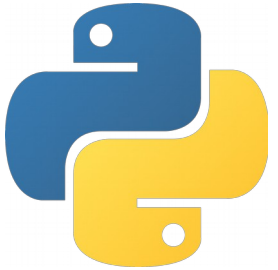
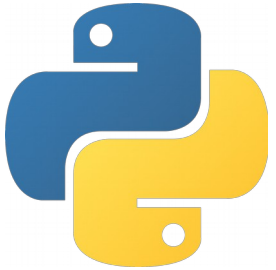
Python Optimierung



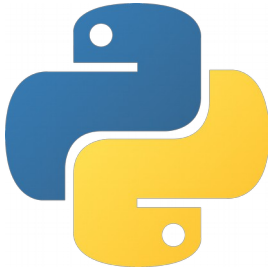
- Die Optimize-Option
  - Grundsätzlich können Sie das Laufzeitverhalten eines Python-Programms beeinflussen, indem Sie es mit der Kommandozeilenoption -O ausführen
  - Diese Option veranlasst den Interpreter dazu, den resultierenden Byte-Code zu optimieren
  - Das bedeutet, dass assert-Anweisungen und Konstrukte wie
    if __debug__:
        mache_etwas()
    nicht ins Kompilat aufgenommen werden und somit keinen Einfluss mehr auf das Laufzeitverhalten des optimierten Programms haben
  - Der optimierte Byte-Code wird in Dateien mit der Dateiendung .pyo gespeichert

Python Optimierung



- Die Optimize-Option
  - Durch die Kommandozeilenoption -OO ist es möglich, das Programm über das normale Maß hinaus zu optimieren
  - Wenn dies gewünscht ist, werden alle im Quelltext enthaltenen Docstrings ignoriert und nicht mit in das Kompilat aufgenommen
  - Auch damit erreichen Sie ein wenig mehr Laufzeiteffizienz, wenngleich sich der Gewinn in Grenzen halten sollte
  - Es ist dann aber beispielsweise für die Built-in Function help nicht mehr möglich, eine Hilfeseite zu Elementen Ihres Moduls zu generieren, weil keine Docstrings mehr vorhanden sind

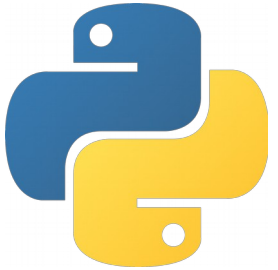
Python Optimierung



- Mutable vs. immutable
  - Dass diese Unterscheidung auch performancetechnische Relevanz hat, wissen wir
  - Im folgenden Beispiel soll ein Tupel mit den Zahlen von 0 bis 1.000 gefüllt werden
  - Selbstverständlich kennen wir dafür effiziente Möglichkeiten, doch versuchen wir es einmal mit der naiven Herangehensweise
  - Ausgang

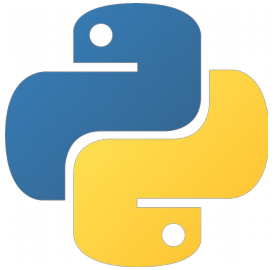
```
tup = ()
for i in range(1000):
        tup += (i,)
```

Python Optimierung



- Mutable vs. immutable
  - Zunächst wird das Tupel angelegt und dann in einer Schleife über alle Zahlen von 0 bis 1.000 die jeweils aktuelle Zahl an das Tupel angehängt
  - Leider haben wir beim Schreiben dieses Codes nicht berücksichtigt, dass es sich bei tuple um einen unveränderlichen Datentyp handelt
  - Das bedeutet, dass beim Hinzufügen eines Wertes zu unserem Tupel tup jedes Mal eine neue tuple-Instanz erzeugt wird und die Einträge der alten Instanz in diese neue umkopiert werden müssen
  - Das kostet Zeit, manchmal auch viel Zeit

Python Optimierung

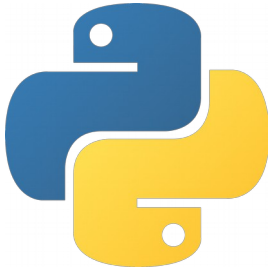


- Mutable vs. immutable
  - Als Alternative zu obigem Beispiel verwenden wir nun das veränderliche Pendant des Tupels, die Liste

```
lst = []
for i in range(1000):
        lst += [i]
```

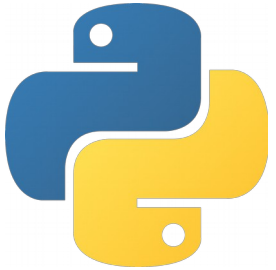
  - Im Gegensatz zur vorherigen Version muss hier beim Anhängen einer neuen Zahl keine neue list -Instanz erzeugt werden, was sich beim Vergleich der Laufzeit der beiden Beispiele deutlich niederschlägt
  - Die zweite Variante kann bis zu 90 % schneller ausgeführt werden als die erste

Python Optimierung



- Mutable vs. immutable
  - Interessant ist in diesem Beispiel auch die Entwicklung des Laufzeitunterschieds in Abhängigkeit von der Anzahl der einzutragenden Zahlen
  - Je mehr Elemente ein Tupel enthält, desto aufwendiger ist es, seine Elemente in eine neue tuple-Instanz zu überführen
  - Hätten wir die obigen Beispiele also mit Zahlen zwischen 0 und 10.000 durchgeführt, hätte sich ein noch um einiges eindrucksvollerer Laufzeitunterschied zwischen den beiden Varianten ergeben

Python Optimierung

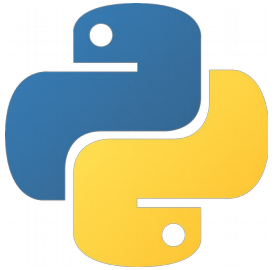


- Schleifen
  - Betrachten wir noch einmal das zweite Beispiel

```
lst = []
for i in range(1000):
        lst += [i]
```

  - Wenn eine Schleife zum Erzeugen einer Liste, eines Dictionarys oder eines Sets verwendet wird, sollten Sie sich stets überlegen, ob Sie dasselbe Ergebnis nicht auch mit einer List bzw. Dict oder Set Comprehension erreichen können
  - Diese können schneller ausgeführt werden als eine analoge for-Schleife

Python Optimierung

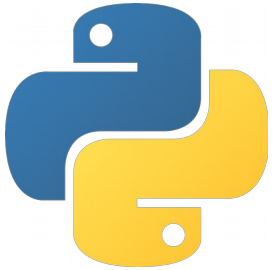


- Schleifen
  - Die folgende List Comprehension, die die gleiche Liste wie die obige Schleife erzeugt, kann um ca. 60 % schneller ausgeführt werden

    lst = [i for i in range(1000)]

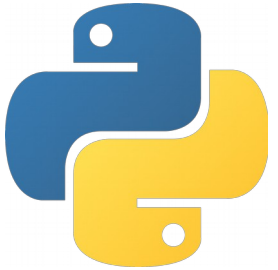
  - Hinweis
    - Ein Aufruf der Built-in Function map ist ähnlich effizient wie eine List Comprehension

Python Optimierung



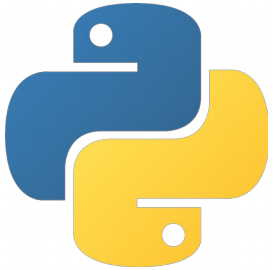
- Funktionsaufrufe
  - Eine weitere, laufzeittechnisch gesehen teure Angelegenheit sind Funktionsaufrufe, weswegen Sie Funktionsaufrufe in häufig durchlaufenen Schleifen auf ihre Notwendigkeit hin überprüfen sollten
  - Bei besonders simplen Funktionen kann es sinnvoll sein, die Funktion zu »inlinen« , den Funktionsinhalt also direkt in die Schleife zu schreiben

Python Optimierung



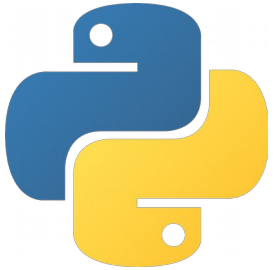
- Programmiersprache C
  - Ein in Python geschriebener Programmteil ist aufgrund des zwischengeschalteten Interpreters in der Regel langsamer als ein vergleichbares C-Programm
  - Aus diesem Grund sollten Sie an einer laufzeitkritischen Stelle so oft wie möglich auf Algorithmen zurückgreifen, die in C implementiert wurden
  - So lohnt es sich beispielsweise immer, eine Built-in Function einzusetzen, anstatt den entsprechenden Algorithmus selbst zu implementieren

Python Optimierung



- Lookup
  - Wenn über einen Modulnamen auf eine Funktion zugegriffen wird, die in diesem Modul enthalten ist, muss bei jedem Funktionsaufruf ein sogenannter Lookup durchgeführt werden
  - Dieser Lookup muss nicht durchgeführt werden, wenn eine direkte Referenz auf das Funktionsobjekt besteht
  - Stellen Sie sich einmal vor, Sie wollten die Quadratwurzeln aller natürlichen Zahlen zwischen 0 und 100 bestimmen

Python Optimierung



- Lookup

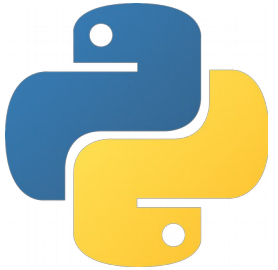
```
import math
wurzeln = [math.sqrt(i) for i in range(100)]
```

  - Wesentlich effizienter ist es jedoch, die Funktion sqrt des Moduls math direkt zu referenzieren und über diese Referenz anzusprechen

```
import math
s = math.sqrt
wurzeln = [s(i) for i in range(100)]
```

  - Letzte kann um ca. 20 % schneller ausgeführt werden

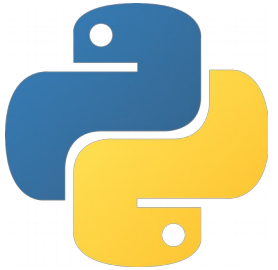
Python Optimierung



- Exceptions
  - Im folgenden Beispiel möchten wir in einer sehr frequentierten Schleife mit einem sich ständig ändernden Index i auf eine Liste liste zugreifen, können uns aber nicht sicher sein, ob ein Element liste[i] tatsächlich existiert
  - Wenn ein Element mit dem Index i existiert, soll dieses zurückgegeben werden, andernfalls 0
  - In einem solchen Fall ist es in der Regel ineffizient, vor dem Zugriff zu prüfen, ob ein i-tes Element existiert

```
def f(liste, i):
    if i in liste:
        return liste[i]
    else:
        return 0
```

Python Optimierung

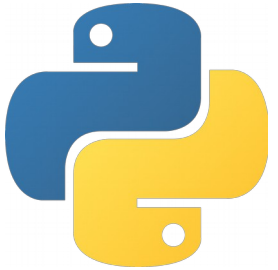


- Exceptions
  - In der Regel ist es wesentlich effizienter, einfach auf das i-te Element zuzugreifen und im Falle einer geworfenen IndexError-Exception den Wert 0 zurückzugeben

```
def f(liste, i):
    try:
        return liste[i]
    except IndexError:
        return 0
```

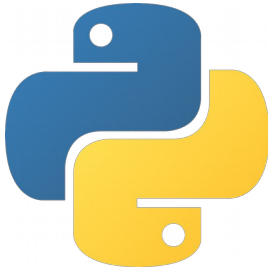
  - Bei einer Liste mit 100 Einträgen ist die untere Variante der Funktion f, unabhängig davon, wie häufig eine Exception geworfen werden muss, um ca. 35 % schneller als die erste

Python Optimierung



- Exceptions
  - Je kleiner die Liste, desto effizienter ist das Durchsuchen im Vergleich zum Behandeln von Exceptions

Python Optimierung



- Keyword Arguments
  - Das Übergeben von Positionsparametern beim Funktions- oder Methodenaufruf ist im Vergleich zu Schlüsselwortparametern grundsätzlich effizienter
  - Dazu soll folgende Funktion betrachtet werden, die vier Parameter erwartet
    def f(a, b, c, d):
        return "{} {} {} {}".format(a,b,c,d)
  - Der Funktionsaufruf
    f("Hallo", "du", "schöne", "Welt")
  - ist ca. 18% schneller als
    f(a="Hallo", b="du", c="schöne", d="Welt")