

Python Tutorial

Udemy Kurs-Handbuch



Einleitung

Python ist eine einfach zu erlernende, leistungsstarke Programmiersprache. Sie verfügt über effiziente Datenstrukturen auf hoher Ebene und einen einfachen, aber effektiven Ansatz für die objektorientierte Programmierung. Pythons elegante Syntax und dynamische Typisierung, zusammen mit seiner interpretierten Natur, machen es zu einer idealen Sprache für Skripting und schnelle Anwendungsentwicklung in vielen Bereichen auf den meisten Plattformen.

Der Python-Interpreter und die umfangreiche Standardbibliothek sind für alle gängigen Plattformen von der Python-Website <https://www.python.org/> in Quell- oder Binärform frei verfügbar und können frei verteilt werden. Die gleiche Website enthält auch Verteilungen und Verweise auf viele kostenlose Python-Module, Programme und Tools von Drittanbietern sowie zusätzliche Dokumentation.

Der Python-Interpreter lässt sich leicht um neue Funktionen und Datentypen erweitern, die in C oder C++ (oder anderen von C aus aufrufbaren Sprachen) implementiert sind. Python eignet sich auch als Erweiterungssprache für anpassbare Anwendungen.

Dieses Tutorial führt den Leser informell in die grundlegenden Konzepte und Funktionen der Sprache und des Systems Python ein. Es ist hilfreich, einen Python-Interpreter für die praktische Arbeit zur Verfügung zu haben, aber alle Beispiele sind in sich geschlossen, sodass das Tutorial auch offline gelesen werden kann.

Dieses Handbuch versucht nicht, umfassend zu sein und deckt nicht jedes einzelne Feature und selbst nicht jedes häufig verwendete Feature ab. Stattdessen stellt es viele der bemerkenswertesten Funktionen von Python vor und gibt dir eine gute Vorstellung vom Stil der Sprache. Nachdem du die Anleitung gelesen hast, kannst du Python-Module und -Programme lesen und schreiben, und du wirst bereit sein, mehr über die verschiedenen Module der Python-Bibliothek zu erfahren, die in der Python Standard Library beschrieben sind.





1	
Einleitung	1
1. Ein kleiner Appetitanreger	7
2. Nutzung des Python Interpreters	10
2.1. Aufruf des Interpreters	10
2.1.1. Argumentübergabe	11
2.1.2. Interaktiver Modus	11
2.2. Der Interpreter und seine Umgebung	13
2.2.1. Quellcode-Kodierung	13
Fußnoten	14
3. Eine informelle Einführung in Python.....	15
3.1. Python als Taschenrechner verwenden	15
3.1.1. Zahlen.....	15
3.1.2. Strings	18
Siehe auch	24
3.1.3. Listen.....	25
3.2. Erste Schritte Richtung Programmierung.....	27
Fußnoten.....	29
4. Mehr Kontrollfluss-Tools.....	30
4.1. if Statements	30
4.2. for Statements.....	30
4.3. Die range() Funktion	31
4.4. Break- und Continue-Anweisungen und sonst Clauses on Loops	33
4.5. pass Statements	34
4.6. Funktionen definieren	35
4.7. Mehr zum Definieren von Funktionen.....	38
4.7.1. Standardargument Werte	38
4.7.2. Keyword Argumente	40
4.7.3. Beliebige Argumentlisten.....	42
4.7.4. Entpacken von Argumentenlisten.....	43
4.7.5. Lambda Ausdrücke	43
4.7.6. Dokumentationszeichenketten.....	44
4.7.7. Funktionsbeschreibungen	45
4.8. Intermezzo: Coding Stil.....	46
Fußnoten.....	47
5. Datenstrukturen.....	48





5.1. Mehr zu Listen.....	48
5.1.1. Listen als Stacks verwenden.....	50
5.1.2. Listen als Warteschlangen verwenden	51
5.1.3. Listen-Abstraktion.....	52
5.1.4. Verschachtelte Listen-Abstraktion	54
5.2. Das del Statement.....	55
5.3. Tupel und Sequenzen.....	56
5.4. Sets	58
5.5. Dictionaries.....	59
5.6. Schleifentechniken.....	61
5.7. Mehr zu Bedingungen	63
5.8. Vergleich von Sequenzen und anderen Typen.....	64
Fußnoten	65
6. Module	66
6.1. Mehr zu Modulen	67
6.1.1. Ausführen von Modulen als Skripte.....	69
6.1.2. Der Modul-Suchpfad	70
6.1.3. “Kompilierte” Python Dateien	70
6.2. Standardmodule	71
6.3. Die dir() Funktion	72
6.4. Packages	74
6.4.1. Importieren * aus einem Paket	77
6.4.2. Referenzen innerhalb von Packages.....	78
6.4.3. Packages in mehreren Verzeichnissen.....	79
Fußnoten.....	79
7. Input und Output.....	80
7.1. Raffiniertere Ausgabeformatierung	80
7.1.1. Formatierte Strings.....	82
7.1.2. Die String format() Methode.....	83
7.1.3. Manuelle Zeichenkettenformatierung	85
7.1.4. Alte Stringformatierung	86
7.2. Lesen und Schreiben von Dateien	86
7.2.1. Methoden von Dateiobjekten	88
7.2.2. Speichern von strukturierten Daten mit json.....	90
8. Fehler und Ausnahmen	93
8.1. Syntaxfehler.....	93
8.2. Exceptions	93



8.3. Behandlung von Exceptions	94
8.4. Auslösen von Exceptions	98
8.5. Benutzerdefinierte Exceptions.....	99
8.6. Definition von Bereinigungsmaßnahmen	101
8.7. Vordefinierte Bereinigungsmaßnahmen.....	102
9. Klassen	104
9.1. Ein Wort zu Namen und Objekten.....	105
9.2. Python Scopes und Namensräume	105
9.2.1. Beispiele für Scopes und Namensräume	108
9.3. Ein erster Blick auf Klassen.....	109
9.3.1. Klassendefinitionssyntax.....	109
9.3.2. Klassenobjekte	110
9.3.3. Instanzen-Objekte.....	111
9.3.4. Methoden Objekte	112
9.3.5. Klassen- und Instanzvariablen.....	113
9.4. Weitere Bemerkungen	115
9.5. Vererbung	117
9.5.1. Mehrfachvererbung	118
9.6. Private Variablen.....	119
9.7. Chancen und Risiken.....	121
9.8. Iteratoren	122
9.9. Generatoren.....	124
9.10. Generatorausdrücke.....	125
Fußnoten.....	126
10. Kurzer Rundgang durch die Standardbibliothek	127
10.1. Betriebssystem-Schnittstelle	127
10.2. Datei-Wildcards	128
10.3. Befehlszeilenargumente	128
10.4. Fehleroutput Redirection und Programmterminierung	128
10.5. String Pattern Matching.....	129
10.6. Mathematik	129
10.7. Internetzugang	130
10.8. Daten und Zeiten	131
10.9. Datenkompression.....	131
10.10. Performance Messung.....	132



10.11. Qualitätskontrolle	132
10.12. Inkludierte Batterien	133
11. Kurzer Rundgang durch die Standardbibliothek - Teil II	135
11.1. Output Formatierung	135
11.2. Templating	136
11.3. Arbeiten mit binären Datensatzlayouts	138
11.4. Multi-threading	139
11.5. Protokollierung	140
11.6. Schwache Referenzen	141
11.7. Tools für die Arbeit mit Listen	142
11.8. Dezimalzahlenarithmetik	143
12. Virtuelle Umgebungen und Pakete	145
12.1. Einleitung	145
12.2. Virtuelle Umgebungen erschaffen	145
12.3. Verwaltung von Paketen mit pip	146
13. Was nun?	150
14. Interaktive Eingabebearbeitung und History Substitution	152
14.1. Vervollständigung der Registerkarte und Bearbeitung der Historie	152
14.2. Alternativen zum interaktiven Interpreter	152
15. Gleitkommaarithmetik: Probleme und Einschränkungen	153
15.1. Darstellungsfehler	158
16. Anhang	161
16.1. Interaktiver Modus	161
16.1.1. Fehlerbehandlung	161
16.1.2. Ausführbare Python-Skripte	161
16.1.3. Die interaktive Startdatei	162
16.1.4. Die Anpassungsmodule	163
Fußnoten	163





1. Ein kleiner Appetitanreger

Wenn du viel mit Computern arbeitest, wirst du vielleicht mit einer Aufgabe konfrontiert sein, die du automatisieren möchtest. Beispielsweise möchtest du ein Suchen und Ersetzen über eine große Anzahl von Textdateien durchführen oder eine Reihe von Fotodateien auf komplizierte Weise umbenennen und neu anordnen. Vielleicht möchtest du eine kleine benutzerdefinierte Datenbank, eine spezielle GUI-Anwendung oder ein einfaches Spiel schreiben.

Wenn du ein professioneller Softwareentwickler bist, musst du vielleicht mit mehreren C/C++/Java-Bibliotheken arbeiten, aber der übliche Zyklus aus schreiben, kompilieren, testen und erneut kompilieren ist zu langsam. Vielleicht schreibst du eine Testsuite für eine solche Bibliothek und findest das Schreiben des Testcodes eine mühsame Aufgabe. Oder vielleicht hast du ein Programm geschrieben, das eine Erweiterungssprache verwenden könnte, und du willst nicht eine ganz neue Sprache für deine Anwendung entwerfen und implementieren.

Dann ist Python genau die richtige Sprache für dich.

Du könntest ein Unix-Shell-Skript oder Windows-Batchdateien für einige dieser Aufgaben schreiben, aber Shell-Skripte sind nur gut dafür geeignet, Dateien zu bewegen und Textdaten zu ändern, aber nicht für GUI-Anwendungen oder Spiele. Du könntest ein C/C++/Java-Programm schreiben, aber es kann viel Entwicklungszeit in Anspruch nehmen, um überhaupt ein First-Draft-Programm zu bekommen. Python ist einfacher zu bedienen, verfügbar unter Windows, Mac OS X und Unix-Betriebssystemen und hilft dir, die Arbeit schneller zu erledigen. Python ist einfach zu bedienen, aber es ist eine echte Programmiersprache, die viel mehr Struktur und Unterstützung für große Programme bietet, als Shellskripte oder Batchdateien bieten können. Auf der anderen Seite bietet Python auch viel mehr Fehlerprüfung als C, und da es sich um eine sehr anspruchsvolle Sprache handelt, verfügt es über eingebaute High-Level-Datentypen, wie flexible Arrays und Dictionaries. Aufgrund seiner allgemeineren Datentypen ist Python auf eine viel breiter gefächerte Probleme als Awk oder gar Perl anwendbar, aber viele Dinge sind in Python mindestens so einfach wie in diesen Sprachen.

Mit Python kannst du dein Programm in Module aufteilen, die in anderen Python-Programmen wiederverwendet werden können. Es enthält eine große Sammlung von Standardmodulen, die du als Grundlage für deine Programme verwenden kannst - oder als Beispiele, um das Programmieren in Python zu lernen. Einige dieser Module bieten Dinge wie Datei-I/O, Systemaufrufe, Sockets und sogar Schnittstellen zu Toolkits für grafische Benutzeroberflächen wie Tk.



DATAMICS
machine intelligence consulting services



Python ist eine interpretierte Sprache, die dir bei der Programmentwicklung viel Zeit sparen kann, da keine Kompilierung und Verknüpfung notwendig sind. Der Interpreter kann interaktiv eingesetzt werden, was es einfach macht, mit Funktionen der Sprache zu experimentieren, Wegwerfprogramme zu schreiben oder Funktionen während der Bottom-up-Programmentwicklung zu testen. Es ist auch ein praktischer Taschenrechner.

Python ermöglicht es, Programme kompakt und lesbar zu schreiben. Programme, die in Python geschrieben wurden, sind in der Regel viel kürzer als gleichwertige C-, C++- oder Java-Programme, und zwar aus mehreren Gründen:

- Die High-Level-Datentypen ermöglichen es dir, komplexe Operationen in einer einzigen Anweisung auszudrücken;
- Anweisungsgruppierung erfolgt durch Einrückung anstelle von beginnenden und endenden Klammern;
- Es sind keine Variablen- oder Argumentdeklarationen erforderlich.

Python ist erweiterbar: Wenn du weißt, wie man in C programmiert, ist es einfach, dem Interpreter eine neue eingebaute Funktion oder ein neues Modul hinzuzufügen, entweder um kritische Operationen mit maximaler Geschwindigkeit durchzuführen oder um Python-Programme mit Bibliotheken zu verknüpfen, die möglicherweise nur in binärer Form verfügbar sind (z.B. eine herstellerspezifische Grafikbibliothek). Wenn du wirklich von C abhängig bist, kannst du den Python-Interpreter in eine in C geschriebene Anwendung einbinden und ihn als Erweiterung oder Befehlssprache für diese Anwendung verwenden.

Die Sprache ist übrigens nach der BBC-Show "Monty Python's Flying Circus" benannt und hat nichts mit Reptilien zu tun. In der Dokumentation ist es nicht nur erlaubt, Monty Python-Sketches zu referenzieren, es wird sogar empfohlen!

Jetzt, da du von Python begeistert bist, solltest du es genauer durchgehen. Da der beste Weg, eine Sprache zu lernen, darin besteht, sie zu verwenden, lädt das Tutorial dazu ein, mit dem Python-Interpreter zu spielen, während du liest.

Im nächsten Kapitel wird die Mechanik der Verwendung des Interpreters erläutert. Dies sind eher alltägliche Informationen, die aber für das Ausprobieren der später gezeigten Beispiele unerlässlich sind.

Der Rest des Tutorials stellt verschiedene Funktionen der Python-Sprache und des -Systems anhand von Beispielen vor, beginnend mit einfachen Ausdrücken, Anweisungen und





Datentypen, über Funktionen und Module bis hin zu erweiterten Konzepten wie Ausnahmen und benutzerdefinierten Klassen.





2. Nutzung des Python Interpreters

2.1. Aufruf des Interpreters

Der Python-Interpreter wird normalerweise als `/usr/local/bin/python3.7` auf den Rechnern installiert, auf denen er verfügbar ist; das Einfügen von `/usr/local/bin` in den Suchpfad deiner Unix-Shell macht es möglich, ihn durch Eingabe des Befehls zu starten:

```
python3.7
```

Da die Wahl des Verzeichnisses, in dem der Interpreter liegt, eine Installationsoption ist, sind andere Orte möglich; frag deinen lokalen Python-Guru oder Systemadministrator. (z.B. ist `/usr/local/python` ein beliebter alternativer Ort.)

Auf Windows-Rechnern befindet sich die Python-Installation normalerweise in `C:\Python37`, obwohl du dies ändern kannst, wenn du das Installationsprogramm ausführst. Um dieses Verzeichnis zu deinem Pfad hinzuzufügen, kannst du den folgenden Befehl in die Befehlszeile in einer DOS-Box eingeben:

```
set path=%path%;C:\python37
```

Die Eingabe eines Zeichens am Ende der Datei (Strg-D unter Unix, Strg-Z unter Windows) an der primären Eingabeaufforderung bewirkt, dass der Interpreter ohne Exit-Status beendet wird. Wenn das nicht funktioniert, kannst du den Interpreter verlassen, indem du den folgenden Befehl eingibst: `quit()`.

Zu den Funktionen zur Zeilenbearbeitung des Interpreters gehören interaktives Editieren, Ersetzen der Historie und Vervollständigung des Codes auf Systemen, die Readline unterstützen. Möglicherweise ist die schnellste Prüfung, um zu sehen, ob die Befehlszeilenbearbeitung unterstützt wird, die Eingabe von Control-P in die erste Python-Eingabeaufforderung, die du erhältst. Wenn es piept, hast du eine Befehlszeilenbearbeitung; siehe Anhang Interaktive Eingabebearbeitung und Historienersetzung für eine Einführung in die Tasten. Wenn nichts zu passieren scheint oder wenn `^P` zurückgeworfen wird, ist die Bearbeitung der Befehlszeile nicht möglich; du kannst nur die Rücktaste verwenden, um Zeichen aus der aktuellen Zeile zu entfernen.



Der Interpreter arbeitet ähnlich wie die Unix-Shell: Beim Aufruf mit Standardeingabe, die an ein tty-Gerät angeschlossen ist, liest und führt er interaktiv Befehle aus; beim Aufruf mit einem Dateinamenargument oder mit einer Datei als Standardeingabe liest und führt er ein Skript aus dieser Datei aus.

Eine zweite Möglichkeit, den Interpreter zu starten, ist der Befehl `python -c[arg].....`, der die Anweisungen im Befehl ausführt, analog zur Option `-c` der Shell. Da Python-Anweisungen oft Leerzeichen oder andere Zeichen enthalten, die spezifisch für die Shell sind, wird in der Regel empfohlen, den Befehl in seiner Gesamtheit mit einfachen Anführungszeichen anzugeben.

Einige Python-Module sind auch als Skripte nützlich. Diese können mit `python -m module[arg]....` aufgerufen werden, was die Quelldatei für das Modul so ausführt, als hättest du ihren vollständigen Namen auf der Kommandozeile angegeben.

Wenn eine Skriptdatei verwendet wird, ist es manchmal sinnvoll, das Skript auszuführen und anschließend in den interaktiven Modus zu wechseln. Dies kann durch die Übergabe von `-i` vor dem Skript erreicht werden.

Alle Befehlszeilenoptionen sind in Befehlszeile und Umgebung beschrieben.


2.1.1. Argumentübergabe

Wenn dem Interpreter bekannt, werden der Skriptname und weitere Argumente danach in eine Liste von Zeichenketten umgewandelt und der Variablen `argv` im `sys`-Modul zugewiesen. Du kannst auf diese Liste zugreifen, indem du `Import sys` ausführst. Die Länge der Liste ist mindestens eins; wenn kein Skript und keine Argumente angegeben werden, ist `sys.argv[0]` eine leere Zeichenkette. Wenn der Skriptname als `'-'` (d.h. Standardeingabe) angegeben wird, wird `sys.argv[0]` auf `'-'` gesetzt. Wenn der Befehl `-ccommand` verwendet wird, wird `sys.argv[0]` auf `'-c'` gesetzt. Wenn das `-m`-Modul verwendet wird, wird `sys.argv[0]` auf den vollständigen Namen des gefundenen Moduls gesetzt. Optionen, die nach dem Befehl `-c` oder dem Modul `-m` gefunden wurden, werden von der Optionsverarbeitung des Python-Interpreters nicht verbraucht, sondern in `sys.argv` für den Befehl oder das Modul belassen.

2.1.2. Interaktiver Modus


Wenn Befehle von einem tty gelesen werden, befindet sich der Interpreter im interaktiven Modus. In diesem Modus wird nach dem nächsten Befehl mit dem primären Prompt gefragt, normalerweise mit drei Größer-Zeichen (`>>>`); für Fortsetzungszeilen wird mit dem sekundären





Prompt gefragt, standardmäßig drei Punkte (...). Der Dolmetscher gibt eine Begrüßungsansage aus mit Angabe der Versionsnummer und eines Copyright-Vermerks, bevor er die erste Eingabeaufforderung ausgibt:

```
$ python3.7
Python 3.7 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



Fortsetzungszeilen werden benötigt, wenn du ein mehrzeiliges Konstrukt eingibst. Sieh dir als Beispiel dieses if-Statement an:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

2.2. Der Interpreter und seine Umgebung

2.2.1. Quellcode-Kodierung


Standardmäßig werden Python-Quelldateien wie in UTF-8 kodiert behandelt. In dieser Kodierung können Zeichen der meisten Sprachen der Welt gleichzeitig in Zeichenkettenliteralen, Bezeichnern und Kommentaren verwendet werden - obwohl die Standardbibliothek nur ASCII-Zeichen für Bezeichner verwendet, eine Konvention, der jeder portable Code folgen sollte. Um alle diese Zeichen richtig anzuzeigen, muss dein Editor erkennen, dass es sich bei der Datei um UTF-8 handelt, und er muss eine Schriftart verwenden, die alle Zeichen in der Datei unterstützt. Um eine andere Kodierung als die Standardkodierung zu deklarieren, sollte eine spezielle Kommentarzeile als erste Zeile der Datei hinzugefügt werden. Die Syntax ist wie folgt:

```
# -*- coding: encoding -*-
```

wobei die Kodierung einer der gültigen Codecs ist, die von Python unterstützt werden. Um beispielsweise zu erklären, dass die Windows-1252-Kodierung verwendet werden soll, sollte die erste Zeile deiner Quellcode-Datei sein:

```
# -*- coding: cp1252 -*-
```





Eine Ausnahme von der Regel der ersten Zeile ist, wenn der Quellcode mit einer UNIX-Zeile "shebang" beginnt. In diesem Fall sollte die Kodierungsdeklaration als zweite Zeile der Datei hinzugefügt werden. Zum Beispiel:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Fußnoten

- [1] Unter Unix ist der Python 3.x-Interpreter standardmäßig nicht mit der ausführbaren Datei python installiert, so dass er nicht mit einer gleichzeitig installierten ausführbaren Python 2.x-Datei in Konflikt gerät.



3. Eine informelle Einführung in Python

In den folgenden Beispielen unterscheiden sich Input und Output durch das Vorhandensein oder Fehlen von Prompts (>>> und): Um das Beispiel zu wiederholen, musst du alles nach dem Prompt eingeben, wenn der Prompt erscheint; Zeilen, die nicht mit einem Prompt beginnen, werden vom Interpreter ausgegeben. Bitte bedenke, dass eine sekundäre Eingabeaufforderung in einer Zeile selbst in einem Beispiel bedeutet, dass du eine Leerzeile eingeben musst; dies wird verwendet, um einen mehrzeiligen Befehl zu beenden.

Viele der Beispiele in diesem Handbuch, auch die, die bei der interaktiven Eingabeaufforderung eingegeben wurden, enthalten Kommentare. Kommentare in Python beginnen mit dem Hash-Zeichen # und reichen bis zum Ende der physikalischen Zeile. Ein Kommentar kann am Anfang einer Zeile oder hinter Leerzeichen oder Code stehen, aber nicht innerhalb eines Strings. Ein Hash-Zeichen innerhalb eines Strings ist nur ein Hash-Zeichen. Da Kommentare zur Erklärung von Code dienen und nicht von Python interpretiert werden, können sie bei der Eingabe von Beispielen weggelassen werden.

Einige Beispiele:

```
# dies ist der erste Kommentar
spam = 1 # und dies ist der zweite Kommentar
        # ... und ein dritter!
text = "# Dies ist kein Kommentar, da er in Anführungszeichen steht."
```

3.1. Python als Taschenrechner verwenden

Testen wir einige einfache Python-Befehle. Starte den Interpreter und warte auf die primäre Eingabeaufforderung, >>>. (Es sollte nicht lange dauern.)

3.1.1. Zahlen

Der Interpreter fungiert als einfacher Taschenrechner: Du kannst einen Ausdruck darin eingeben und er schreibt den Wert. Die Syntax der Ausdrücke ist einfach: Die Operatoren +, -, * und / funktionieren wie in den meisten anderen Sprachen (z.B. Pascal oder C); Klammern (()) können zur Gruppierung verwendet werden. Zum Beispiel:


```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # Division gibt immer eine Gleitkommazahl zurück
1.6
```

Die Ganzzahlen (z.B. 2, 4, 20) haben den Typ `int`, die mit Dezimalstellen (z.B. 5.0, 1.6) den Typ `float`. Wir werden später im Tutorial mehr über numerische Typen erfahren.

Division (`/`) gibt immer einen `Float` zurück. Um abzurunden und ein ganzzahliges Ergebnis zu erhalten (wobei der Bruchrest des Ergebnis verworfen wird), kannst du den `//`-Operator verwenden; um den Rest (modulo) zu berechnen, kannst du `%` verwenden:


```
>>> 17 / 3 # klassische Division gibt ein Float zurück
5.666666666666667
>>>
>>> 17 // 3 # Abrunden verwirft den Bruchrest
5
>>> 17 % 3 # der % Operator gibt den Rest der Division zurück
2
>>> 5 * 3 + 2 # Ergebnis * Divisor + Rest
17
```

Mit Python ist es möglich, mit dem Operator `**` Potenzen zu berechnen[1]:

```
>>> 5 ** 2 # 5 zum Quadrat
25
>>> 2 ** 7 # 2 hoch 7
128
```

Das Gleichheitszeichen (`=`) wird verwendet, um einer Variablen einen Wert zuzuweisen. Danach wird vor der nächsten interaktiven Eingabeaufforderung kein Ergebnis angezeigt:





```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Wenn eine Variable nicht "definiert" (mit einem Wert versehen) ist, wird der Versuch, sie zu verwenden, zu einem Fehler führen:

```
>>> n # versuchter Zugriff auf undefinierte Variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```


Es gibt volle Unterstützung für Gleitkommazahlen; Operatoren mit gemischten Operanden konvertieren den ganzzahligen Operanden in Gleitkommazahlen:

```
>>> 4 * 3.75 - 1
14.0
```

Im interaktiven Modus wird der letzte gedruckte Ausdruck der Variablen `_` zugeordnet. Das bedeutet, dass es bei der Verwendung von Python als Taschenrechner etwas einfacher ist, z.B. Berechnungen fortzusetzen:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```





Diese Variable sollte vom Benutzer als schreibgeschützt behandelt werden. Weise ihm nicht explizit einen Wert zu - Du würdest eine unabhängige lokale Variable mit dem gleichen Namen erstellen, die die eingebaute Variable mit ihrem magischen Verhalten maskiert.

Zusätzlich zu int und float unterstützt Python auch andere Arten von Zahlen, wie Dezimalzahl (Decimal) und Bruchzahl (Fraction). Python hat auch eine eingebaute Unterstützung für komplexe Zahlen und verwendet das j- oder J-Suffix, um den Imaginärteil anzuzeigen (z.B. $3+5j$).

3.1.2. Strings

Neben Zahlen kann Python auch Zeichenketten manipulieren, die auf verschiedene Weise ausgedrückt werden können. Die Daten lassen sich in einfache Anführungszeichen ('...') oder doppelte Anführungszeichen ("...") mit dem gleichen Ergebnis umschließen [2]. \ kann verwendet werden, um anzuzeigen, dass mit dem einfachen Anführungszeichen in diesem Fall nicht Anfang oder Ende eines Zitats gemeint sind:

```
>>> 'spam eggs' # einfache Anführungszeichen
'spam eggs'
>>> 'doesn\t' # verwende \ um das einfache Anführungszeichen zu escapen...
'doesn\t'
>>> "doesn't" # ...oder verwende stattdessen doppelte Anführungszeichen
'doesn't'
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\t," they said.'
'"Isn\t," they said.'
```

Im interaktiven Interpreter wird die Ausgabezeichenkette in Anführungszeichen eingeschlossen und Sonderzeichen mit Backslashes ausgelassen. Während dies manchmal anders aussieht als die Eingabe (die umschließenden Anführungszeichen könnten sich ändern), sind die beiden Zeichenketten gleichwertig. Die Zeichenkette wird in doppelte Anführungszeichen eingeschlossen, wenn die Zeichenkette ein einfaches Anführungszeichen und keine doppelten Anführungszeichen enthält, ansonsten wird sie in einfache Anführungszeichen eingeschlossen.



Die Funktion `print()` erzeugt eine besser lesbare Ausgabe, indem sie die umschließenden Anführungszeichen weglässt und Escape- und Sonderzeichen ausgibt:

```
>>> '"Isn\'t," they said.'
'Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
Isn't," they said.
>>> s = 'First line.\nSecond line.' # \n bedeutet Zeilenumbruch
>>> s # ohne print() wird \n in den Output integriert
'First line.\nSecond line.'
>>> print(s) # mit print() erstellt \n eine neue Zeile
First line.
Second line.
```


Wenn du nicht willst, dass Zeichen, die mit `\` eingeleitet werden, als Sonderzeichen interpretiert werden, kannst du Rohzeichenketten verwenden, indem du ein `r` vor dem ersten Anführungszeichen hinzufügst:

```
>>> print('C:\some\name') # hier heißt \n Zeilenumbruch!
C:\some
ame
>>> print(r'C:\some\name') # Beachte das r vor den Anführungszeichen
C:\some\name
```

Zeichenkettenliterals können sich über mehrere Zeilen erstrecken. Eine Möglichkeit ist die Verwendung von Triple-Quotes: `"""..."""` oder `'''...'''`. Zeilenenden werden automatisch in die Zeichenkette aufgenommen, aber es ist möglich, dies zu verhindern, indem man ein `\` am Ende der Zeile anhängt. Das folgende Beispiel:

```
print("""\
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
""")
```





Erzeugt die folgende Ausgabe (beachte, dass der anfängliche Zeilenumbruch nicht berücksichtigt wird):

```
Usage: thingy [OPTIONS]
  -h                Display this usage message
  -H hostname       Hostname to connect to
```

Zeichenketten können mit dem Operator + verknüpft (konkateniert) und mit * wiederholt werden:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Zwei oder mehr Zeichenkettenlitterale (d.h. die zwischen Anführungszeichen eingeschlossenen) nebeneinander werden automatisch verknüpft.

```
>>> 'Py' 'thon'
'Python'
```

Diese Funktion ist besonders nützlich, wenn du lange Zeichenketten aufteilen möchtest:

```
>>> text = ('Put several strings within parentheses '
...        'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```



Dies funktioniert jedoch nur mit zwei Strings, nicht mit Variablen oder Ausdrücken:

```
>>> prefix = 'Py'
>>> prefix 'thon' # Konkatination einer Variable und eines Strings nicht möglich
File "<stdin>", line 1
    prefix 'thon'
          ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
File "<stdin>", line 1
    ('un' * 3) 'ium'
          ^
SyntaxError: invalid syntax
```


Wenn du Variablen oder eine Variable und einen String verketteten möchtest, verwende +:

```
>>> prefix + 'thon'
'Python'
```

Zeichenketten können indiziert (indexiert) werden, wobei das erste Zeichen den Index 0 hat. Es gibt keinen separaten Zeichentyp; ein Zeichen ist einfach eine Zeichenkette der Größe eins:

```
>>> word = 'Python'
>>> word[0] # Zeichen an Position 0
'P'
>>> word[5] # Zeichen an Position 5
'n'
```

Indizes können auch negative Zahlen sein, um von rechts zu zählen:



```
>>> word[-1] # Letztes Zeichen
'n'
>>> word[-2] # vorletztes Zeichen
'o'
>>> word[-6]
'p'
```

Man beachte, dass -0 gleich 0 ist, so dass negative Indizes bei -1 beginnen. Neben der Indizierung wird auch das Slicing unterstützt. Während die Indizierung verwendet wird, um einzelne Zeichen zu erhalten, ermöglicht das Slicing das Erhalten von Teilzeichenketten:


```
>>> word[0:2] # Zeichen von Position 0 (inklusive) bis 2 (exklusive)
'Py'
>>> word[2:5] # Zeichen von Position 2 (inklusive) bis 5 (exklusive)
'tho'
```

Achte darauf, dass der Anfang immer enthalten ist und das Ende immer ausgeschlossen ist. Dadurch wird sichergestellt, dass `s[:i] + s[i:]` immer gleich `s` ist:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Slice-Indizes haben nützliche Standardwerte; ein weggelassener erster Index ist auf Null voreingestellt, ein weggelassener zweiter Index auf die Größe der zu schneidenden Zeichenkette.

```
>>> word[:2] # Zeichen vom Anfang bis Position 2 (exklusive)
'Py'
>>> word[4:] # Zeichen von Position 4 (inklusive) bis zum Ende
'on'
>>> word[-2:] # Zeichen vom vorletzten (inklusive) bis zum Ende
'on'
```

Eine Möglichkeit, sich daran zu erinnern, wie Slices funktionieren, besteht darin, sich die Indizes als Zeiger zwischen Zeichen vorzustellen, wobei der linke Rand des ersten Zeichens mit der Nummer 0 versehen ist, und dann hat der rechte Rand des letzten Zeichens einer Reihe von n Zeichen beispielsweise den Index n:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Die erste Zahlenreihe gibt die Position der Indizes 0...6 in der Zeichenkette an, die zweite Reihe die entsprechenden negativen Indizes. Die Scheibe von i bis j besteht aus allen Zeichen zwischen den mit i bzw. j gekennzeichneten Kanten.

Bei nicht-negativen Indizes ist die Länge eines Slice die Differenz der Indizes, wenn beide innerhalb der Grenzen liegen. Zum Beispiel ist die Länge des Wortes [1:3] 2.

Der Versuch, einen zu großen Index zu verwenden, führt zu einem Fehler:

```
>>> word[42]  # das Wort hat nur 6 Buchstaben
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Außerhalb des Bereichs liegende Slice-Indizes werden jedoch bei der Verwendung zum Slicen elegant behandelt:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python-Strings können nicht geändert werden - sie sind unveränderlich. Daher führt die Zuordnung zu einer indizierten Position in der Zeichenkette zu einem Fehler:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Wenn du einen anderen String brauchst, solltest du einen neuen erstellen:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

Die eingebaute Funktion `len()` gibt die Länge eines Strings zurück:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Siehe auch

Textsequenz Typ — str

Zeichenketten sind Beispiele für Sequenzarten und unterstützen die von solchen Typen unterstützten gemeinsamen Operationen.

String Methoden

Zeichenketten unterstützen eine große Anzahl von Methoden für grundlegende Transformationen und Suchen

Formatierte Stringlitterale

Zeichenkettenlitterale, die eingebettete Ausdrücke enthalten

Format Stringsyntax



Informationen über die Zeichenkettenformatierung mit `str.format()`.

printf-style String Formatierung

Die alten Formatierungsoperationen, die aufgerufen werden, wenn Zeichenketten der linke Operand des %-Operators sind, werden hier näher beschrieben

3.1.3. Listen

Python kennt eine Reihe von zusammengesetzten Datentypen, mit denen andere Werte zusammengefasst werden können. Die vielseitigste ist die Liste, die als Liste von kommasetrennten Werten (Elementen) zwischen eckigen Klammern geschrieben werden kann. Listen können Elemente verschiedener Typen enthalten, aber normalerweise haben die Elemente alle den gleichen Typ.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```


Wie Zeichenketten (und alle anderen eingebauten Sequenzarten) können Listen indiziert und gesliced werden:

```
>>> squares[0] # indexing gibt das Element zurück
1
>>> squares[-1]
25
>>> squares[-3:] # slicing gibt eine neue Liste zurück
[9, 16, 25]
```

Alle Slice-Operationen liefern eine neue Liste mit den angeforderten Elementen. Das bedeutet, dass folgendes Slicing eine neue (flache) Kopie der Liste zurückgibt:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Listen unterstützen auch Operationen wie Verkettung:



```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Im Gegensatz zu Zeichenketten, die unveränderlich sind, sind Listen ein veränderlicher Typ, d.h. es ist möglich, ihren Inhalt zu ändern:


```
>>> cubes = [1, 8, 27, 65, 125] # hier ist etwas falsch!
>>> 4 ** 3 # die Kubikzahl von 4 ist 64, nicht 65!
64
>>> cubes[3] = 64 # ersetze den falschn Wert
>>> cubes
[1, 8, 27, 64, 125]
```

Du kannst auch neue Elemente am Ende der Liste hinzufügen, indem du die Methode `append()` verwendest (wir werden später mehr über Methoden erfahren):

```
>>> cubes.append(216) # addiere die Kubikzahl von 6
>>> cubes.append(7 ** 3) # und die Kubikzahl von 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Die Zuordnung zu Slices ist ebenfalls möglich, was sogar die Größe der Liste verändern oder ganz löschen kann:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
```



```
['a', 'b', 'f', 'g']
>>> # Lösche die Liste, indem du alle Elemente durch eine leere Liste ersetzt
>>> letters[:] = []
>>> letters
[]
```

Die eingebaute Funktion `len()` gilt auch für Listen:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Es ist z.B. möglich, Listen zu verschachteln (Listen mit anderen Listen zu erstellen):

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. Erste Schritte Richtung Programmierung

Natürlich können wir Python für komplexere Aufgaben verwenden, als zwei und zwei zusammen zu addieren. Zum Beispiel können wir eine erste Untersequenz der Fibonacci-Serie wie folgt schreiben:

```
>>> # Fibonaccireihe:
... # die Summe zweier Elemente definieren das nächste
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
... 
```






```
0
1
1
2
3
5
8
```

In diesem Beispiel werden einige neue Funktionen vorgestellt.

- Die erste Zeile enthält eine Mehrfachzuordnung: Die Variablen a und b erhalten gleichzeitig die neuen Werte 0 und 1. In der letzten Zeile werden diese wiederverwendet, was zeigt, dass die Ausdrücke auf der rechten Seite alle zuerst ausgewertet werden, bevor eine der Zuweisungen erfolgt. Die rechten Ausdrücke werden von links nach rechts ausgewertet.
- Die while-Schleife wird ausgeführt, solange die Bedingung (hier: $a < 10$) erfüllt bleibt. In Python, wie auch in C, ist jeder ganzzahlige Wert ungleich Null wahr, Null ist falsch. Die Bedingung kann auch ein Zeichenketten- oder Listenwert sein, in der Tat eine beliebige Sequenz; alles mit einer Länge ungleich Null ist wahr, leere Sequenzen sind falsch. Der im Beispiel verwendete Test ist ein einfacher Vergleich. Die Standardvergleichsoperatoren werden genauso geschrieben wie in C: $<$ (weniger als), $>$ (mehr als), $==$ (gleich), $<=$ (weniger als oder gleich), $>=$ (mehr als oder gleich) und $!=$ (nicht gleich).
- Der Körper der Schleife ist eingerückt: Einrückung ist Python's Art, Anweisungen zu gruppieren. Bei der interaktiven Eingabeaufforderung musst du für jede eingerückte Zeile ein Tab oder ein Leerzeichen eingeben. In der Praxis wirst du kompliziertere Eingaben für Python mit einem Texteditor vorbereiten; alle anständigen Texteditoren haben eine automatische Einrückung. Wenn eine zusammengesetzte Anweisung interaktiv eingegeben wird, muss ihr eine Leerzeile folgen, um den Abschluss anzuzeigen (da der Parser nicht erraten kann, wann Sie die letzte Zeile eingegeben haben). Es ist zu beachten, dass jede Zeile innerhalb einer Grundkonstruktion um die gleiche Anzahl Leerstellen eingerückt sein muss.
- Die Funktion `print()` schreibt den Wert des/der Argumente, die ihr übergeben werden. Es unterscheidet sich vom Schreiben des zu schreibenden Ausdrucks (wie wir es bereits in den Beispielen des Taschenrechners getan haben) durch die Art und Weise,



wie es mehrere Argumente, Fließkommazahlen und Zeichenketten behandelt. Zeichenketten werden ohne Anführungszeichen ausgegeben und es wird ein Leerzeichen zwischen den Elementen eingefügt, so dass du die Dinge gut formatieren kannst, wie hier:

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

Das Schlüsselwort `argument end` kann verwendet werden, um den Zeilenumbruch nach der Ausgabe zu vermeiden oder die Ausgabe mit einer anderen Zeichenkette zu beenden:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=', ')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Fußnoten

[1] Da `**` höhere Priorität hat als `-`, wird `-3**2` als `-(3**2)` interpretiert und ergibt somit `-9`, um dies zu vermeiden und `9` zu erhalten, kannst du `(-3)**2` verwenden.

[2] Im Gegensatz zu anderen Sprachen haben Sonderzeichen wie `\n` die gleiche Bedeutung, sowohl bei einfachen ('...') als auch bei doppelten ("...") Anführungszeichen. Der einzige Unterschied zwischen den beiden besteht darin, dass man doppelte Anführungszeichen nicht escapen muss (aber man muss `\` escapen) und umgekehrt.



4. Mehr Kontrollfluss-Tools

Neben der gerade eingeführten while-Anweisung kennt Python die üblichen Kontrollflussanweisungen, die aus anderen Sprachen bekannt sind, mit einigen Wendungen.

4.1. if Statements


Der vielleicht bekannteste Anweisungstyp ist die if-Anweisung. Zum Beispiel:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Es kann null oder mehr elif Teile geben, und der else Teil ist optional. Das Schlüsselwort "elif" steht für "else if" und ist nützlich, um übermäßige Einrückungen zu vermeiden. Eine if elif elif ... elif Sequenz ist ein Ersatz für die switch- oder case-Anweisungen, die in anderen Sprachen zu finden sind.

4.2. for Statements

Das for-Statement in Python unterscheidet sich ein wenig von dem, was man in C oder Pascal gewohnt ist. Anstatt immer über eine arithmetische Folge von Zahlen zu iterieren (wie in Pascal), oder dem Benutzer die Möglichkeit zu geben, sowohl den Iterationsschritt als auch die Haltebedingung zu definieren (wie in C), iteriert Python's for-Anweisung über die Elemente einer beliebigen Sequenz (eine Liste oder eine Zeichenkette), in der Reihenfolge, in der sie in der Sequenz erscheinen. Zum Beispiel (kein Wortspiel vorgesehen):



```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Wenn du die Reihenfolge ändern musst, in der du innerhalb der Schleife wiederholst (z.B. um ausgewählte Elemente zu duplizieren), wird empfohlen, zuerst eine Kopie zu erstellen. Das Iterieren über eine Sequenz macht nicht implizit eine Kopie. Die Slice-Notation macht dies besonders komfortabel:


```
>>> for w in words[:]: # Schleife über eine Slice Kopie der gesamten Liste
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

Mit `for w in words:` würde das Beispiel versuchen, eine unendliche Liste zu erstellen, indem es immer wieder `defenestrate` einfügt.

4.3. Die `range()` Funktion

Wenn du über eine Folge von Zahlen iterieren musst, ist die eingebaute Funktion `range()` nützlich. Es erzeugt arithmetische Verläufe:

```
>>> for i in range(5):
...     print(i)
0
1
2
3
4
```



Der angegebene Endpunkt ist nie Teil der erzeugten Sequenz; `range(10)` erzeugt 10 Werte, die gesetzlichen Indizes für Elemente einer Sequenz der Länge 10. Es ist möglich, den Bereich bei einer anderen Zahl beginnen zu lassen oder eine andere Schrittweite anzugeben (auch negativ; manchmal wird dies als "Schritt" bezeichnet):

```
range(5, 10)
    5, 6, 7, 8, 9

range(0, 10, 3)
    0, 3, 6, 9

range(-10, -100, -30)
    -10, -40, -70
```


Um über die Indizes einer Sequenz zu iterieren, kannst du `range()` und `len()` wie folgt kombinieren:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In den meisten dieser Fälle ist es jedoch sinnvoll, die Funktion `enumerate()` zu verwenden, siehe Schleifentechniken.

Eine seltsame Sache passiert, wenn du nur einen Bereich druckst:

```
>>> print(range(10))
range(0, 10)
```



In vielerlei Hinsicht verhält sich das von `range()` zurückgegebene Objekt so, als wäre es eine Liste, aber in Wirklichkeit ist es das nicht. Es ist ein Objekt, das die aufeinanderfolgenden Elemente der gewünschten Sequenz zurückgibt, wenn du darüber iterierst, aber es macht die Liste nicht real, was Platz spart.

Wir sagen, dass ein solches Objekt iterierbar ist, d.h. als Ziel für Funktionen und Konstrukte geeignet ist, die etwas erwarten, von dem sie aufeinanderfolgende Elemente erhalten können, bis die Versorgung erschöpft ist. Wir haben gesehen, dass die `for`-Anweisung ein solcher Iterator ist. Die Funktion `list()` ist eine andere; sie erstellt Listen aus Iterables:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```


Später werden wir weitere Funktionen sehen, die iterables zurückgeben und iterables als Argument nehmen.

4.4. Break- und Continue-Anweisungen und sonst Clauses on Loops

Die `Break`-Anweisung bricht, wie in C, die innerste `for`- oder `while`- Schleife ab.

Schleifenanweisungen können eine `else` Klausel haben; sie wird ausgeführt, wenn die Schleife durch Erschöpfung der Liste beendet wird (mit `for`) oder wenn die Bedingung falsch wird (mit `while`), aber nicht, wenn die Schleife durch eine `Break`-Anweisung beendet wird. Dies wird durch die folgende Schleife veranschaulicht, die nach Primzahlen sucht:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # Schleife life durch ohne einen Faktor zu finden
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
```



```
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Ja, das ist der richtige Code. Schau genau hin: Die *else*-Anweisung gehört zur *for*-Schleife, nicht zur *if*-Anweisung.)

Bei Verwendung mit einer Schleife hat die *else*-Anweisung mehr Gemeinsamkeiten mit der *else*-Anweisung einer *try*-Anweisung als mit der von *if*-Anweisungen: Die *else*-Anweisung einer *try*-Anweisung läuft, wenn keine Ausnahme auftritt, und die *else*-Anweisung einer Schleife läuft, wenn kein *Break* auftritt. Weitere Informationen über die *Try*-Anweisung und Ausnahmen findest du unter Behandlung von Ausnahmen.


Die *Continue*-Anweisung, die ebenfalls von C übernommen wurde, setzt mit der nächsten Iteration der Schleife fort:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5. pass Statements

Das *Pass*-Statement bewirkt nichts. Es kann verwendet werden, wenn eine Anweisung syntaktisch benötigt wird, das Programm aber keine Aktion erfordert. Zum Beispiel:

```
>>> while True:
...     pass # Warten bis Abbruch durch Tastatur (Ctrl+C)
...
```



Dies wird häufig verwendet, um minimale Klassen zu erstellen:

```
>>> class MyEmptyClass:
...     pass
...
```

Eine weitere Verwendung von pass ist als Platzhalter für eine Funktion oder einen bedingten Körper, wenn du an neuem Code arbeitest, so dass du weiterhin auf einer abstrakteren Ebene denken kannst. Das Pass wird stillschweigend ignoriert:

```
>>> def initlog(*args):
...     pass    # Denk dran, dies zu implementieren!
...
```

4.6. Funktionen definieren

Wir können eine Funktion erstellen, die die Fibonacci-Serie bis zu einer beliebigen Grenze ausgibt:

```
>>> def fib(n):    # schreibt Fibonaccireihe bis n
...     """Gib eine Fibonaccireihe bis n aus."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Das Schlüsselwort def führt eine Funktionsdefinition ein. Darauf folgen der Funktionsname und die in Klammern gesetzte Liste der Formalparameter. Die Anweisungen, die den Körper der Funktion bilden, beginnen in der nächsten Zeile und müssen eingerückt werden.

Die erste Anweisung des Funktionskörpers kann optional ein Zeichenkettenliteral sein; dieses Zeichenkettenliteral ist die Dokumentationszeichenkette oder docstring der Funktion. (Mehr über





Docstrings findest du im Abschnitt Documentation Strings.) Es gibt Tools, die Docstrings verwenden, um automatisch Online- oder gedruckte Dokumentation zu erstellen oder den Benutzer interaktiv durch Code blättern zu lassen; es ist eine gute Angewohnheit, Docstrings in den Code aufzunehmen, den du schreibst, also mach es dir zur Gewohnheit.

Die Ausführung einer Funktion führt eine neue Symboltabelle ein, die für die lokalen Variablen der Funktion verwendet wird. Genauer gesagt, speichern alle Variablenzuweisungen in einer Funktion den Wert in der lokalen Symboltabelle; während Variablenreferenzen zuerst in der lokalen Symboltabelle, dann in den lokalen Symboltabellen der umschließenden Funktionen, dann in der globalen Symboltabelle und schließlich in der Tabelle der eingebauten Namen suchen. Daher können globale Variablen nicht direkt einem Wert innerhalb einer Funktion zugewiesen werden (es sei denn, sie werden in einer globalen Anweisung benannt), obwohl sie referenziert werden können.


Die Aktualparameter (Argumente) eines Funktionsaufrufs werden beim Aufruf in die lokale Symboltabelle der aufgerufenen Funktion eingetragen; somit werden Argumente mit call by value übergeben (wobei der Wert immer eine Objektreferenz und nicht der Wert des Objekts ist). [1] Wenn eine Funktion eine andere Funktion aufruft, wird eine neue lokale Symboltabelle für diesen Aufruf erstellt.

Eine Funktionsdefinition führt den Funktionsnamen in die aktuelle Symboltabelle ein. Der Wert des Funktionsnamens hat einen Typ, der vom Interpreter als benutzerdefinierte Funktion erkannt wird. Dieser Wert kann einem anderen Namen zugeordnet werden, der dann auch als Funktion verwendet werden kann. Dies dient als allgemeiner Umbenennungsmechanismus:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Aus anderen Sprachen kommend, könnte man einwenden, dass *fib* keine Funktion, sondern ein Arbeitsablauf ist, da es keinen Wert zurückgibt. Tatsächlich geben auch Funktionen ohne Return-Anweisung einen Wert zurück, wenn auch einen eher langweiligen. Dieser Wert wird als *None* bezeichnet (es handelt sich um einen eingebauten Namen). Das Schreiben des Wertes *None* wird normalerweise vom Interpreter unterdrückt, wenn es der einzige geschriebene Wert wäre. Du kannst es sehen, wenn du `print()` wirklich verwenden willst:





```
>>> fib(0)
>>> print(fib(0))
None
```

Es ist einfach, eine Funktion zu schreiben, die eine Liste der Zahlen der Fibonacci-Serie zurückgibt, anstatt sie zu drucken:

```
>>> def fib2(n): # gibt Fibonaccireihe bis n zurück
...     """Gib eine Liste zurück, die die Fibonaccireihe bis n enthält."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Dieses Beispiel zeigt, wie üblich, einige neue Python-Funktionen:

- Die `return`-Anweisung gibt mit einem Wert aus einer Funktion zurück. `return` ohne Ausdrucksargument gibt `None` zurück. Das Abfallen am Ende einer Funktion gibt ebenfalls `None` zurück.
- Die Anweisung `result.append(a)` ruft eine Methode auf dem Listenobjekt `result` auf. Eine Methode ist eine Funktion, die zu einem Objekt gehört und `obj.methodname` genannt wird, wobei `obj` ein Objekt ist (dies kann ein Ausdruck sein), und `methodname` ist der Name einer Methode, die durch den Typ des Objekts definiert ist. Verschiedene Typen definieren unterschiedliche Methoden. Methoden verschiedener Typen können den gleichen Namen haben, ohne Mehrdeutigkeit zu verursachen. (Es ist möglich, eigene Objekttypen und Methoden über Klassen zu definieren, siehe Klassen) Die im Beispiel gezeigte Methode `append()` ist für Listenobjekte definiert; sie fügt ein neues Element am Ende der Liste hinzu. In diesem Beispiel ist es gleichbedeutend mit `result = result + [a]`, aber effizienter.

4.7. Mehr zum Definieren von Funktionen

Es ist auch möglich, Funktionen mit einer variablen Anzahl von Argumenten zu definieren. Es gibt drei Formen, die kombiniert werden können.

4.7.1. Standardargument Werte

Die nützlichste Form ist die Angabe eines Standardwertes für ein oder mehrere Argumente. Dies erzeugt eine Funktion, die mit weniger Argumenten aufgerufen werden kann, als sie definiert ist. Zum Beispiel:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Diese Funktion kann auf verschiedene Arten aufgerufen werden:

- nur mit dem obligatorischen Argument: `ask_ok('Do you really want to quit?')`
- mit einem der optionalen Argumente: `ask_ok('OK to overwrite the file?', 2)`
- oder sogar alle Argumente angeben: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

In diesem Beispiel wird auch das Schlüsselwort `in` eingeführt. Dadurch wird geprüft, ob eine Sequenz einen bestimmten Wert enthält oder nicht.

Die Vorschlagswerte werden zum Zeitpunkt der Funktionsdefinition im Definitionsumfang ausgewertet, so dass

```
i = 5

def f(arg=i):
    print(arg)
```



```
i = 6  
f()
```

ausgegeben wird.

Wichtige Warnung: Der Standardwert wird nur einmal ausgewertet. Dies macht einen Unterschied, wenn der Standard ein veränderliches Objekt ist, wie beispielsweise eine Liste, ein Dictionary oder Instanzen der meisten Klassen. Die folgende Funktion sammelt beispielsweise die Argumente, die bei nachfolgenden Aufrufen an sie übergeben werden:

```
def f(a, L=[]):  
    L.append(a)  
    return L  
  
print(f(1))  
print(f(2))  
print(f(3))
```

Dies wird ausgegeben

```
[1]  
[1, 2]  
[1, 2, 3]
```

Wenn du nicht möchtest, dass der Standard zwischen nachfolgenden Aufrufen geteilt wird, kannst du die Funktion stattdessen so schreiben:

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

4.7.2. Keyword Argumente

Funktionen können auch mit Schlüsselwortargumenten der Form *kwarg=value* aufgerufen werden. Zum Beispiel die folgende Funktion:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

akzeptiert ein erforderliches Argument (*voltage*) und drei optionale Argumente (*state*, *action* und *type*). Diese Funktion kann auf eine der folgenden Arten aufgerufen werden:

```
parrot(1000) # 1 Positionsargument
parrot(voltage=1000) # 1 Keywordargument
parrot(voltage=1000000, action='VOOOOOM') # 2 Keywordargumente
parrot(action='VOOOOOM', voltage=1000000) # 2 Keywordargumente
parrot('a million', 'bereft of life', 'jump') # 3 Positionsargumente
parrot('a thousand', state='pushing up the daisies') # 1 Positions-, 1 Keyword
```

aber alle folgenden Aufrufe wären ungültig:

```
parrot() # erforderliches Argument fehlt
parrot(voltage=5.0, 'dead') # nicht-keyword Argument nach einem keyword
Argument
parrot(110, voltage=220) # doppelter Wert für dasselbe Argument
parrot(actor='John Cleese') # unbekanntes keyword Argument
```

In einem Funktionsaufruf müssen Schlüsselwortargumente positionellen Argumenten folgen. Alle übergebenen Schlüsselwortargumente müssen mit einem der von der Funktion akzeptierten Argumente übereinstimmen (z.B. ist *actor* kein gültiges Argument für die Funktion *parrot*), und ihre Reihenfolge ist nicht wichtig. Dazu gehören auch nicht-optionale Argumente (z.B. *parrot(voltage=1000)* ist auch gültig). Kein Argument darf einen Wert mehr als einmal erhalten. Hier ist ein Beispiel, das an dieser Einschränkung scheitert:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for keyword argument 'a'
```

Wenn ein letzter Formalparameter der Form ***name* vorhanden ist, erhält er ein Dictionary (siehe Mapping Types - dict), das alle Schlüsselwortargumente enthält, mit Ausnahme derjenigen, die einem Formalparameter entsprechen. Dies kann mit einem Formalparameter der Form **name* (beschrieben im nächsten Unterabschnitt) kombiniert werden, der ein Tupel mit den Positionsargumenten außerhalb der Formalparameterliste erhält. (**name* muss vor ***name* stehen.) Zum Beispiel, wenn wir eine Funktion wie diese definieren:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Man könnte sie so aufrufen:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```



und natürlich würde es ausgeben:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Bitte bedenke, dass die Reihenfolge, in der die Schlüsselwortargumente gedruckt werden, garantiert der Reihenfolge entspricht, in der sie im Funktionsaufruf angegeben wurden.

4.7.3. Beliebige Argumentlisten


Schließlich ist die am seltensten verwendete Option, anzugeben, dass eine Funktion mit einer beliebigen Anzahl von Argumenten aufgerufen werden kann. Diese Argumente werden in einem Tupel zusammengefasst (siehe Tupel und Sequenzen). Vor der variablen Anzahl von Argumenten können null oder mehr normale Argumente auftreten.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normalerweise sind diese variablen Argumente die letzten in der Liste der Formalparameter, da sie alle verbleibenden Eingabeargumente, die an die Funktion übergeben werden, aufnehmen. Alle Formalparameter, die nach dem Parameter `*args` auftreten, sind 'keyword-only'-Argumente, was bedeutet, dass sie nur als Schlüsselwörter und nicht als Positionsargumente verwendet werden können.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
```





```
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.7.4. Entpacken von Argumentenlisten

Die umgekehrte Situation tritt ein, wenn sich die Argumente bereits in einer Liste oder einem Tupel befinden, aber für einen Funktionsaufruf, der separate Positionsargumente erfordert, entpackt werden müssen. Zum Beispiel erwartet die eingebaute `range()` Funktion separate Start- und Stoppargumente. Wenn sie nicht separat verfügbar sind, schreiben Sie den Funktionsaufruf mit dem `*`-Operator, um die Argumente aus einer Liste oder einem Tupel zu entpacken:

```
>>> list(range(3, 6))           # normaler Aufruf mit getrennten Argumenten
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))         # Aufruf mit Elementen aus seiner Liste
entpackt [3, 4, 5]
```

Auf die gleiche Weise können Dictionaries Keyword-Argumente mit dem `**`-Operator liefern:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action":
"VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's
bleedin' demised !
```

4.7.5. Lambda Ausdrücke

Kleine anonyme Funktionen können mit dem Schlüsselwort `lambda` erstellt werden. Diese Funktion gibt die Summe ihrer beiden Argumente zurück: `lambda a, b: a+b`. Lambda-Funktionen können überall dort eingesetzt werden, wo Funktionsobjekte benötigt werden. Sie sind syntaktisch auf einen einzigen Ausdruck beschränkt. Semantisch gesehen sind sie nur syntaktischer Zucker für eine normale Funktionsdefinition. Wie geschachtelte



Funktionsdefinitionen können Lambda-Funktionen auf Variablen aus dem enthaltenen Scope verweisen:

```
>>> def make_incrementor(n):  
...     return lambda x: x + n  
...  
>>> f = make_incrementor(42)  
>>> f(0)  
42  
>>> f(1)  
43
```

Das obige Beispiel verwendet einen Lambda-Ausdruck, um eine Funktion zurückzugeben. Eine weitere Verwendung ist die Übergabe einer kleinen Funktion als Argument:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]  
>>> pairs.sort(key=lambda pair: pair[1])  
>>> pairs  
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6. Dokumentationszeichenketten


Hier sind einige Konventionen über den Inhalt und die Formatierung von Dokumentationszeichenketten.

Die erste Zeile sollte immer eine kurze, prägnante Zusammenfassung des Zwecks des Objekts sein. Der Kürze halber sollte es nicht explizit den Namen oder Typ des Objekts angeben, da diese auf andere Weise verfügbar sind (außer wenn der Name zufällig ein Verb ist, das die Operation einer Funktion beschreibt). Diese Zeile sollte mit einem Großbuchstaben beginnen und mit einem Punkt enden.

Wenn es mehr Zeilen in der Dokumentationszeichenkette gibt, sollte die zweite Zeile leer sein, wodurch die Zusammenfassung optisch vom Rest der Beschreibung getrennt wird. Die folgenden Zeilen sollten ein oder mehrere Absätze sein, die die Aufrufkonventionen des Objekts, seine Nebeneffekte usw. beschreiben.

Der Python-Parser entfernt keine Einrückungen von mehrzeiligen Strings in Python, sodass Werkzeuge, welche die Dokumentation verarbeiten, die Einrückungen auf Wunsch entfernen müssen. Dies geschieht nach der folgenden Konvention. Die erste nicht-leere Zeile nach der





ersten Zeile der Zeichenkette bestimmt den Einrückungswert für die gesamte Dokumentationszeichenkette. (Wir können die erste Zeile nicht verwenden, da sie im Allgemeinen an die Eröffnungsanführungszeichen der Zeichenkette angrenzt, sodass ihre Einrückung nicht im Zeichenkettenliteral sichtbar ist.) Leerzeichen "äquivalent" zu dieser Einrückung werden dann vom Anfang aller Zeilen der Zeichenkette entfernt. Zeilen, die weniger eingerückt sind, sollten nicht auftreten, aber wenn sie auftreten, sollten alle ihre führenden Leerzeichen entfernt werden. Die Äquivalenz von Leerzeichen sollte nach der Erweiterung der Tabs (normalerweise auf 8 Leerzeichen) getestet werden.

Hier ist ein Beispiel für eine mehrzeilige Dokumentation:

```
>>> def my_function():
...     """Mache nichts, aber dokumentiere es.
...
...     Nein, es macht wirklich nichts.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.7.7. Funktionsbeschreibungen

Funktionsannotationen sind völlig optionale Metadateninformationen über die von benutzerdefinierten Funktionen verwendeten Typen (siehe PEP 3107 und PEP 484 für weitere Informationen).

Anmerkungen werden im Attribut `__annotations__` der Funktion als Dictionary gespeichert und haben keinen Einfluss auf den anderen Teil der Funktion. Parameterannotationen werden durch ein Doppelpunkt nach dem Parameternamen definiert, gefolgt von einem Ausdruck, der den Wert der Annotation auswertet. Rückgabekommentare werden durch ein Literal `->`, gefolgt von einem Ausdruck, zwischen der Parameterliste und dem Doppelpunkt definiert, der das Ende der `def`-Anweisung bezeichnet. Das folgende Beispiel hat ein Positionsargument, ein Schlüsselwortargument und den Rückgabewert annotiert:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
```

```
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class
'str'>}}
Arguments: spam eggs
'spam and eggs'
```

4.8. Intermezzo: Coding Stil

Jetzt, da du im Begriff bist, längere, komplexere Stücke von Python zu schreiben, ist es ein guter Zeitpunkt, über den Codierstil zu sprechen. Die meisten Sprachen können in verschiedenen Stilen geschrieben (oder prägnanter, formatiert) werden; einige sind besser lesbar als andere. Es ist immer eine gute Idee, es anderen leicht zu machen, deinen Code zu lesen, und die Einführung eines schönen Codestils hilft dabei enorm.

Für Python hat sich PEP 8 als der Style Guide herausgestellt, an den sich die meisten Projekte halten; es fördert einen sehr lesbaren und ansprechenden Codestil. Jeder Python-Entwickler sollte es irgendwann einmal lesen; hier sind die wichtigsten Punkte für dich extrahiert:


- Benutze 4-Leerzeichen-Einrückung und keine Tabs.

4 Leerzeichen sind ein guter Kompromiss zwischen kleiner Vertiefung (ermöglicht eine größere Verschachtelungstiefe) und großer Vertiefung (besser lesbar). Tabs führen zu Verwirrung und sollten am besten weggelassen werden.

- Zeilen so umbrechen, dass sie 79 Zeichen nicht überschreiten.

Dies hilft Anwendern bei kleinen Displays und ermöglicht es bei größeren Displays, mehrere Code-Dateien nebeneinander zu haben.

- Benutze Leerzeilen, um Funktionen und Klassen zu trennen, und größere Codeblöcke innerhalb von Funktionen.
- Wenn möglich, solltest du Kommentare in eine eigene Zeile setzen.
- Verwende Docstrings.

- 
- Verwende Leerzeichen um Operatoren und nach Kommas, aber nicht direkt innerhalb von Klammerkonstrukten: `a = f(1,2) + g(3,4)`.
 - Benenne Klassen und Funktionen einheitlich; die Konvention ist, *UpperCamelCase* für Klassen und *lower_case_with_underscores* für Funktionen und Methoden zu verwenden. Verwende immer *self* als Namen für das erste Methodenargument.
 - Verwende keine ausgefallenen Kodierungen, wenn dein Code für die Verwendung im internationalen Umfeld bestimmt ist. Pythons Standard, UTF-8 oder sogar einfaches ASCII funktionieren in jedem Fall am besten.
 - Verwende auch keine Nicht-ASCII-Zeichen in Identifikatoren, wenn nur die geringste Wahrscheinlichkeit besteht, dass Personen, die eine andere Sprache sprechen, den Code lesen oder warten.

Fußnoten

- [\[1\]](#) Eigentlich wäre ein Aufruf nach Objektreferenz eine bessere Beschreibung, denn wenn ein veränderliches Objekt übergeben wird, sieht der Aufrufer alle Änderungen, die der Aufrufer daran vornimmt (Elemente, die in eine Liste eingefügt werden).



5. Datenstrukturen

Dieses Kapitel beschreibt einige Dinge, die du bereits näher kennengelernt hast, und fügt auch einige neue Dinge hinzu.

5.1. Mehr zu Listen

Der Listendatentyp hat noch einige weitere Methoden. Hier sind alle Methoden von Listenobjekten aufgeführt:

`list.append(x)`

Fügt ein Element am Ende der Liste hinzu. Entspricht $a[\text{len}(a):] = [x]$.

`list.extend(iterable)`

Erweitere die Liste, indem du alle Elemente aus der iterierbaren Liste anhängst. Entspricht $a[\text{len}(a):] = \text{iterable}$.

`list.insert(i, x)`

Einfügen eines Elements an einer bestimmten Position. Das erste Argument ist der Index des Elements, vor dem eingefügt werden soll, also fügt $a.\text{insert}(0, x)$ am Anfang der Liste ein, und $a.\text{insert}(\text{len}(a), x)$ entspricht $a.\text{append}(x)$.

`list.remove(x)`

Entfernt das erste Element aus der Liste, dessen Wert gleich x ist. Es löst einen `ValueError` aus, wenn es kein solches Element gibt.

`list.pop([i])`

Entfernt das Element an der angegebenen Position in der Liste und gibt es zurück. Wenn kein Index angegeben wird, entfernt $a.\text{pop}()$ den letzten Eintrag in der Liste und gibt ihn zurück. (Die eckigen Klammern um das i in der Methodensignatur bezeichnen, dass der Parameter optional ist, nicht dass du an dieser Stelle eckige Klammern eingeben sollst. Du wirst diese Schreibweise häufig in der Python Library Reference sehen.)



```
list.clear()
```

Alle Elemente aus der Liste entfernen. Entspricht *del a[:]*.

```
list.index(x[, start[, end]])
```

Liefert einen nullbasierten Index in der Liste des ersten Elements, dessen Wert gleich x ist. Wirft einen ValueError, wenn es kein solches Element gibt.

Die optionalen Argumente *start* und *end* werden wie in der Slice-Notation interpretiert und dienen dazu, die Suche auf eine bestimmte Teilfolge der Liste zu beschränken. Der zurückgegebene Index wird relativ zum Anfang der gesamten Sequenz und nicht zum Startargument berechnet.

```
list.count(x)
```

Liefert die Anzahl der Male, die x in der Liste erscheint.

```
list.sort(key=None, reverse=False)
```

Die Elemente der Liste an Ort und Stelle sortieren (die Argumente können zur Anpassung der Sortierung verwendet werden, siehe *sorted()* für ihre Erklärung).

```
list.reverse()
```

Umkehrung der Elemente der vorhandenen Liste.

```
list.copy()
```

Gib eine oberflächliche Kopie der Liste zurück. Äquivalent zu *a[:]*.



Ein Beispiel, das die meisten Listenmethoden verwendet:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4) # Finde nächstes Vorkommen von banana ab
Position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Du hast vielleicht bemerkt, dass Methoden wie *insert*, *remove* oder *sort*, die nur die Liste ändern, keinen Rückgabewert gedruckt haben - sie geben den Standardwert *None* zurück. [1] Dies ist ein Konstruktionsprinzip für alle veränderlichen Datenstrukturen in Python.

5.1.1. Listen als Stacks verwenden

Die Listenmethoden machen es sehr einfach, eine Liste als Stack zu verwenden, wobei das zuletzt hinzugefügte Element das erste abgerufene Element ist ("last-in, first-out"). Um einen Eintrag an die Spitze des Stacks hinzuzufügen, verwende `append()`. Um ein Element vom oberen Ende des Stacks abzurufen, verwende `pop()` ohne expliziten Index. Zum Beispiel:

```

>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]

```

5.1.2. Listen als Warteschlangen verwenden

Es ist auch möglich, eine Liste als Warteschlange (Queue) zu verwenden, wobei das erste hinzugefügte Element das erste abgerufene Element ist ("first-in, first-out"); Listen sind jedoch für diesen Zweck nicht effizient. Während Appends und Pops vom Ende der Liste schnell sind, ist das Einfügen oder Entfernen vom Anfang einer Liste langsam (da alle anderen Elemente um eins verschoben werden müssen).

Um eine Warteschlange zu implementieren, benutze `collections.deque`, das so konzipiert wurde, dass es schnelle Appends und Pops von beiden Seiten hat. Zum Beispiel:

```

>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry kommt an
>>> queue.append("Graham")         # Graham kommt an
>>> queue.popleft()                # der zuerst kam geht jetzt
'Eric'
>>> queue.popleft()                # der als zweites kam geht jetzt
'John'
>>> queue                          # verbleibende Queue in Reihenfolge des
Erscheinens
deque(['Michael', 'Terry', 'Graham'])

```



5.1.3. Listen-Abstraktion

Listen-Abstraktion bietet eine übersichtliche Möglichkeit, Listen zu erstellen. Gängige Anwendungen sind die Erstellung neuer Listen, bei denen jedes Element das Ergebnis einiger Operationen ist, die auf jedes Element einer anderen Sequenz oder eines Iterables angewendet werden, oder die Erstellung einer Teilfolge von Elementen, die eine bestimmte Bedingung erfüllen.

Nehmen wir zum Beispiel an, wir wollen eine Liste von Quadraten erstellen, wie z.B.:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Bitte bedenke, dass dadurch eine Variable namens `x` erzeugt (oder überschrieben) wird, die nach Beendigung der Schleife noch existiert. Wir können die Liste der Quadrate ohne Nebenwirkungen berechnen mit:

```
squares = list(map(lambda x: x**2, range(10)))
```


oder, äquivalent:

```
squares = [x**2 for x in range(10)]
```

was prägnanter und lesbarer ist.

Eine Listen-Abstraktion besteht aus Klammern, die einen Ausdruck enthalten, gefolgt von einer `for`-Anweisung, dann null oder mehr `for`- oder `if`-Anweisungen. Das Ergebnis ist eine neue Liste, die sich aus der Auswertung des Ausdrucks im Kontext der darauffolgenden Anweisungen ergibt. Diese Listen-Abstraktion kombiniert beispielsweise die Elemente zweier Listen, wenn sie ungleich sind:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

und es ist gleichwertig mit:


```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Man beachte, dass die Reihenfolge der for- und if-Anweisungen in diesen beiden Ausschnitten gleich ist.

Wenn der Ausdruck ein Tupel ist (z.B. das (x, y) im vorherigen Beispiel), muss er in Klammern gesetzt werden.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # erstellt eine neue Liste mit den Werten verdoppelt
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtert die Liste, um negative Werte auszuschließen
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # wendet eine Funktion auf alle Elemente an
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # ruft eine Methode auf jedem Element auf
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # erstellt eine Liste von 2-Tupeln wie (Zahl, Quadrat)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # das Tupel muss in Klammern stehen, sonst gibt es einen Error
>>> [x, x**2 for x in range(6)]
```





```
File "<stdin>", line 1, in <module>
  [x, x**2 for x in range(6)]
      ^
```

SyntaxError: invalid syntax

```
>>> # kürzt eine Liste mit einer Listcomp mit 2 "for"
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listen-Abstraktion kann komplexe Ausdrücke und verschachtelte Funktionen enthalten:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. Verschachtelte Listen-Abstraktion

Der erste Ausdruck in einer Listen-Abstraktion kann ein beliebiger Ausdruck sein, einschließlich einer anderen Listen-Abstraktion.


Beachte das folgende Beispiel einer 3x4-Matrix, die als Liste von 3 Listen der Länge 4 implementiert ist:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Die folgende Listen-Abstraktion soll Zeilen und Spalten transponieren:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Wie wir im vorherigen Abschnitt gesehen haben, wird die verschachtelte Listen-Abstraktion im Kontext der folgenden for-Anweisung ausgewertet, so dass dieses Beispiel äquivalent ist zu:



```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

was wiederum dasselbe ist wie:

```
>>> transposed = []
>>> for i in range(4):
...     # die folgenden 3 Zeilen implementieren die verkettete listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In der realen Welt solltest du integrierte Funktionen den komplexen Ablaufanweisungen vorziehen. Die `zip()`-Funktion würde für diesen Anwendungsfall einen guten Job machen:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Siehe Entpacken von Argumentenlisten für Details zum Sternchen in dieser Zeile.

5.2. Das `del` Statement

Es gibt eine Möglichkeit, ein Element aus einer Liste zu entfernen, die seinen Index anstelle seines Wertes enthält: die `del`-Anweisung. Dies unterscheidet sich von der `pop()`-Methode, die einen Wert zurückgibt. Die `Del`-Anweisung kann auch verwendet werden, um Slices aus einer Liste zu entfernen oder die gesamte Liste zu löschen (was wir zuvor durch Zuweisung einer leeren Liste an die Slice getan haben). Zum Beispiel:



```

>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]

```

del kann auch verwendet werden, um ganze Variablen zu löschen:

```

>>> del a

```

Die Bezugnahme auf den Namen *a* ist nach dieser Zeile ein Fehler (zumindest bis ihm ein anderer Wert zugewiesen wird). Wir werden später andere Einsatzmöglichkeiten für del finden.

5.3. Tupel und Sequenzen

Wir haben gesehen, dass Listen und Strings viele gemeinsame Eigenschaften haben, wie z.B. Indexierungs- und Slicingoperationen. Sie sind zwei Beispiele für Sequenzdatentypen (siehe Sequenztypen - Liste, Tupel, Bereich). Da Python eine sich entwickelnde Sprache ist, können weitere Sequenzdatentypen hinzugefügt werden. Es gibt noch einen weiteren Datentyp der Standardsequenz: das Tupel.

Ein Tupel besteht aus einer Reihe von Werten, die z.B. durch Kommas getrennt sind:

```

>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tupel können verschachtelt sein:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tupel sind unveränderlich:

```

```

... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # aber sie können veränderliche Objekte enthalte:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])

```

Wie du siehst, werden bei der Ausgabe Tupel immer in Klammern eingeschlossen, so dass verschachtelte Tupel korrekt interpretiert werden; sie können mit oder ohne umgebende Klammern eingegeben werden, obwohl oft trotzdem Klammern erforderlich sind (wenn das Tupel Teil eines größeren Ausdrucks ist). Es ist nicht möglich, den einzelnen Elementen eines Tupels zuzuordnen, es ist jedoch möglich, Tupel zu erstellen, die veränderbare Objekte wie Listen enthalten.


Obwohl Tupel den Listen ähnlich erscheinen mögen, werden sie oft in verschiedenen Situationen und für verschiedene Zwecke verwendet. Tupel sind unveränderlich und enthalten in der Regel eine heterogene Abfolge von Elementen, auf die durch Entpacken (siehe weiter unten in diesem Abschnitt) oder Indizieren (oder bei Namenstupeln sogar durch Attribute) zugegriffen wird. Listen sind veränderbar, und ihre Elemente sind in der Regel homogen und werden durch Iteration über die Liste aufgerufen.

Ein besonderes Problem ist die Konstruktion von Tupeln mit 0 oder 1 Elementen: Die Syntax hat einige zusätzliche Besonderheiten, um diese zu berücksichtigen. Leere Tupel werden durch ein leeres Paar Klammern konstruiert; ein Tupel mit einem Element wird konstruiert, indem einem Wert ein Komma folgt (es genügt nicht, einen einzelnen Wert in Klammern einzuschließen). Hässlich, aber effektiv. Zum Beispiel:

```

>>> empty = ()
>>> singleton = 'hello',    # <-- beachte das angehängte Komma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)

```



Das Statement `t = 12345, 54321, 'hello!'` ist ein Beispiel für Tupel Packing: Die Werte `12345`, `54321` und `'hello!'` werden in einem Tupel zusammengepackt. Die umgekehrte Operation ist ebenfalls möglich:

```
>>> x, y, z = t
```

Dies wird entsprechend als Sequenzauspacken bezeichnet und funktioniert für jede Sequenz auf der rechten Seite. Das Entpacken der Sequenz erfordert, dass sich auf der linken Seite des Gleichheitszeichens so viele Variablen befinden, wie es Elemente in der Sequenz gibt. Bitte bedenke, dass die Mehrfachzuordnung eigentlich nur eine Kombination aus Tupel- und Sequenzentpacken ist.

5.4. Sets

Python enthält auch einen Datentyp für Mengen (engl.: Sets). Ein Set ist eine ungeordnete Sammlung ohne doppelte Elemente. Zu den grundlegenden Anwendungen gehören das Testen des Enthaltenseins und die Beseitigung doppelter Einträge. Set-Objekte unterstützen auch mathematische Operationen wie Vereinigung, Durchschnitt, Differenz und symmetrische Differenz.

Geschweifte Klammern oder die `set()` Funktion können verwendet werden, um Sets zu erstellen. Hinweis: Um eine leere Menge zu erstellen, musst du `set()` verwenden, nicht `{}`; letzteres erstellt ein leeres Dictionary, eine Datenstruktur, die wir im nächsten Abschnitt besprechen.

Hier ist eine kurze Demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                # zeige, dass Duplikate entfernt wurden
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket           # schnelles Überprüfen auf Vorhandensein
True
>>> 'crabgrass' in basket
False

>>> # Zeigt Mengenoperationen auf einzelnen Buchstaben zweier Wörter
...
```

```

>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                     # Buchstaben speziell in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                                 # Buchstaben in a, aber nicht in b
{'r', 'd', 'b'}
>>> a | b                                 # Buchstaben in a, b oder beiden
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                 # Buchstaben sowohl in a als auch b
{'a', 'c'}
>>> a ^ b                                 # Buchstaben in a oder b, aber nicht in
beiden
{'r', 'd', 'b', 'm', 'z', 'l'}

```

Ähnlich wie bei den Listen-Abstraktionen werden auch die Mengen-Abstraktionen unterstützt:

```


>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}

```

5.5. Dictionaries

Ein weiterer nützlicher Datentyp, der in Python integriert ist, ist das Dictionary (siehe Mapping-Typen - dict). Dictionaries werden manchmal in anderen Sprachen als "assoziative Speicher" oder "assoziative Arrays" gefunden. Im Gegensatz zu Sequenzen, die durch eine Reihe von Zahlen indiziert sind, werden Dictionaries durch Schlüssel indiziert, die jeder unveränderliche Typ sein können; Strings und Zahlen können immer Schlüssel sein. Tupel können als Schlüssel verwendet werden, wenn sie nur Strings, Zahlen oder Tupel enthalten; wenn ein Tupel ein veränderbares Objekt direkt oder indirekt enthält, kann es nicht als Schlüssel verwendet werden. Du kannst Listen nicht als Schlüssel verwenden, da Listen mit Indexzuweisungen, Slice-Zuweisungen oder Methoden wie `append()` und `extend()` geändert werden können.

Es ist am besten, sich ein Dictionary als einen Satz von Schlüssel- und Wertepaaren vorzustellen, mit der Anforderung, dass die Schlüssel eindeutig sind (innerhalb eines Dictionary). Ein Paar Klammern erzeugt ein leeres Wörterbuch: `{}`. Das Platzieren einer kommasetrennten Liste von `key:value`-Paaren innerhalb der geschweiften Klammern fügt dem Dictionary initiale `key:value`-Paare hinzu; so werden auch Dictionaries bei der Ausgabe geschrieben.



Die wichtigsten Operationen in einem Dictionary sind das Speichern eines Wertes mit einem Schlüssel und das Extrahieren des Wertes, der dem Schlüssel gegeben ist. Es ist auch möglich, ein key:value-Paar mit *del* zu löschen. Wenn du mit einem Schlüssel speicherst, der bereits verwendet wird, wird der alte Wert, der diesem Schlüssel zugeordnet ist, verworfen. Es ist ein Fehler, einen Wert mit einem nicht existierenden Schlüssel zu extrahieren.

Die Ausführung von *list(d)* auf einem Dictionary gibt eine Liste aller im Dictionary verwendeten Schlüssel in Einfügereihenfolge zurück (wenn du willst, dass es sortiert wird, verwende stattdessen einfach *sorted(d)*). Um zu überprüfen, ob ein einzelner Schlüssel im Dictionary vorhanden ist, verwende das Schlüsselwort *in*.

Hier ist ein kleines Beispiel mit einem Dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Der *dict()*-Konstruktor erstellt Dictionaries direkt aus Sequenzen von Schlüssel-Wert-Paaren:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```




Darüber hinaus können mit Hilfe von Dict-Abstraktionen Dictionaries aus beliebigen Schlüssel- und Wertausdrücken erstellt werden:

```
>>> {x: x**2 for x in (2, 4, 6)}  
{2: 4, 4: 16, 6: 36}
```

Wenn die Schlüssel einfache Strings sind, ist es manchmal einfacher, Paare mit Schlüsselwortargumenten anzugeben:

```
>>> dict(sape=4139, guido=4127, jack=4098)  
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6. Schleifentechniken

Beim Durchlaufen von Dictionaries können der Schlüssel und der entsprechende Wert gleichzeitig mit der Methode *items()* abgerufen werden.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}  
>>> for k, v in knights.items():  
...     print(k, v)  
...  
gallahad the pure  
robin the brave
```

Beim Durchlaufen einer Sequenz können der Positionsindex und der entsprechende Wert gleichzeitig mit der Funktion *enumerate()* abgerufen werden.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):  
...     print(i, v)  
...  
0 tic  
1 tac  
2 toe
```



Um zwei oder mehr Sequenzen gleichzeitig zu durchlaufen, können die Einträge mit der Funktion `zip()` gekoppelt werden.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```


Um eine Sequenz in umgekehrter Reihenfolge zu durchlaufen, gibst du zuerst die Sequenz in Vorwärtsrichtung an und rufst dann die Funktion `reversed()` auf.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Um eine Sequenz in sortierter Reihenfolge zu durchlaufen, verwende den Befehl `sorted()`, der eine neue sortierte Liste zurückgibt, während die Quelle unverändert bleibt.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```





Es ist manchmal verlockend, eine Liste zu ändern, während du sie überfliegst; jedoch ist es oft einfacher und sicherer, stattdessen eine neue Liste zu erstellen.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. Mehr zu Bedingungen

Die in *while*- und *if*-Anweisungen verwendeten Bedingungen können beliebige Operatoren enthalten, nicht nur Vergleiche.

Die Vergleichsoperatoren *in* und *not in* prüfen, ob ein Wert in einer Sequenz vorkommt (bzw. nicht vorkommt). Die Operatoren *is* und *is not* vergleichen, ob zwei Objekte wirklich das gleiche Objekt sind; dies gilt nur für veränderliche Objekte wie Listen. Alle Vergleichsoperatoren haben die gleiche Priorität, die niedriger ist als die aller numerischen Operatoren.

Vergleiche können verkettet werden. Zum Beispiel testet $a < b == c$, ob *a* kleiner als *b* ist und *b* außerdem gleich *c* ist.

Vergleiche können mit den booleschen Operatoren *and* und *or* kombiniert werden, und das Ergebnis eines Vergleichs (oder eines anderen booleschen Ausdrucks) kann mit *not* negiert werden. Diese haben niedrigere Prioritäten als Vergleichsoperatoren; zwischen ihnen hat *not* die höchste Priorität und *or* die niedrigste, sodass $A \text{ and } \text{not } B \text{ or } C$ gleichbedeutend ist mit $(A \text{ and } (\text{not } B)) \text{ or } C$. Wie immer können Klammern verwendet werden, um die gewünschte Zusammensetzung auszudrücken.

Die booleschen Operatoren *and* und *or* sind so genannte Kurzschlussoperatoren: Ihre Argumente werden von links nach rechts ausgewertet, und die Auswertung stoppt, sobald das Ergebnis bestimmt ist. Wenn beispielsweise *A* und *C* wahr sind, aber *B* falsch ist, wertet $A \text{ and } B \text{ and } C$ den Ausdruck *C* nicht aus. Bei Verwendung als allgemeiner Wert und nicht als Boolean ist der Rückgabewert eines Kurzschlussoperators das zuletzt ausgewertete Argument.

Es ist möglich, das Ergebnis eines Vergleichs oder eines anderen booleschen Ausdrucks einer Variablen zuzuordnen. Zum Beispiel,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```


Es ist zu beachten, dass in Python im Gegensatz zu C keine Zuweisung innerhalb von Ausdrücken erfolgen kann. C-Programmierer mögen darüber meckern, aber es vermeidet eine gemeinsame Klasse von Problemen, die in C-Programmen auftreten: Das Schreiben von = in einem Ausdruck, wenn == gemeint ist.

5.8. Vergleich von Sequenzen und anderen Typen

Sequenz-Objekte können mit anderen Objekten mit dem gleichen Sequenztyp verglichen werden. Der Vergleich verwendet lexikographische Ordnung: Zuerst werden die ersten beiden Elemente verglichen, und wenn sie sich unterscheiden, bestimmt dies das Ergebnis des Vergleichs; wenn sie gleich sind, werden die nächsten beiden Elemente verglichen, usw., bis beide Folgen erschöpft sind. Wenn zwei zu vergleichende Elemente selbst Sequenzen gleichen Typs sind, wird der lexikographische Vergleich rekursiv durchgeführt. Wenn alle Elemente von zwei Sequenzen gleich sind, gelten die Sequenzen als gleich. Wenn eine Sequenz eine anfängliche Untersequenz der anderen ist, ist die kürzere Sequenz die kleinere (geringere). Die Lexikographische Ordnung für Strings verwendet die Unicode Codepunktnummer, um einzelne Zeichen zu ordnen. Einige Beispiele für Vergleiche zwischen Sequenzen des gleichen Typs:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Es ist zu beachten, dass der Vergleich von Objekten verschiedener Typen mit < oder > erlaubt ist, vorausgesetzt, die Objekte verfügen über geeignete Vergleichsmethoden. Beispielsweise werden gemischte numerische Typen entsprechend ihrem Zahlenwert verglichen, so dass 0



gleich 0,0 ist, etc. Andernfalls löst der Interpreter, anstatt eine beliebige Reihenfolge vorzugeben, eine `TypeError` aus.

Fußnoten

- [1] Andere Sprachen können das veränderte Objekt zurückgeben, was Methodenverkettung ermöglicht, wie z.B. `d->insert("a")->remove("b")->sort();`.

6. Module

Wenn du den Python-Interpreter verlässt und ihn erneut eingibst, gehen die von dir vorgenommenen Definitionen (Funktionen und Variablen) verloren. Wenn du also ein etwas längeres Programm schreiben möchtest, solltest du besser einen Texteditor verwenden, um die Eingabe für den Interpreter vorzubereiten und sie stattdessen mit dieser Datei als Eingabe ausführen. Dies wird als Erstellen eines Skripts bezeichnet. Wenn dein Programm länger wird, solltest du es zur leichteren Wartung in mehrere Dateien aufteilen. Du kannst auch eine praktische Funktion verwenden, die du in mehreren Programmen geschrieben hast, ohne ihre Definition in jedes Programm zu kopieren.

Um dies zu unterstützen, hat Python eine Möglichkeit, Definitionen in eine Datei zu schreiben und sie in einem Skript oder in einer interaktiven Instanz des Interpreters zu verwenden. Eine solche Datei wird als Modul bezeichnet; Definitionen aus einem Modul können in andere Module oder in das Hauptmodul importiert werden (die Sammlung von Variablen, auf die Sie in einem auf der obersten Ebene und im Taschenrechnermodus ausgeführten Skript Zugriff haben).

Ein Modul ist eine Datei mit Python-Definitionen und -Anweisungen. Der Dateiname ist der Modulname mit der Endung `.py`. Innerhalb eines Moduls ist der Name des Moduls (als Zeichenkette) als Wert der globalen Variablen `__name__` verfügbar. Benutze beispielsweise deinen bevorzugten Texteditor, um eine Datei namens `fibonacci.py` im aktuellen Verzeichnis mit folgendem Inhalt zu erstellen:

```
# Modul Fibonaccizahlen

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

def fib2(n):   # gibt Fibonaccireihe bis n zurück
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
```



```
return result
```

Gib nun den Python-Interpreter ein und importiere dieses Modul mit dem folgenden Befehl:

```
>>> import fibo
```

Dabei werden die Namen der in *fibo* definierten Funktionen nicht direkt in die aktuelle Symboltabelle eingetragen, sondern nur der Modulname *fibo*. Über den Modulnamen kann auf die Funktionen zugegriffen werden:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Wenn du vorhast, eine Funktion häufig zu verwenden, kannst du sie einem lokalen Namen zuordnen:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. Mehr zu Modulen

Ein Modul kann sowohl ausführbare Anweisungen als auch Funktionsdefinitionen enthalten. Diese Anweisungen dienen zur Initialisierung des Moduls. Sie werden nur bei der ersten Begegnung mit dem Modulnamen in einer Importanweisung ausgeführt. [1] (Sie werden auch ausgeführt, wenn die Datei als Skript ausgeführt wird.)

Jedes Modul hat seine eigene private Symboltabelle, die von allen im Modul definierten Funktionen als globale Symboltabelle verwendet wird. So kann der Autor eines Moduls globale Variablen im Modul verwenden, ohne sich Gedanken über versehentliche Konflikte mit den globalen Variablen eines Benutzers zu machen. Andererseits, wenn du weißt, was du tust,



kannst du die globalen Variablen eines Moduls mit der gleichen Schreibweise angehen, die auch für seine Funktionen verwendet wird, *modname.itemname*.

Module können andere Module importieren. Es ist üblich, aber nicht erforderlich, alle Import-Anweisungen an den Anfang eines Moduls (oder Skripts) zu stellen. Die importierten Modulnamen werden in die globale Symboltabelle des importierenden Moduls gestellt.

Es gibt eine Variante der Importanweisung, die Namen aus einem Modul direkt in die Symboltabelle des importierenden Moduls importiert. Zum Beispiel:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Der Modulname, aus dem die Importe übernommen werden, wird dabei nicht in die lokale Symboltabelle eingetragen (im Beispiel ist also *fibo* nicht definiert).

Es gibt sogar eine Variante, um alle Namen zu importieren, die ein Modul definiert:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```


Dabei werden alle Namen mit Ausnahme derjenigen importiert, die mit einem Unterstrich () beginnen. In den meisten Fällen verwenden Python-Programmierer diese Funktion nicht, da sie einen unbekannten Satz von Namen in den Interpreter einführt und möglicherweise einige Dinge, die du bereits definiert hast, überschreibt.

Bitte bedenke, dass die Praxis des Imports von *** aus einem Modul oder Paket im Allgemeinen verpönt ist, da sie oft zu schlecht lesbarem Code führt. Es ist jedoch in Ordnung, es zu verwenden, um die Eingabe in interaktiven Sitzungen zu speichern.

Wenn auf den Modulnamen ein *as* folgt, dann ist der Name, der auf *as* folgt, direkt an das importierte Modul gebunden.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```





Dies ist praktisch der Import des Moduls auf die gleiche Weise wie mit *import fibo*, mit dem einzigen Unterschied, dass es als *fib* verfügbar ist.

Es kann auch mit ähnlichen Effekten verwendet werden, wenn *from* benutzt:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1.1. Ausführen von Modulen als Skripte

Wenn du ein Python-Modul mit

```
python fibo.py <arguments>
```

aufrufst, wird der Code im Modul ausgeführt, als ob du ihn importiert hättest, aber mit dem `__name__` auf `"__main__"` gesetzt. Das bedeutet, dass du durch Hinzufügen dieses Codes am Ende deines Moduls:

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

du die Datei sowohl als Skript als auch als importierbares Modul nutzbar machen kannst, da der Code, der die Befehlszeile analysiert, nur dann läuft, wenn das Modul als "main"-Datei ausgeführt wird:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Wenn das Modul importiert wird, wird der Code nicht ausgeführt:

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



Dies wird häufig verwendet, um entweder eine komfortable Benutzeroberfläche für ein Modul bereitzustellen oder zu Testzwecken (die Ausführung des Moduls als Skript führt eine Testsuite aus).

6.1.2. Der Modul-Suchpfad

Wenn ein Modul namens `Spam` importiert wird, sucht der Interpreter zunächst nach einem eingebauten Modul mit diesem Namen. Wenn nichts gefunden wird, wird dann nach einer Datei namens `spam.py` in einer Liste von Verzeichnissen gesucht, die durch die Variable `sys.path` gegeben sind. `sys.path` wird von diesen Speicherorten aus initialisiert:

- Das Verzeichnis, das das Eingabeskript enthält (oder das aktuelle Verzeichnis, wenn keine Datei angegeben ist).
- `PYTHONPATH` (eine Liste von Verzeichnisnamen, mit der gleichen Syntax wie die Shell-Variable `PATH`).
- Der installationsspezifische Standard.

Nach der Initialisierung können Python-Programme `sys.path` ändern. Das Verzeichnis, das das ausgeführte Skript enthält, wird am Anfang des Suchpfades vor dem Standardbibliothekspfad platziert. Das bedeutet, dass Skripte in diesem Verzeichnis anstelle von gleichnamigen Modulen im Bibliotheksverzeichnis geladen werden. Dies ist ein Fehler, es sei denn, der Austausch ist beabsichtigt. Weitere Informationen finden Sie im Abschnitt `Standardmodule`.

6.1.3. “Kompilierte” Python Dateien

Um das Laden von Modulen zu beschleunigen, speichert Python die kompilierte Version jedes Moduls im Verzeichnis `__pycache__` unter dem Namen `module.version.pyc`, wo die Version das Format der kompilierten Datei kodiert; sie enthält in der Regel die Python-Versionsnummer. In CPython Release 3.3 beispielsweise würde die kompilierte Version von `spam.py` als `__pycache__/spam.cpython-33.pyc` zwischengespeichert. Diese Namenskonvention ermöglicht es, kompilierte Module aus verschiedenen Releases und verschiedenen Versionen von Python nebeneinander zu existieren.

Python überprüft das Änderungsdatum des Quellcodes anhand der kompilierten Version, um festzustellen, ob er veraltet ist und neu kompiliert werden muss. Dies ist ein vollautomatischer





Prozess. Außerdem sind die kompilierten Module plattformunabhängig, so dass die gleiche Bibliothek für Systeme mit unterschiedlichen Architekturen gemeinsam genutzt werden kann. Python überprüft den Cache nicht in zwei Fällen. Erstens wird das Ergebnis für das Modul, das direkt von der Kommandozeile geladen wird, immer neu kompiliert und nicht gespeichert. Zweitens wird der Cache nicht überprüft, wenn es kein Quellmodul gibt. Um eine Nicht-Source-Distribution (nur kompiliert) zu unterstützen, muss sich das kompilierte Modul im Quellverzeichnis befinden, und es darf kein Quellmodul vorhanden sein.

Einige Tipps für Experten:

- Du kannst die Switches `-O` oder `-OO` der Python Befehle verwenden, um die Größe eines kompilierten Moduls zu reduzieren. Der Switch `-O` entfernt Assert-Anweisungen, der Switch `-OO` entfernt sowohl Assert-Anweisungen als auch `__doc__` Zeichenketten. Da einige Programme möglicherweise darauf angewiesen sind, dass diese verfügbar sind, solltest du diese Option nur verwenden, wenn du weißt, was du tust. "Optimierte" Module haben einen `opt`-Tag und sind in der Regel kleiner. Zukünftige Releases können die Auswirkungen der Optimierung verändern.
- Ein Programm läuft nicht schneller, wenn es aus einer `.pyc`-Datei gelesen wird, als wenn es aus einer `.py`-Datei gelesen wird; das Einzige, was bei `.pyc`-Dateien schneller ist, ist die Geschwindigkeit, mit der sie geladen werden.
- Das Modul `compileall` kann `.pyc`-Dateien für alle Module in einem Verzeichnis erstellen.
- Es gibt mehr Details zu diesem Prozess, einschließlich eines Flussdiagramms der Entscheidungen, in PEP 3147.

6.2. Standardmodule

Python wird mit einer Bibliothek von Standardmodulen geliefert, die in einem separaten Dokument, der Python Library Reference ("Library Reference" im Folgenden), beschrieben werden. Einige Module sind im Interpreter integriert; diese ermöglichen den Zugriff auf Operationen, die nicht zum Kern der Sprache gehören, aber dennoch eingebaut sind, entweder aus Effizienzgründen oder um Zugang zu Betriebssystem-Grundlagen wie Systemaufrufen zu ermöglichen. Das Set solcher Module ist eine Konfigurationsoption, die auch von der zugrunde liegenden Plattform abhängt. So wird beispielsweise das `winreg`-Modul nur auf Windows-Systemen angeboten. Ein besonderes Modul verdient besondere Aufmerksamkeit: `sys`, das in jeden Python-Interpreter integriert ist. Die Variablen `sys.ps1` und `sys.ps2` definieren die Strings, die als primäre und sekundäre Anweisungen verwendet werden:



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

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


Die Variable `sys.path` ist eine Liste von Zeichenketten, die den Suchpfad des Interpreters für Module bestimmt. Es wird auf einen Standardpfad aus der Umgebungsvariablen `PYTHONPATH` oder von einem eingebauten Standard initialisiert, wenn `PYTHONPATH` nicht gesetzt ist. Du kannst dies mit Hilfe von Standardlistenoperationen modifizieren:

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

6.3. Die `dir()` Funktion

Die eingebaute Funktion `dir()` wird verwendet, um herauszufinden, welche Namen ein Modul definiert. Es wird eine sortierte Liste von Strings zurückgegeben:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
'__package__', '__stderr__', '__stdin__', '__stdout__',
'_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
'_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
```



```
'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
'thread_info', 'version', 'version_info', 'warnoptions']
```

Ohne Argumente listet `dir()` die Namen auf, die du aktuell definiert hast:

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

Bitte beachte, dass alle Arten von Namen aufgelistet werden: Variablen, Module, Funktionen, etc.

`dir()` listet die Namen der eingebauten Funktionen und Variablen nicht auf. Wenn du eine Liste dieser Module haben willst, sind sie in den Standardmodulen definiert:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
```



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

6.4. Packages

Packages sind eine Möglichkeit, den Modulnamensraum von Python durch die Verwendung von "dotted module names" zu strukturieren. So bezeichnet beispielsweise der Modulname *A.B* ein Submodul namens *B* in einem Package namens *A*. Genauso wie die Verwendung von Modulen die Autoren verschiedener Module davor bewahrt, sich um die globalen Variablennamen des jeweils anderen kümmern zu müssen, erspart die Verwendung von Modulnamen mit Punkten den Autoren von Multi-Modulpaketen wie NumPy oder Pillow die Notwendigkeit, sich um die Modulnamen des anderen zu kümmern.

Angenommen, du möchtest eine Sammlung von Modulen (ein "Package") für die einheitliche Handhabung von Audiodateien und Audiodaten entwerfen. Es gibt viele verschiedene Sound-Dateiformate (normalerweise an ihrer Erweiterung erkannt, z.B.: *.wav*, *.aiff*, *.au*), sodass du möglicherweise eine wachsende Sammlung von Modulen für die Konvertierung zwischen den verschiedenen Dateiformaten erstellen und pflegen musst. Es gibt auch viele verschiedene Operationen, die du mit Audiodaten durchführen möchtest (z.B. Mischen, Hinzufügen von Echo, Anwenden einer Equalizer-Funktion, Erzeugen eines künstlichen Stereoeffekts), sodass du zusätzlich einen endlosen Strom von Modulen schreiben wirst, um diese Operationen durchzuführen. Hier ist eine mögliche Struktur für dein Package (ausgedrückt in Form eines hierarchischen Dateisystems):



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

Beim Importieren des Package durchsucht Python die Verzeichnisse auf *sys.path* nach dem Unterverzeichnis package.

Die `__init__.py`-Dateien sind erforderlich, damit Python die Verzeichnisse, welche die Datei enthalten, als Packages behandelt. Dies geschieht, um zu verhindern, dass Verzeichnisse mit einem gewöhnlichen Namen, wie z.B. *string*, versehentlich gültige Module verstecken, die später auf dem Modulsuchpfad auftreten. Im einfachsten Fall kann `__init__.py` nur eine leere Datei sein, aber es kann auch Initialisierungscode für das Package ausführen oder die später beschriebene Variable `__all__` setzen.

Benutzer des Packae können z.B. einzelne Module aus dem Package importieren:

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





Dadurch wird das Submodul `sound.effects.echo` geladen. Er muss mit seinem vollständigen Namen referenziert werden.

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

Eine alternative Möglichkeit, das Submodul zu importieren, ist:

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

Dadurch wird auch das Submodul `echo` geladen und ohne sein Packagepräfix verfügbar gemacht, sodass es wie folgt verwendet werden kann:

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

Eine weitere Variante ist der direkte Import der gewünschten Funktion oder Variable:

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

Auch hier wird das Submodul `echo` geladen, aber dadurch ist seine Funktion `echofilter()` direkt verfügbar:

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

Beachte, dass bei der Verwendung von `from package import item` das Item entweder ein Submodul (oder ein Subpackage) des Package sein kann, oder ein anderer im Package definierter Name, wie eine Funktion, Klasse oder Variable. Die Import-Anweisung testet zunächst, ob das Element im Package definiert ist; wenn nicht, nimmt sie an, dass es sich um ein Modul handelt und versucht, es zu laden. Wenn es ihn nicht findet, wird eine `ImportError`-Ausnahme ausgelöst.

Im Gegensatz dazu muss bei der Verwendung von Syntax wie `import item.subitem.subsubitem` jedes Element mit Ausnahme des letzten ein Package sein; das letzte Element kann ein Modul





oder ein Package sein, darf aber keine Klasse oder Funktion oder Variable sein, die im vorherigen Element definiert ist.

6.4.1. Importieren * aus einem Paket

Was passiert nun, wenn der Benutzer *from sound.effects import ** schreibt? Im Idealfall sollte man hoffen, dass dies irgendwie an das Dateisystem geht, herausfindet, welche Submodule im Paket vorhanden sind, und sie alle importiert. Dies kann lange dauern und der Import von Submodulen kann unerwünschte Nebenwirkungen haben, die nur auftreten sollten, wenn das Submodul explizit importiert wird.


Die einzige Lösung besteht darin, dass der Autor des Package einen expliziten Index des Package bereitstellt. Die Import-Anweisung verwendet die folgende Konvention: Wenn der `__init__.py`-Code eines Package eine Liste mit dem Namen `__all__` definiert, wird davon ausgegangen, dass es sich um die Liste der Modulnamen handelt, die importiert werden sollen, wenn beim Packageimport `*` auftritt. Es liegt an dem Packageautor, diese Liste auf dem neuesten Stand zu halten, wenn eine neue Version des Package veröffentlicht wird. Packageautoren können sich auch entscheiden, es nicht zu unterstützen, wenn sie keine Verwendung für den Import von `*` aus ihrem Package sehen. Beispielsweise könnte die Datei `sound/effects/__init__.py` den folgenden Code enthalten:

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

Dies würde bedeuten, dass *from sound.effects import ** die drei genannten Submodule des Soundpackages importieren würde.

Wenn `__all__` nicht definiert ist, importiert die Anweisung *from sound.effects import ** nicht alle Submodule aus dem Package `sound.effects` in den aktuellen Namensraum; sie stellt nur sicher, dass das Package `sound.effects` importiert wurde (möglicherweise mit einem Initialisierungscode in `__init__.py`) und importiert dann die im Package definierten Namen. Dazu gehören alle Namen, die von `__init__.py` definiert wurden (und explizit geladene Submodule). Es enthält auch alle Submodule des Package, die explizit durch frühere Importanweisungen geladen wurden. Schau dir diesen Code an:





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

In diesem Beispiel werden die Module `echo` und `surround` im aktuellen Namensraum importiert, da sie im `sound.effects`-Paket definiert sind, wenn die *from...import*-Anweisung ausgeführt wird. (Dies funktioniert auch, wenn `__all__` definiert ist.)

Obwohl bestimmte Module so konzipiert sind, dass sie nur Namen exportieren, die bestimmten Mustern folgen, wenn du *import ** verwendest, gilt dies immer noch als schlechte Praxis im Produktionscode.

Denke daran, dass nichts Falsches an der Verwendung von *from package import specific_submodule* ist! Dies ist in der Tat die empfohlene Schreibweise, es sei denn, das importierende Modul muss gleichnamige Submodule aus verschiedenen Packages verwenden.

6.4.2. Referenzen innerhalb von Packages

Wenn Packages in Unterpackages strukturiert sind (wie im Beispiel das Package `sound`), kannst du mit absoluten Importen auf Submodule von Geschwisterpackages verweisen. Wenn das Modul `sound.filters.vocoder` beispielsweise das Modul `echo` im `sound.effects`-Paket verwenden muss, kann es *from sound.effects import echo* verwenden.

Du kannst auch relative Importe schreiben, mit der Form *from module import name* der Importanweisung. Diese Importe verwenden führende Punkte, um das aktuelle und übergeordnete Package anzuzeigen, das am relativen Import beteiligt ist. Aus dem Modul `surround` kannst du zum Beispiel verwenden:

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

Bitte bedenke, dass relative Importe auf dem Namen des aktuellen Moduls basieren. Da der Name des Hauptmoduls immer `"__main__"` lautet, müssen Module, die als Hauptmodul einer Python-Anwendung verwendet werden sollen, immer absolute Importe verwenden.



6.4.3. Packages in mehreren Verzeichnissen

Packages unterstützen ein weiteres spezielles Attribut, `__path__`. Dies wird als Liste initialisiert, die den Namen des Verzeichnisses enthält, das die `__init__.py` des Package enthält, bevor der Code in dieser Datei ausgeführt wird. Diese Variable kann geändert werden; dies betrifft zukünftige Suchen nach Modulen und Unterpackages, die im Package enthalten sind.

Obwohl diese Funktion nicht oft benötigt wird, kann sie verwendet werden, um die Menge der in einem Package enthaltenen Module zu erweitern.

Fußnoten

- [1] Tatsächlich sind Funktionsdefinitionen auch "Anweisungen", die "ausgeführt" werden; die Ausführung einer Funktionsdefinition auf Modulebene trägt den Funktionsnamen in die globale Symboltabelle des Moduls ein.

7. Input und Output

Es gibt mehrere Möglichkeiten, die Ausgabe eines Programms darzustellen; Daten können in menschenlesbarer Form gedruckt oder zur späteren Verwendung in eine Datei geschrieben werden. In diesem Kapitel werden einige der Möglichkeiten diskutiert.

7.1. Raffiniertere Ausgabeformatierung

Bisher haben wir zwei Möglichkeiten gefunden, Werte zu schreiben: Ausdrucksanweisungen und die Funktion `print()`. (Eine dritte Möglichkeit ist die Verwendung der `write()`-Methode von Dateiobjekten; die Standardausgabedatei kann als `sys.stdout` bezeichnet werden. Weitere Informationen hierzu findest du in der Library Referenz).


Oftmals wünschst du dir mehr Kontrolle über die Formatierung deiner Ausgabe, als nur das Drucken von durch Leerzeichen getrennten Werten. Es gibt mehrere Möglichkeiten, die Ausgabe zu formatieren.

Um formatierte Strings zu verwenden, beginne einen String mit *f* oder *F* vor dem öffnenden Anführungszeichen oder dem dreifachen Anführungszeichen. Innerhalb dieses Strings kannst du einen Python-Ausdruck zwischen { und } Zeichen schreiben, der sich auf Variablen oder literale Werte beziehen kann.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- Die `str.format()`-Methode von Strings erfordert einen höheren manuellen Aufwand. Du wirst weiterhin { und } verwenden, um zu markieren, wo eine Variable ersetzt wird, und kannst detaillierte Formatierungsanweisungen bereitstellen, aber du musst auch die zu formatierenden Informationen angeben.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
```



```
' 42572654 YES votes 49.67%'
```

- Schließlich kannst du die gesamte Stringbehandlung selbst durchführen, indem du Stringlicing- und Verkettungsoperationen verwendest, um jedes denkbare Layout zu erstellen. Der Stringtyp enthält einige Methoden, die nützliche Operationen zum Auffüllen von Strings auf eine bestimmte Spaltenbreite ausführen.

Wenn du keine ausgefallene Ausgabe brauchst, sondern nur eine schnelle Anzeige einiger Variablen zu Debugging-Zwecken willst, kannst du jeden Wert mit den Funktionen `repr()` oder `str()` in einen String konvertieren.

Die Funktion `str()` soll Darstellungen von Werten zurückgeben, die relativ menschenlesbar sind, während `repr()` Darstellungen erzeugen soll, die vom Interpreter gelesen werden können (oder einen `SyntaxError` erzwingen, wenn es keine gleichwertige Syntax gibt). Für Objekte, die keine besondere Darstellung für das menschliche Auge haben, gibt `str()` den gleichen Wert wie `repr()` zurück. Viele Werte, wie Zahlen oder Strukturen wie Listen und Dictionaries, haben die gleiche Darstellung mit beiden Funktionen. Insbesondere Strings haben zwei unterschiedliche Darstellungen.

Einige Beispiele:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # repr() eines Strings fügt Anführungszeichen und Backslashes hinzu:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
```

```
... repr((x, y, ('spam', 'eggs')))\n\"(32.5, 40000, ('spam', 'eggs'))\"
```

Das Modul `string` enthält eine Template-Klasse, die eine weitere Möglichkeit bietet, Werte in Strings zu ersetzen, indem Platzhalter wie `$x` verwendet werden und diese durch Werte aus einem Dictionary ersetzt werden, aber dies bietet eine wesentlich geringere Kontrolle über die Formatierung.

7.1.1. Formatierte Strings

Formatierte Strings (kurz auch f-Strings genannt) ermöglichen es dir, den Wert von Python-Ausdrücken in einem String aufzunehmen, indem du dem String `f` oder `F` voranstellst und Ausdrücke als `{expression}` schreibst.

Ein optionaler Formatbezeichner kann dem Ausdruck folgen. Dies ermöglicht eine größere Kontrolle darüber, wie der Wert formatiert wird. Das folgende Beispiel rundet `pi` auf drei Nachkommastellen auf:


```
>>> import math\n>>> print(f'The value of pi is approximately {math.pi:.3f}.')\nThe value of pi is approximately 3.142.
```

Die Übergabe einer Ganzzahl nach dem `:` bewirkt, dass dieses Feld eine Mindestanzahl von Zeichen breit ist. Dies ist nützlich, um die Spalten in eine Linie zu bringen.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}\n>>> for name, phone in table.items():\n...     print(f'{name:10} ==> {phone:10d}')
```

Sjoerd	==>	4127
Jack	==>	4098
Dcab	==>	7678

Andere Modifikatoren können verwendet werden, um den Wert zu konvertieren, bevor er formatiert wird. `'a'` wendet `ascii()` an, `'s'` verwendet `str()` und `'r'` verwendet `repr()`:



```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

Eine Referenz zu diesen Formatvorgaben befindet sich im Referenzleitfaden für die Formatvorgabe Mini-Language.

7.1.2. Die String format() Methode

Die grundlegende Verwendung der Methode str.format() sieht wie folgt aus:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

Die Klammern und Zeichen in ihnen (sogenannte Formatfelder) werden durch die Objekte ersetzt, die an die Methode str.format() übergeben werden. Eine Zahl in den Klammern kann verwendet werden, um die Position des Objekts zu bezeichnen, das an die str.format()-Methode übergeben wird.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Wenn Schlüsselwortargumente in der str.format()-Methode verwendet werden, werden ihre Werte unter Verwendung des Namens des Arguments referenziert.

```
>>> print('This {food} is {adjective}'.format(
...     food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Positions- und Schlüsselwortargumente können beliebig kombiniert werden:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                    other='Georg'))
The story of Bill, Manfred, and Georg.
```

Wenn du eine wirklich lange Formatkette hast, die du nicht aufteilen willst, wäre es schön, wenn du die zu formatierenden Variablen nach Namen und nicht nach Position referenzieren könntest. Dies geschieht durch einfaches Übergeben des Dictionary und die Verwendung eckiger Klammern '[' für den Zugriff auf die Schlüssel.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```


Dies kann auch dadurch geschehen, dass die Tabelle als Schlüsselwortargumente mit der Notation '**' übergeben wird.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Dies ist besonders nützlich in Kombination mit der integrierten Funktion vars(), die ein Dictionary mit allen lokalen Variablen zurückgibt.

Als Beispiel ergeben die folgenden Zeilen eine geordnete Menge von Spalten, die ganze Zahlen und deren Quadrate und Kubikzahlen enthalten:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1  1  1
2  4  8
3  9 27
4 16 64
5 25 125
```

6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Einen vollständigen Überblick über die Zeichenkettenformatierung mit `str.format()` findest du unter Format String Syntax.

7.1.3. Manuelle Zeichenkettenformatierung

Hier ist die gleiche Tabelle von Quadraten und Kubikzahlen, die manuell formatiert wurden:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Beachte die Verwendung von 'end' in der vorigen Zeile
...     print(repr(x*x*x).rjust(4))
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```

(Beachte, dass das eine Leerzeichen zwischen jeder Spalte durch die Funktionsweise von `print()` hinzugefügt wurde: Es werden immer Leerzeichen zwischen seinen Argumenten hinzugefügt.) Die `str.rjust()`-Methode von Zeichenkettenobjekten richtet einen String in einem Feld mit einer bestimmten Breite rechtsbündig aus, indem sie sie mit Leerzeichen auf der linken Seite füllt. Es gibt ähnliche Methoden `str.ljust()` und `str.center()`. Diese Methoden schreiben nichts, sie geben nur eine neue Zeichenkette zurück. Wenn der Eingabestring zu lang ist, wird er nicht abgeschnitten, sondern unverändert zurückgegeben; dies wird dein Spaltenlayout durcheinanderbringen, aber das ist normalerweise besser als die Alternative, die wäre, bei



einem Wert zu lügen. (Wenn du wirklich abschneiden willst, kannst du immer eine Slice-Operation hinzufügen, wie in `x.ljust(n)[:n]`.)

Es gibt eine weitere Methode, `str.zfill()`, die eine numerische Zeichenkette auf der linken Seite mit Nullen füllt. Es versteht die Plus- und Minuszeichen:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4. Alte Stringformatierung

Der %-Operator kann auch für die Stringformatierung verwendet werden. Er interpretiert das linke Argument ähnlich wie einen Formatstring im `sprintf()`-Stil, der auf das rechte Argument angewendet wird, und gibt den String zurück, der aus dieser Formatierungsoperation resultiert. Zum Beispiel:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

Weitere Informationen hierzu findest du im Abschnitt Stringformatierung im `printf`-Stil.


7.2. Lesen und Schreiben von Dateien

`open()` gibt ein Dateiojekt zurück und wird am häufigsten mit zwei Argumenten verwendet: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
```

Das erste Argument ist ein String, der den Dateinamen enthält. Das zweite Argument ist ein weiterer String, der einige Zeichen enthält, die die Art und Weise beschreiben, wie die Datei verwendet wird. Der Modus kann `'r'` sein, wenn die Datei nur gelesen wird, `'w'`, wenn in ihr nur geschrieben wird (eine vorhandene Datei mit dem gleichen Namen wird gelöscht), und `'a'` öffnet





die Datei zum Anhängen; alle in die Datei geschriebenen Daten werden automatisch an das Ende hinzugefügt. 'r+' öffnet die Datei sowohl zum Lesen als auch zum Schreiben. Das Modusargument ist optional; 'r' wird angenommen, wenn es weggelassen wird.

Normalerweise werden Dateien im Textmodus geöffnet, d.h. du liest und schreibst Strings von und in die Datei, die in einer bestimmten Kodierung kodiert sind. Wenn keine Kodierung angegeben ist, ist der Standard plattformabhängig (siehe `open()`). 'b' an den Modus angehängt öffnet die Datei im Binärmodus: Jetzt werden die Daten in Form von Bytes-Objekten gelesen und geschrieben. Dieser Modus sollte für alle Dateien verwendet werden, die keinen Text enthalten.

Im Textmodus ist die Standardeinstellung beim Lesen die Konvertierung plattformspezifischer Zeilenenden (`\n` unter Unix, `\r\n` unter Windows) zu nur `\n`. Beim Schreiben im Textmodus wird standardmäßig das Auftreten von `\n` zurück in plattformspezifische Zeilenenden umgewandelt. Diese Hinter-den-Szenen-Änderung an Dateidaten ist für Textdateien in Ordnung, wird aber Binärdaten wie die in JPEG- oder EXE-Dateien beschädigen. Achte darauf, dass du beim Lesen und Schreiben solcher Dateien den Binärmodus verwendest.

Es ist ratsam, das Schlüsselwort `with` zu verwenden, wenn es um Dateiobjekte geht. Der Vorteil ist, dass die Datei nach Beendigung ihrer Suite ordnungsgemäß geschlossen wird, auch wenn irgendwann eine Ausnahme ausgelöst wird. Die Verwendung von `with` ist ebenfalls viel kürzer als das Schreiben von äquivalenten Try-Final-Blöcken:

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

Wenn du das `with`-Schlüsselwort nicht verwendest, dann solltest du `f.close()` aufrufen, um die Datei zu schließen und sofort alle von ihr verwendeten Systemressourcen freizugeben. Wenn du eine Datei nicht explizit schließt, zerstört Pythons Garbage Collector schließlich das Objekt und schließt die geöffnete Datei für dich, aber die Datei kann für eine Weile offenbleiben. Ein weiteres Risiko besteht darin, dass verschiedene Python-Implementierungen diese Bereinigung zu unterschiedlichen Zeiten durchführen.

Nachdem ein Dateiobjekt geschlossen wurde, entweder durch eine `with`-Anweisung oder durch den Aufruf von `f.close()`, schlagen Versuche, das Dateiobjekt zu verwenden, automatisch fehl.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1. Methoden von Dateiobjekten

Die restlichen Beispiele in diesem Abschnitt gehen davon aus, dass bereits ein Dateiobjekt namens *f* erstellt wurde.

Um den Inhalt einer Datei zu lesen, rufst du *f.read(size)* auf, das eine gewisse Menge an Daten liest und sie als Zeichenkette (im Textmodus) oder Bytes-Objekt (im Binärmodus) zurückgibt. *size* ist ein optionales numerisches Argument. Wenn *size* weggelassen wird oder negativ ist, wird der gesamte Inhalt der Datei gelesen und zurückgegeben; es ist dein Problem, wenn die Datei doppelt so groß ist wie der Speicher deines Computers. Andernfalls werden höchstens *size* Bytes gelesen und zurückgegeben. Wenn das Ende der Datei erreicht ist, gibt *f.read()* eine leere Zeichenkette ('') zurück.

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline() liest eine einzelne Zeile aus der Datei; ein Zeilenumbruchzeichen (*\n*) bleibt am Ende der Zeichenkette stehen und wird nur in der letzten Zeile der Datei weggelassen, wenn die Datei nicht in einer Zeilenumbruch endet. Dadurch wird der Rückgabewert eindeutig; wenn *f.readline()* eine leere Zeichenkette zurückgibt, ist das Ende der Datei erreicht, während eine leere Zeile durch *\n* dargestellt wird, eine Zeichenkette, die nur eine einzige Zeilenumbruch enthält.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```



Um Zeilen aus einer Datei zu lesen, kannst du die Schleife über das Dateiojekt ziehen. Dies ist speichereffizient, schnell und führt zu einfachem Code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

Wenn du alle Zeilen einer Datei in einer Liste lesen möchtest, kannst du auch `list(f)` oder `f.readlines()` verwenden.

`f.write(string)` schreibt den Inhalt von `string` in die Datei und gibt die Anzahl der geschriebenen Zeichen zurück.

```
>>> f.write('This is a test\n')
15
```

Andere Arten von Objekten müssen konvertiert werden - entweder in einen String (im Textmodus) oder in ein Bytes-Objekt (im Binärmodus) - bevor sie geschrieben werden:

```
>>> value = ('the answer', 42)
>>> s = str(value) # Konvertiere das Tupel in einen String
>>> f.write(s)
18
```

`f.tell()` gibt eine Ganzzahl zurück, die die aktuelle Position des Dateiojekts in der Datei angibt, die als Anzahl der Bytes ab Beginn der Datei im Binärmodus und eine unklare Zahl im Textmodus dargestellt wird.

Um die Position des Dateiojekts zu ändern, verwendest du `f.seek(offset, from_what)`. Die Position wird berechnet, indem ein Offset zu einem Referenzpunkt hinzugefügt wird; der Referenzpunkt wird durch das Argument `from_what` ausgewählt. Ein `from_what`-Wert von 0 misst vom Anfang der Datei, 1 verwendet die aktuelle Dateiposition und 2 verwendet das Ende





der Datei als Bezugspunkt. *from_what* kann weggelassen werden und ist standardmäßig auf 0 gesetzt, wobei der Anfang der Datei als Bezugspunkt verwendet wird.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)          # Geh zum sechsten Byte in der Datei
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)     # Geh zum dritten Byte vor dem Ende
13
>>> f.read(1)
b'd'
```

In Textdateien (die ohne ein *b* im Modusstring geöffnet sind) sind nur Suchvorgänge relativ zum Anfang der Datei erlaubt (die Ausnahme ist das Suchen bis zum Dateende mit `seek(0, 2)`) und die einzigen gültigen Offsetwerte sind die von *f.tell()* zurückgegebenen oder Null. Jeder andere Offsetwert erzeugt ein undefiniertes Verhalten.


Dateiobjekte haben einige zusätzliche Methoden, wie *isatty()* und *truncate()*, die seltener verwendet werden; lies die Library Reference für eine vollständige Anleitung zu Dateiobjekten.

7.2.2. Speichern von strukturierten Daten mit json

Strings können einfach in eine Datei geschrieben und aus ihr gelesen werden. Zahlen erfordern etwas mehr Aufwand, da die *read()*-Methode nur Strings zurückgibt, die an eine Funktion wie *int()* übergeben werden müssen, die einen String wie '123' nimmt und ihren Zahlenwert 123 zurückgibt. Wenn du komplexere Datentypen wie verschachtelte Listen und Dictionaries speichern möchtest, wird das Parsen und Serialisieren von Hand kompliziert.

Anstatt dass Benutzer ständig Code schreiben und debuggen, um komplizierte Datentypen in Dateien zu speichern, ermöglicht Python die Verwendung des beliebten Datenaustauschformats JSON (JavaScript Object Notation). Das Standardmodul namens *json* kann Python-Datenhierarchien nehmen und in Stringdarstellungen konvertieren; dieser Prozess wird Serialisierung genannt. Die Rekonstruktion der Daten aus der Stringdarstellung wird als Deserialisierung bezeichnet. Zwischen Serialisierung und Deserialisierung kann der String, der





das Objekt repräsentiert, in einer Datei oder einem Datenspeicher gespeichert oder über eine Netzwerkverbindung an einen entfernten Rechner gesendet worden sein.

Hinweis

Das JSON-Format wird von modernen Anwendungen häufig verwendet, um den Datenaustausch zu ermöglichen. Viele Programmierer sind bereits damit vertraut, was es zu einer guten Wahl für die Interoperabilität macht.

Wenn du ein Objekt *x* hast, kannst du seine JSON-String-Darstellung mit einer einfachen Codezeile anzeigen lassen:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Eine weitere Variante der `dumps()` Funktion, genannt `dump()`, serialisiert das Objekt einfach in eine Textdatei. Wenn *f* also ein Textdatei-Objekt ist, das zum Schreiben geöffnet ist, können wir dies tun:


```
json.dump(x, f)
```

Um das Objekt wieder zu dekodieren, wenn *f* ein Textdateiobjekt ist, das zum Lesen geöffnet wurde:

```
x = json.load(f)
```

Diese einfache Serialisierungstechnik kann mit Listen und Dictionaries umgehen, aber die Serialisierung beliebiger Klasseninstanzen in JSON erfordert etwas zusätzlichen Aufwand. Die Referenz für das `json`-Modul enthält eine Erklärung dazu.

Siehe auch



Pickle - das Pickle-Modul

Im Gegensatz zu JSON ist Pickle ein Protokoll, das die Serialisierung von beliebig komplexen Python-Objekten ermöglicht. Daher ist es spezifisch für Python und kann nicht zur Kommunikation mit Anwendungen verwendet werden, die in anderen Sprachen geschrieben wurden. Standardmäßig ist es auch unsicher: Die Deserialisierung von Pickle-Daten aus einer nicht vertrauenswürdigen Quelle kann beliebigen Code ausführen, wenn die Daten von einem erfahrenen Angreifer erstellt wurden.



8. Fehler und Ausnahmen

Bisher wurden Fehlermeldungen nicht mehr als erwähnt, aber wenn du die Beispiele ausprobiert hast, hast du wahrscheinlich einige gesehen. Es gibt (mindestens) zwei Arten von Fehlern: Syntaxfehler und Ausnahmen.

8.1. Syntaxfehler

Syntaxfehler, auch bekannt als Parsing-Fehler, sind vielleicht die häufigste Art von Fehlern, die du bekommst, während du noch Python lernst:


```
>>> while True print('Hello world')
      File "<stdin>", line 1
        while True print('Hello world')
                        ^
SyntaxError: invalid syntax
```

Der Parser wiederholt die fehlerhafte Zeile und zeigt einen kleinen "Pfeil" an, der auf den ersten Punkt in der Zeile zeigt, in der der Fehler erkannt wurde. Der Fehler wird durch das Token vor dem Pfeil verursacht (oder zumindest erkannt): Im Beispiel wird der Fehler bei der Funktion `print()` erkannt, da davor ein Doppelpunkt (`:`) fehlt. Dateiname und Zeilennummer werden gedruckt, so dass du weißt, wo du suchen musst, falls die Eingabe von einem Skript stammt.

8.2. Exceptions

Selbst wenn eine Anweisung oder ein Ausdruck syntaktisch korrekt ist, kann sie beim Versuch, sie auszuführen, einen Fehler verursachen. Fehler, die bei der Ausführung festgestellt werden, werden als Exceptions bezeichnet und sind nicht unbedingt tödlich: Du wirst bald lernen, wie man sie in Python-Programmen behandelt. Die meisten Exceptions werden jedoch nicht von Programmen behandelt und führen, wie hier gezeigt, zu Fehlermeldungen:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
```



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Die letzte Zeile der Fehlermeldung zeigt an, was passiert ist. Exceptions gibt es in verschiedenen Typen, und der Typ wird als Teil der Nachricht ausgegeben: Die Typen im Beispiel sind `ZeroDivisionError`, `NameError` und `TypeError`. Der als Exceptiontyp ausgegebene String ist der Name der eingebauten Exception, die aufgetreten ist. Dies gilt für alle eingebauten Exceptions, muss aber nicht für benutzerdefinierte Exceptions gelten (obwohl es eine nützliche Konvention ist). Standard-Exceptionnamen sind eingebaute Identifikatoren (keine reservierten Schlüsselwörter).

Der Rest der Zeile liefert Details, basierend auf der Art der Exception und der Ursache.

Der vorhergehende Teil der Fehlermeldung zeigt den Kontext, in dem die Exception aufgetreten ist, in Form eines Stack-Tracebacks. Im Allgemeinen enthält es einen Stack-Traceback, der die Quellzeilen auflistet, jedoch keine Zeilen anzeigt, die von der Standardeingabe gelesen wurden.

8.3. Behandlung von Exceptions

Es ist möglich, Programme zu schreiben, die ausgewählte Exceptions behandeln. Sieh dir das folgende Beispiel an, das den Benutzer nach Eingaben fragt, bis eine gültige Ganzzahl eingegeben wurde, aber es dem Benutzer erlaubt, das Programm zu unterbrechen (mit Strg-C, oder was auch immer das Betriebssystem unterstützt); beachte, dass eine benutzergenerierte Unterbrechung signalisiert wird, indem die `KeyboardInterrupt`-Exception ausgelöst wird.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops! That was no valid number. Try again...")
... 
```



Die Try-Anweisung funktioniert wie folgt.

- Zuerst wird die Try-Anweisung (die Anweisung(en) zwischen den Schlüsselwörtern try und except) ausgeführt.
- Wenn keine Exception auftritt, wird die except-Anweisung übersprungen und die Ausführung der Try-Anweisung beendet.
- Wenn während der Ausführung der Try-Anweisung eine Ausnahme auftritt, wird der Rest der Anweisung übersprungen. Wenn sein Typ dann mit der nach dem Schlüsselwort except genannten Exception übereinstimmt, wird die except-Anweisung ausgeführt, und die Ausführung wird dann nach der Try-Anweisung fortgesetzt.
- Wenn eine Exception auftritt, die nicht mit der in der except-Anweisung genannten Exception übereinstimmt, wird sie an äußere Try-Anweisungen weitergegeben; wenn keine Behandlung gefunden wird, handelt es sich um eine unbehandelte Exception und die Ausführung stoppt mit einer Meldung, wie oben gezeigt.

Eine Try-Anweisung kann mehr als eine except-Anweisung haben, um Behandlungen für verschiedene Ausnahmen anzugeben. Es wird höchstens eine Handlung ausgeführt. Handlungen behandeln nur Exceptions, die in der entsprechenden Try-Anweisung auftreten, nicht in anderen Handlungen der gleichen Try-Anweisung. Eine except-Anweisung kann z.B. mehrere Exception als Klammer-Tupel benennen:

```
... except (RuntimeError, TypeError, NameError):  
...     pass
```

Eine Klasse in einer except-Anweisung ist mit einer Exception kompatibel, wenn sie die gleiche Klasse oder eine Basisklasse davon ist (aber nicht umgekehrt - eine except-Anweisung, die eine abgeleitete Klasse auflistet, ist nicht mit einer Basisklasse kompatibel). Der folgende Code druckt beispielsweise B, C, D in dieser Reihenfolge:



```

class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")

```

Es ist zu beachten, dass, wenn die `except`-Anweisungen umgekehrt ausgeführt würden (mit `except B` zuerst), sie `B, B, B, B` gedruckt hätten - die erste Übereinstimmung mit `except`-Anweisungen wird ausgelöst.


Die letzte `except`-Anweisung kann den/die Namen der Exception(s) weglassen, um als Platzhalter zu dienen. Benutze dies mit äußerster Vorsicht, da es leicht ist, einen echten Programmierfehler auf diese Weise zu maskieren! Es kann auch verwendet werden, um eine Fehlermeldung zu drucken und dann die Exception erneut zu werfen (sodass ein Aufrufer die Exception auch behandeln kann):

```

import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:

```



```
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Die Try except-Anweisung hat eine optionale else-Anweisung, die, wenn sie vorhanden ist, allen except-Anweisungen folgen muss. Es ist nützlich für Code, der ausgeführt werden muss, wenn die Try-Anweisung keine Exception auslöst. Zum Beispiel:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Die Verwendung der else-Anweisung ist besser als das Hinzufügen von zusätzlichem Code zur try-Anweisung, da sie verhindert, dass versehentlich eine Exception abgefangen wird, die nicht dadurch ausgelöst wurde, dass der Code durch die try except-Anweisung geschützt ist.

Wenn eine Exception auftritt, kann sie einen zugehörigen Wert haben, der auch als Argument der Exception bekannt ist. Das Vorhandensein und der Typ des Arguments hängen vom Exceptiontyp ab.

Die Exception-Anweisung kann eine Variable nach dem Namen der Exception angeben. Die Variable ist an eine Exceptioninstanz mit den in *instance.args* gespeicherten Argumenten gebunden. Der Einfachheit halber definiert die Ausnahmeinstanz *__str__()*, sodass die Argumente direkt gedruckt werden können, ohne auf *.args* verweisen zu müssen. Man kann eine Exception auch zuerst instanziiieren, bevor man sie auslöst und ihr beliebige Attribute hinzufügt.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # die Exception-Instanz
```



```

...     print(inst.args)      # Argumente gespeichert in .args
...     print(inst)          # __str__ erlaubt direkte Ausgabe von args,
...                           # aber kann in Unterklassen von Exception
überschrieben werden.
...     x, y = inst.args      # args entpacken
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

Wenn eine Exception Argumente hat, werden diese als letzter Teil ('Detail') der Nachricht für unbehandelte Exceptions ausgegeben.

ExceptionHandler behandeln Exceptions nicht nur, wenn sie sofort in der Try-Anweisung auftreten, sondern auch, wenn sie innerhalb von Funktionen auftreten, die (auch indirekt) in der Try-Anweisung aufgerufen werden. Zum Beispiel:

```

>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero

```


8.4. Auslösen von Exceptions

Die raise-Anweisung ermöglicht es dem Programmierer, das Auftreten einer bestimmten Exception zu erzwingen. Zum Beispiel:

```

>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

```



```
NameError: HiThere
```

Das einzige Argument von `raise` ist die geforderte Exception. Dies muss entweder eine Exceptioninstanz oder eine Exceptionklasse (eine Klasse, die sich aus der Exception ableitet) sein. Wenn eine Exceptionklasse übergeben wird, wird sie implizit instanziiert, indem ihr Konstruktor ohne Argumente aufgerufen wird:

```
raise ValueError # kurz für 'raise ValueError()'
```

Wenn du feststellen musst, ob eine Exception ausgelöst wurde, aber nicht beabsichtigst, sie zu behandeln, erlaubt dir eine einfachere Form der `raise`-Anweisung, die Exception erneut zu werfen:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5. Benutzerdefinierte Exceptions

Programme können ihre eigenen Exceptions benennen, indem sie eine neue Exceptionklasse anlegen (siehe Klassen für weitere Informationen zu Python-Klassen). Exceptions sollten typischerweise direkt oder indirekt aus der Klasse `Exception` abgeleitet werden.

Exceptionklassen können definiert werden, die alles tun, was eine andere Klasse tun kann, werden aber in der Regel einfach gehalten und bieten oft nur eine Reihe von Attributen, die es ermöglichen, Informationen über den Fehler durch Behandler der Exception zu extrahieren. Bei der Erstellung eines Moduls, das mehrere verschiedene Fehler auslösen kann, ist es üblich, eine Basisklasse für von diesem Modul definierte Exceptions und eine Unterklasse für die Erstellung spezifischer Exceptionklassen für verschiedene Fehlerbedingungen zu erstellen:

```

class Error(Exception):
    """Basisklasse für Exceptions in diesem Modul."""
    pass

class InputError(Error):
    """Exception, die für Fehler im Input geworfen wird.

    Attribute:
        expression -- input expression, in welcher der Fehler auftrat
        message - Erklärung des Fehlers
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Wird geworfen, wenn eine Operation eine unerlaubte Statusveränderung
    versucht.

    Attribute:
        previous - Status zu Beginn der Veränderung
        next - angestrebter neuer Status
        message - Erklärung warum die Veränderung nicht erlaubt ist
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message

```

Die meisten Exceptions werden mit Namen definiert, die auf "Error" enden, ähnlich wie die Benennung der Standard-Exceptions.

Viele Standardbausteine definieren eigene Exceptions, um Fehler zu melden, die in von ihnen definierten Funktionen auftreten können. Weitere Informationen zu den Klassen sind im Abschnitt Klassen enthalten.


8.6. Definition von Bereinigungsmaßnahmen

Die Try-Anweisung enthält eine weitere optionale Anweisung, die dazu dient, Bereinigungsaktionen zu definieren, die unter allen Umständen ausgeführt werden müssen. Zum Beispiel:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Eine finally-Anweisung wird immer vor dem Verlassen der try-Anweisung ausgeführt, unabhängig davon, ob eine Exception aufgetreten ist oder nicht. Wenn eine Exception in der Try-Anweisung aufgetreten ist und nicht von einer except-Anweisung behandelt wurde (oder in einer except- oder else-Anweisung aufgetreten ist), wird sie nach Ausführung der end-Anweisung erneut ausgelöst. Die finally-Anweisung wird auch "auf dem Weg nach draußen" ausgeführt, wenn eine andere Anweisung der Try-Anweisung über eine Break-, Continue- oder Return-Anweisung verlassen wird. Ein komplizierteres Beispiel:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
```



```
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Wie du sehen kannst, wird die finally-Anweisung auf jeden Fall ausgeführt. Der TypeError, der durch die Division zweier Strings ausgelöst wird, wird nicht von der except-Anweisung behandelt und daher nach Ausführung der finally-Anweisung erneut ausgelöst.

In realen Anwendungen ist die finally-Anweisung nützlich, um externe Ressourcen (wie Dateien oder Netzwerkverbindungen) freizugeben, unabhängig davon, ob die Nutzung der Ressource erfolgreich war.


8.7. Vordefinierte Bereinigungsmaßnahmen

Einige Objekte definieren Standard-Bereinigungsaktionen, die durchzuführen sind, wenn das Objekt nicht mehr benötigt wird, unabhängig davon, ob die Operation mit dem Objekt erfolgreich war oder nicht. Sieh dir das folgende Beispiel an, das versucht, eine Datei zu öffnen und ihren Inhalt auf dem Bildschirm auszugeben.

```
for line in open("myfile.txt"):
    print(line, end="")
```

Das Problem mit diesem Code ist, dass er die Datei für eine unbestimmte Zeit offenlässt, nachdem dieser Teil des Codes die Ausführung beendet hat. Dies ist kein Problem in einfachen Skripten, kann aber bei größeren Anwendungen ein Problem darstellen. Die with-Anweisung ermöglicht es, Objekte wie Dateien so zu verwenden, dass sichergestellt ist, dass sie immer zeitnah und korrekt bereinigt werden.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```



Nach der Ausführung der Anweisung wird die Datei f immer geschlossen, auch wenn bei der Verarbeitung der Zeilen ein Problem aufgetreten ist. Objekte, die, wie Dateien, vordefinierte Bereinigungsaktionen bereitstellen, werden dies in ihrer Dokumentation anzeigen.





9. Klassen

Klassen bieten die Möglichkeit, Daten und Funktionalitäten zu bündeln. Das Erstellen einer neuen Klasse erzeugt einen neuen Objekttyp, sodass neue Instanzen dieses Typs erstellt werden können. Jede Klasseninstanz kann mit Attributen versehen sein, um ihren Zustand zu erhalten. Klasseninstanzen können auch Methoden (definiert durch ihre Klasse) zum Ändern ihres Zustands haben.

Im Vergleich zu anderen Programmiersprachen fügt der Klassenmechanismus von Python Klassen mit einem Minimum an neuer Syntax und Semantik hinzu. Es ist eine Mischung aus den Klassenmechanismen von C++ und Modula-3. Python-Klassen bieten alle Standardfunktionen der objektorientierten Programmierung: Der Klassenvererbungsmechanismus erlaubt mehrere Basisklassen, eine abgeleitete Klasse kann alle Methoden ihrer Basisklasse(n) überschreiben, und eine Methode kann die gleichnamigen Methode einer Basisklasse aufrufen. Objekte können beliebige Mengen und Arten von Daten enthalten. Wie bei Modulen nehmen Klassen die dynamische Natur von Python wahr: Sie werden zur Laufzeit erstellt und können nach der Erstellung weiter modifiziert werden.

In der C++-Terminologie sind normalerweise Teile einer Klasse (einschließlich der Datenelemente) öffentlich (außer siehe unten Private Variablen), und alle Teilfunktionen sind virtuell. Wie in Modula-3 gibt es keine Abkürzungen, um die Elemente des Objekts aus seinen Methoden zu referenzieren: Die Methodenfunktion wird mit einem expliziten ersten Argument deklariert, welches das Objekt repräsentiert, das implizit durch den Aufruf bereitgestellt wird. Wie in Smalltalk sind Klassen selbst Objekte. Dies bietet eine Semantik für den Import und die Umbenennung. Im Gegensatz zu C++ und Modula-3 können eingebaute Typen als Basisklassen für die Erweiterung durch den Benutzer verwendet werden. Außerdem können wie in C++ die meisten eingebauten Operatoren mit spezieller Syntax (arithmetische Operatoren, Indexierung usw.) für Klasseninstanzen neu definiert werden.

(Da es an allgemein akzeptierter Terminologie fehlt, um über Klassen zu sprechen, werde ich gelegentlich Smalltalk und C++-Begriffe verwenden. Ich würde Modula-3-Begriffe verwenden, da seine objektorientierte Semantik näher an der von Python liegt als C++, aber ich denke, dass nur wenige Leser davon gehört haben...)



9.1. Ein Wort zu Namen und Objekten

Objekte haben Individualität, und mehrere Namen (in mehreren Bereichen) können an das gleiche Objekt gebunden werden. In anderen Sprachen wird dies als Aliasing bezeichnet. Dies wird bei Python in der Regel nicht auf den ersten Blick erkannt und kann bei unveränderlichen Grundtypen (Zahlen, Strings, Tupel) sicher ignoriert werden. Aliasing hat jedoch einen möglicherweise überraschenden Einfluss auf die Semantik von Python-Code, der veränderbare Objekte wie Listen, Dictionaries und die meisten anderen Typen betrifft. Dies wird in der Regel zum Vorteil des Programms genutzt, da sich Aliase in mancher Hinsicht wie Zeiger verhalten. Zum Beispiel ist die Übergabe eines Objekts billig, da nur ein Zeiger von der Implementierung übergeben wird; und wenn eine Funktion ein als Argument übergebenes Objekt ändert, sieht der Aufrufer die Änderung - dies macht zwei verschiedene Argumentübergabemechanismen wie in Pascal überflüssig.

9.2. Python Scopes und Namensräume

Bevor ich Klassen einführe, muss ich dir zuerst etwas über die Scope Regeln von Python erzählen. Klassendefinitionen spielen einige ordentliche Streiche mit Namensräumen, und du musst wissen, wie Scopes und Namensräume funktionieren, um vollständig zu verstehen, was los ist. Übrigens ist das Wissen über dieses Thema für jeden fortgeschrittenen Python-Programmierer nützlich.

Beginnen wir mit einigen Definitionen.

Ein Namensraum ist eine Zuordnung von Namen zu Objekten. Die meisten Namensräume sind derzeit als Python-Dictionaries implementiert, aber das ist normalerweise in keiner Weise wahrnehmbar (außer in Bezug auf die Performance), und es kann sich in Zukunft ändern. Beispiele für Namensräume sind: die Menge eingebauter Namen (die Funktionen wie `abs()` und Namen von eingebaute Exceptions enthalten); die globalen Namen in einem Modul; und die lokalen Namen in einem Funktionsaufruf. In gewisser Weise bildet die Menge der Attribute eines Objekts auch einen Namensraum. Das Wichtigste, was man über Namensräume wissen sollte, ist, dass es absolut keine Beziehung zwischen Namen in verschiedenen Namensräumen gibt; z.B. können zwei verschiedene Module beide eine Funktion *maximize* ohne Verwirrung definieren - Benutzer der Module müssen ihr den Modulnamen voranstellen.

Übrigens verwende ich das Wort Attribut für jeden Namen, der einem Punkt folgt - zum Beispiel ist im Ausdruck `z.real` `real` ein Attribut des Objekts `z`. Streng genommen sind Referenzen auf Namen in Modulen Attributreferenzen: Im Ausdruck `modname.funcname` ist `modname` ein





Modulobjekt und *funcname* ist ein Attribut davon. In diesem Fall gibt es eine einfache Zuordnung zwischen den Attributen des Moduls und den im Modul definierten globalen Namen: Sie teilen sich den gleichen Namensraum! [1]

Attribute können schreibgeschützt oder beschreibbar sein. Im letzteren Fall ist eine Zuordnung zu Attributen möglich. Die Moduleigenschaften sind beschreibbar: Du kannst *modname.the_answer = 42* schreiben. Beschreibbare Attribute können mit der *del*-Anweisung auch gelöscht werden. Zum Beispiel entfernt *del modname.the_answer* das Attribut *the_answer* aus dem *modname* genannten Objekt.

Namensräume werden zu unterschiedlichen Zeitpunkten erstellt und haben unterschiedliche Lebenszeiten. Der Namensraum mit den eingebauten Namen wird beim Start des Python-Interpreters angelegt und nie gelöscht. Der globale Namensraum für ein Modul wird beim Einlesen der Moduldefinition angelegt; normalerweise bestehen Modulnamensräume auch, bis der Interpreter beendet wird. Die Anweisungen, die durch den Aufruf des Interpreters auf oberster Ebene ausgeführt werden, entweder aus einer Skriptdatei gelesen oder interaktiv, gelten als Teil eines Moduls namens *__main__*, sodass sie einen eigenen globalen Namensraum haben. (Die eingebauten Namen leben tatsächlich auch in einem Modul; dies wird als Built-in bezeichnet.)

Der lokale Namensraum für eine Funktion wird beim Aufruf der Funktion angelegt und gelöscht, wenn die Funktion eine Exception zurückgibt oder auslöst, die innerhalb der Funktion nicht behandelt wird. (Eigentlich wäre „Vergessen“ eine bessere Art, um zu beschreiben, was tatsächlich passiert.) Natürlich haben rekursive Aufrufe jeweils einen eigenen lokalen Namensraum.

Ein Scope ist eine textuelle Region eines Python-Programms, in der ein Namensraum direkt zugänglich ist. "Direkt zugänglich" bedeutet hier, dass eine ungeeignete Referenz auf einen Namen versucht, den Namen im Namensraum zu finden.

Obwohl Scopes statisch bestimmt werden, werden sie dynamisch verwendet. Während der Ausführung gibt es zu jeder Zeit mindestens drei verschachtelte Bereiche, deren Namensräume direkt zugänglich sind:

- Der innerste Bereich, der zuerst durchsucht wird, enthält die lokalen Namen.
- die Bereiche aller umschließenden Funktionen, die beginnend mit dem nächstgelegenen umschließenden Bereich durchsucht werden, enthalten nicht-lokale, aber auch nicht-globale Namen.
- Der vorletzte Bereich enthält die globalen Namen des aktuellen Moduls.



- der äußerste Bereich (zuletzt gesucht) ist der Namensraum mit eingebauten Namen.

Wenn ein Name als global deklariert wird, dann gehen alle Referenzen und Zuweisungen direkt in den mittleren Bereich, der die globalen Namen des Moduls enthält. Um Variablen, die außerhalb des innersten Bereichs gefunden wurden, neu zu binden, kann die `nonlocal`-Anweisung verwendet werden; wenn sie nicht als nicht-lokal deklariert werden, sind diese Variablen schreibgeschützt (ein Versuch, in eine solche Variable zu schreiben, erzeugt einfach eine neue lokale Variable im innersten Bereich, wobei die identisch benannte äußere Variable unverändert bleibt).

Normalerweise verweist der lokale Bereich auf die lokalen Namen der (textuell) aktuellen Funktion. Außerhalb von Funktionen verweist der lokale Bereich auf den gleichen Namensraum wie der globale Bereich: den Namensraum des Moduls. Klassendefinitionen platzieren einen weiteren Namensraum im lokalen Scope.

Es ist wichtig zu wissen, dass Scopes textuell bestimmt werden: Der globale Scope einer in einem Modul definierten Funktion ist der Namensraum dieses Moduls, unabhängig davon, von wo oder durch welchen Alias die Funktion aufgerufen wird. Andererseits erfolgt die eigentliche Suche nach Namen dynamisch, zur Laufzeit - die Sprachdefinition entwickelt sich jedoch in Richtung statischer Namensauflösung, zur "Kompilierzeit", also verlasse dich nicht auf dynamische Namensauflösung! (Tatsächlich werden lokale Variablen bereits statisch bestimmt.)

Eine besondere Eigenschaft von Python ist, dass - wenn keine globale Anweisung wirksam ist - Zuweisungen an Namen immer in den innersten Bereich gehen. Zuweisungen kopieren keine Daten - sie binden nur Namen an Objekte. Gleiches gilt für Löschungen: Die Anweisung `del x` entfernt die Bindung von `x` aus dem Namensraum, auf den der lokale Scope verweist. Tatsächlich verwenden alle Operationen, die neue Namen einführen, den lokalen Scope: Insbesondere Importausdrücke und Funktionsdefinitionen binden den Modul- oder Funktionsnamen im lokalen Bereich.

Das Statement `global` kann verwendet werden, um anzuzeigen, dass bestimmte Variablen im globalen Bereich leben und dort neu gebunden werden sollten; das Statement `nonlocal` zeigt an, dass bestimmte Variablen in einem umschließenden Scope leben und dort neu gebunden werden sollten.

9.2.1. Beispiele für Scopes und Namensräume

Dies ist ein Beispiel, das zeigt, wie man auf die verschiedenen Bereiche und Namensräume verweist und wie sich *global* und *nonlocal* sich auf die Variablenbindungen auswirken:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"


    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Die Ausgabe des Beispiels ist:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Beachte, dass die Anweisung *local* (die voreingestellt ist) die Bindung von *spam* durch *scope_test* nicht verändert hat. Die Anweisung *nonlocal* änderte die Bindung von *spam* durch *scope_test*, und die Anweisung *global* änderte die Bindung auf Modulebene.



Du siehst auch, dass es vor der Anweisung *global* keine Bindung für Spam gab.

9.3. Ein erster Blick auf Klassen

Die Klassen führen ein wenig neue Syntax, drei neue Objekttypen und eine neue Semantik ein.

9.3.1. Klassendefinitionssyntax

Die einfachste Form der Klassendefinition sieht so aus:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Klassendefinitionen müssen, wie Funktionsdefinitionen (def-Anweisungen), ausgeführt werden, bevor sie Wirkung zeigen. (Du könntest eine Klassendefinition möglicherweise in einen Zweig einer if-Anweisung oder in eine Funktion einfügen.)

In der Praxis werden die Anweisungen innerhalb einer Klassendefinition in der Regel Funktionsdefinitionen sein, aber andere Anweisungen sind erlaubt und manchmal nützlich - darauf kommen wir später noch zurück. Die Funktionsdefinitionen innerhalb einer Klasse haben in der Regel eine eigentümliche Form der Argumentenliste, die durch die Aufrufkonventionen für Methoden vorgegeben ist - dies wird später noch einmal erläutert.

Bei der Eingabe einer Klassendefinition wird ein neuer Namensraum angelegt und als lokaler Scope verwendet - alle Zuweisungen an lokale Variablen gehen also in diesen neuen Namensraum. Insbesondere Funktionsdefinitionen binden hier den Namen der neuen Funktion. Wenn eine Klassendefinition normal (über das Ende) verlassen wird, wird ein Klassenobjekt erzeugt. Dies ist im Grunde genommen ein Wrapper um den Inhalt des durch die Klassendefinition erzeugten Namensraums; wir werden im nächsten Abschnitt mehr über Klassenobjekte erfahren. Der ursprüngliche lokale Scope (derjenige, der kurz vor der Eingabe der Klassendefinition gültig war) wird wiederhergestellt, und das Klassenobjekt wird hier an den im Kopf der Klassendefinition angegebenen Klassennamen (ClassName im Beispiel) gebunden.



9.3.2. Klassenobjekte

Klassenobjekte unterstützen zwei Arten von Operationen: Attributreferenzen und Instanziierung. Attributreferenzen verwenden die Standardsyntax, die für alle Attributreferenzen in Python verwendet wird: *obj.name*. Gültige Attributnamen sind alle Namen, die sich beim Anlegen des Klassenobjekts im Namensraum der Klasse befanden. Also, wenn die Klassendefinition so aussah:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

dann sind *MyClass.i* und *MyClass.f* gültige Attributreferenzen, die jeweils eine ganze Zahl und ein Funktionsobjekt zurückgeben. Klassenattribute können auch zugewiesen werden, sodass du den Wert von *MyClass.i* durch Zuweisung ändern kannst. `__doc__` ist ebenfalls ein gültiges Attribut, das den zur Klasse gehörenden Docstring zurückgibt: *"A simple example class"*.

Die Klasseninstanziierung verwendet die Funktionsnotation. Stelle dir einfach vor, dass das Klassenobjekt eine parameterlose Funktion ist, die eine neue Instanz der Klasse zurückgibt. Zum Beispiel (unter der Annahme der obigen Klasse):


```
x = MyClass()
```

erzeugt eine neue Instanz der Klasse und weist dieses Objekt der lokalen Variablen *x* zu.

Die Instanziierungsoperation ("Aufruf" eines Klassenobjekts) erzeugt ein leeres Objekt. Viele Klassen erstellen gerne Objekte mit Instanzen, die auf einen bestimmten Ausgangszustand zugeschnitten sind. Daher kann eine Klasse eine spezielle Methode namens `__init__()` definieren, wie folgt:

```
def __init__(self):
    self.data = []
```





Wenn eine Klasse eine `__init__()`-Methode definiert, ruft die Klasseninstanziierung automatisch `__init__()` für die neu erstellte Klasseninstanz auf. In diesem Beispiel kann also eine neue, initialisierte Instanz erhalten werden durch:

```
x = MyClass()
```

Natürlich kann die `__init__()` Methode Argumente für mehr Flexibilität haben. In diesem Fall werden Argumente, die dem Instanziierungsoperator der Klasse übergeben werden, an `__init__()` übergeben. Zum Beispiel,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Instanzen-Objekte

Was können wir nun mit Instanzobjekten machen? Die einzigen Operationen, die von Instanzobjekten verstanden werden, sind Attributreferenzen. Es gibt zwei Arten von gültigen Attributnamen, Datenattributen und Methoden.

Datenattribute entsprechen "Instanzvariablen" in Smalltalk und "Datenelementen" in C++. Datenattribute müssen nicht deklariert werden; sie entstehen wie lokale Variablen, wenn sie erstmals zugewiesen werden. Wenn beispielsweise `x` die Instanz von `MyClass` ist, die oben erstellt wurde, druckt der folgende Code den Wert 16, ohne eine Spur zu hinterlassen:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```





Die andere Art der Instanzattributreferenz ist eine Methode. Eine Methode ist eine Funktion, die zu einem Objekt "gehört". (In Python ist der Begriff Methode nicht exklusiv für Klasseninstanzen: Andere Objekttypen können auch Methoden haben. Listenobjekte haben beispielsweise Methoden namens `append`, `insert`, `remove`, `sort`, etc. In der folgenden Diskussion werden wir den Begriff Methode jedoch ausschließlich für Methoden von Klasseninstanzobjekten verwenden, sofern nicht ausdrücklich etwas anderes angegeben ist.)

Gültige Methodennamen eines Instanzobjekts hängen von seiner Klasse ab. Per Definition definieren alle Attribute einer Klasse, die Funktionsobjekte sind, entsprechende Methoden ihrer Instanzen. In unserem Beispiel ist `x.f` also eine gültige Methodenreferenz, da `MyClass.f` eine Funktion ist, aber `x.i` nicht, da `MyClass.i` keine Funktion ist. Aber `x.f` ist nicht dasselbe wie `f` - es ist ein Methodenobjekt, kein Funktionsobjekt.

9.3.4. Methoden Objekte

Normalerweise wird eine Methode direkt nach der Bindung aufgerufen:

```
x.f()
```


Im Beispiel von `MyClass` gibt dies den String *'hello world'* zurück. Es ist jedoch nicht notwendig, eine Methode sofort aufzurufen: `x.f` ist ein Methodenobjekt und kann gespeichert und zu einem späteren Zeitpunkt aufgerufen werden. Zum Beispiel:

```
xf = x.f
while True:
    print(xf())
```

wird weiterhin *'hello world'* bis in alle Ewigkeit ausgegeben.

Was genau passiert, wenn eine Methode aufgerufen wird? Du hast vielleicht bemerkt, dass `x.f()` oben ohne Argument aufgerufen wurde, obwohl die Funktionsdefinition für `f()` ein Argument angegeben hat. Was ist mit dem Argument passiert? Sicherlich löst Python eine Exception aus, wenn eine Funktion, die ein Argument benötigt, ohne aufgerufen wird - auch wenn das Argument nicht tatsächlich verwendet wird....





Vielleicht hast du die Antwort erraten: Das Besondere an Methoden ist, dass das Instanzobjekt als erstes Argument der Funktion übergeben wird. In unserem Beispiel ist der Aufruf `x.f()` genau das gleiche wie `MyClass.f(x)`. Im Allgemeinen ist der Aufruf einer Methode mit einer Liste von `n` Argumenten äquivalent zum Aufruf der entsprechenden Funktion mit einer Argumentliste, die durch das Einfügen des Instanzobjekts der Methode vor dem ersten Argument erzeugt wird. Wenn du immer noch nicht verstehst, wie Methoden funktionieren, kann ein Blick auf die Implementierung vielleicht Klarheit schaffen. Wenn auf ein Nicht-Datenattribut einer Instanz verwiesen wird, wird die Klasse der Instanz durchsucht. Wenn der Name ein gültiges Klassenattribut bezeichnet, das ein Funktionsobjekt ist, wird ein Methodenobjekt erzeugt, indem das Instanzobjekt und das gerade gefundene Funktionsobjekt in einem abstrakten Objekt verpackt (Zeiger darauf) werden: dies ist das Methodenobjekt. Wenn das Methodenobjekt mit einer Argumentenliste aufgerufen wird, wird aus dem Instanzobjekt und der Argumentenliste eine neue Argumentenliste aufgebaut und das Funktionsobjekt mit dieser neuen Argumentenliste aufgerufen.

9.3.5. Klassen- und Instanzvariablen

Im Allgemeinen sind Instanzvariablen für Daten, die für jede Instanz eindeutig sind, und Klassenvariablen für Attribute und Methoden, die von allen Instanzen der Klasse gemeinsam genutzt werden:


```
class Dog:

    kind = 'canine'          # Klassenvariable wird von allen geteilt

    def __init__(self, name):
        self.name = name    # Instanzvariable speziell für jede Instanz

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # wird von allen geteilt
'canine'
>>> e.kind                # wird von allen geteilt
'canine'
>>> d.name                # speziell für d
'Fido'
```





```
>>> e.name          # speziell für e
'Buddy'
```

Gemeinsame Daten können bei der Verwendung von veränderbaren Objekten wie Listen und Dictionaries möglicherweise überraschende Auswirkungen haben. Beispielsweise sollte die Liste *tricks* im folgenden Code nicht als Klassenvariable verwendet werden, da nur eine einzige Liste von allen *Dog*-Instanzen gemeinsam genutzt wird:

```
class Dog:

    tricks = []           # falsche Verwendung einer Klassenvariablen
    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks          # unerwartet geteilt von allen Hunden
['roll over', 'play dead']
```


Ein korrektes Design der Klasse sollte stattdessen eine Instanzvariable verwenden:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # erstellt eine neue leere Liste für jeden Hund
    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
```





```
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. Weitere Bemerkungen


Datenattribute überschreiben Methodenattribute mit dem gleichen Namen; um versehentliche Namenskonflikte zu vermeiden, die in großen Programmen schwer zu findende Fehler verursachen können, ist es ratsam, eine Art Konvention zu verwenden, die die Wahrscheinlichkeit von Konflikten minimiert. Mögliche Konventionen sind die Großschreibung von Methodennamen, das Präfixieren von Datenattributnamen mit einer kleinen eindeutigen Zeichenkette (vielleicht nur einem Unterstrich) oder die Verwendung von Verben für Methoden und Nomen für Datenattribute.

Datenattribute können sowohl von Methoden als auch von normalen Benutzern ("Clients") eines Objekts referenziert werden. Mit anderen Worten, Klassen sind nicht geeignet, reine abstrakte Datentypen zu implementieren. Tatsächlich ermöglicht nichts in Python die Durchsetzung von Datenverstecken - alles basiert auf Konventionen. (Andererseits kann die in C geschriebene Python-Implementierung Implementierungsdetails vollständig ausblenden und bei Bedarf den Zugriff auf ein Objekt steuern; dies kann von Erweiterungen von Python in C verwendet werden.) Clients sollten Datenattribute mit Vorsicht verwenden - Clients können Invarianten, die von den Methoden gepflegt werden, durcheinanderbringen, indem sie ihre Datenattribute mit einem Stempel versehen. Bitte bedenke, dass Clients eigene Datenattribute zu einem Instanzobjekt hinzufügen können, ohne die Gültigkeit der Methoden zu beeinträchtigen, solange Namenskonflikte vermieden werden - auch hier kann eine Namenskonvention viele Kopfschmerzen ersparen.

Es gibt keine Abkürzung für die Referenz von Datenattributen (oder anderen Methoden!) innerhalb von Methoden. Ich finde, dass dies die Lesbarkeit von Methoden tatsächlich erhöht: Es besteht keine Chance, lokale Variablen und Instanzvariablen beim schnellen Überfliegen einer Methode zu verwechseln.

Oft wird das erste Argument einer Methode als *self* bezeichnet. Das ist nichts anderes als eine Konvention: Der Name selbst hat für Python absolut keine besondere Bedeutung. Es ist jedoch zu beachten, dass dein Code, wenn du der Konvention nicht folgst, für andere Python-





Programmierer weniger lesbar sein kann, und es ist auch denkbar, dass ein Klassenbrowserprogramm geschrieben wird, das auf einer solchen Konvention basiert. Jedes Funktionsobjekt, das ein Klassenattribut ist, definiert eine Methode für Instanzen dieser Klasse. Es ist nicht notwendig, dass die Funktionsdefinition textuell in der Klassendefinition enthalten ist: Die Zuordnung eines Funktionsobjekts zu einer lokalen Variablen in der Klasse ist ebenfalls in Ordnung. Zum Beispiel:

```
# Funktion definiert außerhalb der Klasse
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Nun sind *f*, *g* und *h* alle Attribute der Klasse *C*, die sich auf Funktionsobjekte beziehen, und somit sind sie alle Methoden von Instanzen von *C*, wobei *h* genau gleichwertig zu *g* ist. Beachte, dass diese Praxis normalerweise nur dazu führt, den Leser eines Programms zu verwirren. Methoden können andere Methoden aufrufen, indem sie Methodenattribute des Arguments *self* verwenden:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```





Methoden können auf globale Namen in der gleichen Weise verweisen wie gewöhnliche Funktionen. Der mit einer Methode verbundene globale Scope ist das Modul, das ihre Definition enthält. (Eine Klasse wird nie als globaler Scope verwendet.) Während man selten auf einen guten Grund für die Verwendung globaler Daten in einer Methode stößt, gibt es viele legitime Verwendungsmöglichkeiten des globalen Scopes: Zum einen können in den globalen Scope importierte Funktionen und Module von Methoden verwendet werden, ebenso wie in ihr definierte Funktionen und Klassen. Normalerweise ist die Klasse, die die Methode enthält, selbst in diesem globalen Scope definiert, und im nächsten Abschnitt werden wir einige gute Gründe finden, warum eine Methode auf ihre eigene Klasse verweisen möchte.

Jeder Wert ist ein Objekt und hat daher eine Klasse (auch Typ genannt). Sie wird als `object.__class__` gespeichert.


9.5. Vererbung

Natürlich wäre ein Sprachmerkmal ohne die Unterstützung der Vererbung dem Namen "Klasse" nicht würdig. Die Syntax für die Definition einer abgeleiteten Klasse sieht wie folgt aus:

```
class DerivedClassName(BaseClassName):  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```

Der Name *BaseClassName* muss in einem Scope definiert werden, der die Definition der abgeleiteten Klasse enthält. Anstelle eines Basisklassennamens sind auch andere beliebige Ausdrücke erlaubt. Dies kann z.B. sinnvoll sein, wenn die Basisklasse in einem anderen Modul definiert ist:





```
class DerivedClassName(modname.BaseClassName):
```

Die Ausführung einer Definition einer abgeleiteten Klasse verläuft wie bei einer Basisklasse. Wenn das Klassenobjekt konstruiert wird, wird die Basisklasse gespeichert. Dies wird zur Lösung von Attributverweisen verwendet: Wird ein angefordertes Attribut in der Klasse nicht gefunden, wird die Suche in der Basisklasse fortgesetzt. Diese Regel wird rekursiv angewendet, wenn die Basisklasse selbst von einer anderen Klasse abgeleitet ist.

Die Instanziierung von abgeleiteten Klassen ist nichts Besonderes: *DerivedClassName()* erzeugt eine neue Instanz der Klasse. Methodenreferenzen werden wie folgt aufgelöst: Das entsprechende Klassenattribut wird gesucht, ggf. absteigend in der Kette der Basisklassen, und die Methodenreferenz ist gültig, wenn daraus ein Funktionsobjekt resultiert.

Abgeleitete Klassen können Methoden ihrer Basisklassen überschreiben. Da Methoden keine besonderen Privilegien beim Aufruf anderer Methoden desselben Objekts haben, kann es vorkommen, dass eine Methode einer Basisklasse, die eine andere in derselben Basisklasse definierte Methode aufruft, eine Methode einer abgeleiteten Klasse aufruft, die sie überschreibt. (Für C++-Programmierer: Alle Methoden in Python sind effektiv virtuell.)


Eine übergeordnete Methode in einer abgeleiteten Klasse kann tatsächlich eher erweitern als einfach die gleichnamige Basisklassenmethode ersetzen wollen. Es gibt eine einfache Möglichkeit, die Basisklassenmethode direkt aufzurufen: einfach *BaseClassName.methodname(self, arguments)* aufrufen. Dies ist gelegentlich auch für Clients von Nutzen. (Beachte, dass dies nur funktioniert, wenn die Basisklasse im globalen Scope als *BaseClassName* zugänglich ist.)

Python hat zwei eingebaute Funktionen, die mit Vererbung arbeiten:

- Benutze *isinstance()*, um den Typ einer Instanz zu überprüfen: *isinstance(obj, int)* wird nur dann True sein, wenn *obj.__class__* int oder eine von int abgeleitete Klasse ist.
- Benutze *issubclass()*, um die Klassenvererbung zu überprüfen: *issubclass(bool, int)* ist True, da bool eine Unterklasse von int ist. *issubclass(float, int)* ist jedoch False, da float keine Unterklasse von int ist.

9.5.1. Mehrfachvererbung

Python unterstützt auch eine Form der Mehrfachvererbung. Eine Klassendefinition mit mehreren Basisklassen sieht so aus:



```
class DerivedClassName(Base1, Base2, Base3):  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```


In den meisten Fällen kann man sich im einfachsten Fall die Suche nach Attributen, die von einer übergeordneten Klasse geerbt wurden, als depth-first, left-to-right vorstellen und nicht zweimal in derselben Klasse suchen, wo es eine Überschneidung in der Hierarchie gibt. Wenn also ein Attribut nicht in *DerivedClassName* gefunden wird, wird es in *Base1*, dann (rekursiv) in den Basisklassen von *Base1* gesucht, und wenn es dort nicht gefunden wurde, wird es in *Base2* gesucht, und so weiter.

Tatsächlich ist es etwas komplexer als das; die Reihenfolge der Methodenauflösung ändert sich dynamisch, um kooperative Aufrufe von *super()* zu unterstützen. Dieser Ansatz ist in einigen anderen Sprachen mit mehreren Vererbungssprachen als Call-Next-Methode bekannt und leistungsfähiger als der Superaufruf in Single-Inverbationssprachen.

Dynamische Ordnung ist notwendig, da alle Fälle von Mehrfachvererbung eine oder mehrere Diamantbeziehungen aufweisen (wobei mindestens eine der übergeordneten Klassen über mehrere Pfade von der untersten Klasse aus erreichbar ist). Beispielsweise erben alle Klassen von *object*, sodass jeder Fall einer Mehrfachvererbung mehr als einen Pfad zum Erreichen von *object* bietet. Um zu verhindern, dass auf die Basisklassen mehr als einmal zugegriffen werden kann, linearisiert der dynamische Algorithmus die Suchreihenfolge so, dass die in jeder Klasse angegebene Reihenfolge von links nach rechts beibehalten wird, dass jeder Elternteil nur einmal aufgerufen wird und dass sie monoton ist (was bedeutet, dass eine Klasse unterklassifiziert werden kann, ohne die Reihenfolge ihrer Eltern zu beeinflussen). Zusammengefasst ermöglichen diese Eigenschaften die Entwicklung zuverlässiger und erweiterbarer Klassen mit Mehrfachvererbung. Weitere Informationen findest du unter <https://www.python.org/download/releases/2.3/mro/>.

9.6. Private Variablen

"Private" Instanzvariablen, auf die nur innerhalb eines Objekts zugegriffen werden kann, existieren in Python nicht. Es gibt jedoch eine Konvention, der sich die meisten Python-Codes



anschließen: Ein Name, dem ein Unterstrich vorangestellt ist (z.B. `_spam`), sollte als nicht-öffentlicher Teil der API behandelt werden (unabhängig davon, ob es sich um eine Funktion, eine Methode oder ein Datenelement handelt). Es sollte als Detail der Implementierung betrachtet werden und kann ohne Vorankündigung geändert werden.

Da es einen gültigen Anwendungsfall für class-private Mitglieder gibt (nämlich um Namenskollisionen von Namen mit Namen, die durch Unterklassen definiert sind, zu vermeiden), gibt es nur begrenzte Unterstützung für einen solchen Mechanismus, der als Name Mangling bezeichnet wird. Jeder Bezeichner der Form `__spam` (mindestens zwei führende Unterstriche, höchstens ein nachstehender Unterstrich) wird textuell durch `_classname__spam` ersetzt, wobei `classname` der aktuelle Klassenname ist und führende Unterstriche entfernt werden. Dieses Mangling erfolgt ohne Rücksicht auf die syntaktische Position des Identifikators, sofern sie innerhalb der Definition einer Klasse auftritt.

Das Name Mangling ist hilfreich, um Methoden von Unterklassen zu überschreiben, ohne Methodenaufrufe innerhalb der Klasse zu unterbrechen. Zum Beispiel:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)


    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # private Kopie der originalen Methode update()

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # bietet neue Signatur für update()
        # aber bricht nicht __init__() ab
        for item in zip(keys, values):
            self.items_list.append(item)
```





Das obige Beispiel würde auch dann funktionieren, wenn *MappingSubclass* einen `__update` Identifier einführen würde, da er durch `_Mapping__update` in der Klasse *Mapping* bzw. `_MappingSubclass__update` in der Klasse *MappingSubclass* ersetzt wird.

Man beachte, dass die Mangling Regeln hauptsächlich darauf ausgelegt sind, Unfälle zu vermeiden; es ist immer noch möglich, auf eine Variable zuzugreifen oder sie zu ändern, die als privat gilt. Dies kann auch in besonderen Situationen, wie z.B. im Debugger, sinnvoll sein.

Beachte auch, dass Code, der an `exec()` oder `eval()` übergeben wird, den Klassennamen der aufrufenden Klasse nicht als die aktuelle Klasse betrachtet; dies ähnelt dem Effekt der Statements `global`, dessen Effekt ebenfalls auf Code beschränkt ist, der zusammen Byte-kompiliert wird. Die gleiche Einschränkung gilt für `getattr()`, `setattr()` und `delattr()` sowie bei direkter Referenz auf `__dict__`.

9.7. Chancen und Risiken

Manchmal ist es sinnvoll, einen Datentyp ähnlich dem Pascal "record" oder C "struct" zu haben, der einige benannte Datenelemente bündelt. Eine leere Klassendefinition ist gut geeignet:

```
class Employee:
    pass

john = Employee() # Erstellt eine leere Instanz von employee

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Ein Stück Python-Code, das einen bestimmten abstrakten Datentyp erwartet, kann oft einer Klasse übergeben werden, die stattdessen die Methoden dieses Datentyps emuliert. Wenn du zum Beispiel eine Funktion hast, die einige Daten aus einem Dateiobjekt formatiert, kannst du eine Klasse mit den Methoden `read()` und `readline()` definieren, die die Daten stattdessen aus einem Stringbuffer holen und als Argument übergeben.

Objekte der Instanzmethode haben ebenfalls Attribute: `m.__self__` ist das Instanzobjekt mit der Methode `m()`, und `m.__func__` ist das der Methode entsprechende Funktionsobjekt.



9.8. Iteratoren

Inzwischen hast du wahrscheinlich bemerkt, dass die meisten Containerobjekte mit einer for-Anweisung überlaufen werden können:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Diese Art des Zugangs ist klar, prägnant und komfortabel. Die Verwendung von Iteratoren durchdringt und vereinheitlicht Python. Hinter den Kulissen ruft die for-Anweisung *iter()* auf dem Containerobjekt auf. Die Funktion gibt ein Iteratorobjekt zurück, das die Methode *__next__()* definiert, die nacheinander auf Elemente im Container zugreift. Wenn es keine Elemente mehr gibt, löst *__next__()* eine StopIteration-Exception aus, die die for-Schleife anweist, zu beenden. Du kannst die *__next__()* Methode mit der eingebauten Funktion *next()* aufrufen; dieses Beispiel zeigt, wie alles funktioniert:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
```

StopIteration


Nachdem du die Mechanik hinter dem Iteratorprotokoll gesehen hast, ist es einfach, Iteratorverhalten zu deinen Klassen hinzuzufügen. Definiere eine `__iter__()` Methode, die ein Objekt mit einer `__next__()` Methode zurückgibt. Wenn die Klasse `__next__()` definiert, dann kann `__iter__()` einfach `self` zurückgeben:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```





```

>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s

```

9.9. Generatoren

Generatoren sind ein einfaches und leistungsfähiges Werkzeug zur Erstellung von Iteratoren. Sie sind wie normale Funktionen geschrieben, verwenden aber die das Statement `yield`, wenn sie Daten zurückgeben wollen. Jedes Mal, wenn `next()` aufgerufen wird, setzt der Generator dort fort, wo er aufgehört hat (er speichert alle Datenwerte und welche Anweisung zuletzt ausgeführt wurde). Ein Beispiel zeigt, dass Generatoren trivial einfach zu erstellen sein können:

```

def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]


```

```

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g

```

Alles, was mit Generatoren gemacht werden kann, kann auch mit klassenbasierten Iteratoren gemacht werden, wie im vorherigen Abschnitt beschrieben. Was Generatoren so kompakt macht, ist, dass die Methoden `__iter__()` und `__next__()` automatisch erstellt werden. Ein weiteres wichtiges Merkmal ist, dass die lokalen Variablen und der Ausführungszustand zwischen den Aufrufen automatisch gespeichert werden. Dies machte die Funktion einfacher zu



schreiben und viel übersichtlicher als ein Ansatz mit Instanzvariablen wie *self.index* und *self.data*.

Zusätzlich zur automatischen Methodenerstellung und zum Speichern des Programmszustands werfen Generatoren beim Beenden automatisch die StopIteration. In Kombination machen es diese Funktionen einfach, Iteratoren ohne größeren Aufwand zu erstellen, als eine normale Funktion zu schreiben.

9.10. Generatorausdrücke

Einige einfache Generatoren können mit einer Syntax, die der Listen-Abstraktion ähnelt, aber Klammern anstelle von eckigen Klammern enthält, prägnant als Ausdrücke kodiert werden. Diese Ausdrücke sind für Situationen vorgesehen, in denen der Generator sofort durch eine umschließende Funktion verwendet wird. Generatorausdrücke sind kompakter, aber weniger vielseitig als vollständige Generatordefinitionen und neigen dazu, speicherfreundlicher zu sein als äquivalente Listen-Abstraktionen.

Beispiele:

```
>>> sum(i*i for i in range(10))           # Summe der Quadrate
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))   # Kreuzprodukt
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```





Fußnoten

- Bis auf eine Sache. Modulobjekte haben ein geheimes schreibgeschütztes Attribut namens `__dict__`, das das Dictionary zurückgibt, das zur Implementierung des Namensraums des Moduls verwendet wird; der Name `__dict__` ist ein Attribut, aber kein globaler Name. Offensichtlich verletzt die Verwendung dieser Methode die Abstraktion der Namensraum-Implementierung und sollte auf Dinge wie Post-Mortem-Debugger beschränkt sein.
- [1]

10. Kurzer Rundgang durch die Standardbibliothek

10.1. Betriebssystem-Schnittstelle

Das Modul `os` bietet Dutzende von Funktionen zur Interaktion mit dem Betriebssystem:

```
>>> import os
>>> os.getcwd()          # Gibt das aktuelle Working Directory zurück
'C:\\Python37'
>>> os.chdir('/server/accesslogs') # Ändert das aktuelle Working Directory
>>> os.system('mkdir today')      # Führt das Kommando mkdir in der Shell aus
0
```

Vergewissere dich, dass du `import os` anstelle von `from os import *` verwendest. Dies wird `os.open()` davon abhalten, die eingebaute `open()`-Funktion, die ganz anders funktioniert, zu überdecken.

Die eingebauten Funktionen `dir()` und `help()` sind als interaktive Hilfsmittel für die Arbeit mit großen Modulen wie `os` nützlich:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Für tägliche Aufgaben der Datei- und Verzeichnisverwaltung bietet das Modul `shutil` eine höherwertige und einfach zu bedienende Schnittstelle:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```



10.2. Datei-Wildcards

Das Modul `glob` bietet eine Funktion zum Erstellen von Dateilisten aus der Wildcard-Suche im Verzeichnis:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. Befehlszeilenargumente

Gängige Utility-Skripte müssen oft Befehlszeilenargumente verarbeiten. Diese Argumente werden im Attribut `argv` des `sys`-moduls als Liste gespeichert. Zum Beispiel ergibt sich die folgende Ausgabe aus der Ausführung von `python demo.py one two three` auf der Kommandozeile:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

Das Modul `getopt` verarbeitet `sys.argv` unter Verwendung der Konventionen der Unix-Funktion `getopt()`. Eine leistungsfähigere und flexiblere Befehlszeilenverarbeitung bietet das Modul `argparse`.

10.4. Fehleroutput Redirection und Programmterminierung

Das Modul `sys` hat auch Attribute für `stdin`, `stdout` und `stderr`. Letzteres ist nützlich, um Warnungen und Fehlermeldungen auszugeben, um sie auch dann sichtbar zu machen, wenn `stdout` umgeleitet wurde:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Der direkteste Weg, ein Skript zu beenden, ist die Verwendung von `sys.exit()`.

10.5. String Pattern Matching

Das Modul `re` bietet Werkzeuge für reguläre Ausdrücke (regex) für die erweiterte Stringverarbeitung. Für komplexes Matching und Manipulation bieten reguläre Ausdrücke prägnante, optimierte Lösungen:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Wenn nur einfache Fähigkeiten benötigt werden, werden Stringmethoden bevorzugt, da sie einfacher zu lesen und zu debuggen sind:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6. Mathematik

Das Modul `math` ermöglicht den Zugriff auf die zugrundeliegenden Funktionen der C-Bibliothek für die Gleitkomma-Mathematik:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Das Modul `random` bietet Werkzeuge zur Durchführung von Zufallsauswahlen:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10) # sampling ohne Ersetzen
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float
```



```
0.17970987693706186
>>> random.randrange(6)    # zufällige ganze Zahl aus range(6)
4
```

Das Modul `statistics` berechnet grundlegende statistische Eigenschaften (Mittelwert, Median, Varianz usw.) von numerischen Daten:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

Das SciPy-Projekt <<https://scipy.org>> hat viele weitere Module für numerische Berechnungen.

10.7. Internetzugang

Es gibt eine Reihe von Modulen für den Internetzugang und die Verarbeitung von Internetprotokollen. Zwei der einfachsten sind `urllib.request` zum Abrufen von Daten aus URLs und `smtplib` zum Senden von E-Mails:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8') # Decoding der binären Daten zu Text.
...         if 'EST' in line or 'EDT' in line: # Eastern Time einstellen
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
```

```
... From: soothsayer@example.org
...
... Beware the Ides of March.
... "")
>>> server.quit()
```

(Beachte dass das zweite Beispiel einen Mailserver benötigt, der auf localhost läuft.)

10.8. Daten und Zeiten

Das Modul `datetime` stellt Klassen zur Verfügung, um Datum und Uhrzeit sowohl einfach als auch komplex zu manipulieren. Während die Datums- und Zeitarithmetik unterstützt wird, liegt der Schwerpunkt der Implementierung auf der effizienten Elementextraktion zur Ausgabeformatierung und -manipulation. Das Modul unterstützt auch Objekte, die zeitzonenbewusst sind.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9. Datenkompression

Gängige Datenarchivierungs- und Komprimierungsformate werden direkt von Modulen unterstützt, darunter: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` und `tarfile`.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
```

```

41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979

```

10.10. Performance Messung

Einige Python-Anwender entwickeln ein großes Interesse daran, die relative Leistung verschiedener Ansätze für dasselbe Problem zu kennen. Python bietet ein Messwerkzeug, das diese Fragen sofort beantwortet.

Beispielsweise kann es verlockend sein, die Funktion zum Ein- und Auspacken von Tupeln anstelle des traditionellen Ansatzes zum Austauschen von Argumenten zu verwenden. Das Modul `timeit` zeigt schnell einen bescheidenen Leistungsvorteil:

```

>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791

```


Im Gegensatz zur feinen Granularität von `timeit` bieten die Module `profile` und `pstats` Werkzeuge zur Identifizierung zeitkritischer Abschnitte in größeren Codeblöcken.

10.11. Qualitätskontrolle

Ein Ansatz für die Entwicklung hochwertiger Software ist es, für jede Funktion, sobald sie entwickelt wird, Tests zu schreiben und diese häufig während des Entwicklungsprozesses durchzuführen.

Das Modul `doctest` stellt ein Werkzeug zum Scannen eines Moduls und zur Validierung von Tests zur Verfügung, die in den Docstrings eines Programms eingebettet sind. Der Testaufbau ist so einfach wie das Ausschneiden und Einfügen eines typischen Aufrufs mit seinen Ergebnissen in die Dokumentzeile. Dies verbessert die Dokumentation, indem es dem Benutzer





ein Beispiel liefert und es dem Modul doctest ermöglicht, sicherzustellen, dass der Code der Dokumentation treu bleibt:

```
def average(values):  
    """Computes the arithmetic mean of a list of numbers.  
  
    >>> print(average([20, 30, 70]))  
    40.0  
    """  
    return sum(values) / len(values)  
  
import doctest  
doctest.testmod() # validiert automatisch eingebettete Tests
```

Das Modul unittest ist aufwendiger als das doctest Modul, aber es ermöglicht es, einen umfassenderen Satz von Tests in einer separaten Datei zu verwalten:


```
import unittest  
  
class TestStatisticalFunctions(unittest.TestCase):  
    def test_average(self):  
        self.assertEqual(average([20, 30, 70]), 40.0)  
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)  
        with self.assertRaises(ZeroDivisionError):  
            average([])  
        with self.assertRaises(TypeError):  
            average(20, 30, 70)  
  
unittest.main() # Aufruf von der Kommandozeile führt alle Tests aus
```

10.12. Inkludierte Batterien

Python hat eine Philosophie der "inkludierten Batterien". Dies zeigt sich am besten an den ausgeklügelten und robusten Fähigkeiten der größeren Packages. Zum Beispiel:

- Die Module *xmlrpc.client* und *xmlrpc.server* machen die Implementierung von Remote Procedure Calls zu einer fast trivialen Aufgabe. Trotz der Namen der Module sind keine direkten Kenntnisse oder der Umgang mit XML erforderlich.



- 
- Das Package email ist eine Bibliothek zur Verwaltung von E-Mail-Nachrichten, einschließlich MIME und anderer RFC 2822-basierter Nachrichtendokumente. Im Gegensatz zu smtplib und poplib, die tatsächlich Nachrichten senden und empfangen, verfügt email über ein komplettes Toolset zum Erstellen oder Dekodieren komplexer Nachrichtenstrukturen (einschließlich Anhängen) und zur Implementierung von Internet-Codierung und Header-Protokollen.
 - Das Package json bietet eine robuste Unterstützung für das Parsen dieses beliebten Datenaustauschformats. Das csv-Modul unterstützt das direkte Lesen und Schreiben von Dateien im Comma-Separated-Value-Format, das häufig von Datenbanken und Tabellenkalkulationen unterstützt wird. Die XML-Verarbeitung wird von den Packages *xml.etree.ElementTree*, *xml.dom* und *xml.sax* unterstützt. Zusammen vereinfachen diese Module und Pakete den Datenaustausch zwischen Python-Anwendungen und anderen Tools erheblich.
 - Das Modul sqlite3 ein Wrapper für die SQLite-Datenbankbibliothek und bietet eine beständige Datenbank, die aktualisiert und mit einer leicht abweichenden SQL-Syntax aufgerufen werden kann.
 - Die Internationalisierung wird von einer Reihe von Modulen unterstützt, darunter gettext, locale und das Codecs Package.



11. Kurzer Rundgang durch die Standardbibliothek - Teil II

Diese zweite Tour behandelt fortgeschrittenere Module, die professionelle Programmieranforderungen unterstützen. Diese Module kommen in kleinen Skripten selten vor.

11.1. Output Formatierung


Das Modul `reprlib` bietet eine Version von `repr()`, die für abgekürzte Anzeigen von großen oder tief verschachtelten Containern angepasst ist:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

Das Modul `pprint` bietet eine elegantere Kontrolle über das Drucken von eingebauten und benutzerdefinierten Objekten in einer für den Interpreter lesbaren Form. Wenn das Ergebnis länger als eine Zeile ist, fügt der "hübsche Drucker" Zeilenumbrüche und Einrückungen hinzu, um die Datenstruktur besser sichtbar zu machen:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...           'yellow'], 'blue']]
...
>>> pprint.pprint(t, width=30)
[[['black', 'cyan',
    'white',
    ['green', 'red']],
  [['magenta', 'yellow'],
    'blue']]]
```

Das Modul `textwrap` formatiert Textabschnitte so, dass sie einer bestimmten Bildschirmbreite entsprechen:



```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```


Das Modul locale greift auf eine Datenbank mit kulturspezifischen Datenformaten zu. Das Gruppierungsattribut der Formatierungsfunktion von locale bietet eine direkte Möglichkeit, Zahlen mit Gruppentrennzeichen zu formatieren:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()           # erhalte ein Mapping von Konventionen
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                               conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. Templating

Das Modul string enthält eine vielseitige Klasse Template mit einer vereinfachten Syntax, die für die Bearbeitung durch den Endbenutzer geeignet ist. Auf diese Weise können Benutzer ihre Anwendungen anpassen, ohne die Anwendung ändern zu müssen.

Das Format verwendet Platzhalternamen, die aus \$ mit gültigen Python-Identifikatoren gebildet werden (alphanumerische Zeichen und Unterstriche). Wenn der Platzhalter mit geschweiften Klammern umgeben ist, können ihm weitere alphanumerische Zeichen ohne Leerzeichen folgen. Die Eingabe von \$\$\$ erzeugt ein einzelnes escapedes \$:



```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

Die Methode `substitute()` löst einen `KeyError` aus, wenn ein Platzhalter nicht in einem Dictionary oder einem Keywordargument angegeben wird. Für Mail-Merge-Anwendungen können benutzerdefinierte Daten unvollständig sein, und die Methode `safe_substitute()` ist möglicherweise geeigneter - sie lässt Platzhalter unverändert, wenn Daten fehlen:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Unterklassen von `Template` können ein benutzerdefiniertes Trennzeichen angeben. Beispielsweise kann ein Programm zum Umbenennen von Batches für einen Fotobrowser wählen, ob Prozentzeichen für Platzhalter wie das aktuelle Datum, die Bildfolgennummer oder das Dateiformat verwendet werden sollen:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
```



```
...     print('{0} --> {1}'.format(filename, newname))
```

```
img_1074.jpg --> Ashley_0.jpg
```

```
img_1076.jpg --> Ashley_1.jpg
```

```
img_1077.jpg --> Ashley_2.jpg
```

Eine weitere Anwendung für das Templating ist die Trennung der Programmlogik von den Details mehrerer Ausgabeformate. Dies ermöglicht es, benutzerdefinierte Vorlagen für XML-Dateien, einfache Textberichte und HTML-Webberichte zu ersetzen.

11.3. Arbeiten mit binären Datensatzlayouts

Das Strukturmodul bietet `pack()` und `unpack()` Funktionen zum Arbeiten mit binären Datensatzformaten variabler Länge. Das folgende Beispiel zeigt, wie man Headerinformationen in einer ZIP-Datei ohne Verwendung des Moduls `zipfile` durchläuft. Die Packungscodes "H" und "I" stehen für zwei bzw. vier Byte vorzeichenlose Zahlen. Das "<" zeigt an, dass es sich um eine Standardgröße in Little-Endian-Byte-Reihenfolge handelt:

```
import struct
```

```
with open('myfile.zip', 'rb') as f:  
    data = f.read()
```

```
start = 0
```

```
for i in range(3): # zeige die ersten 3 Datei Header
```

```
    start += 14
```

```
    fields = struct.unpack('<IIIHH', data[start:start+16])
```

```
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields
```

```
    start += 16
```

```
    filename = data[start:start+filenamesize]
```

```
    start += filenamesize
```

```
    extra = data[start:start+extra_size]
```

```
    print(filename, hex(crc32), comp_size, uncomp_size)
```

```
    start += extra_size + comp_size # springe zum nächsten Header
```

11.4. Multi-threading

Das Threading ist eine Technik zur Entkopplung von Aufgaben, die nicht sequentiell voneinander abhängig sind. Threads können verwendet werden, um die Reaktionsfähigkeit von Anwendungen zu verbessern, die Benutzereingaben akzeptieren, während andere Aufgaben im Hintergrund ausgeführt werden. Ein verwandter Anwendungsfall ist die Ausführung von I/O parallel zu Berechnungen in einem anderen Thread.

Der folgende Code zeigt, wie das übergeordnete Threading-Modul Aufgaben im Hintergrund ausführen kann, während das Hauptprogramm weiterläuft:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile


    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Warte, bis die Background Aufgabe beendet ist
print('Main program waited until background was done.')
```

Die größte Herausforderung bei Multi-Thread-Anwendungen ist die Koordination von Threads, die Daten oder andere Ressourcen gemeinsam nutzen. Zu diesem Zweck stellt das Threading-Modul eine Reihe von Synchronisations-Primitiven zur Verfügung, darunter Locks, Events, Konditionsvariablen und Semaphoren.

Obwohl diese Tools leistungsfähig sind, können kleine Konstruktionsfehler zu Problemen führen, die schwer zu reproduzieren sind. Der bevorzugte Ansatz für die Aufgabenkoordination besteht



also darin, den gesamten Zugriff auf eine Ressource in einem einzigen Thread zu konzentrieren und dann das Modul `queue` zu verwenden, um diesen Thread mit Anfragen von anderen Threads zu versorgen. Anwendungen, die Queue-Objekte für die Kommunikation und Koordination zwischen den Threads verwenden, sind einfacher zu gestalten, lesbarer und zuverlässiger.

11.5. Protokollierung

Das Modul `logging` bietet ein vollwertiges und flexibles Logging-System. Im einfachsten Fall werden Protokollnachrichten an eine Datei oder an `sys.stderr` gesendet:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Dies führt zu folgender Ausgabe:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Standardmäßig werden Informations- und Debugging-Meldungen unterdrückt und die Ausgabe an den Standardfehler gesendet. Weitere Ausgabemöglichkeiten sind das Routing von Nachrichten über E-Mail, Datagramme, Sockets oder an einen HTTP-Server. Neue Filter können je nach Nachrichtenpriorität unterschiedliche Routings auswählen: `DEBUG`, `INFO`, `WARNING`, `ERROR` und `CRITICAL`.

Das Protokollierungssystem kann direkt aus Python heraus konfiguriert oder aus einer benutzerdefinierten Konfigurationsdatei geladen werden, um die Protokollierung individuell anzupassen, ohne die Anwendung zu verändern.

11.6. Schwache Referenzen

Python führt eine automatische Speicherverwaltung durch (Referenzzählung für die meisten Objekte und Garbage Collection zur Vermeidung von Zyklen). Der Speicher wird kurz nach dem Löschen der letzten Referenz freigegeben.

Dieser Ansatz funktioniert für die meisten Anwendungen gut, aber gelegentlich besteht die Notwendigkeit, Objekte nur so lange zu verfolgen, wie sie von etwas anderem verwendet werden. Leider schafft die bloße Verfolgung eine Referenz, die sie dauerhaft macht. Das Modul `weakref` bietet Werkzeuge zur Verfolgung von Objekten, ohne eine Referenz zu erstellen. Wenn das Objekt nicht mehr benötigt wird, wird es automatisch aus einer `weakref`-Tabelle entfernt und ein Rückruf für `weakref`-Objekte ausgelöst. Typische Anwendungen sind Cachingobjekte, deren Erstellung teuer ist:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                                # Referenz erstellen
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a                          # erstellt keine Referenz
>>> d['primary']                             # Ausgabe des Objekts, falls noch existent
10
>>> del a                                    # entfernt die eine Referenz
>>> gc.collect()                            # Garbage Collection sofort ausführen
0
>>> d['primary']                             # Eintrag wurde automatisch entfernt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                             # Eintrag wurde automatisch entfernt
  File "C:/python37/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```



11.7. Tools für die Arbeit mit Listen

Viele Anforderungen an die Datenstruktur können mit dem integrierten Listentyp erfüllt werden. Manchmal besteht jedoch Bedarf an alternativen Implementierungen mit unterschiedlichen Leistungskompromissen.

Das Modul `array` stellt ein `array()`-Objekt zur Verfügung, das wie eine Liste aussieht, die nur homogene Daten speichert und diese kompakter speichert. Das folgende Beispiel zeigt ein Array von Zahlen, die als vorzeichenlose Zwei-Byte-Binärzahlen (Typcode "H") gespeichert sind, anstatt der üblichen 16 Bytes pro Eintrag für reguläre Listen von Python `int` Objekten:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

Das Modul `collections` stellt ein `deque()`-Objekt zur Verfügung, das wie eine Liste mit schnelleren Appends und Pops von der linken Seite, aber langsameren Lookups in der Mitte ist. Diese Objekte eignen sich gut für die Implementierung von Queues und die Breitensuche in Bäumen:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
    unsearched.append(m)
```

Neben alternativen Listenimplementierungen bietet die Bibliothek auch andere Werkzeuge wie das Modul `bisect` mit Funktionen zur Manipulation sortierter Listen:



```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

Das Modul `heapq` stellt Funktionen zur Verfügung, um Heaps auf Basis von regulären Listen zu implementieren. Der niederwertigste Eintrag wird immer auf der Position Null gehalten. Dies ist nützlich für Anwendungen, die wiederholt auf das kleinste Element zugreifen, aber keine vollständige Listensortierung durchführen wollen:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # Liste umsortieren in Heap Reihenfolge
>>> heappush(data, -5) # neuen Eintrag hinzufügen
>>> [heappop(data) for i in range(3)] # drei kleinsten Einträge abrufen
[-5, 0, 1]
```

11.8. Dezimalzahlenarithmetik

Das Modul `decimal` bietet einen dezimalen Datentyp für die dezimale Gleitkommaarithmetik. Im Vergleich zur eingebauten `Float`-Implementierung von binären Gleitkommazahlen ist die Klasse besonders hilfreich für

- Finanzanwendungen und andere Anwendungen, die eine genaue Dezimaldarstellung erfordern,
- Kontrolle über die Präzision,
- Kontrolle über die Rundung zur Erfüllung gesetzlicher oder regulatorischer Anforderungen,
- Verfolgung signifikanter Dezimalstellen oder
- Anwendungen, bei denen der Benutzer erwartet, dass die Ergebnisse mit den von Hand durchgeführten Berechnungen übereinstimmen.



Zum Beispiel gibt die Berechnung einer 5%-Steuer auf eine 70-Cent-Telefongebühr unterschiedliche Ergebnisse in dezimaler Gleitkommazahl und binärer Gleitkommazahl. Die Differenz wird signifikant, wenn die Ergebnisse auf den nächsten Cent gerundet werden:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

Das Dezimalergebnis behält eine nachgestellte Null und leitet automatisch ein vierstelliges Ergebnis aus Faktoren mit jeweils zwei Stellen ab. Decimal reproduziert die Mathematik wie von Hand und vermeidet Probleme, die auftreten können, wenn binäre Gleitkommazahlen keine exakten Dezimalgrößen darstellen können.

Die genaue Darstellung ermöglicht es der Klasse Decimal, Modulo-Berechnungen und Gleichheitstests durchzuführen, die für binäre Gleitkommazahlen ungeeignet sind:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

Das Modul decimal ermöglicht die Arithmetik mit der erforderlichen Genauigkeit:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```





12. Virtuelle Umgebungen und Pakete

12.1. Einleitung

Python-Anwendungen verwenden oft Pakete und Module, die nicht Teil der Standardbibliothek sind. Anwendungen benötigen manchmal eine bestimmte Version einer Bibliothek, da die Anwendung erfordern kann, dass ein bestimmter Fehler behoben wurde oder die Anwendung mit einer veralteten Version der Benutzeroberfläche der Bibliothek geschrieben werden kann. Dies bedeutet, dass es unter Umständen nicht möglich ist, dass eine Python-Installation die Anforderungen jeder Anwendung erfüllt. Wenn Anwendung A die Version 1.0 eines bestimmten Moduls, aber Anwendung B die Version 2.0 benötigt, dann stehen die Anforderungen im Konflikt und die Installation einer der Versionen 1.0 oder 2.0 führt dazu, dass eine Anwendung nicht ausgeführt werden kann.

Die Lösung für dieses Problem besteht darin, eine virtuelle Umgebung zu erstellen, einen eigenständigen Verzeichnisbaum, der eine Python-Installation für eine bestimmte Version von Python sowie eine Reihe weiterer Pakete enthält.


Verschiedene Anwendungen können dann unterschiedliche virtuelle Umgebungen nutzen. Um das frühere Beispiel für widersprüchliche Anforderungen zu lösen, kann Anwendung A eine eigene virtuelle Umgebung mit der Version 1.0 installiert haben, während Anwendung B eine andere virtuelle Umgebung mit der Version 2.0 hat. Wenn Anwendung B ein Upgrade einer Bibliothek auf Version 3.0 erfordert, hat dies keinen Einfluss auf die Umgebung von Anwendung A.

12.2. Virtuelle Umgebungen erschaffen

Das Modul zum Erstellen und Verwalten virtueller Umgebungen heißt `venv`. `venv` wird in der Regel die neueste Version von Python installieren, die zur Verfügung steht. Wenn du mehrere Versionen von Python auf deinem System hast, kannst du eine bestimmte Python-Version auswählen, indem du `python3` ausführst oder welche Version du willst.

Um eine virtuelle Umgebung zu erstellen, entscheidest du dich für ein Verzeichnis, in dem du es platzieren möchtest, und führst das `venv`-Modul als Skript mit dem Verzeichnispfad aus:

```
python3 -m venv tutorial-env
```



Dadurch wird das Verzeichnis `tutorial-env` erstellt, wenn es nicht existiert, und es werden auch Verzeichnisse darin erstellt, die eine Kopie des Python-Interpreters, der Standardbibliothek und verschiedener unterstützender Dateien enthalten.

Sobald du eine virtuelle Umgebung erstellt hast, kannst du sie aktivieren.

Unter Windows führst du Folgendes aus:

```
tutorial-env\Scripts\activate.bat
```

Unter Unix oder MacOS, starte:


```
source tutorial-env/bin/activate
```

(Dieses Skript ist für die Bash-Shell geschrieben. Wenn du die Csh- oder fish shells verwendest, gibt es alternativ *activate.csh* und *activate.fish* Skripte, die du stattdessen verwenden solltest.) Die Aktivierung der virtuellen Umgebung ändert die Eingabeaufforderung deiner Shell, um anzuzeigen, welche virtuelle Umgebung du verwendest, und ändert die Umgebung, sodass du mit dem Befehl *python* diese spezielle Version und Installation von Python erhältst. Zum Beispiel:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36)
...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

12.3. Verwaltung von Paketen mit pip

Du kannst Pakete mit einem Programm namens *pip* installieren, aktualisieren und entfernen. Standardmäßig installiert der Befehl *pip* Pakete aus dem Python Package Index, <<https://pypi.org>>. Du kannst den Python Package Index durchsuchen, indem du ihn in deinem Webbrowser öffnest, oder du kannst die eingeschränkte Suchfunktion von *pip* verwenden:



```
(tutorial-env) $ pip search astronomy
skyfield          - Elegant astronomy for Python
gary              - Galactic astronomy and gravitational dynamics.
novas             - The United States Naval Observatory NOVAS astronomy
library
astroobs          - Provides astronomy ephemeris to plan telescope
observations
PyAstronomy       - A collection of astronomy related tools for Python.
...
```

pip hat eine Reihe von Unterbefehlen: "search", "install", "uninstall", "freeze", etc. (siehe Anleitung zur Installation von Python-Modulen für eine vollständige Dokumentation für Pip.) Du kannst die neueste Version eines Package installieren, indem du den Namen eines Package angibst:


```
(tutorial-env) $ pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Du kannst auch eine bestimmte Version eines Package installieren, indem du den Packagenamen gefolgt von == und der Versionsnummer angibst:

```
(tutorial-env) $ pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Wenn du diesen Befehl erneut ausführst, wird pip feststellen, dass die gewünschte Version bereits installiert ist und nichts tun. Du kannst eine andere Versionsnummer angeben, um diese





Version zu erhalten, oder du kannst `pip install --upgrade requests` ausführen, um das Package auf die neueste Version zu aktualisieren:

```
(tutorial-env) $ pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

pip uninstall gefolgt von einem oder mehreren Packagenamen entfernt die Packages aus der virtuellen Umgebung.


pip show zeigt Informationen über ein bestimmtes Package an:

```
(tutorial-env) $ pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

pip list zeigt alle in der virtuellen Umgebung installierten Packages an:

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```





pip freeze erzeugt eine ähnliche Liste der installierten Packages, aber die Ausgabe verwendet das Format, das *pip install* erwartet. Eine gängige Konvention ist es, diese Liste in eine *requirements.txt*-Datei zu schreiben:

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

Die *requirements.txt* kann dann der Versionskontrolle übergeben und als Teil einer Anwendung mitgeliefert werden. Benutzer können dann alle notwendigen Packages mit *install -r* installieren:

```
(tutorial-env) $ pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

pip hat viele weitere Optionen. In der Anleitung zur Installation von Python-Modulen findest du eine vollständige Dokumentation für Pipelines. Wenn du ein Package geschrieben hast und es im Python-Packageindex verfügbar machen möchtest, beachte die Anleitung Verteilende Python-Module.

13. Was nun?

Das Lesen dieses Tutorials hat dein Interesse an der Verwendung von Python wahrscheinlich verstärkt - du solltest darauf bedacht sein, Python zur Lösung deiner Probleme in der Praxis einzusetzen. Wo solltest du hingehen, um mehr zu erfahren?

Dieses Tutorial ist Teil der Dokumentation von Python. Einige andere Dokumente im Set sind:

- [The Python Standard Library](#):

Du solltest dieses Handbuch durchstöbern, das vollständiges (wenn auch kurzes) Referenzmaterial über Typen, Funktionen und die Module der Standardbibliothek enthält. Die Standard-Python-Distribution enthält eine Menge zusätzlichen Code. Es gibt Module zum Lesen von Unix-Postfächern, Abrufen von Dokumenten über HTTP, Erzeugen von Zufallszahlen, Parsen von Befehlszeilenoptionen, Schreiben von CGI-Programmen, Komprimieren von Daten und viele andere Aufgaben. Wenn du durch die Bibliotheksreferenz blätterst, bekommst du eine Vorstellung davon, was verfügbar ist.

- [Installing Python Modules](#) erklärt, wie man zusätzliche Module installiert, die von anderen Python-Benutzern geschrieben wurden.
- [The Python Language Reference](#): Eine detaillierte Erklärung der Syntax und Semantik von Python. Es ist schwer zu lesen, aber es ist nützlich als kompletter Leitfaden für die Sprache selbst.

Mehr Python Ressourcen:

- <https://www.python.org>: Die wichtigste Python-Website. Sie enthält Code, Dokumentation und Verweise auf Python-bezogene Seiten im Web. Diese Website wird an verschiedenen Orten auf der ganzen Welt gemirrt, z.B. in Europa, Japan und Australien; ein Mirror kann je nach deinem geografischen Standort schneller sein als die Hauptseite.
- <https://docs.python.org>: Schneller Zugriff auf die Dokumentation von Python.
- <https://pypi.org>: Der Python Package Index, früher auch Cheese Shop genannt, ist ein Index von benutzerdefinierten Python-Modulen, die zum Download zur Verfügung stehen. Sobald du anfängst, den Code freizugeben, kannst du ihn hier registrieren, damit andere ihn finden können.

- 
- <https://code.activestate.com/recipes/langs/python/>: Das Python Cookbook ist eine umfangreiche Sammlung von Codebeispielen, größeren Modulen und nützlichen Skripten. Besonders bemerkenswerte Beiträge sind in einem Buch mit dem Titel Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.) zusammengefasst.
 - <http://www.pyvideo.org> sammelt Links zu Python-bezogenen Videos von Konferenzen und Usergruppentreffen.
 - <https://scipy.org>: Das wissenschaftliche Python-Projekt beinhaltet Module für schnelle Array-Berechnungen und -Manipulationen sowie eine Vielzahl von Paketen für Dinge wie lineare Algebra, Fourier-Transformationen, nichtlineare Solver, Zufallszahlenverteilungen, statistische Analysen und dergleichen.

Python-bezogene Fragen und Problembereiche kannst du in der Newsgroup comp.lang.python posten oder sie an die Mailingliste unter python-list@python.org senden. Die Newsgroup und die Mailingliste werden über ein Gateway verbunden, sodass Nachrichten, die an die eine Gruppe gesendet werden, automatisch an die andere weitergeleitet werden. Es gibt Hunderte von Beiträgen pro Tag, die Fragen stellen (und beantworten), neue Funktionen vorschlagen und neue Module ankündigen. Archive von Mailinglisten sind verfügbar unter <https://mail.python.org/pipermail/>.

Bevor du etwas veröffentlichst, solltest du unbedingt die Liste der häufig gestellten Fragen (auch FAQ genannt) lesen. Die FAQ beantworten viele der immer wieder auftauchenden Fragen und können bereits die Lösung für dein Problem enthalten.





14. Interaktive Eingabebearbeitung und History Substitution

Einige Versionen des Python-Interpreters unterstützen die Bearbeitung der aktuellen Eingabezeile und die History-Substitution, ähnlich wie in der Korn-Shell und der GNU Bash-Shell. Dies wird mit Hilfe der GNU Readline Bibliothek realisiert, die verschiedene Bearbeitungsstile unterstützt. Diese Bibliothek hat ihre eigene Dokumentation, die wir hier nicht duplizieren werden.

14.1. Vervollständigung der Registerkarte und Bearbeitung der Historie

Die Vervollständigung von Variablen- und Modelnamen wird beim Start des Interpreters automatisch aktiviert, so dass die Tab-Taste die Vervollständigungsfunktion aufruft; sie betrachtet Python-Anweisungsnamen, die aktuellen lokalen Variablen und die verfügbaren Modelnamen. Bei gestrichelten Ausdrücken wie *string.a* wird der Ausdruck bis zum letzten '.' ausgewertet und dann Vervollständigungen aus den Attributen des resultierenden Objekts vorgeschlagen. Beachte, dass dies anwendungsdefinierten Code ausführen kann, wenn ein Objekt mit einer `__getattr__()` Methode Teil des Ausdrucks ist. Die Standardkonfiguration speichert deinen Verlauf auch in einer Datei namens `.python_history` in deinem Benutzerverzeichnis. Der Verlauf wird bei der nächsten interaktiven Dolmetscher-Sitzung wieder verfügbar sein.

14.2. Alternativen zum interaktiven Interpreter

Diese Einrichtung ist ein enormer Fortschritt gegenüber früheren Versionen des Interpreters, allerdings bleiben einige Wünsche offen: Es wäre schön, wenn die richtige Einrückung in Fortsetzungszeilen vorgeschlagen würde (der Parser weiß, ob als nächstes ein Einrückungs-Token benötigt wird). Der Vervollständigungsmechanismus kann die Symboltabelle des Interpreters verwenden. Ein Befehl zum Überprüfen (oder sogar Vorschlagen) passender Klammern, Anführungszeichen usw. wäre ebenfalls nützlich.

Ein alternativer erweiterter interaktiver Interpreter, der seit geraumer Zeit auf dem Markt ist, ist IPython, welches die Vervollständigung von Registerkarten, die Erforschung von Objekten und die erweiterte Verlaufsverwaltung bietet. Es kann auch vollständig angepasst und in andere Anwendungen eingebettet werden. Eine weitere ähnlich erweiterte interaktive Umgebung ist bpython.



15. Gleitkommaarithmetik: Probleme und Einschränkungen

Gleitkommazahlen werden in der Computerhardware als Basis 2 (binäre) Brüche dargestellt. Zum Beispiel hat der Dezimalbruch

0.125

den Wert $1/10 + 2/100 + 5/1000$, und in gleicher Weise hat der binäre Bruch

0.001

den Wert $0/2 + 0/4 + 1/8$. Diese beiden Brüche haben identische Werte, wobei der einzige wirkliche Unterschied darin besteht, dass der erste zur Basis 10 und der zweite zur Basis 2 geschrieben wird.

Leider können die meisten Dezimalbrüche nicht exakt als Binärbrüche dargestellt werden. Dies hat zur Folge, dass die von dir eingegebenen dezimalen Gleitkommazahlen im Allgemeinen nur annähernd durch die tatsächlich in der Maschine gespeicherten binären Gleitkommazahlen bestimmt werden.

Das Problem ist zunächst in Basis 10 leichter zu verstehen. Betrachte den Bruch $1/3$. Du kannst das als Dezimalbruch schätzen:


0.3

oder, besser,

0.33

oder, besser,

0.333



und so weiter. Egal wie viele Stellen du aufschreibst, das Ergebnis wird nie genau $1/3$ sein, sondern eine immer bessere Annäherung an $1/3$.

Ebenso kann der Dezimalwert 0.1 nicht exakt als binärer Bruch dargestellt werden, unabhängig davon, wie viele binäre Ziffern du verwendest. Zur Basis 2 ist $1/10$ der sich unendlich wiederholende Bruch

```
0.000110011001100110011001100110011001100110011001100110011...
```

Halte bei einer beliebigen endlichen Anzahl von Bits an und du erhältst eine Näherung. Auf den meisten heutigen Maschinen werden Floats mit einem binären Bruchteil approximiert, wobei der Zähler die ersten 53 Bits verwendet, beginnend mit dem höchstwertigen Bit und mit dem Nenner als Zweierpotenz. Im Falle von $1/10$ ist der binäre Anteil $3602879701896397 / 2^{55}$, der nahe, aber nicht genau gleich dem wahren Wert von $1/10$ ist.

Vielen Anwendern ist die Approximation aufgrund der Art und Weise, wie Werte angezeigt werden, nicht bekannt. Python druckt nur eine dezimale Annäherung an den wahren Dezimalwert der von der Maschine gespeicherten binären Annäherung. Wenn Python auf den meisten Maschinen den wahren Dezimalwert der für 0.1 gespeicherten binären Näherung ausgeben würde, müsste es Folgendes anzeigen


```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Das sind mehr Ziffern, als die meisten Leute nützlich finden, so dass Python die Anzahl der Ziffern überschaubar hält, indem es stattdessen einen gerundeten Wert anzeigt.

```
>>> 1 / 10
0.1
```

Denke nur daran, dass, obwohl das gedruckte Ergebnis genau wie der Wert von $1/10$ aussieht, der tatsächlich gespeicherte Wert der nächstgelegene darstellbare Binäranteil ist.

Interessanterweise gibt es viele verschiedene Dezimalzahlen, die sich den gleichen binären Näherungsbruch teilen. So werden beispielsweise die Zahlen 0.1 und 0.1000000000000000000000000000001 und



```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Obwohl die Zahlen nicht näher an ihre beabsichtigten exakten Werte herangeführt werden können, kann die Funktion `round()` für die Nachrundung nützlich sein, so dass Ergebnisse mit ungenauen Werten miteinander vergleichbar werden:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```


Wie es am Ende heißt: "Es gibt keine einfachen Antworten." Aber sei nicht zu vorsichtig mit dem Gleitkomma! Die Fehler in Python-Float-Operationen werden von der Gleitkomma-Hardware vererbt, und auf den meisten Maschinen liegen sie in der Größenordnung von nicht mehr als einem Teil von 2^{53} pro Operation. Das ist für die meisten Aufgaben mehr als ausreichend, aber du musst bedenken, dass es keine dezimale Arithmetik ist und dass jede Float-Operation einen neuen Rundungsfehler erleiden kann.

Obwohl es pathologische Fälle gibt, siehst du für die meiste beiläufige Verwendung der Gleitkomma-Arithmetik das Ergebnis, das du am Ende erwartest, wenn du die Anzeige deiner Endergebnisse einfach auf die Anzahl der Dezimalstellen rundest, die du erwartest. `str()` genügt in der Regel, und für eine feinere Kontrolle siehe die Formatangaben der `str.format()`-Methode in der Format-String-Syntax.

Für Anwendungsfälle, die eine genaue Dezimaldarstellung erfordern, verwendest du das Modul `decimal`, das die Dezimalarithmetik für Buchhaltungsanwendungen und hochpräzise Anwendungen implementiert.

Eine weitere Form der exakten Arithmetik wird durch das Modul `fractions` unterstützt, das die Arithmetik basierend auf rationalen Zahlen implementiert (so können die Zahlen wie $1/3$ exakt dargestellt werden).

Wenn du ein starker Anwender von Gleitkommaoperationen sind, solltest du einen Blick auf das Numerical Python-Paket und viele andere Pakete für mathematische und statistische Operationen werfen, die vom SciPy-Projekt bereitgestellt werden. Siehe <https://scipy.org>.



Python bietet Werkzeuge, die in den seltenen Fällen helfen können, in denen du wirklich den genauen Wert eines Floats wissen willst. Die Methode `float.as_integer_ratio()` drückt den Wert eines Floats als Bruch aus:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Da das Verhältnis genau ist, kann es verwendet werden, um den ursprünglichen Wert verlustfrei wiederherzustellen:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

Die Methode `float.hex()` drückt einen Float in hexadezimaler Form (Basis 16) aus und gibt dabei wiederum genau den von Ihrem Computer gespeicherten Wert an:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Diese präzise hexadezimale Darstellung kann verwendet werden, um den Float-Wert exakt zu rekonstruieren:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Da die Darstellung exakt ist, ist sie nützlich, um Werte über verschiedene Versionen von Python hinweg zuverlässig zu portieren (Plattformunabhängigkeit) und Daten mit anderen Sprachen auszutauschen, die das gleiche Format unterstützen (z.B. Java und C99).

Ein weiteres hilfreiches Werkzeug ist die Funktion `math.fsum()`, die hilft, den Genauigkeitsverlust bei der Summierung zu minimieren. Es verfolgt "verlorene Ziffern", wenn Werte zu einer laufenden Summe addiert werden. Das kann einen Unterschied in der Gesamtgenauigkeit



machen, so dass sich die Fehler nicht bis zu dem Punkt ansammeln, an dem sie die Endsumme beeinflussen:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1. Darstellungsfehler

In diesem Abschnitt wird das Beispiel "0.1" ausführlich erläutert und gezeigt, wie du eine genaue Analyse solcher Fälle selbst durchführen kannst. Grundlegende Kenntnisse der binären Gleitkomma-Darstellung werden vorausgesetzt.

Der Darstellungsfehler bezieht sich auf die Tatsache, dass einige (die meisten, tatsächlich) Dezimalbrüche nicht genau als binäre Brüche dargestellt werden können. Dies ist der Hauptgrund, warum Python (oder Perl, C, C++, Java, Fortran und viele andere) oft nicht genau die Dezimalzahl anzeigt, die du erwartest.

Warum ist das so? $1/10$ ist nicht genau als binärer Bruch darstellbar. Fast alle heutigen Maschinen (November 2000) verwenden IEEE-754 Fließkomma-Arithmetik, und fast alle Plattformen bilden Python-Floats auf IEEE-754 "double precision" ab. 754-Doppel enthalten 53 Bits Genauigkeit, sodass der Computer bei der Eingabe danach strebt, 0.1 in den nächstgelegenen Bruch der Form $J/2^{**N}$ zu konvertieren, wobei J eine ganze Zahl mit genau 53 Bits ist. Wir schreiben

$$1 / 10 \approx J / (2^{**N})$$

als

$$J \approx 2^{**N} / 10$$

und unter Hinweis darauf, dass J genau 53 Bit hat (ist $\geq 2^{**52}$, aber $< 2^{**53}$), ist der beste Wert für N 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
```





```
True
```

Das heißt, 56 ist der einzige Wert für N, der J mit genau 53 Bit zurücklässt. Der bestmögliche Wert für J ist dann dieser Quotient gerundet:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Da der Rest mehr als die Hälfte von 10 beträgt, wird die beste Annäherung durch Aufrunden erreicht:

```
>>> q+1
7205759403792794
```

Daher ist die bestmögliche Annäherung an $1/10$ in 754 doppelte Genauigkeit:

```
7205759403792794 / 2 ** 56
```

Wenn sowohl der Zähler als auch der Nenner durch zwei geteilt werden, reduziert sich der Anteil auf:

```
3602879701896397 / 2 ** 55
```

Merke, dass, da wir aufgerundet haben, diese tatsächlich etwas größer als $1/10$ ist; wenn wir nicht aufgerundet hätten, wäre der Quotient etwas kleiner als $1/10$ gewesen. Aber auf keinen Fall darf es genau $1/10$ sein!

Der Computer "sieht" also nie $1/10$: Was er sieht, ist genau der oben angegebene Bruchteil, die beste 754 doppelte Annäherung, die er erhalten kann:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```



Wenn wir diesen Bruch mit 10^{55} multiplizieren, können wir den Wert auf 55 Dezimalstellen sehen:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
10000000000000000055511151231257827021181583404541015625
```

was bedeutet, dass die genaue im Computer gespeicherte Zahl gleich dem Dezimalwert 0.1000000000000000000000000055511151231257827021181583404541015625 ist. Anstatt den vollen Dezimalwert anzuzeigen, runden viele Sprachen (einschließlich älterer Versionen von Python) das Ergebnis auf 17 signifikante Stellen:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Die Module `fractions` und `decimal` machen diese Berechnungen einfach:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17f')
'0.10000000000000001'
```





16. Anhang

16.1. Interaktiver Modus

16.1.1. Fehlerbehandlung

Wenn ein Fehler auftritt, gibt der Interpreter eine Fehlermeldung und eine Stackverfolgung aus. Im interaktiven Modus kehrt er dann zur primären Eingabeaufforderung zurück; wenn die Eingabe aus einer Datei erfolgt, wird sie nach dem Drucken der Stackverfolgung mit einem Ausgangsstatus ungleich Null beendet. (Ausnahmen, die von einer `except`-Anweisung in einer `try`-Anweisung behandelt werden, sind in diesem Zusammenhang keine Fehler.) Einige Fehler sind bedingungslos fatal und führen zu einem Ausgang mit einem „nonzero exit“; dies geschieht durch interne Inkonsistenzen und in einigen Fällen von Speicherauslastung. Alle Fehlermeldungen werden in den Standardfehlerstrom geschrieben; die normale Ausgabe von ausgeführten Befehlen wird in die Standardausgabe geschrieben.


Wenn du das Interrupt-Zeichen (normalerweise Control-C oder Delete) für die primäre oder sekundäre Eingabeaufforderung eingibst, wird die Eingabe abgebrochen und zur primären Eingabeaufforderung zurückgekehrt. [1] Die Eingabe eines Interrupts während der Ausführung eines Befehls löst die Exception *KeyboardInterrupt* aus, die von einer Try-Anweisung behandelt werden kann.

16.1.2. Ausführbare Python-Skripte

Auf BSD'schen Unix-Systemen können Python-Skripte direkt ausführbar gemacht werden, wie Shell-Skripte, indem man die Zeile

```
#!/usr/bin/env python3.5
```

(vorausgesetzt, dass sich der Interpreter im PATH des Benutzers befindet) am Anfang des Skripts einfügt und der Datei einen ausführbaren Modus hinzufügt. Das `#!` muss die ersten beiden Zeichen der Datei sein. Auf einigen Plattformen muss diese erste Zeile mit einem Unix-ähnlichen Zeilenende (`\n`) enden, nicht mit einem Windows (`\r\n`) Zeilenende. Bitte bedenke, dass das Kreuz oder Hashzeichen `#` verwendet wird, um einen Kommentar in Python zu beginnen.



Das Skript kann mit dem Befehl `chmod` einen ausführbaren Modus oder eine Berechtigung erhalten.

```
$ chmod +x myscript.py
```

Auf Windows-Systemen gibt es keine Vorstellung von einem "ausführbaren Modus". Der Python-Installer ordnet `.py`-Dateien automatisch `python.exe` zu, so dass ein Doppelklick auf eine Python-Datei diese als Skript ausführt. Die Endung kann auch `.pyw` sein, in diesem Fall wird das normalerweise erscheinende Konsolenfenster unterdrückt.

16.1.3. Die interaktive Startdatei

Wenn du Python interaktiv verwendest, ist es oft sinnvoll, bei jedem Start des Interpreters einige Standardbefehle ausführen zu lassen. Dazu setzt man eine Umgebungsvariable namens `PYTHONSTARTUP` auf den Namen einer Datei, die die Startbefehle enthält. Dies ist vergleichbar mit der `.profile`-Funktion der Unix-Shells.

Diese Datei wird nur in interaktiven Sitzungen gelesen, nicht, wenn Python Befehle aus einem Skript liest, und nicht, wenn `/dev/tty` als explizite Befehlsquelle angegeben wird (die sich ansonsten wie eine interaktive Sitzung verhält). Sie wird im gleichen Namensraum ausgeführt, in dem auch interaktive Befehle ausgeführt werden, sodass von ihr definierte oder importierte Objekte ohne Einschränkung in der interaktiven Sitzung verwendet werden können. Du kannst auch die Prompts `sys.ps1` und `sys.ps2` in dieser Datei ändern.

Wenn du eine zusätzliche Startdatei aus dem aktuellen Verzeichnis lesen möchtest, kannst du diese in der globalen Startdatei mit Code programmieren, wie z.B. `os.path.isfile('.pythonrc.py')`: `exec(open('.pythonrc.py').read())`. Wenn du die Startdatei in einem Skript verwenden möchtest, musst du dies explizit im Skript tun:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
        exec(startup_file)
```

16.1.4. Die Anpassungsmodule

Python bietet zwei Hooks, mit denen du es anpassen kannst: *sitecustomize* und *usercustomize*. Um zu sehen, wie es funktioniert, musst du zuerst den Speicherort deines Verzeichnisses für Benutzer-Site-Packages finden. Beginne Python und führe diesen Code aus:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.5/site-packages'
```

Jetzt kannst du eine Datei namens *usercustomize.py* in diesem Verzeichnis erstellen und alles, was du willst, hineinlegen. Es wirkt sich auf jeden Aufruf von Python aus, es sei denn, es wird mit der Option *-s* gestartet, um den automatischen Import zu deaktivieren.

sitecustomize funktioniert auf die gleiche Weise, wird aber normalerweise von einem Administrator des Computers im globalen Site-Packages-Verzeichnis erstellt und vor der Benutzeranpassung importiert. Weitere Informationen findest du in der Dokumentation des Moduls *site*.

Fußnoten

[1] Ein Problem mit dem GNU Readline-Package kann dies verhindern.