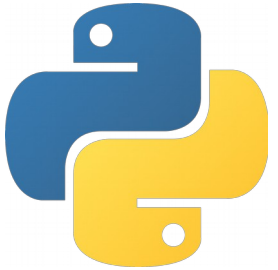
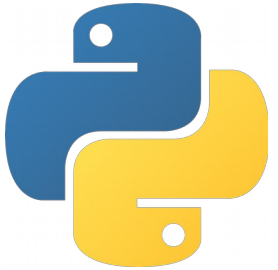
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Wir sind als Benutzer moderner Computer gewohnt, dass ein Rechner mehrere Programme gleichzeitig ausführen kann
  - Beispielsweise schreiben wir eine E-Mail, während im Hintergrund das letzte Urlaubsvideo in ein anderes Format umgewandelt wird und eine MP3-Software unseren Lieblingssong aus den Computerlautsprechern ertönen lässt
  - Eine typische Arbeitssitzung, wobei jeder Kasten für ein laufendes Programm steht
  - Die Länge der Kästen entlang der Zeitachse zeigt an, wie lange der jeweilige Prozess läuft

# Python
# Parallele Programmierung

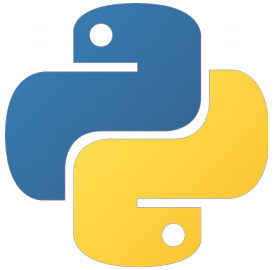


- Prozesse, Multitasking und Threads


  - Faktisch kann ein Prozessor aber nur genau eine Aufgabe zu einem bestimmten Zeitpunkt übernehmen und nicht mehrere gleichzeitig

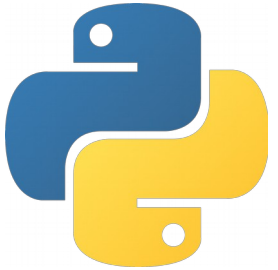
  - Selbst bei modernen Prozessoren mit mehr als einem Kern oder bei Rechnern mit vielen Prozessoren ist die Anzahl der gleichzeitig ausführbaren Programme durch die Anzahl der Kerne bzw. Prozessoren beschränkt

# Python
# Parallele Programmierung

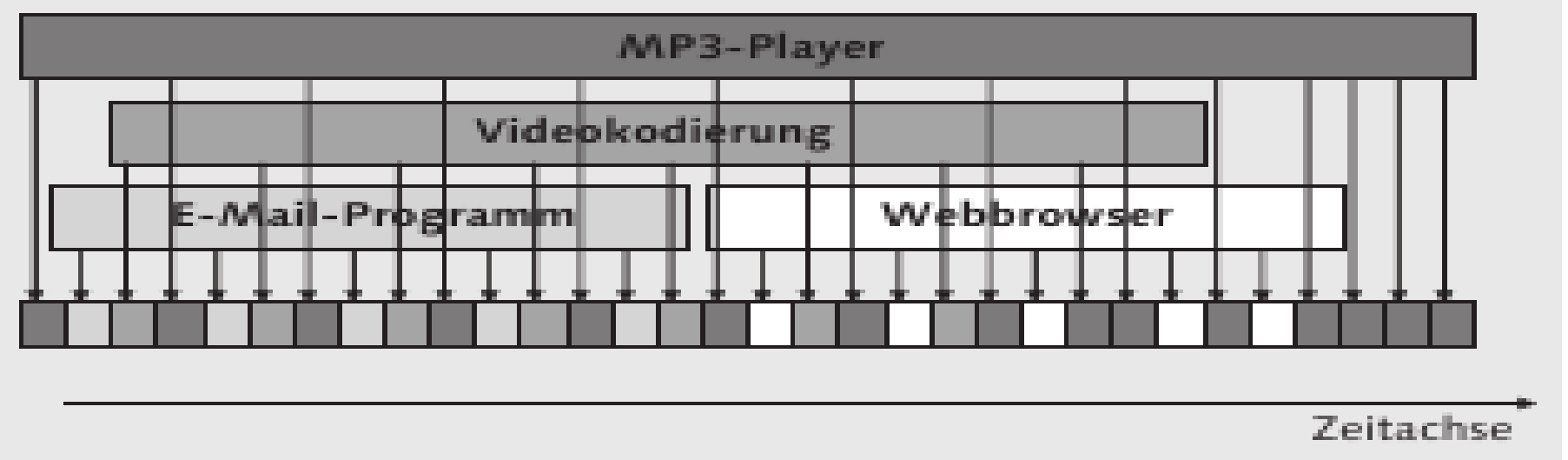


- Prozesse, Multitasking und Threads
  - Wie ist es also möglich, dass das einleitend beschriebene Szenario auch auf einem Computer mit nur einem Prozessor, der nur einen einzigen Kern besitzt, funktioniert?
  - Der dahinterstehende Trick ist im Grunde sehr einfach, denn man versteckt die Limitierung der Maschine geschickt vor dem Benutzer, indem man ihm vorgaukelt, es würden mehrere Programme simultan laufen
  - Dies wird dadurch erreicht, dass man jedem Programm ganz kurz die Kontrolle über den Prozessor zuteilt, es also laufen lässt
  - Nach Ablauf der sogenannten Zeitscheibe wird dem Programm die Kontrolle wieder entzogen, wobei sein aktueller Zustand gespeichert wird

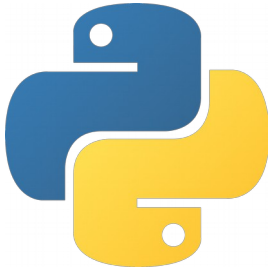
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Sie können sich die Arbeit eines Computers so vorstellen, dass in rasender Geschwindigkeit alle laufenden Programme geweckt, für eine kurze Zeit ausgeführt und dann wieder schlafen gelegt werden
  - Durch die hohe Geschwindigkeit des Umschaltens zwischen den Prozessen nimmt der Benutzer dies nicht wahr



# Python
# Parallele Programmierung

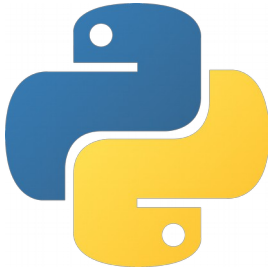


- Prozesse, Multitasking und Threads
  - Die Leichtgewichte unter den Prozessen – Threads
    - Innerhalb eines Prozesses selbst kann aber weiterhin nur eine Aufgabe zur selben Zeit ausgeführt werden, da das Programm linear abgearbeitet wird

    - In vielen Situationen ist es aber erforderlich, dass ein Programm mehrere Operationen zeitgleich durchführt

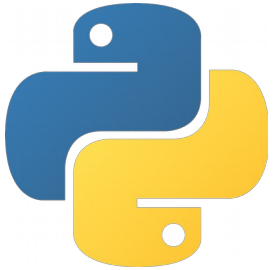
    - Beispielsweise darf die Benutzeroberfläche während einer aufwendigen Berechnung nicht blockieren, sondern soll den aktuellen Status anzeigen, und der Benutzer muss die Möglichkeit haben, die Berechnung gegebenenfalls abbrechen zu Prozesse, Multitasking und Threads können

# Python
# Parallele Programmierung



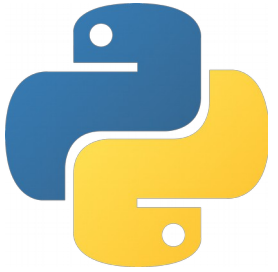
- Prozesse, Multitasking und Threads
  - Die Leichtgewichte unter den Prozessen – Threads
    - Es ist zwar möglich, die Beschränkung auf nur eine Operation zur selben Zeit dadurch zu umgehen, dass weitere Prozesse erzeugt werden
    - Allerdings müssen dann Daten zwischen verschiedenen Prozessen ausgetauscht werden, wofür relativ viel Aufwand nötig ist, weil jeder Prozess seine eigenen Variablen hat, die von den anderen Prozessen abgeschirmt sind
    - Eine befriedigende Lösung für das Problem liefern sogenannte Threads
    - Ein Thread ist ein Ausführungsstrang in einem Prozess

# Python
# Parallele Programmierung

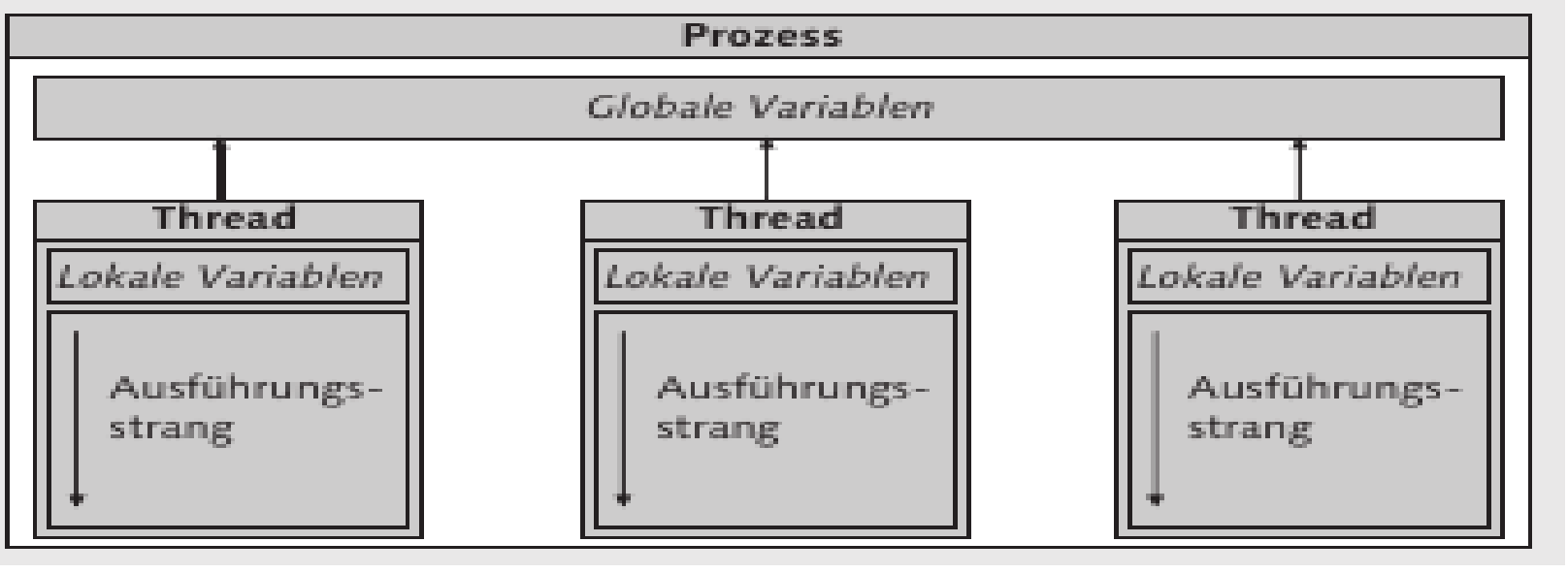


- Prozesse, Multitasking und Threads
  - Die Leichtgewichte unter den Prozessen – Threads
    - Nun kann ein Prozess aber auch mehrere Threads starten, die dann durch das Betriebssystem wie Prozesse scheinbar gleichzeitig ausgeführt werden
    - Der Vorteil von Threads gegenüber Prozessen besteht darin, dass sich die Threads eines Prozesses denselben Speicherbereich für globale Variablen teilen
    - Wenn also in einem Thread eine globale Variable verändert wird, ist der neue Wert auch sofort für alle anderen Threads des Prozesses sichtbar

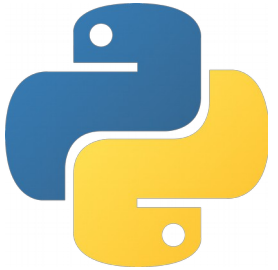
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Die Leichtgewichte unter den Prozessen – Threads
    - Außerdem ist die Verwaltung von Threads für das Betriebssystem weniger aufwendig als die Verwaltung von Prozessen
    - Deshalb werden Threads auch Leichtgewichtprozesse genannt

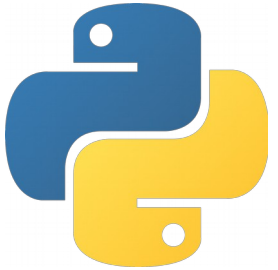


# Python
# Parallele Programmierung



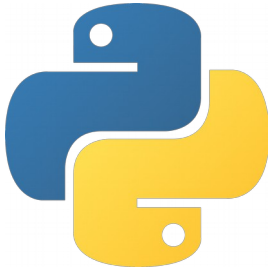
- Prozesse, Multitasking und Threads
  - Die Leichtgewichte unter den Prozessen – Threads
    - In Python gibt es leider keine Möglichkeit, verschiedene Threads auf verschiedenen Prozessoren oder Prozessorkernen auszuführen
    - Dies hat zur Folge, dass selbst Python-Programme, die intensiv auf Threading setzen, nur einen einzigen Prozessor oder Prozessorkern nutzen können
    - Wenn Sie sehr rechenintensive Programme schreiben, die die gesamte Rechenpower des Computers ausschöpfen sollen, werfen Sie einen Blick auf das multiprocessing-Modul, mit dessen Hilfe mehrere Prozesse verwaltet werden können, die auch echt parallel auf verschiedenen Prozessoren laufen

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Die Thread-Unterstützung in Python
    - Python bietet zwei Module für den Umgang mit Threads an: _thread und threading
    - Das erste Modul namens _thread ist die einfachere Variante und sieht jeden Thread als Funktion
    - Mit threading wird ein objektorientierter Ansatz implementiert, bei dem jeder Thread ein eigenes Objekt darstellt
    - Wir werden uns mit beiden Ansätzen beschäftigen, wobei wir mit dem einfacheren Modul _thread beginnen werden

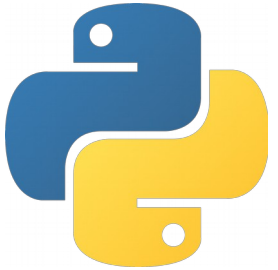
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
    - Das Modul _thread kann einzelne Funktionen in einem separaten Thread ausführen
    - Als Beispiel erstellen wir eine Funktion, die das laufende Programm blockiert
    - Als Beispiel nehmen wir die Approximation von PI (Annäherung an PI mithilfe des Wallis'schen Produkts)

$$\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdots = \frac{\pi}{2}$$

# Python
# Parallele Programmierung

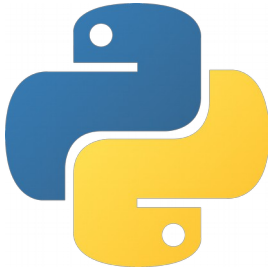


- Prozesse, Multitasking und Threads
  - Das Modul _thread

    - Die Funktion ← Wert n = 10000000 Zeit ca. 7 sec.

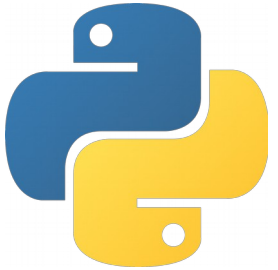
```
def naehere_pi_an(n):
    pi_halbe = 1
    zaehler, nenner = 2.0, 1.0
    for i in range(n):
        pi_halbe *= zaehler / nenner
        if i % 2:
            zaehler += 2
        else:
            nenner += 2
    print("Annaeherung mit {} Faktoren:
    {:.16f}".format(n, 2*pi_halbe))
```

# Python
# Parallele Programmierung



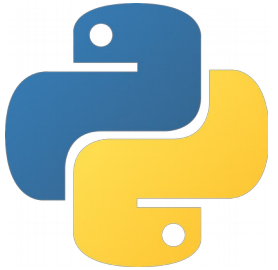
- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - thread.start_new_thread(function, args[, kwargs])
    - Der Parameter function muss dabei eine Referenz auf die Funktion enthalten, die ausgeführt werden soll
    - Mit args muss eine tuple-Instanz übergeben werden, die die Parameter für function enthält
    - Mit dem optionalen Parameter kwargs kann ein Dictionary übergeben werden, das zusätzliche Schlüsselwortparameter für die Funktion function bereitstellt

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - thread.start_new_thread(function, args[, kwargs])
    - Als Rückgabewert gibt _thread.start_new_thread eine Zahl zurück, die den erzeugten Thread eindeutig identifiziert
    - Nachdem function verlassen wurde, wird der Thread automatisch gelöscht
    - Auf der folgenden Seite werden wir mithilfe von _thread.start_new_thread mehrere Threads erzeugen, die die Funktion naehere_pi_an für verschiedene n aufrufen

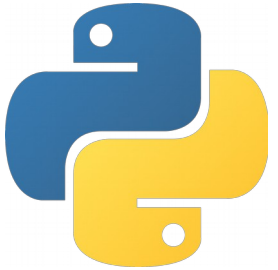
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - thread.start_new_thread(function, args[, kwargs])
    - Beispiel

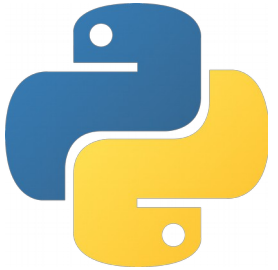
```
import _thread
_thread.start_new_thread(naehere_pi_an,  (11111111,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (100000,))
_thread.start_new_thread(naehere_pi_an, (1234569,))
_thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while True:
   pass
```

# Python
# Parallele Programmierung



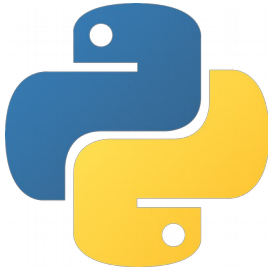
- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - thread.start_new_thread(function, args[, kwargs])
    - Die Endlosschleife am Ende des Programms ist notwendig, damit der Thread des Hauptprogramms auf die anderen Threads wartet und nicht sofort beendet wird
    - Alle Threads eines Programms werden nämlich sofort abgebrochen, wenn das Hauptprogramm sein Ende erreicht hat

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - thread.start_new_thread(function, args[, kwargs])
    - Das Interessante an diesem Programm ist die Reihenfolge der Ausgabe, die nicht mit der Reihenfolge der Aufrufe übereinstimmt
    - Ausgabe
      Annaeherung mit 1337 Faktoren: 3.142766811489281
      Annaeherung mit 10000 Faktoren: 3.141435935898644
      Annaeherung mit 100000 Faktoren: 3.141576945822637 7
      Annaeherung mit 1234569 Faktoren: 3.141593925932 1926
      Annaeherung mit 11111111 Faktoren: 3.141592794960 1699

# Python
# Parallele Programmierung



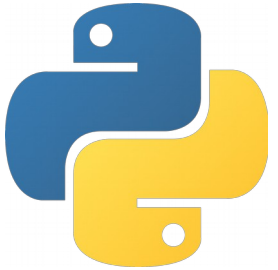
- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

  - Threads haben gegenüber Prozessen den Vorteil, dass sie sich dieselben globalen Variablen teilen und deshalb sehr einfach Daten austauschen können

  - Trotzdem gibt es ein paar Stolperfallen, die Sie beim Zugriff auf dieselbe Variable durch mehrere Threads beachten müssen

  - Wir wollen unser Programm um einen Zähler für die zur Zeit aktiven Threads erweitern, folgende Seite zeigt die implementierung

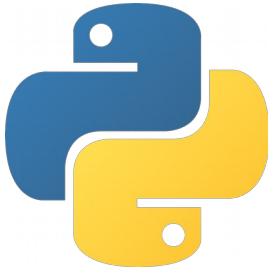
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

```
import _thread
anzahl_threads = 0
def naehere_pi_an(n):
    global anzahl_threads
    anzahl_threads += 1
    # Berechnungscode zur Übersicht ausgelassen
    anzahl_threads -= 1
_thread.start_new_thread(naehere_pi_an, (10000000,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (99999999,))
_thread.start_new_thread(naehere_pi_an,  (123456789,))
_thread.start_new_thread(naehere_pi_an,  (), {"n" : 1337})
while anzahl_threads > 0:
    pass
```

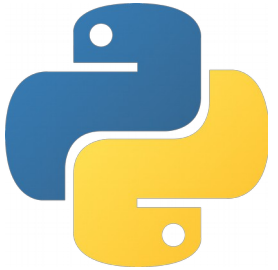
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

  - Problematik?
    - Dieses Programm hat zwei schwerwiegende Fehler
      - Erstens funktioniert es nicht immer, weil möglicherweise die while-Schleife erreicht ist, bevor überhaupt ein Thread gestartet werden konnte
        - In diesem Fall hat anzahl_threads den Wert 0, und damit wird die Schleife gar nicht durchlaufen, sondern das Programm beendet
      - Aber selbst, wenn dieses Problem bereits gelöst wäre, verhält sich das Programm unter Umständen fehlerhaft
        - Die Gefahr lauert in den beiden Zeilen, die den Wert der globalen Variable anzahl_threads verändern

# Python
# Parallele Programmierung



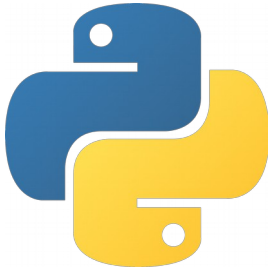
- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

  - Um solche Probleme zu vermeiden, kann ein Programm Stellen markieren, die nicht parallel in mehreren Threads laufen dürfen

  - Man bezeichnet solche Stellen auch als Critical Sections

  - Critical Sections werden durch sogenannte Lock-Objekte realisiert
    - lock_objekt = _thread.allocate_lock()

# Python
# Parallele Programmierung



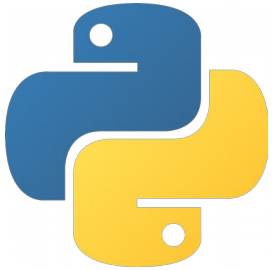
- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

  - Lock-Objekte haben die beiden wichtigen Methoden acquire und release, die jeweils beim Betreten bzw. beim Verlassen einer Critical Section aufgerufen werden müssen

  - Wenn die acquire-Methode eines Lock-Objekts aufgerufen wurde, ist es gesperrt

  - Ruft ein Thread die acquire-Methode eines gesperrten Lock-Objekts auf, muss er so lange warten, bis das Lock-Objekt wieder mit release freigegeben worden ist

# Python
# Parallele Programmierung

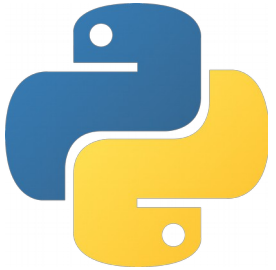


- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

  - Wir können unser Beispielprogramm folgendermaßen um Critical Sections erweitern, wobei wir außerdem einen Schalter namens thread_gestartet einfügen, damit das Hauptprogramm mindestens so lange wartet, bis die Threads gestartet worden sind

  - Das angepasste Beispiel finden wir auf der folgenden Seite

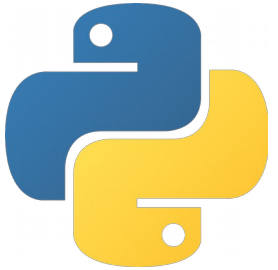
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

```
import _thread

anzahl_threads = 0
thread_gestartet = False

lock = _thread.allocate_lock()

def naehere_pi_an(n):
    global anzahl_threads, thread_gestartet

    lock.acquire()
    anzahl_threads += 1
    thread_gestartet = True
    lock.release()
```

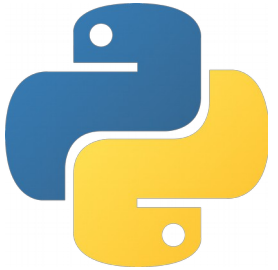
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

```
#Berechnungscode zur Übersicht ausgelassen
    lock.acquire()
    anzahl_threads -= 1
    lock.release()

_thread.start_new_thread(naehere_pi_an, (100000,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (11111111,))
_thread.start_new_thread(naehere_pi_an, (1234569,))
_thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while not thread_gestartet:
    pass
while anzahl_threads > 0:
    pass
```

# Python
# Parallele Programmierung

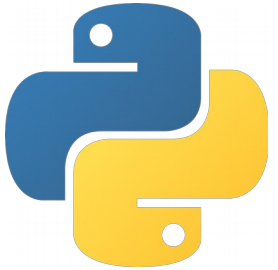


- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

  - Hinweis
    - Wenn Sie mehrere Lock-Objekte verwenden, kann es passieren, dass sich ein Programm in einem sogenannten Deadlock aufhängt, weil zwei gelockte Threads gegenseitig aufeinander warten

  - Auf der folgenden Seite wird ein Ablaufprotokoll gezeigt, wie ein Deadlock entstehen kann
    - Dabei gebe es zwei Threads A und B, und M und L seien Lock-Objekte

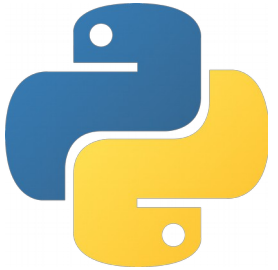
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul _thread
  - Datenaustausch zwischen Threads – locking

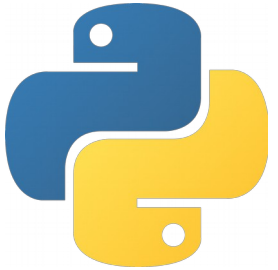
| Zeitfenster | Thread A | Thread B |
|---|---|---|
| 1 | das Lock-Objekt L mit `L.acquire()` sperren | *schläft* |
| **Zeitfenster von A endet, und Thread B wird aktiviert.** | | |
| 2 | *schläft* | Mit `M.acquire()` wird das Lock-Objekt *M* gesperrt. |
| **Zeitfenster von B endet, und Thread *A* wird aktiviert.** | | |
| 3 | `M.acquire` wird gerufen. Da M bereits gesperrt ist, wird A schlafen gelegt. | *schläft* |
| ***A* wurde durch M.acquire schlafen gelegt. *B* wird weiter ausgeführt.** | | |
| 4 | *schläft* | Ruft `L.acquire`, woraufhin B schlafen gelegt wird, da L bereits gesperrt ist. |
| **A wurde durch M.aquire und B durch L.aquire gesperrt.** | | |
| 5 | *schläft* | *schläft* |

# Python
# Parallele Programmierung



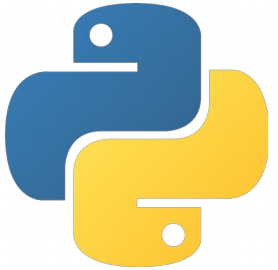
- Prozesse, Multitasking und Threads
  - Das Modul threading
  - Mit dem Modul threading wird eine objektorientierte Schnittstelle für Threads angeboten
  - Jeder Thread ist dabei eine Instanz einer Klasse, die von threading.Thread erbt
  - Da die Klasse selbst ein Teil des globalen Namensraums ist, eignen sich ihre statischen Member gut, um Daten zwischen den Threads auszutauschen
  - Natürlich muss auch hier der Zugriff auf die von mehreren Threads genutzten Variablen durch Critical Sections gesichert werden

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul threading
  - Wir wollen ein Programm schreiben, das in mehreren Threads parallel prüft, ob vom Benutzer eingegebene Zahlen Primzahlen sind
  - eine Klasse PrimzahlThread , die von threading.Thread erbt und als Parameter für den Konstruktor die zu überprüfende Zahl
  - Die Klasse threading.Thread besitzt eine Methode namens start, die den Thread ausführt
  - Was genau ausgeführt werden soll, bestimmt die run-Methode, die überschrieben wird

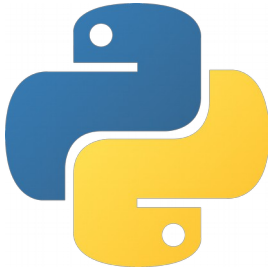
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul threading

```
import threading

class PrimzahlThread(threading.Thread):

  def __init__(self, zahl):
    threading.Thread.__init__(self)
    self.Zahl = zahl
```

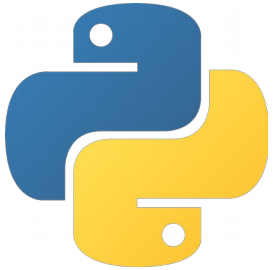
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul threading

```
...
    def run(self):
        i = 2
        while i*i <= self.Zahl:
            if self.Zahl % i == 0:
                print("{0} ist nicht prim, "
                      "da {1} = {2} * {3}".format( self.Zahl,
                                                   self.Zahl, i, self.Zahl / i))
                return
            i += 1
        print("{0} ist prim".format(self.Zahl))
```

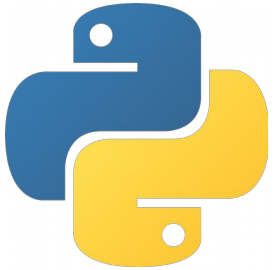
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul threading

```
...
meine_threads = [ ]
eingabe = input("> ")

while eingabe != "ende":
    thread = PrimzahlThread(int(eingabe))
    meine_threads.append(thread)
    thread.start()
    eingabe = input("> ")

for t in meine_threads:
    t.join()
```

# Python
# Parallele Programmierung



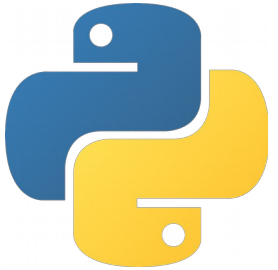
- Prozesse, Multitasking und Threads
  - Das Modul threading

  - Innerhalb der Schleife wird die Eingabe vom Benutzer eingelesen, und es wird geprüft, ob es sich um das Schlüsselwort "ende" zum Beenden des Programms handelt

  - Wurde etwas anderes als "ende" eingegeben, wird eine neue Instanz der Klasse PrimzahlThread mit der Benutzereingabe als Parameter erzeugt und mit der start - Methode gestartet

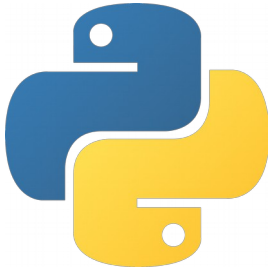
  - Das Programm verwaltet außerdem eine Liste namens meine_threads, in der alleThreads gespeichert werden

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul threading
  - Nach dem Verlassen der Eingabeschleife wird über meine_threads iteriert und für jeden Thread die join -Methode aufgerufen
  - Die Methode join sorgt dafür, dass das Hauptprogramm so lange wartet, bis alle gestarteten Threads beendet worden sind
  - join unterbricht die Programmausführung so lange, bis der Thread, für den es aufgerufen wurde, terminiert wurde

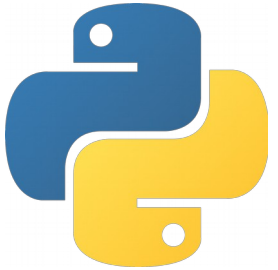
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul threading

  - Ein Programmlauf könnte dann so aussehen, die teils verzögerten Ausgaben zeigen, dass tatsächlich parallel gerechnet wurde:
    > 737373737373737
    > 5672435793
    5672435793 ist nicht prim, da 5672435793 = 3 * 1890811931
    > 909091
    909091 ist prim
    > 10000000000037
    > 5643257
    5643257 ist nicht prim, da 5643257 = 23 * 245359
    > 4567
    4567 ist prim
    10000000000037 ist prim
    737373737373737 ist prim
    > ende

# Python
# Parallele Programmierung



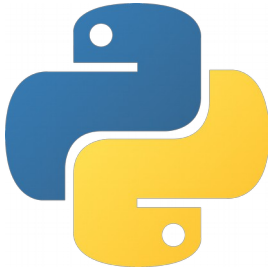
- Prozesse, Multitasking und Threads
  - Das Modul threading
  - Locking im threading-Modul

  - Genau wie das Modul _thread bietet auch threading Methoden an, um den Zugriff auf Variablen abzusichern, die in mehreren Threads verwendet werden

  - Die dazu benutzten Lock-Objekte lassen sich dabei genauso wie die von thread.allocate_lock zurückgegebenen Objekte verwenden

    - ErgebnisLock = threading.Lock()
    - ErgebnisLock.acquire()
    - ErgebnisLock.release()

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Das Modul threading

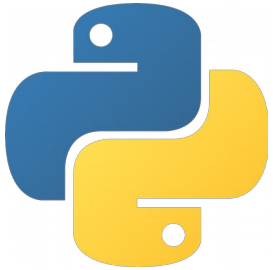
  - Aufgabe
    - Die Primzahlenberechnung mit dem Modul threading umsetzen

    - Die Ergebnisse in einem Dictionary speichern

      Aufbau

      ```
      {
          737373737373737 : "in Arbeit",
          5672435793 : "3 * 1890811931",
          909091 : "prim",
          10000000000037 : "in Arbeit",
          5643257 : "23 * 245359"
      }
      ```

# Python
# Parallele Programmierung

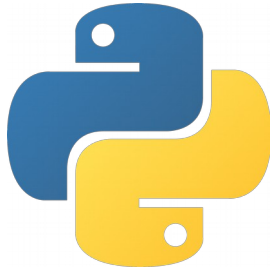


- Prozesse, Multitasking und Threads
  - Das Modul threading

  - Aufgabe
    - Um eine Eingabe 'status' erweitern, welche den aktuellen Stand aller Berechnungen ausgibt

      Aufbau
      -------- Aktueller Status --------
      5643257 = 5643257 * 245359
      909091 = prim
      737373737373737 = in Arbeit
      10000000000037 = in Arbeit
      56547 = 56547 * 18849
      --------------------------------

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Worker-Threads und Queues

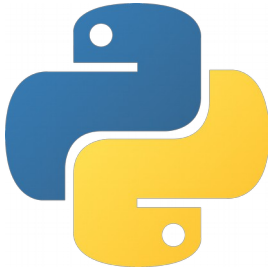
    - In unseren bisherigen Programmen haben wir immer für jede Aufgabe einen neuen Thread gestartet, sodass es theoretisch beliebig viele Threads geben konnte

    - Wie schon angemerkt wurde, kann dies zu Geschwindigkeitsproblemen führen, wenn sehr viele Threads gleichzeitig laufen

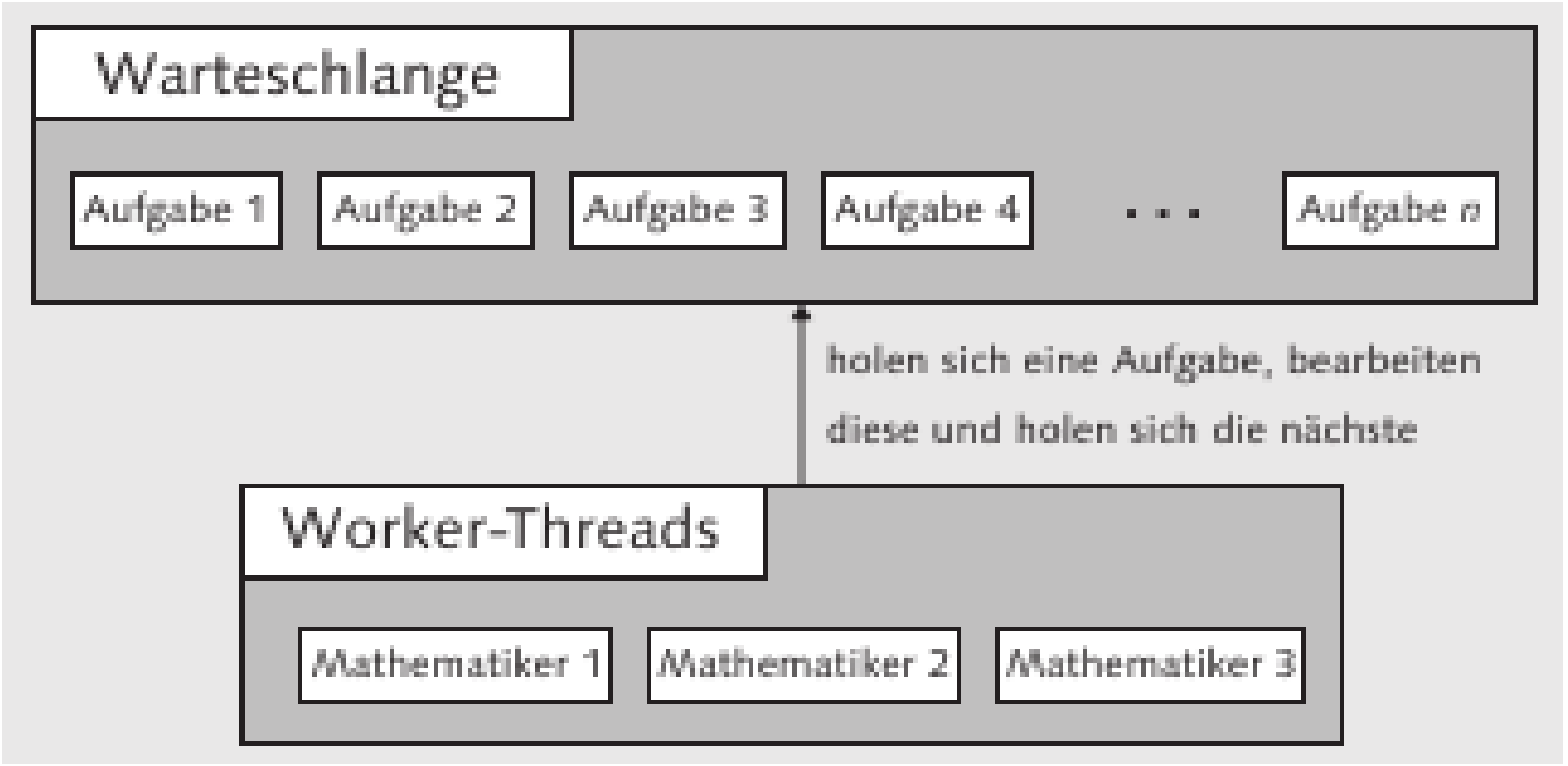
    - Die Worker holen sich dann selbstständig neue Aufgaben aus einer Warteschlange, sobald sie ihre vorherige Tätigkeit vollendet haben

    - Ist Warteschlange einmal leer, warten die Arbeiter so lange, bis neue Aufgaben zur Verfügung gestellt werden

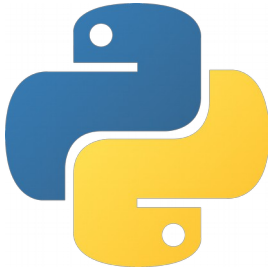
# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Worker-Threads und Queues

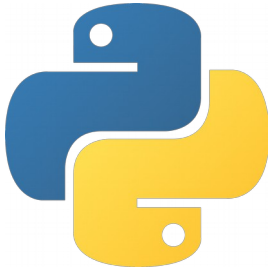


# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Worker-Threads und Queues
    - Python hat ein eigenes Modul namens queue, um mit Warteschlangen zu arbeiten
    - Der Konstruktor von queue erwartet eine ganze Zahl als Parameter, die angibt, wie viele Elemente maximal in der Warteschlange stehen können
    - Ist der Parameter kleiner oder gleich 0, ist die Länge der Queue nicht begrenzt

# Python
# Parallele Programmierung



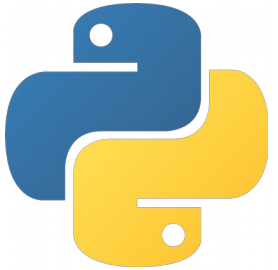
- Prozesse, Multitasking und Threads
  - Worker-Threads und Queues

    - Qeue-Instanzen haben drei wichtige Methoden
      put
      get
      task_done

      - Mit der put-Methode werden neue Aufträge in die Warteschlage gestellt

      - Die Methode get liefert die nächste Aufgabe der Queue
        - Befindet sich gerade kein Arbeitsauftrag in der Warteschlange, blockiert get den Thread so lange, bis der nächste Auftrag verfügbar ist

# Python
# Parallele Programmierung



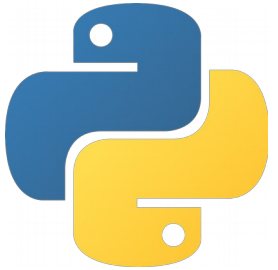
- Prozesse, Multitasking und Threads
  - Worker-Threads und Queues

    - Qeue

      - Hat ein Thread die Prüfung einer Zahl abgeschlossen, muss er dies der Queue mitteilen, indem er task_done aufruft

      - Die Warteschlange kümmert sich dabei selbstständig darum, dass das fertig verarbeitete Element entfernt wird

# Python
# Parallele Programmierung

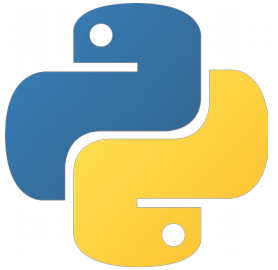


- Prozesse, Multitasking und Threads
  - Worker-Threads und Queues

    - Aufgabe
      - Vorhergehende Aufgabe um die Nutzung von Worker-Threads und Queues erweitern
        → import threading
        → import queue

      - Im Beispiel vom Hauptprogramm soll die 'put' Methode benutzt werden, um neue Zahlen in den »Briefkasten« (Warteschlange) zu werfen

# Python
# Parallele Programmierung



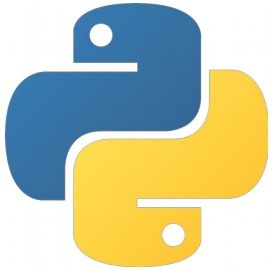
- Prozesse, Multitasking und Threads
  - Worker-Threads und Queues

    - Hinweis
      - Durch den Aufruf von thread.setDaemon(True) werden die Threads als sogenannte Dämon-Threads markiert

      - Der Unterschied zwischen Dämon-Threads und normalen Threads besteht darin, dass ein Programm beendet wird, wenn nur noch Dämon-Threads laufen

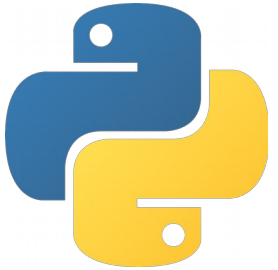
      - Bei normalen Threads kann das Programm so lange laufen, bis auch der letzte Thread beendet worden ist

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Klassen für sehr spezielle Zwecke im threading
  - Ereignisse definieren – threading.Event
    - Mit der Klasse threading.Event können sogenannte Ereignisse definiert werden, um Threads bis zum Eintritt eines bestimmten Ereignisses zu unterbrechen
    - Ein Thread, der die wait-Methode eines frisch erzeugten threading.Event-Objekts aufruft, wird so lange unterbrochen, bis ein anderer Thread das Event mit set auslöst
    - Ausführliche Informationen der Doku entnehmen

# Python
# Parallele Programmierung

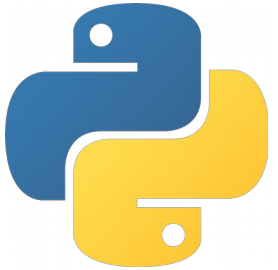


- Prozesse, Multitasking und Threads
  - Klassen für sehr spezielle Zwecke im threading
    - Barrieren definieren – threading.Barrier

      - Um mehrere Threads gegenseitig aufeinander warten zu lassen, dient die Klasse threading.Barrier

      - Ausführliche Informationen der Doku entnehmen

# Python
# Parallele Programmierung

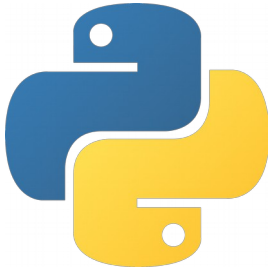


- Prozesse, Multitasking und Threads
  - Klassen für sehr spezielle Zwecke im threading
    - Eine Funktion zeitlich versetzt ausführen – threading.Timer

      - Das threading-Modul bietet eine praktische Klasse namens threading.Timer, um Funktionen nach dem Verstreichen einer gewissen Zeit aufzurufen

      - Aufbau
        threading.Timer(interval, function, args=[], kwargs={})

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Klassen für sehr spezielle Zwecke im threading
    - threading.Timer

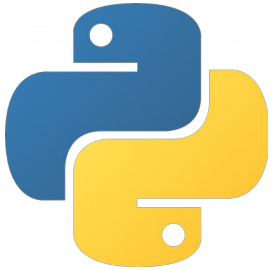
      - Aufbau
        threading.Timer(interval, function, args=[], kwargs={})

      - interval des Konstruktors gibt die Zeit in Sekunden an, die gewartet werden soll (Ganz-, Fließkommazahlen)

      - die für function übergebene Funktion wird aufgerufen, wenn das interval abgelaufen ist

      - Für args und kwargs kann eine Liste bzw. ein Dictionary übergeben werden, das die Parameter enthält, mit denen function aufgerufen werden soll

# Python
# Parallele Programmierung

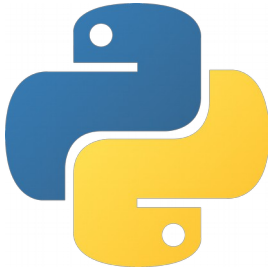


- Prozesse, Multitasking und Threads
  - Klassen für sehr spezielle Zwecke im threading
    - threading.Timer

      - Beispiel

```
>>> import time, threading
>>> def wecker(gestellt):
        print("RIIIIIIIING!!!")
        print("Der Wecker wurde um {0} Uhr
                                gestellt.".format(gestellt))
print("Es ist {0} Uhr".format(time.strftime("%H:%M:
                                        %S")))
>>> timer = threading.Timer(30, wecker,
                            [time.strftime("%H:%M:%S")])
>>> timer.start()
```

# Python
# Parallele Programmierung



- Prozesse, Multitasking und Threads
  - Klassen für sehr spezielle Zwecke im threading
    - threading.Timer

      - Ausgabe ( 30 sekunden später)
        >>> RIIIIIIIING!!!
        Der Wecker wurde um 08:11:26 Uhr gestellt.
        Es ist 08:11:58 Uhr

      - Mit der Methode start beginnt der Timer zu laufen
      - ruft dann nach der festgelegten Zeitspanne die übergebene Funktion ( wecker ) auf
      - Die Differenz von zwei Sekunden rührt daher, dass zwischen dem Erstellen des Timer-Objekts und dem Aufrufen der start -Methode zwei Sekunden vergangen sind
      - Nachdem die start-Methode aufgerufen wurde, kann der Timer außerdem mit der parameterlosen cancel-Methode wieder abgebrochen werden