



TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Koroutine, Threads und Asyncio

Studienarbeit im FWPM Skriptsprache Python

vorgelegt von

Christoph Kirschner

Matrikelnummer: 3248161

Ausgabe:	23.09.2021
Abgabe:	03.10.2021
Prüfer:	Prof. Dr.-Ing. Jürgen Krumm
Zweitprüfer:	Prof. Dr. Matthias Wiczorek

Hinweis: Diese Erklärung ist in alle Exemplare der Prüfungsarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Kirschner

Vorname: Christoph

Matrikel-Nr.: 3248161

Fakultät: EFI

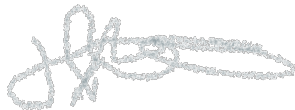
Studiengang: BEI

Semester: 7

Titel der Prüfungsarbeit: Koroutine, Threads und Asyncio

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg, 24.09.2021



Ort, Datum, Unterschrift Studierende/Studierender

Die Abgabe dieser Arbeit gilt als Anmeldung zur vorgezogene Prüfung im FWPF Skriptsprache Python. Für die reibungslose Übertragung der Note ist eine nachträgliche Prüfungsanmeldung in diesem Fach erforderlich.

Inhaltsverzeichnis

1	Einleitung	1
2	Konzepte	2
2.1	Subroutinen	2
2.2	Koroutinen	3
2.3	Threads	3
2.4	Asyncio	4
3	Beispiele	5
3.1	Koroutinen	5
3.2	Threads	6
3.3	Asyncio	8
4	Fazit	10
	Auflistungsverzeichnis	11
	Literatur	12

Kapitel 1

Einleitung

Die Welt der Technik befindet sich seit ihrem Beginn im ständigen Wandel. Neben immer komplexer und kleiner werdender Hardware steigen auch die Anforderung stetig bezüglich leistungsfähiger und flexibler Software. Besonders die Verarbeitungsgeschwindigkeit und Reaktionsfähigkeit von Software-Systemen rückten dabei zunehmend in den Fokus. So haben in den Anfangszeiten der Informatik gerade sequentiell oder seriel abarbeitende Programme Anwendung gefunden. Dies bedeutet, dass jede Funktion nacheinander Zeile für Zeile abgearbeitet wurde. Die Abarbeitung entspricht einem linearen Vorgehen. Für viele Anwendungen ist dieses Verhalten jedoch gänzlich ungeeignet. Somit etablierte sich neben der sequentiellen Abarbeitung zunehmend die parallele oder quasiparallele Durchführung. Gerade bei den heutigen Mikrocontroller- oder Prozessor-Familien werden mit geschickten Scheduling-Mechanismen die vielfältig einsetzbaren eingebetten Systeme, die beinahe in jedem Bereich des Alltags vorgedrungen sind, erst möglich. In Python sind diese Werkzeuge in Form von Koroutinen, Threads oder Asnycio auf Programmiererebene gegeben. Sinnvoll und bedacht eingesetzt, können sie merkliche Geschwindigkeits- und Flexibilitätsvorteile für den Anwender erbringen. Neben der grundlegenden theoretischen Erläuterung der verschiedenen Konzepte sollen durch die nähere Betrachtung eines exemplarischen Anwendungsfalls die Zusammenhänge besser verdeutlicht werden.

Kapitel 2

Konzepte

2.1 Subroutinen

Bei klassischen Subroutinen findet ein statisches Scheduling Anwendung. Dies bedeutet, dass die Ausführ-Reihenfolge als geradliniger Ablauf vordefiniert ist. Ein Abweichen von dieser Reihenfolge ist ohne Programmänderung oder IF-Verzweigung nachträglich nicht mehr möglich. Oft wird diese Art von Routinen für zeitgesteuerte oder synchron ablaufende Vorgänge verwendet. Durch den strikten Ablauf ist die Planung besonders einfach und Probleme wie Dead- oder Livelocks sind nicht möglich. Nachteilig ist jedoch, dass auf spontane Ereignisse nicht besonders gut reagiert werden kann. So müssen Variablen oder beispielsweise hardwareseitige IOs mittels Polling zyklisch abgefragt werden, was zu einer erheblichen Erhöhung der Laufzeiten führen kann[5, S. 4-6]. Klassische Subroutinen verfügen, vereinfacht betrachtet, nur über einen einzigen Einstiegspunkt. Sie können mittels Argumenten oder Variablen mit Informationen versorgt werden und können Informationen am Funktionsende zurückgeben. Beim Aufruf einer Funktion in einem Programm oder durch eine andere Funktion wird zunächst der derzeitige Zustand bzw. der Kontext gesichert. Die aufgerufene Funktion erstellt darauf einen neuen Kontext, welcher nur während der Durchführung der Subroutine gilt. Nach Abarbeitung der Funktion sind die lokal existierenden Variablen der Funktion nicht mehr verfügbar und der Zustand vor Ausführung der Funktion wird über den gespeicherten Kontext wiederhergestellt.

In Python werden solche Funktionen mit dem reinen `def`-Befehl erzeugt. Dies führt zur Erstellung eines Funktionsobjektes mit entsprechendem Namen und Inhalt. Bei Aufruf der Funktion werden die Anweisungen der Funktion abgearbeitet und erst, wenn die letzte Befehlszeile ausgeführt wurde oder ein `return`-Befehl folgt, endet der Funktionsaufruf [6, S. 113-116].

Im folgenden Beispiel ist eine simple Funktion “`summierFunktion_1`” mit den beiden Übergabeparametern “`uebergabeParA`” und “`uebergabeParB`” aufgezeigt. Die Funktion dient lediglich zum besseren Vergleich mit den noch folgenden Funktionskonzepten.

```
1 def summierFunktion_1(uebergabeParA , uebergabeParB ):
2     summe = uebergabeParA + uebergabeParB
3     return summe
```

Auflistung 2.1: Beispiel für Funktionsdefinition als Subroutine

2.2 Koroutinen

Der Begriff der Koroutinen wurde ursprünglich von Melvin Conway im Jahre 1963 ins Leben gerufen. Diese werden von ihm als gleichberechtigte Subroutinen beschrieben. Als besonders nützlich werden sie für nicht synchron ablaufende Kommunikationen zwischen einzelnen Programmen oder Funktionen angesehen [2, S. 396]. Im Gegensatz zur sequentiellen Programmbefolge von Subroutinen werden Koroutinen genutzt, um kooperatives Multitasking zu ermöglichen. Kooperativ bedeutet, dass in diesem Fall die Routinen selbst die belegten Ressourcen (Prozessor etc.) wieder freigeben müssen, damit eine andere Routine diese wieder nutzen kann. Durch diesen Sachverhalt gilt das Kooperative Multitasking als einfach realisierbar [5, S. 9-14]. Darüber hinaus verfügen Koroutinen nicht nur über einen Einstiegspunkt sondern können ihre Ausführung anhalten und an einem späteren Zeitpunkt wieder fortsetzen, was eine dauerhafte Kontextsicherung nötig macht. Eine Koroutine muss also nicht zwingend durch eine main-Funktion aufgerufen werden. Koroutinen verhalten sich ähnlich wie Generatoren, jedoch mit dem Unterschied, dass diese nicht nur Generieren können, sondern auch Daten konsumieren können. Beliebt sind Koroutinen, weil einfach Erzeuger-Verbraucher-Konstrukte möglich sind. Als Schlüsselbegriff dient bei Koroutinen der Ausdruck “yield”, dieser wird als Ausdruck in der Funktion verwendet [1]. Trifft die Koroutine auf einen “yield”-Ausdruck, so pausiert die Routine. Erst wenn mittels eines “send”-Befehles Daten an die Koroutine übermittelt werden, wird mit der Ausführung fortgefahren [8]. Die genauere Verwendung wird in Kapitel 3.1 beschrieben.

2.3 Threads

Bei Threads handelt es sich um ein ähnliches Konzept wie bei den Koroutinen. Hier entscheidet jedoch nicht der Programmierer wann ein Thread-Wechsel erfolgt oder das Programm selbst, sondern dies wird über den Scheduler des Betriebssystems bestimmt [1]. Ein Thread ist dabei ein Bestandteil eines Prozesses, der scheinbar parallel zum Hauptprozess läuft. Neben den Kontext-Informationen des Haupt-Prozesses (“Process Control Block”) werden im jeweiligen “Thread Control Block” auch alle nötigen Informationen der dazugehörigen Threads abgelegt, damit bei einem Thread-Wechsel durch das Betriebssystem dieser später wieder fortgesetzt werden kann [11, S. 83]. Aktionen, bei denen auf spontane Ereignisse gewartet wird, wie zum Beispiel auf externe IO-Aktionen, bieten sich zur Realisierung mit Threads an und können den Gesamt Ablauf des Prozesses merklich beschleunigen. Realisiert wird dies mit dem “threading”-Modul. Besonders wichtig bei der Verwendung von Thread-Funktionalitäten ist es, ein gleichzeitiges Zugreifen auf gemeinsame Ressourcen, sogenannte kritische Bereiche, zu verhindern. Dies kann über manuelle Synchronisation, wie im Thread-Beispiel im nächsten Kapitel beschrieben, oder mit dem “Lock”-Objekt erfolgen [10]. Darüber hinaus müssen Livelock- oder Deadlock-Szenarien in Betracht gezogen werden. Ein Deadlock kann beispielsweise entstehen, wenn zwei Threads gegenseitig aufeinander warten bzw. mit Semaphoren benötigte Bereiche gesperrt sind. Dies hätte zur Folge, dass keiner der beiden Threads mit der Programmausführung fortfahren kann [3, Kap. 32.5.3]. In Kapitel 3.2 wird der Aufbau von Threads weiter erläutert.

2.4 Asyncio

Mit Asyncio wird eine Python-Bibliothek angeboten, welche kooperatives Multitasking im Einzelprozess- und Einzelthread-Stil ermöglicht. Wie aus dem Namen ablesbar, bietet sich der Einsatz gerade im Bezug auf die Verwendung von I/O-Anwendungen an und ist dem Konzept der Koroutinen sehr ähnlich. Auch hier wird kooperatives Multitasking verwendet, um von Wartezeiten geprägte Abläufe zu beschleunigen. Mit der Einführung des asyncio-Paketes in der Python Version 3.4 wurde Python um die Ausdrücke “async” und “await” erweitert. Das “async”-Schlüsselwort muss vor dem def-Ausdruck stehen und signalisiert, dass es sich um eine asynchrone Funktion handelt. Mittels “await”-Aufruf wird die derzeit ausgeführte asynchrone Routine pausiert und auf den Rückgabewert der übergebenen Funktion gewartet. In der Zwischenzeit können andere Routinen mit ihrer Abarbeitung fortfahren. Wenn es sich bei der Funktion, auf die gewartet wird, um eine “sleep”-Funktion handelt, ist der Geschwindigkeitsvorteil leicht ersichtlich [7]. Ein weiteres nützliches Feature des “asyncio”-Paketes ist die Warteschlangen-Funktionalität, welche in Form der sogenannten “queues” vorliegt. Hier lassen sich per “put()”-Befehl Elemente in die Schlange einreihen und können per “get()” entnommen werden. Dies ermöglicht somit eine simple Erzeuger-Verbraucher-Struktur [9]. Die genaue Verwendung des “asyncio”-Paketes wird in Kapitel 3.3 beschrieben.

Kapitel 3

Beispiele

Im Folgenden werden abseits der reinen Subroutinen-Funktion alle Konstrukte mittels kurzer Beispiele erläutert. Die verwendete Python-Version ist hierbei “3.9.7” und diese wurde auf einem Windows-PC (64bit-Betriebssystem) mit IDLE ausgeführt. Die im Erklärtext beigefügten Zeilenangaben beziehen sich immer auf das jeweilige Beispielprogramm.

3.1 Koroutinen

```
1 def summierFunktion_2():
2     print('Warte auf Werte..')
3     while True:
4         parA = (yield)
5         parB = (yield)
6         summe = int(parA) + int(parB)
7         print(int(summe))
8
9 #Aufruf der Koroutine
10 summierer = summierFunktion_2()
11
12 #Ausfuehren der Koroutine bis zum ersten "yield"-Ausdruck
13 summierer.__next__()
14
15 #Senden der Parameter
16 summierer.send(35)
17 summierer.send(12)
18
19 #Schliessen der Koroutine
20 summierer.close()
```

Auflistung 3.1: Beispiel für die Verwendung von Koroutinen

Wird das Programm ausgeführt, passiert beim Aufruf von “summierFunktion_2” noch nichts. Erst mit dem Befehl “summierer.__next__()” (Z. 13) gibt die Funktion den “print”-Befehl (Z. 2) und rückt bis zum ersten Auftreten des Schlüsselwortes “yield” vor. Mittels “summierer.send” (Z. 16-17) können Werte, in diesem Fall die Ganzzahlen 35 und 12, übermittelt werden. Diese Zahlen werden anschließend von den “(yield)”-Aufrufen (Z. 4-5) an die lokalen Variablen der Funktion übergeben. Wichtig ist hierbei ist, die Reihenfolge nach dem FIFO-Prinzip zu beachten. Im Beispiel der Addition spielt diese zwar keine Rolle, aber in anderen Applikationen kann dies entscheidend sein. Nach Erhalt beider Zahlen wird der Rest der Funktion ausgeführt und

das Ergebnis der Addition wird ausgegeben (Z. 7). Im vorher genannten Beispiel ergibt dies den Wert “47”. Würde die Koroutine nicht mittels “summierer.close” (Z. 20) geschlossen werden, könnte man weiterhin neue Zahlen an “summierer” senden.

Die exemplarische Darstellung der Koroutinen-Funktionalität macht sehr deutlich, wie nützlich und unkompliziert die Verwendung dieses Konstruktes ist. Abseits dieses Verwendungsbeispiels wären auch komplexere Erzeuger-Verbraucher-Strukturen möglich, bei denen mehrere Koroutinen, in einer Kette eingereiht oder parallel aufgebaut, Daten austauschen können.

3.2 Threads

```

1 import threading
2 import time
3
4 neueZahlenWerte = False
5 neueSumme = False
6 summiererAktiv = True
7 zahlA , zahlB , ergebnis = 0
8
9 def summmierFunktion_3():
10     print('Der Summierer wurde gestartet...')
11     global neueZahlenWerte, neueSumme, ergebnis
12
13     while summiererAktiv:
14         if neueZahlenWerte == True:
15             neueZahlenWerte = False
16             ergebnis = int(zahlA) + int(zahlB)
17             neueSumme = True
18
19 if __name__ == "__main__":
20     print('Starten des Threads im Hauptprozess...')
21     summierer = threading.Thread(target=summmierFunktion_3)
22     summierer.start()
23
24     zahlA = 3
25     zahlB = 5
26     neueZahlenWerte = True
27
28     while neueSumme == False:
29         time.sleep(1)
30         print("...")
31
32     print(f'Die errechnete Summe ist {ergebnis}!')
33     summiererAktiv = False
34
35     summierer.join()
36     print('Summierer-Thread wurde beendet...')

```

Auflistung 3.2: Beispiel für die Verwendung von Threads

Mittels den Importieraufufen (Z. 1-2) werden die Module “threading” und “time” eingebunden. “threading” wird dabei für das komfortable Erstellen von Threads in einem Prozess benötigt [10]. “time” wird benötigt, um den “time.sleep(1)“-Befehl verwenden zu können, welcher eine Wartezeit verursacht [12]. Da der Thread dauerhaft lauffähig ist und gleichzeitiges Zugreifen auf Variablen vermieden werden soll, muss zwischen Thread und Hauptprozess eine Synchronisation erfolgen. Dies wird über die globalen Variablen “neueZahlenWerte”, “neueSumme” und “summiererAktiv” erreicht (Z. 4-6). Alternativ hätte man dies auch mit einem Semaphor-Objekt lösen können. Mittels “summierer = threading.Thread(target = summmierFunktion_3)” (Z. 21) wird ein Thread-Objekt mit entsprechendem Verweis auf die “summierFunktion_3()” (Z. 9) erzeugt. Über den Befehl “summierer.start” (Z. 22) wird der Thread dann gestartet. Darauf hin ist der Thread lauffähig und die Funktion wird bis zur While-Schleife (Z. 13) abgearbeitet. Solange über die globale Variable “neueZahlenWerte” (Z. 14) keine Signalisierung neuer Werte übermittelt wird, findet kein Vorrücken statt. Bei Freigabe durch Setzen der “neueZahlenWerte”-Variable werden erst die Variablen “parA” und “parB” zu einer Summe addiert und anschließend mittels Setzen der Variable “neueSumme” das Ende der Rechenoperation signalisiert (Z. 16-17). Der Hauptprozess hat mittels aktiven Wartens auf die Freigabe mit der Fortführung pausiert (Z. 28-30). Ist die Freigabe erteilt wird mit der Ausführung fortgefahren und das Ergebnis der Rechnung ausgegeben (Z. 32). Um die Endlosschleife des “summierer”-Threads zu beenden, wird die Variable “summiererAktiv = False” gesetzt und mittels “summierer.join()” wird auf das Beenden des Threads gewartet (Z. 35).

Die vielfältigen Möglichkeiten, Threads einzusetzen, übersteigen bei weitem die Komplexität des gewählten Beispiels. Gerade im Bereich der Semaphoren kann mittels den Funktionen “acquire()” und “release()” sehr einfach eine Synchronisation zwischen Threads erreicht werden [10]. Dies ermöglicht eine komfortable Kommunikation zwischen Threads per globaler Variablen, welche deutlich unkomplizierter ist als bei der Kommunikation von eigenständigen Prozessen.

3.3 Asyncio

```

1 import asyncio
2 import random
3 import time
4
5 async def summierFunktion_4(parA, parB):
6
7     #asynchrones Warten mit variierender Wartezeit zwischen 1 und 5s
8     wartezeit = random.randint(1,10)
9     await asyncio.sleep(wartezeit)
10
11     summe = int(parA) + int(parB)
12     return summe, wartezeit
13
14 async def async_main():
15     #Aufrufen der Summier-Funktion mit versch. Werten
16     ergebnis_liste = await asyncio.gather(\
17         summierFunktion_4(5, 39),\
18         summierFunktion_4(5, 46),\
19         summierFunktion_4(78, 69))
20     return ergebnis_liste
21
22 if __name__ == "__main__":
23     #Starten der Zeitmessung und Aufruf der asynchronen Hauptroutine
24     start_Zeit = time.perf_counter()
25     ergebnisse = asyncio.run(async_main())
26     #Errechnen der verstrichenen Zeit
27     end_Zeit = time.perf_counter()
28     vergangene_Zeit = end_Zeit - start_Zeit
29
30     i = 0
31     for elem in ergebnisse:
32         i = i + 1
33         print(f'Das Ergebnis der {i}ten Rechnung ist {elem[0]} \
34             und die Wartezeit war {elem[1]}.'')
35
36     print(f'Die Rechnung hat {vergangene_Zeit:0.2f} Sekunden gedauert.'')

```

Auflistung 3.3: Beispiel für die Verwendung von Asyncio

```

1 Das Ergebnis der 1ten Rechnung ist 44 und die Wartezeit war 8.
2 Das Ergebnis der 2ten Rechnung ist 51 und die Wartezeit war 3.
3 Das Ergebnis der 3ten Rechnung ist 147 und die Wartezeit war 4.
4 Die Rechnung hat 8.01 Sekunden gedauert.

```

Auflistung 3.4: Ausgabe in IDLE bei Asyncio-Beispiel

In Auflistung 3.3 wird der Beispiel-Code für das “asyncio”-Paket gezeigt. Mit “summierFunktion_4” (Z. 5) und “async_main” (Z. 14) werden zwei Funktionen mit dem “async”-Schlüsselwort deklariert. In “summierFunktion_4” wird, ähnlich wie in den vorherigen Beispielen, wieder ein Summieren zweier Werte ausgeführt (Z. 11), jedoch mit dem Unterschied, dass zusätzlich eine Wartezeit mittels des “asyncio.sleep”-Befehls eingebunden wird (Z. 9). Dieser Befehl ist mit dem “await”-Schlüsselwort gekennzeichnet. und bewirkt, dass nach Aufruf die derzeitige Summier-routine pausiert wird und eine andere angefangen bzw. fortgesetzt werden kann. Nach Ablauf der Wartezeit kann zur ursprüngliche Routine zurückgekehrt werden. “summierFunktion_4” liefert neben dem Ergebnis der Rechnung auch die Wartezeit zurück, um anschließend die Effizienz des Programmes besser nachvollziehen zu können. In “async_main” werden mehrere Aufrufe der “summierFunktion_4” mit verschiedenen Werten mittels “asyncio.gather” getätigt (Z. 16-19). Auch hier wird zum Ende der Funktion das Ergebnis in Form einer Liste zurückgegeben (Z. 20). Um später die Dauer der Ausführung von “async_main” (Z. 25) besser interpretieren zu können, wird die Start- (Z. 24) und Endzeit (Z. 27) gespeichert [12]. Es wird sowohl das Ergebnis der Rechnung als auch die eingebaute Wartezeit in IDLE ausgegeben (Z. 30-34). Das Skript endet mit der Ausgabe der verstrichenen Zeit (Z. 36).

Die Ausgabe von IDLE ist in Auflistung 3.4 dargestellt. Wenig verwunderlich ist, dass die errechnete Summe in allen drei Fällen richtig ist. Weitaus interessanter sind jedoch die Wartezeiten. Hätte man die Summierfunktionen sequentiell ausgeführt, läge die verstrichene Zeit weit höher. Im Falle der gegebenen Wartezeiten wäre diese bei etwa 15 Sekunden. Diese Zahl ergibt sich aus der Addition der drei Zeiten. In der asynchronen Version haben wir hingegen eine gesamte Dauer von 8,01 Sekunden. Dies verdeutlicht auf einfache Weise, wie günstig sich die Verwendung von “asyncio” auf die Ausführdauer auswirken kann. Die etwa acht Sekunden entstehen, weil die Wartezeit der ersten Rechnung dieser entspricht und beim “asyncio.gather” (Z. 16) auf das Ende aller übergebener Funktionen gewartet wird. Alle anderen Rechenoperation werden während dieser acht Sekunden abgeschlossen und die erste erst nach Ablauf der Wartezeit [7].

Kapitel 4

Fazit

Mit den in der Studienarbeit vorgestellten Konzepten, Koroutine, Thread und Asyncio werden sehr nützliche Verfahren angeboten. Sie ermöglichen dem Python-Programmierer parallelartig arbeitende Programme zu entwickeln, um sich von der starren Struktur des sequentiellen Ablaufes zu lösen. Mit den aufgeführten Beispielen konnte der Ablauf und auch die zu implementierende Struktur der Konzepte aufgezeigt werden. Neben den Threads, welche im Allgemeinen recht bekannt und verbreitet sind, erweisen sich gerade die Koroutine bzw. Asyncio als sehr nützlich. Besonders die Verwendung des kooperativen Multitaskings macht diese Verfahren robust gegenüber Zugriffsfehlern, welche bei Threads ohne Synchronisation sehr leicht auftreten können. Der zeitliche Vorteil, der beim Einsatz dieser Pakete entstehen kann, ist bei Asyncio deutlich sichtbar und auch leicht nachvollziehbar. Vorsicht und bedachtes Vorgehen sind jedoch geboten, da leicht unübersichtliche Situationen entstehen können.

Auflistungsverzeichnis

2.1 Beispiel für Funktionsdefinition als Subroutine	2
3.1 Beispiel für die Verwendung von Koroutinen	5
3.2 Beispiel für die Verwendung von Threads	6
3.3 Beispiel für die Verwendung von Asyncio	8
3.4 Ausgabe in IDLE bei Asyncio-Beispiel	8

Literatur

- [1] Atul Kumar. *Coroutine in Python*. 2021. URL: <https://www.geeksforgeeks.org/coroutine-in-python/> (besucht am 23.09.2021).
- [2] Melvin E. Conway. „Design of a separable transition-diagram compiler“. In: *Communications of the ACM* 6.7 (1963), S. 396–408. ISSN: 0001-0782. DOI: [10.1145/366663.366704](https://doi.org/10.1145/366663.366704).
- [3] Johannes Ernesti und Peter Kaiser. *Python 3. Das umfassende Handbuch*. ger. 5., aktualisierte Auflage 2017, 1., korrigierter Nachdruck. Rheinwerk Computing. Ernesti, Johannes (VerfasserIn) Kaiser, Peter (VerfasserIn). Bonn: Rheinwerk Verlag, 2018. 1040 S. ISBN: 9783836258647. URL: <https://openbook.rheinwerk-verlag.de/python/> (besucht am 24.09.2021).
- [4] Mike Müller. *Asynchron Programmieren mit Python*. Hrsg. von Linux-Magazin. 3/2015. URL: <https://www.linux-magazin.de/ausgaben/2015/03/python-asynchron/> (besucht am 24.09.2021).
- [5] Prof. Dr.-Ing. Jürgen Krumm. „Echtzeitsysteme - Teil 2: Scheduling-Mechanismen“. Nürnberg, 2021. (Besucht am 24.09.2021).
- [6] Prof. Dr.-Ing. Jürgen Krumm. „Skriptsprache Python - Teil 1: Arbeiten mit Python 3“. Nürnberg, 2021. (Besucht am 24.09.2021).
- [7] Python Software Foundation, Hrsg. *asyncio — Asynchronous I/O*. Python Software Foundation. URL: <https://docs.python.org/3/library/asyncio.html> (besucht am 21.09.2021).
- [8] Python Software Foundation, Hrsg. *PEP 342 – Coroutines via Enhanced Generators*. Python Software Foundation. URL: <https://www.python.org/dev/peps/pep-0342/> (besucht am 24.09.2021).
- [9] Python Software Foundation, Hrsg. *Queues*. Python Software Foundation. URL: <https://docs.python.org/3/library/asyncio-queue.html> (besucht am 26.09.2021).
- [10] Python Software Foundation, Hrsg. *threading — Thread-based parallelism*. Python Software Foundation. URL: <https://docs.python.org/3/library/threading.html> (besucht am 24.09.2021).
- [11] Abraham Silberschatz, Peter B. Galvin und Greg Gagne. *Operating system concepts*. eng. 6. ed. New York, NY: Wiley, 2003. 951 S. ISBN: 0471262722. (Besucht am 24.09.2021).
- [12] *time — Time access and conversions*. Python Software Foundation. URL: <https://docs.python.org/3/library/time.html> (besucht am 26.09.2021).