4 Code wiederverwenden

Funktionen und Module

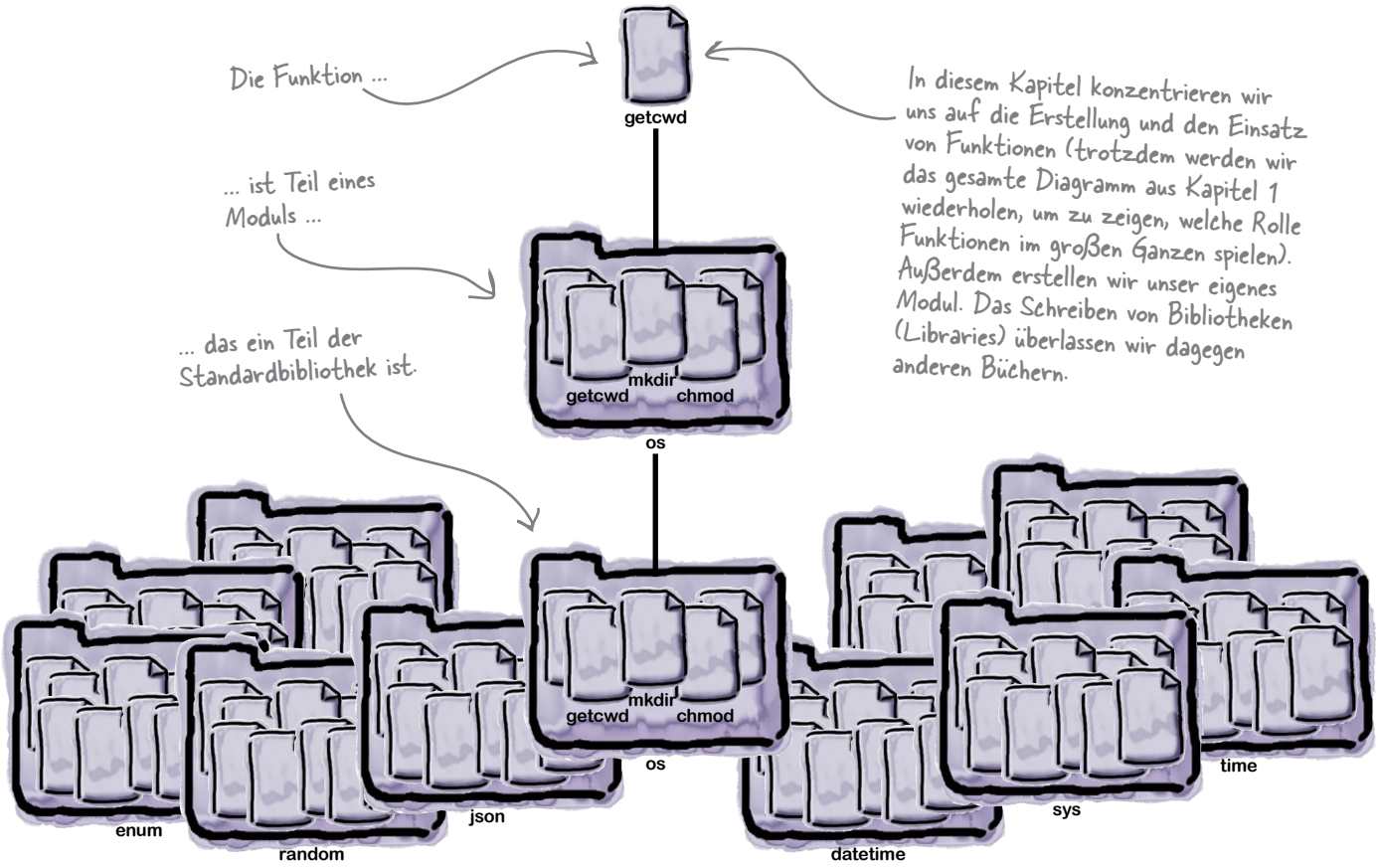
Die Wiederverwendbarkeit von Code ist Voraussetzung für ein wartbares System.

Und bei Python ist Anfang und Ende aller Wiederverwendbarkeit die **Funktion**. Nehmen Sie ein paar Codezeilen, geben Sie ihnen einen Namen, und schon haben Sie eine (wiederverwendbare) Funktion. Nehmen Sie eine Sammlung von Funktionen und packen Sie sie in eine eigene Datei, und schon haben Sie ein **Modul** (das ebenfalls wiederverwendet werden kann). Es stimmt schon: *Teilen hilft*. Am Ende dieses Kapitels werden Sie wissen, wie Code mithilfe von Pythons Funktionen und Modulen **wiederverwendet** und **mit anderen geteilt** werden kann.

Code mithilfe von Funktionen wiederverwenden

In Python können Sie schon mit ein paar Codezeilen eine Menge bewirken. Dennoch wird die Codebasis Ihres Programms beständig wachsen, und mit der Größe wächst auch der Wartungsaufwand. 20 Zeilen Code können sich schnell zu einem 500-Zeilen-Monstrum entwickeln! Dann ist es Zeit, über Strategien nachzudenken, um die Komplexität der Codebasis zu verringern.

Wie viele andere Programmiersprachen unterstützt auch Python das Konzept der **Modularität**, wodurch Sie Ihren Code in kleinere, leichter wartbare Stücke aufteilen können. Das Mittel zum Zweck sind **Funktionen**, die Sie sich als benannte Codeabschnitte vorstellen können. Sehen Sie sich hierzu noch einmal das Diagramm aus Kapitel 1 an, das die Beziehungen zwischen Funktionen, Modulen und der Standardbibliothek darstellt:



In diesem Kapitel liegt unser Fokus auf den Dingen, die für die Erstellung von Funktionen wichtig sind, wie ganz oben im Diagramm zu sehen. Sobald Sie mit dem Bau von Funktionen vertraut sind, werden wir Ihnen außerdem zeigen, wie Module angelegt werden.

Einführung in Funktionen

Bevor wir unseren bereits geschriebenen Code in eine Funktion verwandeln können, wollen wir sehen, wie eine Python-Funktion *im Allgemeinen* aufgebaut ist. Nach dieser Einführung werden wir anhand des bereits geschriebenen Codes (zumindest eines Teils davon) die Schritte erörtern, die für die Erstellung einer wiederverwendbaren Funktion nötig sind.

Machen Sie sich jetzt noch keinen Kopf über die Details. Es geht hier und auf den folgenden Seiten zunächst nur darum, ein Gefühl für das Aussehen von Python-Funktionen zu bekommen. Alle weiteren Details zeigen wir im Verlauf dieses Kapitels. Das IDLE-Fenster auf dieser Seite zeigt eine Funktionsvorlage, die Sie benutzen können, um neue Funktionen zu erstellen. Bedenken Sie beim Betrachten der Abbildung die folgenden Dinge:

1. **Funktionen verwenden zwei neue Schlüsselwörter: def und return.**
Beide Schlüsselwörter werden in IDLE orange hervorgehoben. Das Schlüsselwort `def` benennt die Funktion (in Blau) und dient zur Angabe möglicher Argumente. Das Schlüsselwort `return` ist optional, es wird benutzt, um einen Wert an den aufrufenden Code zurückzugeben.

2. **Funktionen können Daten aus Argumenten übernehmen.**
Eine Funktion kann Daten aus Argumenten übernehmen (z. B. die Parameter oder Eingaben für die Funktion). Innerhalb der runden Klammern auf der `def`-Zeile können Sie nach dem Funktionsnamen eine Liste mit Argumenten angeben.

3. **Funktionen enthalten Code und (üblicherweise) Dokumentation.**
Unterhalb der `def`-Zeile wird der Code einer Funktion um eine Ebene eingerückt. Wo es sinnvoll ist, sollten Kommentare eingefügt werden. Wir zeigen Ihnen hier zwei Möglichkeiten, den Code zu kommentieren: anhand von dreifachen Anführungszeichen (`"""`, in der Vorlage grün markiert und üblicherweise als **Docstring** bezeichnet) und anhand eines einzeiligen Kommentars, dem ein (unten in Rot gezeigten) #-Zeichen vorangestellt wird.

Eine praktische Vorlage für Funktionen.

Die »def«-Zeile benennt die Funktion und listet Ihre Argumente auf.

Der »Docstring« beschreibt den Zweck der Funktion.

Hier kommt Ihr Code hin (anstelle der einzeiligen Kommentar-Platzhalter).

function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

```
def a_descriptive_name(optional_arguments):
    """A documentation string."""
    # Your function's code goes here.
    # Your function's code goes here.
    # Your function's code goes here.
    return optional_value
```

Ln: 8 Col: 0

Geek-Bits

In Python wird wiederverwendbarer Code als **Funktion** (Function) bezeichnet. Andere Programmiersprachen verwenden hierfür Namen wie »Prozedur«, »Subroutine« und »Methode«. Ist eine Funktion Teil einer Python-Klasse, wird sie als »Methode« bezeichnet. Mehr zu Pythons Klassen und Methoden zeigen wir Ihnen in einem anderen Kapitel.

Was ist mit Informationen zum Datentyp?

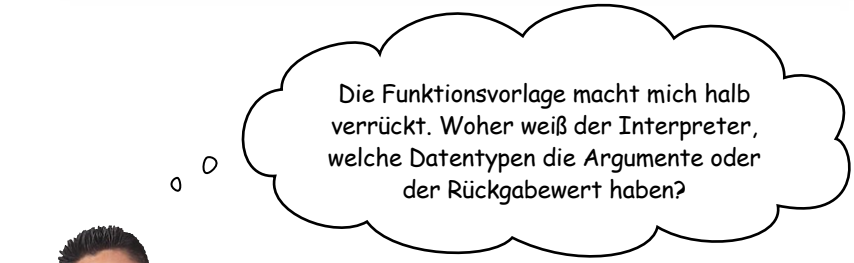
Sehen Sie sich unsere Funktionsvorlage noch einmal an. Fehlt da, abgesehen von ausführbarem Code, nicht noch etwas? Gibt es etwas, das man noch angeben sollte? Sehen Sie noch einmal hin:

function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

```
def a_descriptive_name(optional_arguments):
    """A documentation string."""
    # Your function's code goes here.
    # Your function's code goes here.
    # Your function's code goes here.
    return optional_value
```

Ln: 8 Col: 0

Fehlt bei der Funktionsvorlage noch etwas?



Das weiß er nicht, aber keine Sorge.

Der Python-Interpreter zwingt Sie nicht, die Datentypen Ihrer Funktionsargumente oder einen Rückgabewert anzugeben. Je nachdem, welche Programmiersprachen Sie vorher benutzt haben, irritiert Sie das vielleicht – sollte es aber nicht!

In Python können Sie ein beliebiges *Objekt* als Argument übergeben und ein beliebiges *Objekt* als Rückgabewert verwenden. Dem Interpreter ist es dabei vollkommen egal, welchen Datentyp diese Objekte haben. Er überprüft sie nicht einmal (sondern nur, ob sie angegeben werden).

Ab Python 3 ist es möglich, *anzugeben*, welche Datentypen für Argumente und Rückgabewerte *erwartet* werden, und das werden wir später in diesem Kapitel auch tun. Allerdings sorgt die Angabe des Typs *nicht* dafür, dass die Überprüfung auf »magische« Weise aktiviert wird, da Python die Typen der Argumente bzw. Rückgabewerte *nie* überprüft.

Codestücke mit »def« einen Namen geben

Nachdem Sie ein wiederverwendbares Codestück gefunden haben, ist es Zeit, daraus eine Funktion zu machen. Das geschieht, indem Sie das Schlüsselwort `def` (kurz für *definiere*) verwenden, gefolgt vom Namen der Funktion und einer (optionalen, hier leeren) Argumentliste in runden Klammern, einem Doppelpunkt sowie einer oder mehrerer Zeilen eingerückten Codes.

Das Programm `vowels7.py` vom Ende des letzten Kapitels übernahm ein Wort und gab dann die darin enthaltenen Vokale aus:

Das hier ist »vowels7.py« vom Ende des Kapitels 3.

Nehmen Sie eine Gruppe Vokale ...

... sowie ein Wort ...

... und bilden Sie dann die Schnittmenge.

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Ergebnisse ausgeben.

Stellen Sie sich vor, Sie wollten diese fünf Codezeilen in ein viel größeres Programm einbauen. Auf keinen Fall sollten Sie den Code an jeder Stelle, an der er gebraucht wird, von Hand einfügen. Stattdessen wollen wir eine **Funktion** erstellen. So bleibt der Code wartbar, und Sie müssen nur **eine Kopie** des Codes pflegen.

Lassen Sie sich Zeit, um einen guten und beschreibenden Namen für die Funktion zu finden.

Wir zeigen Ihnen (im Moment noch) auf der Python-Shell, wie das geht. Um die obigen fünf Codezeilen in eine Funktion umzuwandeln, brauchen Sie das Schlüsselwort `def`, um den Beginn der Funktion zu markieren. Geben Sie der Funktion einen möglichst beschreibenden Namen (*immer* eine gute Idee!). Geben Sie optional in runden Klammern eine leere Liste mit Argumenten an. Danach rücken Sie die folgenden Zeilen relativ zum `def`-Schlüsselwort ein, wie hier gezeigt:

Geben Sie Ihrer Funktion einen hübschen, beschreibenden Namen.

Geben Sie eine optionale Argumentliste an – hier hat die Funktion keine Argumente, und die Liste bleibt leer.

Beginnen Sie mit dem Schlüsselwort »def«.

Vergessen Sie nicht den Doppelpunkt.

```
>>> def search4vowels():
        vowels = set('aeiou')
        word = input("Provide a word to search for vowels: ")
        found = vowels.intersection(set(word))
        for vowel in found:
            print(vowel)
```

Die fünf Codezeilen aus »vowels7.py«, korrekt eingerückt.

Da dies die Shell ist, müssen Sie die Enter-Taste ZWEIMAL drücken, um anzuzeigen, dass der eingerückte Code beendet ist.

Jetzt ist die Funktion fertig, und wir können sie aufrufen, um zu sehen, ob sie wie erwartet funktioniert.

Rufen Sie Ihre Funktionen auf

Um in Python eine Funktion aufzurufen, geben Sie den Funktionsnamen sowie die Werte der möglicherweise erwarteten Argumente an. Da die `search4vowels`-Funktion (im Moment) keine Argumente erwartet, können wir sie mit einer leeren Argumentliste aufrufen:

```
>>> search4vowels()
Provide a word to search for vowels: hitch-hiker
e
i
```

Ein erneuter Aufruf der Funktion führt sie ein weiteres Mal aus:

```
>>> search4vowels()
Provide a word to search for vowels: galaxy
a
```

Hier gibt es keine Überraschungen: Der Funktionsaufruf führt den enthaltenen Code aus.

Editieren Sie die Funktion in einem Editorfenster und nicht am >>>-Prompt.

Zu diesem Zeitpunkt haben wir den Code der `search4vowels`-Funktion am >>>-Prompt eingegeben. Aktuell sieht er so aus:

Wir haben unsere Funktion am >>>-Prompt eingegeben.

```
>>> def search4vowels():
        vowels = set('aeiou')
        word = input("Provide a word to search for vowels: ")
        found = vowels.intersection(set(word))
        for vowel in found:
            print(vowel)
```

Um mit diesem Code weiterzuarbeiten, können Sie ihn am >>>-Prompt aufrufen und bearbeiten. Das wird aber schnell unpraktisch. Besser ist es, Sie kopieren Ihren Code bei der Arbeit mit dem >>>-Prompt in ein Editorfenster, sobald er mehr als ein paar Zeilen lang ist. Dort lässt er sich wesentlich einfacher bearbeiten. Das wollen wir tun, bevor wir weitermachen.

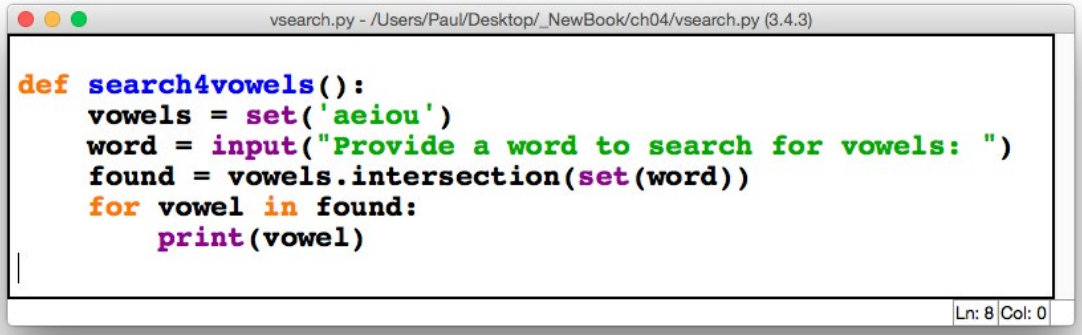
Öffnen Sie in IDLE ein neues leeres Editorfenster und kopieren Sie den Funktionscode vom >>>-Prompt hinein (ohne dabei die >>>-Zeichen mitzukopieren!). Sobald die Formatierung Ihren Vorstellungen entspricht, sollten Sie die Datei als `vsearch.py` speichern, bevor Sie weitermachen.

Vergessen Sie nicht, Ihren Code nach dem Kopieren des Funktionscodes von der Shell als »vsearch.py« zu speichern.

Den IDLE-Editor für Änderungen verwenden

Die Datei `vsearch.py` sieht in IDLE jetzt so aus:

Der Funktionscode befindet sich nun in einem IDLE-Editorfenster und wurde als >>vsearch.py<< gespeichert.



Keine Panik, wenn IDLE beim Drücken von F5 einen Fehler ausgibt! Überprüfen Sie im Editorfenster noch einmal, ob Ihr Code genauso aussieht wie in unserem Beispiel und versuchen Sie es ein weiteres Mal.

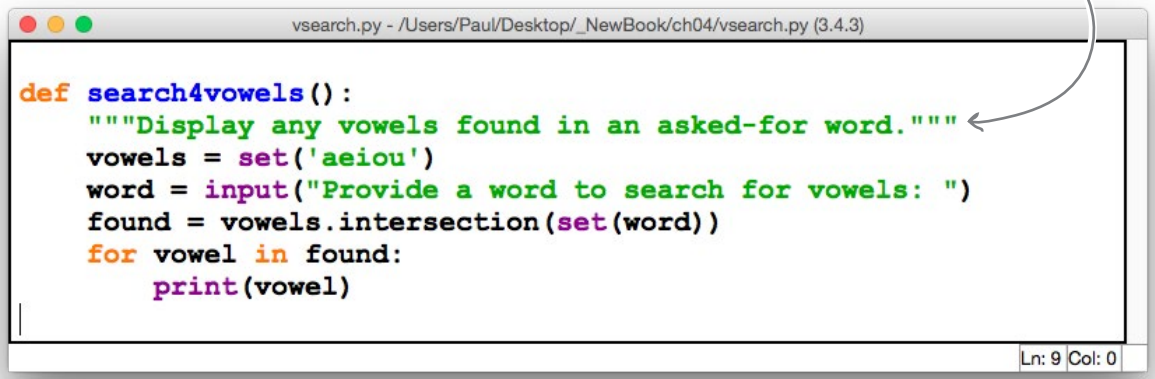
Wenn Sie im Editorfenster F5 drücken, passieren zwei Dinge: Die IDLE-Shell kommt in den Vordergrund und wird neu gestartet. Auf dem Bildschirm passiert dabei noch nichts. Um zu sehen, was wir meinen, drücken Sie jetzt bitte F5.

Es gibt noch keine Ausgaben, weil Sie die Funktion zuerst aufrufen müssen, was wir auch gleich tun werden. Zuvor wollen wir noch eine kleine, aber wichtige Änderung an der Funktion vornehmen.

Um Code mit einem mehrzeiligen Kommentar (einem **Docstring**) zu versehen, umgeben Sie ihn mit dreifachen Anführungszeichen.

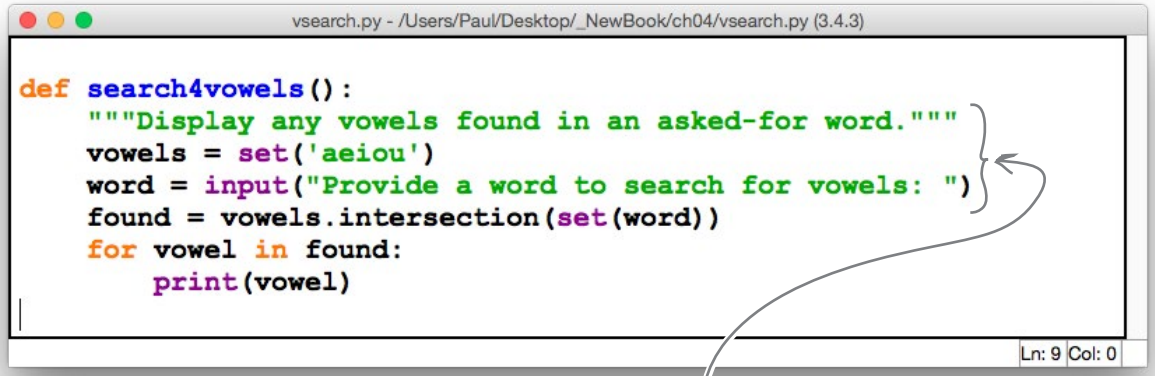
Hier sehen Sie noch einmal die Datei `vsearch.py`, diesmal allerdings mit dem neuen Docstring am Anfang der Funktion. Bauen Sie die Änderung gleich in Ihren Code ein:

Wir haben dem Code der Funktion einen Docstring hinzugefügt, der den Zweck der Funktion (kurz) erklärt.



Was ist mit den ganzen Strings los?

Sehen Sie sich die aktuelle Funktion noch einmal an. Achten Sie dabei besonders auf die drei Strings in diesem Code, die in IDLE alle grün markiert sind:



Das Syntax-Highlighting von IDLE zeigt, dass wir ein Konsistenzproblem bei der Verwendung der Anführungszeichen für unseren String haben. Wann benutzen wir welchen Stil?

Die Anführungszeichen für Strings verstehen

In Python können Strings mit einfachen (`'`), doppelten (`"`) oder dreifachen Anführungszeichen (`"""` oder `'''`) umgeben werden.

Wie bereits erwähnt, werden dreifache Anführungszeichen auch als **Docstring** bezeichnet, da sie hauptsächlich zur Dokumentation des Funktionszwecks benutzt werden (siehe oben). Obwohl Sie Ihre Docstrings mit `"""` oder `'''` umgeben können, bevorzugen die meisten Python-Programmierer `"""`. Docstrings können sich über mehrere Zeilen erstrecken (andere Programmiersprachen verwenden den Begriff »Heredoc« für das gleiche Konzept).

Strings, die mit einfachen (`'`) oder doppelten Anführungszeichen (`"`) umgeben sind, müssen dagegen auf einer Zeile stehen. Der String muss mit einem passenden Anführungszeichen *auf der gleichen Zeile* beendet werden (weil Python das Zeilenende als Ende einer Anweisung interpretiert).

Sie können selbst entscheiden, welches Anführungszeichen Sie verwenden, wobei die Mehrheit der Python-Programmierer einfache Anführungszeichen bevorzugt. Das Wichtigste ist allerdings, dass Sie in der Verwendung vor allem konsistent bleiben.

Der Code am Anfang dieser Seite nutzt Anführungszeichen dagegen *nicht* konsistent (obwohl es nur ein paar Zeilen sind). Der Code wird trotzdem problemlos ausgeführt (weil es dem Interpreter egal ist, welchen Stil Sie benutzen). Trotzdem kann die Vermischung verschiedener Stile der Lesbarkeit Ihres Code schaden (was eine Schande wäre).

Seien Sie bei der Wahl Ihrer Anführungszeichen konsistent. Verwenden Sie nach Möglichkeit einfache Anführungszeichen (`'`).

Folgen Sie den Best Practices der PEPs

Die Python-Gemeinschaft hat viel Zeit damit verbracht, die besten Vorgehensweisen für die Formatierung von Code (nicht nur von Strings) zu etablieren und zu dokumentieren. Diese beste Vorgehensweise ist als **PEP 8** bekannt. Hierbei steht PEP für »Python Enhancement Protocol« (»Protokoll zur Verbesserung von Python«).

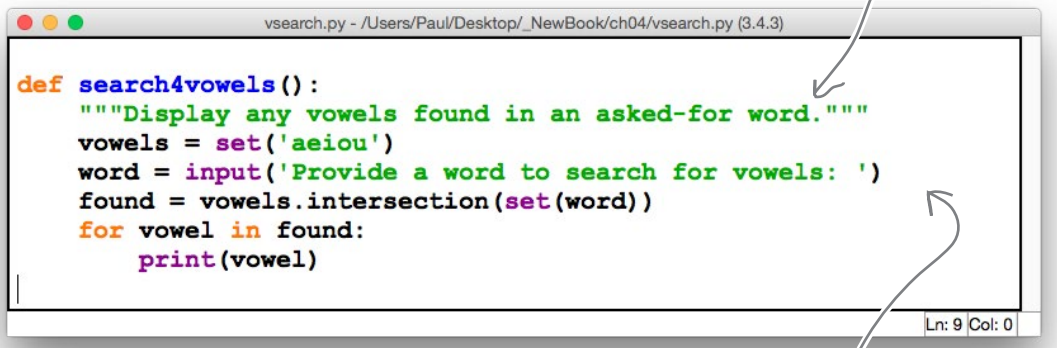
Es existiert eine Vielzahl von PEP-Dokumenten. Größtenteils geht es um vorgeschlagene und implementierte Verbesserungen von Python als Programmiersprache. Sie dokumentieren aber auch eine Reihe von Ratschlägen (was man tun sollte und was besser nicht) und verschiedene Python-Vorgehensweisen. Die Details der PEP-Dokumente sind oft reichlich technisch und wirken häufig sogar esoterisch. Daher sind sich die meisten Python-Programmierer dieser Dokumente zwar bewusst, kümmern sich aber nicht um jedes Detail. Das gilt für die meisten PEPs *außer* PEP 8.

Eine Liste aller PEPs finden Sie hier: https://www.python.org/dev/peps/.

PEP 8 enthält *die* Stilrichtlinien für Python-Code. Es wird allen Python-Programmierern empfohlen, sie zu lesen. Dies ist das Dokument, das empfiehlt: »Seien Sie konsistent!«, wenn es um die auf der vorigen Seite beschriebenen Anführungszeichen um Strings geht. Nehmen Sie sich die Zeit, PEP 8 zumindest einmal durchzulesen. Ein weiteres Dokument namens **PEP 257** stellt eine Reihe von Konventionen für die Formatierung von Docstrings vor und ist ebenfalls sehr lesenswert.

Hier sehen Sie die `search4vowels`-Funktion erneut in ihrer PEP 8- und PEP 257-konformen Schreibweise. Die Änderungen sind nicht weltbewegend, aber standardisierte einfache Anführungszeichen für Strings (jedoch nicht um unsere Docstrings) sehen einfach besser aus:

Dies ist ein PEP 257-konformer Docstring.



Wir haben den PEP 8-Rat befolgt und unsere Strings durchgehend mit einfachen Anführungszeichen umgeben.

Natürlich muss Ihr Code sich nicht exakt an die PEP 8-Regeln halten. So entspricht unser Funktionsname `search4vowels` beispielsweise nicht den Richtlinien, nach denen die Wörter in Funktionsnamen durch Unterstriche getrennt werden sollten. Passender wäre eigentlich `search_for_vowels`. PEP 8 ist eher eine Sammlung von Richtlinien als eine Reihe von Regeln. Sie müssen sich nicht daran halten, aber Sie sollten es in Erwägung ziehen. In diesem Fall bleiben wir also bei `search4vowels`.

Trotzdem wird die große Mehrheit der Python-Programmierer Ihnen sehr dankbar sein, wenn Sie PEP 8-konformen Code schreiben, weil er schlicht leichter zu lesen ist.

Jetzt wollen wir die `search4vowels`-Funktion so erweitern, dass sie Argumente übernehmen kann.

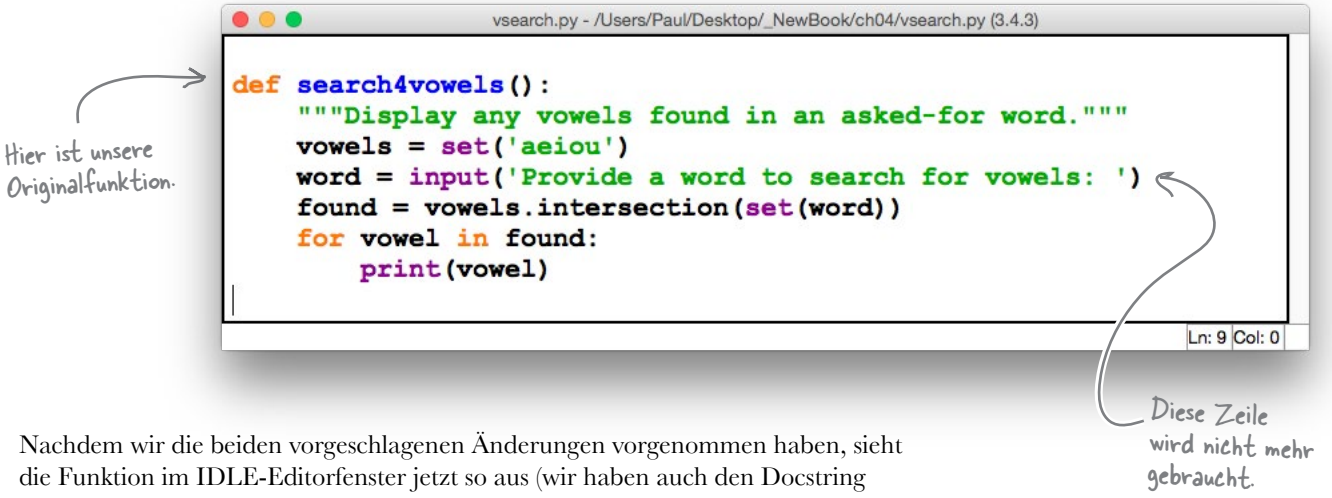
Funktionen können Argumente übernehmen

Wir wollen `search4vowels` so abändern, dass die Funktion den Benutzer nicht mehr zur Eingabe eines Worts auffordert. Stattdessen soll das Wort als Eingabe für ein Argument verwendet werden.

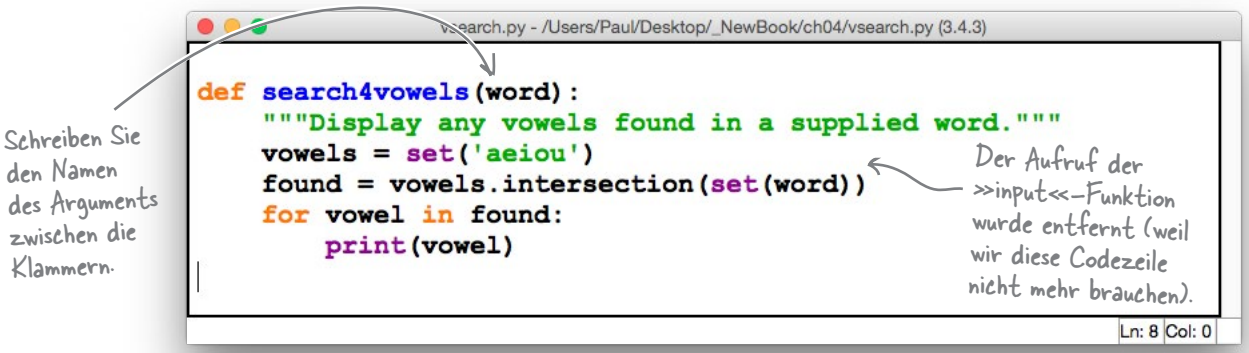
Das Hinzufügen eines Arguments geht ganz einfach: Sie geben auf der `def`-Zeile einfach den Namen des Arguments an und umgeben ihn mit runden Klammern. Der Argumentname kann danach innerhalb der Suite als Variable benutzt werden. Diese Änderung war leicht.

Außerdem wollen wir die Codezeile entfernen, die den Benutzer zur Eingabe eines Worts auffordert. Auch diese Änderung ist einfach. Werfen wir einen Blick auf den aktuellen Zustand unseres Codes:

Nicht vergessen: »Suite« ist Python-Slang für »Block«.



Nachdem wir die beiden vorgeschlagenen Änderungen vorgenommen haben, sieht die Funktion im IDLE-Editorfenster jetzt so aus (wir haben auch den Docstring aktualisiert, was *immer* eine gute Idee ist):



Vergessen Sie nicht, Ihre Datei nach jeder Änderung am Code zu sichern, bevor Sie F5 drücken, um die neue Version Ihrer Funktion zu testen.

PROBEFAHRT

Nachdem Sie den Code ins Editorfenster von IDLE geladen (und dort gespeichert) haben, drücken Sie F5. Dann rufen Sie die Funktion ein paarmal auf, um zu sehen, was passiert:

Der aktuelle >>search4vowels<<-Code.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
>>> search4vowels()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    search4vowels()
TypeError: search4vowels() missing 1 required positional argument: 'word'
>>> search4vowels('hitch-hiker')
e
i
>>> search4vowels('hitch-hiker', 'galaxy')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    search4vowels('hitch-hiker', 'galaxy')
TypeError: search4vowels() takes 1 positional argument but 2 were given
>>>
```

Ln: 12 Col: 4

Obwohl wir >>search4vowels<< für diese Probefahrt dreimal aufgerufen haben, wurde nur die Version, der wir einen String in Anführungszeichen als Argument übergeben haben, erfolgreich ausgeführt. Die anderen sind fehlgeschlagen. Lesen Sie die Fehlermeldungen sorgfältig durch, um zu sehen, was hier schiefgelaufen ist.

Es gibt keine Dummen Fragen

F: Kann ich beim Erstellen von Funktionen in Python immer nur ein Argument benutzen?

A: Nein, Sie können so viele Argumente definieren, wie Sie wollen, je nachdem, was Ihre Funktion können soll. Wir haben für den Anfang absichtlich ein einfaches Beispiel gewählt. Im Verlauf dieses Kapitels werden wir aber auch kompliziertere Funktionen sehen. Argumente können in Python einen großen Einfluss auf Funktionen haben. Auf den nächsten (etwa) zehn Seiten werden wir diese Möglichkeiten etwas genauer ausloten.

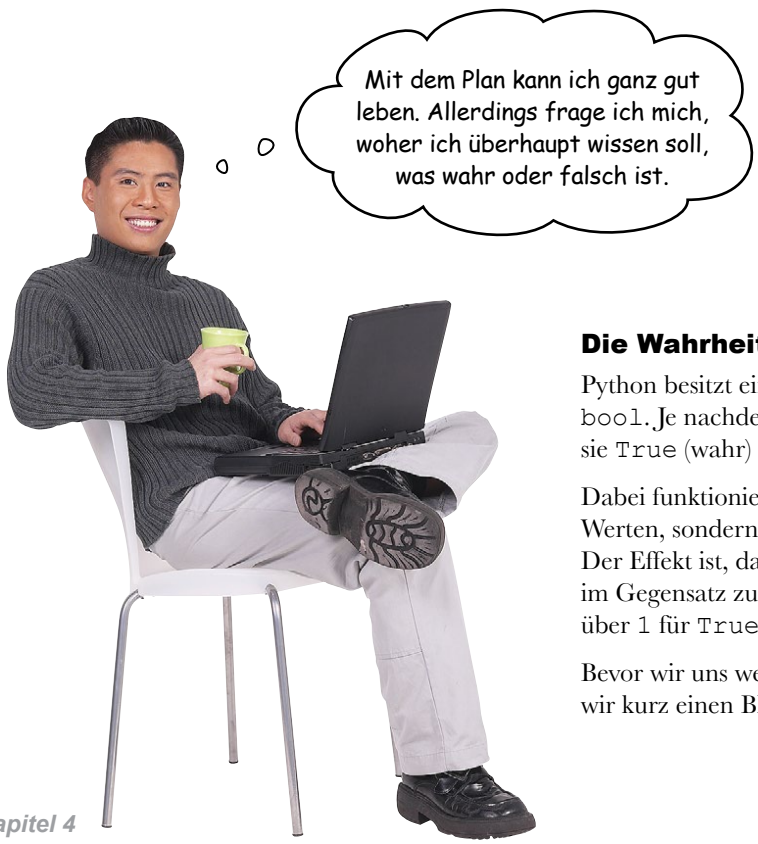
Funktionen geben ein Ergebnis zurück

So wie Programmierer Funktionen benutzen, um Code zu abstrahieren und ihn mit einem Namen zu versehen, wollen sie in der Regel, dass Funktionen einen berechneten Wert (oder mehrere) an den aufrufenden Code zurückgeben, mit dem dann weitergearbeitet werden kann. Hierfür gibt es es in Python die `return`-Anweisung.

Findet der Interpreter in einer Funktionssuite eine `return`-Anweisung, geschehen zwei Dinge: Die Funktion wird bei der `return`-Anweisung beendet. Gleichzeitig werden an die Anweisung übergebene Werte an den aufrufenden Code zurückgegeben. Dieses Verhalten ist größtenteils mit anderen Programmiersprachen identisch.

Wir beginnen mit einem einfachen Beispiel, in dem ein einzelner Wert aus der `search4vowels`-Funktion zurückgegeben wird. In unserem Fall soll dies entweder `True` oder `False` sein, je nachdem, ob das übergebene Wort Vokale enthält oder nicht.

Damit entfernen wir uns ein gutes Stück vom ursprünglichen Zweck unserer Funktion. Haben Sie dennoch etwas Geduld. Wir werden in Kürze etwas deutlich Komplexeres (und Nützlicheres) bauen. Mithilfe dieses einfachen Beispiels stellen wir sicher, dass die Grundlagen wirklich klar sind, bevor wir weitermachen.



Die Wahrheit ist ...

Python besitzt eine eingebaute Funktion namens `bool`. Je nachdem, welchen Wert Sie übergeben, gibt sie `True` (wahr) oder `False` (falsch) zurück.

Dabei funktioniert `bool` nicht nur mit beliebigen Werten, sondern auch auch mit Python-Objekten. Der Effekt ist, dass Pythons Auffassung von Wahrheit im Gegensatz zu anderen Programmiersprachen weit über `1` für `True` und `0` für `False` hinausgeht.

Bevor wir uns weiter mit `return` befassen, wollen wir kurz einen Blick auf `True` und `False` werfen.

Wahrheit unter der Lupe

Jedes Objekt besitzt einen »Wahrheitswert«, d. h., es kann zu `True` (wahr) oder zu `False` (falsch) ausgewertet werden.

Dinge gelten als `False`, wenn sie zu `0`, zum Wert `None`, zu einem leeren String oder einer leeren eingebauten Datenstruktur ausgewertet werden. Das bedeutet, dass alle folgenden Beispiele `False` sind:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False
>>> bool(None)
False
```

Wird ein Objekt zu 0 ausgewertet, ist es grunsdätzlich False.

Ein leerer String, eine leere Liste und ein leeres Dictionary werden alle zu False ausgewertet.

Pythons spezieller Wert »None« ist immer False.

Fast alle Objekte in Python werden zu `True` ausgewertet. Hier ein paar Beispiele für Objekte, die `True` sind:

```
>>> bool(1)
True
>>> bool(-1)
True
>>> bool(42)
True
>>> bool(0.00000000000000000000000000001)
True
>>> bool('Panic')
True
>>> bool([42, 43, 44])
True
>>> bool({'a': 42, 'b':42})
True
```

Eine Zahl, die nicht 0 ist, ist immer True, auch wenn sie negativ ist.

Der Wert ist zwar sehr klein, aber nicht 0, also ist er True.

Ein nicht leerer String ist immer True.

Eine nicht leere Datenstruktur ist »wahr« (True).

Wir können der `bool`-Funktion ein beliebiges Objekt übergeben, um zu sehen, ob es `True` oder `False` ist.

Nicht leere Datenstrukturen werden dabei grundsätzlich zu `True` ausgewertet.

Einen Wert zurückgeben

Sehen Sie sich noch einmal den Funktionscode an. Dieser akzeptiert im Moment einen beliebigen Wert als Argument, durchsucht diesen nach Vokalen und gibt die gefundenen Buchstaben auf dem Bildschirm aus:

```
def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Diese zwei Zeilen werden wir ändern.

Die Funktion lässt sich leicht so verändern, dass entweder `True` oder `False` zurückgegeben wird, je nachdem, ob Vokale gefunden wurden oder nicht. Ersetzen Sie die ersten zwei Codezeilen (die `for`-Schleife) einfach durch diese Zeile:

```
return bool(found)
```

Ruft die >>bool<<-Funktion auf und ...

... übergibt den Namen der Datenstruktur, die die Ergebnisse unserer Vokalsuche enthält.

Wurde nichts gefunden, gibt die Funktion `False` zurück, ansonsten `True`. Mit diesen Änderungen können Sie die neue Version Ihrer Funktion jetzt auf der Python-Shell testen, um zu sehen, was passiert:

```
>>> search4vowels('hitch-hiker')
True
>>> search4vowels('galaxy')
True
>>> search4vowels('sky')
False
```

Die >>return<<-Anweisung gibt (dank >>bool<<) entweder >>True<< oder >>False<< zurück.

Wie in früheren Kapiteln betrachten wir >>y<< hier nicht als Vokal.

Wenn Sie weiterhin das Verhalten der vorigen Version sehen, sollten Sie sicherstellen, dass Sie die neue Version Ihres Codes auch gespeichert und F5 vom Editorfenster aus gedrückt haben.

Geek-Bits

Widerstehen Sie der Versuchung, das von return an den aufrufenden Code zurückgegebene Objekt mit runden Klammern zu umgeben. Das ist nicht nötig. Die `return`-Anweisung ist kein Funktionsaufruf, und runde Klammern sind keine syntaktische Anforderung. Sie können sie benutzen (wenn Sie *unbedingt* wollen) – die meisten Python-Programmierer verzichten aber darauf.

Mehr als einen Wert zurückgeben

Grundsätzlich sind Funktionen darauf angelegt, einzelne Werte zurückzugeben. Manchmal reicht das aber nicht aus. Die einzige Möglichkeit, das zu umgehen, besteht darin, mehrere Werte in eine einzelne Datenstruktur zu verpacken und *diese* zurückzugeben. So können Sie weiterhin ein Ding zurückgeben, auch wenn es mehrere Einzeldaten enthält.

Im Moment gibt unsere Funktion nur einen booleschen Wert (d. h. eine Sache) zurück, wie hier gezeigt:

```
def search4vowels(word):
    """Return a boolean based on any vowels found."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return bool(found)
```

Hinweis: Wir haben den Kommentar aktualisiert.

Damit die Funktion mehrere Werte (in einem Set) anstelle eines booleschen Werts zurückgibt, ist nur eine einfache Änderung nötig. Wir müssen lediglich den Aufruf von `bool` weglassen.

```
def search4vowels(word):
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return found
```

Die Ergebnisse als Datenstruktur (Set) zurückgeben.

Wir haben den Kommentar erneut aktualisiert.

Wir können die letzten beiden Zeilen in der obigen Version unserer Funktion auf eine einzige Zeile reduzieren, indem wir die unnötige Verwendung der Variablen `found` weglassen. Anstatt `found` das Ergebnis der Schnittmenge zuzuweisen und dieses zurückzugeben, geben wir das Ergebnis von `intersection` einfach direkt zurück.

```
def search4vowels(word):
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

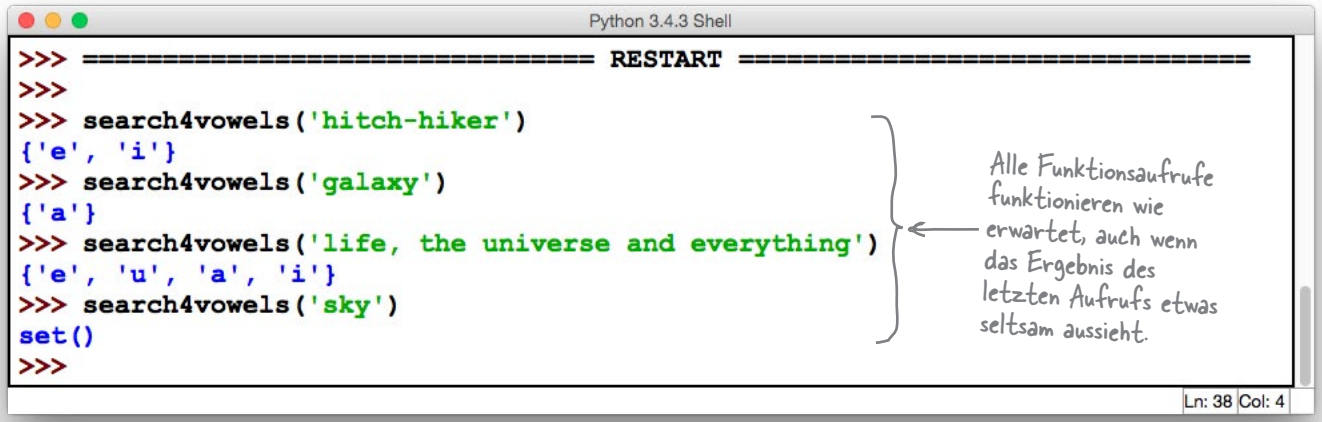
Die Daten ohne die unnötige »found«-Variable zurückgeben.

Unsere Funktion gibt jetzt, wie gewünscht, die im Wort gefundenen Vokale zurück.

Eines der zurückgegebenen Ergebnisse gibt uns allerdings zu denken …

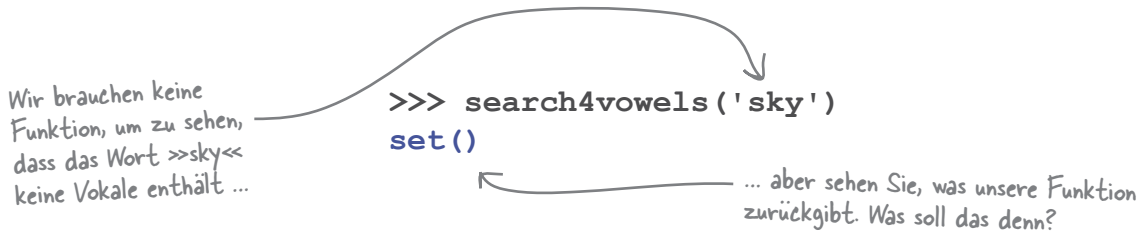
PROBEFAHRT

Jetzt wollen die aktuelle Version von `search4vowels` ausprobieren, um zu sehen, wie sie sich verhält. Drücken Sie, mit dem aktuellen Code im IDLE-Editorfenster, F5, um die Funktion in die Python-Shell zu laden. Danach rufen Sie die Funktion ein paarmal auf:



Was ist eigentlich mit »set()« los?

Die Beispiele in der obigen *Probefahrt* funktionieren alle. Sie übernehmen einen einzelnen String als Argument und geben die gefundenen Vokale als Set wieder zurück. Ein Ergebnis (das Set) enthält mehrere Werte. Trotzdem sieht das letzte Ergebnis ein wenig seltsam aus, oder? Das wollen wir uns etwas genauer ansehen:



Es ist ein häufiges Missverständnis, zu denken, dass die Funktion `{}` zurückgibt, um ein leeres Set darzustellen. Tatsächlich steht `{}` aber für ein leeres Dictionary und *nicht* für ein Set.

Ein leeres Set wird vom Interpreter als `()` dargestellt.

Das erscheint vielleicht etwas seltsam, aber so laufen die Dinge in Python nun einmal. Lassen Sie uns noch einen Blick auf die eingebauten Datenstrukturen und ihre Darstellung durch den Interpreter werfen.

Eingebaute Datenstrukturen: Wiederholung

Werfen Sie ein weiteres Mal einen Blick auf die möglichen eingebauten Datenstrukturen. Wir werden uns der Reihe nach mit Liste, Dictionary, Set und schließlich Tupel beschäftigen.

Auf der Shell wollen wir anhand der Funktionen für eingebaute Datenstrukturen (in der Python-Dokumentation auch als »BIFs« bezeichnet) jeweils eine leere Datenstruktur erstellen und ihr eine kleine Menge Daten zuweisen. Nach der Zuweisung geben wir den Inhalt der Strukturen aus:

BIF ist die Abkürzung für »build-in function« (eingebaute Funktion).

```
>>> l = list()
>>> l
[]
>>> l = [ 1, 2, 3 ]
>>> l
[1, 2, 3]
```

Benutzen Sie die eingebaute »list«-Funktion, um eine leere Liste zu erstellen und ein paar Daten zuzuweisen.

Eine leere Liste.

```
>>> d = dict()
>>> d
{}
>>> d = { 'first': 1, 'second': 2, 'third': 3 }
>>> d
{'second': 2, 'third': 3, 'first': 1}
```

Benutzen Sie die eingebaute »dict«-Funktion, um ein leeres Dictionary zu erstellen und ein paar Daten zuzuweisen.

Ein leeres Dictionary.

```
>>> s = set()
>>> s
set()
>>> s = {1, 2, 3}
>>> s
{1, 2, 3}
```

Benutzen Sie die eingebaute »set«-Funktion, um ein leeres Set zu erstellen und ein paar Daten zuzuweisen.

Ein leeres Set.

Sets werden normalerweise mit geschweiften Klammern umgeben. Das Gleiche gilt aber auch für Dictionaries. Ein leeres Dictionary verwendet ebenfalls geschweifte Klammern, daher wird ein leeres Set als »set()« formuliert.

```
>>> t = tuple()
>>> t
()
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
```

Benutzen Sie die eingebaute »tupel«-Funktion, um ein leeres Tupel zu erstellen und ein paar Daten zuzuweisen.

Ein leeres Tupel.

Bevor Sie weitermachen, sollten Sie sich vergegenwärtigen, wie der Interpreter leere Datenstrukturen darstellt (z. B. auf dieser Seite).

Benutzen Sie Annotationen, um Ihre Dokumentation zu verbessern

Unsere Wiederholung der vier Datenstrukturen zeigt, dass die `search4vowels`-Funktion ein Set zurückgibt. Aber wie können die Benutzer unserer Funktion das wissen, ohne die Funktion zuerst benutzen zu müssen? Woher sollen sie wissen, was zu erwarten ist?

Eine Lösung besteht darin, die nötigen Informationen in den Docstring zu schreiben. Dabei wird davon ausgegangen, dass im Docstring sehr genau angegeben wird, welche Argumente und Rückgabewerte zu erwarten sind, und dass diese Informationen leicht zu finden sind. Allerdings können sich Programmierer nur schwer auf einen Standard für die Dokumentation von Funktionen einigen (PEP 257 schlägt nur ein bestimmtes *Format* für Docstrings vor). Daher unterstützt Python 3 eine Schreibweise, die als **Annotationen** (»Annotations«, Anmerkungen, auch *type hints*, Typhinweise, genannt) bezeichnet wird. Richtig benutzt, dokumentieren Annotationen auf standardisierte Weise den Datentyp des Rückgabewerts sowie die Datentypen möglicher Argumente. Merken Sie sich Folgendes:

1. **Annotationen sind optional.**
   Es ist in Ordnung, sie nicht zu benutzen. Tatsächlich gibt es viel Python-Code ohne Annotationen (weil sie bisher nur Programmierern der neueren Versionen von Python 3 zugänglich waren).

2. **Annotationen geben Informationen weiter.**
   Sie enthalten Details zu Ihrer Funktion, implizieren aber kein anderes Verhalten (z. B. die Überprüfung des Datentyps).

Wir wollen eine Annotation für die Argumente unserer `search4vowels`-Funktion hinzufügen. Der erste Teil besagt, dass die Funktion als Datentyp für das Argument `word` einen String (`:str`) erwartet, während der zweite Teil darauf hinweist, dass die Funktion ein Set an den aufrufenden Code zurückgibt (`-> set`).

Wir geben an, dass als Datentyp für das Argument »word« ein String erwartet wird.

Wir geben an, dass die Funktion ein Set an den aufrufenden Code zurückgibt.

```
def search4vowels(word:str) -> set:
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

Die Syntax für Annotationen ist ziemlich einfach. Jedem Funktionsargument wird ein Doppelpunkt nachgestellt, gefolgt vom erwarteten Datentyp. In unserem Beispiel gibt `:str` an, dass die Funktion einen String erwartet. Der Rückgabetyp wird nach der Argumentliste angegeben und durch ein Pfeilsymbol gekennzeichnet (einem Minuszeichen gefolgt von einem Kleiner-als-Zeichen). Hierauf folgt der Datentyp, dann der Doppelpunkt. In unserem Fall bedeutet `-> set:`, dass die Funktion ein Set zurückgibt.

So weit, so gut.

Damit haben wir die Funktion standardgemäß mit einer Annotation versehen. Hierdurch wissen Programmierer, die Ihre Funktion benutzen, auf den ersten Blick, was Sie und Ihre Funktion erwarten. Allerdings wird der Interpreter **nicht überprüfen**, ob die Funktion tatsächlich mit einem String aufgerufen wurde oder immer ein Set zurückgibt. Das wirft eine ziemlich offensichtliche Frage auf …

Weitere Details zu den Annotationen finden Sie in PEP 3107 unter https://www.python.org/dev/peps/pep-3107/.

Warum sollte man Funktionsannotationen überhaupt benutzen?

Was hat es überhaupt für einen Sinn, Annotationen zu benutzen, wenn der Interpreter sie nicht einmal verwendet, um die Datentypen Ihrer Funktionsargumente und Rückgabewerte zu überprüfen?

Bei den Annotationen geht es *nicht* darum, dem Interpreter das Leben leichter zu machen, sondern dem Benutzer der Funktion. Annotationen sind ein **Dokumentationsstandard**, *kein* Mechanismus zum Erzwingen bestimmter Datentypen.

Tatsächlich ist es dem Interpreter vollkommen egal, welchen Datentyp Ihre Argumente bzw. Rückgabewerte haben. Der Interpreter ruft Ihre Funktion mit den Argumenten auf, die Sie übergeben, und gibt die in der `return`-Anweisung definierten Werte an den aufrufenden Code zurück. Die dabei verwendeten Datentypen interessieren den Interpreter nicht.

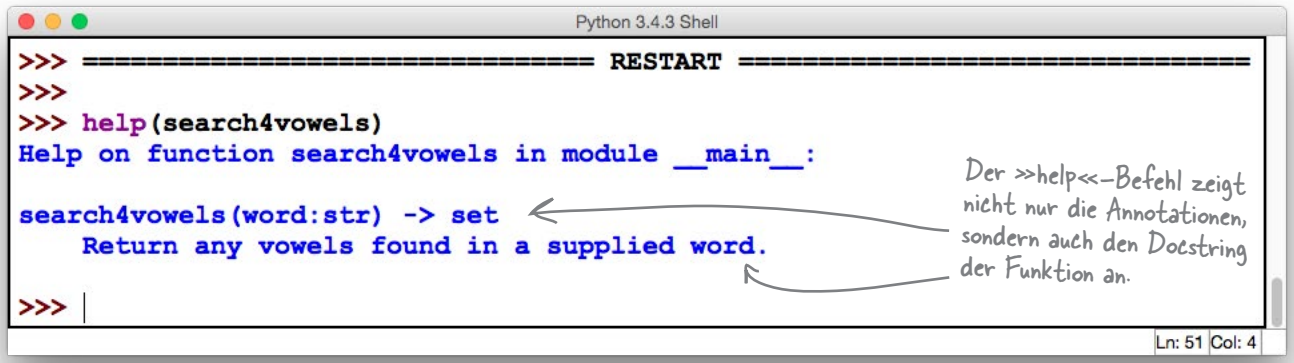
Durch Annotationen müssen Programmierer sich nicht mehr mühsam durch den Code Ihrer Funktion graben, um herauszubekommen, welche Datentypen erwartet werden bzw. zu erwarten sind. Selbst der schönste Docstring muss gelesen werden, wenn er keine Annotationen enthält.

Und das bringt uns zu einer weiteren Frage: Wie können wir die Annotationen anzeigen, ohne den Code der Funktion lesen zu müssen? Drücken Sie im Editorfenster von IDLE F5 und verwenden Sie dann die eingebaute `help`-Funktion am >>>-Prompt.

Nutzen Sie Annotationen, um Ihre Funktionen besser zu dokumentieren, und die eingebaute help-Funktion, um sie anzuzeigen.

PROBEFAHRT

Falls noch nicht geschehen, benutzen Sie jetzt den IDLE-Editor, um Ihre Kopie von `search4vowels` mit Annotationen zu versehen. Speichern Sie Ihren Code und drücken Sie dann F5. Die Python-Shell wird neu gestartet, und der >>>-Prompt wartet auf Ihre Eingaben. Verwenden Sie die eingebaute Funktion (»BIF«), um die Dokumentation für `search4vowels` anzuzeigen, wie hier gezeigt:



Funktionen: Was wir bereits wissen

Machen wir eine kleine Pause und denken wir kurz darüber nach, was wir bereits über Funktionen wissen.

Punkt für Punkt

- Funktionen sind benannte Codestücke.
- Das Schlüsselwort `def` wird benutzt, um eine Funktion zu benennen. Der Funktionscode wird unter dem `def`-Schlüsselwort (und relativ dazu) eingerückt platziert.
- Pythons Strings mit dreifachen Anführungszeichen können verwendet werden, um Funktionen mit mehrzeiligen Kommentaren zu versehen. Sie werden als *Docstrings* bezeichnet.
- Funktionen können eine beliebige Anzahl an Argumenten übernehmen, auch gar keine.
- Anhand der `return`-Anweisung können Funktionen eine beliebige Anzahl von Werten (auch gar keine) zurückgeben.
- Anhand von Funktionsannotationen können die Datentypen der Funktionsargumente und Rückgabewerte dokumentiert werden.

Sehen wir uns den Code von `search4vowels` noch einmal an. Die Funktion übernimmt im Moment ein einzelnes Argument und gibt ein Set zurück. Sie ist deutlich nützlicher als die erste Version unserer Funktion vom Beginn dieses Kapitels, da wir sie an verschiedenen Orten einsetzen können:

Die aktuelle Version unserer Funktion.

```
def search4vowels(word:str) -> set:
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

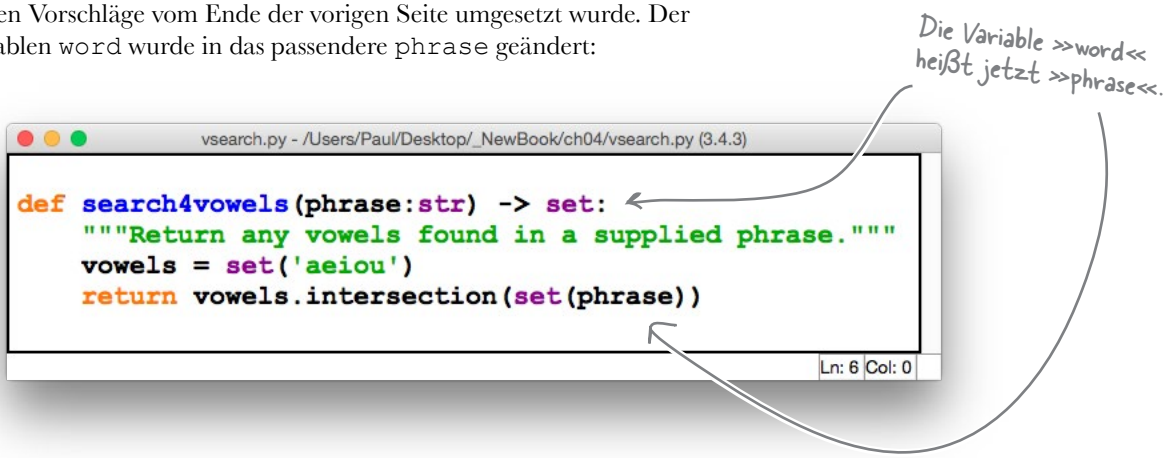
Diese Funktion wäre noch nützlicher, wenn sie nicht nur das Suchwort als Argument akzeptieren würde, sondern zusätzlich ein zweites Argument, das angibt, wonach gesucht werden soll. So könnten wir nicht nur nach Vokalen, sondern nach beliebigen Buchstaben suchen.

Außerdem ist der Name `word` für das Argument zwar OK, aber nicht besonders vielsagend, zumal die Funktion tatsächlich einen beliebigen String als Benutzereingabe akzeptiert und nicht nur ein einzelnes Wort. Ein besserer Variablenname wäre vielleicht `phrase` (Satz), da er genauer beschreibt, was wir tatsächlich erwarten.

Bauen wir den letzten Vorschlag gleich in die Funktion ein.

Eine allgemein nützliche Funktion erstellen

Hier sehen Sie eine Version von `search4vowels` (in IDLE), nachdem der zweite der beiden Vorschläge vom Ende der vorigen Seite umgesetzt wurde. Der Name der Variablen `word` wurde in das passendere `phrase` geändert:



Der andere Vorschlag vom Ende der vorigen Seite bestand darin, die zu suchenden Buchstaben vom Benutzer angeben zu lassen, anstatt immer die fünf Vokale zu verwenden. Hierfür können Sie die Funktion mit einem zweiten Argument versehen, das angibt, welche Buchstaben für die Suche verwendet werden sollen. Die Änderung ist einfach. Allerdings ist der bisherige Name der Funktion nicht mehr passend, da wir nicht mehr nur nach Vokalen suchen, sondern nach beliebigen Buchstaben. Anstatt die aktuelle Funktion (`search4vowels`) anzupassen, wollen wir eine zweite Funktion erstellen, die auf der ersten basiert. Hier unsere Vorschläge für die nötigen Arbeitsschritte:

1. **Der Funktion einen allgemeineren Namen geben**
   Anstatt `search4vowels` weiter anzupassen, erstellen wir besser eine neue Funktion mit dem Namen `search4letters`. Dieser Name beschreibt den Zweck der Funktion deutlich besser.

2. **Ein zweites Argument hinzufügen**
   Ein zweites Argument ermöglicht uns, die Buchstaben anzugeben, nach denen gesucht werden soll. Nennen wir das zweite Argument `letters` (Buchstaben) und vergessen wir nicht, `letters` ebenfalls mit Annotationen zu versehen.

3. **Die `vowels`-Variable entfernen**
   Der Variablenname `vowels` ergibt in der Suite der Funktion keinen Sinn mehr, da wir nach beliebigen vom Benutzer angegebenen Buchstaben suchen.

4. **Den Docstring aktualisieren**
   Es ist nicht sinnvoll, den Code zu kopieren und zu aktualisieren, ohne auch den Docstring anzupassen. Unsere Dokumentation muss ebenfalls beschreiben, was die neue Funktion tut.

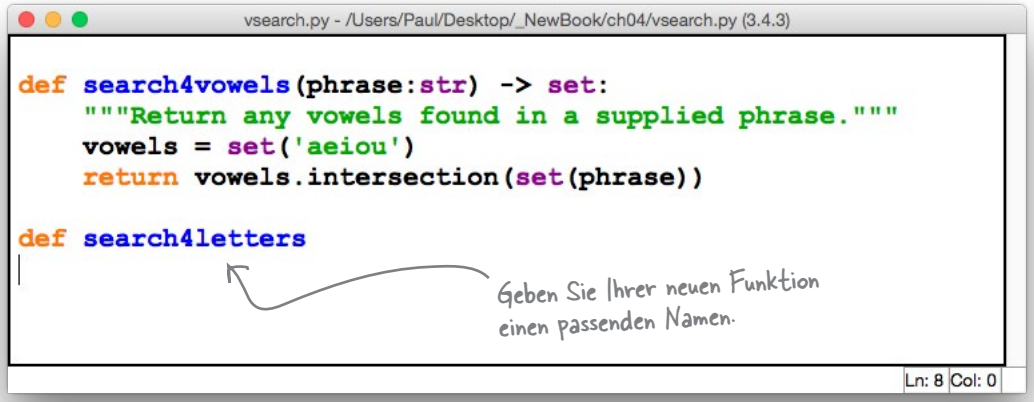
Wir werden diese Aufgaben der Reihe nach abarbeiten. Vergessen Sie beim Durchgehen dieser Schritte nicht, die `vsearch.py`-Datei entsprechend den Änderungen anzupassen und zu speichern.

Eine neue Funktion erstellen, 1 von 3

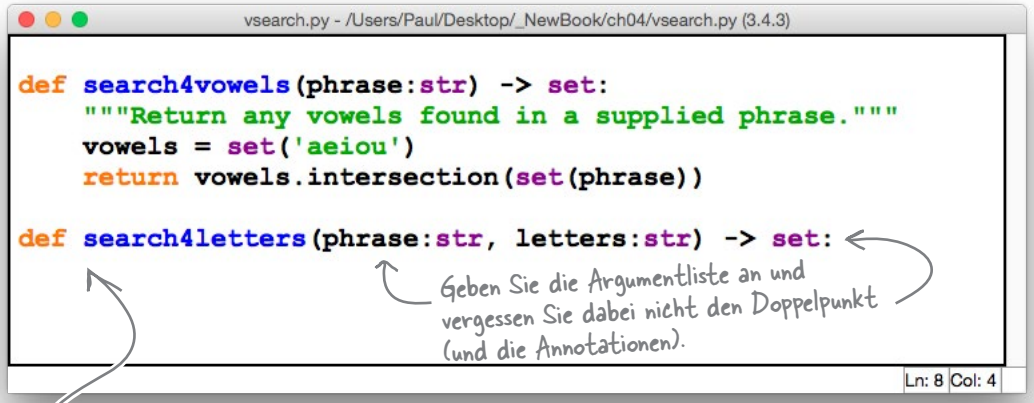
Falls noch nicht geschehen, öffnen Sie die Datei `vsearch.py` jetzt in einem IDLE-Editorfenster.

Schritt 1 sieht die Erstellung einer neuen Funktion vor, die wir `search4letters` nennen. Beachten Sie, dass PEP 8 dazu rät, alle Funktionen der obersten Ebene mit zwei Leerzeilen zu umgeben. Sämtliche Downloads für dieses Buch halten sich an diese Richtlinien, der Code in der Druckversion dieses Buchs möglicherweise aber nicht (weil Leerraum hier schlicht Geld kostet).

Schreiben Sie ans Ende der Datei das Schlüsselwort ***def***, gefolgt vom Namen Ihrer neuen Funktion:



In **Schritt 2** stellen wir die `def`-Zeile der Funktion fertig, indem wir die Namen der zwei erforderlichen Argumente, `phrase` und `letters`, angeben. Vergessen Sie nicht, die Argumente mit runden Klammern zu umgeben und den nachgestellten Doppelpunkt (und natürlich die Annotationen) hinzuzufügen:



Haben Sie gemerkt, dass der IDLE-Editor die nächste Codezeile automatisch eingerückt (und die Einfügemarke entsprechend positioniert) hat?

Sind die Schritte 1 und 2 getan, können wir jetzt den Funktionscode schreiben. Der Code wird starke Ähnlichkeit mit der `search4vowels`-Funktion haben, abgesehen davon, dass wir planen, die Variable `vowels` nicht mehr zu verwenden.

Eine neue Funktion erstellen, 2 von 3

Und weiter geht's mit **Schritt 3**. Wir schreiben den Funktionscode so um, dass die `vowels`-Variable nicht mehr gebraucht wird. Wir könnten sie zwar einfach umbenennen (weil `vowels`, also »Vokale«, nicht mehr korrekt ist) und weiterbenutzen, aber eine temporäre Variable wird hier nicht mehr benötigt. Der Grund ist der gleiche wie bei der bereits entfernten Variablen `found`. Sehen Sie sich die neue Codezeile in `search4letters` an, die die gleiche Aufgabe erledigt wie die zwei Zeilen in `search4vowels`:

Die beiden Codezeilen werden zu einer zusammengefasst.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    return set(letters).intersection(set(phrase))
```

Ln: 9 Col: 0

Verzweifeln Sie nicht, wenn die einzelne Codezeile Sie verwirrt. Sie sieht komplexer aus, als sie ist. Sehen wir uns das einmal im Detail an, um genau herauszufinden, was hier eigentlich passiert. Die Zeile beginnt damit, den Wert des `letters`-Arguments in ein Set umzuwandeln:

```
set(letters)
```

Ein Set-Objekt aus »letters« erstellen.

Dieser Aufruf der eingebauten `set`-Funktion erzeugt aus den Buchstaben in der Variablen `letters` ein neues Set-Objekt. Wir müssen das Objekt übrigens keiner Variablen zuweisen, da wir es direkt weiterverwenden wollen, anstatt es für später zu speichern. Um das gerade erzeugte Set-Objekt zu benutzen, hängen Sie ihm einen Punkt an, gefolgt von der Methode, die Sie daran aufrufen wollen. Und ja, selbst Objekte, die keiner Variablen zugewiesen wurden, besitzen Methoden. Wie wir von der Verwendung der Sets aus dem vorigen Kapitel wissen, übernimmt die `intersection`-Methode das in seinem Argument enthaltene Set mit Buchstaben (`phrase`) und bildet daraus und aus einem bereits vorhandenen Objekt (`letters`) die Schnittmenge:

```
set(letters).intersection(set(phrase))
```

Die Schnittmenge (intersection) aus dem Set-Objekt von »letters« und dem Set-Objekt von »phrase« bilden.

Schließlich wird das Ergebnis der Schnittmengenbildung durch die `return`-Anweisung an den aufrufenden Code zurückgegeben.

```
return set(letters).intersection(set(phrase))
```

Das Ergebnis an den aufrufenden Code zurückgeben.

Eine neue Funktion erstellen, 3 von 3

Bleibt nur noch **Schritt 4**, in dem wir die neue Funktion mit einem passenden Docstring versehen. Dafür fügen Sie direkt nach der `def`-Zeile einen String in dreifachen Anführungszeichen (`'''`) ein. Hier ist unsere Version (wie bei Kommentaren üblich: kurz, aber effektiv):

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))

```

Ln: 10 Col: 0

Ein Doc-string.

Damit sind unsere vier Arbeitsschritte vollendet, und `search4letters` ist bereit zum Testen.

Wozu der Aufwand, eine einzeilige Funktion zu erstellen? Ist es nicht leichter, den Code einfach zu kopieren und dort einzufügen, wo er gerade gebraucht wird?

Funktionen können Komplexität verstecken.

Es *stimmt*, dass die Erstellung einer einzeiligen Funktion keine besonderen »Einsparungen« mit sich bringt. Allerdings sollten Sie nicht übersehen, dass unsere Funktion diese einzelne komplexe Codezeile vor den Benutzern versteckt. Und das kann wiederum sehr hilfreich sein (und ist ohnehin um einiges besser, als den Code ständig zu kopieren und woanders wieder einzufügen).

Die meisten Programmierer sind vermutlich in der Lage, zu erraten, was `search4letters` tut, wenn die Funktion in einem Programm aufgerufen wird. Würden Sie stattdessen die komplexe Codezeile in einem Programm sehen, fragten Sie sich wahrscheinlich, was dieser Code eigentlich tut. Deshalb ist es trotz der »Kürze« von `search4letters` sinnvoll, diese Komplexität innerhalb einer Funktion und von der direkten Benutzung weg zu abstrahieren.

PROBEFAHRT

Speichern Sie `vsearch.py` erneut und drücken Sie F5, um die `search4letters`-Funktion auszuprobieren:

```
Python 3.4.3 Shell
>>> ========== RESTART ==========
>>>
>>> help(search4letters)
Help on function search4letters in module __main__:

search4letters(phrase:str, letters:str) -> set
    Return a set of the 'letters' found in 'phrase'.

>>> search4letters('hitch-hiker', 'aeiou')
{'e', 'i'}
>>> search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> search4letters('life, the universe, and everything', 'o')
set()
>>>
                                                    Ln: 78 Col: 4
```

Benutzen Sie die eingebaute Funktion >>help<<, um anzuzeigen, wie >>search4letter<< verwendet werden soll.

Alle diese Beispiele erzeugen das erwartete Ergebnis.

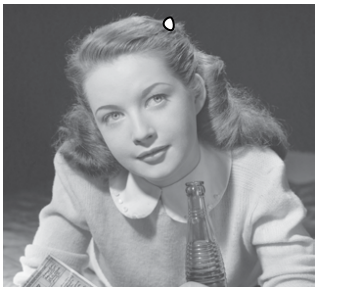
Die `search4letters`-Funktion ist deutlich flexibler als `search4vowels`. Sie übernimmt *beliebige* Buchstaben und durchsucht den ebenfalls übergebenen String (in `phrase`) danach, anstatt nur nach den Buchstaben `a`, `e`, `i`, `o` und `u` zu suchen. Stellen wir uns vor, wir hätten eine umfangreiche Codebasis, in der `search4vowels` sehr häufig benutzt wurde. Es wurde entschieden, `search4vowels` auszumustern und durch `search4letters` zu ersetzen, da die »Entscheider« keinen Bedarf für beide Funktionen sahen. Schließlich kann `search4letters` die Aufgaben von `search4vowels` mit erledigen. Ein globaler Suchlauf in unserer Codebasis, der »search4vowels« durch »search4letters« ersetzt, würde allerdings fehlschlagen. Schließlich wird für `search4vowels` ein zweites Argument benötigt. Dieses muss immer `aeiou` sein, wenn das Verhalten von `search4vowels` simuliert werden soll. Nehmen wir beispielsweise diesen Aufruf:

```
search4vowels("Don't panic!")
```

Dies muss jetzt durch die Zwei-Argumente-Form ersetzt werden (was wesentlich schwerer automatisch zu auszutauschen ist):

```
search4letters("Don't panic!", 'aeiou')
```

Wäre es nicht praktisch, wenn wir `search4letters` mit einem Standardwert für das zweite Argument versehen könnten? Diesen könnte die Funktion dann verwenden, falls kein alternativer Wert angegeben wurde. Wäre `aeiou` der Standardwert, könnten wir die Codebasis durch einen globalen Suchlauf aktualisieren, um die Änderungen vorzunehmen.



Standardwerte für Argumente definieren

Jedem Argument einer Python-Funktion kann ein Standardwert zugewiesen werden. Dieser wird automatisch verwendet, falls der aufrufende Code keinen expliziten Wert angibt. Der Mechanismus für die Definition eines Standardwerts ist einfach: Geben Sie den Standardwert in Form einer Zuweisung auf der `def`-Zeile der Funktion mit an.

Hier die aktuelle `def`-Zeile von `search4letters`:

```
def search4letters(phrase:str, letters:str) -> set:
```

Diese Version der `def`-Zeile unserer Funktion (oben) erwartet genau zwei Argumente: eines für `phrase` und ein zweites für `letters`. Weisen wir `letters` allerdings einen Standardwert zu, ändert sich die `def`-Zeile wie folgt:

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Wir haben dem »letters«-Argument einen Standardwert zugewiesen, der benutzt wird, falls kein expliziter Wert angegeben wurde.

Wir können `search4letters` trotzdem so weiterbenutzen wie zuvor: Beide Argumente können weiterhin angegeben werden. Vergessen wir allerdings das zweite Argument (`letters`), benutzt der Interpreter stattdessen den Standardwert `aeiou`.

Nähmen wir diese Änderung in unserer `vsearch.py`-Datei vor (und speicherten sie), könnten wir die Funktion auch folgendermaßen aufrufen:

Diese drei Funktionsaufrufe haben das gleiche Ergebnis.

```
>>> search4letters('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
>>> search4letters('life, the universe, and everything', 'aeiou')
{'a', 'e', 'i', 'u'}
>>> search4vowels('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
```

Hier rufen wir »search4vowels« und nicht »search4letters« auf.

Diese Funktionsaufrufe erzeugen nicht nur das gleiche Ergebnis, sie zeigen auch, dass die `search4vowels`-Funktion nicht länger gebraucht wird, nachdem das `letters`-Argument für `search4letters` einen Standardwert unterstützt (vergleichen Sie hierfür den ersten und letzten Aufruf im obigen Beispiel).

Jetzt ist es kein Problem mehr, `search4vowels` auszumustern und die Aufrufe in der Codebasis durch `search4letters` auszutauschen. Durch die Möglichkeit, Standardwerte für Funktionsargumente zu definieren, reicht ein globaler Suchlauf aus, um die alte durch die neue Funktion zu ersetzen. Dabei ist `search4letters` nicht auf die Suche nach Vokalen beschränkt. Als zweites Argument können Sie *beliebige* Zeichen angeben, nach denen gesucht werden soll. Dadurch ist `search4letters` allgemeiner *und gleichzeitig* nützlicher.

Positionelle und Schlüsselwortzuweisung im Vergleich

Wie gesehen, kann `search4letters` mit einem oder mit zwei Argumenten aufgerufen werden, wobei das zweite Argument optional ist. Geben Sie nur ein Argument an, wird für das `letters`-Argument standardmäßig ein vordefinierter String mit Vokalen verwendet. Sehen wir uns die `def`-Zeile der Funktion noch einmal an:

Die »def«-Zeile unserer Funktion.

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Neben der Unterstützung von Standardargumenten ermöglicht der Interpreter auch die Verwendung von **Schlüsselwortargumenten**. Um das zu verstehen, überlegen wir noch einmal, wie wir `search4letters` bisher aufgerufen haben:

```
search4letters('galaxy', 'xyz')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Beim obigen Aufruf werden den Argumenten `phrase` und `letters` die Strings basierend auf ihrer Position zugewiesen. Hierbei erhält `phrase` den ersten String und `letters` den zweiten. Dieses Verfahren wird auch als **positionelle Zuweisung** (Positional Assignment) bezeichnet, da es auf der Reihenfolge der Argumente basiert.

In Python ist es außerdem möglich, Argumenten anhand ihres Namens Werte zuzuweisen. Dabei spielt die Reihenfolge keine Rolle mehr. Dieser Weg wird auch **Schlüsselwortzuweisung** (Keyword Assignment) genannt. Bei der Verwendung von Schlüsselwörtern können Sie den Argumenten die Strings beim Funktionsaufruf *in beliebiger Reihenfolge* zuweisen, wie hier gezeigt:

Die Reihenfolge der Argumente ist nicht wichtig, wenn sie beim Aufruf als Schlüsselwörter angegeben werden.

```
search4letters(letters='xyz', phrase='galaxy')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Beide Aufrufe von `search4letters` auf dieser Seite ergeben das gleiche Ergebnis: ein Set mit den Buchstaben `y` und `z`. Bei unserer kleinen `search4letters`-Funktion ist die Verwendung von Schlüsselwortargumenten nicht unbedingt nachvollziehbar. Das ändert sich aber, wenn Sie eine Funktion aufrufen, der sehr viele Argumente übergeben werden können. Ein Beispiel für solch eine Funktion (Teil der Standardbibliothek) werden wir gegen Ende dieses Kapitels noch sehen.

Aktualisierung unseres Wissens über Funktionen

Nachdem Sie einige Zeit mit der Erforschung von Funktionsargumenten verbracht haben, wollen wir Ihr Wissen über Funktionen aktualisieren:

Punkt für Punkt

- Neben der Möglichkeit, Code wiederzuverwenden, können Funktionen Komplexität vor dem Benutzer verstecken. Wenn Sie eine komplexe Codezeile haben, die Sie oft benutzen wollen, sollten Sie sie hinter einem einfachen Funktionsaufruf verbergen.
- Sie können jedem Funktionsargument auf der `def`-Zeile einen Standardwert zuweisen. In diesem Fall ist die Angabe eines Werts für das betreffende Argument beim Funktionsaufruf optional.
- Neben der Zuweisung von Argumenten anhand ihrer Position können Sie auch Schlüsselwörter verwenden. In diesem Fall ist die Reihenfolge der Argumente beliebig (da Missverständnisse durch die Verwendung der Schlüsselwörter ausgeräumt sind und die Position keine Rolle mehr spielt).

Es gibt mehr als einen Weg, die Dinge zu erledigen.

Jetzt haben Sie Code, den Sie guten Gewissens mit anderen teilen können. Daher ist die Frage berechtigt, wie diese Funktionen am besten verwendet und mit anderen geteilt werden können. Wie meistens gibt es auf diese Frage mehrere Antworten. Auf den folgenden Seiten werden Sie lernen, wie Sie Ihre Funktionen am besten verpacken und weitergeben, damit Sie und andere den größtmöglichen Nutzen von Ihrer Arbeit haben.

Aus Funktionen werden Module

Nachdem wir uns die Arbeit gemacht haben, eine wiederverwendbare Funktion zu schreiben (eigentlich enthält `vsearch.py` sogar zwei Funktionen), drängt sich die Frage auf: *»Wie kann ich meine Funktionen am besten mit anderen teilen?«*

Zwar können Sie Funktionen durch Kopieren und Einfügen in Ihrer Codebasis wiederverwenden, das ist aber weder besonders praktisch noch besonders effizient. Daher werden wir uns auch nicht weiter damit beschäftigen. Mehrere Kopien einer Funktion, die Ihre Codebasis verstopfen, sind ein todsicheres Rezept für eine Katastrophe (spätestens wenn Sie Änderungen an Ihrer Funktion vornehmen müssen). Es ist viel besser, stattdessen ein Modul zu erstellen, das die einzigen anerkannten Kopien der Funktionen enthält, die Sie mit anderen teilen wollen. Und das bringt uns zur nächsten Frage: *»Wie erstellt man Python-Module?«*
Die Antwort könnte nicht leichter sein: Ein Modul ist eine beliebige Datei, die Funktionen enthält. Und das bedeutet, dass `vsearch.py` *bereits ein Modul ist*. Hier noch einmal der gesamte Code in voller Schönheit:

Geben Sie Ihre Funktionen in Modulen weiter.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Ln: 10 Col: 0

>>vsearch.py<< enthält Funktionen in einer Datei und ist damit bereits ein vollständiges Modul.

Das Erstellen von Modulen könnte nicht leichter sein, allerdings ...

Es ist kinderleicht, Module zu erstellen: Schreiben Sie die Funktionen, die Sie mit anderen teilen wollen, einfach in eine Datei.

Sobald das Modul existiert, können Sie es problemlos anderen Programmen zugänglich machen: Hierfür müssen Sie das Modul einfach mit Pythons `import`-Anweisung importieren.

Das ist für sich genommen noch nicht besonders spannend. Allerdings geht der Interpreter davon aus, dass sich das Modul im **Suchpfad** (»Search Path«) befindet, und das kann in diesem Fall schwierig werden. Auf den folgenden Seiten wollen wir uns daher mit den Einzelheiten zum Importieren von Modulen beschäftigen.

Wie werden die Module gefunden?

Modul

Erinnern Sie sich noch, wie wir im ersten Kapitel die `randint`-Funktion aus dem `random`-Modul importiert und dann benutzt haben? Dieses Modul ist Teil von Pythons Standardbibliothek. Hier die Befehle auf der Shell:

Nennen Sie das zu importierende Modul und ...

```
>>> import random
>>> random.randint(0, 255)
42
```

... rufen Sie eine der Funktionen des Moduls auf.

Was beim Import von Modulen geschieht, ist sehr detailliert in der Python-Dokumentation beschrieben, die Sie sich natürlich jederzeit ansehen dürfen, falls all die feinen Details Sie wirklich interessieren. Im Moment brauchen Sie aber nur die drei Orte, an denen der Interpreter standardmäßig nach Modulen sucht. Dies sind:

1. **Ihr aktuelles Arbeitsverzeichnis**
   Das ist der Ordner, von dem der Interpreter denkt, dass Sie gerade darin arbeiten.

2. **die Orte, an denen der Interpreter nach »site-packages« sucht**
   Das sind die Ordner, die Python-Module von Drittherstellern enthalten, die Sie möglicherweise zusätzlich installiert haben.

3. **die Speicherorte der Standardbibliothek**
   Das sind die Ordner, die die Module enthalten, aus denen die Standardbibliothek besteht.

Geek-Bits

Je nach Betriebssystem wird der Ort, der Dateien enthält, entweder **Verzeichnis** oder auch **Ordner** genannt. In diesem Buch benutzen wir den Begriff »Ordner«, außer wenn wir vom *aktuellen Arbeitsverzeichnis* (einem etablierten Fachbegriff) sprechen.

Die Reihenfolge, in der der Interpreter die Orte 1 und 2 durchsucht, hängt von vielen Faktoren ab. Aber keine Sorge: Sie müssen nicht wissen, wie der Suchmechanismus im Einzelnen funktioniert. *Wichtig ist*, dass Sie verstehen, dass der Interpreter immer *zuerst* Ihr aktuelles Arbeitsverzeichnis durchsucht, was gelegentlich zu Problemen führen kann, wenn Sie mit eigenen Modulen arbeiten.

Um zu zeigen, was hier schiefgehen kann, wollen wir eine kleine Übung angehen, die dieses Problem illustriert. Zu Beginn brauchen Sie die folgenden Dinge:

- ☐ Erstellen Sie einen neuen Ordner mit dem Namen `mymodules`, in dem unsere Module gespeichert werden. Es ist nicht wichtig, wo im Dateisystem dieser Ordner liegt. Stellen Sie nur sicher, dass Sie die Lese- und Schreibrechte haben.

- ☐ Bewegen Sie die Datei `vsearch.py` in den neu erstellten Ordner `mymodules`. Diese Datei sollte dabei die einzige mit dem Namen `vsearch.py` auf Ihrem Computer sein.

Python auf der Kommandozeile ausführen

Wir werden den Python-Interpreter von der Kommandozeile (oder dem Terminal) Ihres Betriebssystems aus ausführen, um zu zeigen, was dabei schiefgehen kann (obwohl die hier gezeigten Probleme durchaus auch in IDLE auftreten können).

Wenn Sie mit einer beliebigen Version von *Windows* arbeiten, öffnen Sie eine Eingabeaufforderung und folgen den hier gezeigten Schritten. Wenn Sie nicht unter *Windows* arbeiten, werden wir Ihre Plattform ungefähr in der Mitte der folgenden Seite besprechen (lesen Sie trotzdem weiter!). Danach können Sie den Python-Interpreter mit dem Befehl **`py -3`** auf der *Windows*-`C:\>`-Eingabeaufforderung aufrufen. Wir benutzen den Befehl **`cd`**, um den Ordner `mymodules` zum aktuellen Arbeitsverzeichnis zu machen. Sie können den Interpreter übrigens jederzeit beenden, indem Sie den Befehl **`quit()`** am >>>-Prompt eingeben:

In den Ordner »mymodules« wechseln.

Python 3 starten.

Das Modul importieren.

Die Funktionen des Moduls verwenden.

Beenden Sie den Python-Interpreter und kehren Sie zur Kommandozeile Ihres Betriebssystems zurück.

```
Datei Bearbeiten Fenster Hilfe Redmond #1
C:\Users\Head First> cd mymodules

C:\Users\Head First\mymodules> py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'y', 'x'}
>>> quit()

C:\Users\Head First\mymodules>
```

Das funktioniert wie erwartet: Das Modul `vsearch` wurde erfolgreich importiert. Danach werden die enthaltenen Funktionen nacheinander verwendet, indem wir dem Funktionsnamen den Namen des Moduls und einen Punkt voranstellen. Hierbei ist das Verhalten am >>>-Prompt mit dem Verhalten in IDLE identisch (es fehlt nur das Syntax-Highlighting). Schließlich wird ja auch der gleiche Python-Interpreter benutzt.

Obwohl diese Interaktion mit dem Interpreter erfolgreich war, hat sie nur funktioniert, weil wir in einem Ordner begonnen haben, der `vsearch.py` enthielt. Dadurch wurde der Ordner zum aktuellen Arbeitsverzeichnis. Da wir wissen, dass der Interpreter hier zuerst nach Modulen sucht, überrascht es nicht weiter, dass diese Interaktion funktioniert und der Interpreter unser Modul gefunden hat.

Was passiert aber, wenn das Modul nicht im aktuellen Arbeitsverzeichnis liegt?

Nicht gefundene Module erzeugen einen ImportError

Wiederholen Sie die Übung von der vorigen Seite, nachdem Sie den Ordner verlassen haben, der unser Modul enthält. Schauen wir, was passiert, wenn Sie versuchen, das Modul jetzt zu importieren. Hier eine weitere Interaktion mit der *Windows*-Eingabeaufforderung:

Wechseln Sie in einen anderen Ordner (in unserem Beispiel den obersten Ordner)

Python 3 erneut starten.

Wir versuchen, das Modul zu importieren ...

... aber diesmal erhalten wir eine Fehlermeldung!

```
Datei Bearbeiten Fenster Hilfe Redmond #2
C:\Users\Head First> cd \

C:\>py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

C:\>
```

Die Datei `vsearch.py` befindet sich nicht länger im aktuellen Arbeitsverzeichnis des Interpreters, da wir jetzt nicht mehr in `mymodules` arbeiten. Dadurch wird unsere Datei nicht mehr gefunden und kann nicht mehr importiert werden, und wir erhalten einen `ImportError` vom Interpreter.

Führen wir die Übung auf einer anderen Plattform als *Windows* aus, erhalten wir die gleichen Ergebnisse (unabhängig davon, ob wir unter *Linux*, *Unix* oder *macOS* arbeiten). Hier sehen Sie die obige Interaktion mit dem Interpreter innerhalb des `mymodules`-Ordners unter *macOS*:

Wechseln Sie in den Ordner und geben Sie >>python3<< ein, um den Interpreter zu starten.

Das Modul importieren.

Es klappt: Wir können die Funktionen des Moduls benutzen.

```
Datei Bearbeiten Fenster Hilfe Cupertino #1
$ cd mymodules

mymodules$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> quit()

mymodules$
```

Beenden Sie den Python-Interpreter und kehren Sie zur Eingabeaufforderung Ihres Betriebssystems zurück.

ImportErrors treten unabhängig vom Betriebssystem auf

Wenn Sie glauben, die Ausführung auf einem Nicht-*Windows*-System würde das Problem lösen, sollten Sie noch einmal nachdenken: Sobald wir in einen anderen Ordner wechseln, tritt der gleiche `ImportError` auch bei Unix-artigen Systemen auf.

Wechseln Sie in einen anderen Ordner (wir verwenden hier die oberste Ordnerebene).

Python 3 erneut starten.

Wir versuchen, das Modul zu importieren ...

... aber diesmal erhalten wir eine Fehlermeldung!

```
Datei Bearbeiten Fenster Hilfe Cupertino #2
mymodules$ cd

$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

$
```

Wie im *Windows*-Beispiel befindet sich die `vsearch.py`-Datei nicht länger im aktuellen Arbeitsverzeichnis des Interpreters, da wir jetzt in einem anderen Ordner als `mymodules` arbeiten. Dadurch wird das Modul nicht mehr gefunden, kann also auch nicht importiert werden, wodurch der Interpreter einen `ImportError` ausgibt. Dieses Problem tritt unabhängig vom verwendeten Betriebssystem auf.

Es gibt keine Dummen Fragen

F: Können wir den Ort nicht einfach mit angeben, z. B. als `import C:\mymodules\vsearch` unter Windows oder vielleicht `import /mymodules/vsearch` auf UNIX-artigen Systemen?

A: Das geht leider nicht. Auch wenn es verlockend klingt, können Sie Pfade in Pythons `import`-Anweisung nicht auf diese Weise verwenden. Außerdem wollen Sie ja eigentlich auch keine hartcodierten Pfadangaben in Ihren Programmen, weil sich Pfade (aus vielen Gründen) oft ändern können. Am besten vermeiden Sie hartcodierte Pfade in Ihrem Code komplett.

F: Wie teile ich dem Interpreter dann mit, wo er die Module findet, wenn ich keine Pfade angeben kann?

A: Kann der Interpreter Ihr Modul nicht im aktuellen Arbeitsverzeichnis finden, sieht er an Orten wie **site-packages** und in der Standardbibliothek nach (mehr zu site-packages auf der folgenden Seite). Wenn Sie Ihr Modul an einem der Orte für die **site-packages** ablegen können, kann der Interpreter sie dort finden (unabhängig vom tatsächlichen Pfad).

Ein Modul unter site-packages installieren


Modul

Vor einigen Seiten haben wir gesagt, dass **site-packages** einer von drei Orten ist, an denen der Interpreter beim importieren nach Modulen sucht.

2 **die Orte, an denen der Interpreter nach site-packages sucht**
Das sind die Verzeichnisse (Ordner), in denen die Python-Module von Drittherstellern (zum Beispiele Ihre) installiert werden.

Die Bereitstellung und Unterstützung von Modulen von Drittherstellern macht einen wichtigen Teil von Pythons Strategie zur Wiederverwendung von Code aus. Daher überrascht es nicht, dass der Interpreter bereits die nötigen Fähigkeiten zum Hinzufügen von Modulen zu Ihrer Python-Installation enthält.

Beachten Sie, dass die Module der Standardbibliothek von den Python-Core-Entwicklern gepflegt und weiterentwickelt werden. Diese Modulsammlung ist für die häufige Benutzung ausgelegt, sollte aber nicht verändert werden. Daher sollten Sie der Standardbibliothek keine eigenen Module hinzufügen oder welche daraus entfernen. Hierfür sind die **site-packages**-Verzeichnisse gedacht. Dort ist das Hinzufügen und Entfernen eigener Module sogar erwünscht. Die Python beiliegenden Werkzeuge machen dieses Vorgehen daher besonders einfach.

Die Verwendung von »setuptools« zur Installation in site-packages

Ab Version 3.4 enthält Pythons Standardbibliothek ein Modul mit dem Namen `setuptools`, mit dem beliebige Module in den `site-packages`-Ordnern installiert werden können. Auch wenn die Details zur Weitergabe von Modulen anfänglich komplex erscheinen, wollen wir eigentlich nur unser `vsearch`-Modul in `site-packages` installieren. Und das ist mit `setuptools` problemlos möglich. Drei Schritte sind dafür nötig:

1 **Erstellen Sie eine Distributionsbeschreibung.**
Hiermit wird das Modul identifiziert, das `setuptools` installieren soll.

2 **Erstellen Sie eine Distributionsdatei.**
Wir benutzen Python auf der Kommandozeile, um eine weitergabefähige Distributionsdatei zu erzeugen, die den Code unseres Moduls enthält.

3 **Installieren Sie die Distributionsdatei.**
Wir benutzen Python erneut auf der Kommandozeile, um die Distributionsdatei (die unser Modul enthält) in `site-packages` zu installieren.

In Schritt 1 müssen wir (mindestens) zwei beschreibende Dateien für unser Modul anlegen: `setup.py` and `README.txt`. Das sehen wir uns mal im Detail an.

Python 3.4 (oder neuer) macht die Verwendung von setuptools zu einem Kinderspiel. Wenn Sie nicht bereits mit Version 3.4 (oder neuer) arbeiten, sollten Sie jetzt übers Aktualisieren nachdenken.

Die erforderlichen Setup-Dateien erstellen

- [ ] Die Distributionsbeschreibung erstellen.
- [ ] Die Distributionsdatei erstellen.
- [ ] Die Distributionsdatei installieren.

Wir werden diese Punkte beim Durcharbeiten nacheinander abhaken.

Wenn wir den Schritten vom unteren Ende der vorigen Seite folgen, erhalten wir am Ende ein **Distributionspaket** für unser Modul. Dieses Paket ist eine einzelne komprimierte Datei, die alles enthält, was für die Installation unseres Moduls in site-packages gebraucht wird.

Für Schritt 1, *Distributionsbeschreibung erstellen*, müssen Sie im selben Ordner, in dem sich `vsearch.py` befindet, zwei Dateien erstellen, und zwar unabhängig von Ihrem gerade genutzten Betriebssystem. Die erste Datei heißt grundsätzlich `setup.py`. Sie enthält eine detaillierte Beschreibung unseres Moduls.

Unten sehen Sie die von uns erstellte **setup.py**-Datei, mit der das Modul in `vsearch.py` beschrieben wird. Sie enthält zwei Zeilen Python-Code: Die erste Zeile importiert die `setup`-Funktion aus dem `setuptools`-Modul, die zweite Zeile ruft die `setup`-Funktion auf.

Der `setup`-Funktion kann eine Vielzahl von Argumenten übergeben werden, von denen viele allerdings optional sind. Um die Lesbarkeit zu erhöhen, haben wir unseren Aufruf von `setup` auf mehrere Zeilen verteilt. Wir nutzen hier Pythons Unterstützung für Schlüsselwortargumente, um zu verdeutlichen, welcher Wert in diesem Aufruf welchem Argument zugewiesen wird. Die wichtigsten Argumente haben wir hervorgehoben: Das erste (`name`) benennt die Distribution, das zweite (`py_modules`) listet die `.py`-Dateien auf, die beim Erstellen des Distributionspakets enthalten sein sollen:

Die Funktion »setup« aus dem Modul »setuptools« wird importiert.

```
from setuptools import setup

setup(
      name='vsearch',
      version='1.0',
      description='The Head First Python Search Tools',
      author='HF Python 2e',
      author_email='hfpy2e@gmail.com',
      url='headfirstlabs.com',
      py_modules=['vsearch'],
)
```

Das Argument »name« identifiziert die Distribution. Üblicherweise wird die Distribution nach dem Modul benannt.

Dies ist ein Aufruf der »setup«-Funktion. Wir haben die Argumente auf mehrere Zeilen verteilt.

Dies ist eine Liste von ».py«-Dateien, die im Paket enthalten sein müssen. In unserem Beispiel ist das nur »vsearch«.

Neben `setup.py` benötigt der Mechanismus von `setuptools` eine »readme«-(»Lies mich!«-)Datei. Hier können Sie Ihr Paket in Textform beschreiben. Die Datei selbst ist erforderlich, ihr Inhalt jedoch nicht. Sie können also (im Moment) einfach im selben Ordner wie `setup.py` eine leere Datei namens `README.txt` erstellen, um die Anforderungen für eine zweite Datei in Schritt 1 zu erfüllen.

Eine Distributionsdatei erstellen

- [x] Die Distributionsbeschreibung erstellen.
- [ ] Die Distributionsdatei erstellen.
- [ ] Die Distributionsdatei installieren.

Ihr `mymodules`-Ordner sollte jetzt drei Dateien enthalten: `vsearch.py`, `setup.py` und `README.txt`.

Jetzt können wir das Distributionspaket bauen. Dies ist Schritt 2 unserer Liste: *Die Distributionsdatei erstellen*. Das erledigen wir auf der Kommandozeile. Der Schritt an sich ist nicht schwer, allerdings müssen Sie je nach Betriebssystem (*Linux*, *Unix* oder *Windows*) unterschiedliche Befehle eingeben.

Eine Distributionsdatei unter Windows erstellen

Wenn Sie unter *Windows* arbeiten, öffnen Sie im Ordner, der Ihre drei Dateien enthält, eine Eingabeaufforderung und geben den folgenden Befehl ein:

Python 3 unter Windows ausführen.

```
C:\Users\Head First\mymodules> py -3 setup.py sdist
```

Den Code in >>setup.py<< ausführen ...

... und >>sdist<< als Argument übergeben.

Nachdem Sie diesen Befehl eingegeben haben, geht der Python-Interpreter sofort an die Arbeit. Auf dem Bildschirm erscheint eine Vielzahl von Meldungen (die wir hier in gekürzter Form zeigen):

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
creating dist
creating 'dist\vsearch-1.0.zip' and adding 'vsearch-1.0' to it
adding 'vsearch-1.0\PKG-INFO'
adding 'vsearch-1.0\README.txt'
      ...
adding 'vsearch-1.0\vsearch.egg-info\top_level.txt'
removing 'vsearch-1.0' (and everything under it)
```

Wenn Sie diese Meldung sehen, ist alles in Ordnung. Erhalten Sie eine Fehlermeldung, stellen Sie sicher, dass Sie mindestens Python 3.4 benutzen und dass Ihre >>setup.py<<-Datei mit unserer identisch ist.

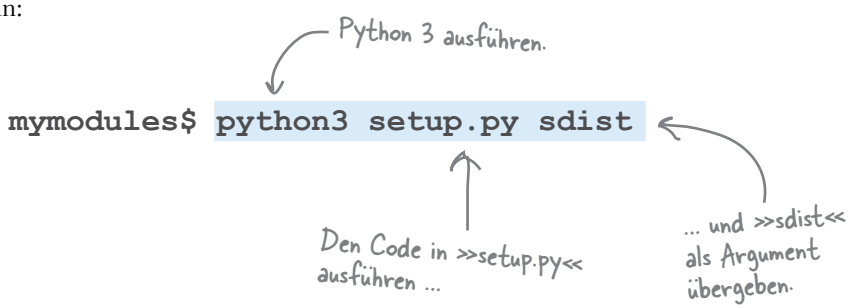
Wenn die *Windows*-Eingabeaufforderung wieder erscheint, wurden die drei Dateien zu einer **Distributionsdatei** kombiniert. Dies ist eine installierbare Datei, die den Quellcode Ihres Moduls enthält und, in unserem Fall, `vsearch-1.0.zip` heißt.

Sie finden die neu erstellte ZIP-Datei in einem Ordner namens `dist`, der innerhalb Ihres aktuellen Arbeitsverzeichnisses (in unserem Fall `mymodules`) ebenfalls von `setuptools` erstellt wurde.

Distributionsdateien auf Unix-artigen Betriebssystemen

- [x] Die Distributionsbeschreibung erstellen.
- [ ] Die Distributionsdatei erstellen.
- [ ] Die Distributionsdatei installieren.

Auch wenn Sie nicht unter *Windows* arbeiten, können Sie die Distributionsdatei fast auf die gleiche Weise erstellen, wie auf der vorigen Seite beschrieben. Sobald Sie die drei nötigen Dateien (`setup.py`, `README.txt` und `vsearch.py`) im selben Ordner gespeichert haben, geben Sie auf der Kommandozeile Ihres Systems den folgenden Befehl ein:



Wie unter *Windows* erzeugt dieser Befehl eine Reihe von Meldungen auf dem Bildschirm:

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
running check
creating vsearch-1.0
creating vsearch-1.0/vsearch.egg-info
      ...
creating dist
Creating tar archive
removing 'vsearch-1.0' (and everything under it)
```

Die Meldungen unterscheiden sich leicht von denen unter Windows. Wenn Sie diese Meldung erhalten, ist alles in Ordnung. Falls nicht, sollten Sie (wie in Windows) alles noch einmal überprüfen.

Wenn die Kommandozeile Ihres Betriebssystems wieder erscheint, wurden Ihre drei Dateien zu einer **Distributionsdatei** kombiniert. Dies ist eine installierbare Datei, die den Quellcode Ihres Moduls enthält und in diesem Fall den Namen `vsearch-1.0.tar.gz` trägt.

Sie finden die neu erstellte Archivdatei in einem Ordner namens `dist`, der ebenfalls von `setuptools` in Ihrem aktuellen Arbeitsverzeichnis (hier: `mymodules`) erstellt wurde.

Nach Erstellen der Distributionsdatei (als ZIP-Datei oder als komprimiertes tar-Archiv) können Sie Ihr Modul nun unter site-packages installieren.

Pakete mit »pip« installieren

- [x] Die Distributionsbeschreibung erstellen.
- [x] Die Distributionsdatei erstellen.
- [ ] Die Distributionsdatei installieren.

Nachdem die Distributionsdatei jetzt existiert, ist es nun Zeit für Schritt 3: *Die Distributionsdatei installieren*. Wie bei vielen Dingen dieser Art besitzt Python eine Reihe von Werkzeugen, die die Arbeit stark vereinfachen. Ab Python 3.4 (und neuer) gibt es das Werkzeug pip. Es ist *das* Installationsprogramm für Pakete schlechthin.

Schritt 3 unter Windows

Finden Sie die neu erstellte ZIP-Datei im `dist`-Ordner (die Datei heißt `vsearch-1.0.zip`). Drücken Sie im *Windows-Explorer* die Umschalt-Taste und klicken Sie gleichzeitig mit der rechten Maustaste, um ein Kontextmenü anzuzeigen. Wählen Sie den Befehl *Eingabeaufforderung hier öffnen*. Es öffnet sich eine neue Windows-Eingabeaufforderung. Um Schritt 3 zu beenden, geben Sie dort die folgende Zeile ein:

Führen Sie Python 3 mit dem pip-Modul aus und weisen Sie pip an, die angegebene ZIP-Datei zu installieren.

```
C:\Users\...\dist> py -3 -m pip install vsearch-1.0.zip
```

Bricht dieser Befehl mit einer Fehlermeldung über Benutzerrechte ab, müssen Sie die Windows-Eingabeaufforderung als *Windows*-Administrator erneut starten und es noch einmal probieren.

Ist der obige Befehl erfolgreich, sehen Sie die folgenden Meldungen auf dem Bildschirm:

```
Processing c:\users\...\dist\vsearch-1.0.zip
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Erfolg!

Schritt 3 auf Unix-artigen Betriebssystemen

Öffnen Sie unter *Linux*, *Unix* bzw. *macOS* ein Terminal im neu erzeugten `dist`-Ordner und geben Sie auf der Kommandozeile den folgenden Befehl ein:

Führen Sie Python 3 mit dem pip-Modul aus und weisen Sie pip an, die angegebene komprimierte tar-Datei zu installieren.

```
.../dist$ sudo python3 -m pip install vsearch-1.0.tar.gz
```

Wurde der obige Befehl erfolgreich ausgeführt, sehen Sie die folgenden Meldungen auf dem Bildschirm:

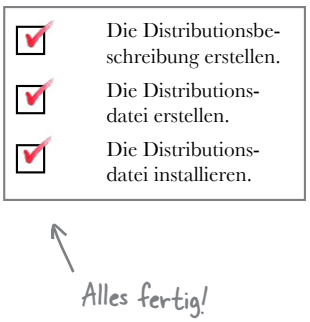
```
Processing ./vsearch-1.0.tar.gz
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Erfolg!

Wir verwenden hier den Befehl »sudo«, um sicherzustellen, dass wir die korrekten Rechte für die Installation haben.

Das vsearch-Modul ist nun als Teil von site-packages installiert.

Module: Was wir bereits wissen



Nach der Installation von `vsearch` können wir in beliebigen Programmen die Anweisung `import vsearch` benutzen, um sicherzugehen, dass der Interpreter die Funktionen des Moduls bei Bedarf findet.

Wenn wir den Code des Moduls später aktualisieren wollen, können wir die drei Schritte wiederholen, um das Modul erneut in `site-packages` zu installieren. Erstellen Sie eine neue Version Ihres Moduls, vergessen Sie nicht, in `setup.py` eine neue Versionsnummer anzugeben.

Nehmen wir uns einen Moment Zeit, um zusammenzufassen, was wir bereits über Module wissen:

Punkt für Punkt

- Ein Modul besteht aus einer oder mehreren Funktionen, die in einer Datei gespeichert wurden.
- Sie können ein Modul teilen, indem Sie sicherstellen, dass es sich immer im *aktuellen Arbeitsverzeichnis* des Interpreters befindet (das ist möglich, aber umständlich) oder an einem der Orte für *site-packages* (eine wesentlich bessere Wahl).
- Nach der Ausführung der drei Arbeitsschritte für `setuptools` ist sichergestellt, dass Ihr Modul unter *site-packages* installiert wurde. Dadurch können Sie das Modul und seine Funktionen importieren, egal wo sich das *aktuelle Arbeitsverzeichnis* gerade befindet.

Ihren Code mit anderen teilen

Nachdem Sie eine Distributionsdatei erstellt haben, können Sie Ihre Datei mit anderen Python-Programmierern teilen, die sie dann ebenfalls per `pip` installieren dürfen. Sie können Ihre Datei auf zwei Arten weitergeben:

Um Ihr Modul informell mit anderen zu teilen, können Sie es auf beliebige Weise an wen auch immer weitergeben (z. B. per E-Mail, auf einem USB-Stick oder per Download von Ihrer Website). Es ist Ihre Entscheidung.

Um Ihr Modul formell mit anderen zu teilen, können Sie die Distributionsdatei in Pythons zentral verwaltetes webbasiertes Repository namens PyPI (den *Python Package Index,* ausgesprochen als »Pei-Pi-Ei«) hochladen. Diese Website existiert, damit Python-Programmierer alle möglichen Python-Module von Drittherstellern austauschen können. Weitere Informationen finden Sie auf der PyPI-Website unter: ***https://pypi.python.org/pypi***. Mehr Informationen zum Hochladen und Teilen Ihrer Distributionsdateien per PyPI finden Sie in der Onlinedokumentation, die von der *Python Packaging Authority* gepflegt wird, unter: ***https://www.pypa.io***. Das ist zwar nicht schwer, aber die Details würden den Rahmen dieses Buchs sprengen.

Und damit sind wir mit unserer Einführung zu Funktionen und Modulen fast am Ende. Ein kleines Rätsel müssen wir allerdings noch lösen (das dauert nicht mehr als fünf Minuten). Blättern Sie um, wenn Sie so weit sind.

Jeder Python-Programmierer kann pip verwenden, um Ihr Modul zu installieren.

Das Problem der ungezogenen Funktionsargumente

Tom und Sarah haben gerade dieses Kapitel durchgearbeitet und streiten sich jetzt über das Verhalten von Funktionsargumenten.

Tom ist davon überzeugt, dass die Argumente als **Werteparameter** an die Funktion übergeben werden, und hat eine kleine Funktion namens `double` geschrieben, um das zu beweisen. Toms `double`-Funktion kann beliebige Daten übernehmen.
Hier ist Toms Code:


```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)
```

Sarah ist dagegen sicher, dass die Argumente als **Referenzparameter** übergeben werden. Sie hat ebenfalls eine kleine Funktion namens `change` geschrieben, die mit Listen arbeitet und ihren Standpunkt beweisen soll. Hier der Code:

```
def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Da Tom und Sarah bis jetzt die besten Programmierkumpel waren, wollen wir natürlich nicht, dass darüber ein Streit entsteht. Um zu sehen, wer recht hat, experimentieren wir am >>>-Prompt. Ist es »Werteparameter«-Tom oder »Referenzparameter«-Sarah? Es können ja schließlich nicht beide recht haben – oder? Das Rätsel hinter dieser Situation lässt sich auf eine Frage reduzieren: ***Unterstützen Funktionsaufrufe Werteparameter oder Referenzparameter?***

Geek-Bits

Falls Sie eine kleine Gedächtnisstütze brauchen: Bei der **Übergabe eines Arguments als Wertparameter** wird anstelle des Arguments der Wert einer Variablen übergeben. Ändert sich der Wert in der Suite der Funktion, hat dies keinen Einfluss auf den Wert der Variablen im aufrufenden Code. Stellen Sie sich das Argument als *Kopie* des ursprünglichen Werts der Variablen vor. Bei der **Übergabe eines Arguments als Referenzparameter** (anhand der Speicheradresse des Werts) wird stattdessen ein *Verweis* auf die Variable im aufrufenden Code übergeben. Ändert sich die Variable in der Suite der Funktion, verändert sich der Wert auch im aufrufenden Code. Stellen Sie sich das Argument als *Alias* auf die Originalvariable vor.

Demonstration von Werteparametern

Um herauszufinden, worüber Tom und Sarah sich streiten, wollen wir beide Funktionen in ein eigenes Modul verpacken, das wir `mystery.py` nennen. Hier sehen Sie das Modul im IDLE-Editorfenster:

Beide Funktionen sind ähnlich. Jede übernimmt ein einzelnes Argument, zeigt es auf dem Bildschirm an, verändert seinen Wert und gibt ihn danach erneut auf dem Bildschirm aus.

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Diese Funktion verdoppelt den übergebenen Wert.

Diese Funktion hängt einer übergebenen Liste einen String an.

Ln: 11 Col: 0

Tom sieht dieses Modul auf dem Bildschirm, setzt sich hin, drückt F5 und gibt die folgenden Anweisungen am >>>-Prompt ein. Danach lehnt er sich zurück, verschränkt die Arme und sagt: »Siehst du? Ich habe doch gesagt, dass Werteparameter benutzt werden.« Werfen Sie einen Blick auf Toms Interaktionen mit der Shell:

Tom ruft die »double«-Funktion dreimal auf: einmal mit einem Integer-Wert, dann mit einem String und schließlich mit einer Liste.

```
>>> num = 10
>>> double(num)
Before:  10
After:   20
>>> num
10
>>> saying = 'Hello '
>>> double(saying)
Before:  Hello
After:   Hello Hello
>>> saying
'Hello '
>>> numbers = [ 42, 256, 16 ]
>>> double(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 42, 256, 16]
>>> numbers
[42, 256, 16]
```

Alle Aufrufe bestätigen, dass der als Argument übergebene Wert in der Suite der Funktion verändert wird, während der Wert auf der Shell gleich bleibt. Offenbar werden die Funktionsargumente als Werteparameter übergeben.

Demonstration von Referenzparametern

Ungerührt von Toms scheinbarem Sieg, setzt Sarah sich ebenfalls hin und bereitet sich auf die Interaktion mit der Shell vor. Hier noch einmal der Code im IDLE-Editorfenster – jetzt ist Sarahs `change`-Funktion bereit für den Aufruf:

Dies ist das >>mystery.py<<- Modul.

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Toms Funktion.

Sarahs Funktion.

Ln: 11 Col: 0

Sarah gibt ein paar Codezeilen am >>>-Prompt ein, lehnt sich zurück, verschränkt die Arme und sagt zu Tom: »Wie erklärst du dieses Verhalten, wenn Python tatsächlich nur Werteparameter unterstützt?« Tom ist sprachlos.

Hier sehen wir Sarahs Interaktion mit der Shell:

Mit den gleichen Listendaten wie Tom ruft Sarah jetzt ihre >>change<<- Funktion auf.

```
>>> numbers = [ 42, 256, 16 ]
>>> change(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 'More data']
>>> numbers
[42, 256, 16, 'More data']
```

Sehen Sie, was passiert ist! Jetzt hat sich der Wert des Arguments in der Funktion und auch in der Shell verändert. Demnach würden Python-Funktionen *auch* Referenzparameter unterstützen.

Dieses Verhalten *ist* seltsam.

Toms Funktion zeigt, dass Werteparameter verwendet werden, während Sarahs Code eindeutig Referenzparameter benutzt.

Wie kann das sein? Was geht hier vor? Unterstützt Python etwa *beide Formen*?

Gelöst: Das Problem der ungezogenen Funktionsargumente

Unterstützen Pythons Funktionsargumente Werteparameter oder Referenzparameter?

Das ist der Knaller: Tom und Sarah haben beide recht. Je nach Situation unterstützen Pythons Funktionsargumente **sowohl** Werteparameter *als auch* Referenzparameter.

Python-Variablen verhalten sich anders als Variablen in anderen Programmiersprachen. Variablen sind **Objektreferenzen**. Das heißt: Die Variable enthält nicht den Wert selbst, sondern einen Verweis auf dessen Speicheradresse. Pythons Funktionen unterstützen also tatsächlich *Referenzparameter*.

Ja nachdem, welchen Typ das verwendete Objekt hat, kann sich die Semantik (Wert oder Referenz) unterscheiden. Warum verhalten sich die Argumente in Toms und Sarahs Funktionen aber einmal als Werteparameter `und` ein anderes Mal als Referenzparameter? Das ist nur scheinbar der Fall. Tatsächlich untersucht der Interpreter, welchen Typ der Wert hat, auf den die Objektreferenz (bzw. die Speicheradresse) verweist. Ist es ein **mutabler** (veränderlicher) Wert, wird die Referenzparameter-Semantik verwendet. Ist der Wert dagegen **immutabel** (unveränderlich), wird er als Werteparameter übergeben. Wir wollen sehen, was das für Ihre Daten bedeutet.

Listen, Dictionaries und Sets (die mutabel sind) werden immer als Referenz an die Funktion übergeben. Änderungen, die innerhalb der Suite an der Datenstruktur vorgenommen werden, wirken sich auf den aufrufenden Code aus. Schließlich sind die Daten mutabel.

Strings, Integer und Tupel (die immutabel sind) werden immer als Werteparameter an die Funktion übergeben. Änderungen an der Variablen finden nur innerhalb der Funktion statt (sind »privat«) und haben keine Auswirkung auf den aufrufenden Code. Da die Daten immutabel sind, können sie nicht verändert werden.

Das alles ergibt einen Sinn, bis Sie sich die folgende Codezeile ansehen:

```
arg = arg * 2
```

Scheinbar hat diese Codezeile die übergebene Liste in der Suite der Funktion verändert. Bei der Ausgabe der Liste auf der Shell sind aber keine Änderungen zu sehen (wodurch Tom fälschlicherweise glaubte, dass alle Argumente als Werteparameter übergeben werden). Wie kann das sein? Oberflächlich betrachtet, scheint dies ein Interpreter-Bug zu sein. Schließlich haben wir gerade gesagt, dass Veränderungen an mutablen Werten sich auf den aufrufenden Code auswirken, was hier aber nicht passiert. Toms Funktion hat die `numbers`-Liste im aufrufenden Code *nicht verändert*, obwohl Listen mutabel sind. Was soll das denn?

Um es zu verstehen, müssen Sie bedenken, dass die obige Codezeile eine **Zuweisungsanweisung** ist. Dabei passiert Folgendes: Zuerst wird der Code auf der rechten Seite des Gleichheitszeichens (=) ausgewertet. Durch die Ausführung von `arg * 2` wird ein neuer Wert erzeugt, der einer neuen Objektreferenz zugewiesen wird. Diese wird der Variablen `arg` zugewiesen. Dadurch wird die Objektreferenz überschrieben, die zuvor innerhalb der Funktionssuite in `arg` gespeichert wurde. Im aufrufenden Code existiert die »alte« Objektreferenz allerdings noch. Dieser Wert hat sich nicht verändert, wodurch die Shell weiterhin die Originalliste und nicht das von Toms Code erstellte Duplikat sieht. Sarahs Code verwendet dagegen die `append`-Methode an der existierenden Liste. Da hier keine Zuweisung stattfindet, werden auch keine Objektreferenzen überschrieben. Sarahs Code verändert die Liste auch auf der Shell, weil sowohl die Liste in der Funktionssuite als auch die Liste im aufrufenden Code auf *die gleiche* Objektreferenz verweisen.

Nachdem wir das Rätsel gelöst haben, sind wir fast schon bereit für Kapitel 5. Es gibt nur noch eine wichtige Sache.

Kann ich auf PEP 8-Konformität testen?

Ja, das ist möglich.

Allerdings nicht mit Python allein, denn der Interpreter selbst hat keine Möglichkeit, Code auf PEP-8-Konformität zu überprüfen. Aber es gibt Werkzeuge von Drittherstellern, die das können.

Bevor wir mit Kapitel 5 weitermachen, wollen wir einen kleinen Umweg nehmen und uns ein Werkzeug ansehen, das Ihnen helfen kann, Ihren Code PEP 8-konform zu halten.

Vorbereitungen für den Test auf PEP 8-Konformität

Machen wir einen kleinen Umweg, um unseren Code auf PEP 8-Konformität zu testen.

Die Python-Gemeinschaft hat jede Menge Zeit damit verbracht, um Entwicklerwerkzeuge zu erstellen, um das Leben der Programmierer etwas zu erleichtern. Eines dieser Werkzeuge ist **pytest**, ein *Test-Framework*, das das Testen von Python-Programmen erleichtern soll. Egal welche Art von Tests Sie schreiben, **pytest** kann dabei helfen. Außerdem können Sie die Fähigkeiten von **pytest** anhand von Plug-ins erweitern.

Eines dieser Plug-ins ist **pep8**, das das **pytest**-Framework benutzt, um zu testen, ob sich Ihr Code an die PEP 8-Richtlinien hält.

Weitere Informationen über pytest finden Sie unter http://doc.pytest.org/en/latest/.

Ein weiterer Blick auf unseren Code

Rufen wir uns den Code in `vsearch.py` noch einmal ins Gedächtnis, bevor wir ihn an die Kombination aus **pytest** und **pep8** verfüttern, um herauszufinden, wie PEP 8-konform er ist. Hierfür müssen Sie zunächst die beiden Entwicklerwerkzeuge installieren, da sie Python nicht beiliegen (auf der folgenden Seite).

Hier noch einmal der Code aus dem vsearch.py-Modul, der auf PEP 8-Konformität getestet werden soll:

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Dieser Code steht in >>vsearch.py<<.

pytest und das pep8-Plug-in installieren

Weiter oben in diesem Kapitel haben wir mithilfe des `pip`-Werkzeugs Ihr `vsearch.py`-Modul im Python-Interpreter auf Ihrem Computer installiert.

Hierfür müssen Sie auf der Kommandozeile Ihres Systems arbeiten (und eine Internetverbindung haben). Im folgenden Kapitel werden Sie mit `pip` eine Bibliothek eines Drittherstellers installieren. Im Moment wollen wir `pip` verwenden, um das `pytest`-Test-Framework und das **pep8**-Plug-in zu installieren.

Die Entwicklerwerkzeuge zum Testen installieren

Die folgenden Abbildungen zeigen die Meldungen, die unter *Windows* angezeigt werden. Dort wird Python 3 mit dem Befehl `py -3` aufgerufen. Benutzen Sie Linux oder macOS, ersetzen Sie diesen Befehl durch `sudo python3`. Für die Installation von **pytest** anhand von `pip` unter *Windows* müssen Sie den Befehl als Administrator auf der Kommandozeile eingeben (suchen Sie nach `cmd.exe`, führen Sie einen Rechtsklick darauf aus und wählen Sie im Kontextmenü den Befehl *Als Administrator ausführen*):

```
py  -3  -m  pip  install  pytest
```

Im Administrator-modus starten ...

... dann den >>pip<<-Befehl verwenden, um >>pytest<< zu installieren ...

... und schließlich überprüfen, ob alles erfolgreich installiert wurde.

```
Administrator: Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\system32>py -3 -m pip install pytest
Collecting pytest
  Downloading pytest-2.8.7-py2.py3-none-any.whl (151kB)
    100% |################################| 155kB 1.3MB/s
Collecting colorama (from pytest)
  Downloading colorama-0.3.6-py2.py3-none-any.whl
Collecting py>=1.4.29 (from pytest)
  Downloading py-1.4.31-py2.py3-none-any.whl (81kB)
    100% |################################| 86kB 131kB/s
Installing collected packages: colorama, py, pytest
Successfully installed colorama-0.3.6 py-1.4.31 pytest-2.8.7

C:\Windows\system32>
```

Wenn Sie sich die Meldungen von `pip` ansehen, stellen Sie fest, dass außerdem zwei Abhängigkeiten von **pytest** installiert wurden (**colorama** und **py**). Das passiert auch, wenn Sie `pip` für die Installation des **pep8**-Plug-ins nutzen. Hier der nötige Befehl:

```
py  -3  -m  pip  install  pytest-pep8
```

Nicht vergessen: Wenn Sie nicht unter Windows arbeiten, ersetzen Sie >>py -3<< durch >>sudo python3<<.

Geben Sie, immer noch als Administrator, diesen Befehl ein, der das >>pep8<<-Plug-in installiert.

Dieser Befehl war ebenfalls erfolgreich, und die benötigten Abhängigkeiten wurden auch installiert.

```
Administrator: Command Prompt
C:\Windows\system32>py -3 -m pip install pytest-pep8
Collecting pytest-pep8
  Downloading pytest-pep8-1.0.6.tar.gz
Collecting pytest-cache (from pytest-pep8)
  Downloading pytest-cache-1.0.tar.gz
Requirement already satisfied (use --upgrade to upgrade): pytest>=2.4.2 in c:\pr
ogram files\python 3.5\lib\site-packages (from pytest-pep8)
Collecting pep8>=1.3 (from pytest-pep8)
  Downloading pep8-1.7.0-py2.py3-none-any.whl (41kB)
    100% |################################| 45kB 174kB/s
Collecting execnet>=1.1.dev1 (from pytest-cache->pytest-pep8)
  Downloading execnet-1.4.1-py2.py3-none-any.whl (40kB)
    100% |################################| 40kB 174kB/s
Requirement already satisfied (use --upgrade to upgrade): py>=1.4.29 in c:\progr
am files\python 3.5\lib\site-packages (from pytest>=2.4.2->pytest-pep8)
Requirement already satisfied (use --upgrade to upgrade): colorama in c:\program
 files\python 3.5\lib\site-packages (from pytest>=2.4.2->pytest-pep8)
Collecting apipkg>=1.4 (from execnet>=1.1.dev1->pytest-cache->pytest-pep8)
  Downloading apipkg-1.4-py2.py3-none-any.whl
Installing collected packages: apipkg, execnet, pytest-cache, pep8, pytest-pep8
  Running setup.py install for pytest-cache ... done
  Running setup.py install for pytest-pep8 ... done
Successfully installed apipkg-1.4 execnet-1.4.1 pep8-1.7.0 pytest-cache-1.0 pyte
st-pep8-1.0.6

C:\Windows\system32>
```

Wie PEP 8-konform ist unser Code?

Nachdem **pytest** und **pep8** installiert sind, können wir unseren Code auf PEP 8-Konformität testen. Unabhängig vom Betriebssystem müssen Sie nun den gleichen Befehl eingeben (da sich nur die Installationsanweisungen unterscheiden).

Der **pytest**-Installationsprozess hat auf Ihrem Computer ein neues Programm namens `py.test` installiert. Mit diesem Programm können wir den Code in `vsearch.py` auf PEP 8-Konformität testen. Sorgen Sie dafür, dass das aktuelle Arbeitsverzeichnis dem Ordner entspricht, in dem die `vsearch.py`-Datei liegt. Dann geben Sie folgenden Befehl ein:

```
py.test  --pep8  vsearch.py
```

Au weia. Die roten Meldungen bedeuten bestimmt nichts Gutes, oder?

Hier sind die Ausgaben, die wir auf unserem *Windows*-Computer erhalten haben:

```
C:\Windows\system32\cmd.exe

E:\_NewBook\ch04>py.test --pep8 vsearch.py
============================= test session starts =============================
platform win32 -- Python 3.5.0, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: E:\_NewBook\ch04, inifile:
plugins: pep8-1.0.6
collected 1 items

vsearch.py F

================================== FAILURES ===================================
_____________________________ PEP8-check ______________________________
E:\_NewBook\ch04\vsearch.py:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                        ^
E:\_NewBook\ch04\vsearch.py:3:56: W291 trailing whitespace
    """Return any vowels found in a supplied phrase."""
                                                       ^
E:\_NewBook\ch04\vsearch.py:7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
^
E:\_NewBook\ch04\vsearch.py:7:26: E231 missing whitespace after ':'
def search4letters(phrase:str, letters:str='aeiou') -> set:
                         ^
E:\_NewBook\ch04\vsearch.py:7:39: E231 missing whitespace after ':'
def search4letters(phrase:str, letters:str='aeiou') -> set:
                                       ^

========================== 1 failed in 0.05 seconds ===========================

E:\_NewBook\ch04>
```

Hoppla! Scheinbar gibt es einige **Fehler** (Failures). Offenbar ist unser Code nicht so PEP 8-konform, wie er sein könnte.

Nehmen Sie sich etwas Zeit, um die hier gezeigten Meldungen zu lesen (oder auf Ihrem Bildschirm, falls Sie den Befehl direkt ausprobieren). Sämtliche »Fehler« beziehen sich offenbar auf *Whitespace*-Zeichen. (Damit sind beispielsweise Leerzeichen, Tabulatorzeichen, Zeilenumbrüche usw. gemeint.) Sehen wir uns die Zeilen etwas genauer an.

Die Fehlermeldungen verstehen

Zusammen haben **pytest** und das **pep8**-Plug-in fünf Probleme in unserem `vsearch.py`-Code gefunden.

Das erste Problem hat damit zu tun, dass wir beim Erstellen der Annotationen für unsere Funktionsargumente nach dem Doppelpunkt (`:`) kein Leerzeichen eingefügt haben. Das ist an drei Stellen der Fall. In der ersten Fehlermeldung von **pytest** bezeichnet das Caret-Zeichen (^), wo genau das Problem auftritt:

```
...:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                         ^
```

Hier ist der Fehler.

Hier ist der Fehler.

Wenn Sie sich die zwei Probleme am Ende der Ausgaben von **pytest** ansehen, werden Sie feststellen, dass der Fehler an drei Stellen auftritt: einmal auf Zeile 2 und zweimal auf Zeile 7. Das lässt sich leicht lösen: *Fügen Sie nach dem Doppelpunkt ein einzelnes Leerzeichen ein*.

Das nächste Problem scheint nicht so wichtig zu sein. Dennoch wird ein Fehler ausgegeben, weil die fragliche Codezeile (Zeile 3) eine PEP 8-Richtlinie verletzt, die besagt, dass am Zeilenende keine zusätzlichen Leerzeichen stehen sollten:

```
...:3:56: W291 trailing whitespace
"""Return any vowels found in a supplied phrase."""
                                                  ^
```

Der Fehler.

Hier ist der Fehler.

Das Problem auf Zeile 3 lässt sich schnell beheben: *Entfernen Sie alle nachgestellten Leerzeichen.*

Das letzte Problem (am Anfang von Zeile 7) stellt sich so dar:

```
...7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
^
```

Dieses Problem befindet sich am Anfang von Zeile 7.

Hier ist der Fehler.

Eine PEP 8-Richtlinie gibt Hilfestellung bei der Erzeugung von Funktionen in einem Modul: *Umgeben Sie Funktionen der obersten Ebene und Klassendefinitionen mit zwei Leerzeilen*. In unserem Code befinden sich die Funktionen `search4vowels` und `search4letters` beide auf der »obersten Ebene« der `vsearch.py`-Datei und werden durch eine Leerzeile voneinander getrennt. Um PEP 8-konform zu sein, sollten hier *zwei* Leerzeilen verwendet werden.

Auch das lässt sich leicht beheben: *Fügen Sie zwischen den beiden Funktionen eine weitere Leerzeile ein*. Lassen Sie uns die Änderungen gleich vornehmen und unseren Code erneut testen.

Übrigens: Unter http://pep8.org/ finden Sie eine wunderschöne Darstellung von Pythons Stilrichtlinien.

PEP 8-Konformität bestätigen

Nachdem wir die Änderungen in `vsearch.py` vorgenommen haben, sieht unser Code nun so aus:

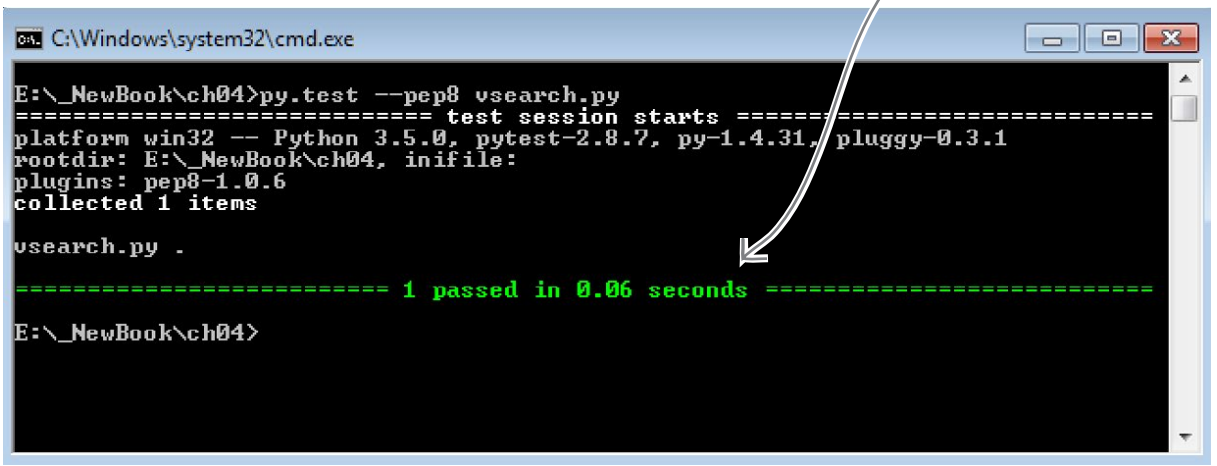
```
def search4vowels(phrase: str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Die PEP 8-konforme Version von >>vsearch.py<<.

Wenn diese Version unseres Code das **pep8**-Plug-in von **pytest** durchläuft, bestätigen die Ausgaben, dass es keine Probleme mit der PEP 8-Konformität mehr gibt. Hier die Ausgaben bei einem erneuten Testlauf (auch hier wieder unter *Windows*):

Grün ist gut – dieser Code hat keine PEP 8-Probleme mehr.



PEP 8-Konformität ist eine gute Sache

Wenn Sie sich beim Anblick dieser Meldungen fragen, was das alles soll (besonders wenn es nur um ein paar Leerzeichen geht), sollten Sie noch einmal überlegen, warum man sich an PEP 8 halten sollte. Die PEP 8-Dokumentation besagt, dass *Lesbarkeit zählt* und dass *Code wesentlich öfter gelesen als geschrieben* wird. Wenn Sie sich beim Programmieren an bestimmte Richtlinien halten, folgt daraus, dass der Code leichter lesbar ist und so aussieht wie alles andere, was der Programmierer bisher gesehen hat. Konsistenz ist eine sehr gute Sache.

Dies ist das Ende des pytest-Umwegs. Bis gleich in Kapitel 5.

Ab diesem Punkt wird sich sämtlicher Code in diesem Buch an die PEP 8-Richtlinien halten (soweit das machbar ist). Versuchen Sie, das auch für Ihren Code sicherzustellen.

Der Code aus Kapitel 4

```
def search4vowels(phrase: str) -> set:
    """Returns the set of vowels found in 'phrase'."""
    return set('aeiou').intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Returns the set of 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Das hier ist der Code des >>vsearch.py<<-Moduls, das die beiden Funktionen >>search4vowels<< und >>search4letters<< enthält.

Das ist die >>setup.py<<-Datei, mit der wir unser Modul in eine installierbare Distribution verwandelt haben.

```
from setuptools import setup

setup(
    name='vsearch',
    version='1.0',
    description='The Head First Python Search Tools',
    author='HF Python 2e',
    author_email='hfpy2e@gmail.com',
    url='headfirstlabs.com',
    py_modules=['vsearch'],
)
```

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)


def change(arg: list):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Und hier ist das >>mystery.py<<-Modul, über das Tom und Sarah in Streit geraten sind. Glücklicherweise konnten wir das Rätsel lösen, und Tom und Sarah sind wieder die dicksten Programmierkumpel.