Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Debugging(pdb und Integration mit Gdb)

vorgelegt von

Yuliia Lysa

Studiengang:	Informatik
Matrikelnummer:	7160066

Hamburg, 2019-09-03

Inhaltsverzechnis

1 Einleitung

Das Problem, das bei umfangreichen komplexen Softwaresystemen auftritt, ist, dass sie nicht dafür geeignet sind, manuell nach Fehlern durch Code-Reviews analysiert zu werden, deswegen stellen die Debugger oft notwendige Maßnahme dar, um viele Codezeilen zu durchsuchen. Ein Debugger stellt dementsprechend ein wichtiges Werkzeug in der Softwareentwicklung dar, um Fehler in den Computerprogrammen systematisch aufzufinden, zu diagnostizieren und zu beheben. In den folgenden Abschnitten werden zunächst die allgemeine Funktionsweise von einem Debugger und danach die Python-Debugger pdb und gdb und ihre Besonderheiten vorgestellt.

1.1 Allgemeines

Für die Fehlersuche bieten die modernen Debugger normalerweise folgende Funktionalitäten an:

- das Inspizieren von Daten im Quellcode, wobei die Inhalte von Variablen und Speichers zu einem bestimmten Zeitpunkt überprüft werden;
- das Steuern des Programmablaufs durch das Setzen von Haltepunkten und schrittweisen Ablauf des Codes;
- die Modifizierung vom Programmcode im Speicher;

Teilweise ist das Abfangen von Exceptions mit Debuggern erlaubt, um den genauen Standort und die Art von Exception automatisch zu ermitteln. Eine weitere nützliche Funktion von bestimmten Debuggern besteht darin, die Werte der Variablen vorzutäuschen, damit das Verhalten des Programms im veränderten Zustand beobachtet werden kann, ohne die permanente Modifikation des Quellcodes.

Für die interne Implementation von Unix-Debuggern wird üblicherweise der Systemaufruf ptrace benutzt. Er ermöglicht einem Prozess einen anderen vollständig zu kontrollieren und damit seinen internen Zustand zu manipulieren. Das geschieht durch das Anhängen an den fremden Prozess, wobei durch den Zugang zu dem Speicher und Registern der Quellcode von dem Zielprogramm durchgelaufen werden kann und der Speicher mit neuen Inhalten beschrieben.

2 pdb

pdb ist die naheliegende Lösung für die Fehlersuche in der Computersprache Python, insbesondere weil dieser Debugger in Python-Standardbibliothek integriert ist. pdb hat die Form eines Moduls und ist durch das Kommandozeile-Interface bedienbar.

In Python ist pdb als eine Klasse realisiert und ist dementsprechend wie jedes andere Modul durch den Benutzer erweiterbar. Beim Aufruf wird folgender Klassenkonstruktor automatisch benutzt:

- class pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False, readrc=True)

Die vorgegebenen Parameter können direkt spezifiziert werden, um eigene Veränderungen vorzunehmen. Das Extencion-Interface wird durch Module bdb und cmd realisiert, die beiden sind ebenfalls ein Bestandteil der Standardbibliothek von Python.

Die Definition von Parametern completekey, stdin und stdout werden dem Modul cmd als Parameter übergeben. Cmd bildet ein weiteres Framework, zuständig für die Erstellung von Befehle-Interpretern mit Kommandozeile-Interface. Es wird dementsprechend nie alleine als Objekt erzeugt, sondern lediglich für die Implementation von anderen Klassen benutzt. Mit dem Parameter completekey wird angegeben, welche Methode für die Befehl-Vervollständigung benutzt wird, und bezieht sich in diesem Fall auf readline.

Und Readline ist dementsprechend ein weiteres Modul, das vom pdb-Debugger benutzt wird, um die Vervollständigung von Wörtern durch die Tastaturtasse Tab zu ermöglichen. Stdin und stdout geben den Input und den Output für die benutzten Files vor.

Bdb ist ein weiter Debugger-Framework, also eine weitere Klasse, die grundlegenden Debugger-Methoden definiert und beschäftigt sich mit den typischen Debugging-Aspekten, wie die Haltepunktesetzung oder die Abarbeitung des Codes.

2.1 Befehlssyntax

pdb gibt einen bestimmten Befehlssyntax vor, der beachtet werden muss und der bei dem Verständnis von seinen Anweisungen hilft. pdb erlaubt vordefinierte Abkürzungen, die Anweisungen dürfen in den meisten Fällen nicht komplett ausgeschrieben werden, der Debugger betrachtet die Formulierungen „h“ und „help“ als gleichwertige Befehle. Die Argumente für die Befehle müssen mit einem Whitespace von der Anweisung abgetrennt werden. Wenn die nächste Zeile leer gelassen wird, wird der Befehl davor ausgeführt, damit wird die mehrfache Wiederholung ermöglicht. Die eckigen Klammern stehen in der Dokumentation für optionale Parameter und der senkrechte Strich dient

dazu, die alternativen Angaben zu kennzeichnen, diese beiden Zeichen sollten nicht in die Kommandozeile getippt werden, sie dienen nur dem besseren Verständnis von der Debugger-Dokumentation. Die Befehle, die nicht vordefiniert sind und damit vom Debugger nicht erkannt werden können, werden als Statements oder Anweisungen betrachtet. Eigene Statements stehen für die Deklaration zur Verfügung mit einem Ausrufezeichen davor. Hilfreich ist dieser Ausdruck zur Unterscheidung bei der Variablendeklaration, wenn die Variable den gleichen Namen wie der vordefinierte Befehl trägt. Wenn das Exception in einem Statement auftaucht, wird der Name (also die Art) von dieser Exception automatisch durch den Debugger auf die Konsole gedruckt. Mehrere Befehle dürfen in einer Zeile hintereinander geschrieben werden, wenn sie zwischendurch mit doppelten Semikola getrennt werden.

2.2 Aufruf des Debuggers

Pdb-Debugger lässt sich auf verschiedenen Weisen aufrufen. Zunächst an der Stelle im Code, wo das Programm angehalten werden muss, kann unten stehender Befehl eingegeben werden oder seit Python 3.7 die Methode breakpoint benutzt. Beide Befehle rufen den Debugger auf, wobei bei dem zweiten Befehl der Import des Moduls pdb automatisch geschieht.

```
1 import pdb; pdb.set_trace ()
2 breakpoint()
```

Quelle: [2]

Die folgende alternative Methode hat den Vorteil, dass der Quellcode unverändert bleibt. Der Debugger lässt sich direkt aus der Kommandozeile aufrufen mit dem Befehl:

```
1 python3 -m pdb myscript.py
```

Quelle: [2]

Pdb erlaubt auch das sogenannte Post-Mortem-Debugging, in diesem Modus wird der Debugger automatisch gestartet, nachdem eine Exception im Programm geworfen wurde. Dieser Modus wird außerdem aufgerufen, falls das Programm als Skript aus der Kommandozeile ausgeführt wird. Um das Post-Mortem-Debugging manuell zu starten, existieren zwei Methoden:

```
1 pdb.post_mortem()
2 pdb.pm()
```

Quelle: [2]

Um das Debugger-Modus zu verlassen, bietet sich der Befehl **quit** an.

2.3 Wichtige Befehle

pdb unterstützt alle grundlegenden Befehle, die bei der Fehlersuche und Diagnose helfen sollen.

Um die **Haltepunkte zu setzen**, also die Stellen, wo der Programmfluss unterbrochen wird und der Debugger anhält, kann entweder der Befehl **break** (unter anderem können hier auch den Methodennamen oder eine Programmzeile angegeben werden) angewandt werden oder wenn eine bestimmte gewählte Bedingung wahr ist. Der Syntax sieht in diesem Fall folgendermaßen aus, falls der filename unspezifiziert bleibt, dann wird der aktuelle Quellcode aus dem benutzten File verwendet, der zweite Parameter bleibt auch optional. Hier kann dementsprechend ein boolescher Ausdruck angegeben werden, und durch die Auswertung wird die Ausführung von break fortgesetzt, nur wenn die Bedingung zutrifft. Der Befehl **continue** setzt die Ausführung des Programms fort, bis ein gültiger Haltepunkt gefunden wird. Wenn der Befehl break ohne jegliche Argumente aufgerufen wird, wird eine Liste von den bereits gesetzten Haltepunkten angezeigt. Die gesetzten Haltepunkte können vorübergehend aktiviert oder deaktiviert werden durch die Befehle **disable** und **enable**. Jedem Haltepunkt wird durch den Debugger einer Zahl zugeordnet, diese Identifikationsnummer darf benutzt werden, um die Haltepunkte, die aktiviert werden müssen, nach dem Befehl anzugeben. Um sie endgültig zu löschen, kann der Befehl **clear** genutzt werden. Auch zeitweilige Haltepunkte sind möglich mit der Nutzung vom Befehl **ctbreak**, solche Haltepunkte werden schon nach dem ersten Aufruf des Programms wieder gelöscht.

```
1 (Pdb) b util:5
2 Breakpoint 1 at /code/util.py:5
3 (Pdb) disable 1
4 Disabled  breakpoint 1 at /code/util.py:1
```

Quelle: [3]

Eine weitere Funktion von jedem Debugger besteht aus Reihe der Befehlen, die dazu dienen, den **Wert von Variablen ausgeben zu lassen**:

- Mit dem Hauptbefehl **p** (steht für print) wird der Ausdruck angegeben, dessen Wert ausgegeben werden soll, also typischerweise der Variablenname. Der aktuelle Wert der vorgegebenen Variable wird dabei auf die Konsole gedruckt, es ist auch erlaubt einen Python-Ausdruck bzw. eine Anweisung anzugeben, damit p das Ergebnis auswertet und ausgibt.

- **pp** wird dazu benutzt, um auf die Ausdrücke die Funktion pretty print anzuwenden. Der Befehl dient dazu, wenn mit den Variablen mit umfangreicher Datenausgabe (wie bei manchen Listen) gearbeitet wird, und sie verständlicher für den Nutzer gestalten werden müssen. Er versucht das Ergebnis dem Fenster anzupassen und druckt die Objekte auf einer Zeile, wenn es möglich ist oder teilt sie auf mehrere Zeile auf, besitzt außerdem die Möglichkeit die Objekte visuell voneinander zum Beispiel durch Farbe zu trennen.

- Der Befehl **display** stellt eine Alternative zu print dar, wie die Ausdrücke angezeigt werden können. Mit display wird automatisch der Variablenwert angezeigt nur in dem Falle, wenn er sich verändert hat, nachdem die Ausführung des Programms zu Ende ist. Den Befehl kann man auch ohne einen bestimmten Ausdruck benutzten, allerdings werden dann nur die veränderten Werte im aktuellen Frame angezeigt. **undisplay** dient dementsprechend dem Zweck, den angegebenen Ausdruck im aktuellen Frame nicht mehr zu zeigen und ohne den genauen Ausdruck werden alle davor benutzten Display-Ausdrücke gelöscht.

```
1 (Pdb) p filename
2 './ example2.py'
3 (Pdb) p head , tail
4 ( '.','example2.py')
```

Quelle: [3]

Um **den Code zu durchlaufen**, gibt es folgende Befehle:

- Durch den Befehl **next** wird die Ausführung des Programms bis zur nächsten Codezeile durchgeführt und bleibt stets innerhalb der aktuellen Methode. Wenn eine fremde Methode innerhalb davon aufgerufen wird, wird sie lediglich übersprungen.

- Mit dem Befehl **step** wird die aktuelle Zeile ausgeführt, falls eine fremde Methode aufgerufen wird, wird erst da das Programm angehalten. Falls das geschieht, wird auf der Konsole das Wort Call erscheinen. Beide Befehle halten an, wenn die aktuelle Methode zu Ende ist und geben Return zurück mit dem Ergebnis der Auswertung der aktuellen Methode in der nächsten Zeile.

- Es gibt noch einen dritten ähnlichen Befehl, **until**, der in ähnlicher Weise wie continue funktioniert, also die Ausführung des Programms fortsetzt, allerdings sich je nach Parametern auf zwei verschiedene Arten verhalten kann. Falls keine konkrete Zeilenangabe vorliegt, wird ähnlich wie beim Befehl next bis der Zeile mit der nächstgrößten Nummer von der aktuellen Zeile aus ausgeführt. Mit der Zeilenangabe wird dementsprechend bei der Zeile, die größer als die angegebe nicht mehr fortgesetzt. Until hört mit der Programmausführung auf in beiden Fällen, wenn der aktuelle Frame zurückgegeben wird, genauso wie die Befehle next und step.

- Die Anweisung **longlist** ist hilfreich, wenn ein unbekannter Code abgearbeitet oder der Kontext bzw. die Arebeitweise einer Methode erschlossen werden muss. Um einen kürzeren Auszug vom Code zu bekommen, wird der Befehl **list** benutzt. Wenn keine Argumente hinzugefügt werden, wird er 11 Zeilen um die aktuelle Zeile ausdrucken, also fünf davor und fünf danach. Um stets den Code um die aktuelle Zeile anzeigen zu lassen, braucht man als Argument einen Punkt nach dem Befehl zu setzen.

```
(Pdb) n
> /code/example3.py(15)<module  >()
-> print(f'path = {filename_path}')
(Pdb) s
--Call  --6 > /code/example3.py(6) get_path ()
-> def  get_path(filename):'
```

Quelle: [3]

Folgende Befehle finden Verwendung, um **den Stacktrace anzeigen zu lassen**:

- Mit dem Befehl **where** wird der Stacktrace angezeigt (der eine geordnete Liste von allen Frames an einem bestimmten Zeitpunkt darstellt, der aktuellste Frame wird ganz unten angezeigt). Ein Frame (also eine Datenstruktur) wird erzeugt, sobald eine Methode aufgerufen wird und wird wieder gelöscht, wenn die Rückkehr aus der Methode erfolgt. Der Pfeil dient als Stapelzeiger beim Aufrufstapel und damit ist der aktuelle Frame stets eindeutig identifizierbar.
- Die beiden Befehle **up** und **down** werden benutzt, um den aktuellen Frame jeweils nach oben oder unten zu verlegen. In Kombination mit p können alle Variablen und der Zustand des Speichers in der Anwendung an jedem Zeitpunkt innerhalb des Aufrufestapels angesehen werden. Der Stapelzeiger wird eine Stufel nach unten oder nach oben in Stacktrace verschoben.

```
(Pdb) w
/code/example5.py(7)  get_file_info ()
-> file_path = fileutil.get_path(full_fname)
> /code/fileutil.py(5) get_path ()
```

Quelle: [3]

Der Debugger unterstützt **Alias**, es können eigene Shortcuts definiert werden, um komplexe Anweisungen nicht immer wieder angeben zu müssen. Alias übernimmt den Namen des Befehls durch das erste angegebene Wort. Der Rumpf vom Alias kann aus allen für den Debugger definierten Befehlen oder gewöhnlichen Python-Ausdrücken bestehen, demnach aus allem, was im Debugger-Prompt akzeptiert werden kann. Alias erlaubt strukturelle Rekursion, es können verschiedene Alias voneinander aus aufgerufen werden und in der Definition enthalten sein. Weiteren Nutzen bringt die Möglichkeit von pdb, schon bestehende Befehle zu überschreiben und neu zu definieren. Wenn der Befehl **alias** ohne Argumente übergeben wird, wird die Liste mit allen von Nutzer davor definierten Alias angezeigt. Ein einziger Argument wird als der Name von der Abkürzung interpretiert und dessen Definition wird dementsprechend auf die Konsole gedruckt. Und um Definition von einem Alias zu löschen, ist der Befehl **unalias** definiert.

```
(Pdb) alias  pl pp  locals ().keys()
(Pdb) pl
```

```
3 dict_keys (['output','n'])
4 (Pdb) alias
5 pl = pp  locals ().keys()
```

Quelle: [4]

3 Gdb

Gdb ist ein weiterer Debugger, der für die Fehlersuche in Python verwendet werden kann, der aber nicht Python-spezifisch ist. Gdb ist ein UNIX-Programm, unter anderem GNU-Debugger genannt und zeichnet sich dadurch unter anderen Debuggern aus, dass er das Programmverhalten während der Laufzeit verändern kann durch das Modifizieren von Speichern. Er besitzt keine grafische Benutzerschnittstelle, stattdessen bietet er ein Kommandozeile-Interface. Insbesondere wird er für das Debugging von C und C++ Anwendungen verwendet, unterstützt aber viele verschiedene Programmiersprachen wie Java oder Python. Im Gegensatz zu anderen Standardsprachen wie C++ wird der Quellcode in Python nicht für die jeweilige Plattform compiliert, sondern ein Interpreter benutzt, der den Bytecode ausführt. Das heißt, dass das Debugging mit Gdb auf der Interpreterebene im Gegensatz zu Compilerebene erfolgt. Außerdem ist es möglich für diesen Debugger in der Sprache Python Erweiterungen zu entwickeln.

3.1 Vorteile des Debuggers

Die Implementierung von pdb lehnt sich stark an Gdb an, sodass die oben erwähnte Klassendefinition auch auf Gdb anwendbar ist, somit besitzt Gdb unter anderem dieselben Befehle und Befehlssyntax. Gdb bietet einige Vorteile, die ihn gegenüber pdb auszeichnen und findet am besten Verwendung in den Fällen, wo die Bugs auftreten, die man innerhalb von Python selbst eher schlecht debuggen kann:

- Durch die Fähigkeit sich an einen laufenden Prozess anzuhängen, kann Gdb bei dem Problem von hängenden Prozessen hilfreich sein. Falls ein Prozess während der Ausführung hängen bleibt, wird er entweder auf neuen Input warten oder in einer Endlosschleife geraten sein, in beiden Fällen würde es helfen, sich an den laufenden Prozess anzuhängen, um den Backtrace zu fordern und die Ursache dafür zu finden.

- Wenn das Programm einen Segfault (Schutzverletzung) enthält, ist sie mit integrierten Debuggern wie pdb schlecht behandelbar, da das Programm den Python-Interpreter eher zum Absturz bringt, bevor es den Debugger-Modus aufrufen kann, in diesem Fall würde es wieder helfen, sich an den laufenden Prozess anzuhängen.

- Gdb kann zur Analyse von einer Core-Datei bei einem Core-Dump verwendet. Core-Dump bezeichnet einen Speicherauszug und wird normalerweise zur Fehlerdiagnose bei einem Programmabsturz benutzt. Unter Linux wird er automatisch erstellt, in

dem der Speicher des abgestürzten Prozesses in eine Datei geschrieben wird und kann danach mit einem Debugger, in diesem Falle Gdb analysiert werden, da er die entsprechende Funktionalitäten dafür anbietet.

3.2 Aufruf des Debuggers

Sowohl Gdb, als auch der Extencion-Packet, der die Symbole und die speziell für Python definierten Befehle enthält, müssen zunächst installiert werden.

Gdb lässt sich auf zwei verschiedene Weisen installieren, die von der jeweiligen Linux-Distribution abhängen, zum Beispiel der Befehl für Ubuntu sieht folgendermaßen aus:

```
1 # apt -get  install  gdb
```

Quelle: [8]

Danach müssten die Debugging-Symbole separat installieren werden mit dem folgenden Befehl:

```
1 # apt -get  install  python -dbg
```

Quelle: [8]

Gdb-Debugger kann auf zwei verschiedene Weisen aufgerufen werden. Mit dem folgenden Befehl lässt sich Python unter Gdb starten:

```
1 $ gdb python
2 ...
3 (gdb) run <programname  >.py <arguments  >
```

Quelle: [6]

Im zweiten Fall wird der Debugger durch das Anhängen an einen laufenden Prozess gestartet, dafür muss die Identifikationsnummer von dem aktuellen Prozess dem Nutzer bekannt sein.

```
1 $ gdb python <pid of running process >
```

Quelle: [6]

3.3 Wichtige Befehle

Die Befehle von Gdb funktionieren in ähnlicher Weise wie bei pdb und die Standardfunktionen bleiben somit erhalten, die Haltepunkte können demnach mit einem break-Befehl gesetzt werden, der Code schrittweise mit next und step abgearbeitet werden und so weiter. Außerdem sind die bei pdb schon bestehenden Abkürzungen erlaubt, es existieren aber weitere speziell für Python-Extencion definierte Befehle:

- Der Befehl **py-list** wird benutzt, um herauszufinden, welche Programmzeile zur Zeit ausgeführt wird und den Quellcode (11 Zeilen) für den aktuellen Frame aufzulisten. Die aktuelle Zeile ist durch die Markierung mit einem echt kleiner Zeichen zu erkennen. Es ist unter anderem möglich mit einem list Start bzw. list Start End Befehl genauso wie bei pdb nur einen bestimmten Auszug aus dem Quellcode anzeigen zu lassen.

- Der Befehl **py-bt** versucht, den auf Python-Level gemachten Backtrace anzuzeigen. Backtrace ist eine Zusammenfassung von den Informationen, aus denen erschließbar ist, wie das Programm zu der aktuellen Stelle im Code gekommen ist. Es wird für jeden Frame eine Zeile des Codes angezeigt, der aktuelle besitzt dabei die ID-Nummer Null, der Caller davon (die Codezeile, aus der der Aufruf passiert) die Nummer Eins und die Frames bewegen sich in dieser Reihenfolge weiter nach oben im Aufrufstapel.

- Der Befehl **py-print** sucht im Programm nach dem angegeben Python-Namen und versucht ihn auf der Konsole auszudrucken. Es werden in der vorgegebenen Reihenfolge zunächst die lokalen Variablen innerhalb des aktuellen Threads, dann globalen Variablen und schließlich die Builtins gesucht.

- Beim Befehl **py-locals** dagegen werden nur die Werte von lokalen Variablen innerhalb des aktuellen Frames auf die Konsole ausgegeben bzw. es wird versucht, die Repräsentation davon auszudrucken.

- **Py-up** und **py-down** funktionieren analog zu den Befehlen up und down in pdb, um sich zwischen den Frames zu bewegen. Bei diesen Befehlen handelt es sich allerdings um die Cpython-Frames, die auch durch ID-Nummern gekennzeichnet werden.

4 Alternative Python-Debugger

Es gibt viele unterschiedliche Debugger, die für die Fehlersuche in Python zusätzlich verwenden werden können und die eigenen Vorteile für das Debuggen anbieten. Folgende Debugger stellen Alternativen zu pdb und gdb dar:

- **Ipdb** stellt eine erweiterte Version von pdb dar, die demnach in der iPython-Extencion integriert ist. Ipdb besitzt dieselbe Bedienung wie pdb, dementsprechend verfügt über dieselbe Befehle und Syntax. Allerdings besitzt er weitere Funktionen, die den Debugging-Prozess vereinfachen, für die bessere Lesbarkeit und Gestaltung sorgen, wie zum Beispiel Syntax-Highlighting und Befehlszeilenergänzung, und informativere Stacktrace und Introspektion als gewöhnlicher pdb bieten.

- Um den Nachteil von Debuggern wie pdb zu vermeiden, und zwar dass alle benötigten Befehle auswendig gelernt werden müssen, bietet die integrierte Entwicklungsumgebung **PyCharm** einen eigenen gleichnamigen Debugger mit einer grafischen Oberfläche. In diesem Debugger gibt es unterschiedliche grafische Buttons für jeweilige Befehle, um leichtere Nutzbarkeit zu ermöglichen, es können demzufolge die Haltepunkte mit einem Mausklick neben der Zeilennummer gesetzt oder entfernt werden. In verschiedenen Tabs können dabei die aktuellen Werte von Variablen, das Ergebnis von Methoden und der Aufrufstapel separat inspiziert werden.

- **pudb** ist ein weiterer visueller Debugger, der auf die Steuerung durch die Kommandozeilel basiert. Beim Starten erzeugt er ein Bildschirm-Layout auf der Konsole ähnlich wie bei einer IDE. Von Vorteil sind bei diesem Debbuger die Eigenschaft, dass mehr Informationen auf dem Bildschirm gleichzeitig sichtbar sind, es werden weitere Funktionen wie das Syntax-Highlighting angeboten und die Möglichkeit ihn zusätzlich durch die Tastatur-Befehle zu steuern, was schnelleres und effizienteres Debuggen erlaubt.

- **wdb** ist ein Web-Debugger, der unter anderem im Browser benutzt werden kann, er ist folglich auf kein spezielles Software oder IDE angewiesen. wdb eignet sich dafür, die webbasierten Anwendungen mit einem Web-Framework für Python wie Django oder Flask zu entwickeln.

5 Zusammenfassung

Es existieren zahlreiche Methoden zum Debugging von Programmen in Python, die beliebtesten Debugger bleiben dabei pdb und Gdb. pdb ist ein interaktiver Debugger und ein Teil der Standardbibliothek von Python, der verschiedene Standardfunktionen unterstützt wie das Inspizieren von Daten und Stacktrace, die Setzung von bedingten Haltepunkten und das Auflisten vom Quellcode. Gdb ist ein Programm, das standardmäßig auf Linux zugeschnitten ist und womit die Anwendungen in verschiedenen Programmiersprachen, und unter anderem Python effizient debuggt werden können. Normalerweise wird aber Gdb eher im Falle von den Bugs eingesetzt, die von intern implementierten Debuggern von Python nicht effizient behandelt werden können.

Literatur

[1] Kristian Rother. Pro Python Best Practices - Debugging, Testing and Maintenance. 2017. Letzter Zugriff: 03.09.2019

[2] Python Software Foundation: "pdb — The Python Debugger"
https://docs.python.org/3/library/pdb.html, Letzter Zugriff: 03.09.2019

[3] Nathan Jennings: "Python Debugging With Pdb"
https://realpython.com/python-debugging-pdb/, Letzter Zugriff: 03.09.2019

[4] Doug Hellmann: "pdb — Interactive Debugger"
https://pymotw.com/3/pdb/, Letzter Zugriff: 03.09.2019

[5] Python Software Foundation: "gdb Support"
https://devguide.python.org/gdb/, Letzter Zugriff: 03.09.2019

[6] Wiki.python. "Debugging-WithGdb"
https://wiki.python.org/moin/DebuggingWithGdb, Letzter Zugriff: 03.09.2019

[7] David Malcolm. "Features/EasierPythonDebugging"
https://fedoraproject.org/wiki/Features/EasierPythonDebugging, Letzter Zugriff: 03.09.2019

[8] Roman Podoliaka. "Debugging of CPython processes with gdb"
https://www.podoliaka.org/2016/04/10/debugging-cpython-gdb/, Letzter Zugriff: 03.09.2019

[9] Elliot Forbes. "Debugging with the Python Debugger-PDB"
https://tutorialedge.net/python/debugging-with-pdb-python/, Letzter Zugriff: 03.09.2019

[10] Wikipedia, the free encyclopedia https://en.wikipedia.org/wiki/Ptrace, Letzter Zugriff: 03.09.2019

[11] Stephan Augsten. "Was ist ein Debugger?"
https://www.dev-insider.de/was-ist-ein-debugger-a-730931/, Letzter Zugriff: 03.09.2019

[12] Wikipedia, the free encyclopedia https://en.wikipedia.org/wiki/Debugger, Letzter Zugriff: 03.09.2019