6. Module

Wenn man den Python-Interpreter beendet und erneut startet, gehen alle vorgenommenen Definitionen (Funktionen und Variablen) verloren. Deshalb benutzt man vorzugsweise einen Text-Editor, um längere und komplexere Programme zu schreiben und verwendet eine Datei als Eingabe für den Interpreter. Diese Datei wird auch als Skript bezeichnet. Wird ein Programm länger, teilt man es besser in mehrere Dateien auf, um es leichter warten zu können. Außerdem ist es von Vorteil, nützliche Funktionen in mehreren Programmen verwenden zu können, ohne sie in jedem Programm erneut definieren zu müssen.

Hierfür bietet Python die Möglichkeit, etwas in einer Datei zu definieren und diese in einer anderen Datei oder in der interaktiven Konsole wieder zu verwenden. So eine Datei wird als Modul bezeichnet. Definitionen eines Moduls können in anderen Modulen oder in das Hauptmodul importiert werden, welches die Gesamtheit aller Funktionen und Variablen enthält, auf die man in einem Skript zugreifen kann.

Ein Modul ist eine Datei, die Python-Definitionen und -Anweisungen beinhaltet. Der Dateiname mit dem `.py`-Suffix entspricht dem Namen des Moduls. Innerhalb eines Moduls ist der Modulname als `__name__` verfügbar (globale Variable des Typs String). Zum Beispiel: Öffne einen Editor Deiner Wahl und erstelle eine Datei im aktuellen Verzeichnis mit dem Namen `fibo.py` und folgendem Inhalt:

```
# Fibonacci-Zahlen-Modul

def fib(n):    # schreibe Fibonacci-Folge bis n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # gib die Fibonacci-Folge zurück bis n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Öffne danach Deinen Python-Interpreter und importiere das Modul mit folgendem Befehl:

```
>>> import fibo
```

Dieser Befehl fügt die von `fibo.py` definierten Funktionen nicht automatisch in die globale Symboltabelle (symbol table) ein, sondern nur den Modulnamen `fibo`. Um die Funktionen anzusprechen, benutzt man den Modulnamen:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```
Wenn man plant, eine Funktion öfters zu verwenden, kann man diese an einen lokalen Namen binden.

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
6.1. Mehr zum Thema Module

Ein Modul kann sowohl ausführbare Anweisungen, als auch Definitionen einer Funktion enthalten. Diese Anweisungen sind dazu gedacht, das Modul zu initialisieren, und werden nur ausgeführt, wenn das Modul zum ersten Mal importiert wird.

Jedes Modul hat seine eigene, private Symboltabelle, welche wiederum als globale Symboltabelle von allen Funktion in diesem Modul verwendet wird. Daher kann der Verfasser eines Moduls ohne Bedenken globale Variablen in seinem Modul verwenden, da sie sich nicht mit den globalen Variablen des Benutzers überschneiden können. Andererseits kann man (wenn man weiß, was man tut) auch die globalen Variablen eines Moduls verändern, indem man die gleiche Schreibweise verwendet, um auch dessen Funktionen anzusprechen, `modname.itemname`.

Module können andere Module importieren. Es ist üblich, aber nicht zwingend notwendig, dass man alle `import`-Anweisungen an den Anfang eines Moduls setzt (oder in diesem Fall Skripts). Die Namen der importierten Module werden in die Symboltabelle des importierenden Moduls eingefügt.

Es gibt eine Variante der `import`-Anweisung, welche bestimmte Namen aus einem Modul direkt in die Symboltabelle des importierenden Moduls einfügt. Beispielsweise:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
Diese Variante fügt allerdings nicht den Modulnamen, aus dem die Namen importiert werden, in die lokale Symboltabelle ein, sondern nur die aufgeführten. In diesem Beispiel wird `fibo` also nicht eingefügt.

Zusätzlich gibt es eine Variante um alle Namen eines Moduls zu importieren:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 37
```
Hiermit werden alle Namen, sofern sie nicht mit einem Unterstrich (_) beginnen , importiert. In den meisten Fällen wird diese Variante nicht verwendet, denn dadurch werden

unbekannte Namen in den Interpreter importiert und so kann es vorkommen, dass einige Namen überschrieben werden, die bereits definiert worden sind.

Beachte, dass der `*`-Import eines Moduls oder Paketes im allgemeinen verpönt ist, da er oft schlecht lesbaren Code verursacht. Allerdings ist es in Ordnung ihn im interaktiven Interpreter zu benutzen, um weniger tippen zu müssen.

Bemerkung

Aus Effizienzgründen wird jedes Modul nur einmal durch eine Interpreter-Sitzung importiert. Deshalb muss man den Interpreter bei Veränderung der Module neustarten - oder man benutzt **`imp.reload()`**, beispielsweise `import imp; imp.reload(modulename)`, falls es nur ein Modul ist, welches man interaktiv testen will.

6.1.1. Module als Skript aufrufen

Wenn man ein Python-Modul folgendermaßen aufruft:

```
python fibo.py <Argumente>
```

wird der Code im Modul genauso ausgeführt, als hätte man das Modul importiert. Der einzige Unterschied ist, dass `__name__` jetzt `"__main__"` ist und nicht mehr der Name des Moduls. Wenn man nun folgende Zeilen an das Ende des Moduls anfügt:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

kann man die Datei sowohl als Skript als auch als importierbares Modul nutzen, da der Code, der die Kommandozeile auswertet, nur ausgeführt wird, wenn das Modul direkt als Hauptdatei ausgeführt wird:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Beim Import des Moduls wird dieser Code nicht ausgeführt:

```
>>> import fibo
>>>
```

Dies wird oft dazu verwendet, um entweder eine bequeme Benutzerschnittstelle zum Modul bereitzustellen oder zu Testzwecken (wenn das Modul als Skript ausgeführt wird, wird eine Testsuite gestartet).

6.1.2. Der Modul-Suchpfad

Wenn ein Modul mit dem Namen **`spam`** importiert wird, sucht der Interpreter zuerst nach einem eingebauten Modul mit diesem Namen. Wird es nicht gefunden, sucht er nach einer Datei, die `spam.py` benannt ist, in der Liste von Verzeichnissen der Variable **`sys.path`**. **`sys.path`** wird mit

- dem Verzeichnis, das das Eingabeskript enthält (oder das aktuelle Verzeichnis),

- `PYTHONPATH` (eine Liste von Verzeichnissen, mit derselben Syntax wie die Shell-Variable `PATH`)

- der installationsabhängige Standard.

Nach der Initialisierung können Python-Programme `sys.path` modifizieren. Das Verzeichnis, das das laufende Skript enthält befindet sich am Anfang des Suchpfads, noch vor den Pfaden der Standardbibliothek. Das bedeutet, dass Module in diesem Verzeichnis anstelle von Modulen der Standardbibliothek geladen werden, wenn sie denselben Namen haben. Dies ist ein Fehler, sofern man das Modul der Standardbibliothek nicht ersetzen will. Siehe Abschnitt Standardmodule für mehr Informationen.

6.1.3. “Kompilierte” Python-Dateien

Um den Start von kurzen Programmen, die viele Standardmodule verwenden, schneller zu machen, werden Dateien erstellt, welche bereits “byte-kompiliert” sind. Existiert eine Datei mit dem Namen `spam.pyc`, so ist das eine “byte-kompilierte” Version der Datei `spam.py` und des Moduls `spam`. Der Zeitpunkt an dem die Datei `spam.py` zuletzt geändert wurde, wird in `spam.pyc` festgehalten. Falls die Zeiten nicht übereinstimmen, wird die `.pyc` ignoriert.

Normalerweise muss man nichts tun, damit die `spam.pyc`-Datei erstellt wird. Immer, wenn `spam.py` erfolgreich kompiliert wird, wird auch versucht die kompilierte Version in `spam.pyc` zu schreiben. Es wird kein Fehler geworfen, wenn der Vorgang scheitert; wenn aus irgendeinem Grund die Datei nicht vollständig geschrieben sein sollte, wird die daraus resultierende `spam.pyc` automatisch als fehlerhaft erkannt und damit später ignoriert. Der Inhalt der `spam.pyc` ist plattformunabhängig, wodurch man ein Modul Verzeichnis mit anderen Maschinen ohne Rücksicht auf ihre Architektur teilen kann.

Einige Tipps für Experten:

- Wird der Python-Interpreter mit dem `-O`-Flag gestartet, so wird der optimierte Code in `.pyo`-Dateien gespeichert. Optimierter Code hilft momentan nicht viel, da er lediglich `assert`-Anweisungen entfernt. Wird `-O` verwendet, wird der komplette Bytecode optimiert; `.pyc` werden ignoriert und `.py`-Dateien werden zu optimiertem Bytecode kompiliert.

- Werden dem Python-Interpreter zwei `-O`-Flags übergeben, vollzieht der Bytecode-Compiler Optimierungen, die zu einer Fehlfunktion des Programms führen können. Momentan werden nur `__doc__`-Strings aus dem Bytecode entfernt, was zu

kleineren `.pyo`-Dateien führt. Da einige Programme sich darauf verlassen, dass sie verfügbar sind, sollte man diese Option nur aktivieren, wenn man weiß, was man tut.

- Ein Programm wird in keinster Weise schneller ausgeführt, wenn es aus einer `.pyc` oder `.pyo` anstatt aus einer `.py`-Datei gelesen wird. Der einzige Geschwindigkeitsvorteil ist beim Starten der Dateien.

- Wenn ein Skript durch das Aufrufen über die Kommandozeile ausgeführt wird, wird der Bytecode nie in eine `.pyc`- oder `.pyo`-Datei geschrieben. Deshalb kann die Startzeit eines Skripts durch das Auslagern des Codes in ein Modul reduziert werden. Es ist auch möglich eine `.pyc`- oder `.pyo`-Datei direkt in der Kommandozeile auszuführen.

- Es ist möglich, eine `.pyc`- oder `.pyo`-Datei zu haben, ohne dass eine Datei mit dem Namen `spam.py` für selbiges Modul existiert. Dies kann dazu genutzt werden, Python-Code auszuliefern, der relativ schwer rekonstruiert werden kann.

- Das Modul `compileall` kann `.pyc`-Dateien (oder auch `.pyo`, wenn `-O` genutzt wird) aus allen Modulen eines Verzeichnisses erzeugen.

6.2. Standardmodule

Python wird mit einer Bibliothek von Standardmodulen ausgeliefert, welche in der Python Library Reference beschrieben werden. Einige Module sind in den Interpreter eingebaut. Diese bieten Zugang zu Operationen, die nicht Teil des Sprachkerns sind, aber trotzdem eingebaut sind. Entweder, um Zugang zu Systemoperationen (wie z. B. Systemaufrufe) bereitzustellen oder aus Effizienzgründen. Die Zusammenstellung dieser Module ist eine Option in der Konfiguration, welche auch von der verwendeten Plattform abhängig ist. Beispielsweise ist das `winreg`-Modul nur unter Windows-Systemen verfügbar. Ein bestimmtes Modul verdient besondere Aufmerksamkeit: `sys`, welches in jeden Python-Interpreter eingebaut ist. Die Variablen `sys.ps1` und `sys.ps2` definieren die primären und sekundären Eingabeaufforderungen, die in der Kommandozeile verwendet werden:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

Diese beiden Variablen werden nur definiert, wenn der Interpreter im interaktiven Modus ist.

Die Variable `sys.path` ist eine Liste von Zeichenketten, die den Suchpfad des Interpreters vorgibt. Sie ist mit einem Standardpfad voreingestellt, der aus der Umgebungsvariable **`PYTHONPATH`** entnommen wird oder aus einem eingebauten Standardwert, falls **`PYTHONPATH`** nicht gesetzt ist. Man kann diese Variable mit normalen Listenoperationen verändern:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. Die `dir()`-Funktion

Die eingebaute Funktion **`dir()`** wird benutzt, um herauszufinden, welche Namen in einem Modul definiert sind. Es wird eine sortierte Liste von Strings zurückgegeben:

```
    >>> import fibo, sys
    >>> dir(fibo)
    ['__name__', 'fib', 'fib2']
    >>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'callstats', 'copyright',
'displayhook', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'getdefaultencoding', 'getdlopenflags', 'getrecursionlimit',
'getrefcount', 'hexversion', 'maxint', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
'version_info', 'warnoptions']
```

Wenn man keine Parameter übergibt, liefert **`dir()`** eine Liste der aktuell definierten Namen:

```
    >>> a = [1, 2, 3, 4, 5]
    >>> import fibo
    >>> fib = fibo.fib
    >>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo',
'sys']
```

Zu Beachten ist, dass alle Typen von Namen ausgegeben werden: Variablen, Module, Funktionen, etc.

`dir()` listet allerdings nicht die Namen der eingebauten Funktionen und Variablen auf. Falls man diese auflisten will, muss man das Standardmodul **`builtins`** verwenden:

```
    >>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
```

```
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'chr',
'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr',
'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter',
'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash',
'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min',
'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

6.4. Pakete

Pakete werden dazu verwendet, den Modul-Namensraum von Python zu strukturieren, indem man Modulnamen durch Punkte trennt. Zum Beispiel verweist `A.B` auf ein Untermodul `B` im Paket `A`. Genauso wie die Verwendung von Modulen den Autor von Modulen davor bewahrt, sich Sorgen um andere globale Variablennamen zu machen, so bewahrt die Verwendung von Modulen, die durch mehrere Punkte getrennt sind, den Autor davor, sich Sorgen um andere Modulnamen machen zu müssen.

Angenommen man will eine Sammlung von Modulen (ein “Paket”) erstellen, um Audiodateien und -daten einheitlich zu bearbeiten. Es gibt unzählige verschiedene Audioformate (gewöhnlicherweise erkennt man diese an Ihrer Dateiendung, z.B. `.wav`, `.aiff`, `.au`), sodass man eine ständig wachsende Sammlung von Modulen erstellen und warten muss. Außerdem will man auch verschiedene Arbeiten an den Audiodaten verrichten (wie zum Beispiel Mixen, Echo hinzufügen, etc.), also wird man immer wieder Module schreiben, die diese Arbeiten ausführen. Hier eine mögliche Struktur für so ein Paket (ausgedrückt in der Form eines hierarchischen Dateisystems):

```
sound/                          Top-level package
      __init__.py                Initialize the sound package
      formats/                   Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                   Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                   Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Wenn man das Paket importiert, sucht Python durch die Verzeichnisse im `sys.path`, um nach dem Paket in einem Unterverzeichnis zu suchen.

Die `__init__.py`-Datei wird benötigt, damit Python das Verzeichnis als Paket behandelt. Dies hat den Grund, dass Verzeichnisse mit einem normalen Namen, wie z.B. `string`, nicht unbeabsichtigt Module verstecken, die weiter hinten im Suchpfad erscheinen. Im einfachsten Fall ist `__init__.py` eine leere Datei, sie kann allerdings auch Initialisierungscode für das Paket enthalten oder die `__all__`-Variable setzen, welche später genauer beschrieben wird.

Benutzer eines Pakets können individuelle Module aus dem Paket importieren:

```
import sound.effects.echo
```

Dieser Vorgang lädt das Untermodul **`sound.effects.echo`**. Es muss mit seinem kompletten Namen referenziert werden:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Eine alternative Methode, um dieses Untermodul zu importieren:

```
from sound.effects import echo
```

Diese Variante lädt auch das Untermodul **`echo`**, macht es aber ohne seinen Paket-Präfix verfügbar. Man verwendet es folgendermaßen:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Eine weitere Möglichkeit ist, die beschriebene Funktion oder Variable direkt zu importieren:

```
from sound.effects.echo import echofilter
```

Genau wie in den anderen Beispielen, lädt dies das Untermodul **`echo`**. In diesem Fall wird aber die **`echofilter()`** Funktion direkt verfügbar gemacht:

```
echofilter(input, output, delay=0.7, atten=4)
```

Wenn man `from package import item` verwendet, kann das `item` entweder ein Untermodul und -paket sein oder ein Name, der in diesem Paket definiert ist (z. B. eine Funktion, eine Klasse oder Variable). Die `import`-Anweisung überprüft zuerst, ob das `item` in diesem Paket definiert ist; falls nicht, wird von einem Modul ausgegangen und versucht es zu laden. Wenn nichts gefunden wird, wird eine **`ImportError`**-Ausnahme geworfen.

Im Gegensatz dazu, muss bei Verwendung von `import item.subitem.subsubitem` jedes `item` ein Paket sein; das letzte `item` kann ein Modul oder ein Paket sein, aber es darf keine Klasse, Funktion oder Variable im darüber geordneten `item` sein.

6.4.1. * aus einem Paket importieren

Was passiert nun, wenn der Benutzer `from sound.effects import *` schreibt? Idealerweise würde man hoffen, dass dies irgendwie an das Dateisystem weitergereicht wird und alle Untermodule, die es im Paket gibt, findet und sie alle importiert. Das könnte sehr lange dauern und Untermodule zu importieren könnte Nebeneffekte hervorrufen, die nur dann auftreten sollten, wenn das Untermodul explizit importiert wird.

Die einzige Lösung ist, dass der Autor des Paketes einen expliziten Index des Paketes bereitstellt. Die **`import`**-Anweisung folgt folgender Konvention: Definiert die Datei `__init__.py` des Paketes eine Liste namens `__all__`, wird diese als Liste der Modulnamen behandelt, die bei `from package import *` importiert werden sollen. Es ist Aufgabe des Paketautoren diese Liste aktuell zu halten, wenn er eine neue Version des Paketes veröffentlicht. Paketautoren können sich auch entscheiden es nicht zu unterstützen, wenn sie einen *-Import ihres Paketes für nutzlos halten. Zum Beispiel könnte die Datei `sound/effects/__init__.py` folgenden Code enthalten:

```
__all__ = ["echo", "surround", "reverse"]
```

Dies würde bedeuten, dass `from sound.effects import *` die drei genannten Untermodule des **`sound`** Paketes importiert.

Ist `__all__` nicht definiert, importiert die Anweisung `from sound.effects import *` nicht alle Untermodule des Paketes **`sound.effects`** in den aktuellen Namensraum; es stellt nur sicher, dass das Paket **`sound.effects`** importiert wurde (möglicherweise führt es jeglichen Initialisierungscode in `__init__.py` aus) und importiert dann alle Namen, die im Paket definiert wurden. Inklusive der Namen, die in `__init__.py` definiert werden (und Untermodule die explizit geladen werden). Es bindet auch jegliche Untermodule des Paketes ein, die durch vorherige **`import`**-Anweisungen explizit geladen wurden. Schau Dir mal diesen Code an:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Hier werden die Module **`echo`** und **`surround`** in den aktuellen Namensraum importiert, da sie im Paket **`sound.effects`** definiert sind, wenn die Anweisung `from ... import` ausgeführt wird. (Das funktioniert auch wenn `__all__` definiert ist.)

Auch wenn manche Module so entworfen wurden, nur Namen, die einem bestimmten Namensschema folgen, bei einem `import *` zu exportieren, wird es dennoch als schlechte Praxis betrachtet.

Aber bedenke, dass an der Benutzung von `from Package import specific_submodule` nichts falsch ist! In der Tat ist es die empfohlene Schreibweise, es sei denn das importierende Modul benutzt gleichnamige Untermodule von anderen Paketen.

6.4.2. Referenzen innerhalb des Paketes

Werden Pakete in Unterpakete strukturiert (wie das `sound` -Paket im Beispiel), kann man absolute Importe benutzen um Untermodule der Geschwisterpakete zu referenzieren. Wenn das Modul `sound.filters.vocoder` beispielsweise das Modul `echo` im Paket `sound.effects` benutzen muss, kann es `from sound.effects import echo` nutzen.

Man kann auch relative Importe in der `from module import name` -Form der `import` -Anweisung nutzen. Diese Importe nutzen führende Punkte, um das aktuelle und die Elternpakete, die im relativen Import beteiligt sind, anzugeben. Aus dem `surround` -Modul heraus, könnte man beispielsweise folgendes nutzen:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Beachte, dass relative Importe auf dem Namen des aktuellen Moduls basieren. Da der Name des Hauptmoduls immer `"__main__"` ist, müssen Module, die als Hauptmodul einer Python-Anwendung gedacht sind, immer absolute Importe nutzen.

6.4.3. Pakete in mehreren Verzeichnissen

Pakete unterstützen ein weiteres besonderes Attribut: `__path__` . Es wird als Liste initialisiert, die den Namen des Verzeichnisses mit der `__init__.py` des Pakets enthält. Dies geschieht, bevor der Code in dieser Datei ausgeführt wird. Diese Variable kann verändert werden und eine Änderung beeinflusst zukünftige Suchen nach Modulen und Unterpaketen, die im Paket enthalten sind.

Auch wenn dieses Feature nicht oft gebraucht wird, kann sie benutzt werden um die Menge der Module, die in einem Paket gefunden werden, zu erweitern.

Fußnoten

[1] Eigentlich sind Funktionsdefinitionen auch ‘Anweisungen’, die ‘ausgeführt’ werden; die Ausführung einer Funktionsdefinition auf Modulebene fügt den Funktionsnamen in die globale Symboltabelle des Moduls ein.