CloudCommand

Cyber Security

CloudCommand GmbH chr.schumacher@gmx.tm

CloudCommand

Grundlagen der Programmierung

CloudCommand GmbH chr.schumacher@gmx.tm

Funktionen und Module

Grundlagen

Definition und Zweck:

- Funktionen sind wiederverwendbare Codeblöcke, die eine bestimmte Aufgabe ausführen.
- Sie helfen dabei, den Code übersichtlicher, modularer und wiederverwendbar zu gestalten.

Erstellen von Funktionen

Syntax:

```
def funktionsname(parameter):
    # Körper der Funktion
    return wert
```

- Beispiel:

```
def gruß(name):
    return f"Hallo, {name}!"
```

Aufruf von Funktionen

Syntax:

```
def gruß(name):
    return f"Hallo, {name}!"

nachricht = gruß("Anna")
print(nachricht)

# Ausgabe: Hallo, Anna!
```

Parameter und Argumente

Funktionen können Parameter annehmen, die beim Aufruf mit Werten (Argumenten) belegt werden.

- **Positionale Argumente** müssen in der Reihenfolge, wie sie definiert sind, übergeben werden.
- **Schlüsselwortargumente** erlauben das Überspringen der Reihenfolge durch explizite Namensnennung.

Parameter und Argumente

Code-Beispiel - Schlüsselwortargumente:

```python
def erstelle_profil(vorname, nachname, **weitere_infos):
    profil = {
        'Vorname': vorname,
        'Nachname': nachname
    }
    for key, value in weitere_infos.items():
        profil[key] = value
    return profil

# Funktionsaufruf mit Schlüsselwortargumenten

benutzer_profil = erstelle_profil('Maria', 'Müller', alter=28, wohnort='Berlin', beruf='Softwareentwicklerin')

print(benutzer_profil)
```

Parameter und Argumente

Code-Beispiel - Positionale Argumente:

```python
def berechne_durchschnitt(*zahlen):

    summe = sum(zahlen)  # Summiert alle übergebenen Zahlen
    anzahl = len(zahlen)  # Zählt, wie viele Zahlen übergeben wurden
    durchschnitt = summe / anzahl if anzahl > 0 else 0
    return durchschnitt

# Funktionsaufruf mit Positionalen Argumenten

durchschnitt = berechne_durchschnitt(10, 20, 30, 40, 50)

print(f"Der Durchschnitt der Zahlen ist: {durchschnitt}")
```

Standardwerte für Parameter

Verwendung von Standardwerten:

```
def gruessen(name, nachricht="Guten Tag"):
    print(f"{nachricht}, {name}!")
```

Vorteil:

- Flexibilität bei Funktionsaufrufen.

Standardwerte für Parameter

- Aufruf mit Standardanrede:

```
gruß("Max")
# Hallo, Max!
```

- Aufruf mit spezifischer Anrede:

```
gruß("Max", anrede="Guten Tag")
# Guten Tag, Max!
```

Funktionen mit mehreren Parametern

Erklärung:

- Reihenfolge und Anzahl der Argumente beim Aufruf.

```
def voller_name(vorname, nachname):
    return f"{vorname} {nachname}"
```

Keyword-Argumente

Erklärung:

- Argumente, die beim Aufruf explizit benannt werden.

```
def anmelden(benutzername, passwort, admin=False):
    print("Benutzername:", benutzername)
    if admin:
        print("Zugriff auf Admin-Bereich")
```

- Aufruf mit Keyword-Argument

```
anmelden(benutzername="Alice", admin=True)
```

Variable Argumentenlisten

Arbiträre Argumente:

- Funktionen, die eine unbestimmte Anzahl an Argumenten akzeptieren.
- Beispiel mit ***args** und ****kwargs**

```
def mehrere_items(*items, **details):
    print("Items:", items)
    print("Details:", details)
```

Anonyme Funktionen: lambda

Verwendungszweck:

- Einfache, einmalige Funktionen ohne Namen.

```
quadrieren = lambda x: x * x
print(quadrieren(5))
```

Funktionsdokumentation

Dokumentationsstrings (docstrings):

- Erklärung der Funktion direkt im Code, ähnlich wie ein Kommentar.

```
def multiplizieren(x, y):

    """
    Multipliziert zwei Zahlen und gibt das Ergebnis zurück.
    """

    return x * y
```

Einführung in die Main-Funktion

Definition:

- Die Main-Funktion als zentraler Einstiegspunkt eines Python-Programms.

Zweck:

- Steuerung des Programmablaufs.

Warum eine Main-Funktion verwenden?

Strukturierung:

- Hilft, Code sauber und organisiert zu halten.

Wiederverwendbarkeit:

- Ermöglicht das einfache Importieren und Wiederverwenden von Code als Modul.

Die Rolle von __name__

```python
def main():
    print("Hallo Welt")

if __name__ == "__main__":
    main()
```

Zweck von __name__:

- Bestimmt, wie das Skript ausgeführt wird (als Programm oder als Modul).

Standardwert von __name__:

- __main__, wenn das Skript direkt ausgeführt wird.

Argumente in der Main-Funktion

Übergeben von Argumenten:

- Nutzung von **sys.argv** für Konsolenargumente.

```python
import sys
def main(args):
    for arg in args:
        print(arg)

if __name__ == "__main__":
    main(sys.argv[1:])
```

Nutzung von argparse für komplexere Argumente

Vorteile von argparse:

- Bessere Verwaltung und Validierung von Kommandozeilenargumenten.

```python
import argparse

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--name', type=str, required=True)
    args = parser.parse_args()
    print(f"Hello, {args.name}!")

if __name__ == "__main__":
    main()
```

Fehlerbehandlung in der Main-Funktion

Umgang mit Ausnahmen:

- Einsatz von **try** und **except** Blöcken zur Fehlerbehandlung.

```
def main():
    try:
        # Riskanter Code
        print(1 / 0)
    except ZeroDivisionError:
        print("Division durch Null ist nicht erlaubt!")

if __name__ == "__main__":
    main()
```

Testen der Main-Funktion

Einrichtung von Unit Tests:

- Einsatz von **unittest** zum Testen der Funktionalität.

```python
import unittest

def add(x, y):
    return x + y

class TestAddition(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(add(3, 4), 7)

if __name__ == '__main__':
    unittest.main()
```

Was ist ein Modul?

Definition:

- Ein Modul ist eine Datei, die Python-Definitionen und Anweisungen enthält. Die Dateiendung ist **.py**

Nutzung:

- Nutzung: Module werden genutzt, um Funktionen, Klassen und Variablen zu gruppieren, die logisch zusammengehören.

Was ist ein Modul?

Vorteile:

- **Wiederverwendbarkeit:** Code kann in verschiedenen Programmen wiederverwendet werden, ohne ihn neu schreiben zu müssen.
- **Namensräume:** Module helfen, Namenskonflikte in großen Projekten zu vermeiden.
- **Übersichtlichkeit:** Modularer Code ist einfacher zu lesen und zu warten.

Importieren von Modulen

Grundlagen des Imports:

- Ein Modul wird mit dem Schlüsselwort import eingebunden.
- Beispiel:

```
import math
```

Teilimport:

- Spezifische Elemente eines Moduls können importiert werden, um den Namensraum sauber zu halten.
- Beispiel:

```
from datetime import datetime
```

Importieren von Modulen

Aliasing:

- Module können beim Importieren umbenannt werden, um die Nutzung zu vereinfachen.
- Beispiel:

```
import numpy as np
```

Was ist ein Teilmodul?

Definition:

- Teilmodule sind Module, die innerhalb eines Pakets organisiert sind.

Beispiel:

- Das **os**-Modul enthält mehrere Teilmodule wie **os.path**.

Import von Teilmodulen:

```
from os import path
import os.path as opath
```

Was ist die Standardbibliothek?

Definition:

- Eine Sammlung von Modulen und Paketen, die standardmäßig mit Python installiert werden.

Beispiele für häufig verwendete Module:

- **sys:** Zugriff auf einige Variablen und Funktionen, die eng mit dem Python-Interpreter zusammenarbeiten.
- **os:** Bietet eine Möglichkeit, standardabhängige Funktionalität wie das Lesen oder Schreiben von Dateien zu nutzen.

Was ist die Standardbibliothek?

Nutzung der Standardbibliothek:

- Keine zusätzliche Installation erforderlich.
- Hilft, die Portabilität des Codes zu gewährleisten.

Mathematische Operationen

Modul math:

- Stellt mathematische Funktionen und Konstanten bereit.

Beispiele für Funktionen:

- **math.sqrt(x):** Berechnet die Quadratwurzel von x.
- **math.pi:** Die mathematische Konstante π.

```
import math

print("Die Quadratwurzel von 16 ist:", math.sqrt(16))
print("Der Wert von Pi ist etwa:", math.pi)
```

Datum und Zeit

Modul datetime:

- Bietet Klassen für die Manipulation von Datum und Zeit.

Häufige Verwendungen:

- Erzeugung von Datums- und Zeitobjekten.
- Berechnung von Zeitdifferenzen.

```
from datetime import datetime

jetzt = datetime.now()
print("Aktuelles Datum und Uhrzeit:", jetzt)
```

Pfade verwalten

Modul os:

- Ermöglicht die Interaktion mit dem Betriebssystem, insbesondere für Datei- und Verzeichnisoperationen.

Funktionen:

- **os.mkdir(path):** Erstellt ein neues Verzeichnis.
- **os.rename(src, dst):** Benennt eine Datei oder ein Verzeichnis um.

```
import os

os.mkdir('neues_verzeichnis')
os.rename('neues_verzeichnis', 'umbenanntes_verzeichnis')
```

Pfade verwalten

Modul pathlib:

- Modernere Schnittstelle zur Pfadmanipulation, objektorientiert.

Vorteil gegenüber os.path:

- Elegantere und intuitivere Methoden für häufige Pfadoperationen.

```
from pathlib import Path

p = Path('.')
for datei in p.glob('*.txt'):
    print(datei)
```

Datenkompression

Modul zlib, gzip:

- Ermöglichen die Kompression und Dekompression von Daten mit verschiedenen Algorithmen

Verwendung:

- Daten effizienter speichern oder übertragen.

```
import gzip

content = "Viel Text hier".encode('utf-8')
with gzip.open('file.txt.gz', 'wb') as f:
    f.write(content)
```

Datenkompression

Modul requests:

- Erlaubt das Senden von HTTP-Anfragen leicht zu handhaben.

Vorteile:

- Einfacher Zugriff auf Webseiten, APIs usw.
- Unterstützt alle HTTP-Methoden.

```python
import requests

response = requests.get('https://api.example.com/data')
if response.status_code == 200:
    print('Daten erfolgreich abgerufen:', response.json())
else:
    print('Fehler beim Abrufen der Daten')
```

CloudCommand

CloudCommand GmbH chr.schumacher@gmx.tm