنسی کمپیوٹنگ، ڈیٹا سائنس، اور مشین لرننگ کے کاموں کے لیے مقبول اوپن سورس تقسیم ہے۔ اس کا مقصد ڈیٹا کے تجزیہ، عددی کمپیوٹنگ، ویژولائزیشن، اور مشین لرننگ کے لیے ٹولز اور لائبریریوں کا ایک جامع ماحولیاتی نظام فراہم کرکے پیکیج کے انتظام اور تعیناتی کو آسان بنانا ہے۔

:اناکونڈا کی کچھ اہم خصوصیات اور اجزاء یہ ہیں

:کونڈا پیکیج مینیجر .1

اناکونڈا میں کونڈا پیکیج مینیجر شامل ہے، جو سافٹ ویئر پیکجوں اور ماحول کی تنصیب اور انتظام کو آسان بناتا ہے۔ کونڈا صارفین کو آسانی سے پیکجز کو انسٹال، اپ ڈیٹ اور ہٹانے کے ساتھ ساتھ مخصوص پیکیج پر انحصار کے ساتھ الگ تھلگ ماحول بنانے اور ان کا نظم کرنے کی اجازت دیتا ہے۔

:پیکیج ماحولیاتی نظام .2

اناکونڈا سائنسی کمپیوٹنگ، ڈیٹا تجزیہ، مشین لرننگ، اور بہت کچھ کے لیے پہلے سے تیار کردہ پیکجوں کا ایک وسیع ذخیرہ فراہم کرتا ہے۔ ان پیکجوں میں مشہور لائبریریاں جیسے NumPy، Pandas، Matplotlib، SciPy، scikit-learn، TensorFlow، اور PyTorch شامل ہیں۔

:(IDE) مربوط ترقیاتی ماحول .3

جو صارفین کو ماحول کا انتظام کرنے، پیکجوں (GUI) اناکونڈا میں اناکونڈا نیویگیٹر شامل ہے، ایک گرافیکل یوزر انٹرفیس جیسے IDEs کو انسٹال کرنے اور ایپلیکیشنز لانچ کرنے کی اجازت دیتا ہے۔ مزید برآں، اناکونڈا کو مقبول کوڈ ایڈیٹرز اور کے ساتھ ضم کیا جا سکتا ہے، جو بغیر کسی رکاوٹ کے VS Code اور Jupyter Notebook، JupyterLab، Spyder، ترقی کا تجربہ فراہم کرتا ہے۔

:کراس پلیٹ فارم سپورٹ .4

اناکونڈا ونڈوز، میک او ایس، اور لینکس آپریٹنگ سسٹمز کے لیے دستیاب ہے، جو اسے مختلف پلیٹ فارمز کے صارفین کی وسیع رینج کے لیے قابل رسائی بناتا ہے۔

:ڈیٹا سائنس لائبریریاں اور اوزار .5

اناکونڈا ڈیٹا سائنس کے لیے تیار کیا گیا ہے اور ڈیٹا میں ہیرا پھیری، تجزیہ، ویژولائزیشن، اور مشین لرننگ کے لیے ٹولز اور جیسے ڈیٹا فارمیٹس کے لیے HDF5 اور CSV، JSON، Excel، لائبریریوں کا ایک جامع سیٹ فراہم کرتا ہے۔ اس میں معاونت کے ساتھ ساتھ شماریاتی تجزیہ اور تصور کے لیے ٹولز شامل ہیں۔

:کمیونٹی سپورٹ اور وسائل .6

اناکونڈا میں صارفین اور ڈویلپرز کی ایک بڑی اور فعال کمیونٹی ہے جو اس کی ترقی میں تعاون کرتے ہیں، معاونت فراہم کرتے ہیں، اور وسائل کا اشتراک کرتے ہیں جیسے کہ سبق، دستاویزات، اور کوڈ کی مثالیں۔

مجموعی طور پر، اناکونڈا ایک طاقتور پلیٹ فارم ہے جو سائنسی کمپیوٹنگ اور ڈیٹا کے تجزیہ کے لیے ازگر کے ماحول کے سیٹ اپ اور کنفیگریشن کو ہموار کرتا ہے۔ یہ ڈیٹا سائنسدانوں، محققین، اور ڈویلپرز کے درمیان اس کے استعمال میں آسانی، جامع پیکیج ایکو سسٹم، اور ڈیٹا سے چلنے والے ورک فلو کے لیے مضبوط خصوصیات کے لیے ایک مقبول انتخاب بن گیا ہے۔

Contact information :

Author : Khawar Nehal

Web : http://atrc.net.pk

Email : khawar@atrc.net.pk

Phone : +92 343 270 2932

رابطے کی معلومات:

مصنف: خاور نہال

ویب سائٹ: http://atrc.net.pk

ای میل: khawar@atrc.net.pk

فون:

+92 343 270 2932