KAPITEL 3

Ihre Entwicklungsumgebung

Dieses Kapitel gibt einen Überblick über Texteditoren, integrierte Entwicklungsumgebungen (IDEs) und andere Entwicklungswerkzeuge, die aktuell für den Editier-Test-Debug-Zyklus von Python beliebt sind.

Wir geben unumwunden zu, dass wir Sublime Text (siehe »Sublime Text« auf Seite 25) als Editor und PyCharm/IntelliJ IDEA (siehe »PyCharm/IntelliJ IDEA« auf Seite 31) als IDE bevorzugen, doch wir verstehen auch, dass die beste Wahl von der Aufgabe und den anderen von Ihnen genutzten Sprachen abhängt. Dieses Kapitel stellt eine Reihe der beliebtesten Werkzeuge vor und nennt Gründe für ihre Wahl.

Python benötigt keine Build-Tools wie Make oder Javas Ant bzw. Maven, da es interpretiert wird und nicht kompiliert.[1] Wir werden sie hier daher nicht weiter behandeln. In Kapitel 6 erläutern wir aber, wie man Setuptools nutzt, um Projekte zu paketieren, und Sphinx für die Dokumentation verwendet.

Wir werden auch nicht auf Versionskontrollsysteme eingehen, da diese sprachunabhängig sind, doch die meisten derjenigen, die die C-(Referenz-)Implementierung von Python pflegen, sind gerade von Mercurial auf Git umgestiegen (siehe *PEP 512 (https://www.python.org/dev/peps/pep-0512/)*). Die ursprüngliche Begründung für den Einsatz von Mercurial in *PEP 374 (https://www.python.org/dev/peps/pep-0374/)* enthält einen kurzen, aber nützlichen Vergleich zwischen den vier heute am weitesten verbreiteten Optionen Subversion, Bazaar, Git und Mercurial.

Das Kapitel endet mit einem kurzen Blick auf die verschiedenen Möglichkeiten, unterschiedliche Interpreter zu betreiben, um die möglichen Deployment-Szenarien während der Entwicklung zu replizieren.

1 Wenn Sie an irgendeinem Punkt C-Erweiterungen für Python entwickeln wollen, sehen Sie sich *Extending Python with C or C++ (https://docs.python.org/3/extending/extending.html)* an. Weitere Details finden Sie in Kapitel 15 des *Python Cookbook (http://bit.ly/python-cookbook)*.

Texteditoren

Mit nahezu allem, mit dem man auch nur einen einfachen Text schreiben kann, kann man Python-Code entwickeln. Doch die Wahl des richtigen Editors kann Ihnen einige Stunden Arbeit pro Woche einsparen. Alle in diesem Abschnitt vorgestellten Texteditoren unterstützen die Syntaxhervorhebung und können über Plugins um statische Codeprüfungen (»Linter«) und Debugger erweitert werden.

Tabelle 3-1 führt die von uns bevorzugten Texteditoren (nach Präferenz absteigend sortiert) auf und nennt Gründe, warum ein Entwickler den einen oder anderen nutzen sollte. Der Rest des Kapitels geht detaillierter auf jeden Editor ein. Falls Sie nach speziellen Features suchen, finden Sie auf Wikipedia einen *sehr detaillierten Vergleich zu Texteditoren (https://en.wikipedia.org/wiki/Comparison_of_text_editors)*.

Tabelle 3-1: Texteditoren auf einen Blick

Tool	Verfügbarkeit	Gründe für den Einsatz
Sublime Text	• Offene API/kostenlose Testversion • OS X, Linux, Windows	• Schnell, geringer Speicherdarf. • Kommt auch mit großen Dateien (> 2 GB) zurecht. • Erweiterungen sind in Python geschrieben.
Vim	• Open Source/Spenden willkommen • OS X, Linux, Windows, Unix	• Sie lieben Vi/Vim. • Zumindest Vi ist auf jedem Betriebssystem (außer Windows) vorinstalliert. • Läuft in der Konsole.
Emacs	• Open Source/Spenden willkommen • OS X, Linux, Windows, Unix	• Sie lieben Emacs. • Erweiterungen sind in Lisp geschrieben. • Läuft in der Konsole.
TextMate	• Open Source/lizenzpflichtig • Nur OS X	• Sehr gute Benutzerschnittstelle. • Nahezu alle Schnittstellen (statischer Codecheck/Debug/Test) sind vorinstalliert. • Gute Apple-Tools, z. B. die Schnittstelle zu xcodebuild (über das Xcode-Bundle).
Atom	• Open Source/frei • OS X, Linux, Windows	• Erweiterungen sind in JavaScript/HTML/CSS geschrieben. • Sehr schöne GitHub-Integration.
Code	• Offene API (endlich)/frei • OS X, Linux, Windows	• IntelliSense (Codevervollständigung) mit Microsofts Visual Studio vergleichbar. • Gut für Windows-Entwickler. Unterstützt .Net, C# und F#. • Nachteil: noch nicht erweiterbar (aber angekündigt).

Sublime Text

Sublime Text (http://www.sublimetext.com/) ist der von uns empfohlene Texteditor für Code, Markup und Prosa. Seine Geschwindigkeit ist das erste Argument für eine Empfehlung, die Zahl der verfügbaren Pakete (über 3.000) das nächste.

Sublime Text wurde 2008 von Jon Skinner erstmals veröffentlicht. Er ist in Python geschrieben, unterstützt das Editieren von Python-Code perfekt und nutzt Python auch für seine Paketerweiterungs-API. Ein *Projects*-Feature erlaubt dem Benutzer, Dateien oder Ordner hinzuzufügen (bzw. zu löschen). Diese können dann über die Funktion *Goto Anything* durchsucht werden, die dann die Stellen innerhalb des Projekts aufspürt, die den oder die Suchbegriffe enthalten.

Sie benötigen*PackageControl (https://packagecontrol.io/installation)*, um auf das Sublime Text-Paket-Repository zugreifen zu können. Zu den beliebten Paketen gehört SublimeLinter, ein Interface zu den vom Benutzer installierten statischen Codecheckern, Emmett für Webentwicklungs-Snippets[2] und Sublime SFTP für entferntes Editieren per FTP.

Anaconda (http://damnwidget.github.io/anaconda/) (das nichts mit der kommerziellen Python-Distribution gleichen Namens zu tun hat) wurde 2013 veröffentlicht und verwandelt Sublime fast in eine IDE mit statischen Codecheckern, Docstring-Checks, einem Testsystem und der Fähigkeit, die Definition markierter Objekte nachzuschlagen oder Beispiele für deren Einsatz zu zeigen.

Vim

Vim ist ein konsolenbasierter Texteditor (mit optionaler GUI). Anstelle von Menüs und Symbolen verwendet er Tastaturkürzel für das Editieren. Er wurde 1991 von Bram Moolenaar veröffentlicht, während sein Vorgänger Vi 1976 von Bill Joy veröffentlicht wurde. Beide sind in C geschrieben.

Vim ist über eine einfache Skriptsprache namens vimscript erweiterbar. Es gibt viele Optionen für den Einsatz anderer Sprachen. Das Python-Skripting aktivieren Sie, indem Sie bei der Kompilierung des C-Quellcodes die Flags `--enable-python-interp` und/oder `--enable-python3interp` setzen. Um zu prüfen, ob Python oder Python3 aktiviert ist, geben Sie `:echo has("python")` bzw. `:echo has("python3")` ein. Das Ergebnis ist »1«, wenn Python unterstützt wird, andernfalls »0«.

Vi (und häufig auch Vim) ist auf so gut wie jedem System (außer Windows) standardmäßig vorinstalliert. Für Windows gibt es einen *Installer für Vim (http://www.vim.org/download.php#pc)*. Wer die Lernkurve meistert, wird extrem effektiv. Das

2 Snippets sind oft verwendete Codefragemente wie CSS-Styles oder Klassendefinitionen, die automatisch vervollständigt werden können, wenn der Benutzer ein paar Zeichen eingibt und dann die Tabulatortaste drückt.

geht so weit, dass die grundlegenden Vi-Tastenkürzel als Konfigurationsoptionen für die meisten anderen Editoren und IDEs zur Verfügung stehen.

Wenn Sie für ein großes Unternehmen in irgendeiner IT-Position arbeiten wollen, sollten Sie zumindest rudimentär mit Vi umgehen können.[3] Vim bietet wesentlich mehr Möglichkeiten als Vi, doch es ist ähnlich genug, sodass ein Vim-Nutzer auch mit Vi zurechtkommt.

Wenn Sie nur in Python entwickeln, können Sie die Standardeinstellungen für Einrückungen und Zeilenumbrüche auf *PEP 8 (https://www.python.org/dev/peps/pep-0008)*-konforme Werte setzen. Dazu legen Sie eine Datei namens *.vimrc* in Ihrem Home-Verzeichnis an[4] und fügen die folgenden Zeilen hinzu:

```
set textwidth=79  " Zeilen mit mehr als 79 Zeichen werden umbrochen
set shiftwidth=4  " >> rückt um 4 Spalten ein; mit << geht's 4 Spalten zurück
set tabstop=4     " Tabs sind 4 Spalten lang
set expandtab     " Tabs werden durch Leerzeichen ersetzt
set softtabstop=4 " 4 Leerzeichen einfügen/löschen, wenn Tab/Backspace gedrückt wird
set shiftround    " Einrückung auf Vielfaches von 'shiftwidth' aufrunden
set autoindent    " Einrückung an der vorigen Zeile ausrichten
```

Bei diesen Einstellungen erfolgt nach 79 Zeichen ein Zeilenumbruch, eingerückt wird mit vier Zeichen pro Tab, und innerhalb einer eingerückten Zeile wird auch die nächste Zeile entsprechend eingerückt.

Es gibt ein Syntax-Plug-in namens *python.vim (http://bit.ly/python-vim)*, das gegenüber der bei Vim 6.1 mitgelieferten Syntaxdatei einige Verbesserungen bietet. Ein kleines Plug-in namens *SuperTab (http://bit.ly/supertab-vim)* vereinfacht die Codevervollständigung über die Tabulatortaste oder beliebige andere Tasten. Sollten Sie Vim auch für andere Sprachen nutzen, gibt es ein praktisches Plug-in namens *indent (http://bit.ly/indent-vim)*, das die Einstellung der Einrückung für Python-Quelldateien übernimmt.

Diese Plug-ins versorgen Sie mit einer grundlegenden Entwicklungsumgebung für Python. Wenn Ihr Vim mit `+python` kompiliert wurde (seit Vim 7 der Standard), können Sie auch das Plug-in *vim-flake8 (https://github.com/nvie/vim-flake8)* verwenden, mit dem statische Codechecks direkt im Editor möglich sind. Es stellt die Funktion `Flake8` zur Verfügung, die *PEP 8 (http://pypi.python.org/pypi/pep8/)* und *Pyflakes (http://pypi.python.org/pypi/pyflakes/)* ausführt und mit einer beliebigen Taste oder Aktion verknüpft werden kann. Das Plug-in gibt Fehler am unteren Bildrand aus und bietet eine einfache Möglichkeit, zur entsprechenden Zeile zu springen.

3 Öffnen Sie einfach den Editor, indem Sie `vi` (oder `vim`) in der Kommandozeile eingeben. Im Editor geben Sie dann `:help` gefolgt von Enter ein, um das Tutorial aufzurufen.

4 Um Ihr Home-Verzeichnis unter Windows zu ermitteln, öffnen Sie Vim und geben `:echo $HOME` ein.

Wenn Sie wollen, können Sie es so einrichten, dass Vim `Flake8` immer aufruft, wenn eine Python-Datei gespeichert wird. Fügen Sie dazu die folgende Zeile in Ihre *.vimrc* ein:

```
autocmd BufWritePost *.py call Flake8()
```

Wenn Sie bereits mit *syntastic (https://github.com/scrooloose/syntastic)* arbeiten, können Sie es so einstellen, dass es Pyflakes beim Schreiben ausführt und Fehler bzw. Warnungen im *quickfix*-Fenster ausgibt. Hier eine entsprechende Beispielkonfiguration, die auch Status- und Warnmeldungen in der Statuszeile ausgibt:

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

Python-Mode

Der *Python-Mode (https://github.com/klen/python-mode)* ist eine komplexe Lösung zur Bearbeitung von Python-Code unter Vim. Wenn Sie die hier vorgestellten Features mögen, sollten Sie ihn nutzen (beachten Sie aber, dass Vim dann etwas länger zum Starten braucht):

- asynchrones Python-Code-Checking (`pylint`, `pyflakes`, `pep8`, `mccabe`) in beliebiger Kombination
- Code-Refactoring und Autovervollständigung mit *rope (https://github.com/python-rope/rope)*
- schnelles Python-Folding (Sie können eingerückten Code »verstecken«)
- Unterstützung für virtualenv
- die Fähigkeit, die Python-Dokumentation zu durchsuchen und Python-Code auszuführen
- automatische *PEP 8 (http://pypi.python.org/pypi/pep8/)*-Fehlerkorrekturen

Emacs

Emacs ist ein weiterer mächtiger Texteditor. Er verfügt mittlerweile über eine GUI, kann aber auch direkt in der Konsole ausgeführt werden. Er ist vollständig programmierbar (in Lisp) und kann mit ein wenig Mühe zu einer Python-IDE aufgebohrt werden. Masochisten und *Raymond Hettinger (http://pyvideo.org/speaker/138/raymond-hettinger)*[5] nutzen ihn.

5 Wir lieben Raymond Hettinger. Wenn jeder so coden würde, wie er es empfiehlt, wäre die Welt ein wesentlich besserer Ort.

Emacs ist in Lisp geschrieben und wurde 1976 von Richard Stallman und Guy L. Steele, Jr. vorgestellt. Zu den fest integrierten Features gehört entferntes Editieren (per FTP), ein Kalender, das Senden und Lesen von E-Mail und sogar ein Psychiater (Esc, dann `x`, dann `doctor`). Populäre Plug-ins sind unter anderem *YASnippet*, mit dem Sie Code-Snippets auf Tasten legen können, und Tramp für das Debugging. Es ist über einen eigenen Lisp-Dialekt namens elisp plus erweiterbar.

Wenn Sie bereits mit Emacs arbeiten, finden Sie im EmacsWiki *»Python Programming in Emacs« (http://emacswiki.org/emacs/PythonProgrammingInEmacs)* Tipps und Tricks zu Python-Paketen und zur Konfiguration. Emacs-Neulinge sollten mit dem *offiziellen Emacs-Tutorial (http://bit.ly/gnu-emacs-tutorial)* starten.

Momentan gibt es für Emacs drei wesentliche Python-Modi:

- Fabián Ezequiel Gallinas python.el, das seit der Emacs-Version 24.3+ im Lieferumfang enthalten ist, implementiert Syntaxhervorhebung, Einrückung, Bewegung, Interaktion mit der Shell und eine Reihe weiterer *gängiger Features des Emacs-Editier-Modus (https://github.com/fgallina/python.el#introduction)*.
- Jorgen Schäfers *Elpy (http://elpy.readthedocs.org/)* zielt auf eine vollständige, interaktive Entwicklungsumgebung in Emacs ab – inklusive Debugging, Linter und Codevervollständigung.
- *Pythons Quelldistribution (https://www.python.org/downloads/source/)* wird mit einer alternativen Version ausgeliefert, die Sie im Verzeichnis *Misc/python-mode.el* finden. Sie können sie im Web auch separat von *launchpad (https://launchpad.net/python-mode)* herunterladen. Sie besitzt einige Tools zur Programmierung per Spracheingabe sowie zusätzliche Tastaturkürzel und ermöglicht die *Einrichtung einer vollständigen Python-IDE (http://www.emacswiki.org/emacs/ProgrammingWithPythonModeDotEl)*.

TextMate

TextMate (http://macromates.com/) ist ein GUI-basierter Editor mit Emacs-Wurzeln, der nur unter OS X läuft. Er besitzt ein Apple-typisches Interface, das es irgendwie hinbekommt, unaufdringlich zu sein und gleichzeitig alle Befehle mit minimalem Aufwand zur Verfügung zu stellen.

TextMate ist in C++ geschrieben und wurde 2004 erstmals von Allan Oddgard und Ciarán Walsh vorgestellt. Sublime Text (siehe »Sublime Text« auf Seite 25) kann TextMate-Snippets direkt importieren, und Microsofts Editor Code (siehe »Code« auf Seite 29) kann Syntaxhervorhebungen von TextMate direkt importieren.

In beliebigen Sprachen geschriebene Snippets können in gebündelten Gruppen eingefügt und darüber hinaus über Shell-Skripte erweitert werden. Außerdem ist es möglich, einen beliebigen Text zu markieren und über die Tastenkombination Cmd+| (Pipe) als Standardeingabe an ein Skript weiterzuleiten. Die Ausgabe des Skripts ersetzt den markierten Text.

Die Syntaxhervorhebung für Apples Swift und Objective-C ist fest integriert. Über das Xcode-Bundle steht eine Schnittstelle zu xcodebuild zur Verfügung. Ein TextMate-Veteran wird keine Probleme haben, mit diesem Editor in Python zu codieren. Neulinge, die nicht viel Zeit mit der Entwicklung von Apple-Produkten verbringen, sind mit den neueren plattformübergreifenden Editoren (die sich häufig ausgiebig bei TextMates beliebtesten Features bedienen) besser bedient.

Atom

Atom (https://atom.io/) ist ein »hackbarer Texteditor für das 21. Jahrhundert«. So lautet zumindest die Aussage der Leute von GitHub, die ihn entwickelt haben. Er wurde 2014 veröffentlicht und ist in CoffeeScript (JavaScript) und Less (CSS) geschrieben. Er basiert auf Electron (früher Atom Shell)[6], also auf GitHubs Applikations-Shell, die wiederum auf io.js und Chromium aufsetzt.

Atom ist über JavaScript und CSS erweiterbar, und Nutzer können Snippets in jedweder Sprache hinzufügen (auch TextMate-basierte Snippet-Definitionen). Wie sicher erwartet, besitzt er eine sehr schöne GitHub-Integration. Er beinhaltet ein natives Paketmanagement und eine Vielzahl von Paketen (über 2.000). Für die Python-Entwicklung empfohlen wird *Linter (https://github.com/AtomLinter/Linter)* in Kombination mit *linter-flake8 (https://github.com/AtomLinter/linter-flake8)*. Webentwicklern könnte auch der *Atom-Entwicklungsserver (https://atom.io/packages/atom-development-server)* gefallen, der einen kleinen HTTP-Server ausführt und eine HTML-Vorschau in Atom erlaubt.

Code

Microsoft hat Code im Jahr 2015 angekündigt. Es ist ein kostenloser Closed-Source-Texteditor innerhalb der Visual-Studio-Familie und basiert ebenfalls auf GitHubs Electron. Er läuft plattformübergreifend, und die Tastenkombinationen erinnern stark an TextMate.

Code verfügt über eine *Erweiterungs- API (https://code.visualstudio.com/Docs/extensions/overview)* – verfügbare Erweiterungen finden Sie im *Marktplatz für VS-Code-Erweiterungen (https://code.visualstudio.com/docs/editor/extension-gallery)*. Er vereint alles, was seine Entwickler für das Beste von TextMate und Atom mit Microsoft halten. Er kommt mit IntelliSense (Codevervollständigung), die an Visual Studio heranreicht, und unterstützt .Net, C# und F#.

6 Electron ist eine Plattform zur Entwicklung plattformübergreifender Desktopanwendungen mit HTML, CSS und JavaScript.

IDEs

Viele Entwickler nutzen sowohl einen Texeditor als auch eine IDE, wobei bei größeren, komplexeren oder eher kollaborativen Projekten meist zur IDE gewechselt wird. Tabelle 3-2 hebt die Unterscheidungsmerkmale einiger beliebter IDEs hervor. Die nachfolgenden Abschnitte gehen dann etwas genauer auf die jeweiligen IDEs ein.

Ein Feature, das häufig als Grund für den Einsatz einer vollwertigen IDE genannt wird (neben Codevervollständigung und Debugging-Tools), ist die Fähigkeit, schnell zwischen den Python-Interpretern zu wechseln (z. B. von Python 2 zu Python 3 zu IronPython). Diese Möglichkeit beinhalten alle der in Tabelle 3-2 aufgeführten freien IDE-Versionen. Visual Studio bietet das mittlerweile bei allen Varianten.[7]

Zusätzliche Features (die kostenlos sein können oder auch nicht) sind Schnittstellen zu Ticketing-Systemen, Tools für das Deployment (z. B. Heroku oder Google App Engine), Werkzeuge für das kollaborative Arbeiten, entferntes Debugging sowie zusätzliche Features zur Nutzung von Web-Frameworks wie Django.

Tabelle 3-2: IDEs auf einen Blick

Tool	Verfügbarkeit	Gründe für den Einsatz
PyCharm/Intellij IDEA	• Offene API/kostenpflichtige Professional-Edition • Open Source/kostenlose Community-Edition • OS X, Linux, Windows	• Nahezu perfekte Codevervollständigung. • Gute Unterstützung virtueller Umgebungen. • Gute Unterstützung von Web-Frameworks (in der kostenpflichtigen Version).
Aptana Studio 3/ Eclipse + LiClipse + PyDev	• Open Source/kostenlos • OS X, Linux, Windows	• Sie lieben Eclipse. • Java-Unterstützung (LiClipse/Eclipse).
WingIDE	• Offene API/kostenlose Testversion • OS X, Linux, Windows	• Großartiger (Web-)Debugger, der beste der hier aufgeführten IDEs. • Per Python erweiterbar.
Spyder	• Open Source/kostenlos • OS X, Linux, Windows	• Data Science: IPython ist in Spyder integriert, wird im Paket mit NumPy, SciPy und Matplotlib geliefert. • Die Standard-IDE beliebter wissenschaftlicher Python-Distributionen: Anaconda, Python(x,y) und WinPython.
NINJA-IDE	• Open Source/Spenden willkommen • OS X, Linux, Windows	• Bewusst leichtgewichtig. • Starker Python-Fokus.

7 *https://github.com/Microsoft/PTVS/wiki/Features-Matrix*

Tabelle 3-2: IDEs auf einen Blick (Fortsetzung)

Tool	Verfügbarkeit	Gründe für den Einsatz
Komodo IDE	• Offene API/Texteditor (Komodo Edit) ist Open Source • OS X, Linux, Windows	• Python, PHP, Perl, Ruby, Node. • Erweiterungen basieren auf Mozilla-Add-ons.
Eric (Eric Python IDE)	• Open Source/Spenden willkommen • OS X, Linux, Windows	• Ruby + Python. • Bewusst leichtgewichtig. • Großartiger (wissenschaftlicher) Debugger. Kann einen Thread debuggen, während andere weiterlaufen.
Visual Studio (Community)	• Offene API/kostenlose Community-Edition • Kostenpflichtige Professional- und Enterprise-Edition	• Sehr gute Integration von Microsoft-Sprachen und -Tools. • IntelliSense (Codevervollständigung) ist fantastisch. • Projektmanagement und Unterstützung beim Deployment inklusive Tools zur Sprint-Planung und Manifest-Template in der Enterprise-Edition. • Vorbehalt: Virtuelle Umgebungen werden nur in der (teuersten) Enterprise-Edition unterstützt.

PyCharm/IntelliJ IDEA

PyCharm (http://www.jetbrains.com/pycharm/) ist die von uns bevorzugte Python-IDE. Die Hauptgründe dafür sind die nahezu perfekte Codevervollständigung sowie die Qualität der Tools für die Webentwicklung. In der Wissenschaftscommunity wird die kostenlose Version (bei der die Webentwicklungstools fehlen) als für ihre Bedürfnisse ausreichend empfunden, es wird aber öfter auf Spyder (siehe »Spyder« auf Seite 33) zurückgegriffen.

PyCharm wird von JetBrains entwickelt, das auch für IntelliJ IDEA bekannt ist, eine proprietäre Java-IDE, die mit Eclipse konkurriert. PyCharm (veröffentlicht 2010) und IntelliJ IDEA (veröffentlicht 2001) nutzen die gleiche Codebasis, und ein Großteil der Features von PyCharm lässt sich bei IntelliJ durch das kostenlose *Python-Plug-in (http://bit.ly/intellij-python)* ergänzen.

JetBrains empfiehlt PyCharm für ein einfacheres Benutzer-Interface und IntelliJ IDEA, wenn Sie Introspektion von Jython-Funktionen vornehmen möchten, zwischen Sprachen wechseln müssen oder ein Java-zu-Python-Refactoring nötig ist. (PyCharm funktioniert mit Jython, aber nur als mögliche Wahl für den Interpreter. Die Werkzeuge zur Introspektion stehen nicht zur Verfügung.) Die beiden werden separat lizenziert, Sie müssen sich also vor dem Kauf entscheiden.

Die IntelliJ Community Edition und die PyCharm Commuity Edition sind Open Source (Apache-2.0-Lizenz) und kostenlos.

Aptana Studio 3/Eclipse + LiClipse + PyDev

Eclipse ist in Java geschrieben und wurde 2001 von IBM als offene, vielseitige Java-IDE veröffentlicht. *PyDev (http://pydev.org/)*, das Eclipse-Plug-in für die Python-Emtwicklung, wurde 2003 von Aleks Totic veröffentlicht, der den Stab später an Fabio Zadrozny übergab. Es ist das beliebteste Eclipse-Plug-in für die Python-Entwicklung.

Obwohl sich die Eclipse-Community nicht wirklich gewehrt hat, als Forenbesucher anfingen, IntelliJ IDEA und Eclipse zu vergleichen und IntelliJ den Vorzug zu geben, ist Eclipse immer noch die am häufigsten verwendete Java-IDE. Das ist für Python-Entwickler von Interesse, die Schnittstellen zu in Java entwickelten Tools nutzen müssen, da es für viele beliebte Werkzeuge (wie etwa Hadoop, Spark und die entsprechenden proprietären Varianten) Instruktionen und Plug-ins für die Entwicklung mit Eclipse gibt.

Ein PyDev-Fork ist in *Aptana Studio 3 (http://www.aptana.com/products/studio3.html)* integriert, einer Open-Source-Suite mit Plug-ins für Eclipse, die eine Python-IDE (mit Django), Ruby (und Rails), HTML, CSS und PHP bietet. Der primäre Fokus von Aptanas Eigentümer (der Fa. Appcelerator) liegt auf Appcelerator Studio, einer proprietären Mobilplattform für HTML, CSS und JavaScript, bei der monatliche Gebühren fällig werden (sobald eine App live geht). Allgemeine Unterstützung für PyDev und Python ist vorhanden, hat aber keine Priorität. Abgesehen davon ist Aptanas Studio 3 eine gute Wahl, wenn Sie Eclipse mögen, als JavaScript-Entwickler hauptsächlich für mobile Plattformen entwickeln und gelegentliche Ausflüge in Richtung Python unternehmen müssen.

LiClipse wurde aus dem Wunsch nach verbesserter Mehrsprachigkeit und dem einfachen Zugriff auf vollständig dunkle Themes (was bedeutet, neben dem Texthintergrund sind auch Menüs und Rahmen dunkel) geboren. Es besteht aus einer von Zadrozny entwickelten proprietären Suite von Eclipse-Plug-ins. Ein Teil der (optionalen) Lizenzgebühren fließt an PyDev, um es auch weiterhin vollständig kostenlos und als Open Source anbieten zu können (es steht unter der EPL-Lizenz, die auch von Eclipse genutzt wird). Es wird zusammen mit PyDev ausgeliefert, Python-Nutzer müssen es nicht mehr selbst installieren.

WingIDE

WingIDE (http://wingware.com/) ist eine Python-spezifische IDE und nach PyCharm die wohl zweitbeliebteste Python-IDE. Sie läuft unter Linux, Windows und OS X.

Die Debugging-Tools sind sehr gut und umfassen auch Tools zum Debugging von Django-Templates. Die Nutzer von WingIDE nennen als Hauptgründe für den Einsatz dieser IDE den Debugger, die steile Lernkurve und den geringen Speicherbedarf.

Wing wurde im Jahr 2000 von Wingware veröffentlicht und ist in Python, C und C++ geschrieben. Erweiterungen werden unterstützt, ein Plug-in-Repository gibt

es bisher aber noch nicht. Nutzer müssen also in anderen Blogs und GitHub-Accounts nach verfügbaren Paketen suchen.

Spyder

Spyder (https://github.com/spyder-ide/spyder) (eine Abkürzung für *Scientific PYthon DevelopmentEnviRonment*) ist eine IDE, die sich gezielt an Nutzer wissenschaftlicher Python-Bibliotheken richtet.

Spyder wurde von Carlos Córdoba in Python geschrieben. Es ist Open Source (MIT-Lizenz) und bietet Codevervollständigung, Syntaxhervorhebung, einen Klassen- und Funktionsbrowser sowie Objektinspektion. Zusätzliche Features sind über Community-Plug-ins verfügbar.

Spyder integriert *pyflakes (http://pypi.python.org/pypi/pyflakes/)*, *pylint (https://www.pylint.org/)* und *rope (https://github.com/python-rope/rope)* und wird mit NumPy, SciPy, IPython und Matplotlib geliefert. Es wird selbst mit den beliebten wissenschaftlichen Python-Distributionen Anaconda, Python(x, y) und WinPython ausgeliefert.

NINJA-IDE

NINJA-IDE (http://www.ninja-ide.org/) (nach dem rekursiven Akronym *Ninja-IDE Is Not Just Another IDE*) ist eine plattformübergreifende IDE für die Entwicklung von Python-Anwendungen. Sie läuft unter Linux/X11, Mac OS X und Windows. Installer für die verschiedenen Plattformen können von der NINJA-IDE-Website heruntergeladen werden.

Die NINJA-IDE ist in Python und Qt geschrieben, Open Source (GPLv3-Lizenz) und bewusst leichtgewichtig gehalten. Die beliebtesten Features sind die Hervorhebung von Problemcode bei der Ausführung statischer Codechecker und beim Debugging sowie die Vorschaufähigkeit für Webseiten. Sie kann über Python erweitert werden, und es gibt auch ein Plug-in-Repository. Dahinter steht die Idee, dass der Nutzer nur die Tools hinzufügt, die er benötigt.

Die Entwicklung ging eine Weile etwas langsam voran, doch die neue NINJA-IDE v3 ist für die nahe Zukunft angekündigt, und über *NINJA-IDE listserv (http://bit.ly/ninja-ide-listserv)* wird immer noch fleißig kommuniziert. In der Community finden sich viele Spanisch sprechende Mitglieder, darunter auch die Kernentwickler.

Komodo IDE

Komodo IDE (http://www.activestate.com/komodo-ide) wird von ActiveState entwickelt und ist eine kommerzielle IDE für Windows, Mac und Linux. *KomodoEdit (https://github.com/Komodo/KomodoEdit)*, der Texteditor der IDE, ist die Open-Source-Alternative (Mozilla Public License).

Komodo wurde im Jahr 2000 von ActiveState veröffentlicht und nutzt die Codebasis von Mozilla und Scintilla. Es ist über Mozilla-Add-ons erweiterbar und unterstützt Python, Perl, Ruby, PHP, Tcl, SQL, Smarty, CSS, HTML sowie XML. Komodo Edit besitzt keinen Debugger, dieser kann aber über ein Plug-in nachgerüstet werden. Die IDE unterstützt keine virtuellen Umgebungen, doch der Benutzer kann wählen, welcher Python-Interpreter verwendet werden soll. Die Django-Unterstützung ist nicht so umfangreich wie bei WingIDE, PyCharm und Eclipse + PyDev.

Eric (die Eric Python IDE)

Eric (http://eric-ide.python-projects.org/) ist Open Source (GPLv3-Lizenz) und wird seit über zehn Jahren aktiv weiterentwickelt. Die Eric Python IDE ist in Python geschrieben, basiert auf dem GUI-Toolkit Qt und integriert den Scintilla-Editor. Benannt ist sie nach Eric Idle, einem Mitglied von Monty Python (und eine Hommage an die IDLE-IDE). Sie wird zusammen mit den Python-Distributionen ausgeliefert.

Die Features umfassen Codevervollständigung, Syntaxhervorhebung, Unterstützung von Versionskontrollsystemen, Unterstützung von Python 3, einen integrierten Webbrowser, eine Python-Shell, einen integrierten Debugger und ein flexibles Plug-in-System. Eric besitzt keine zusätzlichen Werkzeuge für Web-Frameworks.

Wie NINJA-IDE und Komodo IDE ist diese IDE bewusst leichtgewichtig gehalten. Treue Nutzer glauben, dass es die besten Debugging-Tools hat, einschließlich der Fähigkeit, einen Thread anzuhalten und zu debuggen, während die anderen weiterlaufen. Wenn Sie Matplotlib für interaktive Plots in dieser IDE nutzen wollen, müssen Sie das Qt4-Backend verwenden:

```
# Muss zuerst kommen:
import matplotlib
matplotlib.use('Qt4Agg')

# Und dann nutzt pyplot das Qt4-Backend:
import matplotlib.pyplot as plt
```

Dieser Link verweist auf die aktuelle *Dokumentation der Eric IDE (http://eric-ide.python-projects.org/eric-documentation.html)*. Positives Feedback auf der Website zur Eric IDE stammt meist aus der Scientific-Computing-Community (Wettermodelle und Strömungsdynamik).

Visual Studio

Professionelle Entwickler, die mit Microsoft-Produkten unter Windows arbeiten, wünschen sich *Visual Studio (https://www.visualstudio.com/products)*. Es ist in C++ und C# geschrieben, und die erste Version erschien 1995. Ende 2014 wurde die

erste Visual Studio Community Edition für nicht kommerzielle Zwecke kostenlos zur Verfügung gestellt.

Wenn Sie hauptsächlich mit Enterprise-Software arbeiten und Microsoft-Produkte wie C# und F# nutzen, ist das Ihre IDE.

Stellen Sie sicher, dass Sie die *Python Tools for Visual Studio (PTVS) (https://www.visualstudio.com/en-us/features/python-vs.aspx)* mitinstallieren, was in den Installationsoptionen über eine Checkbox aktiviert werden muss, die standardmäßig nicht aktiv ist. Anweisungen zur Installation mit Visual Studio bzw. der Nachinstallation finden Sie auf der *PTVS-Wiki-Seite (https://github.com/Microsoft/PTVS/wiki/PTVS-Installation)*.

Interaktive Tools

Die hier aufgeführten Tools verbessern die Arbeit in der interaktiven Shell. IDLE ist eigentlich eine IDE, ist im vorigen Abschnitt aber nicht enthalten, da es im Gegensatz zu den anderen genannten IDEs von den meisten Anwendern als nicht robust genug empfunden wird (für Enterprise-Projekte). Es eignet sich aber hervorragend für die Lehre. IPython ist in Spyder standardmäßig enthalten und kann auch in andere IDEs eingebunden werden. Diese Tools ersetzen den Python-Interpreter nicht, sondern ergänzen die vom Benutzer gewählte Interpreter-Shell um zusätzliche Tools und Features.

IDLE

IDLE (http://docs.python.org/library/idle.html#idle), steht für *Integrated Development and Learning Environment*, (zu Deutsch etwa »Integrierte Entwicklungs- und Lernumgebung«) und ist gleichzeitig der Nachname des Monty-Python-Mitglieds Eric Idle). IDLE ist Teil der Python-Standardbibliothek und wird mit Python ausgeliefert.

IDLE wurde von Guido van Rossum (Pythons »Benevolent Dictator for Life«) vollständig in Python geschrieben und nutzt das Tkinter-GUI-Toolkit. Zwar eignet sich IDLE nicht für umfangreiche Python-Entwicklungen, doch es ist recht hilfreich, wenn man kleinere Python-Snippets ausprobieren oder mit verschiedenen Python-Features exerimentieren möchte.

Es enthält die folgenden Features:

- ein Python-Shell-Fenster (Interpreter),
- einen Multifenster-Texteditor mit farbig hervorgehobenem Python-Code sowie
- minimale Debugging-Fähigkeiten.

IPython

IPython (http://ipython.org/) bietet ein umfangreiches Toolkit für die interaktive Nutzung von Python. Die Hauptkomponenten sind:

- Leistungsfähige Python-Shells (Terminal- und Qt-basiert).
- Ein webbasierter »Notizblock« mit den gleichen Features wie die Terminal-Shell sowie Unterstützung für Rich Media, Text, Code, mathematische Ausdrücke und Inline-Plots.
- Unterstützung für die interaktive Datenvisualisierung (ist es entsprechend konfiguriert, erscheinen ihre Matplotlib-Plots in einem separaten Fenster) und die Nutzung von GUI-Toolkits.
- Flexible, einbettbare Interpreter, die in die eigenen Projekte geladen werden können.
- Tools für High-Level- und interaktives paralleles Computing.

Um IPython zu installieren, geben Sie Folgendes in einem Terminal oder in der PowerShell ein:

```
$ pip install ipython
```

bpython

bpython (http://bpython-interpreter.org/) ist eine alternative Schnittstelle zum Python-Interpreter für Unix-artige Betriebssysteme. Es bietet die folgenden Features:

- Inline-Syntaxhervorhebung.
- Automatische Einrückung und Vervollständigung.
- Liste der erwarteten Parameter für jede Python-Funktion.
- Eine »Rückspulfunktion«, die die letzte Codezeile aus dem Speicher abruft und erneut evaluiert.
- Die Möglichkeit, eingegebenen Code an ein Pastebin zu senden (um den Code online zu teilen).
- Die Möglichkeit, eingegebenen Code in einer Datei zu speichern.

Um bpython zu installieren, geben Sie in einem Terminal Folgendes ein:

```
$ pip install bpython
```

Isolationstools

Dieser Abschnitt geht etwas detaillierter auf die am häufigsten genutzten Isolationstools wie virtualenv (das Python-Umgebungen voneinander isoliert) und Docker (das ganze Systeme virtualisiert) ein.

Diese Tools bieten unterschiedliche Isolationsgrade zwischen den laufenden Anwendungen und der jeweiligen Hostumgebung. Damit wird es möglich, Code mit verschiedenen Python-Versionen und Bibliotheken zu testen und zu debuggen. Auf diese Weise können Sie eine konsistente Deployment-Umgebung aufbauen.

Virtuelle Umgebungen

Eine virtuelle Python-Umgebung hält die Abhängigkeiten verschiedener Projekte an unterschiedlichen Orten vor. Durch die Installation mehrerer Python-Umgebungen können Sie das *globale Paketverzeichnis* (in dem vom Benutzer installierte Python-Pakete abgelegt werden) sauber halten und an einem Projekt arbeiten, das beispielsweise Django 1.3 verlangt, während Sie außerdem ein Projekt pflegen können, das noch Django 1.0 nutzt.

Der Befehl `virtualenv` legt dazu einen separaten Ordner an, der einen Softlink zum Python-Executable enthält, eine Kopie des `pip`-Befehls sowie einen Ort für die Python-Bibliotheken. Dieser Ordner wird bei der Aktivierung dem aktuellen `PATH` vorangestellt. Bei Deaktivierung wird der ursprüngliche Zustand wiederhergestellt. Über Kommandozeilenoptionen ist es auch möglich, die vom System installierten Python- und Bibliotheksversionen zu nutzen.

Sie können eine virtuelle Umgebung nicht einfach verschieben, sobald sie einmal angelegt ist, da die Pfade der Executables alle fest codiert sind und auf den absoluten Pfad des Interpreters im *bin/*-Verzeichnis der virtuellen Umgebung verweisen.

Die virtuelle Umgebung anlegen und aktivieren

Das Setup und die Aktivierung virtueller Python-Umgebungen sind bei den verschiedenen Betriebssystemen leicht unterschiedlich.

Mac OS X und Linux. Sie können die gewünschte Python-Version mit dem Argument `--python` festlegen. Dann nutzen Sie das `activate`-Skript, um den `PATH` der virtuellen Umgebung festzulegen:

```
$ cd my-project-folder
$ virtualenv --python python3 my-venv
$ source my-venv/bin/activate
```

Windows. Falls noch nicht geschehen, müssen Sie die Ausführungs-Policy so anpassen, dass lokal erzeugte Skripte ausgeführt werden dürfen.[8] Hierzu öffnen Sie als Administrator eine PowerShell und geben Folgendes ein:

```
PS C:\> Set-ExecutionPolicy RemoteSigned
```

8 Oder verwenden Sie stattdessen `Set-ExecutionPolicy AllSigned`, sollten Sie das bevorzugen.

Beantworten Sie die erscheinende Frage mit `Y` und beenden Sie die Shell mit `exit`. In einer normalen PowerShell können Sie eine virtuelle Umgebung dann wie folgt erzeugen:

```
PS C:\> cd my-project-folder
PS C:\> virtualenv --python python3 my-venv
PS C:\> .\my-venv\Scripts\activate
```

Virtuelle Umgebungen um Bibliotheken ergänzen

Sobald Sie die virtuelle Umgebung aktiviert haben, ist das erste `pip`-Executable in ihrem Pfad dasjenige, das im gerade von Ihnen erzeugten *my-venv*-Ordner abgelegt wurde. Bibliotheken installiert es im folgenden Verzeichnis:

- *my-venv/lib/python3.4/site-packages/* (bei POSIX-Systemen[9])
- *my-venv\Lib\site-packages* (Windows)

Wenn Sie eigene Pakete oder Projekte für andere schnüren, geben Sie:

```
$ pip freeze > requirements.txt
```

ein, während die virtuelle Umgebung aktiv ist. Das schreibt alle aktuell installierten Pakete (die hoffentlich auch Projektabhängigkeiten sind) in die Datei *requirements.txt*. Nutzer können alle Abhängigkeiten in eigenen virtuellen Umgebungen installieren, indem sie die *requirements.txt* an `pip` übergeben:

```
$ pip install -r requirements.txt
```

`pip` installiert die aufgeführten Abhängigkeiten und setzt bei Konflikten die Abhängigkeiten der Unterpakete außer Kraft. Die in der *requirements.txt* angegebenen Abhängigkeiten sind für die gesamte Python-Umgebung gedacht. Sollen Abhängigkeiten bei der Distribution einer Bibliothek angegeben werden, verwenden Sie besser das Argument `install_requires` beim Aufruf der `setup()`-Funktion in einer *setup.py*-Datei.

Verwenden Sie `pip install -r requirements.txt` nicht außerhalb einer virtuellen Umgebung. Tun Sie es dennoch und in *requirements.txt* wird eine andere Version verlangt, als auf Ihrem Computer installiert ist, ersetzt `pip` die andere Version der Bibliothek durch die in der *requirements.txt* angegebene Version.

Die virtuelle Umgebung deaktivieren

Um die normalen Systemeinstellungen wiederherzustellen, geben Sie Folgendes ein:

```
$ deactivate
```

9 POSIX steht fürPortable Operating System Interface. Es besteht aus einem Satz von IEEE-Standards, der beschreibt, wie sich ein Betriebssystem verhalten soll. Das umfasst das Verhalten von und die Schnittstelle zu grundlegenden Shell-Befehlen, Ein-/Ausgabe, Threading sowie andere Dienste und Utilities. Die meisten Linux- und Unix-Distributionen werden als POSIX-kompatibel betrachtet, und Darwin (das Mac OS X und iOS zugrunde liegende Betriebssystem) ist seit Leopard (10.5) ebenfalls kompatibel. Ein »POSIX-System« ist ein System, das als POSIX-kompatibel betrachtet wird.

Weitere Informationen finden Sie in der *Dokumentation zu virtuellen Umgebungen (http://bit.ly/virtualenv-guide)*, der *offiziellen virtualenv-Dokumentation (https://virtualenv.pypa.io/en/latest/userguide.html)* und dem *offiziellen Python Packaging-Guide (https://packaging.python.org)*. Das pyvenv-Paket, das seit Python Version 3.3 Teil der Python-Standardbibliothek ist, ersetzt virtualenv nicht (tatsächlich ist es eine Abhängigkeit von virtualenv), das heißt, die Anweisungen funktionieren mit allen Versionen von Python.

pyenv

pyenv (https://github.com/yyuu/pyenv) ist ein Tool, das die gleichzeitige Nutzung verschiedener Python-Interpreter erlaubt. Damit lösen Sie das Problem, unterschiedliche Python-Versionen in verschiedenen Projekten verwenden zu müssen. Liegt allerdings ein Abhängigkeitskonflikt in den Bibliotheken vor (etwa wenn unterschiedliche Django-Versionen verwendet werden), müssen Sie weiterhin virtuelle Umgebungen nutzen. Zum Beispiel können Sie Python 2.7 installieren, um die Kompatibilität in einem Projekt zu erhalten, und gleichzeitig Python 3.5 als Standard-Interpreter verwenden. pyenv ist nicht auf die CPython-Versionen beschränkt – es installiert auch PyPy-, Anaconda-, Miniconda-, Stackless-, Jython- und IronPython-Interpreter.

pyenv befüllt dazu ein *shims*-Verzeichnis mit einer Shim-Version des Python-Interpreters und Programmen wie pip und 2to3. Diese Programme werden dann gefunden, wenn das Verzeichnis in der `$PATH`-Umgebungsvariablen vorangestellt wird. Ein *shim* ist eine »Durchleitefunktion«, die die aktuelle Situation interpretiert und die geeignetste Funktion auswählt, um die gewünschte Aufgabe zu erfüllen. Sucht das System beispielsweise nach einem Programm namens phyton, schaut es zuerst im *shims*-Verzeichnis nach und nutzt die shim-Version, die den Befehl wiederum an pyenv weitergibt. pyenv wählt dann die Python-Version basierend auf Umgebungsvariablen, **.python-version*-Dateien und den globalen Standardeinstellungen.

Für virtuelle Umgebungen gibt es das Plug-in *pyenv-virtualenv (https://github.com/yyuu/pyenv-virtualenv)*, das das Anlegen virtueller Umgebungen automatisiert und es erlaubt, die vorhandenen pyenv-Tools zum Wechsel in verschiedene Umgebungen zu nutzen.

Autoenv

Autoenv (https://github.com/kennethreitz/autoenv) bietet abseits von virtualenv eine leichtgewichtige Möglichkeit zur Verwaltung verschiedener virtueller Umgebungen. Es überschreibt den Shell-Befehl `cd` so, dass, wenn beim Wechsel in ein Verzeichnis eine *.env*-Datei gefunden wird (die z. B. den `PATH` und eine Umgebungsvariable mit einer Datenbank-URL setzt), diese Umgebung automatisch aktiviert wird. Sobald Sie das Verzeichnis per `cd` wieder verlassen, wird der Effekt aufgehoben. Bei der Windows PowerShell funktioniert das leider nicht.

Unter Mac OS X installieren Sie es mit `brew`:

```
$ brew install autoenv
```

und bei Linux mit:

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

Öffnen Sie dann eine neue Shell.

virtualenvwrapper

virtualenvwrapper (http://bit.ly/virtualenvwrapper-docs) bietet eine Reihe von Befehlen an, die virtuelle Python-Umgebungen erweitern, um bessere Kontroll- und Verwaltungsmöglichkeiten zu bieten. Es platziert alle virtuellen Umgebungen in ein einziges Verzeichnis und stellt leere Hook-Funktionen zur Verfügung, die vor oder nach der Erzeugung bzw. Aktivierung der virtuellen Umgebung oder eines Projekts ausgeführt werden können. Ein solcher Hook kann beispielsweise Umgebungsvariablen setzen, indem er die *.env*-Datei innerhalb des Verzeichnisses verarbeitet.

Das Problem mit der Platzierung solcher Funktionen mit den installierten Elementen besteht darin, dass der Benutzer irgendwie an diese Skripte gelangen muss, um die Umgebung auf einer anderen Maschine vollständig duplizieren zu können. Dennoch kann es auf einem gemeinsam genutzten Entwicklungsserver nützlich sein, wenn alle Umgebungen in einem gemeinsam genutzten Ordner abgelegt sind und von mehreren Nutzern verwendet werden.

Um die *vollständigen virtualenvwrapper-Installationsanweisungen (http://bit.ly/virtualenvwrapper-install)* zu überspringen, stellen Sie zuerst sicher, dass virtualenv bereits installiert ist. Danach geben Sie unter OS X oder Linux in einem Terminal Folgendes ein:

```
$ pip install virtualenvwrapper
```

Oder verwenden Sie `pip install virtualenvwrapper`, wenn Sie mit Python 2 arbeiten. Tragen Sie dann

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
```

in Ihre *~/.profile* ein und ergänzen Sie Ihre *~/.bash_profile* (oder das von Ihnen bevorzugte Shell-Profil) wie folgt:

```
source /usr/local/bin/virtualenvwrapper.sh
```

Zum Abschluss beenden Sie das aktuelle Terminalfenster und öffnen ein neues, um das neue Profil zu aktivieren. Danach steht virtualenvwrapper zur Verfügung.

Unter Windows verwenden Sie stattdessen *virtualenvwrapper-win (http://bit.ly/virtualenvwrapper-win)*. Wenn `virtualenv` bereits installiert ist, geben Sie Folgendes ein:

```
PS C:\> pip install virtualenvwrapper-win
```

Die folgenden Befehle werden auf beiden Plattformen häufig genutzt:

`mkvirtualenv` *`my_venv`*
: Erzeugt eine virtuelle Umgebung im Ordner *~/.virtualenvs/my_venv*. Unter Windows wird *my_venv* in dem Verzeichnis angelegt, das Sie in der Kommandozeile durch Eingabe von `%USERPROFILE%\Envs` ermitteln können. Die genaue Position kann über die Umgebungsvariable `$WORKON_HOME` angepasst werden.

`workon` *`my_venv`*
: Aktiviert die virtuelle Umgebung oder wechselt aus der aktuellen Umgebung in die angegebene.

`deactivate`
: Deaktiviert die virtuelle Umgebung.

`rmvirtualenv` *`my_venv`*
: Löscht die virtuelle Umgebung.

virtualenvwrapper bietet eine Tabulatorvervollständigung für Umgebungsnamen, was in vielen Umgebungen (und wenn Sie sich die Namen nicht merken können) hilfreich ist. Eine Reihe weiterer nützlicher Funktionen ist in der *Liste der virtualenvwrapper-Befehle (http://bit.ly/virtualenvwrapper-command)* dokumentiert.

Buildout

Buildout (http://www.buildout.org/en/latest/) ist ein Python-Framework, das das Anlegen und Aufbauen sogenannter *Rezepte* (engl. Recipes) erlaubt. Dabei handelt es sich um Python-Module mit beliebigem Code (üblicherweise Systemaufrufen zum Erzeugen von Verzeichnissen oder zum Checkout und Build von Quellcode sowie dem Einbinden der Nicht-Python-Teile eines Projekts wie etwa einer Datenbank oder einem Webserver). Sie installieren es mittels `pip`:

```
$ pip install zc.buildout
```

Buildout nutzende Projekte nehmen `zc.buildout` und die benötigten Rezepte in der *requirements.txt* auf oder würden eigene Rezepte direkt mit dem Quellcode liefern. Sie beinhalten auch die Konfigurationsdatei *buildout.cfg* und das Skript *bootstrap.py* im obersten Verzeichnis. Rufen Sie das Skript mit `python bootstrap.py` auf, wird durch Einlesen der Konfigurationsdatei ermittelt, welche Rezepte verwendet werden sollen, sowie die Konfigurationsoptionen der jeweiligen Rezepte (z. B. die zu verwendenden Kompilier- und Linker-Flags).

Einem Python-Projekt mit Nicht-Python-Teilen verleiht Buildout Portabilität – ein anderer Benutzer kann die gleiche Umgebung rekonstruieren. Das unterscheidet sich von den Skript-Hooks bei virtualenvwrapper, die zusammen mit der *requirements.txt* kopiert und übertragen werden müssen, um die virtuelle Umgebung wiederherstellen zu können.

Buildout umfasst Teile zur Installation von Eggs.[10] Bei neueren Python-Versionen können Sie das überspringen, da hier mit wheels gearbeitet wird. Weitere Informationen finden Sie im *Buildout-Tutorial (http://www.buildout.org/en/latest/docs/tutorial.html)*.

Conda

Conda (http://conda.pydata.org/docs/) ist wie pip, virtualenv und Buildout in einem. Es wird mit der Python-Distribution Anaconda geliefert und ist Anacondas Standardpaketmanager. Es kann über `pip` installiert werden:

```
$ pip install conda
```

Und `pip` kann über `conda` installiert werden:

```
$ conda install pip
```

Die Pakete liegen in unterschiedlichen Repositories (`pip` bezieht von *http://pypi.python.org (http://pypi.python.org)* und `conda` von *https://repo.continuum.io/*). Da mit unterschiedlichen Formaten gearbeitet wird, sind die Tools nicht austauschbar.

Diese von Continuum (den Schöpfern von Anaconda) entwickelte *Tabelle (http://bit.ly/conda-pip-virtualenvl)* vergleicht die drei Optionen `conda`, `pip` und `virtualenv` miteinander.

conda-build, Continuums Gegenstück zu Buildout, kann auf allen Plattformen wie folgt installiert werden:

```
conda install conda-build
```

Wie bei Buildout wird auch das Format der conda-build-Konfigurationsdatei als »Rezept« bezeichnet, und diese Rezepte sind nicht auf die Verwendung von Python-Tools beschränkt. Im Gegensatz zu Buildout wird der Code als Shell-Skript angegeben (nicht in Python), und die Konfiguration erfolgt in YAML,[11] und nicht in Pythons *ConfigParser-Format (https://docs.python.org/3/library/configparser.html)*.

Der eigentliche Vorteil von Conda gegenüber pip und virtualenv liegt aufseiten der Windows-Nutzer: Als C-Erweiterungen generierte Python-Bibliotheken können als wheels vorhanden sein – oder auch nicht, doch sie stehen fast immer im *Anaconda-Paketindex (http://docs.continuum.io/anaconda/pkg-docs)*. Und falls ein Paket nicht

10 Ein *Egg* (zu Deutsch »Ei«) ist eine ZIP-Datei mit einer spezifischen Struktur, die Distributionsinhalte enthält. Eggs wurden mit *PEP 427 (https://www.python.org/dev/peps/pep-0427/)* durch wheels ersetzt. Sie wurden durch die sehr beliebte Packaging-Bibliothek Setuptools eingeführt, die eine praktische Schnittstelle zu den *distutils (https://docs.python.org/3/library/distutils.html)* der Python-Standardbibliothek bietet. Alles über die Unterschiede zwischen den Formaten finden Sie in *Wheel vs Egg (https://packaging.python.org/en/latest/wheel_egg/)* im Python Packaging User Guide.

11 *YAML (https://en.wikipedia.org/wiki/YAML)* (YAML Ain't Markup Language) ist eine Markup-Sprache, die gleichermaßen von Menschen und Maschinen gelesen werden kann.

über Conda verfügbar ist, können Sie pip installieren und auf *PyPI (https://pypi.python.org/pypi)* gehostete Pakete installieren.

Docker

Docker (https://www.docker.com/) isoliert Umgebungen genau wie virtualenv, Conda oder Buildout, doch anstelle einer virtuellen Umgebung stellt es einen *Docker-Container* zur Verfügung. Container bieten eine größere Isolation als virtuelle Umgebungen. Zum Beispiel können Sie Container betreiben, die jeweils eigene Netzwerkschnittstellen, Firewall-Regeln und Hostnamen besitzen. Die laufenden Container werden von einem separaten Utility, der *Docker Engine (https://docs.docker.com/engine/)*, verwaltet, die den Zugriff auf das darunterliegende Betriebssystem koordiniert. Wenn Sie Docker-Container unter OS X, Windows oder auf einem entfernten Host laufen lassen, benötigen Sie auch die *Docker Machine (https://docs.docker.com/machine/)*, die die Schnittstelle zu der virtuellen Maschine[12] herstellt, die die Docker-Engine ausführt.

Docker-Container basierten zunächst auf Linux-Containern, die ursprünglich selbst auf dem Shell-Befehl `chroot` *(https://en.wikipedia.org/wiki/Chroot)* basierten. `chroot` ist eine Art auf Systemebene angesiedelte Version des `virtualenv`-Befehls: Es lässt das Root-Verzeichnis (*/*) in einem vom Benutzer festgelegten Pfad erscheinen und nicht im echten Root-Verzeichnis. Auf diese Weise können Sie einen völlig separaten *User Space (https://en.wikipedia.org/wiki/User-space)* bereitstellen.

Docker nutzt `chroot` nicht und mittlerweile auch keine Linux-Container mehr (was das Universum von Docker-Images um Citrix und Solaris-Maschinen erweitert hat), doch die Docker-Container machen immer noch das Gleiche. Die Konfigurationsdateien werden als *Dockerfiles (https://docs.docker.com/engine/reference/builder/)* bezeichnet, die *Docker-Images (https://docs.docker.com/engine/userguide/containers/dockerimages/)* erzeugen, die auf dem *Docker-Hub (https://docs.docker.com/docker-hub/)* gehostet werden, dem Docker-Paket-Repository (vergleichbar mit PyPI).

Docker-Images benötigen, wenn sie richtig konfiguriert sind, weniger Platz als von Buildout oder Conda erzeugte Umgebungen, weil Docker das *AUFS (https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/)*-Dateisystem nutzt, das nur das »diff« eines Images speichert und nicht das gesamte Image. Wenn Sie beispielsweise ein Paket für mehrere Releases einer Abhängigkeit generieren und testen wollen, können Sie ein Basis-Docker-Image erzeugen, das eine virtuelle Umge-

12 Eine *virtuelle Maschine* ist eine Anwendung, die auf einem Hostcomputer ein Computersystem emuliert, indem es die gewünschte Hardware imitiert und das gewünschte Betriebssystem bereitstellt.

bung[13] (oder Buildout- oder Conda-Umgebung) mit allen anderen Abhängigkeiten enthält. Sie vererben dann diese Basis an alle anderen Images und ändern nur die jeweilige Abhängigkeit in der letzten Schicht. So enthalten alle abgeleiteten Images nur die jeweils neue Bibliothek, während sie sich den Inhalt des Basis-Images teilen. Weitere Informationen finden Sie in der *Docker-Dokumentation (https://docs.docker.com/)*.

13 Eine virtuelle Umgebung innerhalb eines Docker-Containers isoliert Ihre Python-Umgebung und konserviert das Python des Betriebssystems für die Utilities, die zur Unterstützung Ihrer Anwendung notwendig sein könnten. Befolgen Sie also unseren Rat, nichts über `pip` (oder mit etwas anderem) im Python-Verzeichnis des Systems zu installieren.