open source library

Farid Hajji

Das Python Praxisbuch

Der große Profi-Leitfaden für Programmierer

ADDISON-WESLEY

3 Hello, World!

Während wir in Kapitel 2, *Die Python-Shell*, kleine Wegwerf-Code-Fragmente ausprobiert haben, ist es nun an der Zeit, ein richtiges Python-Programm zu erstellen und auszuführen.

Der überlieferten Tradition entsprechend, wird auch unser erstes Python Programm das berühmte *Hello, World!* sein. In seiner traditionellen Form gibt dieses Programm lediglich die Zeichenkette *Hello, World!* aus und beendet sich anschließend. Da dies bei Python ein Einzeiler wäre und somit ziemlich langweilig, ziehen wir die erweiterte Form vor, welche vom Benutzer eine Eingabe erwartet und diese im Anschluss zusammen mit einem Gruß ausgibt.

Der Zweck von *Hello, World!* liegt weniger darin, was es tut, sondern wie man es ausführt. Wenn *Hello, World!* läuft, dann sollten auch andere Programme laufen, sofern sie keine Fehler enthalten; denn dann weiß man, dass die Python-Installation auf dem jeweiligen Rechner voraussichtlich voll einsatzfähig ist.

3.1 Das Hello, World!-Programm

Python-Programme werden in Dateien mit der Endung `.py` mit einem beliebigen Editor geschrieben. Jeder Editor, der reine ASCII-Dateien ohne versteckte Markierungen speichern kann, ist hierfür geeignet: von *emacs* oder *vim* mit ihren Python-Modi, die für Syntaxcoloring und noch leichtere Eingabe sorgen, über den integrierten Editor der mitgelieferten Python-Entwicklungsumgebung *idle* oder einem Plug-in für Eclipse bis hin zum Editor Ihrer Wahl.

Welchen Editor Sie auch immer bevorzugen, geben Sie Folgendes in eine Datei namens *hello.py* ein:

```
#!/usr/bin/env python
# hello.py -- the traditional hello world program.

print "Hello, World!"
```

```
name = raw_input("What's your name? ")
print "Nice to meet you,", name
```

Wie aus dem Quellcode zu erkennen ist, würde dieses Programm erst den Text `Hello, World!` ausgeben und anschließend den Benutzer nach seinem Namen fragen. Mit dem Namen des Benutzers ausgestattet, begrüßt *hello.py* diesen mit einer schönen, netten, personalisierten Nachricht.

3.1.1 hello.py verstehen

Der Code von *hello.py* dürfte weitestgehend selbsterklärend sein:

- Die erste Zeile `#!/usr/bin/env python` heißt *she-bang*-Zeile. Sie wird weiter unten ausführlich erklärt.
- Die zweite Zeile `# hello.py -- ...` ist eine Kommentarzeile. In Python wird alles, was mit # anfängt, bis zum Ende der Zeile vom Python-Interpreter ignoriert. Es dient der Dokumentation für den Programmierer.
- Mit der Anweisung `print` wird ausgegeben. Wir geben hier die Zeichenkette `»Hello, World!«` auf die Standardausgabe aus. Die Standardausgabe ist typischerweise der Bildschirm, das Terminal-Fenster etc. aus dem das Programm gestartet wurde, es sei denn, sie wurde mit `>` oder `|` umgeleitet.
- Die Benutzereingabe wird mit Hilfe der eingebauten Funktion `raw_input` von der Standardeingabe entgegengenommen. Die Standardeingabe ist normalerweise die Tastatur, es sei denn, sie wurde mit `<` beim Aufruf des Programms zuvor umgeleitet.
- Der von `raw_input` eingelesene Wert wird der Variablen `name` zugewiesen. Das ist genauso wie unsere Zuweisung `radius = 5.0` aus dem Taschenrechner-Beispiel des Kapitels 2, *Die Python-Shell*.
- Die Ausgabe der personalisierten Nachricht erfolgt wieder mit `print` in der letzten Zeile des Programms. Zu beachten ist hier, dass `print` alle seine durch Komma getrennten Argumente (`"Nice to meet you"` und `name`) durch Leerzeichen getrennt ausgibt!

Bis auf die Funktion `raw_input` haben Sie sicher *hello.py* selbst erraten. Die eingebaute Hilfe (Docstring) von `raw_input` kann mit den aus Kapitel 2, *Die Python-Shell*, bekannten Techniken angezeigt werden:

```
raw_input(...)
    raw_input([prompt]) -> string

    Read a string from standard input.  The trailing newline is stripped.
    If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
    On Unix, GNU readline is used if enabled.  The prompt string, if given,
    is printed without a trailing newline before reading.
```

3.1.2 hello.py unter Unix ausführen

Um dieses Programm auszuführen, übergibt man *hello.py* dem Python-Interpreter: `python hello.py`. Sind mehrere Python-Versionen auf demselben Rechner installiert, ruft man einfach den passenden Interpreter auf: `/usr/local/bin/python2.5 hello.py`, `~/python/bin/python hello.py` etc.

Führen wir *hello.py* mit dem Python-Interpreter aus, den wir in Kapitel 1, *Python installieren*, im Home-Verzeichnis unter *~/python/bin/python* installiert hatten:

```
$ ~/python/bin/python hello.py
Hello, World!
What's your name? John Doe
Nice to meet you, John Doe
```

Diese Methode hat zwei Vorteile:

- Sie läuft sowohl unter Unix als auch unter Windows.
- Sie überlässt die Wahl des zu benutzenden Python-Interpreters nicht der Umgebungsvariablen `PATH`.

Unter Unix gibt es eine weitere Methode: Setzt man das ausführbare Bit (`x`-Bit) von *hello.py*, kann man auf die Nennung des Python-Interpreters unter Umständen verzichten. Das geht so:

Normalerweise ist das `x`-Bit einer neuen Datei nicht gesetzt:

```
$ ls -l hello.py
-rw-r--r--  1 farid  users  168 Feb 11 08:26 hello.py
```

Damit kann man *hello.py* nicht direkt aufrufen:

```
$ ./hello.py
./hello.py: Permission denied.
```

Das `x`-Bit für den User setzt man wie folgt:

```
$ chmod u+x hello.py
$ ls -l hello.py
-rwxr--r--  1 farid  users  168 Feb 11 08:26 hello.py
```

Jetzt kann ich *hello.py* direkt aufrufen:

```
$ ./hello.py
Hello, World!
What's your name? John Doe
Nice to meet you, John Doe
```

Sollen auch andere Benutzer derselben Gruppe (hier *users*) *hello.py* ausführen, sollte man auch das `x`-Bit für die Gruppe setzen:

```
$ chmod g+x hello.py
$ ls -l hello.py
-rwxr-xr--  1 farid  users  168 Feb 11 08:26 hello.py
```

Jetzt können alle Benutzer, die der Gruppe `users` angehören, *hello.py* ausführen, *sofern sie Zugriff auf das Verzeichnis haben, das diese Datei enthält*.

Last, but not least, kann man auch das `x`-Bit für alle anderen Benutzer (die nicht in der Gruppe *users* sind) setzen:

```
$ chmod o+x hello.py
$ ls -l hello.py
-rwxr-xr-x  1 farid  users  168 Feb 11 08:26 hello.py
```

Statt alle drei `chmod`-Aufrufe kann man auch das `x`-Bit für alle Benutzer gleichzeitig mit einer Kommandozeile wie `chmod a+x hello.py` setzen.

Was soll aber diese komische `./`-Schreibweise? Geht's nicht ohne?

```
$ hello.py
hello.py: Command not found.
```

Der Grund liegt darin, dass die Unix-Shell das Programm *hello.py* nur in den Verzeichnissen sucht, die in der Umgebungsvariablen `PATH` enthalten sind, da es sich nicht um einen relativen oder absoluten Pfad handelt. Zufälligerweise ist das aktuelle Verzeichnis nicht in `PATH` enthalten:

```
$ echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:\
/users/farid/bin
```

Damit also die Unix-Shell *hello.py* dennoch findet, kann man:

- entweder einen relativen oder absoluten Pfad angeben
- oder `PATH` um das aktuelle Verzeichnis ergänzen.

Die erste Möglichkeit wird mit `./hello.py` realisiert, denn `./hello.py` ist ein relativer Pfad (relativ zum aktuellen Verzeichnis '`.`'). Die zweite Möglichkeit ist etwas umständlicher, geht aber auch:

```
$ PATH=$PATH:.; export PATH

$ echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:\
/users/farid/bin:.

$ hello.py
Hello, World!
What's your name? Path Finder
Nice to meet you, Path Finder
```

Natürlich würde die Änderung von `PATH` nur so lange gelten, bis es erneut verändert wird oder bis zum Ende der Shell. Damit sie dauerhaft ist, sollte sie in der Shell-Startup-Datei eingetragen werden. Wie das geht, hängt von der jeweiligen Shell ab.

Aus Sicherheitsgründen sollten Sie `.` lieber nicht in `PATH` übernehmen. Wenn Sie jedoch nicht darauf verzichten wollen oder können, fügen Sie `.` besser ans Ende von `PATH` ein. Warum? Angenommen, Sie befinden sich in einem Verzeichnis eines anderen Users, und dieser hat dort ein Programm abgelegt, das genauso heißt wie ein Systemprogramm. Falls Sie das nicht wissen, könnten Sie statt des Systemprogramms das Benutzerprogramm aufrufen und wüssten nicht, dass Sie es mit dem falschen Programm zu tun haben!

Wir werden in Zukunft den Aufruf eines Programms *prog.py* immer mit `./prog.py` bewerkstelligen, aber Sie wissen jetzt, dass es auch anders gehen kann.

Die Frage, die sich jedoch hier stellt ist: *Welcher Python-Interpreter hat hello.py ausgeführt?*

Um diese Frage zu beantworten, probieren wir folgendes Programm aus:

```
#!/usr/bin/env python
# pyversion.py -- print current version of python

import sys
print sys.version, sys.prefix
```

Wir rufen es mit den verschiedenen Methoden auf, die wir gerade kennengelernt haben. Achten Sie dabei auf das Datum der Kompilierung des Python-Interpreters und auf dessen Präfix:

```
$ python pyversion.py
2.5.2 (r252:60911, Mar  1 2008, 14:15:45)
[GCC 4.2.1 20070719  [FreeBSD]] /usr/local

$ ~/python/bin/python pyversion.py
2.5.2 (r252:60911, Mar  1 2008, 18:37:16)
[GCC 4.2.1 20070719  [FreeBSD]] /users/farid/python

$ chmod u+x pyversion.py
$ ./pyversion.py
2.5.2 (r252:60911, Mar  1 2008, 14:15:45)
[GCC 4.2.1 20070719  [FreeBSD]] /usr/local

$ PATH=$PATH:.; export PATH
$ pyversion.py
2.5.2 (r252:60911, Mar  1 2008, 14:15:45)
[GCC 4.2.1 20070719  [FreeBSD]] /usr/local
```

Was für *pyversion.py* gilt, gilt natürlich auch für *hello.py*!

In diesem Beispiel gibt es zwei Python-Interpreter:

- den systemweiten Python-Interpreter mit Präfix */usr/local*
- unseren Home-Verzeichnis-Interpreter mit Präfix */users/farid/python*

Der Aufruf `python pyversion.py` hat den System-Interpreter ausgeführt, ebenso der Aufruf `./pyversion.py` bzw. `pyversion.py`. Nur `~/python/bin/python pyversion.py` hat unseren Home-Verzeichnis-Interpreter gestartet!

Wie kommt das?

- Bei `python pyversion.py` hat die Shell das Programm `python` aufgerufen. Da `PATH` */usr/local/bin* aber nicht */users/farid/python/bin* enthält, hat die Shell den systemweiten Interpreter */usr/local/bin/python* gestartet, und dieser hat dann *pyversion.py* ausgeführt.
- Bei `~/python/bin/python pyversion.py` hat die Shell keine andere Wahl gehabt, als */users/farid/python/bin/python* zu starten, weil es sich um einen absoluten Pfad handelt. `PATH` wird in dem Fall überhaupt nicht berücksichtigt.
- Bei `./pyversion.py` und `pyversion.py` hat die Shell versucht, die Datei *pyversion.py* selbst auszuführen. Der Unix-Kernel kann aber eine solche Datei nicht direkt zur Ausführung laden, weil sie kein Binärprogramm enthält. Dies erkennt der Kernel daran, dass eine bestimmte Präambel fehlt. Also schaut sich der Kernel dann die Datei genauer an und sucht nach einer *she-bang*-Zeile. Das, was in der she-bang Zeile steht, wird dann gestartet.
  Der Ausdruck *she-bang* ist eine Abkürzung für *sharp* (*#*) und *bang* (*!*).

Die *she-bang* Zeile von *pyversion.py* (und *hello.py*) lautet:

```
#!/usr/bin/env python
```

Daher wird der Unix-Kernel das Programm */usr/bin/env* starten und ihm die Argumente *python* und den Namen der Datei (also *pyversion.py* bzw. *hello.py*) übergeben. Nun ist */usr/bin/env* dafür verantwortlich, die `.py`-Datei auszuführen.

Jetzt ist */usr/bin/env* natürlich kein Python-Interpreter, sondern nur ein Zwischenprogramm. Dieses ruft hier *python* auf, genauso wie es die Shell tun würde; und genauso wie bei der Shell berücksichtigt es dabei `PATH`! Und jetzt kommt's: Da `PATH` den Pfad */usr/local/bin*, aber nicht */users/farid/python/bin* enthält, würde */usr/bin/env* den systemweiten Python-Interpreter */usr/local/bin/python* starten und den Namen der `.py`-Datei übergeben. Ergo wird die Datei vom Python-Interpreter mit dem Präfix */usr/local* ausgeführt!

Hinweis

Übrigens: Sogar wenn */users/farid/python/bin* in `PATH` gewesen wäre, jedoch nicht */usr/local/bin*, würde `/usr/bin/env` natürlich den systemweiten Python-Interpreter zuerst gefunden und ausgeführt haben.

Damit also unser Home-Verzeichnis-Python-Interpreter die Datei ausführt, können wir `PATH` entsprechend ergänzen:

```
$ PATH=$HOME/python/bin:$PATH; export PATH

$ echo $PATH
/users/farid/python/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:\
/usr/local/bin:/users/farid/bin

$ ./pyversion.py
2.5.2 (r252:60911, Mar  1 2008, 18:37:16)
[GCC 4.2.1 20070719  [FreeBSD]] /users/farid/python
```

Oder, wenn's nur für einen einzigen Aufruf gelten soll:

```
$ PATH=$HOME/python/bin:$PATH ./pyversion.py
2.5.2 (r252:60911, Mar  1 2008, 18:37:16)
[GCC 4.2.1 20070719  [FreeBSD]] /users/farid/python
```

Wie wir sehen, können wir also durch geschicktes Manipulieren des Pfades `PATH` zur Laufzeit den richtigen Python-Interpreter bestimmen. Das ist der Grund, warum die bevorzugte *she-bang*-Zeile `#!/usr/bin/env python` lautet und nicht etwa `#!/usr/local/bin/python`. Durch den Aufruf von */usr/bin/env* überlässt man es dem Systemadministrator, durch Setzen von `PATH` den Default Python-Interpreter auszuwählen. Außerdem sind Programme mit dieser *she-bang*-Zeile portabler, denn der Python-Interpreter muss nicht bei allen Rechnern in */usr/local/bin* vorhanden sein.

Hinweis

Linux-User: bei einigen Linux-Distributionen befindet sich *env* nicht unter */usr/bin*, sondern unter */bin*. In diesem Fall müssten Sie entweder alle Python-Skripte dahingehend ändern, dass ihre *she-bang* Zeile `#!/bin/env` statt `#!/usr/bin/env` lautet, oder, viel besser, Sie setzen ein Symlink von */usr/bin/env* nach */bin/env* mit der als *root* auszuführenden Anweisung `ln -sf /bin/env /usr/bin/env` .

In manchen Fällen ist die `#!/usr/bin/env python` *she-bang*-Zeile aber nicht die beste Wahl, trotz all ihrer Vorteile! Falls das Programm z.B. in einer CGI-Umgebung ausgeführt werden soll, würde der Webserver i.d.R. `PATH` *sanitizen*, d.h. aus Sicherheitsgründen auf eine minimale Anzahl von als »sauber« geltenden Pfade setzen (etwa auf */bin* und */usr/bin*). Da man dort kaum Einfluss auf `PATH` hat, besonders bei *shared hosting*-Umgebungen, hat man keine andere Wahl als die richtige Version von Python in der *she-bang*-Zeile explizit zu benennen: `#!/usr/local/bin/python2.5`.

Ein weiterer Grund, eine explizitere *she-bang*-Zeile anzugeben, könnte darin liegen, dass das jeweilige Programm auf eine ganz präzise, spezifische Version von Python angewiesen ist und mit der Systemversion von Python ohnehin nicht laufen würde.

Doch in der Regel sollten Sie bei der kanonischen *she-bang*-Zeile `#!/usr/bin/env python` bleiben und es dem Systemadministrator überlassen, durch Setzen von `PATH` die Default-Version von Python zu bestimmen, und den Anwender wählen lassen, zur Laufzeit bestimmen zu können, welcher Python-Interpreter aufgerufen werden soll.

All das ist nun schön und gut, aber warum spielt die Auswahl des richtigen Python-Interpreters überhaupt eine Rolle? Sind nicht alle Python gleich? Unsere beiden Programme laufen doch in beiden Fällen! Also was soll die ganze Aufregung?

Python-Interpreter scheinen nur gleich zu sein, aber es stimmt nicht! Zum einen sind unterschiedliche Versionen von Python nicht abwärtskompatibel: Ein Python 2.5-Programm läuft nicht unbedingt auf einem Python 2.4-Interpreter. Zum anderen sind sogar Python-Interpreter derselben Version (hier 2.5.2) trotzdem nicht zwingend gleich, weil sie eine unterschiedliche Anzahl von Drittanbietermodulen installiert haben können.

Erinnern Sie sich an das `ZODB`-Package, das wir im Kapitel 1, *Python installieren*, unserem Home-Verzeichnis-Interpreter hinzugefügt haben? Da wir es nicht in der Systemversion von Python installiert haben, würde ein Programm, das `import ZODB` enthält, nur mit dem Home-Verzeichnis-Interpreter laufen, nicht aber mit dem Interpreter der Systemversion:

```
$ ~/python/bin/python
Python 2.5.2 (r252:60911, Mar  1 2008, 18:37:16)
[GCC 4.2.1 20070719  [FreeBSD]] on freebsd7
Type "help", "copyright", "credits" or "license" for more information.
>>> import ZODB
>>> quit()

$ python
Python 2.5.2 (r252:60911, Mar  1 2008, 14:15:45)
[GCC 4.2.1 20070719  [FreeBSD]] on freebsd7
Type "help", "copyright", "credits" or "license" for more information.
>>> import ZODB
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named ZODB
>>> quit()
```

Darum ist es immer wichtig, dass ein `.py`-Programm vom richtigen Python-Interpreter ausgeführt wird!

3.1.3 hello.py unter Windows ausführen

Um *hello.py* auszuführen, rufen wir einfach den Python-Interpreter *python.exe* auf und übergeben ihm *hello.py* als Argument.

Angenommen, *hello.py* befindet sich in *D:\PythonBook*, und unser Python-Interpreter wurde in *D:\Python25* installiert. Wir starten ein *cmd.exe*-Shell-Fenster, und wechseln nach *D:\PythonBook*, wo sich unser Programm befindet:

```
E:\Documents and Settings\farid>D:

D:\>cd \PythonBook

D:\Pythonbook>
```

Nun rufen wir den Python-Interpreter *D:\Python25\python.exe* auf und übergeben *hello.py* als Argument:

```
D:\PythonBook>D:\Python25\python.exe hello.py
Hello, World!
What's your name? John Doe
Nice to meet you, John Doe

D:\Pythonbook>
```

Wenn der Installation Wizard seine Arbeit richtig getan hat, müsste man auch *hello.py* ohne Angabe von *python.exe* starten können:

```
D:\Pythonbook>hello.py
Hello, World!
What's your name? John Doe
Nice to meet you, John Doe
```

Das ist deswegen möglich, weil der Installation Wizard die Endung `.py` mit dem Python-Interpreter *D:\Python25\python.exe* in der Windows-Registry verbunden hat und *cmd.exe* somit wusste, welches Programm diese `.py`-Datei ausführen konnte.

Außerdem lässt sich *hello.py* einfach im Explorer anklicken. Es startet dann ein Shell-Fenster, in dem wir unseren Namen eingeben können. Die Antwort ist in dem Fall leider kaum zu erkennen, denn das Fenster schließt sich unmittelbar, nachdem sich *hello.py* beendet hat.

Zu beachten ist, dass *python.exe* normalerweise nicht in `Path` enthalten ist:

```
D:\PythonBook>python.exe
'python.exe' is not recognized as an internal or external command,
operable program or batch file.
```

```
D:\PythonBook>
```

In Kapitel 1, *Python installieren,* haben wir bei der Installation von *easy_install* unter Windows gezeigt, wie `Path` eingestellt wird.

Bei richtig gesetztem `Path` lässt sich der Python-Interpreter direkt als *python.exe* statt *D:\Python25\python.exe* aufrufen:

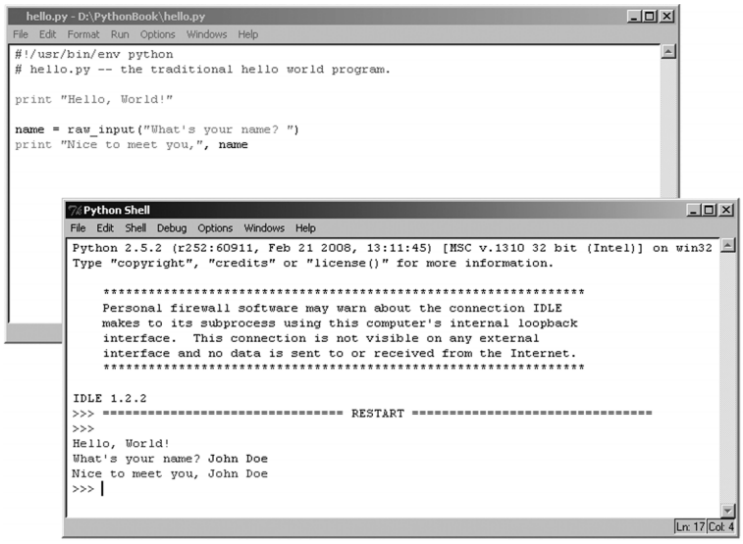
```
D:\PythonBook>python.exe hello.py
Hello, World!
What's your name? Path Found
Nice to meet you, Path Found

D:\PythonBook>
```

Befinden sich weitere Python-Versionen dort, können Sie Ihren Python-Interpreter entweder direkt aufrufen oder mittels `Path` auswählen.

3.1.4 hello.py in IDLE ausführen

Um *hello.py* in der Entwicklungsumgebung IDLE auszuführen, muss man lediglich diese Datei in den integrierten Editor laden und dort die Option Run / Run Module F5 aus dessen Menüleiste ausführen:



Das Programm läuft dann im Fenster der Python-Shell.

3.2 hello2.py mit sys.argv

Das Programm *hello.py* hat den Namen des Benutzers mit Hilfe der Funktion `raw_input` erhalten. Alternativ können Daten auch über die Kommandozeile in ein Programm eingegeben werden. Dies wird am Beispiel von *hello2.py* vorgeführt:

```
#!/usr/bin/env python
# hello2.py -- the traditional hello world program, cli version.

'''
This program greets the user and asks him for a name,
but only if the name has not been specified on the
command line interface as its first argument. Then
it welcomes the user with a nice personalized message.

Call this program either as:
  hello2.py "John Doe"
or as
  hello2.py

'''

import sys

def say_hello():
    "Say hello to the world"
    print "Hello, World!"

def ask_user_from_cli():
    "Fetch user name from the command line interface"
    if len(sys.argv) > 1:
        return sys.argv[1]
    else:
        return None

def ask_user_interactively():
    "Ask user for his name"
    return raw_input("What's your name? ")

def greet_user(name):
    "Send user a personalized greeting"
    print "Nice to meet you,", name
```

```
def main():
    "This is the main program"
    say_hello()
    name = ask_user_from_cli()
    if name is None:
        name = ask_user_interactively()
    greet_user(name)

if __name__ == '__main__':
    main()
```

3.2.1 hello2.py verstehen

Anders als *hello.py* ist *hello2.py* in *Funktionen* unterteilt. Jede Funktion hat eine genau definierte Aufgabe, die in ihrem Docstring zusammengefasst ist:

- `say_hello` gibt einfach `"Hello, World!"` aus.
- `ask_user_from_cli` liest die Kommandozeile aus und gibt entweder einen String oder das spezielle Python-Objekt `None` an den Aufrufer zurück.
- `ask_user_interactively` fragt wie in *hello.py* den Benutzer interaktiv nach seinem Namen und liefert diesen an den Aufrufer zurück.
- `greet_user` wird mit einem String (`name`) aufgerufen und gibt eine personalisierte Nachricht an den Benutzer aus.
- `main` ruft der Reihe nach all diese Funktionen auf: erst wird gegrüßt, dann wird, wenn möglich, der Name des Benutzers von der Kommandozeile eingelesen; und wenn es nicht gelang (z.B. weil beim Aufruf kein Name angegeben wurde), wird der Benutzer interaktiv aufgefordert, seinen Namen einzugeben. Anschließend wird der Benutzer mit einer personalisierten Nachricht willkommen geheißen.

Achten Sie auf den großzügigen Einsatz von Docstrings. Sie sollten es sich zur Gewohnheit machen, Ihre Programme immer gut zu dokumentieren, denn so lässt sich auch ganz bequem eine fertige Doku erstellen:

```
$ ~/python/bin/pydoc hello2
Help on module hello2:

NAME
    hello2

FILE
    /users/farid/PythonBook/hello2.py

DESCRIPTION
    This program greets the user and asks him for a name,
    but only if the name has not been specified on the
```

```
    command line interface as its first argument. Then
    it welcomes the user with a nice personalized message.

    Call this program either as:
      hello2.py "John Doe"
    or as
      hello2.py

FUNCTIONS
    ask_user_from_cli()
        Fetch user name from the command line interface

    ask_user_interactively()
        Ask user for his name

    greet_user(name)
        Send user a personalized greeting

    main()
        This is the main program

    say_hello()
        Say hello to the world
```

Neben der Modularisierung von *hello2.py* in Funktionen ist die wesentliche Neuerung der Inhalt von `ask_user_from_cli`. Um dies zu verstehen, muss man wissen, dass `sys.argv` ein Array ist, das die Argumente der Kommandozeile enthält. Dabei ist `sys.argv[0]` der Name des Programms selbst, `sys.argv[1]` ist der Wert des ersten Kommandozeilenarguments nach dem Programmnamen, `sys.argv[2]` der Wert des zweiten Kommandozeilenarguments usw. Die Zahl der Werte im Array `sys.argv` erhält man mit der `len`-Funktion: Ist also `len(sys.argv)` mehr als `1`, dann gibt es auf jeden Fall ein oder mehrere Kommandozeilenargumente. Wir schnappen uns nur das erste, `sys.argv[1]`, und liefern es als String zurück.

Eine weitere Neuerung ist das, was am Ende von *hello2.py* steht:

```
if __name__ == '__main__':
    main()
```

Was hat es damit auf sich? Man kann *hello2.py* entweder direkt aufrufen oder als Modul in die Python-Shell laden. Wenn es direkt aufgerufen wird, möchte man, dass `main` ausgeführt wird; aber wenn es in die Python-Shell geladen wird, soll `main` *nicht* aufgerufen werden (vielleicht will man es selbst tun oder lieber die eine oder andere Funktion austesten?):

```
>>> import hello2

>>> dir(hello2)
['__builtins__', '__doc__', '__file__', '__name__', 'ask_user_from_cli',
'ask_user_interactively', 'greet_user', 'main', 'say_hello', 'sys']

>>> hello2.say_hello()
Hello, World!

>>> quit()
```

Wir sehen, dass die Funktion `main` hier nicht aufgerufen wurde!

3.2.2 hello2.py ausführen

Führen wir *hello2.py* mal aus:

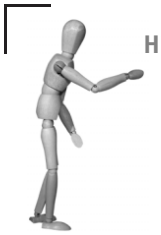
```
$ ~/python/bin/python hello2.py
Hello, World!
What's your name? John Doe
Nice to meet you, John Doe

$ ~/python/bin/python hello2.py John Doe
Hello, World!
Nice to meet you, John

$ ~/python/bin/python hello2.py "John Doe"
Hello, World!
Nice to meet you, John Doe
```

Der erste und dritte Aufruf müssten beide klar sein. Aber warum wurde beim zweiten Aufruf nur `John` und nicht `John Doe` übernommen? Des Rätsels Lösung liegt in der Unix-Shell: Sie erinnern sich, dass wir als Namen lediglich das erste Kommandozeilenargument `sys.argv[1]` nehmen. Die Unix Shell übergibt jedoch die Argumente `John` und `Doe` als getrennte Argumente, d.h. `John` würde nach `sys.argv[1]` landen und `Doe` wäre in `sys.argv[2]` zu finden! Es ist daher kein Wunder, dass `ask_user_from_cli` lediglich `John` als Namen zurückgeben würde, was auch die Ausgabe erklärt.

Damit `John Doe` als Name angenommen wird, muss es als einziges Argument von der Unix-Shell angesehen und übergeben werden, damit es nach `sys.argv[1]` landet. Und wie geht das? Indem man der Shell mit Anführungszeichen signalisiert, dass sie diese Argumente gruppieren und als ein einziges Argument übergeben soll.



Hinweis

Unter Windows ist der Aufruf identisch wie unter Unix.

3.2.3 hello2.py unterm Debugger

hello2.py kann man auch unter dem *Python-Debugger* `pdb` ausführen. Damit können Breakpoints gesetzt, der Stack und Variablen untersucht und Funktionen schrittweise ausgeführt werden.

Den Python-Debugger startet man sowohl unter Unix als auch Windows mit dem Aufruf: `python -m pdb prog.py arg1 arg2 ...`

```
$ ~/python/bin/python -m pdb hello2.py "John Doe"
> /users/farid/PythonBook/hello2.py(14)<module>()
-> '''
(Pdb)
```

Die Liste der Kommandos, die man am `Pdb`-Prompt eingeben kann, erhält man wie erwartet mit einem Fragezeichen:

```
(Pdb) ?

Documented commands (type help <topic>):
========================================
EOF    break  commands   debug    h       l     pp      s        up
a      bt     condition  disable  help    list  q       step     w
alias  c      cont       down     ignore  n     quit    tbreak   whatis
args   cl     continue   enable   j       next  r       u        where
b      clear  d          exit     jump    p     return  unalias

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Jedes Kommando hat eine weitere Hilfe, z.B.:

```
(Pdb) ?b
b(reak) ([file:]lineno | function) [, condition]
With a line number argument, set a break there in the current
file.  With a function name, set a break at first executable line
of that function.  Without argument, list all breaks.  If a second
argument is present, it is a string specifying an expression
which must evaluate to true before the breakpoint is honored.

The line number may be prefixed with a filename and a colon,
to specify a breakpoint in another file (probably one that
hasn't been loaded yet).  The file is searched for on sys.path;
the .py suffix may be omitted.
```

Setzen wir einen Breakpoint auf `ask_user_from_cli`:

```
(Pdb) b ask_user_from_cli
Breakpoint 1 at /users/farid/PythonBook/hello2.py:22
```

Und nun starten wir das Programm:

```
(Pdb) r
Hello, World!
> /users/farid/PythonBook/hello2.py(24)ask_user_from_cli()
-> if len(sys.argv) > 1:
```

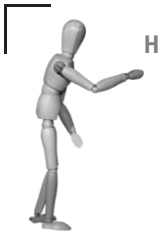
Wo sind wir gerade? Ein Stacktrace wäre nicht schlecht:

```
(Pdb) bt
  /users/farid/python/lib/python2.5/bdb.py(366)run()
-> exec cmd in globals, locals
  <string>(1)<module>()
  /users/farid/PythonBook/hello2.py(46)<module>()
-> main()
  /users/farid/PythonBook/hello2.py(40)main()
-> name = ask_user_from_cli()
> /users/farid/PythonBook/hello2.py(24)ask_user_from_cli()
-> if len(sys.argv) > 1:
```

Wir erkennen hier, indem wir den Stacktrace von unten nach oben lesen, dass wir uns in der Aufrufreihenfolge in `ask_user_from_cli` befinden, welches von `main` aufgerufen ist (welches seinerseits vom Debugger aufgerufen ist, aber das ist hier nicht wichtig).

Der Stacktrace verrät uns die Reihenfolge der aufgerufenen Funktionen. Die Stelle im Quellcode, an der wir uns befinden, erhalten wir anders:

```
(Pdb) l
 19         "Say hello to the world"
 20         print "Hello, World!"
 21
 22 B   def ask_user_from_cli():
 23         "Fetch user name from the command line interface"
 24  ->     if len(sys.argv) > 1:
 25             return sys.argv[1]
 26         else:
 27             return None
 28
 29     def ask_user_interactively():
```



Hinweis

Der Breakpoint wird hier mit einem `B` angezeigt und die Stelle, an der wir sind, mit einem Pfeil.

Variablen können hier zur Laufzeit inspiziert und verändert werden. Zum Spaß verändern wir mal schnell `sys.argv[1]`, um zu sehen, ob wir eine abweichende Ausgabe erhalten:

```
(Pdb) sys.argv
['hello2.py', 'John Doe']
(Pdb) sys.argv[1] = 'John "the debugger" Doe'
(Pdb) sys.argv
['hello2.py', 'John "the debugger" Doe']
```

Natürlich ist diese Änderung nicht dauerhaft! Sie ist nur während dieser Ausführung von *hello2.py* wirksam. Der Quellcode in *hello2.py* bleibt davon selbstverständlich unberührt! Falls die Ausführung innerhalb derselben Debugger-Sitzung neu gestartet wird, wird diese Änderung ebenfalls überschrieben.

Einen Einzelschritt kann man mit `s(tep)` ausführen:

```
(Pdb) s
> /users/farid/PythonBook/hello2.py(25)ask_user_from_cli()
-> return sys.argv[1]
```

Das stimmt: `len(sys.argv)` war tatsächlich größer als `1`, darum sind wir hier und nicht im `else`-Zweig gelandet!

Die Funktion `ask_user_from_cli` führen wir dann weiter aus, bis sie verlassen wird. Das geht mit dem `r(eturn)`-Befehl:

```
(Pdb) r
> /users/farid/PythonBook/hello2.py(41)main()
-> if name is None:
```

Wir sind wieder in der `main`-Funktion:

```
(Pdb) l
 36
 37     def main():
 38         "This is the main program"
 39         say_hello()
 40         name = ask_user_from_cli()
 41  ->     if name is None:
 42             name = ask_user_interactively()
 43         greet_user(name)
 44
 45     if __name__ == '__main__':
 46         main()
```

Inspizieren wir den Rückgabewert, der in `name` gespeichert wurde:

```
(Pdb) name
'John "the debugger" Doe'
```

Yep, wie erwartet!

Nun führen wir mit `c(ontinue)` die Ausführung fort, und zwar bis zum nächsten Breakpoint, oder, wie hier, wo keiner mehr da ist, bis zum Ende des Programms:

```
(Pdb) c
Nice to meet you, John "the debugger" Doe
The program finished and will be restarted
> /users/farid/PythonBook/hello2.py(14)<module>()
-> '''
```

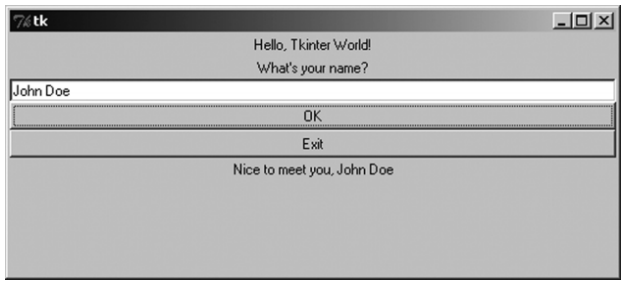
Haben Sie die Ausgabe erkannt? Unsere Änderung von `sys.argv` war offenbar wirksam.

Den Debugger verlassen wir dann mit `q(uit)`:

```
(Pdb) quit
$
```

3.3 tkhello.py mit Tkinter

Python-Programme mit GUI (*graphical user interface*) lassen sich genauso einfach wie traditionelle Nicht-GUI-Programme starten. Um dies zu illustrieren, schreiben wir unser *Hello, World!*-Programm so um, dass es folgende grafische Oberfläche bekommt:



Hinweis

Die Oberfläche mag nicht so ästhetisch aussehen, wie man es gewohnt ist. Wir verzichten hier auf Schönheit zugunsten der Einfachheit (des Quellcodes).

Wie geht so etwas? Mit Hilfe eines GUI-Toolkits natürlich! Oder, genauer gesagt, eines Python-Packages, das eine Schnittstelle zu einem GUI-Toolkit anbietet. Toolkits gibt's wie Sand am Meer (na ja, nicht ganz so viele, aber in ausreichender Vielfalt). Darum stellt sich die Frage, welches dieser Toolkits wir an dieser Stelle auswählen wollen.

Da man die allermeisten GUI-Toolkits und ihre Python-Packages erst nachträglich installieren muss, bietet sich als einziges Toolkit dasjenige, das automatisch Bestandteil der *batteries included* Python Standard Library ist, als natürlicher Kandidat an! Dieses Toolkit ist Tk, samt Python Package `Tkinter`. Tk mag nicht so hübsche Widgets besitzen wie andere GUI-Toolkits, hat aber zwei unbestreitbare Vorteile für schnelle Wegwerfprogramme:

- Tk gibt's für Unix, Windows und weitere Plattformen.
- `Tkinter` wird standardmäßig mit Python ausgeliefert.

Kommen wir nun zum Programm *tkhello.py*, das den oben gezeigten Screenshot erzeugt:

```
#!/usr/bin/env python
# tkhello.py -- Hello, World as a Tkinter application
import sys
from Tkinter import *
```

```
def build_gui():
    "Build the GUI. Return root, entry, and personalized greeting label"

    rootWindow = Tk()
    rootWindow.wm_geometry("500x200")

    label1 = Label(rootWindow)
    label1['text'] = "Hello, Tkinter World!"
    label1.pack()

    label2 = Label(rootWindow)
    label2['text'] = "What's your name?"
    label2.pack()

    nameEntry = Entry(rootWindow)
    nameEntry.bind('<Key-Return>', entry_callback)
    nameEntry.pack(fill=X)

    okButton = Button(rootWindow)
    okButton['text'] = 'OK'
    okButton['command'] = entry_callback
    okButton.pack(fill=X)

    exitButton = Button(rootWindow)
    exitButton['text'] = 'Exit'
    exitButton['command'] = exit_callback
    exitButton.pack(fill=X)

    outLabel = Label(rootWindow)
    outLabel['text'] = ''
    outLabel.pack()

    return rootWindow, nameEntry, outLabel

def entry_callback(event=None):
    "Called when the Return key is hit in the entry field or OK is clicked"
    name = theEntry.get()
    theLabel['text'] = "Nice to meet you, %s" % name

def exit_callback():
    "Called when the Exit button is hit"
    sys.exit(0)
```

```
def main():
    global theRoot, theEntry, theLabel
    theRoot, theEntry, theLabel = build_gui()
    theRoot.mainloop()

if __name__ == '__main__':
    main()
```

Als Erstes merken wir, dass der Quellcode deutlich länger ist als derjenige von *hello.py*. Das dürfte aber nicht überraschen: GUI-Programme sind *immer* länger als einfache Konsolen-Skripte. Doch das Programm müsste auch für `Tkinter`- oder gar GUI-Newbies zumindest nachvollziehbar sein.

Bevor wir zu den Details von *tkhello.py* kommen, wollen wir es starten. Unter Unix ruft man es einfach genauso auf, wie man jedes andere Python-Programm aufruft:

```
$ ~/python/bin/python tkhello.py
```

Das setzt aber voraus, dass

- man es bei laufendem X-Server tut,
- der X-Server Verbindungen vom Aufrufer zulässt, und
- die Umgebungsvariable `DISPLAY` richtig gesetzt ist.

All diese Bedingungen sind sicher erfüllt, wenn das Programm aus einem X-Fenster gestartet wird (z.B. einem *xterm*-Fenster), unabhängig davon, welchen Window-Manager Sie benutzen. Wird das Programm jedoch von der Konsole gestartet, ist `DISPLAY` natürlich nicht gesetzt, und wir erhalten eine Fehlermeldung (achten Sie auf die letzte Zeile):

```
$ ~/python/bin/python tkhello.py
Traceback (most recent call last):
  File "tkhello.py", line 56, in <module>
    main()
  File "tkhello.py", line 52, in main
    theRoot, theEntry, theLabel = build_gui()
  File "tkhello.py", line 10, in build_gui
    rootWindow = Tk()
  File "/users/farid/python/lib/python2.5/lib-tk/Tkinter.py",
      line 1636, in __init__
    self.tk = _tkinter.create(screenName, baseName, className, interactive,
                              wantobjects, useTk, sync, use)
_tkinter.TclError: no display name and no $DISPLAY environment variable
```

Unter Windows reicht es aus, auf *tkhello.py* zu klicken oder es wie gewohnt in der DOS-Box mit *tkhello.py* oder unter Angabe des Interpreters, z.B. als `python.exe tkhello.py`, zu starten.

Kehren wir nun zu *tkhello.py* zurück und schauen es uns etwas näher an, ohne dabei jedoch zu sehr in die Details zu versinken:

- *tkhello.py* ist in Funktionen modularisiert, wobei `main` die erste Funktion ist, welche aufgerufen wird.
- `main` ruft erst die Funktion `build_gui` auf, um das GUI aufzubauen.
- Nach `build_gui` ruft `main` die Haupteventschleife des Hauptfensters `theRoot` auf.

Das war's auch schon! Nun wartet diese Hauptschleife `mainloop` auf Ereignisse, d.h. darauf, dass der User etwas macht:

- Klickt er auf okButton, wird der Callback `entry_callback` automatisch aufgerufen. Dasselbe gilt für den Fall, dass der Benutzer im Texteingabefeld `nameEntry` die Return-Taste betätigt.
- Klickt er hingegen auf den Exit-Button exitButton, wird der Callback `exit_callback` aufgerufen; und dieser beendet das Programm wie gewünscht. Das Programm wird auch sauber beendet, wenn der Benutzer versucht, das Fenster mit Hilfe des Window-Managers zu schließen (z.B. durch Anklicken des `X` in der oberen rechten Ecke beim Screenshot).

Die Callbacks `entry_callback` und `exit_callback` sind gewöhnliche Python-Funktionen, welche vom `Tkinter`-Framework aufgerufen werden, wenn die in `build_gui` definierten Ereignisse eintreten. Mit anderen Worten, nicht wir rufen sie auf, sie werden über die Haupteventschleife `mainloop` automatisch bei Bedarf aufgerufen. Das nennt man *eventgesteuerte Programmierung*.

Interessant ist lediglich `entry_callback`: dort wird mittels der Methode `get` das Texteingabefeld `theEntry` ausgelesen und anschließend, um einen kleinen Satz ergänzt, als Text des Ausgabelabels eingetragen. Dadurch ändert sich die personalisierte Begrüßung.

Bleibt nur noch die Funktion `build_gui` zu erklären. Zunächst wird ein Hauptfenster mit `Tk()` erzeugt und auf eine bestimmte Größe gesetzt. Danach werden nacheinander die einzelnen Widgets erzeugt, die ins Hauptfenster gehören: zwei Textzeilen, ein Texteingabefeld nameEntry, zwei Buttons okButton und exitButton sowie ein Textausgabefeld outLabel.

Damit *tkhello.py* auf Eingaben reagiert, wird nameEntry an den Callback `entry_callback` angekoppelt. Somit würde das Klicken der Return-Taste automatisch den Aufruf dieses Callbacks bewerkstelligen (probieren Sie es mal aus!). Auch der okButton wird an diesen Callback angekoppelt, so dass der Benutzer die Wahl hat, entweder Return im Texteingabefeld einzugeben oder den ok-Button anzuklicken. Der Effekt bleibt derselbe.

exitButton hingegen wird an den Callback `exit_callback` angekoppelt, so dass sich das Programm beendet, sobald auf Exit geklickt wird.

Die personalisierte Begrüßung erfolgt in `outLabel`. Er bekommt in `build_gui` noch keinen Wert, denn er wird erst von `entry_callback` bei vorliegender Benutzereingabe belegt.

Es bleibt nur noch zu erwähnen, dass die Funktion `main` die Widgets `theRoot`, `theEntry` und `theLabel` als `global` deklariert hat, damit sie nicht nur im Body von `main` leben, sondern auch von `entry_callback` aus benutzt werden können.

Übrigens erhält man die Dokumentation zu den verschiedenen Widget-APIs, die Signatur einzelner Widget-Methoden oder die gesamte API wie gewohnt mit dem *pydoc*-Tool:

- `$ ~/python/bin/pydoc Tkinter.Label`
- `$ ~/python/bin/pydoc Tkinter.Entry`
- `$ ~/python/bin/pydoc Tkinter.Button`
- `$ ~/python/bin/pydoc Tkinter.Entry.bind`
- `$ ~/python/bin/pydoc Tkinter`

Auch GUI-Programme lassen sich debuggen, selbst wenn es etwas schwieriger ist als bei normalen CLI Programmen. Um *tkhello.py* zu debuggen, rufen wir den Python-Debugger wie folgt auf und setzen gleich einen Breakpoint an der interessantesten Stelle, dem Callback `entry_callback`:

```
$ ~/python/bin/python -m pdb tkhello.py
> /users/farid/PythonBook/tkhello.py(4)<module>()
-> import sys
(Pdb) b entry_callback
Breakpoint 1 at /users/farid/PythonBook/tkhello.py:41
```

Anschließend starten wir einfach das Programm mit `r(un)`:

```
(Pdb) r
```

An der Stelle erscheint das Hauptfenster. Wir geben im Eingabefeld etwas ein und drücken dann entweder die Return-Taste oder auf klicken den OK-Button. Im Debugger erscheint dann Folgendes:

```
> /users/farid/PythonBook/tkhello.py(43)entry_callback()
-> name = theEntry.get()
```

Im GUI-Fenster selbst sehen wir natürlich noch nichts. Nun inspizieren wir ein bisschen unsere Umgebung:

```
(Pdb) bt
  /users/farid/python/lib/python2.5/bdb.py(366)run()
-> exec cmd in globals, locals
  <string>(1)<module>()
  /users/farid/PythonBook/tkhello.py(56)<module>()
-> main()
  /users/farid/PythonBook/tkhello.py(53)main()
```

```
-> theRoot.mainloop()
  /users/farid/python/lib/python2.5/lib-tk/Tkinter.py(1023)mainloop()
-> self.tk.mainloop(n)
  /users/farid/python/lib/python2.5/lib-tk/Tkinter.py(1403)__call__()
-> return self.func(*args)
> /users/farid/PythonBook/tkhello.py(43)entry_callback()
-> name = theEntry.get()
```

Die letzte Zeile des Stacktrace zeigt, dass wir im Callback `entry_callback` angekommen sind. Interessant ist, wenn man den Stacktrace weiter nach oben klettert, dass `entry_callback` von einer Funktion `__call__` tief aus den Innereien des `Tkinter`-Moduls aufgerufen wurde. Erinnern Sie sich, dass es sich hier um eventgesteuerte Programmierung handelt? Diese ominöse `__call__`-Funktion (die für unsere Diskussion hier keine Rolle spielt) wird ihrerseits von der `mainloop` des `Tkinter`-Moduls aufgerufen. Diese `tk.mainloop`-Funktion ist die Haupteventschleife, die wir oben erwähnt haben. Sie selbst wurde von `theRoot.mainloop()` aus gestartet.

Aus dem Stacktrace kann man daher durchaus einiges über die Struktur von `Tkinter`-Programmen lernen, insbesondere über die Haupteventschleife.

Aber schauen wir uns nun an, wo wir im Quellcode gelandet sind:

```
(Pdb) list
 38
 39         return rootWindow, nameEntry, outLabel
 40
 41 B   def entry_callback(event=None):
 42         "Called when the Return key is hit in the entry field \
              or OK is clicked"
 43  ->     name = theEntry.get()
 44         theLabel['text'] = "Nice to meet you, %s" % name
 45
 46     def exit_callback():
 47         "Called when the Exit button is hit"
 48         sys.exit(0)
```

Wir sind also in `entry_callback` (das wussten wir schon), direkt an der Stelle, wo jetzt der Wert aus dem Texteingabefeld `theEntry` ausgelesen werden soll. Führen wir nun einen einzelnen Schritt aus, um zu sehen, was wir da erhalten:

```
(Pdb) s
--Call--
> /users/farid/python/lib/python2.5/lib-tk/Tkinter.py(2369)get()
-> def get(self):
```

Statt die ganze Funktion auszuführen, haben wir uns hier in die `tk.get`-Methode tief in die Innereien des `Tkinter`-Moduls verirrt (wir hätten `n(ext)` statt `s(tep)` benutzen

sollen, um zur nächsten Zeile von `entry_callback` zu springen, statt in die Funktion `get` des `Tkinter`-Moduls einzutauchen – aber hier geht es ja darum, eine echte Debugger-Sitzung vorzuführen, und in der Realität macht man eben mal den einen oder anderen Fehler). Da uns dessen Details nicht interessieren, führen wir diese Methode bis zum Ende aus und kehren zurück. Doch wie?

```
(Pdb) ?r
r(eturn)
Continue execution until the current function returns.
```

Ah, ja, mit `r(eturn)`:

```
(Pdb) r
--Return--
> /users/farid/python/lib/python2.5/lib-tk/Tkinter.py(2371)get()->'John Doe'
-> return self.tk.call(self._w, 'get')
```

Sind wir wirklich zurück in `entry_callback`? Mit `l(ist)` finden wir es heraus:

```
(Pdb) l
2366        def delete(self, first, last=None):
2367            """Delete text from FIRST to LAST (not included)."""
2368            self.tk.call(self._w, 'delete', first, last)
2369        def get(self):
2370            """Return the text."""
2371 ->         return self.tk.call(self._w, 'get')
2372        def icursor(self, index):
2373            """Insert cursor at INDEX."""
2374            self.tk.call(self._w, 'icursor', index)
2375        def index(self, index):
2376            """Return position of cursor."""
```

Nein, doch noch nicht. Also weiter eine Ebene zurück: diese Funktion aus den Innereien von `Tkinter` soll auch bis zu ihrem Ende laufen und zum Aufrufer (hoffentlich `entry_callback`) zurückkehren:

```
(Pdb) r
> /users/farid/PythonBook/tkhello.py(44)entry_callback()
-> theLabel['text'] = "Nice to meet you, %s" % name
```

Na, das sieht schon viel besser aus! Wir sind zurück, aber jetzt auf der folgenden Zeile:

```
(Pdb) l
 39         return rootWindow, nameEntry, outLabel
 40
 41 B   def entry_callback(event=None):
 42         "Called when the Return key is hit in the entry field \
```

```
              or OK is clicked"
 43         name = theEntry.get()
 44  ->     theLabel['text'] = "Nice to meet you, %s" % name
 45
 46     def exit_callback():
 47         "Called when the Exit button is hit"
 48         sys.exit(0)
 49
```

Was haben wir in `name` erhalten? Das, was im Texteingabefeld der User eingetragen hat?

```
(Pdb) name
'John Doe'
```

Genau!

Jetzt führen wir das Programm bis zum Ende aus, nicht jedoch, ohne zuvor den Breakpoint an `entry_callback` zu löschen:

```
(Pdb) clear
Clear all breaks? y
(Pdb) r
--Return--
> /users/farid/PythonBook/tkhello.py(44)entry_callback()->None
-> theLabel['text'] = "Nice to meet you, %s" % name
```

An dieser Stelle ist die personalisierte Nachricht im Hauptfenster erschienen! Nun kann der Benutzer noch ein paar weitere Eingaben vornehmen, um den Text unter dem Exit-Button zu verändern. Beim Klicken auf Exit oder Schließen des Fensters wird das Programm beendet. Die Sitzung im Python-Debugger wird ebenfalls beendet, und man kehrt zur Kommandozeile zurück.

3.4 Zusammenfassung

In diesem Kapitel haben wir drei Versionen eines einfachen Programms des Typs *Hello, World!* mit Benutzereingabe vorgeführt:

- *hello.py* holt sich seine Eingabe mit `raw_input` direkt vom Benutzer.
- *hello2.py* holt sich seine Eingabe aus der Kommandozeile `sys.argv`.
- *tkhello.py* ist die GUI-Version von *hello.py*.

Bei *Hello, World!*-Programmen geht es nicht so sehr darum, was sie tun, sondern wie sie aufgerufen werden.

Das einfache *hello.py*-Programm wird wie folgt aufgerufen:

- Unter Unix:
  - Vom von `PATH` gewählten Interpreter: `$ python hello.py`
  - Von einem spezifischen Interpreter: `$ ~/python/bin/python hello.py`
  - Bei `$ chmod u+x hello.py; ./hello.py` wird der von `PATH` gewählte Interpreter *hello.py* ausführen, es sei denn, die *she-bang*-Zeile spezifiziert etwas anderes.
  - Die kanonische *she-bang*-Zeile lautet bei allen Plattformen:
    `#!/usr/bin/env python`
- Unter Windows:
  - *D:\Python25\python.exe hello.py*, wenn Python unter *D:\Python25* installiert ist
  - *hello.py*, da die Endung `.py` mit dem (hoffentlich) richtigen Python-Interpreter vom Installation Wizard verbunden wurde
  - `python.exe hello.py`, wenn `Path` den Pfad von *python.exe* enthält (also z.B. um *D:\Python25* ergänzt). Bei XP: Start -> Control Panel -> System -> Advanced -> Environment variables
- In IDLE (Unix, Windows, ...):
  - *idle* starten
  - *hello.py* im Editor laden
  - im Editor-Fenster Run -> Run Module aufrufen

Als Nächstes haben wir das Programm *hello2.py* betrachtet, das seine Eingabe aus der Kommandozeile `sys.argv` annimmt. *hello2.py* ist, anders als *hello.py*, in Funktionen modularisiert gewesen:

- Bei der Ausführung gibt man einfach die zusätzlichen Argumente nach dem Namen des Programms an: `$ ~/python/bin/python hello2.py "John Doe"`.
- Damit der Benutzername und -vorname als ein einziges Argument in `sys.argv[1]` landen, muss man der Shell mit Anführungszeichen mitteilen, dass sie diese Kommandozeilenargumente zu einem einzigen Kommandozeilenargument zusammenfassen soll.

hello2.py haben wir auch unter dem Python-Debugger ausgeführt:

- `$ ~/python/bin/python -m pdb hello2.py »John Doe«`
- Im Pdb erhält man eine Liste aller Kommandos mit `(Pdb) ?` und kann sich über einzelne Kommandos informieren mit dem Fragezeichen: `(Pdb) ?b`.
- Man kann Breakpoints setzen: `(Pdb) b ask_user_from_cli`. Mit `r` wird das Programm unter Debugger-Kontrolle gestartet. Der Debugger stoppt beim ersten Breakpoint. Mit `bt` erhält man einen Stacktrace und mit `l(ist)` die aktuelle Position im Quellcode. Variablen kann man wie in der Python-Shell ausgeben und u.U. sogar verändern. Mit `n(ext)` geht es zur nächsten Zeile, wobei Funktionen übersprungen werden, die dort aufgerufen werden; mit `s(tep)` werden sie nicht übersprungen, man kann sie bis zum Ende durchlaufen lassen mit `r(eturn)`. Last but not least, mit `c(ontinue)` führt man das Programm bis zum Ende durch.

Als drittes Programm haben wir die GUI-Version von *hello.py* namens *tkhello.py* anhand des standardmäßig vorhandenen `Tkinter`-Moduls vorgeführt:

- *tkhello.py* lässt sich wie jedes andere Python-Programm starten: `$ ~/python/bin/python tkhello.py` oder bei Windows z.B. durch Anklicken. Läuft bei Unix der X-Server nicht, wird eine Ausnahme ausgelöst, und das Programm wird beendet.
- *tkhello.py* ist in Funktionen modularisiert und unterliegt dem Design der *event-gesteuerten Programmierung*. Das heißt, dass nach dem Zusammenstellen des GUIs und dem Aufruf der Haupteventschleife die einzelnen Funktionen (Callbacks) automatisch vom Toolkit in Abhängigkeit von den Benutzeraktionen aufgerufen werden.
- Die API-Dokumentation zu den verschiedenen `Tkinter`-Widgets erhält man wie gewohnt mit *pydoc*.
- Das Debuggen von GUI-Programmen erfolgt wie dasjenige von Nicht-GUI-Programmen mit Hilfe des Python-Debuggers.

Im nächsten Teil werden die Python-Grundlagen behandelt.