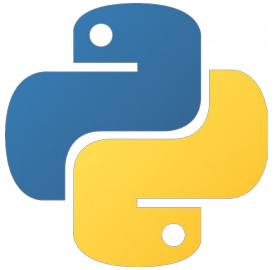
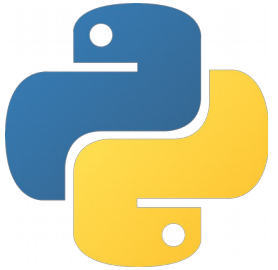
Python Debugging



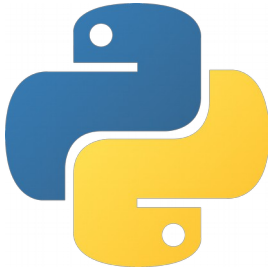
- Das Debugging und die Aufgabe
  - Automatisiertes Testen
    - Pythons Standardbibliothek stellt zwei Module zur testgetriebenen Entwicklung (engl. test-driven development) bereit
    - Unter testgetriebener Entwicklung versteht man eine Art der Programmierung, bei der viele kleine Abschnitte des Programms, sogenannte Units, durch automatisierte Testdurchläufe auf Fehler geprüft werden
    - Bei der testgetriebenen Entwicklung wird das Programm nach kleineren, in sich geschlossenen Arbeitsschritten so lange verbessert, bis es wieder alle bisherigen und alle hinzugekommenen Tests besteht

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
    - Auf diese Weise können sich durch das Hinzufügen von neuem Code keine Fehler in alten, bereits getesteten Code einschleichen
    - In Python ist das Ihnen möglicherweise bekannte Konzept der Unit Tests im Modul unittest implementiert
    - Das Modul doctest ermöglicht es, Testfälle innerhalb eines Docstrings, beispielsweise einer Funktion, unterzubringen
    - Erst Modul doctest, dann Modul unittest

Python Debugging



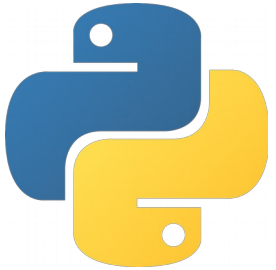
- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

    - Das Modul doctest erlaubt es, Testfälle innerhalb des Docstrings einer Funktion, Methode, Klasse oder eines Moduls zu erstellen, die beim Aufruf der im Modul doctest enthaltenen Funktion testmod getestet werden

    - Die Testfälle innerhalb eines Docstrings werden dabei nicht in einer neuen Definitionssprache verfasst, sondern können direkt aus einer Sitzung im interaktiven Modus in den Docstring kopiert werden

    - Docstrings sind auch bzw. hauptsächlich für die Dokumentation beispielsweise einer Funktion gedacht

Python Debugging



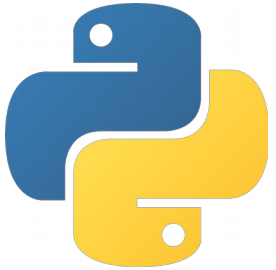
- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

    - Aus diesem Grund sollten Sie die Testfälle im Docstring möglichst einfach und lehrreich halten, sodass der resultierende Docstring auch in Dokumentationen Ihres Programms verwendet werden kann

    - Das folgende Beispiel erläutert die Verwendung des Moduls doctest anhand der Funktion fak, die die Fakultät einer ganzen Zahl berechnen und zurückgeben soll

      import doctest

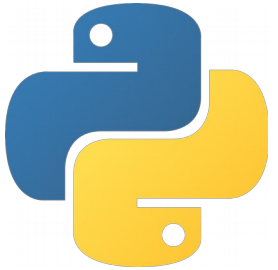
Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

```
def fak(n):
    """
    Berechnet die Fakultaet einer ganzen Zahl.

    >>> fak(5)
    120
    .
    >>> fak(10)
    3628800

    >>> fak(20)
    2432902008176640000

    Es muss eine positive ganze Zahl uebergeben werden.
```

Python Debugging

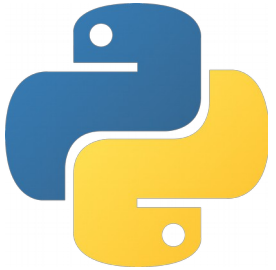


- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

```
    >>> fak(-1)
    Traceback (most recent call last):
    ...
    ValueError: Keine negativen Zahlen!
    """
  res = 1
  for i in range(2, n+1):
    res *= i
  return res

if __name__ == "__main__":
doctest.testmod()
```

  - Im Docstring der Funktion fak steht zunächst ein erklärender Text

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

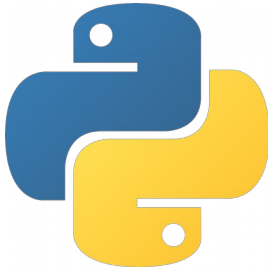
    - Dann folgt, durch eine leere Zeile davon abgetrennt, ein Auszug aus Pythons interaktivem Modus, in dem Funktionsaufrufe von fak mit ihren Rückgabewerten stehen

    - Diese Testfälle werden beim Ausführen des Tests nachvollzogen und entweder für wahr oder für falsch befunden

    - Auf diese einfachen Fälle folgen, jeweils durch eine Leerzeile eingeleitet, ein weiterer erklärender Text sowie ein Ausnahmefall, in dem eine negative Zahl übergeben wurde

    - Beachten Sie, dass Sie den Stacktrace eines auftretenden Tracebacks im Docstring weglassen können

Python Debugging

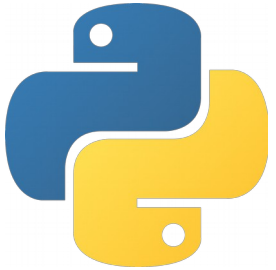


- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest
    - Auch die im Beispiel stattdessen geschriebenen Auslassungszeichen sind optional
    - Der letzte Testfall wurde in der Funktion noch nicht berücksichtigt, sodass dieser im Test fehlschlagen wird
    - Um den Test zu starten, muss die Funktion testmod des Moduls doctest aufgerufen werden
    - Aufgrund der if-Abfrage

      ```
      if __name__ == "__main__":
          doctest.testmod()
      ```

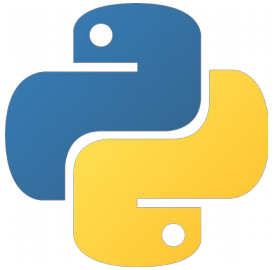
      wird diese Funktion immer dann aufgerufen, wenn die Programmdatei direkt ausgeführt wird

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest
    - Der Test wird hingegen nicht durchgeführt, wenn die Programmdatei von einem anderen Python-Programm als Modul eingebunden wird
    - Ausgabe durch provozierten Fehlerfall auf der nächsten Folie

Python Debugging

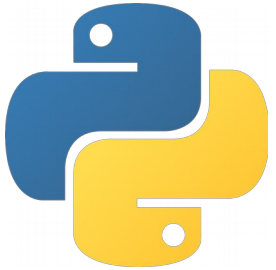


- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

    - Jetzt erweitern wir die Funktion fak dahingehend, dass sie im Falle eines negativen Parameters die gewünschte Exception wirft

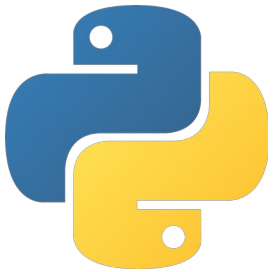
```
def fak(n):
    """
    [...]
    """
    if n < 0:
        raise ValueError("Keine negativen Zahlen!")

    res = 1
    for i in range(2, n+1):
        res *= i
    return res
```

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest
    - Durch diese Änderung werden bei erneutem Durchführen des Tests keine Fehler mehr angezeigt
    - Um genau zu sein: Es wird überhaupt nichts angezeigt
    - Das liegt daran, dass generell nur fehlgeschlagene Testfälle auf dem Bildschirm ausgegeben werden
    - Sollten Sie auch auf die Ausgabe geglückter Testfälle bestehen, starten Sie die Programmdatei mit der Option -v (für verbose)

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

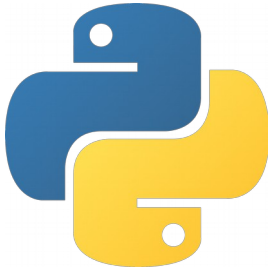
    - Bei der Verwendung von Doctests ist zu beachten, dass die in den Docstrings geschriebenen Vorgaben Zeichen für Zeichen mit den Ausgaben der intern ausgeführten Testfälle verglichen werden

    - Dabei sollten Sie stets im Hinterkopf behalten, dass die Ausgaben bestimmter Datentypen nicht immer gleich sind

    - So stehen beispielsweise die Schlüssel-Wert-Paare eines Dictionarys in keiner garantierten Reihenfolge, sodass Sie innerhalb eines Doctests nie ein Dictionary als Ergebnis ausgeben sollten

    - Des Weiteren gibt es Informationen, die vom Interpreter oder anderen Gegebenheiten abhängen
      - Beispielsweise entspricht die Identität einer Instanz intern ihrer Speicheradresse und wird sich deswegen natürlich beim Neustart des Programms ändern

Python Debugging

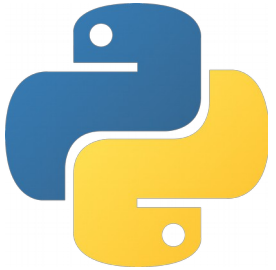


- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest

    - Eine weitere Besonderheit, auf die Sie achten müssen, ist, dass eine Leerzeile in der erwarteten Ausgabe einer Funktion durch den String <BLANKLINE> gekennzeichnet werden muss, da eine Leerzeile als Trennung zwischen Testfällen und Dokumentation fungiert

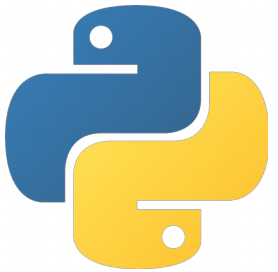
```
def f(a, b):
    """
    >>> f(3, 4)
    7
    <BLANKLINE>
    12
    """
    print(a + b)
    print()
    print(a * b)
```

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest
  - Flags
    - Um einen Testfall genau an Ihre Bedürfnisse anzupassen, können Sie sogenannte Flags vorgeben
    - Das sind Einstellungen, die Sie aktivieren oder deaktivieren können
    - Ein Flag wird in Form eines Kommentars hinter den Testfall im Docstring geschrieben
    - Wird das Flag von einem Plus (+) eingeleitet, wird es aktiviert, bei einem Minus (-) deaktiviert
    - Bevor wir zu einem konkreten Beispiel kommen, sollen die drei wichtigsten Flags eingeführt werden

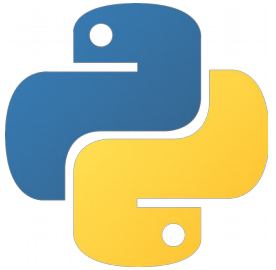
Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest
  - Flags

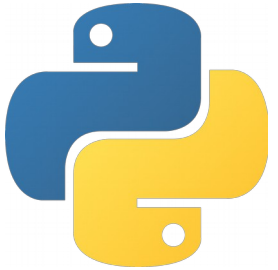
Flag	Bedeutung
ELLIPSIS	Wenn dieses Flag gesetzt ist, kann die Angabe . . . für eine beliebige Ausgabe einer Funktion verwendet werden. So können veränderliche Angaben wie Speicheradressen oder Ähnliches in größeren Ausgaben überlesen werden.
NORMALIZE_WHITESPACES	Wenn dieses Flag gesetzt ist, werden Whitespace-Zeichen nicht in den Ergebnisvergleich einbezogen. Das ist besonders dann interessant, wenn Sie ein langes Ergebnis auf mehrere Zeilen umbrechen möchten.
SKIP	Dieses Flag veranlasst das Überspringen des Tests. Das ist beispielsweise dann nützlich, wenn Sie im Docstring zu Dokumentationszwecken eine Reihe von Beispielen liefern, aber nur wenige davon bei einem Testlauf berücksichtigt werden sollen.

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest
  - Flags
    - In einem einfachen Beispiel möchten wir den Doctest der bereits bekannten Fakultätsfunktion um die Berechnung der Fakultät einer relativ großen Zahl erweitern
    - Da es müßig wäre, alle Stellen des Ergebnisses im Doctest anzugeben, soll die Zahl mithilfe des Flags ELLIPSIS gekürzt angegeben werden
    - Beispiel auf der folgenden Folie

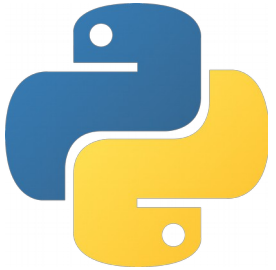
Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Testfälle in Docstrings – doctest
  - Flags

```
def fak(n):
    """
    Berechnet die Fakultaet einer ganzen Zahl.

    >>> fak(1000) # doctest: +ELLIPSIS
    402387260077093773543702...000
    >>> fak("Bla") # doctest: +SKIP
    'BlubbBlubb'
    """
    res = 1
    for i in range(2, n+1):
        res *= i
    return res

if __name__ == "__main__":
doctest.testmod()
```

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest

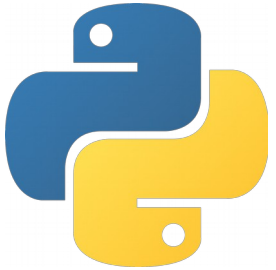
    - Das zweite Modul zur testgetriebenen Entwicklung heißt unittest und ist ebenfalls in der Standardbibliothek enthalten

    - Das Modul unittest implementiert die Funktionalität des aus Java bekannten Moduls JUnit, das den De-facto-Standard zur testgetriebenen Entwicklung in Java darstellt

    - Der Unterschied zum Modul doctest besteht darin, dass die Testfälle bei unittest außerhalb des eigentlichen Programmcodes in einer eigenen Programmdatei in Form von regulärem Python-Code definiert werden

    - Das vereinfacht die Ausführung der Tests und hält die Programmdokumentation sauber

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest

    - Umgekehrt ist mit dem Erstellen der Testfälle allerdings mehr Aufwand verbunden

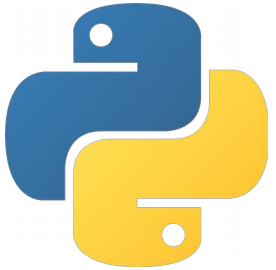
    - Um einen neuen Testfall mit unittest zu erstellen, müssen Sie eine von der Basisklasse unittest.TestCase abgeleitete Klasse erstellen, in der einzelne Testfälle als Methoden implementiert sind

    - Die folgende Klasse implementiert die gleichen Testfälle, die wir im vorherigen Abschnitt mit dem Modul doctest durchgeführt haben

    - Dabei muss die zu testende Funktion fak in der Programmdatei fak.py implementiert sein, die von unserer Test-Programmdatei als Modul eingebunden wird

    - Folgende Seite zeigt ein entsprechendes Beispiel

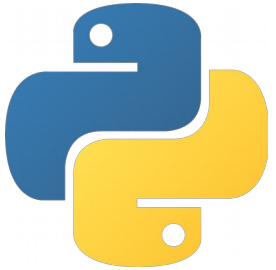
Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest

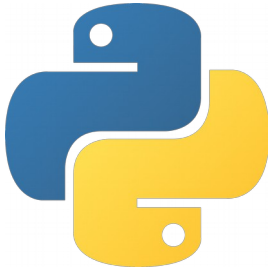
```
import unittest
import fak

class MeinTest(unittest.TestCase):
    def testBerechnung(self):
        self.assertEqual(fak.fak(5), 120)
        self.assertEqual(fak.fak(10), 3628800)
        self.assertEqual(fak.fak(20), 2432902008176640000)

    def testAusnahmen(self):
        self.assertRaises(ValueError, fak.fak, -1)

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

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest
    - Es wurde eine Klasse namens MeinTest erzeugt, welche von der Basisklasse unittest.TestCase erbt
    - In der Klasse MeinTest wurden zwei Testmethoden namens testBerechnung und testAusnahmen implementiert
    - Beachten Sie, dass der Name solcher Testmethoden mit test beginnen muss, damit sie später auch tatsächlich zum Testen gefunden und ausgeführt werden
    - Innerhalb der Testmethoden werden die Methoden assertEqual bzw. assertRaises verwendet, die den Test fehlschlagen lassen, wenn die beiden angegebenen Werte nicht gleich sind bzw. wenn die angegebene Exception nicht geworfen wurde

Python Debugging



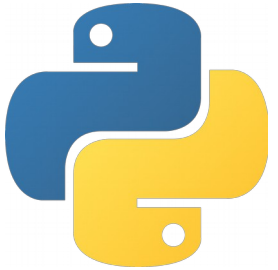
- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest
    - Um den Testlauf zu starten, wird die Funktion unittest.main aufgerufen. Die Fallunterscheidung

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

      bewirkt, dass der Unit Test nur durchgeführt wird, wenn die Programmdatei direkt ausgeführt wird, und ausdrücklich nicht, wenn die Programmdatei als Modul in ein anderes Python-Programm importiert wurde

    - Die aufgerufene Funktion unittest.main erzeugt, um den Test durchzuführen, Instanzen aller Klassen, die im aktuellen Namensraum existieren und von unittest.TestCase erben

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest

    - Dann werden alle Methoden dieser Instanzen aufgerufen, deren Namen mit test beginnen

      Die Ausgabe des Beispiels lautet im Erfolgsfall:

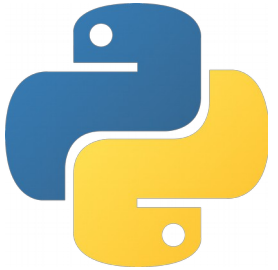
      ..

      Ran 2 tests in 0.000s
      20
      OK

    - Dabei stehen die beiden Punkte zu Beginn für zwei erfolgreich durchgeführte Tests

    - Ein fehlgeschlagener Test würde durch ein F gekennzeichnet

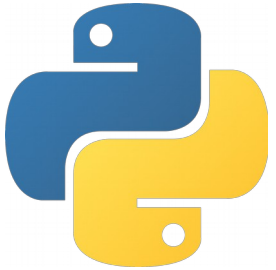
Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest
    - Im Fehlerfall wird die genaue Bedingung angegeben, die zum Fehler geführt hat

```
.F
======================================
FAIL: testBerechnung (__main__.MeinTest)
--------------------------------------------------------------
Traceback (most recent call last):
   File "testen.py", line 7, in testBerechnung
      self.assertEqual(fak.fak(5), 12)
AssertionError: 120 != 12
--------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Python Debugging



- Das Debugging und die Aufgabe
  - Automatisiertes Testen
  - Unit Tests – unittest

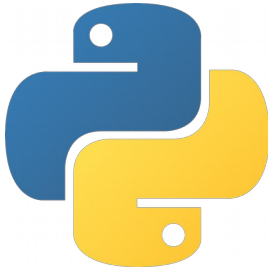
    - Aufgabe
      Schauen Sie sich die Klasse TestCase des Moduls an

      Implementieren Sie einen Testablauf, auf Basis der Nutzung der Klasse TestCase

      Probieren Sie enthaltenen Methoden aus

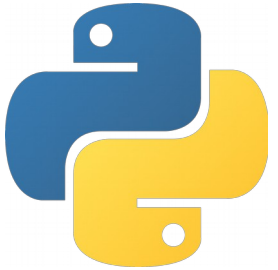
      Besprechen Sie die Ergebnisse und Erkenntnisse mit Ihren Python Kollegen

Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - Ein Traceback-Objekt hält den Kontext fest, aus dem eine Exception geworfen wurde, und liefert damit die Informationen, die bei einem Traceback auf dem Bildschirm angezeigt werden
    - Zu diesen Informationen gehört vor allem die Funktionshierarchie, der sogenannte Callstack
    - Ein Traceback-Objekt wird beim Werfen einer Exception automatisch erzeugt
    - Generell können Sie auf das Traceback-Objekt einer gerade abgefangenen Exception über die Funktion sys.exc_info des Moduls sys zugreifen

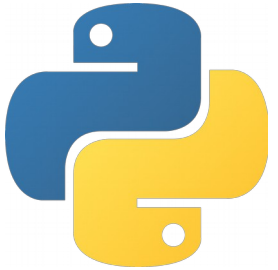
Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - Alle Beispiele beziehen sich auf folgenden Aufbau (Kontext)

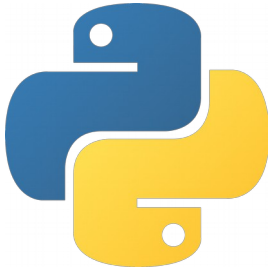
```
>>> import traceback
>>> import sys
>>> def f1():
…         raise TypeError
...
>>> def f2():
…         f1()
...
>>> try:
…         f2()
...    except TypeError:
…         tb = sys.exc_info()[2]
...
>>> tb
<traceback object at 0xb7c49414>
```

Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - Es wird ein Traceback-Objekt tb erzeugt, das den Callstack einer TypeError-Exception beschreibt, die zuvor aus einer verschachtelten Funktionshierarchie heraus geworfen wurde
    - print_tb(traceback[, limit[, file]])
      - Diese Funktion gibt den Stacktrace des Traceback-Objekts formatiert auf dem Bildschirm aus
      - Über den optionalen Parameter limit geben Sie an, wie viele Einträge des Stacktraces maximal ausgegeben werden sollen
      - Für den dritten, optionalen Parameter kann ein geöffnetes Dateiobjekt übergeben werden, in das der Stacktrace geschrieben wird
      - Standardmäßig wird in den Stream sys.stderr geschrieben

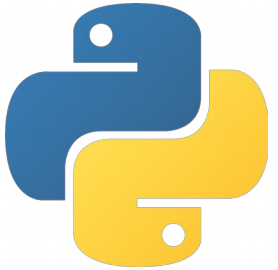
Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - print_tb(traceback[, limit[, file]])

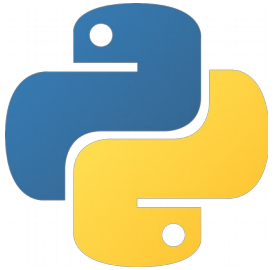
```
>>> traceback.print_tb(tb)
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in f2
  File "<stdin>", line 2, in f1
```

Python Debugging



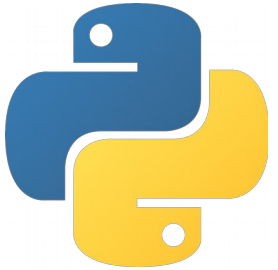
- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - print_exception(type, value, traceback[, limit[, file[, chain]]])
      - Diese Funktion gibt einen vollständigen Traceback auf dem Bildschirm aus
      - Die Ausgabe ist genauso formatiert wie die einer normalen, nicht abgefangenen Exception
      - Für die beiden Parameter type und value müssen Exception-Typ und Exception-Wert übergeben werden
      - Die restlichen Parameter haben dieselbe Bedeutung wie bei print_tb

Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - print_exc([limit[, file[, chain]]])
      - Diese Funktion arbeitet wie print_exception, jedoch immer für die aktuell abgefangene Exception
      - Die Parameter limit, file und chain haben dieselbe Bedeutung wie bei print_exception
      - Diese Funktion kann nur innerhalb eines except-Zweiges aufgerufen werden

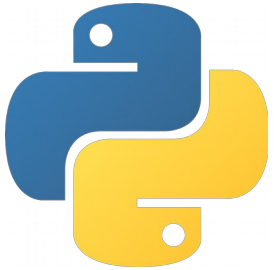
Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - print_exc([limit[, file[, chain]]])

```
>>> try:
...         raise TypeError
...    except TypeError:
...         traceback.print_exc()
...
Traceback (most recent call last):
   File "<stdin>", line 2, in <module>
TypeError
```

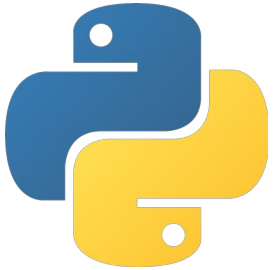
Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - extract_tb(traceback[, limit])
      - Diese Funktion gibt eine Liste von aufbereiteten Stacktrace-Einträgen des Traceback-Objekts traceback zurück
      - Ein aufbereiteter Stacktrace-Eintrag ist ein Tupel der folgenden Form
        (Dateiname, Zeilennummer, Funktionsname, Text)
      - Der optionale Parameter limit hat die gleiche Bedeutung wie bei print_tb

```
>>> traceback.extract_tb(tb)
[('<stdin>', 2, '<module>', None),
('<stdin>', 2, 'f2', None),
('<stdin>', 2, 'f1', None)]
```

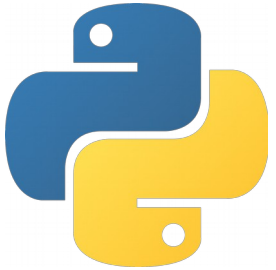
Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - format_list(lst)
      - Die Funktion format_list bekommt eine Liste von Tupeln übergeben, wie sie beispielsweise von der Funktion extract_tb zurückgegeben wird
      - Aus diesen Informationen erzeugt format_list eine Liste von aufbereiteten Strings der folgenden Form

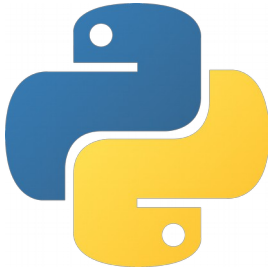
        >>> traceback.format_list(traceback.extract_tb(tb))
        [' File "<stdin>", line 2, in <module>\n',
        ' File "<stdin>", line 2, in f2\n',
        ' File "<stdin>", line 2, in f1\n']

Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - format_exception(type, value, tb[, limit[, chain]])
      - Diese Funktion formatiert eine Exception mit dem Typ type, dem Wert value und dem Stacktrace tb zu einer Liste von Strings
      - Jeder String dieser Liste repräsentiert eine Zeile der Ausgabe
      - Die Parameter limit und chain haben die gleiche Bedeutung wie bei der Funktion print_exception

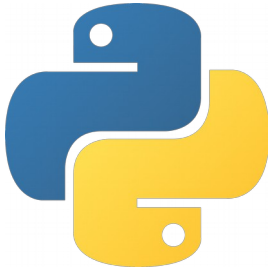
Python Debugging



- Das Debugging und die Aufgabe
  - Traceback-Objekte – traceback
    - format_exception(type, value, tb[, limit[, chain]])

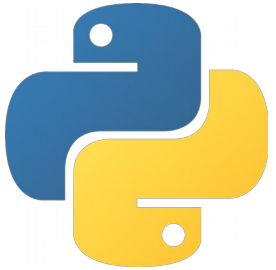
```
>>> traceback.format_exception(TypeError,
... TypeError("Hallo Welt"), tb)

['Traceback (most recent call last):\n',
'      File "<stdin>", line 2, in <module>\n',
'      File "<stdin>", line 2, in f2\n',
'      File "<stdin>", line 2, in f1\n',
'TypeError: Hallo Welt\n']
```

Python Debugging



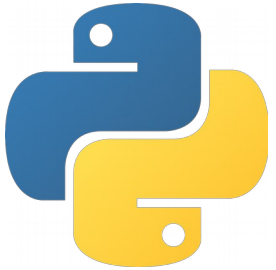
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
    - Die Optimierung eines Programms ist ein wichtiger Teilbereich der Programmierung und kann viel Zeit in Anspruch nehmen
    - In der Regel wird zunächst ein lauffähiges Programm erstellt, das alle gewünschten Anforderungen erfüllt, bei dem jedoch noch nicht unbedingt Wert auf die Optimierung der Algorithmik gelegt wird
    - Das liegt vor allem daran, dass man oftmals erst beim fertigen Programm die tatsächlichen Engpässe erkennt und im frühen Stadium somit eventuell viel Zeit in die Optimierung völlig unkritischer Bereiche investieren würde
    - Um das Laufzeitverhalten eines Python-Programms möglichst genau zu erfassen, existieren die drei Module timeit, profile und cProfile in der Standardbibliothek von Python

Python Debugging



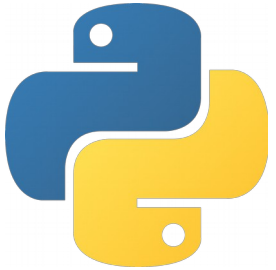
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - Das Modul timeit der Standardbibliothek ermöglicht es, genau zu messen, wie lange ein Python-Programm zur Ausführung braucht
    - Um die Laufzeit eines Python-Codes zu testen, muss die im Modul timeit enthaltene Klasse Timer instanziiert werden
    - Der Konstruktor der Klasse Timer
      Timer([stmt[, setup[, timer]]])
    - Nachdem eine Instanz der Klasse Timer erzeugt wurde, besitzt sie drei Methoden, die im Folgenden besprochen werden sollen
    - Dabei sei t eine Instanz der Klasse Timer

Python Debugging



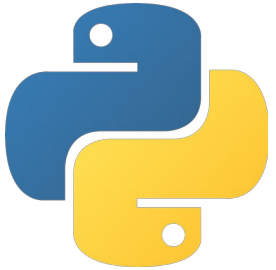
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - t.timeit([number])
      - Diese Methode führt zunächst den setup-Code einmalig aus und wiederholt danach den beim Konstruktor für stmt übergebenen Code number-mal
      - Wenn der optionale Parameter number nicht angegeben wurde, wird der zu messende Code 1.000.000-mal ausgeführt
      - Die Funktion gibt die Zeit zurück, die das Ausführen des gesamten Codes (also inklusive aller Wiederholungen, jedoch exklusive des Setup-Codes) in Anspruch genommen hat
      - Der Wert wird in Sekunden als Gleitkommazahl zurückgegeben

Python Debugging



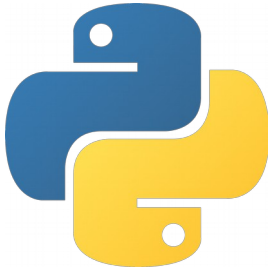
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - t.timeit([number])
      - Hinweis
        - Um das Ergebnis von äußeren Faktoren möglichst unabhängig zu machen, wird für die Dauer der Messung die Garbage Collection des Python-Interpreters deaktiviert
        - Sollte die Garbage Collection ein wichtiger mitzumessender Teil Ihres Codes sein, so lässt sie sich mit einem Setup-Code von "gc.enable()" wieder aktivieren

Python Debugging



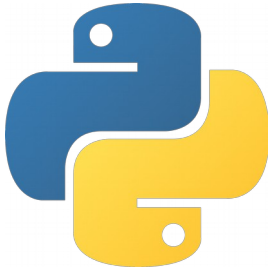
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - t.repeat([repeat[, number]])
      - Diese Methode ruft die Methode timeit repeat-mal auf und gibt die Ergebnisse in Form einer Liste von Gleitkommazahlen zurück
      - Der Parameter number wird dabei der Methode timeit bei jedem Aufruf übergeben

Python Debugging



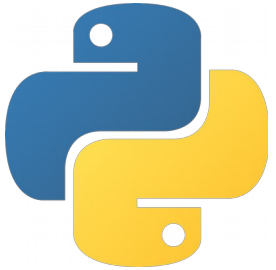
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - t.repeat([repeat[, number]])
      - Hinweis
        - Es ist normalerweise keine gute Idee, den Mittelwert aller von repeat zurückgegebenen Werte zu bilden und diesen als durchschnittliche Laufzeit auszugeben
        - Andere Prozesse, die auf Ihrem System laufen, verfälschen die Ergebnisse aller Messungen
        - Vielmehr sollten Sie den kleinsten Wert der zurückgegebenen Liste als minimale Laufzeit annehmen, da dies die Messung mit der geringsten Systemaktivität war

Python Debugging



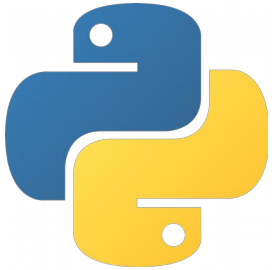
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - t.print_exc([file])
      - Sollte im zu analysierenden Code eine Exception geworfen werden, wird die Analyse sofort abgebrochen und ein Traceback ausgegeben
      - Der Stacktrace dieses Tracebacks ist jedoch nicht immer optimal, da er sich nicht auf den tatsächlich ausgeführten Quellcode bezieht
      - Um einen aussagekräftigeren Stacktrace auszugeben, können Sie eine geworfene Exception abfangen und die Methode print_exc aufrufen

Python Debugging



- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - t.print_exc([file])
      - Diese Methode gibt einen Traceback auf dem Bildschirm aus, der sich direkt auf den zu analysierenden Code bezieht und damit die Fehlersuche erleichtert
      - Durch Angabe des optionalen Parameters file leiten Sie die Ausgabe in eine Datei um

Python Debugging

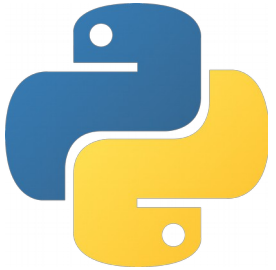


- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit

    - Beispiel

```
import timeit

def fak1(n):
    res = 1
    for i in range(2, n+1):
        res *= i
    return res

def fak2(n):
    if n > 0:
        return fak2(n-1)*n
    else:
        return 1
```

Python Debugging



- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit

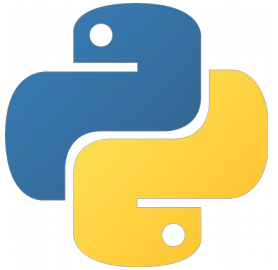
    - Beispiel
      Danach erzeugen wir für beide Funktionen jeweils eine Instanz der Klasse Timer

      t1 = timeit.Timer("fak1(50)", "from __main__ import fak1")
      t2 = timeit.Timer("fak2(50)", "from __main__ import fak2")

    - Beachten Sie, dass wir im Setup-Code zunächst die gewünschte Berechnungsfunktion aus dem Namensraum des Hauptprogramms __main__ in den Namensraum des zu testenden Programms importieren müssen

    - Im eigentlich zu analysierenden Code wird nur noch die Berechnung der Fakultät von 50 unter Verwendung der jeweiligen Berechnungsfunktion angestoßen

Python Debugging



- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit

    - Beispiel
      Schlussendlich wird die Laufzeitmessung mit 1.000.000 Wiederholungen gestartet und das jeweilige Ergebnis ausgegeben

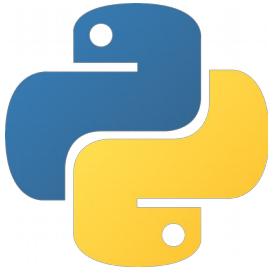
      ```
      print("Iterativ: ", t1.timeit())
      print("Rekursiv: ", t2.timeit())
      ```

      Die Ausgabe des Programms lautet

      ```
      Iterativ:    12.8919649124
      Rekursiv:    28.9529950619
      ```

    - Das bedeutet, dass der iterative Algorithmus etwa doppelt so schnell ist wie der rekursive

Python Debugging



- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit

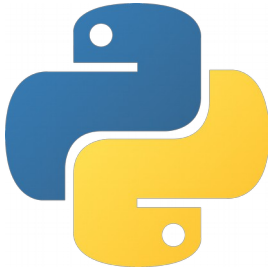
    - Beispiel
    - Doch diese Daten sind noch nicht wirklich repräsentativ, denn es könnte sein, dass der Test der rekursiven Funktion durch einen im System laufenden Prozess ausgebremst wurde

    - Aus diesem Grund starten wir einen erneuten Test

      ```
      print("Iterativ: ", min(t1.repeat(100, 10000)))
      print("Rekursiv: ", min(t2.repeat(100, 10000)))
      ```

    - Dieses Mal führen wir eine Testreihe durch, die einen Test mit 10.000 Einzelwiederholungen 100-mal wiederholt und das kleinste der Ergebnisse ausgibt

Python Debugging



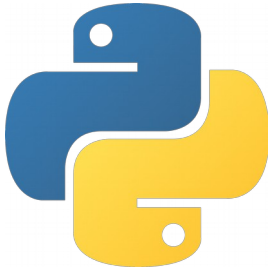
- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - Beispiel
    - Die Ergebnisse sind annäherungsweise deckungsgleich mit denen der vorherigen Tests

      Iterativ: 0.162111997604
      Rekursiv: 0.272562026978

    - Die absoluten Zahlenwerte hängen stark vom verwendeten System ab

    - Auf einem schnelleren Computer sind sie dementsprechend kleiner

Python Debugging



- Das Debugging und die Aufgabe
  - Analyse des Laufzeitverhaltens
  - Laufzeitmessung – timeit
    - Das soll's soweit gewesen sein, in Verbindung mit Optimierung dieses Kapitels
    - Eine eigene Thematik kommt später.
    - cProfile und profile muss selbst erarbeitet werden, dazu gehört auch das Modul 'trace'