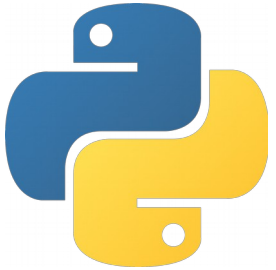
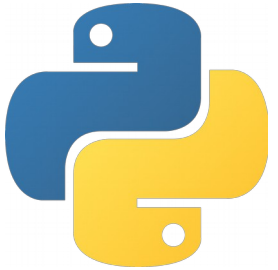
Python Debugging



- Das Debugging und die Aufgabe
  - Das Debugging bezeichnet das Aufspüren und Beseitigen von Fehlern, sogenannten Bugs, in einem Programm
  - Üblicherweise steht dem Programmierer dabei ein Debugger zur Verfügung
  - Das ist ein wichtiges Entwicklerwerkzeug, das es ermöglicht, den Ablauf eines Programms zu überwachen und an bestimmten Stellen anzuhalten
  - Wenn der Programmablauf in einem Debugger angehalten wurde, kann der momentane Programmstatus genau analysiert werden
  - Auf diese Weise können Fehler sehr viel schneller gefunden werden als durch bloßes gedankliches Durchgehen des Quellcodes oder die Analyse von Programmausgaben

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger

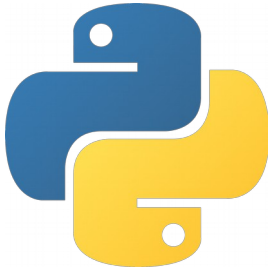
    - Im Lieferumfang von Python ist ein Programm zum Debuggen von Python-Code enthalten, der sogenannte PDB (Python Debugger)

    - Dieser Debugger läuft in einem Konsolenfenster und ist damit weder übersichtlich noch intuitiv

    - Viele moderne Entwicklungsumgebungen für Python beinhalten einen umfangreichen, integrierten Debugger mit grafischer Benutzeroberfläche, der die Fehlersuche in einem Python-Programm recht komfortabel gestaltet

    - Auch IDLE bietet einen rudimentären grafischen Debugger

Python Debugging



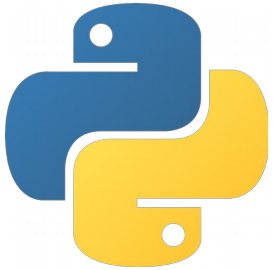
- Das Debugging und die Aufgabe
  - Der Debugger

    - Um den Debugger in IDLE zu aktivieren, klicken Sie in der Python-Shell auf den Menüpunkt Debug • Debugger und führen dann das auf Fehler zu untersuchende Programm ganz normal per Run • Run Module aus

    - Es erscheint zusätzlich zum Editorfenster ein Fenster, in dem die aktuell ausgeführte Codezeile steht

    - Durch einen Doppelklick auf diese Zeile wird sie im Programmcode hervorgehoben, sodass Sie stets wissen, wo genau Sie sich im Programmablauf befinden

Python Debugging

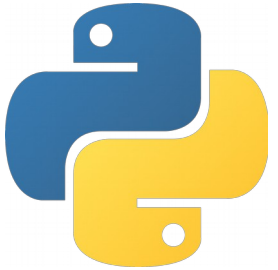


- Das Debugging und die Aufgabe
  - Der Debugger

    - Da es abgesehen von IDLE noch eine Menge weitere Python-IDEs gibt und IDLE bei Weitem nicht das Nonplusultra ist, wäre es müßig, an dieser Stelle eine detaillierte Einführung in den grafischen Debugger von IDLE zu geben

    - Allerdings ähneln sich die Funktionen der diversen grafischen Debugger sehr stark, sodass wir allgemein darauf eingehen möchten, welche Funktionen ein grafischer Debugger in der Regel anbietet

Python Debugging



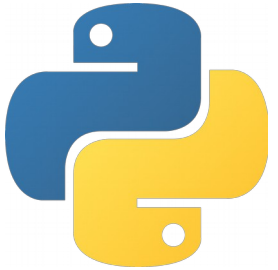
- Das Debugging und die Aufgabe
  - Der Debugger

    - Wenn das Programm angehalten wurde und der Programmfluss somit an einer bestimmten Zeile im Quellcode steht, hat der Programmierer mehrere Möglichkeiten, den weiteren Programmlauf zu steuern

    - Hierbei handelt es sich dabei um die essentiellen Fähigkeiten eines Debuggers

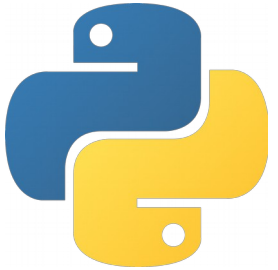
    - Diese Fähigkeiten sind in nahezu allen IDEs ähnlich

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
    - → Mit dem Befehl Step over veranlassen Sie den Debugger dazu, zur nächsten Quellcodezeile zu springen und dort erneut zu halten
    - → Der Befehl Step into verhält sich ähnlich wie Step over, mit dem Unterschied, dass bei Step into auch in Funktions- oder Methodenaufrufe hineingesprungen wird, während diese bei Step over übergangen werden
    - → Der Befehl Step out springt aus der momentanen Unterfunktion heraus wieder dorthin, wo die Funktion aufgerufen wurde. Step out kann damit gewissermaßen als Umkehrfunktion zu Step into gesehen werden

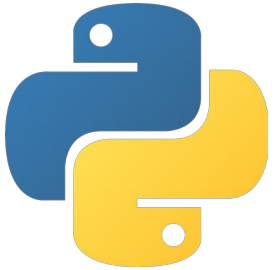
Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger

    - → Der Befehl Run führt das Programm weiter aus, bis der Programmfluss auf den nächsten Breakpoint stößt oder das Programmende eintritt. Einige Debugger erlauben es mit einem ähnlichen Befehl, zu einer bestimmten Quellcodezeile zu springen oder den Programmcode bis zur Cursorposition auszuführen

Python Debugging



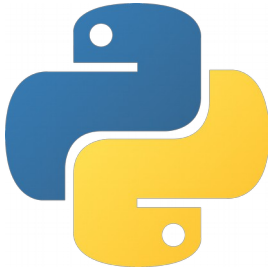
- Das Debugging und die Aufgabe
  - Der Debugger

    - Neben diesen Befehlen, mit denen sich der Programmlauf steuern lässt, stellt ein Debugger einige Hilfsmittel bereit, mit deren Hilfe der Programmierer den Zustand des angehaltenen Programms vollständig erfassen kann

    - Welche dieser Werkzeuge vorhanden sind und wie sie bezeichnet werden, ist von Debugger zu Debugger verschieden

    - Es folgt eine Übersicht über die gebräuchlichsten Hilfsmittel

Python Debugging

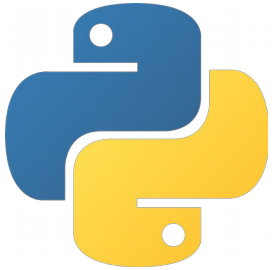


- Das Debugging und die Aufgabe
  - Der Debugger

    - → Das grundlegendste Hilfsmittel ist eine Liste aller lokalen und globalen Referenzen mitsamt referenzierter Instanz, die im momentanen Programmkontext existieren. Auf diese Weise lassen sich Wertänderungen verfolgen und Fehler, die dabei entstehen, leichter aufspüren

    - → Zusätzlich zu den lokalen und globalen Referenzen ist der sogenannte Stack von Interesse. In diesem wird die momentane Funktionshierarchie aufgelistet, sodass sich genau verfolgen lässt, welche Funktion welche Unterfunktion aufgerufen hat

Python Debugging

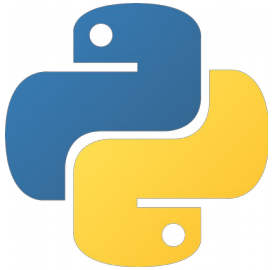


- Das Debugging und die Aufgabe
  - Der Debugger

    - → Gerade in Bezug auf die Programmiersprache Python bieten einige Debugger eine interaktive Shell, die sich im Kontext des angehaltenen Programms befindet und es dem Programmierer erlaubt, komfortabel Referenzen zu verändern, um somit in den Programmfluss einzugreifen

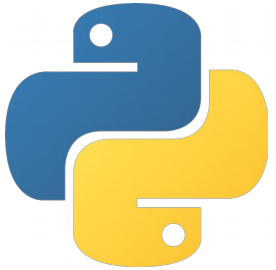
    - → Ein sogenannter Post-Mortem-Debugger kann in Anlehnung an den vorherigen Punkt betrachtet werden. In einem solchen Modus hält der Debugger das Programm erst an, wenn eine nicht abgefangene Exception aufgetreten ist. Im angehaltenen Zustand verfügt der Programmierer wieder über eine Shell sowie über die genannten Hilfsmittel, um dem Fehler auf die Spur zu kommen. Diese Form des Debuggens wird »post mortem« genannt, da sie erst nach dem Auftreten des tatsächlichen Fehlers, also nach dem »Tod« des Programms, aktiviert wird

Python Debugging



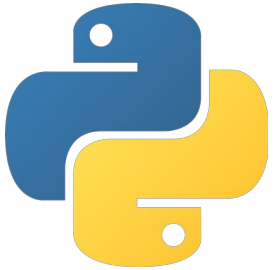
- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
    - Das Modul inspect stellt Funktionen bereit, über die der Programmierer detaillierte Informationen über eine Instanz erlangen kann
    - So könnten Sie beispielsweise den Inhalt einer Klasse oder die Parameterliste einer Funktion ermitteln
    - Damit eignet sich inspect besonders zum Erstellen von detaillierten Debug-Ausgaben

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
    - Grundsätzlich lässt sich die Funktionalität von inspect in drei Teilbereiche gliedern
      - Funktionen, die sich auf Datentypen, Attribute und Methoden einer Instanz beziehen
      - Funktionen, die sich auf ein Stück des Quellcodes beziehen, das im Zusammenhang mit einer Instanz steht
      - Funktionen, die sich auf Klassen- und Funktionsobjekte beziehen

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect

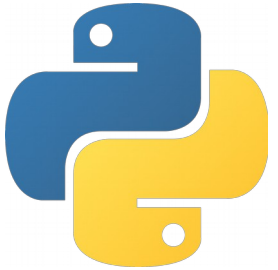
    - Als Grundlage der einzelnen Funktionen, nehmen wir folgende Klasse als Beispiel

```
class klasse:

  def __init__(self):
    self.a = 1
    self.b = 2
    self.c = 3

  def hallo(self):
    return "welt"
```

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Datentypen, Attribute und Methoden

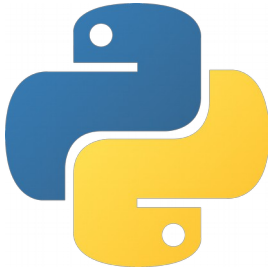
    - getmembers(object[, predicate])

      - Diese Funktion gibt alle Attribute und Methoden, auch Member genannt, der Instanz object in Form einer Liste von Tupeln zurück

      - Jedes Tupel enthält dabei den Namen des jeweiligen Members als erstes Element und den Wert des Members als zweites Element

      - Im Falle einer Methode entspricht das Funktionsobjekt dem Wert des Members. Die zurückgegebene Liste ist nach Member-Namen sortiert

Python Debugging

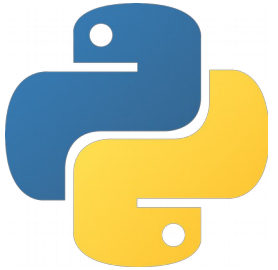


- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Datentypen, Attribute und Methoden

    - getmembers(object[, predicate])

```
>>> inspect.getmembers(klasse())
[('__class__', <class '__main__.klasse'>), [...]
        ('a', 1), ('b', 2), ('c', 3), ('hallo', <bound method
        klasse.hallo of <__main__.klasse object at
                                        0xb7a9d08c>>)]
```

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Datentypen, Attribute und Methoden

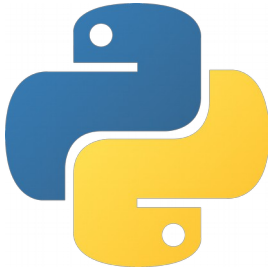
    - getmembers(object[, predicate])

      - Für den optionalen Parameter predicate kann eine Filterfunktion übergeben werden

      - Diese Funktion wird für jedes Member der Instanz object aufgerufen und bekommt diesen als einzigen Parameter übergeben

      - Es werden alle Member in die Ergebnisliste aufgenommen, für die die Filterfunktion den Wert True zurückgegeben hat

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Datentypen, Attribute und Methoden

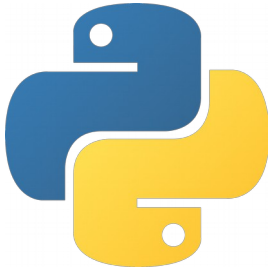
    - getmembers(object[, predicate])

      ```
      >>> inspect.getmembers(klasse(), inspect.ismethod)
      [('__init__', <bound method klasse.__init__ of
      <__main__.klasse object at 0xb7a9df2c>>), ('hallo',
      <bound method klasse.hallo of <__main__.klasse
      object at 0xb7a9df2c>>)]
      ```

      - In diesem Fall wurde die Funktion ismethod übergeben, die genau dann True zurück gibt, wenn es sich bei dem Member um eine Methode handelt

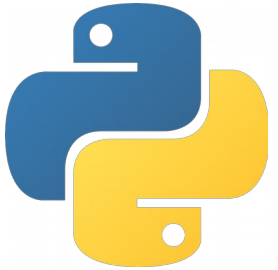
      - Dementsprechend klein ist die Ergebnisliste

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Quellcode
    - getfile(object)
      - Die Funktion getfile gibt den Namen der Datei zurück, in der das Objekt object definiert wurde
      - Dabei kann es sich sowohl um eine Quellcode-Datei als auch um eine Bytecode-Datei handeln
      - Diese Funktion wirft eine TypeError-Exception, wenn es sich bei object um ein eingebautes Objekt, beispielsweise eine Built-in Function, handelt

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Quellcode

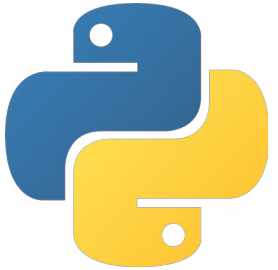
    - getfile(object)

      - Das liegt daran, dass Built-in Functions intern in C implementiert sind und somit keiner Quelldatei zugeordnet werden können

        >>> inspect.getfile(inspect.getfile)
        'C:\\Python32\\lib\\inspect.py'

      - In diesem Beispiel wurde die Funktion getfile verwendet, um herauszufinden, in welcher Quelldatei sie selbst definiert ist

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Quellcode

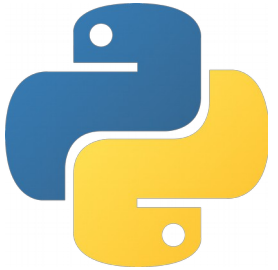
    - getsourcefile(object)

      - Diese Funktion gibt den Namen der Quellcode-Datei zurück, in der das Objekt object definiert wurde

      - Diese Funktion wirft eine TypeError-Exception, wenn es sich bei object um ein eingebautes Objekt, beispielsweise eine Built-in Function, handelt

      - Das liegt daran, dass diese Objekte entweder intern in C implementiert sind oder die Quelldatei nur als Kompilat vorliegt

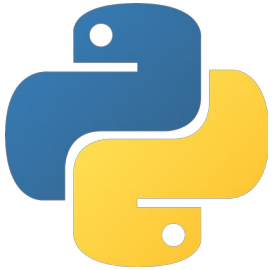
Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Quellcode
    - getsourcefile(object)
      - Dem Objekt kann also kein tatsächlicher Quellcode zugeordnet werden
      - Diese Einschränkung betrifft auch viele Funktionen der Standardbibliothek

        >>> inspect.getsourcefile(inspect.getsourcefile)
        'C:\\Python32\\lib\\inspect.py'
      - In diesem Fall wurde getsourcefile dazu verwendet, die Quellcodedatei herauszufinden, in der die Funktion selbst definiert ist

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Quellcode

    - getsourcelines(object)

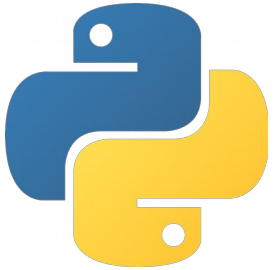
      - Diese Funktion gibt ein Tupel mit zwei Elementen zurück

      - Das erste ist eine Liste von Strings, die alle dem Objekt object zugeordneten Quellcodezeilen enthält

      - Das zweite Element des zurückgegebenen Tupels ist die Zeilennummer der ersten dem Objekt object zugeordneten Quellcodezeile

      - Für object kann ein Modul, eine Methode, eine Funktion, ein Traceback-Objekt oder ein Frame-Objekt übergeben werden

Python Debugging

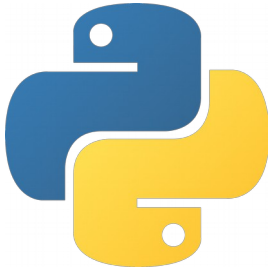


- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Quellcode
    - getsourcelines(object)
      - Die Funktion wirft eine IOError-Exception, wenn der Quellcode zum Objekt object nicht geladen werden konnte

        >>> inspect.getsourcelines(inspect.getsourcelines)
        (['def getsourcelines(object):\n', [...]
        '   lines, lnum = findsource(object)\n', '\n',
        '   if ismodule(object): return lines, 0\n',
        ' else: return getblock(lines[lnum:]), lnum + 1\n'], 678)

      - die Quellcodezeilen ihrer eigenen Definition

Python Debugging

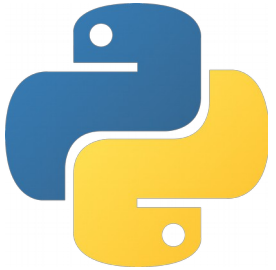


- Das Debugging und die Aufgabe
  - Der Debugger
  - Inspizieren von Instanzen – inspect
  - Klassen und Funktionen

    - Aufgabe
      Verwenden Sie das Modul 'inspect' und ermitteln Sie die Funktionsweisen auf Klassen und Funktionen

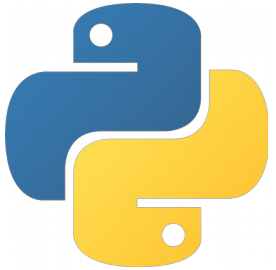
    - Besprechen Sie die Ergebnisse und Erkenntnisse mit den Kollegen, prüfen Sie die Gleichheit

Python Debugging



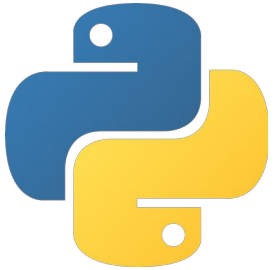
- Das Debugging und die Aufgabe
  - Der Debugger
  - Formatierte Ausgabe von Instanzen – pprint
    - pprint(object[, stream[, indent[, width[, depth]]]])
      - Die Funktion pprint gibt die Instanz object, formatiert auf dem Stream stream, aus
      - Wenn Sie den Parameter stream nicht übergeben, wird nach sys.stdout geschrieben
      - Über die Parameter indent, width und depth lässt sich die Formatierung der Ausgabe steuern
      - Dabei kann für indent die Anzahl der Leerzeichen übergeben werden, die für eine Einrückung verwendet werden soll

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Formatierte Ausgabe von Instanzen – pprint
    - pprint(object[, stream[, indent[, width[, depth]]]])
      - Der Parameter indent ist mit 1 vorbelegt
      - Über den optionalen Parameter width kann die maximale Anzahl an Zeichen angegeben werden, die die Ausgabe breit sein darf
      - Dieser Parameter ist mit 80 Zeichen vorbelegt
      - Der Parameter depth ist ebenfalls eine ganze Zahl und bestimmt, bis zu welcher Tiefe Unterinstanzen, beispielsweise also verschachtelte Listen, ausgegeben werden sollen

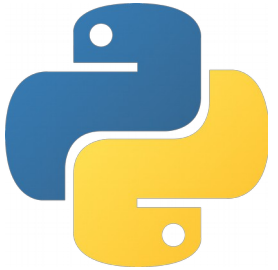
Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Formatierte Ausgabe von Instanzen – pprint
    - pprint(object[, stream[, indent[, width[, depth]]]])
      - Die Funktion pprint wird dazu verwendet, die Liste sys.path formatiert und damit lesbar auszugeben

```
>>> import sys
>>> pprint.pprint(sys.path, indent=4)
[    '',
     'C:\\WINDOWS\\system32\\python32.zip',
     'C:\\Python32\\DLLs',
     'C:\\Python32\\lib',
     'C:\\Python32',
     'C:\\Python32\\lib\\site-packages']
```

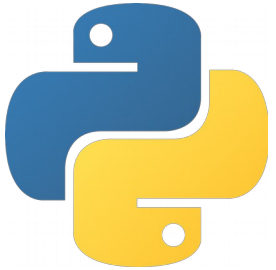
Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Formatierte Ausgabe von Instanzen – pprint
    - pprint(object[, stream[, indent[, width[, depth]]]])
      - Zum Vergleich geben wir sys.path noch einmal unformatiert mit print aus

```
>>> print(sys.path)
['', 'C:\\WINDOWS\\system32\\python32.zip',
'C:\\Python32\\DLLs', 'C:\\Python32\\lib', 'C:\\Python32',
                                    'C:\\Python32\\lib\\site-packages']
```

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Formatierte Ausgabe von Instanzen – pprint

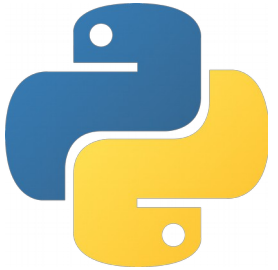
    - isreadable(object)

      - Diese Funktion gibt True zurück, wenn die formatierte Ausgabe der Instanz object lesbar ist

      - Lesbar bedeutet in diesem Zusammenhang, dass die Ausgabe als Python-Code gelesen werden kann und mithilfe der Built-in Function eval aus der Ausgabe wieder die zugrundeliegende Python-Instanz rekonstruiert werden könnte

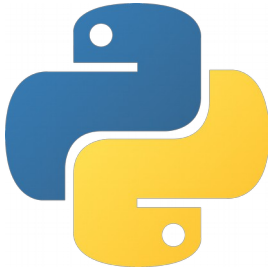
      - Wenn object eine Instanz eines Basisdatentyps ist, gibt die Funktion in den meisten Fällen den Wert True zurück

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Formatierte Ausgabe von Instanzen – pprint
    - isreadable(object)
      - Im zweiten Teil des Beispiels wurde False zurückgegeben, weil es sich um eine rekursive Liste handelt, die sich selbst als Element enthält
      - Eine solche Liste kann aus naheliegenden Gründen nicht vollständig ausgegeben werden
      - Auch Instanzen selbst erstellter Klassen können nicht lesbar repräsentiert werden
        → Letzteres liegt daran, dass die Ausgabe dann den Code zum Erstellen der Klasse sowie der Instanz enthalten müsste, was keinen Sinn ergibt

Python Debugging



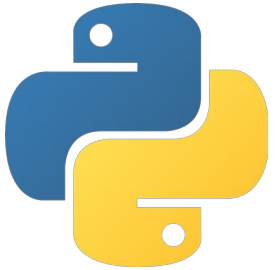
- Das Debugging und die Aufgabe
  - Der Debugger
  - Formatierte Ausgabe von Instanzen – pprint

    - isrecursive(object)

      - Diese Funktion gibt True zurück, wenn die Instanz object rekursiv ist, also sich selbst als Element enthält

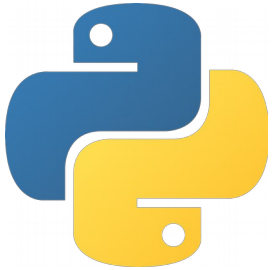
        ```
        >>> a = []
        >>> a.append(a)
        >>> pprint.isrecursive(a)
        True
        ```

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Logdateien – logging
    - Das Modul logging stellt ein flexibles Interface zum Protokollieren eines Programmlaufs bereit
    - Protokolliert wird der Programmablauf, indem an unterschiedlichen Stellen im Programm Meldungen an das logging-Modul abgesetzt werden
    - Diese Meldungen können unterschiedliche Dringlichkeitsstufen haben
    - So gibt es beispielsweise Fehlermeldungen, Warnungen oder Informationen

Python Debugging



- Das Debugging und die Aufgabe
  - Der Debugger
  - Logdateien – logging

    - Das Modul logging kann diese Meldungen auf vielfältige Weise verarbeiten

    - Üblich ist es, die Meldung mit einem Zeitstempel zu versehen und entweder auf dem Bildschirm auszugeben oder in eine Datei zu schreiben

      import logging

    - Aufgabe
      Setzen Sie sich mit dem Modul auseinander, erweitern Sie Ihr Projekt um das Protokollieren von Daten