# Lerne Python, indem du Textspiele entwickels

Alexander (Sasha) Pastukhov

2023-10-25

# Contents

# Chapter 1

# Einleitung

Dieses Buch wird dich das Programmieren lehren. Hoffentlich auf eine spaßige Art und Weise, denn wenn es etwas Befriedigenderes gibt als ein Videospiel zu spielen, dann ist es, eines zu erstellen. Obwohl es für den Kurs *"Python für soziale und experimentelle Psychologie"* geschrieben wurde, ist mein Hauptziel nicht, dir Python an sich beizubringen. Python ist ein fantastisches Werkzeug (mehr dazu später), aber es ist nur eine von vielen existierenden Programmiersprachen. Mein ultimatives Ziel ist es, dir dabei zu helfen, allgemeine Programmierfähigkeiten zu entwickeln, die nicht von einer spezifischen Programmiersprache abhängen, und sicherzustellen, dass du gute Gewohnheiten entwickelst, die deinen Code klar, leicht lesbar und leicht zu warten machen. Dieser letzte Teil ist entscheidend. Programmieren geht nicht nur darum, Code zu schreiben, der funktioniert. Das muss natürlich gegeben sein, ist aber nur die Mindestanforderung. Programmieren geht darum, einen klaren und leicht lesbaren Code zu schreiben, den andere und, noch wichtiger, du selbst zwei Wochen später verstehen kannst.

## 1.1 Ziel des Buches

Das Ziel ist, dass du in der Lage bist, ein anspruchsvolles Experiment zu programmieren, das mehrere Blöcke und Versuche, verschiedene Bedingungen, komplizierte visuelle Darstellungen, automatische Datenaufzeichnung und Fehlerbehandlung haben kann. Wenn das ambitioniert klingt, dann ist es das auch, aber wir werden in kleinen Schritten vorgehen und im Prozess wirst du lernen.

- Kernkonzepte von Python einschließlich
  - Variablen und Konstanten
  - unveränderliche Datentypen wie Ganzzahlen, Fließkommazahlen, Zeichenketten, logische Werte und Tupel
  - veränderliche Typen wie Listen und Dictionaries

  - Funktionen
  - Steuerungsstrukturen wie if-else-Bedingungen und Schleifen
  - objektorientierte Programmierung einschließlich Konzepte des Vererbung, (optional) Duck-Typing und Mischung
  - Ausnahmen
  - Dateioperationen
- PsychoPy: Dies ist nicht Kern-Python, aber es ist eine erstklassige Bibliothek für psychophysische Experimente und du wirst die Schlüsselwerkzeuge lernen, die für die Programmierung eines Experiments erforderlich sind.
- Guter Programmierstil einschließlich
  - Klaren Code in iterativer Weise schreiben
  - Lesen (deines eigenen) Codes
  - Dokumentieren deines Codes
  - Debuggen deines Programms in VS Code

Ich habe versucht, Konzepte in dem Kontext zu präsentieren, der sie erfordert und daher am besten erklärt und ihre typischen Anwendungsszenarien darstellt. Das bedeutet, dass das Material rund um verteilt ist und auf Bedarfsbasis präsentiert wird. Zum Beispiel wird das Konzept der Listen zuerst präsentiert, aber Operationen daran werden in einem späteren Kapitel vorgestellt, sowohl weil wir dies erst später benötigen als auch um dein Gefühl, überwältigt zu sein, in einem vernünftigen Rahmen zu halten. Dies macht es schwieriger, dieses Buch als Referenz zu verwenden (es gibt ausgezeichnete Referenzen da draußen, angefangen mit der offiziellen Python-Dokumentation), aber die Hoffnung ist, dass es dir durch die Bereitstellung von mundgerechten Informationsbrocken leichter fallen wird, das Material zu verstehen und es mit dem zu integrieren, was du bereits weißt.

Die gleiche "verteilte" Logik gilt für Übungen. Anstatt am Ende jedes Kapitels aufgelistet zu sein, sind sie in den Text eingebettet und du solltest sie zu diesem Zeitpunkt erledigen. Viele von ihnen dienen dazu, Konzepte zu verdeutlichen, die zuvor anhand von anschaulichen Beispielen präsentiert wurden, daher wäre es hilfreich, sie sofort durchzuführen. Gleiches gilt für die Programmierübung, obwohl du in diesem Fall das gesamte Material lesen kannst, um eine "Vogelperspektive" auf das gesamte Programm zu bekommen und dann den Text erneut zu lesen und die Programmierübung durchzuführen.

## 1.2 Voraussetzungen

Dieses Material setzt keine Vorkenntnisse in Python oder Programmierung beim Leser voraus. Sein Ziel ist es, dein Wissen schrittweise aufzubauen und dir zu ermöglichen, immer komplexere Spiele zu erstellen.

## 1.3 Warum Spiele?

Der eigentliche Zweck dieses Kurses ist es, Psychologie- und Sozialwissenschaftsstudenten beizubringen, wie man *Experimente* programmiert. Das ist es, worum es in der echten Forschung geht. Allerdings gibt es zwischen den beiden praktisch wenig Unterschied. Die grundlegenden Zutaten sind die gleichen und, man könnte sagen, Experimente sind einfach nur langweilige Spiele. Und sei versichert, wenn du ein Spiel programmieren kannst, kannst du sicherlich auch ein Experiment programmieren.

## 1.4 Warum sollte ein Psychologe Programmieren lernen?

Warum sollte ein Psychologe, der an Menschen interessiert ist, lernen, wie man Computer programmiert? Die offensichtlichste Antwort ist, dass dies eine nützliche Fähigkeit ist. Wenn du programmieren kannst, hast du die Freiheit, ein Experiment zu erstellen, das deine Forschungsfrage beantwortet, und nicht ein Experiment, das aufgrund der Beschränkungen deiner Software implementiert werden kann.

Noch wichtiger, zumindest aus meiner Sicht, ist, dass das Erlernen des Programmierens die Art und Weise, wie du im Allgemeinen denkst, verändert. Menschen sind klug, aber Computer sind dumm[1]. Wenn du dein Experiment oder deine Reisepläne jemandem erklärst, kannst du ziemlich vage sein, einen kleinen Fehler machen, sogar bestimmte Teile überspringen. Menschen sind klug, daher werden sie die fehlenden Informationen mit ihrem Wissen ergänzen, einen Fehler finden und korrigieren, dich nach mehr Informationen fragen und selbst improvisieren können, sobald sie auf etwas stoßen, das du nicht abgedeckt hast. Computer sind dumm, also musst du präzise sein, du kannst keine Grauzonen haben, du kannst nichts dem "es wird herausfinden, wenn es passiert" (es wird es nicht) überlassen. Meine persönliche Erfahrung, die von Psychologen bestätigt wurde, die das Programmieren gelernt haben, ist, dass es dir bewusst macht, wie vage und ungenau Menschen sein können, ohne es zu merken (und ohne dass du es merkst). Programmieren zwingt dich dazu, präzise und gründlich zu sein, für alle Eventualitäten zu planen, die es geben könnte. Und das ist an sich eine sehr nützliche Fähigkeit, da sie auf alle Aktivitäten angewendet werden kann, die Planung erfordern, sei es ein experimentelles Design oder Reisevorkehrungen.

## 1.5 Warum Python?

Es gibt viele Möglichkeiten, ein Experiment für psychologische Forschung zu erstellen. Du kannst Drag-and-Drop-Systeme verwenden, entweder kommerzielle

[1] Dies wurde geschrieben, bevor große Sprachmodelle aufkamen, aber trifft immer noch zu, wenn es ums Programmieren geht.

wie Presentation, Experiment Builder oder kostenlose wie PsychoPy Builder Interface. Sie haben eine viel flachere Lernkurve, sodass du schneller mit dem Erstellen und Durchführen deiner Experimente beginnen kannst. Ihre einfache Handhabung hat jedoch einen Preis: Sie sind recht begrenzt, welche Reize du verwenden kannst und wie du die Präsentation, Bedingungen, Feedback etc. steuern kannst. In der Regel erlauben sie dir, sie zu erweitern, indem du das gewünschte Verhalten programmierst, aber du musst wissen, wie man das macht (Python-Kenntnisse bereichern deine PsychoPy-Experimente). Daher denke ich, dass diese Systeme, insbesondere PsychoPy, großartige Werkzeuge sind, um schnell ein einfaches Experiment zusammenzusetzen. Sie sind jedoch am nützlichsten, wenn du verstehst, *wie* sie den zugrunde liegenden Code erstellen und wie du ihn selbst programmieren würdest. Dann wirst du nicht durch die Software eingeschränkt, da du weißt, dass du etwas programmieren kannst, was der Standard-Drag-and-Drop nicht zulässt. Gleichzeitig kannst du dich immer dafür entscheiden, wenn Drag-and-Drop ausreichend, aber schneller ist, oder du kannst eine Kombination aus beiden Ansätzen verwenden. Am Ende geht es darum, Optionen und kreative Freiheit zu haben, ein Experiment zu programmieren, das deine Forschungsfrage beantwortet, und nicht nur ein Experiment, das deine Software erlaubt zu programmieren.

Wir werden das Programmieren in Python lernen, eine großartige Sprache, die eine einfache und klare Syntax mit der Kraft und Fähigkeit kombiniert, fast jedes Problem zu bewältigen. In diesem Seminar konzentrieren wir uns auf Desktop-Experimente, aber Sie können es für Online-Experimente (oTree und PsychoPy), wissenschaftliche Programmierung (NumPy und SciPy), Datenanalyse (pandas), Maschinelles Lernen (scikit-learn), Deep Learning (keras), Website-Programmierung (django), Computer Vision (OpenCV) usw. verwenden. Daher ist Python eines der vielseitigsten Programmierwerkzeuge, die Sie für alle Phasen Ihrer Forschung oder Arbeit verwenden können. Und Python ist kostenlos, so dass Sie sich keine Sorgen machen müssen, ob Sie oder Ihr zukünftiger Arbeitgeber die Lizenzgebühren bezahlen können (ein sehr reales Problem, wenn Sie Matlab verwenden).

## 1.6 Seminar-spezifische Informationen

Dies ist ein Material für das Seminar *Python für Sozial- und Experimentelle Psychologie*, wie ich es an der Universität Bamberg lehre. Jedes Kapitel behandelt ein einzelnes Spiel und führt die notwendigen Ideen ein und wird von Übungen begleitet, die Sie absolvieren und einreichen müssen. Um das Seminar zu bestehen, müssen Sie alle Aufgaben absolvieren, d.h., alle Spiele schreiben. Sie müssen nicht alle Übungen abschließen oder korrekte Lösungen für *alle* Übungen liefern, um den Kurs zu bestehen und Informationen darüber, wie die Punkte für Übungen in eine tatsächliche Note (falls Sie eine benötigen) oder "Bestanden" umgerechnet werden, werden während des Seminars verfügbar sein.

Das Material ist so strukturiert, dass jedes Kapitel oder Kapitelabschnitt typis-

cherweise einer einzelnen Sitzung entspricht, außer für die abschließenden Kapitel, die auf komplexeren Spielen basieren und daher mehr Zeit in Anspruch nehmen. Wir sind jedoch alle verschieden, also arbeiten Sie in Ihrem eigenen Tempo, lesen Sie das Material und reichen Sie Aufgaben eigenständig ein. Ich werde für jede Aufgabe detailliertes Feedback geben und Ihnen die Möglichkeit bieten, Probleme anzugehen und erneut einzureichen, ohne Punkte zu verlieren. Beachten Sie, dass mein Feedback nicht nur die tatsächlichen Probleme mit dem Code abdeckt, sondern auch die Art und Weise, wie Sie die Lösung implementiert haben und wie sauber und gut dokumentiert Ihr Code ist. Denken Sie daran, unsere Aufgabe ist es nicht nur, zu lernen, wie man ein funktionierendes Spiel programmiert, sondern wie man einen schönen, klaren, leicht zu lesenden und zu wartenden Code schreibt[2].

Sehr wichtig: Zögern Sie nicht, Fragen zu stellen. Wenn ich das Gefühl habe, dass Sie die Informationen im Material übersehen haben, werde ich Sie auf die genaue Stelle hinweisen. Wenn Sie verwirrt sind, werde ich Sie sanft mit Fragen dazu anregen, Ihr eigenes Problem zu lösen. Wenn Sie mehr Informationen benötigen, werde ich diese liefern. Wenn Sie einfach mehr wissen wollen, fragen Sie und ich werde erklären, warum die Dinge so sind, wie sie sind, oder vorschlagen, was Sie lesen sollten. Wenn ich das Gefühl habe, dass Sie das Problem ohne meine Hilfe lösen können, werde ich Ihnen das sagen (obwohl ich wahrscheinlich trotzdem noch ein paar andeutende Fragen stellen würde).

## 1.7 Über das Material



[2]Gute Gewohnheiten! Bilden Sie gute Gewohnheiten! Danke, dass Sie diese unterschwellige Botschaft gelesen haben.

# Chapter 2

# Software

Für dieses Buch und das Seminar müssen wir installieren

- PsychoPy.
- IDE Ihrer Wahl. Meine Anleitungen werden für Visual Studio Code sein, das eine sehr gute Python-Unterstützung bietet.
- Jupyter Notebook zum Ausprobieren kleiner Code-Snippets.

Ich werde keine detaillierten Anleitungen zur Installation der notwendigen Software geben, sondern Sie eher auf die offiziellen Handbücher verweisen. Dies macht diesen Text zukunftssicherer, da sich spezifische Details leicht ändern könnten[1].

## 2.1 PsychoPy

Wenn Sie Windows verwenden, laden Sie die Standalone PsychoPy Version herunter und installieren Sie diese. Verwenden Sie die neueste (und beste) Ihnen vorgeschlagene PsychoPy-Version (PsychoPy 2023.2.2 mit Python 3.8 zum Zeitpunkt des Schreibens) und folgen Sie den Anweisungen.

Wenn Sie Mac oder Linux verwenden, sind die Installation von PsychoPy über pip oder Anaconda Ihre Optionen. Bitte folgen Sie den aktuellen Anweisungen.

## 2.2 VS Code

Visual Studio Code ist ein kostenloser, leichtgewichtiger Open-Source-Editor mit starker Unterstützung für Python. Laden Sie den Installer für Ihre Plattform herunter und folgen Sie den Anweisungen.

[1] Wenn Sie Teil des Seminars sind, fragen Sie mich, wann immer Sie Probleme haben oder unsicher sind, wie Sie vorgehen sollen

Befolgen Sie als nächstes das Tutorial Getting Started with Python in VS Code. Wenn Sie Windows und die Standalone-Installation von PsychoPy verwenden, **überspringen** Sie den Abschnitt *Install a Python interpreter*, da Sie bereits eine Python-Installation haben, die mit PsychoPy gebündelt ist. Dies ist der Interpreter, den Sie im Abschnitt *Select a Python interpreter* verwenden sollten. In meinem Fall ist der Pfad `C:\Program Files\PsychoPy3\python.exe`.

Installieren und aktivieren Sie einen Linter, eine Software, die syntaxtische und stilistische Probleme in Ihrem Python-Quellcode hervorhebt. Folgen Sie dem Handbuch auf der Webseite von VS Code.

## 2.3 Jupyter Notebooks

Jupyter Notebooks bieten eine sehr bequeme Möglichkeit, Text, Bilder und Code in einem einzigen Dokument zu mischen. Sie erleichtern auch das Ausprobieren verschiedener kleiner Code-Snippets parallel ohne das Ausführen von Skripten. Wir werden uns für unser erstes Kapitel und gelegentliche Übungen oder Code-Tests später darauf verlassen. Es gibt zwei Möglichkeiten, wie Sie sie verwenden können: 1) in VS Code mit der Jupyter-Erweiterung, 2) in Ihrem Browser mit der klassischen Oberfläche.

### 2.3.1 Jupyter Notebooks in VS Code

Folgen Sie der Anleitung, wie Sie das Jupyter-Paket installieren und Notebooks in VS Code verwenden.

### 2.3.2 Jupyter Notebooks in Anaconda

Die einfachste Möglichkeit, Jupyter Notebooks zusammen mit vielen anderen nützlichen Data-Science-Tools zu verwenden, ist über das Anaconda Toolkit. Beachten Sie jedoch, dass dies eine *zweite* Python-Distribution in Ihrem System installiert. Dies könnte wiederum zu Verwirrung führen, wenn Sie mit Skripten in VS Code arbeiten und versehentlich den Anaconda-Interpreter statt den PsychoPy-Interpreter aktiv haben. Keine Panik, folgen Sie den Select a Python interpreter Anweisungen und stellen Sie sicher, dass Sie den PsychoPy-Interpreter als den aktiven haben.

Ansonsten laden Sie Anaconda herunter und installieren Sie es. Die Website hat einen ausgezeichneten Getting started Abschnitt.

## 2.4 Ordnung halten

Bevor wir anfangen, schlage ich vor, dass Sie einen Ordner namens *games-with-python* (oder so ähnlich) erstellen. Wenn Sie sich dafür entschieden haben, Jupyter Notebooks über Anaconda zu nutzen, sollten Sie diesen Ordner in Ihrem

Benutzerordner erstellen, da Anaconda dort die Dateien erwartet. Dann erstellen Sie einen neuen Unterordner für jedes Kapitel / Spiel. Für das Seminar müssten Sie einen Ordner mit allen Dateien zippen und hochladen.

# Chapter 3

# Programmier-Tipps und -Tricks

Bevor Sie Ihren ersten Code schreiben, müssen wir über die Kunst des Programmierens sprechen. Wie ich bereits erwähnt habe, geht es nicht nur darum, dass der Code funktioniert, sondern darum, dass er leicht zu verstehen ist. Korrekt funktionierender Code ist ein schönes Plus, aber wenn ich zwischen einem Spaghetti-Code, der momentan korrekt funktioniert, und einem klar geschriebenen und dokumentierten Code, der noch repariert werden muss, wählen muss, werde ich immer Letzteren bevorzugen. Ich kann Dinge reparieren, die ich verstehe, ich kann nur hoffen, wenn ich das nicht tue.

Unten sind einige Tipps zum Schreiben und Lesen von Code. Einige mögen beim ersten Lesen kryptisch klingen (sie werden klar, sobald wir das notwendige Material behandeln). Einige werden sich für die einfachen Projekte, die wir umsetzen werden, als Overkill anfühlen. Ich schlage vor, dass Sie diesen Abschnitt beim ersten Mal ungezwungen lesen, aber oft darauf zurückkommen, sobald wir ernsthaft mit Programmieren beginnen. Leider werden diese Tricks nicht funktionieren, wenn Sie sie nicht benutzen! Daher sollten Sie sie *immer* benutzen und sie sollten Ihre *guten Gewohnheiten* werden, wie das Anlegen eines Sicherheitsgurtes. Der Sicherheitsgurt ist an den meisten (hoffentlich allen) Tagen nicht nützlich, aber Sie tragen ihn, weil er plötzlich und sehr dringend extrem nützlich werden könnte und Sie nie sicher sein können, wann das passieren wird. Gleiches gilt für das Programmieren. Oft werden Sie versucht sein, einen "quick-n-dirty" Code zu schreiben, weil es sich nur um einen "einfachen Test", eine temporäre Lösung, einen Prototyp, ein Pilotexperiment, usw. handelt. Aber wie man so schön sagt: "Es gibt nichts Beständigeres als eine provisorische Lösung". Häufiger als nicht werden Sie feststellen, dass Ihr Spielzeugcode zu einem ausgewachsenen Experiment herangewachsen ist und es ein Durcheinander ist. Oder Sie möchten zu dem Pilotexperiment zurückkehren, das Sie vor

ein paar Monaten durchgeführt haben, aber Sie stellen fest, dass es einfacher ist, von vorne zu beginnen, als zu verstehen, wie dieses Monster funktioniert[1]. Widerstehen Sie also der Versuchung! Bilden Sie gute Gewohnheiten und Ihr zukünftiges Ich wird sehr dankbar sein!

## 3.1 Den Code schreiben

### 3.1.1 Verwende einen Linter

Ein Linter ist ein Programm, das deinen Code-Stil analysiert und highlightet Probleme, die es findet: unnötige Leerzeichen, fehlende Leerzeichen, falsche Namen, übermäßig lange Zeilen, usw. Diese haben keinen Einflluss darauf, wie der Code ausgeführt wird, aber das Befolgen der Linter-Ratschläge führt zu einem konsistenten Standard, wenn auch langweiligen[^ "Langweilig ist gut!", siehe den Film "The Hitman's Bodyguard."] Python Code. Versuche, alle Probleme zu beheben, die der Linter aufgeworfen hat. Verwende jedoch dein besseres Urteilsvermögen, denn manchmal sind Zeilen, die länger sind als der Linter es bevorzugen würde, lesbarer als zwei kürzere. Ebenso kann ein "schlechter" Variablenname nach den Standards des Linters für einen Psychologen eine aussagekräftige Bezeichnung sein. Denke daran, dein Code ist für Menschen, nicht für den Linter.

### 3.1.2 Dokumentiere deinen Code

Jedes Mal, wenn du eine neue Datei erstellst: dokumentiere sie und aktualisiere die Dokumentation, wann immer du neue Funktionen oder Klassen hinzufügst/änderst/löschst. Jedes Mal, wenn du eine neue Funktion erstellst: dokumentiere sie. Neue Klasse: dokumentiere sie. Neue Konstante: es sei denn, es ist alleine aus dem Namen klar, dokumentiere sie. Du wirst eine NumPy Methode der Dokumentation in dem Buch lernen.

Ich kann nicht genug betonen, wie wichtig es ist, deinen Code zu dokumentieren. VS Code (ein Editor, den wir verwenden werden) ist intelligent genug, um NumPy Docstrings zu parsen, also wird es dir diese Hilfe anzeigen, wann immer du deine eigenen Funktionen verwendest (hilft dir, dir selbst zu helfen!). Noch wichtiger ist, dass das Schreiben von Dokumentationen dich dazu zwingt, in menschlicher Sprache zu denken und zu formulieren, was die Funktion oder Klasse macht, welchen Typ die Argumente / Attribute / Methoden haben, was der Bereich der gültigen Werte ist, was die Standards sind, was eine Funktion zurückgeben sollte usw. Mehr als oft nicht, wirst du feststellen, dass du ein wichtiges Detail übersehen hast, das aus dem Code selbst nicht ersichtlich ist.

[1] Mir ist das öfter passiert, als ich zugeben möchte.

### 3.1.3 Füge etwas Luft hinzu

Trenne Codeblöcke durch einige leere Zeilen. Denke an Absätze im normalen Text. Du würdest nicht wollen, dass dein Buch ein einziger Absatz-Alptraum ist? Platziere vor jedem Codeblock einen Kommentar, der erklärt, *was* er macht, aber nicht *wie* er es macht. Zum Beispiel wird es in unserem typischen PsychoPy-basierten Spiel einen Punkt geben, an dem wir alle Reize zeichnen und das Fenster aktualisieren. Das ist ein schöner selbstständiger Codeblock, der als `# Zeichnen aller Reize` beschrieben werden kann. Der Code liefert Details darüber, was genau gezeichnet wird, was die Zeichenreihenfolge ist, usw. Aber dieser einzelne Kommentar hilft dir zu verstehen, worum es in diesem Codeblock geht und ob er für dich momentan relevant ist. Das Gleiche gilt für `# Verarbeitung von Tastendrücken` oder `# Überprüfung der Spielende-Bedingungen`, usw. Aber sei vorsichtig und stelle sicher, dass der Kommentar den Code korrekt beschreibt. Wenn der Kommentar zum Beispiel `# Zeichnen aller Reize` sagt, sollte es nirgendwo anders einen Code zum Zeichnen von Reizen geben und keinen weiteren Code, der etwas anderes tut!

### 3.1.4 Schreibe deinen Code Schritt für Schritt

Dein Motto sollte "langsam aber stetig" lauten. Auf diese Weise werde ich dich durch die Spiele führen. Beginne immer mit etwas extrem Einfachem wie einem statischen Rechteck oder Bild. Stelle sicher, dass es funktioniert. Füge eine kleinere Funktionalität hinzu: Farbwechsel, Positionsänderung, ein weiteres Rechteck, speichere es als Attribut, usw. Stelle sicher, dass es funktioniert. Gehe niemals zum nächsten Schritt, es sei denn, du verstehst vollständig, was dein aktueller Code macht und du bist dir zu 100% sicher, dass er sich so verhält, wie er sollte. Und ich meine 100% ernst! Wenn du auch nur den Schatten eines Zweifels hast, überprüfe es erneut. Andernfalls wird dieser Schatten wachsen und dich zunehmend unsicher über deinen Code machen. Dieser Ansatz in Schildkrötengeschwindigkeit mag albern und übermäßig langsam erscheinen, ist aber immer noch schneller, als einen großen Codeblock zu schreiben und dann zu versuchen, ihn zum Laufen zu bringen. Es ist viel einfacher, einfache Probleme einzeln zu lösen, als viele auf einmal.

### 3.1.5 Es ist nichts Falsches an StackOverflow

Ja, man kann immer versuchen, eine Lösung für sein Problem auf StackOverflow zu finden[2]. Ich mache das die ganze Zeit! Sie sollten die bereitgestellte Lösung jedoch *nur verwenden, wenn Sie sie verstehen*! Kopieren Sie nicht einfach den Code, der ein Problem wie Ihres zu lösen *scheint*. Wenn Sie das tun und Sie haben Glück, könnte es funktionieren. Oder, wieder wenn Sie Glück haben, funktioniert es auf eine offensichtliche Weise nicht. Aber wenn Sie nicht so viel Glück haben, wird es (manchmal) subtil falsch funktionieren. Und da Sie nicht wirklich wussten, was der Code tat, als Sie ihn einfügten, werden Sie noch mehr

[2]Wenn Sie jedoch an dem Seminar teilnehmen, fragen Sie mich bitte zuerst!

verwirrt sein. Nutzen Sie also StackOverflow als eine Quelle des Wissens, nicht als eine Quelle zum Kopieren und Einfügen von Code!

## 3.2 Den Code lesen

Das Lesen von Code ist einfach, weil Computer dumm sind und Sie schlau sind. Das bedeutet, dass die Anweisungen, die Sie dem Computer geben, notwendigerweise sehr einfach sind und daher für einen Menschen sehr leicht zu verstehen sind. Das Lesen von Code ist jedoch auch schwierig, weil Computer dumm sind und Sie schlau sind. Sie sind so schlau, dass Sie nicht einmal den gesamten Code lesen müssen, um zu verstehen, was er tut. Sie lesen nur die Schlüsselstellen und füllen die Lücken aus. Leider neigen Sie dazu, Fehler zu überlesen. Dies ist nicht nur beim Programmieren der Fall, wenn Sie jemals einen Text Korrektur gelesen haben, wissen Sie, wie schwierig es ist, Tippfehler zu finden. Ihr Gehirn korrigiert sie in Echtzeit mit Hilfe des Kontexts und Sie lesen das Wort so, wie es sein sollte, nicht so, wie es tatsächlich geschrieben ist[3].

Meine Erfahrung mit Programmierung im Allgemeinen und im Besonderen auf diesem Seminar ist, dass die meisten Probleme, bei denen Sie stecken bleiben, im Nachhinein einfach bis dumm und offensichtlich sind[4]. Verzweifeln Sie nicht! Es liegt nicht an Ihnen, sondern ist lediglich eine Folge davon, wie wunderbar Ihr Gehirn für die Mustererkennung verdrahtet ist. Im Folgenden finden Sie einige Vorschläge, die Ihnen helfen könnten, das Lesen von Code robuster zu gestalten.

### 3.2.1 Denken Sie wie ein Computer

Lesen Sie den Code Zeile für Zeile und "führen Sie ihn aus", so wie es der Computer tun würde. Verwenden Sie Stift und Papier, um den Überblick über die Variablen zu behalten. Verfolgen Sie, welche Codeblöcke wann erreicht werden können. Verlangsamen Sie sich und stellen Sie sicher, dass Sie jede Zeile verstehen und in der Lage sind, den Überblick über die Variablen zu behalten. Sobald Sie das tun, wird es einfach sein, einen Fehler zu erkennen.

### 3.2.2 Tun Sie so, als hätten Sie diesen Code in Ihrem Leben noch nie gesehen

Nehmen Sie an, Sie haben keine Ahnung, was der Code tut. Wie ich schrieb, sehen Sie oft *wörtlich* einen Fehler nicht, weil Ihr Gehirn Details auffüllt und die Realität so biegt, dass sie Ihren Erwartungen entspricht[5]. Sie *wissen*, was dieser

[3] Tipp: Lesen Sie Ihren Text ein Satz nach dem anderen von hinten nach vorne oder lesen Sie zufällig einen Satz nach dem anderen. Dies bricht den Fluss des Textes und hilft Ihnen, sich auf die Wörter zu konzentrieren, anstatt auf die Bedeutung und die Geschichte.

[4] Im Nachhinein ist man immer schlauer!

[5] Kürzlich verbrachte ich eine halbe Stunde damit, zu verstehen, warum zwei identische Codeblöcke unterschiedliche Ergebnisse liefern. Mein Sohn fand fast sofort einen Unterschied

Codeblock tun sollte, also lesen Sie ihn statt dessen nur flüchtig und nehmen an, dass er tut, was er soll, es sei denn, es sieht offensichtlich schrecklich falsch aus. Es ist schwer, Ihre Erwartungen auszuschalten, aber es ist immens hilfreich.

### 3.2.3 Suchen Sie nicht nur unter der Straßenlampe

Immer wenn Sie neuen Code verwenden oder etwas implementieren müssen, das kompliziert erscheint, und Ihr Code nicht so funktioniert, wie er sollte, neigen Sie dazu, anzunehmen, dass das Problem beim neuen, ausgefallenen Code liegt. Einfach, weil er neu, ausgefallen und kompliziert ist. Meiner Erfahrung nach versteckt sich der Fehler jedoch typischerweise in der einfacheren "trivialen" Codezeile in der Nähe, die Sie nie richtig anschauen, weil sie einfach und trivial ist. Überprüfen Sie alles, nicht nur die Stellen, an denen Sie einen Fehler vermuten.

### 3.2.4 Nutzen Sie den Debugger

Im Buch werden Sie lernen, wie Sie die Ausführung Ihres Spiels anhalten können, um seinen Zustand zu untersuchen. Nutzen Sie dieses Wissen! Setzen Sie Haltepunkte und führen Sie den Code Schritt für Schritt aus. Überprüfen Sie die Werte von Variablen im Tab "Watch". Verwenden Sie die Debug-Konsole, um zu überprüfen, ob Funktionen die Ergebnisse liefern, die sie sollten. Teilen Sie komplexe Bedingungen oder mathematische Formeln in kleine Teile auf, kopieren und führen Sie diese Teile in der Debug-Konsole aus und überprüfen Sie, ob die Zahlen zusammenpassen. Stellen Sie sicher, dass ein Codeblock in Ordnung ist und analysieren Sie dann den nächsten. Das Debuggen ist besonders hilfreich, um den Code zu identifizieren, der nicht erreicht wird oder zum falschen Zeitpunkt erreicht wird.

## 3.3 Zen von Python

Ich fand den Zen von Python als gute Inspiration, um die Programmierung anzugehen.

(ein fehlendes Komma in einem von ihnen), weil es für ihn nur ein Haufen Buchstaben und Zahlen war.

# Chapter 4

# Python Grundlagen

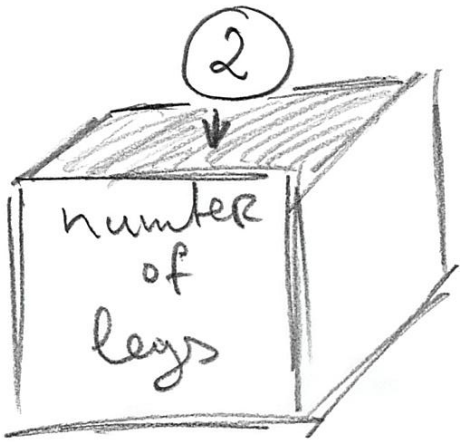
Hoffentlich hast du bereits einen speziellen Ordner für dieses Buch erstellt. Lade das Übungs-Notebook herunter (Alt+Klick sollte es eher herunterladen als öffnen), lege es in den Ordner des Kapitels und öffne es, siehe die relevanten Anweisungen. Du musst zwischen den Erklärungen hier und den Übungen im Notizbuch hin und her wechseln, also halte sie beide offen.

## 4.1 Konzepte des Kapitels

- Variablen.
- Konstanten.
- Grundlegende Werttypen.
- Dinge ausdrucken.
- Werte in Zeichenketten einfügen.

## 4.2 Variablen

Das erste grundlegende Konzept, mit dem wir uns vertraut machen müssen, ist die **Variable**. Variablen werden verwendet, um Informationen zu speichern, und du kannst sie dir als eine Kiste mit einem Namensschild vorstellen, in die du etwas hineinlegen kannst. Das Namensschild auf dieser Kiste ist der Name der Variable und ihr Wert ist das, was du darin speicherst. Zum Beispiel können wir eine Variable erstellen, die die Anzahl der Beine speichert, die ein Spielfigur hat. Wir beginnen mit einer für einen Menschen typischen Zahl.




In Python würdest du schreiben

```
anzahl_der_beine = 2
```

Die obige **Zuweisungsanweisung** hat eine sehr einfache Struktur:

```
<variablen-name> = <wert>
```

Der Variablenname (das Namensschild auf der Kiste) sollte aussagekräftig sein, er kann mit Buchstaben oder _ beginnen und kann Buchstaben, Zahlen und das Symbol _ enthalten, aber keine Leerzeichen, Tabs, Sonderzeichen usw. Python empfiehlt[1] dass du die **snake_case** Schreibweise (alles in Kleinbuchstaben, Unterstrich für Leerzeichen) verwendest, um deine Variablennamen zu formatieren. Der `<Wert>` auf der rechten Seite ist eine komplexere Geschichte, da er fest codiert sein kann (wie im obigen Beispiel), berechnet werden kann, indem andere Variablen oder dieselbe Variable, zurückgegeben von einer Funktion, usw. verwendet werden.

Die Verwendung von Variablen bedeutet, dass du dich auf die **Bedeutung** der entsprechenden Werte konzentrieren kannst, anstatt dir Sorgen darüber zu machen, was diese Werte sind. Beispielsweise kannst du das nächste Mal, wenn du etwas auf Grundlage der Anzahl der Beine eines Charakters berechnen musst (z.B., wie viele Paar Schuhe benötigt ein Charakter), dies auf Grundlage des aktuellen Wertes der Variablen `anzahl_der_beine` berechnen, anstatt anzunehmen, dass es `1` ist.

```
# SCHLECHT: Warum 1? Ist es, weil der Character zwei Beine hat oder
# weil wir jedem Character ein Paar Schuhe geben, unabhängig von
# seiner tatsächlichen Anzahl von Beinen?
paar_schuhe = 1

# BESSER (aber was, wenn unser Charakter nur ein Bein hat?)
```

[1] Naja, eigentlich besteht es darauf.

```
paar_schuhe = anzahl_der_beine / 2
```

Variablen geben dir auch Flexibilität. Ihre Werte können sich während des Programmablaufs ändern: Der Punktestand des Spielers steigt, die Anzahl der Leben nimmt ab, die Anzahl der Zauber, die er wirken kann, steigt oder fällt je nach ihrem Einsatz, usw. Dennoch kannst du immer den Wert in der Variablen verwenden, um die notwendigen Berechnungen durchzuführen. Hier ist zum Beispiel ein leicht erweitertes Beispiel für `anzahl_der_schuhe`.

```
anzahl_der_beine = 2

# ...
# etwas passiert und unser Charakter wird in einen Tintenfisch verwandelt
anzahl_der_beine = 8
# ...

# der gleiche Code funktioniert immer noch und wir können immer noch die richtige Anzahl an Schuh
paar_schuhe = anzahl_der_beine / 2
```

Wie bereits erwähnt, kannst du eine Variable als eine beschriftete Kiste betrachten, in die du etwas hineinlegen kannst. Das bedeutet, dass du immer den alten Wert "wegwerfen" und etwas Neues hineinlegen kannst. Im Falle von Variablen geschieht der "Wegwerf"-Teil automatisch, da ein neuer Wert den alten überschreibt. Überprüfe selbst, welcher der endgültige Wert der Variable im unten stehenden Code ist?

```
anzahl_der_beine = 2
anzahl_der_beine = 5
anzahl_der_beine = 1
anzahl_der_beine
```

Mache Übung #1.

Beachte, dass eine Variable (eine "Kiste mit Namensschild") erst existiert, nachdem du ihr etwas zugewiesen hast. Der folgende Code erzeugt also einen `NameError`, die Python-art zu sagen, dass sie noch nie von der Variable `anzahl_der_haende` gehört hat.

```
anzahl_der_beine = 2
anzahl_der_handschuhe = anzahl_der_haende / 2
```

Du kannst jedoch eine Variable erstellen, die keinen *spezifischen* Wert hat, indem du ihr `None` zuweist. `None` wurde speziell für die Bedeutung *kein Wert* oder *nichts* zur Sprache hinzugefügt.

```
anzahl_der_haende = None # Die Variable existiert jetzt, hat aber keinen speziellen Wert.
```

Wie du bereits gesehen hast, kannst du einen Wert *berechnen*, anstatt ihn anzugeben. Was wäre die Antwort hier?

```
anzahl_der_beine = 2 * 2
anzahl_der_beine = 7 - 2
anzahl_der_beine
```

Mache Übung #2.

## 4.3 Zuweisungen sind keine Gleichungen!

**Sehr wichtig**: obwohl Zuweisungen *wie* mathematische Gleichungen *aussehen*, sind sie **keine Gleichungen!** Sie folgen einer **sehr wichtigen** Regel, die Sie im Kopf behalten müssen, wenn Sie Zuweisungen verstehen: die rechte Seite eines Ausdrucks wird *zuerst* ausgewertet, bis der Endwert berechnet ist. Erst dann wird dieser Endwert der auf der linken Seite angegebenen Variable zugewiesen (also in die Kiste gelegt). Das bedeutet, dass Sie die gleiche Variable auf *beiden* Seiten verwenden können! Sehen wir uns diesen Code an:

```
x = 2
y = 5
x = x + y - 4
```

Was passiert, wenn der Computer die letzte Zeile auswertet? Zunächst nimmt er die *aktuellen* Werte aller Variablen (`2` für `x` und `5` für `y`) und setzt sie in den Ausdruck. Nach diesem internen Schritt sieht der Ausdruck so aus:

```
x = 2 + 5 - 4
```

Dann berechnet er den Ausdruck auf der rechten Seite und speichert, **sobald die Berechnung abgeschlossen ist**, diesen neuen Wert in `x`

```
x = 3
```

Machen Sie die Übung #3, um sicherzugehen, dass Sie dies verstanden haben.

## 4.4 Konstanten

Obwohl die eigentliche Stärke von Variablen darin besteht, dass Sie ihren Wert ändern können, sollten Sie sie auch dann verwenden, wenn der Wert im gesamten Programm konstant bleibt. In Python gibt es keine echten Konstanten, sondern die Übereinkunft, dass ihre Namen vollständig `GROSSGESCHRIEBEN` sein sollten. Entsprechend wissen Sie, wenn Sie `SOLCH_EINE_VARIABLE` sehen, dass Sie ihren Wert nicht ändern sollten. Technisch gesehen ist das nur eine Empfehlung, denn niemand kann Sie davon abhalten, den Wert einer `KONSTANTE` zu ändern. Aber ein großer Teil der Benutzerfreundlichkeit von Python resultiert aus solchen Übereinkünften (wie der `snake_case` Konvention oben). Wir werden später mehr von solchen Übereinkünften treffen, zum Beispiel beim Lernen über Objekte.

Unter Berücksichtigung all dessen, wenn die Anzahl der Beine im Spiel konstant bleibt, sollten Sie diese Konstanz betonen und schreiben

```
ANZAHL_DER_BEINE = 2
```

Ich empfehle dringend die Verwendung von Konstanten und vermeide das Hardcoding von Werten. Erstens, wenn Sie mehrere identische Werte haben, die verschiedene Dinge bedeuten (2 Beine, 2 Augen, 2 Ohren, 2 Fahrzeuge pro Figur, etc.), wird Ihnen eine `2` im Code nicht verraten, was diese `2` bedeutet (die Beine? Die Ohren? Der Punktemultiplikator?). Sie können das natürlich herausfinden, basierend auf dem Code, der diese Nummer verwendet, aber Sie könnten sich diese zusätzliche Mühe ersparen und stattdessen eine ordnungsgemäß benannte Konstante verwenden. Dann lesen Sie einfach ihren Namen und die Bedeutung des Wertes wird offensichtlich, und es ist die Bedeutung und nicht der tatsächliche Wert, der Sie hauptsächlich interessiert. Zweitens, wenn Sie entscheiden, diesen Wert dauerhaft zu *ändern* (sagen wir, unsere Hauptfigur ist jetzt ein Dreifuß), bedeutet die Verwendung einer Konstante, dass Sie sich nur an einer Stelle Sorgen machen müssen, der Rest des Codes bleibt unverändert. Wenn Sie diese Zahl hart codiert haben, erwartet Sie eine aufregende [2] und definitiv lange Suche und Ersetzung im gesamten Code.

Machen Sie die Übung #4.

## 4.5 Datentypen

Bisher haben wir nur ganzzahlige numerische Werte verwendet (1, 2, 5, 1000…). Obwohl Python viele verschiedene Datentypen unterstützt, konzentrieren wir uns zunächst auf eine kleine Auswahl davon:

- Ganze Zahlen, die wir bereits verwendet haben, z.B. `-1`, `100000`, `42`.
- Fließkommazahlen, die jeden realen Wert annehmen können, z.B. `42.0`, `3.14159265359`, `2.71828`.
- Zeichenketten, die Text speichern können. Der Text ist zwischen entweder gepaarten Anführungszeichen `"einiger Text"` oder Apostrophen `'einiger Text'` eingeschlossen. Das bedeutet, dass Sie Anführungszeichen oder Apostrophe innerhalb der Zeichenkette verwenden können, solange sie von der Alternative umschlossen ist. Z.B., `"Schüleraufgaben"` (eingeschlossen in `"`, Apostroph `'` innen) oder `'"Alle Verallgemeinerungen sind falsch, auch diese." Mark Twain'` (Zitat von Apostrophen eingeschlossen). Es gibt noch viel mehr zu Zeichenketten und wir werden dieses Material im Laufe des Kurses behandeln.
- Logische / boolesche Werte, die entweder `True` oder `False` sind.

Bei der Verwendung einer Variable ist es wichtig, dass Sie wissen, welchen Datentyp sie speichert, und das liegt meist bei Ihnen. In einigen Fällen wird Python

[2] nicht wirklich

einen Fehler ausgeben, wenn Sie versuchen, eine Rechnung mit inkompatiblen Datentypen durchzuführen. In anderen Fällen wird Python Werte automatisch zwischen bestimmten Typen konvertieren, z.B. ist jeder Ganzzahlwert auch ein Realwert, so dass die Konvertierung von `1` zu `1.0` meist trivial und automatisch ist. In anderen Fällen müssen Sie jedoch möglicherweise eine explizite Konvertierung verwenden. Gehen Sie zur Übung #5 und versuchen Sie zu erraten, welcher Code laufen wird und welcher einen Fehler wegen inkompatiblen Typen werfen wird?

```
5 + 2.0
'5' + 2
'5' + '2'
'5' + True
5 + True
```

Mache Übung #5.

Überrascht vom letzten? Das liegt daran, dass intern `True` auch `1` und `False` `0` ist!

Sie können explizit von einem Typ in einen anderen umwandeln, indem Sie spezielle Funktionen verwenden. Beispielsweise können Sie eine Zahl oder einen logischen Wert in einen String umwandeln, indem Sie einfach str(<value>) schreiben. Was wäre das Ergebnis in den untenstehenden Beispielen?

```
str(10 / 2)
str(2.5 + True)
str(True)
```

Mache Übung #6.

Ähnlich können Sie mit der Funktion bool(<value>) in eine logische/boolesche Variable umwandeln. Die Regeln sind einfach, für numerische Werte ist `0` gleich `False`, jeder andere Nicht-Null-Wert wird in `True` umgewandelt. Für Zeichenketten wird eine leere Zeichenkette `''` als `False` bewertet und eine nicht leere Zeichenkette wird in `True` umgewandelt. Was wäre die Ausgabe in den untenstehenden Beispielen?

```
bool(-10)
bool(0.0)

secret_message = ''
bool(secret_message)

bool('False')
```

Mache Übung #7.

Die Umwandlung in Ganzzahlen oder Fließkommazahlen mit int(<value>) bzw. float(<value>) ist komplizierter. Der einfachste Fall ist von logisch auf Ganz-

zahl/Fließkommazahl, da `True` Ihnen `int(True)` ist `1` und `float(True)` ist `1.0` gibt und `False` gibt Ihnen `0`/`0.0`. Beim Umwandeln von Fließkommazahl auf Ganzzahl lässt Python einfach den Bruchteilteil fallen (es rundet nicht richtig!). Bei der Umwandlung einer Zeichenkette muss es sich um eine gültige Zahl des entsprechenden Typs handeln, sonst wird ein Fehler erzeugt. Sie können z. B. eine Zeichenkette wie `"123"` in eine Ganzzahl oder eine Fließkommazahl umwandeln, aber das funktioniert nicht für `"a123"`. Darüber hinaus können Sie `"123.4"` in eine Fließkommazahl umwandeln, aber nicht in eine Ganzzahl, da sie einen Bruchteil enthält. Angesichts all dessen, welche Zellen würden funktionieren und welche Ausgabe würden sie erzeugen?

```
float(False)
int(-3.3)
float("67.8")
int("123+3")
```

Mache Übung #8.

## 4.6 Ausgabe drucken

Um den Wert auszudrucken, müssen Sie die Funktion print() verwenden (wir werden später allgemein über Funktionen sprechen). Im einfachsten Fall übergeben Sie den Wert und er wird ausgegeben.

```
print(5)
#> 5
```

oder

```
print("fünf")
#> fünf
```

Natürlich wissen Sie bereits über die Variablen Bescheid, also statt den Wert direkt einzugeben, können Sie stattdessen eine Variable übergeben und ihr *Wert* wird ausgegeben.

```
anzahl_der_pfannkuchen = 10
print(anzahl_der_pfannkuchen)
#> 10
```

oder

```
frühstück = "pfannkuchen"
print(frühstück)
#> pfannkuchen
```

Sie können auch mehr als einen Wert/Variablen an die Druckfunktion übergeben und alle Werte werden nacheinander gedruckt. Wenn wir dem Benutzer zum Beispiel sagen wollen, was ich zum Frühstück hatte, können wir das tun

```
frühstück = "pfannkuchen"
anzahl_der_artikel = 10
print(frühstück, anzahl_der_artikel)
#> pfannkuchen 10
```

Was wird von dem untenstehenden Code gedruckt?

```
abendessen = "steak"
zähler = 4
nachtisch = "muffins"

print(zähler, abendessen, zähler, nachtisch)
```

Mache Übung #9.

Allerdings möchten Sie wahrscheinlich expliziter sein, wenn Sie die Informationen ausdrucken. Stellen Sie sich zum Beispiel vor, Sie haben diese drei Variablen:

```
mahlzeit = "Frühstück"
gericht = "Pfannkuchen"
anzahl = 10
```

Sie könnten natürlich `print(mahlzeit, gericht, anzahl)` machen, aber es wäre schöner, "*Ich hatte* ***10*** ***Pfannkuchen*** *zum* ***Frühstück***" zu drucken, wobei die in Fettschrift gedruckten Elemente die eingefügten Variablenwerte wären. Dafür müssen wir die Formatierung von Zeichenketten verwenden. Bitte beachten Sie, dass die Formatierung von Zeichenketten nicht spezifisch für das Drucken ist, Sie können einen neuen Zeichenkettenwert über die Formatierung erstellen und ihn in einer Variable speichern, ohne ihn auszudrucken, oder ihn ausdrucken, ohne ihn zu speichern.

## 4.7 Formatierung von Zeichenketten

Eine großartige Ressource zur Formatierung von Zeichenketten in Python ist pyformat.info. Da sich Python ständig weiterentwickelt, gibt es nun mehr als eine Art, Zeichenketten zu formatieren. Im Folgenden werde ich das "alte" Format vorstellen, das auf der klassischen Formatierung von Zeichenketten basiert, die in der Funktion `sprintf` in C, Matlab, R und vielen anderen Programmiersprachen verwendet wird. Es ist etwas weniger flexibel als neuere, aber für einfache Aufgaben ist der Unterschied vernachlässigbar. Das Wissen über das alte Format ist nützlich wegen seiner Allgemeinheit. Wenn Sie Alternativen lernen möchten, lesen Sie unter dem oben angegebenen Link.

Der allgemeine Aufruf lautet `"ein String mit Formatierung"%(Tupel von Werten, die während der Formatierung verwendet werden)`. Sie werden später mehr über Tupel lernen. Gehen Sie im Moment davon aus, dass es sich einfach um eine durch Kommas getrennte Liste von Werten handelt, die in runden Klammern eingeschlossen sind: `(1, 2, 3)`.

In `"ein String mit Formatierung"`, geben Sie an, wo Sie den Wert mit dem Zeichen `%` einfügen möchten, das von einer *optionalen* Formatierungsinformation und dem *erforderlichen* Symbol, das den **Typ** des Wertes definiert, gefolgt wird. Die Typsymbole sind

- `s` für Zeichenkette
- `d` für eine Ganzzahl
- `f` für einen Fließkommawert
- `g` für einen "optimal" gedruckten Fließkommawert, so dass für große Werte die wissenschaftliche Notation verwendet wird (*z.B.*, `10e5` statt `100000`).

Hier ist ein Beispiel, wie man einen String mit einer Ganzzahl formatiert:

```
print("Ich hatte %d Pfannkuchen zum Frühstück"%(10))
#> Ich hatte 10 Pfannkuchen zum Frühstück
```

Sie sind nicht darauf beschränkt, einen einzigen Wert in einen String einzufügen. Sie können weitere Positionen über `%` angeben, müssen jedoch sicherstellen, dass Sie die richtige Anzahl von Werten in der richtigen Reihenfolge übergeben. Können Sie vor dem Ausführen herausfinden, welcher Aufruf tatsächlich funktioniert (und was die Ausgabe sein wird) und welcher einen Fehler verursacht?

```
print('Ich hatte %d Pfannkuchen und entweder %d oder %d Steaks zum Abendessen'%(2))
print('Ich hatte %d Pfannkuchen und %d Steaks zum Abendessen'%(7, 10))
print('Ich hatte %d Pfannkuchen und %d Steaks zum Abendessen'%(1, 7, 10))
```

Machen Sie Übung #10.

Wie oben erwähnt, haben Sie im Falle von echten Werten zwei Möglichkeiten: `%f` und `%g`. Letzterer verwendet die wissenschaftliche Notation (z.B. `1e10` für `10000000000`), um eine Darstellung kompakter zu machen.

Machen Sie Übung #11, um ein besseres Gefühl für den Unterschied zu bekommen.

Es gibt noch viel mehr zur Formatierung und Sie können auf pyformat.info darüber lesen. Diese Grundlagen sind jedoch ausreichend, um in dem nächsten Kapitel mit der Programmierung unseres ersten Spiels zu beginnen.

# Chapter 5

# Errate die Zahl: eine einzelne Runden Edition

Das vorherige Kapitel deckte die Grundlagen von Python ab, sodass Du nun bereit bist, Dein erstes Spiel zu entwickeln! Wir werden es Schritt für Schritt aufbauen, da es viel zu lernen gibt über Eingaben, Bibliotheken, bedingte Aussagen und Einrückungen.

Bevor Du anfängst, erstelle einen neuen Ordner (innerhalb Deines Kursordners), benenne ihn zum Beispiel "guess-the-number", lade das Übungsnotizbuch herunter, kopiere es in den neu erstellten Ordner und öffne es in Jupyter Notebook. Wie im vorherigen Kapitel wird es Übungen zum Lesen und Verstehen des Codes enthalten.

Wir werden jedoch VS Code verwenden, um Skripte mit dem eigentlichen Spiel zu programmieren. Du musst für jede Code-Praxis eine separate Datei erstellen[1] (z.B., *code01.py*[2], *code02.py*, etc.) Dies ist nicht die effizienteste Implementierung einer Versionskontrolle und wird sicherlich den Ordner überladen. Aber es würde mir ermöglichen, Deine Lösungen für jeden Schritt zu sehen, was es mir leichter machen würde, Feedback zu geben. Zum Einreichen der Aufgabe, zippe einfach den Ordner und reiche die Zip-Datei ein.

## 5.1 Konzepte des Kapitels

- Dokumentation des Codes.
- Debuggen von Code.

[1] Du kannst den vorherigen Code "Speichern unter…" um das Herumkopieren von Dingen per Hand zu vermeiden.

[2] Ich empfehle die Verwendung von `01` statt von `1`, da dies eine konsistente Dateisortierung in Deinem Dateimanager gewährleistet

- Eingabe einer Eingabe von einem Benutzer.
- Verwendung von Vergleichen in bedingten Aussagen.
- Verwendung von Einrückungen, um Aussagen zu gruppieren.
- Verwendung von Python Bibliotheken.
- Generieren von Zufallszahlen.

## 5.2 Das Spiel

Wir werden ein Spiel programmieren, bei dem ein Teilnehmer (Computer) eine Zahl innerhalb eines bestimmten Bereichs auswählt (sagen wir, zwischen 1 und 10) und der andere Teilnehmer (menschlicher Spieler) versucht, diese zu erraten. Nach jedem Versuch des Menschen antwortet der Computer, ob die tatsächliche Zahl niedriger als die Vermutung ist, höher als die Vermutung ist oder mit ihr übereinstimmt. Das Spiel ist vorbei, wenn der Spieler die Zahl richtig errät oder (in der späteren Version des Spiels) keine Versuche mehr hat.

Unsere erste Version wird nur einen Versuch zulassen und der Gesamtspielalgorithmus wird folgendermaßen aussehen:

1. der Computer generiert eine Zufallszahl zwischen 1 und 10
2. druckt sie aus für Debugging-Zwecke
3. fordert den Benutzer auf, eine Vermutung einzugeben
4. vergleicht die beiden Zahlen und gibt das Ergebnis aus: “Meine Zahl ist niedriger”, “Meine Zahl ist höher” oder “Genau richtig!”

## 5.3 Lass uns eine Zahl auswählen

Beginnen wir damit, nur die ersten zwei Schritte des Programms zu implementieren. Erstelle zuerst eine Variable, die eine Zahl enthält, die der Computer “ausgesucht” hat. Wir nennen sie `number_picked` (Du kannst auch einen anderen aussagekräftigen Namen verwenden, aber es könnte einfacher sein, wenn wir alle den gleichen Namen verwenden). Um die Dinge am Anfang einfacher zu machen, codieren wir eine beliebige Zahl zwischen 1 und 10 hart (wähle die, die Dir gefällt). Dann lass uns diese ausdrucken, so dass wir die Zahl selbst kennen[3]. Verwende die Zeichenkettenformatierung, um die Dinge benutzerfreundlich zu gestalten, z. B. drucke etwas aus wie “Die Zahl, die ich ausgewählt habe, ist …”. Du solltest in der Lage sein, dies mit dem Wissen aus dem vorherigen Kapitel zu tun. Dein Code sollte aus zwei Zeilen bestehen:

```
# 1. erstelle Variable und setze ihren Wert
# 2. gebe den Wert aus
```

Probiere diesen Zweizeiler in einem Jupyter Notebook aus (erstelle ein leeres Notebook nur dafür). Wenn Du damit zufrieden bist, kopiere-den Code in

[3]Natürlich wissen wir es, weil wir es hart kodiert haben, aber das wird nicht der Fall sein, wenn der Computer es zufällig generiert, also lassen wir uns für die Zukunft planen

`code01.py` und lies weiter, um zu erfahren, wie Du es dokumentieren und ausführen kannst.

Füge Deinen Code in `code01.py` ein.

## 5.4 Dokumentiere Deinen Code

Jetzt, wo Du Deine erste Datei mit einem Python-Programm hast, solltest Du es dokumentieren. Die Dokumentation eines zwei Zeilen langen und einfachen Programms mag albern erscheinen, aber es sollte etwas Automatisches sein. Später wirst Du mehrere Zeilen Kommentare verwenden, um eine einzelne Zeilenfunktion zu dokumentieren. Noch einmal, es geht nicht um den Code, der funktioniert, es geht um den Code, den Du verstehen kannst. In gewisser Weise ist es besser, ein sauberes, gut dokumentiertes Programm zu haben, das momentan nicht korrekt funktioniert, als einen undokumentierten Spaghetti-Code, der funktioniert. Du kannst das Erstere korrigieren und aktualisieren, das Letztere zu warten oder zu aktualisieren…

In Python hast Du zwei Möglichkeiten, Kommentare zu schreiben: mehrzeilig und einzeilig

```
'''Ein
mehrzeiliger
Kommentar
'''

#  Ein einzeiliger Kommentar.
```

Verwende mehrzeilige Kommentare, um die Dokumentation für einzelne Dateien, Funktionen, Klassen, Methoden usw. zu schreiben. Du wirst lernen, wie Du diese Dokumentation im Numpy-Docstring-Stil formatierst, sobald Du Funktionen kennst. In unserem Fall solltest Du Deine `code01.py`-Datei mit einem mehrzeiligen Kommentar beginnen, der kurz beschreibt, welches Programm sie enthält. Mindestens solltest Du schreiben, dass dies ein *Guess a Number*-Spiel ist. Wahrscheinlich ist es eine gute Idee, zu skizzieren, worum es in dem Spiel geht.

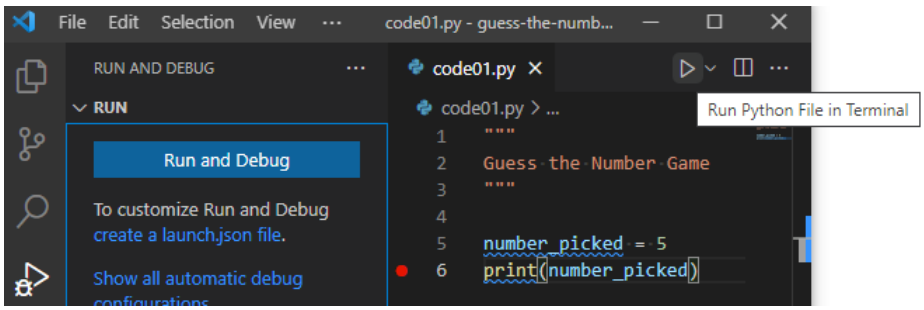
Verwende einzeilige Kommentare, um zu erklären, was in einem bestimmten Codeblock passiert. Du musst nicht jede Zeile kommentieren und Du solltest den Code nicht in menschlicher Sprache replizieren. Ein Kommentar sollte darüber sein, *was* passiert, nicht *wie*. Du hast bereits einen Block Code, also dokumentiere ihn mit einem einzelnen Satz.

Dokumentiere die `code01.py`.

## 5.5 Ausführen und Debuggen deines Spiels in VS Code

Jetzt, da wir ein zwei Zeilen langes Programm haben, können wir es ausführen und schon anfangen zu lernen, wie man es debuggt. Ja, unser aktuelles Programm ist wahrscheinlich zu einfach, um ein tatsächliches Debugging zu benötigen, aber es ist auch einfach genug, um das Verständnis des Debuggings zu erleichtern. Und das Debuggen ist eine entscheidende Fähigkeit, die ein laufendes Programm von einer Blackbox in etwas transparentes und leichtes[4] zu verstehen verwandelt. Weiter unten werde ich beschreiben, wie man in VS Code debuggt, aber du könntest ein offizielles Handbuch zu Rate ziehen, falls sich in der Zwischenzeit etwas geändert hat.

Es gibt zwei Möglichkeiten, ein Python-Programm in VS Code auszuführen. Zuerst kannst Du den *"Run Python File in Terminal"* Abspielknopf auf der rechten Seite verwenden. Dies führt dein Programm *ohne* Debugger aus, so dass Du es nur für die tatsächlichen Läufe des finalisierten Codes verwenden solltest. Trotzdem kannst Du versuchen, es auszuführen und zu sehen, ob es das ausgibt, was es sollte.




Die Alternative ist der Debugging-Reiter, der eine kleine Wanze auf dem Run-Knopf hat. Wenn Du ihn auswählst, wird *"Run and Debug"* angezeigt. Klicke auf den Knopf und es werden verschiedene Optionen für verschiedene Arten von Python-Projekten und -Frameworks angeboten. Für unsere Absichten und Zwecke benötigen wir nur *"Python File: Debug the currently active Python file"*. Klicke darauf und es wird Deinen Code ausführen (sollte genau so laufen wie mit dem anderen Knopf).

Du willst wahrscheinlich nicht die ganze Zeit durch Debugging-Tab → Run and Debug-Knopf → Konfiguration auswählen klicken. Ein besserer Weg ist es, es einmal für alle zu konfigurieren und dann die **F5**-Taste zu verwenden, um Deinen Code auszuführen. Klicke zunächst auf *"Erstelle eine launch.json-Datei"* und wähle erneut *"Python File: Debug the currently active Python file"* aus. Du wirst sehen, dass eine neue *launch.json* Datei im Editor erscheint, die so aussehen sollte:

[4] Oder zumindest leichter.

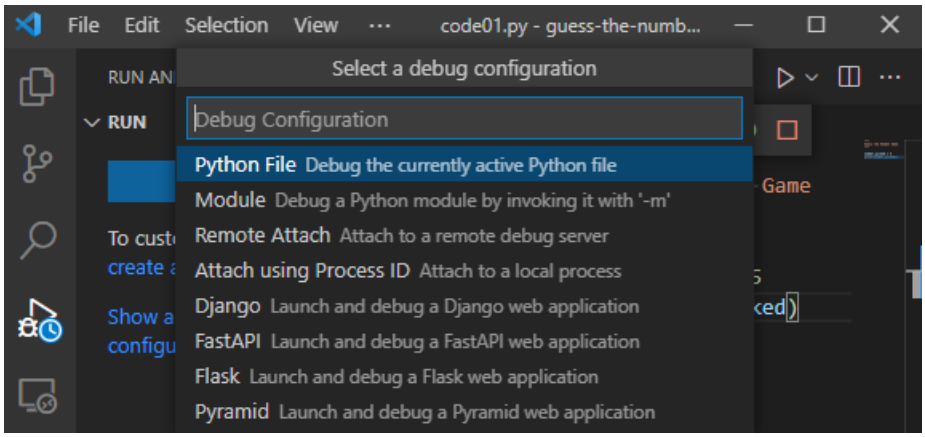



Figure 5.1: Selecting debugging configuration.

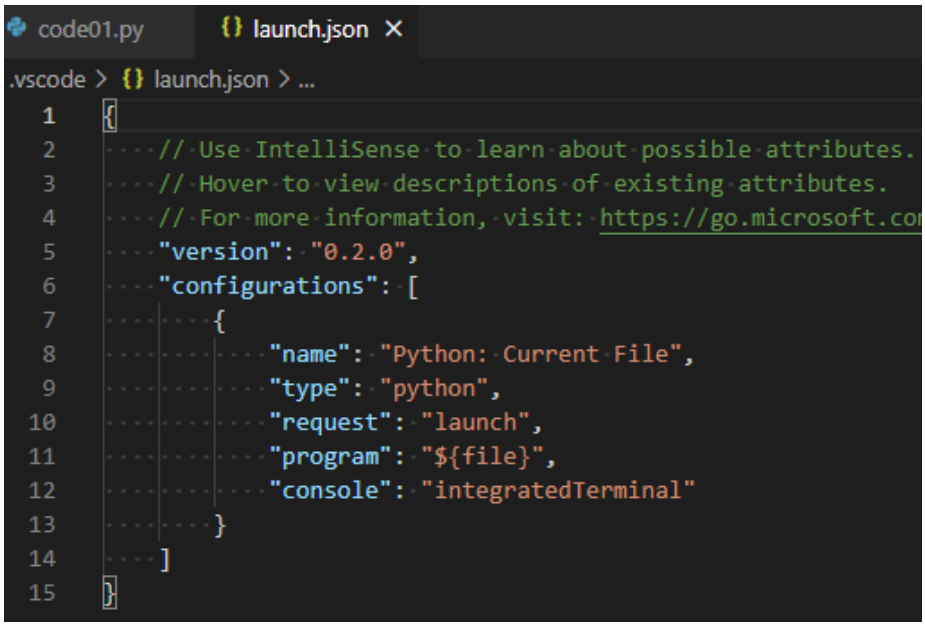



Figure 5.2: Debug configuration in launch.json file.

Das war's! VS Code hat für Dich eine Konfiguration erstellt. Jetzt kannst Du die *launch.json*-Datei schließen und Dein Programm durch einfaches Drücken der **F5**-Taste ausführen. Versuch es! Wieder einmal sollte es genauso funktionieren wie zuvor, aber warum sind wir dann durch all diese Mühe gegangen? Weil der Debugger die Ausführung Deines Codes *pausiert*, wann immer er auf ein Problem trifft und Dir die Chance gibt, Variablen zu untersuchen, Code-Snippets auszuführen usw. Im Gegensatz dazu wird das Ausführen der Python-Datei im Terminal (die erste Option) nur eine Fehlermeldung ausgeben und das Programm beenden. Außerdem kannst Du *Breakpoints* verwenden, um das Programm an jeder Zeile anzuhalten, was Dir die Möglichkeit gibt, Deinen Code an jeder Stelle, die Du benötigst, zu untersuchen.

Du aktivierst Breakpoints, indem Du links von der Zeilennummer, die Dich interessiert, klickst. Hier habe ich auf Zeile 6 geklickt und Du kannst einen roten Punkt sehen, der einen aktiven Breakpoint anzeigt

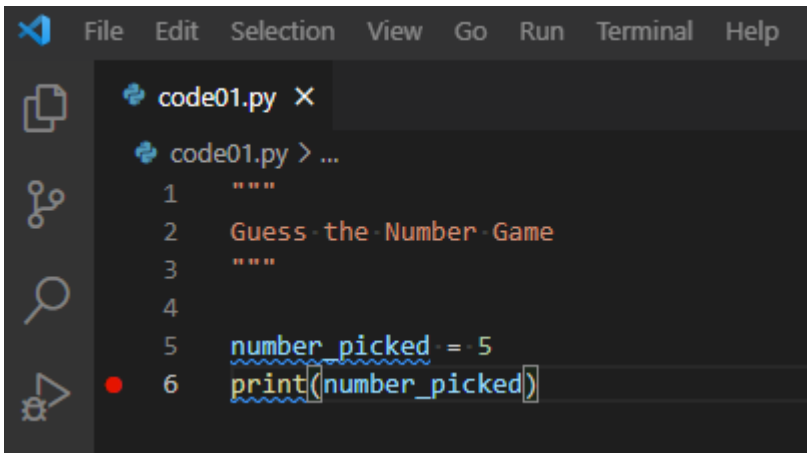



Figure 5.3: Active breakpoint.

Wenn ich jetzt den Code über **F5** ausführe, hält das Programm an dieser Zeile *bevor* es sie ausführt.

Das gibt mir die Möglichkeit zu sehen, welchen Wert meine Variable `number_picked` hat. Sie ist bereits in den lokalen Variablen aufgeführt (oben links). Aber ich habe sie auch zur Liste der beobachteten Variablen (*Watch*, links in der Mitte) hinzugefügt und mir ihren Wert in der *Debug Console* (unten) angesehen, die es mir ermöglicht, *beliebigen* Python-Code auszuführen, während mein Programm angehalten ist. Mach das Gleiche und erkunde selbst diese unterschiedlichen Möglichkeiten. Sieh zum Beispiel, wie du `number_picked + 1` oder `number_picked * number_picked` im *Watch* Tab und in der *Debug Console* berechnen kannst.

Sobald du fertig bist, den aktuellen Zustand des Programms zu untersuchen,

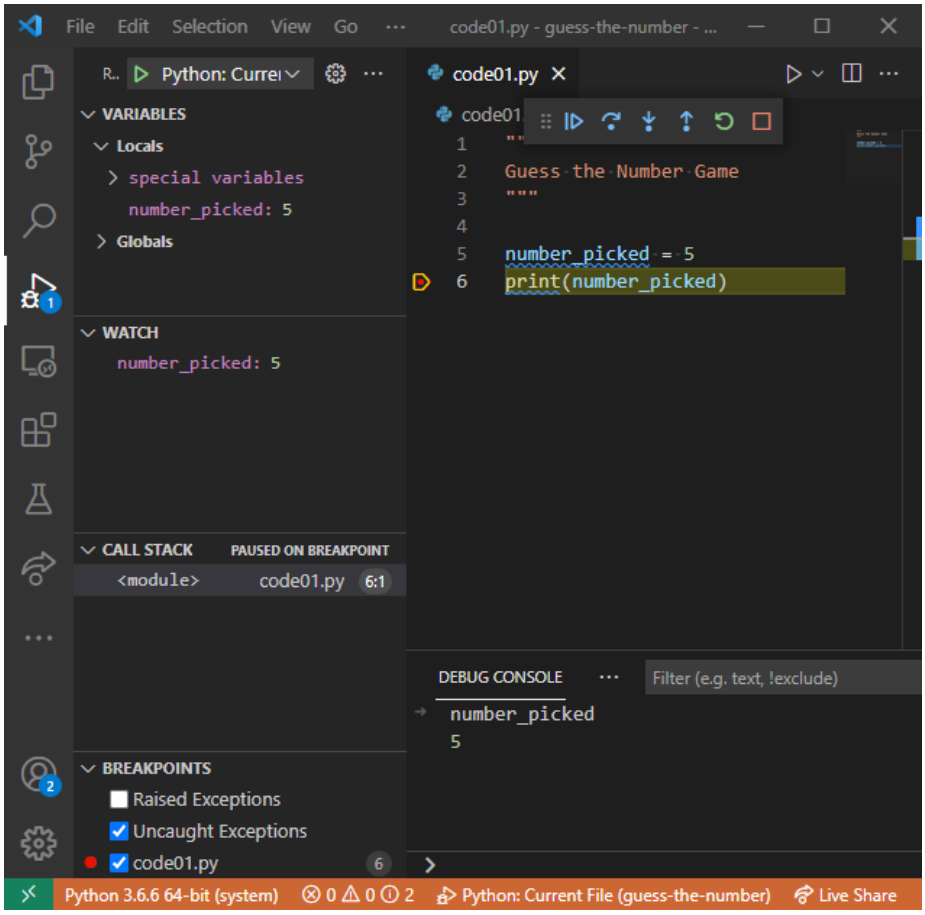



Figure 5.4: Program paused at the breakpoint.

hast du sechs Buttons oben zur Verfügung, um zu entscheiden, was als nächstes zu tun ist (fahre mit der Maus darüber, um Hinweise zu sehen). Sie sind, von links nach rechts

- Fortfahren (**F5**): Setze das Programm fort.
- Step Over (**F10**): Führt den Code aus, ohne in Funktionen zu gehen (diese und die beiden folgenden Optionen werden klarer, sobald du lernst, Funktionen zu schreiben).
- In den Code hineinsteigen (**F11**)
- Aus dem Code heraussteigen (**Shift+F11**).
- Das Programm neu starten (**Ctrl+Shift+F5**).
- Das Programm stoppen (**Shift+F5**).

Um besser zu verstehen, wie das funktioniert, stoppe das Programm (**Shift+F5**) und setze einen zusätzlichen Haltepunkt in die erste Zeile deines Codes (Zeile #5 in meinem Programm, die anderen Zeilen sind Kommentare oder leer). Führe das Programm wieder über **F5** aus und es wird an dieser ersten Zeile pausieren. Kannst du jetzt den Wert der Variable `number_picked` ermitteln?

Die Antwort lautet "nein", weil diese Variable noch nicht existiert. Denke daran, das Programm pausiert *vor* dem Ausführen der Zeile. Benutze **F10**, um Schritt für Schritt durch den Code zu gehen und zu sehen, wie die Variable erscheint und die Information ausgegeben wird.

Diese Debugging-Übung war vielleicht nicht notwendig, um Probleme mit deinem aktuellen Code zu lösen, aber sie hat gezeigt, wie das in der Zukunft funktioniert. Zögere nicht, einen Haltepunkt zu setzen, um das Programm zu pausieren und zu überprüfen, ob die Realität (d.h. die tatsächlichen Werte der Variablen) deinen Erwartungen entspricht. Nutze das Durchlaufen des Codes, um die Dinge zu verlangsamen und zu beobachten und zu überlegen.

## 5.6 Einen Spieler nach einer Vermutung fragen

Um das Spiel *Guess the Number* zu spielen, braucht es zwei. Der Computer hat seinen Teil erledigt, indem er eine Zahl ausgewählt hat, jetzt müssen wir einen Spieler bitten, seine Vermutung einzugeben. Dafür verwenden wir die input() *Funktion*.

Eine Funktion ist ein isolierter Code, der (optionale) *Eingaben* akzeptiert, eine *Aktion* ausführt und optional einen Wert (*Ausgabe*) zurückgibt. Dies ermöglicht einerseits, den Code in kleinere Blöcke zu unterteilen, die einfacher zu warten sind und andererseits, den gleichen Code wiederzuverwenden. Du hast bereits die print() Funktion verwendet, um Dinge auszudrucken, und die str(), bool(), int() und float() Funktionen, um Werte zu konvertieren. Bei der print() Funktion ist die *Eingabe* eine beliebige Anzahl von Werten (sogar keine, probiere das in Jupiter Notebook aus!), ihre *Aktion* ist es, Dinge auszudrucken, aber sie

gibt nichts zurück (keine *Ausgabe*). Die float() Funktion nimmt (etwas überraschend) null oder einen Wert als *Eingabe* (versuche, ihr keinen oder mehr als einen in Jupiter Notebook zu geben und sieh den Unterschied), versucht, den gegebenen Wert in float zu konvertieren (wirft einen Fehler, wenn sie das nicht kann), und gibt einen float-Wert als *Ausgabe* zurück.

Ein ähnliches *Eingabe* → *Aktion* → *Ausgabe* Schema gilt für die input(prompt) Funktion. Sie akzeptiert optional einen `prompt` String als Eingabe. Dann druckt sie die `prompt` Nachricht aus und wartet darauf, dass ein Benutzer einen *String* eingibt, bis dieser **Enter** drückt. Dann gibt sie diesen *String*-Wert zurück. Dieser letzte Punkt über den *String* ist wichtig, denn in unserem Spiel müssen die Spieler eine *ganze Zahl* und keinen String eingeben. Gehen wir für einen Moment davon aus, dass die Eingabe immer eine gültige Ganzzahl ist, also gib nur gültige Ganzzahlen ein, wenn du das Programm testest! Auf diese Weise können wir sie ohne zusätzliche Prüfungen (die wir in der Zukunft hinzufügen werden) in eine Ganzzahl umwandeln und dieser Wert einer neuen Variable namens `guess` zuweisen. Du musst also eine einzelne Zuweisungsanweisung mit der `guess` Variablen auf der linken Seite und dem Aufruf der input() Funktion auf der rechten Seite hinzufügen (denke an eine schöne Aufforderungsnachricht), eingehüllt (innerhalb) der Typumwandlung in eine Ganzzahl über int(). Teste diesen Code, aber gib wieder nur gültige Ganzzahlen ein, damit die Umwandlung ohne Fehler funktioniert.

Aktualisiere deine `code01.py`

## 5.7 Bedingte *if*-Anweisung

Jetzt haben wir zwei Zahlen: Eine, die der Computer ausgewählt hat (`number_picked`), und eine, die der Spieler eingegeben hat (`guess`). Wir müssen sie vergleichen, um die korrekte Ausgabemeldung zu liefern. Dafür verwenden wir die bedingte if-Anweisung:

```
if eine_Bedingung_ist_wahr:
    # Tu etwas
elif eine_andere_Bedingung_ist_wahr:
    # Tu etwas anderes
elif noch_eine_andere_Bedingung_ist_wahr:
    # Tu noch etwas anderes
else:
    # Tu etwas nur, wenn alle Bedingungen oben falsch sind.
```

Nur der `if`-Teil ist erforderlich, während `elif` (kurz für "else, if") und `else` optional sind. So kannst du etwas tun, nur wenn eine Bedingung wahr ist:

```
if eine_Bedingung_ist_wahr:
    # Tu etwas, aber ANSONSTEN TU NICHTS
    # und fahre mit der Codeausführung fort
```

```
# Einige Codezeilen, die nach der if-Anweisung ausgeführt werden,
# unabhängig davon, ob die Bedingung wahr war oder nicht.
```

Bevor wir bedingte Anweisungen in unserem Spiel verwenden können, musst du (1) die Bedingungen selbst und (2) die Verwendung von Einrückungen als Mittel zur Gruppierung von Anweisungen verstehen.

## 5.8 Bedingungen und Vergleiche

Eine Bedingung ist jeder Ausdruck, der überprüft werden kann, um zu sehen, ob er `True` oder `False` ist. Ein einfaches Beispiel für einen solchen Ausdruck sind Vergleiche, die in menschlicher Sprache ausgedrückt werden als: *Ist heute Donnerstag? Ist die Antwort (gleich) 42? Regnet es und habe ich einen Regenschirm?* Wir werden uns für einen Moment auf solche Vergleiche konzentrieren, aber später wirst Du sehen, dass in Python *jeder* Ausdruck entweder `True` oder `False` ist, auch wenn er nicht wie ein Vergleich aussieht[5].

Für den Vergleich kannst Du die folgenden Operatoren verwenden:

- *"A ist gleich B"* wird geschrieben als `A == B`.
- *"A ist nicht gleich B"* wird geschrieben als `A != B`.
- *"A ist größer als B"* und *"A ist kleiner als B"* sind entsprechend `A > B` und `A < B`.
- *"A ist größer als oder gleich B"* und *"A ist kleiner als oder gleich B"* sind entsprechend `A >= B` und `A <= B` (bitte beachte die Reihenfolge der Symbole, da `=>` und `=<` einen Fehler erzeugen werden).

Löse einige Vergleiche in Übung #1.

Beachte, dass Python auch einen is Operator hat, der *identisch* mit `==` *aussehen* kann (z.B. sieht `x == 2` äquivalent zu `x is 2` aus). Darüber hinaus funktioniert er in *einigen* Fällen auch auf die gleiche Weise. Es gibt jedoch einen subtilen Unterschied: `==` überprüft, ob *Werte* identisch sind, während `is` überprüft, ob *Objekte* (die "Werte halten") identisch sind. Du musst Klassen und Objekte verstehen, bevor Du diesen Unterschied schätzen kannst, also behalte vorerst im Hinterkopf, dass Du nur `==` verwenden solltest (ich werde explizit erwähnen, wann `is` benötigt wird).

Du kannst den logischen Wert mit dem Operator `not` *invertieren*, da `not True` `False` und `not False True` ist. Das bedeutet, dass `A != B` dasselbe ist wie `not A == B` und entsprechend ist `A == B` `not A != B`. Um zu sehen, wie das funktioniert, betrachte beide Fälle, wenn `A` tatsächlich gleich `B` ist und wenn es nicht so ist.

[5] Dies liegt daran, dass Du jeden Wert über die Funktion bool(), die Du letztes Mal kennengelernt hast, in einen logischen Wert umwandeln kannst und so jeder Wert (umgewandelt) entweder `True` oder `False` ist.

- Wenn A gleich B ist, dann ergibt `A == B` `True`. Das `A != B` ist dann `False`, also `not A != B` → `not False` → `True`.
- Wenn A nicht gleich B ist, dann ergibt `A == B` `False`. Das `A != B` ist dann `True`, also `not A != B` → `not True` → `False`.

Überprüfe diese Inversion selbst in Übung #2.

Du kannst auch mehrere Vergleiche mit den Operatoren `and` und/oder[6] `or` kombinieren. Wie in der menschlichen Sprache bedeutet `and`, dass beide Teile wahr sein müssen: `True and True` → `True`, aber `True and False` → `False`, `False and True` → `False`, und `False and False` → `False`. Dasselbe gilt, wenn Du mehr als zwei Bedingungen/Vergleiche über `and` verknüpfst: **Alle** müssen wahr sein. Im Fall von `or` muss nur eine der Aussagen wahr sein, z.B. `True or True` → `True`, `True or False` → `True`, `False or True` → `True`, aber `False or False` → `False`. Auch hier gilt für mehr als zwei Vergleiche/Bedingungen, dass mindestens eine von ihnen wahr sein sollte, damit der gesamte Ausdruck wahr ist.

Mache die Übungen #3 und #4.

Ein subtiler, aber wichtiger Punkt: Bedingungen werden von links nach rechts ausgewertet, bis der gesamte Ausdruck auf die eine oder andere Weise aufgelöst ist. Das bedeutet, dass wenn der erste Ausdruck in einem `and False` ist, der zweite (der Rest von ihnen) **nie ausgewertet** wird. D.h., wenn sowohl `erster` als auch `zweiter` Ausdruck `True` sein müssen und Du weißt, dass `erster` Ausdruck bereits `False` ist, ist der gesamte Ausdruck in jedem Fall `False`. Das bedeutet, dass es im folgenden Code keinen Fehler geben wird, obwohl die Auswertung von `int("e123")` allein einen `ValueError` auslösen würde.

```
2 * 2 == 5 and int("e123") == 123
#> False
```

Umkehrst Du jedoch die Reihenfolge, so dass `int("e123") == 123` zuerst ausgewertet werden muss, erhältst Du eine Fehlermeldung

```
int("e123") == 123 and 2 * 2 == 4
#> invalid literal for int() with base 10: 'e123'
```

Ähnlich verhält es sich, wenn bei `or` *irgendein* Ausdruck `True` ist, musst Du den Rest nicht überprüfen.

```
2 * 2 == 4 or int("e123") == 123
#> True
```

Ist die erste Bedingung jedoch `False`, müssen wir fortfahren (und stolpern dabei über einen Fehler):

```
2 * 2 == 5 or int("e123") == 123
#> invalid literal for int() with base 10: 'e123'
```

[6] Wortspiel beabsichtigt

Mache Übung #5.

Schließlich kannst Du, wie in der einfachen Arithmetik, Klammern `()` verwenden, um Bedingungen zu gruppieren. So kann die Aussage "Ich esse immer Schokolade, aber ich esse Spinat nur, wenn ich hungrig bin" wie folgt geschrieben werden: `food == "chocolate" or (food == "spinach" and hungry)`. Hier werden `food == "chocolate"` und `food == "spinach" and hungry` unabhängig voneinander ausgewertet, ihre Werte werden an ihrer Stelle eingesetzt und dann wird die `and`-Bedingung ausgewertet.

Mache Übung #6.

Ein letzter Gedanke zu Vergleichen: Zögere nicht, sie in Jupyter Notebook mit verschiedenen Kombinationen von Werten zu testen oder das Programm an der Bedingung über einen Haltepunkt anzuhalten und einen Vergleich in *Watch* oder *Debug Console* auszuwerten.

## 5.9 Gruppieren von Anweisungen über Einrückungen

Lass uns zu einer bedingten if-Anweisung zurückkehren. Sieh dir das folgende Codebeispiel an (und beachte ein `:` am Ende von `if some_condition_is_true:`), in dem *Anweisung #1* nur ausgeführt wird, wenn *eine Bedingung* wahr ist, während *Anweisung #2* danach ausgeführt wird, *unabhängig* von der Bedingung.

```
if some_condition_is_true:
    Anweisung #1
Anweisung #2
```

Beide Aussagen #1 und #2 folgen nach der if-Anweisung. Aber wie erkennt Python, dass die erste nur ausgeführt wird, wenn die Bedingung wahr ist, und die andere immer ausgeführt wird? Die Antwort ist Einrückung: Die **4 (vier!) Leerzeichen**, die automatisch hinzugefügt werden, wenn du in VS Code auf **Tab** drückst und entfernt werden, wenn du auf **Shift+Tab** drückst. Die Einrückung platziert Anweisung #1 *innerhalb* der if-Anweisung. Daher zeigt die Einrückung an, ob Anweisungen zur gleichen Gruppe gehören und nacheinander ausgeführt werden müssen (gleiche Einrückungsebene für `if` und `Anweisung #2`) oder sich innerhalb einer bedingten Anweisung, Schleife, Funktion, Klasse usw. befinden (`Anweisung #1`). Für komplexeren Code, der zum Beispiel eine if-Anweisung innerhalb einer if-Anweisung innerhalb einer Schleife haben wird, drückst du dies aus, indem du weitere Ebenen der Einrückung hinzufügst. Zum Beispiel:

```
# einige Anweisungen außerhalb der Schleife (0 Einrückung)
while Spiel_ist_nicht_vorbei: # (0 Einrückung)
    # Anweisungen innerhalb der Schleife
```

```
    if Taste_gedrückt: # (Einrückung von 4)
        # innerhalb der Schleife und if-Anweisung
        if Taste == "Space": # (Einrückung von 8)
            # innerhalb der Schleife, if-Anweisung und einer weiteren if-Anweisung
            springen() # (Einrückung von 12)
        else: # (Einrückung von 4)
            # innerhalb der Schleife, if-Anweisung und im sonst-Teil einer weiteren if-Anweisung
            stehen() # (Einrückung von 12)

    # Anweisungen innerhalb der Schleife, aber außerhalb der äußersten if-Anweisung
    drucke(Taste) # (Einrückung von 4)

# einige Anweisungen außerhalb der Schleife (0 Einrückung)
```

Achte sehr genau auf die Einrückung, denn sie bestimmt, welche Anweisungen zusammen ausgeführt werden! Ein falscher Einrückungsgrad ist leider ein sehr häufiger Fehler.

Mache Übung #7.

Die `if` und `ifelse` Anweisungen werden so lange ausgewertet, bis eine von ihnen sich als `True` herausstellt. Danach werden alle folgenden `ifelse` und `else` Anweisungen einfach ignoriert.

Mache Übung #8.

## 5.10 Überprüfung der Antwort

Jetzt hast du alle notwendigen Werkzeuge, um die erste Version unseres Spiels zu beenden. Füge deinem `code01.py` bedingte Anweisungen hinzu, so dass

- wenn die Computerwahl kleiner als die Vermutung des Spielers ist, druckt es `"Meine Zahl ist niedriger!"`
- wenn die Computerwahl größer als die Vermutung des Spielers ist, druckt es `"Meine Zahl ist höher!"`
- wenn die beiden Zahlen identisch sind, druckt es `"Volltreffer!"`

Speichere zunächst eine Kopie deines ursprünglichen Codes in `code02.py` und füge dann den Vergleich und das Drucken hinzu. ::: {.program} Erweitere dein Programm mit dem Vergleich in `code02.py` :::

Teste, ob dein Code funktioniert. Nutze wieder Breakpoints, wenn du den Kontrollfluss besser verstehen und überprüfen möchtest, ob die Vergleiche so funktionieren, wie du es erwartest.

## 5.11 Verwenden von Bibliotheken

Unser Spiel ist "funktionsvollständig": der Computer wählt eine Zahl, der Spieler macht eine Vermutung, der Computer reagiert entsprechend. Derzeit spielen wir jedoch für beide Seiten. Lass uns den Computer selbst eine zufällige Zahl auswählen. Dafür müssen wir die Funktion randint(a, b) verwenden. Sie ist Teil jeder Python-Distribution, so dass du sie auch hättest, wenn du eine einfache Python-Distribution installierst, anstatt eine von PsychoPy zu verwenden. Du kannst sie jedoch nicht sofort so verwenden, wie du es mit `print()` oder `input()` getan hast. Gib `randint(1, 3)` in dein Jupyter Notebook ein und beobachte den *NameError: name 'randint' is not defined*.

Der Grund dafür ist, dass Python unglaublich viele Funktionen hat und das Laden aller gleichzeitig den Speicher mit Dingen verstopfen würde, die du nie zu verwenden beabsichtigt hast. Stattdessen sind sie in *Bibliotheken* verpackt, so dass du nur die Funktionen (oder Bibliotheken) importieren kannst, die du tatsächlich für dein Programm benötigst. Du importierst sie über eine `import`-Anweisung, die an den Anfang deiner Datei gehört (aber unter dem Kommentar zum Inhalt der Datei). Es gibt mehrere Möglichkeiten, wie du Bibliotheken importieren kannst. Erstens kannst du eine *gesamte* Bibliothek importieren (wie zum Beispiel die *random* Bibliothek, die die Funktion `randint()` hat, die wir benötigen) und dann ihre Funktionen als `<Bibliothek>.<Funktion>` verwenden. Für `randint` wäre das

```
import random

computer_wahl = random.randint(1, 5)
```

Ich würde dies als bevorzugte Art der Verwendung von Bibliotheken empfehlen, da es dich zwingt, den Namen der Bibliothek explizit zu erwähnen, wenn du eine Funktion aufrufst, d.h. `random.randint()` statt nur `randint()`. Dies mag nicht wichtig erscheinen, wenn nur eine Bibliothek importiert wird, aber selbst in einem moderat großen Projekt wirst du viele Bibliotheken importieren, so dass es schwierig wird herauszufinden, zu welcher Bibliothek die Funktion gehört. Noch wichtiger ist, dass verschiedene Bibliotheken Funktionen mit *demselben Namen* haben können. In diesem Fall stammt die Funktion, die du verwendest, aus der *letzten* Bibliothek, die du importiert hast. Aber du wirst das vielleicht nicht bemerken und dies ist eine Art von Fehler, der wirklich schwer aufzuspüren ist. Importiere also immer die gesamte Bibliothek und verwende die `Bibliothek.` Notation, es sei denn, du hast einen sehr guten Grund, etwas anderes zu tun!

Eine weitere und weniger explizite Option besteht darin, nur *einige* Funktionen zu importieren und sie *ohne* den `library.` Präfix zu verwenden. Du kannst mehr als eine Funktion importieren, indem du sie alle auflistest.

```
from random import randint, randrange
```

```
computer_wahl = randint(1, 5)
```

Du kannst auch eine Bibliothek oder eine Funktion beim Importieren über `as` *umbenennen*. Kurz gesagt, solltest du dies nicht tun, da die Verwendung eines anderen Namens für eine Bibliothek oder eine Funktion es anderen (und sogar dem zukünftigen Du) erschweren würde, deinen Code zu verstehen. Es gibt jedoch einige "standardmäßige" Umbenennungsmuster, die universell verwendet werden und auf die du wahrscheinlich stoßen wirst.

```
# dies ist die übliche Art, diese zwei Bibliotheken zu importieren
import numpy as np
import pandas as pd

np.abs(-1)

# du kannst auch einzelne Funktionen umbenennen, wenn du unbedingt musst (aber bitte nicht!)
from random import randint as zufalls_zahl

computer_wahl = zufalls_zahl(1, 5)
```

Zum Schluss gibt es noch eine **sehr schlechte Möglichkeit**, Funktionen aus einer Bibliothek zu importieren: `from random import *`. Das Sternchen bedeutet, dass du *alle* Funktionen aus der Bibliothek importieren möchtest und sie ohne `random.` Präfix aufrufen möchtest. Tu das niemals, niemals, niemals[7]! Dadurch wird deine Umgebung mit Funktionen gefüllt, von denen du vielleicht nichts weißt, die möglicherweise andere Funktionen überschreiben, Konflikte verursachen etc. Niemals! Ich zeige dir das nur, weil du irgendwann einen Code sehen wirst, der diesen Ansatz verwendet, und du könntest denken, dass das eine gute Idee ist. Es ist eine schreckliche Idee! Importiere die Bibliothek, nicht die Funktionen, damit du explizit zeigen kannst, auf welche Bibliothek du dich verlässt, wenn du eine Funktion aufrufst. Denke immer an den Zen of Python: "Explizit ist besser als implizit."

## 5.12 Eine Zufallszahl auswählen

Jetzt wo du weißt, wie man eine Bibliothek importiert, können wir die Funktion randint() verwenden. Speichere dafür eine Kopie deines vorherigen Codes in `code03.py`. Importiere die Bibliothek und verwende randint(), um eine Zufallszahl zwischen 1 und 10 zu generieren. Lese die Dokumentation zu randint(), um zu verstehen, wie man es verwendet. Das Lesen von Handbüchern ist ein notwendiger Teil des Programmierens, also ist dies ein guter Zeitpunkt, um das Üben zu beginnen.

Sobald du dies in `code02.py` implementiert hast, führe es mehrmals aus, um zu überprüfen, dass der Computer tatsächlich unterschiedliche Zufallswerte

[7] Habe ich schon niemals gesagt? Niemals!

auswählt. Verwende erneut Breakpoints, wenn du genau überprüfen möchtest, was passiert.

Gib deinen Code in `code03.py` ein.

Herzlichen Glückwunsch, du hast gerade dein erstes Computerspiel programmiert! Ja, es ist sehr einfach, aber es hat die Schlüsselzutaten: eine zufällige Entscheidung des Computers, Benutzereingabe und Feedback. Beim nächsten Mal wirst du über Schleifen lernen, um mehrere Versuche zu ermöglichen, und wirst damit beginnen, Funktionen zu schreiben, um deinen Code modular und zuverlässig zu machen. In der Zwischenzeit vertiefen wir dein Wissen, indem wir noch ein Spiel programmieren!

## 5.13 Einarmiger Bandit (Einzelrundenedition)

Du weißt alles, was du brauchst, um eine einfache Version eines "Einarmigen Banditen"-Spiels zu programmieren. Hier ist die Spiellogik:

1. Importiere die Zufallsbibliothek, damit du die Funktion `randint` verwenden kannst.
2. Generiere drei zufällige Ganzzahlen (sagen wir, zwischen 1 und 5) und speichere sie in drei Variablen `slot1`, `slot2` und `slot3`.
3. Drucke die Zahlen aus, verwende die String-Formatierung, um sie schön aussehen zu lassen.
4. Zusätzlich,
    - wenn alle drei Werte gleich sind, drucke `"Dreierpasch!"`.
    - Wenn nur zwei Zahlen übereinstimmen, drucke `"Paar!"`.
    - Drucke nichts, wenn alle Zahlen unterschiedlich sind.

Vergiss nicht, die neue Datei `code03.py` zu dokumentieren und verwende Breakpoints, um sie zu debuggen, wenn du möchtest.

Gib deinen Code in `code04.py` ein.

## 5.14 Abgabe für das Seminar

Für das Seminar gib einen gezippten Ordner mit dem Übungs-Notebook und allen vier Programmen ab.

# Chapter 6

# Errate die Zahl: Eine Mehr-Runden-Ausgabe

Im vorherigen Kapitel hast Du ein "Guess the Number"-Spiel programmiert, das nur einen einzigen Versuch erlaubt. Jetzt werden wir es erweitern, um mehrere Versuche zu ermöglichen und werden andere Extras hinzufügen, um es noch spannender zu machen. Erstelle einen neuen Unterordner und lade das Übungs-Notebook herunter, bevor wir beginnen!

## 6.1 Konzepte des Kapitels

- Wiederholen von Code mit while Schleife.
- Machen in Notausgang aus einer Schleife.

## 6.2 While-Schleife

Wenn Du etwas wiederholen möchtest, musst Du Schleifen verwenden. Es gibt zwei Arten von Schleifen: die while Schleife, welche wiederholt wird, *während* eine Bedingung wahr ist, und die for Schleife, die über Elemente iteriert (wir werden sie später verwenden).

Die grundlegende Struktur einer *while*-Schleife ist

```
# Anweisungen vor der Schleife

while <Bedingung>:
    # die inneren Anweisungen werden
    # so lange ausgeführt, wie
    # die Bedingung wahr ist
```

```
# Anweisungen nach der Schleife
```

Die `<Bedingung>` hier ist ein Ausdruck, der entweder `True` oder `False` ergibt, genau wie in einer `if...elif...else` Bedingungsanweisung. Auch gelten dieselben Einrückungsregeln, die bestimmen, welcher Code innerhalb der Schleife und welcher außerhalb ist.

Mache Übung #1.

Lass uns die *while*-Schleife verwenden, um dem Spieler zu erlauben, weiter zu raten, bis er es endlich richtig hat. Du kannst den Code, den Du während des letzten Seminars programmiert hast, kopieren und einfügen oder Du kannst ihn von Grund auf neu erstellen (ich würde Dir dringend empfehlen, Letzteres zu tun!). Die allgemeine Programmstruktur sollte folgende sein

```
# importiere die Zufallsbibliothek, damit du die randint Funktion verwenden kannst

# generiere eine zufällige Zahl und speichere sie in der Variablen number_picked
# hole die Eingabe des Spielers, konvertiere sie in eine Ganzzahl und speichere sie in

# während die Vermutung des Spielers nicht gleich dem Wert ist, den der Computer ausge
    # gib "meine Zahl ist kleiner" oder "meine Zahl ist größer" aus, indem Du die if-e
    # hole die Eingabe des Spielers, konvertiere sie in eine Ganzzahl und speichere si

# gib "Ganz genau!" aus
# (denn wenn wir hier angekommen sind, bedeutet das, dass die Vermutung gleich der Wah
```

Speichere deinen Code in `code01.py`.

Vergiss nicht, die Datei zu dokumentieren und Breakpoints und Step overs zu verwenden, um den Programmfluss zu erkunden.

## 6.3 Versuche zählen

Jetzt fügen wir eine Variable hinzu, die die Gesamtzahl der Versuche des Spielers zählt. Dazu erstelle eine neue Variable (nennen wir sie `attempts` oder ähnliches) *vor der Schleife* und initialisiere sie mit `1` (weil der erste Versuch vor dem Betreten der Schleife vom Spieler gemacht wird). Füge jedes Mal, wenn der Spieler eine Vermutung eingibt, `1` hinzu. Erweitern Sie nach der Schleife die Nachricht `"Ganz genau!"` um Informationen über die Anzahl der Versuche. Nutze String-Formatierung, um alles schön aussehen zu lassen, z.B.: `"Ganz genau und du hast nur 5 Versuche gebraucht!"`. Überprüfe, ob die Anzahl der benötigten Versuche mit der Anzahl der vom Programm gemeldeten Versuche übereinstimmt!

Speichere deinen Code in `code02.py`.

## 6.4 Abbruch (und Ausstieg)

Der Code innerhalb der *while*-Schleife wird wiederholt ausgeführt, während die Bedingung `True` ist und, was wichtig ist, der gesamte Code innerhalb wird ausgeführt, bevor die Bedingung erneut bewertet wird. Manchmal musst du jedoch früher abbrechen, ohne den verbleibenden Code auszuführen. Dafür hat Python eine break Anweisung, die dazu führt, dass das Programm die Schleife sofort verlässt, ohne den Rest des Codes innerhalb der Schleife auszuführen, so dass das Programm mit dem Code *nach* der Schleife fortfährt.

```
# dieser Code wird vor der Schleife ausgeführt

while <irgendeine_Bedingung>:
  # dieser Code wird bei jeder Iteration ausgeführt

    if <irgendeine_andere_Bedingung>:
        break

  # dieser Code wird bei jeder Iteration ausgeführt, aber nicht, wenn du aus der Schleife heraust

# dieser Code wird nach der Schleife ausgeführt
```

Mache Übung #2, um dein Verständnis zu vertiefen.

## 6.5 Begrenzung der Anzahl der Versuche mittels Break

Setzen wir den Spieler unter Druck! Entscheide dich für eine maximale Anzahl an Versuchen, die du erlaubst, und speichere sie als KONSTANTE. Wähle einen passenden Namen (z.B. `MAX_ATTEMPTS`) und BEACHTE, GROßBUCHSTABEN für den Namen einer Konstanten! Nun nutze `break` um die `while`-Schleife zu verlassen, wenn die aktuelle Versuchszahl größer als `MAX_ATTEMPTS` ist. Überlege, wann (innerhalb des Codes innerhalb der Schleife) du dies prüfen solltest.

Speichere deinen Code in `code03.py`.

## 6.6 Korrekte End-of-Game-Nachricht

Aktualisieren wir die finale Nachricht. Im Moment steht dort “Ganz genau…”, weil wir davon ausgegangen sind, dass das Programm die Schleife nur dann verlässt, wenn der Spieler die richtige Antwort gegeben hat. Bei begrenzten Versuchen ist das nicht unbedingt der Fall. Es gibt jetzt zwei Gründe, warum es die while-Schleife verlassen hat:

1. Der Spieler hat die richtige Antwort gegeben.
2. Dem Spieler sind die Versuche ausgegangen.

Verwende die `if-else` Bedingungsanweisung, um eine passende Nachricht auszugeben. Zum Beispiel drucke `"Viel Glück beim nächsten Mal!"`, wenn der Spieler verloren hat (die Versuche ausgegangen sind).

Speichere deinen Code in `code04.py`.

## 6.7 Begrenzung der Anzahl der Versuche ohne Break

Obwohl es meine Idee war, die `break` Anweisung hinzuzufügen, solltest du sie sparsam verwenden. Ohne `break` gibt es eine *einzige* Stelle im Code, die du überprüfen musst, um zu verstehen, wann das Programm die Schleife verlassen wird: die Bedingung. Wenn du jedoch ein `break` hinzufügst, hast du jetzt *zwei* Stellen, die geprüft werden müssen. Und jedes zusätzliche `break` fügt weitere hinzu. Das bedeutet aber nicht, dass du sie um jeden Preis vermeiden solltest! Du *solltest* sie verwenden, wenn dadurch der Code leichter zu verstehen ist. Aber überprüfe immer, ob eine modifizierte Bedingung auch den Trick tun könnte.

Probieren wir genau das aus. Ändere deinen Code so ab, dass er *ohne* die `break` Anweisung funktioniert. Du brauchst eine kompliziertere Bedingung für deine while-Schleife, so dass sie sich wiederholt, solange die Vermutung des Spielers falsch ist und die Anzahl der Versuche noch kleiner ist als die maximal erlaubte. Teste, ob dein Code sowohl funktioniert, wenn du gewinnst als auch wenn du verlierst.

Speichere deinen Code in `code05.py`.

## 6.8 Verbleibende Versuche anzeigen

Es geht alles um die Benutzeroberfläche! Ändere die Eingabeaufforderung so, dass sie die Anzahl der *verbleibenden* Versuche enthält. Z.B. `"Bitte gebe eine Vermutung ein, du hast noch X Versuche übrig"`.

Speichere deinen Code in `code06.py`.

## 6.9 Wiederholung des Spiels

Lass uns dem Spieler die Option geben, noch einmal zu spielen. Das bedeutet, dass wir *allen* aktuellen Code in eine weitere `while`-Schleife packen (das nennt man *verschachtelte Schleifen*), die so lange wiederholt wird, wie der Spieler weiterspielen möchte. Der Code sollte folgendermaßen aussehen:

```
# importiere die random Bibliothek, damit du die Funktion randint verwenden kannst

# definiere MAX_ATTEMPTS
```

```
# definiere eine Variable namens "want_to_play" und setze sie auf True
# solange der Spieler noch spielen möchte

  # dein aktueller funktionierender Spielcode kommt hierher

  # frage den Benutzer mit der input-Funktion, z.B. "Möchtest du nochmal spielen? J/N"
  # want_to_play sollte True sein, wenn die Benutzereingabe gleich "J" oder "j" ist

# allerletzte Nachricht, zum Beispiel "Vielen Dank fürs Spielen!"
```

**Achte besonders auf die Einrückungen, um den Code richtig zu gruppieren!**

Setze deinen Code in `code07.py`.

## 6.10 Du benötigst keinen Vergleich, wenn du bereits den Wert hast

In deinem aktualisierten Code hast du eine Variable `want_to_play`, die entweder `True` oder `False` ist. Sie wird in der Schleife verwendet, die sich wiederholt, solange ihr Wert `True` ist. Manchmal schreiben Leute `want_to_play == True`, um das auszudrücken. Obwohl es technisch korrekt und mit Sicherheit korrekt funktionieren wird, ist es auch redundant. Da `want_to_play` nur `True` oder `False` sein kann, verwandelt sich dieser Vergleich in `True == True` (was natürlich `True` ist) oder `False == True` (was `False` ist). Das Vergleichen eines jeden Werts mit `True` ergibt genau denselben Wert. Daher kannst du einfach `while want_to_play:` schreiben und den logischen Wert direkt verwenden.

## 6.11 Bestes Ergebnis

Ein "richtiges" Spiel behält normalerweise die Leistung der Spieler im Auge. Lass uns die geringste Anzahl von Versuchen aufzeichnen, die der Spieler benötigt hat, um die Zahl zu erraten. Dazu erstellst du eine neue Variable `fewest_attempts` und setzt sie auf `MAX_ATTEMPTS` (das ist so schlecht, wie der Spieler sein kann). Überlege, wo du sie erstellen musst. Du solltest sie nach jeder Spielrunde aktualisieren. Füge die Information über "Bisher das Beste" in die finale Rundennachricht ein.

Setze deinen Code in `code08.py`.

## 6.12 Zählen der Spielrunden

Lass uns zählen, wie oft der Spieler das Spiel gespielt hat. Die Idee und Umsetzung ist dieselbe wie beim Zählen der Versuche. Erstelle eine neue Variable,

initialisiere sie auf 0, erhöhe sie um 1, wann immer eine neue Runde beginnt. Füge die Gesamtanzahl der gespielten Spiele in die allerletzte Nachricht ein, z.B. “Danke, dass du das Spiel $X$ Mal gespielt hast!”

Setze deinen Code in `code09.py`.

## 6.13 Einarmiger Bandit mit mehreren Runden

Am Ende des vorherigen Kapitels hast du ein einarmiger Banditen-Spiel mit einer einzigen Runde programmiert. Du weißt bereits alles, was du brauchst, um eine Version mit mehreren Runden zu implementieren, und ihre Struktur ähnelt der des Mehr-Runden-Zahlenraten-Spiels, das du gerade implementiert hast, ist aber einfacher.

Lass den Spieler mit einem Anfangspot an Geld starten, sagen wir 10 Münzen. Jede Runde kostet 1 Münze, bei drei gleichen bekommt man 10 Münzen ausgezahlt, während man bei einem Paar 2 Münzen ausgezahlt bekommt (du kannst die Auszahlungen nach Belieben ändern). In jeder Runde:

- Nimm eine Münze aus dem Topf (Preis für das Spiel).
- Würfle die Würfel (das hast du bereits implementiert).
- Informiere den Spieler über das Ergebnis (das hast du auch schon implementiert).
- Füge Münzen zum Topf hinzu, falls nötig.
- Drucke die Anzahl der im Topf verbliebenen Münzen aus.
- Frage den Spieler, ob er weitermachen möchte.

Sonderfall: Wenn dem Spieler die Münzen ausgehen, ist das Spiel definitiv vorbei.

Setze deinen Code in `code10.py`.

## 6.14 Abschluss

Sehr gut, du hast jetzt zwei richtige funktionierende Computerspiele mit Spielrunden, begrenzten Versuchen, Bestleistung und vielem mehr! Pack den Ordner in eine Zip-Datei und reiche sie ein.

# Chapter 7

# Guess the Number: KI ist dran

Lass uns das Spiel Rate die Zahl *noch einmal* programmieren[1] aber die Rollen *umkehren*. Jetzt wählst *du* eine Zahl und der Computer wird versuchen, sie zu erraten. Denke über den Algorithmus nach, den ein Computer dafür verwenden könnte, bevor du den nächsten Absatz liest[2].

Der optimale Weg, dies zu tun, besteht darin, die Mitte des Intervalls für eine Vermutung zu verwenden. Auf diese Weise schließt du *die Hälfte* der Zahlen aus, die entweder größer oder kleiner sind als deine Vermutung (oder du errätst die Zahl korrekt, natürlich). Wenn du also weißt, dass die Zahl zwischen 1 und 10 liegt, solltest du die Dinge in der Mitte teilen, also 5 oder 6 wählen, da du nicht 5,5 wählen kannst (wir gehen davon aus, dass du nur ganze Zahlen verwenden kannst). Wenn dein Gegner dir sagt, dass seine Zahl größer ist als deine Wahl, weißt du, dass sie irgendwo zwischen deiner Vermutung und der ursprünglichen Obergrenze liegen muss, z. B. zwischen 5 und 10. Umgekehrt, wenn der Gegner "niedriger" antwortet, liegt die Zahl zwischen der unteren Grenze und deiner Vermutung, z. B. zwischen 1 und 5. Bei deinem nächsten Versuch teilst du das neue Intervall und wiederholst dies, bis du entweder die richtige Zahl errätst oder nur noch ein Intervall mit nur einer Zahl übrig hast. Dann musst du nicht mehr raten.

Um dieses Programm zu implementieren, musst du Funktionen kennen lernen, wie man sie dokumentiert und wie man eigene Bibliotheken verwendet. Hol dir das Übungsnotebook, bevor wir anfangen!

[1] Das ist das letzte Mal, versprochen!

[2] Du solltest dir vorstellen, dass ich Dora die Entdeckerin bin, die dich dabei beobachtet, wie du nachdenkst.

## 7.1 Konzepte des Kapitels.

- Schreiben deiner eigenen Funktionen.
- Verstehen von variablen Bereichen.
- Annahme von standardisierten Wegen, um deinen Code zu dokumentieren.
- Verwendung deiner eigenen Bibliotheken.

## 7.2 Spielerantwort

Lass uns warm werden, indem wir einen Code schreiben, der es einem Spieler ermöglicht, auf den Tipp des Computers zu reagieren. Denke daran, es gibt nur drei Möglichkeiten: deine Zahl ist größer, kleiner oder gleich der Vermutung des Computers. Ich würde vorschlagen, die Symbole `>`, `<` und `=` zu verwenden, um dies zu kommunizieren. Du musst den Code schreiben, der den Spieler auffordert, seine Antwort einzugeben, bis er eines dieser Symbole eingibt. D.h., die Aufforderung zur Eingabe sollte wiederholt werden, wenn sie etwas anderes eingeben. Du musst also definitiv die input([prompt]) und eine while Schleife verwenden. Überlege dir eine nützliche und informative Aufforderungsnachricht dafür. Teste, ob es funktioniert. Breakpoints können hier sehr hilfreich sein.

Füge deinen Code in `code01.py` ein.

## 7.3 Funktionen

Du weißt bereits, wie man Funktionen verwendet, jetzt ist es an der Zeit, mehr darüber zu lernen, warum du dich darum kümmern solltest. Der Zweck einer Funktion besteht darin, bestimmten Code, der eine einzige Berechnung durchführt, zu isolieren und ihn somit testbar und wiederverwendbar zu machen. Lass uns den letzten Satz Stück für Stück anhand von Beispielen durchgehen.

### 7.3.1 Eine Funktion führt eine einzige Berechnung durch

Ich habe dir bereits erzählt, dass das Lesen von Code einfach ist, weil jede Aktion für Computer auf einfache und klare Weise ausgedrückt werden muss. Allerdings können *viele* einfache Dinge sehr überwältigend und verwirrend sein. Denke an den finalen Code vom letzten Seminar: Wir hatten zwei Schleifen mit bedingten Anweisungen, die darin verschachtelt waren. Füge ein paar mehr hinzu und du hast so viele Zweige zu verfolgen, dass du nie ganz sicher sein kannst, was passieren wird. Das liegt daran, dass unsere Wahrnehmung und unser Arbeitsgedächtnis, die du benutzt, um alle Zweige zu verfolgen, auf nur etwa vier Elemente beschränkt sind[3].

Daher sollte eine Funktion *eine* Berechnung / Aktion durchführen, die konzeptionell klar ist und deren Zweck direkt aus ihrem Namen oder höchstens aus

[3]Die offizielle magische Zahl ist 7±2 aber wenn man das Originalpapier liest, stellt man fest, dass dies für die meisten von uns eher vier sind

einem einzelnen Satz, der sie beschreibt, verstanden werden sollte[4]. Der Name einer Funktion sollte typischerweise ein *Verb* sein, denn es geht darum, eine Aktion auszuführen. Wenn du mehr als einen Satz brauchst, um zu erklären, was die Funktion tut, solltest du erwägen, den Code weiter aufzuteilen. Das bedeutet nicht, dass die gesamte Beschreibung/Dokumentation in einen einzigen Satz passen muss. Die vollständige Beschreibung kann lang sein, insbesondere wenn die zugrunde liegende Berechnung komplex ist und es viele Parameter zu berücksichtigen gibt. Dies sind jedoch optionale Details, die dem Leser sagen, *wie* die Funktion ihre Aufgabe erledigt oder wie ihr Verhalten verändert werden kann. Trotzdem sollten sie in der Lage sein, zu verstehen, *was* die Aufgabe ist, nur aus dem Namen oder aus einem einzigen Satz. Ich wiederhole mich und betone dies so sehr, weil konzeptionell einfache Funktionen, die nur eine Aufgabe erfüllen, die Grundlage für einen klaren, robusten, wiederverwendbaren Code sind. Und das zukünftige Du wird sehr dankbar sein, dass es mit dem einfach zu verstehenden, isolierten, zuverlässigen Code arbeiten muss, den du geschrieben hast.

### 7.3.2 Funktion isoliert Code vom Rest des Programms

Isolierung bedeutet, dass dein Code in einem separaten Bereich ausgeführt wird, in dem nur Funktionselemente (begrenzte Anzahl von Werten, die du von außen mit fester Bedeutung übergibst) und lokale Variablen, die du innerhalb der Funktion definierst, existieren. Du hast keinen Zugriff auf Variablen, die im außenstehenden Skript definiert sind[5] oder auf Variablen, die in anderen Funktionen definiert sind. Umgekehrt haben weder das globale Skript noch andere Funktionen Zugriff auf Variablen und Werte, die du innen verwendest. Das bedeutet, dass du nur den Code *innerhalb* der Funktion studieren musst, um zu verstehen, wie sie funktioniert. Entsprechend sollte der Code, den du schreibst, *unabhängig* von jedem globalen Kontext sein, in dem die Funktion verwendet werden kann. Die Isolierung ist sowohl praktisch (kein Zugriff auf Variablen von außen während der Laufzeit bedeutet weniger Chancen, dass Dinge schrecklich schief gehen) als auch konzeptionell (kein weiterer Kontext ist erforderlich, um den Code zu verstehen).

### 7.3.3 Funktion macht Code einfacher zu testen

Du kannst sogar mäßig komplexe Programme nur dann erstellen, wenn du sicher sein kannst, was individuelle Codeabschnitte unter jeder möglichen Bedingung tun. Erzeugen sie die richtigen Ergebnisse? Scheitern sie deutlich und heben sie einen richtigen Fehler hervor, wenn die Eingaben falsch sind? Verwenden

[4]Dies ist ähnlich wie beim wissenschaftlichen Schreiben, wo ein einzelner Absatz eine einzelne Idee vermittelt. Mir hilft es, zuerst die Idee des Absatzes in einem einzigen Satz zu schreiben, bevor ich den Absatz selbst schreibe. Wenn ein Satz nicht ausreicht, muss ich den Text in mehrere Absätze aufteilen.

[5]Das ist streng genommen nicht wahr, aber das wird uns erst betreffen, wenn wir zu so genannten "veränderlichen" Objekten wie Listen oder Wörterbüchern kommen.

sie bei Bedarf Standardwerte? Wenn du jedoch alle Teile zusammen testest, bedeutet das eine extreme Anzahl von Durchläufen, da du alle möglichen Kombinationen von Bedingungen für einen Teil testen musst, gegeben alle möglichen Bedingungen für einen anderen Teil usw. Funktionen machen dein Leben viel einfacher. Da sie einen einzigen Einstiegspunkt, eine feste Anzahl von Parametern, einen einzigen Rückgabewert und Isolation (siehe oben) haben, kannst du sie unabhängig von anderen Funktionen und dem Rest des Codes einzeln testen. Dies nennt man *Unit Testing* und es ist ein intensiver Einsatz von automatischem Unit Testing[6] das sicherstellt, dass der Code für die absolute Mehrheit der Programme und Apps, die du verwendest, zuverlässig ist[7].

### 7.3.4 Funktion macht Code wiederverwendbar

Manchmal wird dies als Hauptgrund angegeben, Funktionen zu verwenden. Wenn du den Code in eine Funktion umwandelst, bedeutet das, dass du die Funktion aufrufen kannst, anstatt den Code zu kopieren und einzufügen. Der letztere Ansatz ist eine schreckliche Idee, da es bedeutet, dass du denselben Code an vielen Stellen pflegen musst und du vielleicht nicht einmal sicher bist, an wie vielen. Das ist ein Problem, selbst wenn der Code extrem einfach ist. Hier definieren wir einen *standardisierten* Weg, ein Initial zu berechnen, indem wir das erste Symbol aus einer Zeichenkette nehmen (du wirst später mehr über Indexierung und Slicing erfahren). Der Code ist so einfach wie möglich.

```
...
initial = "test"[0]
...
initial_for_file = filename[0]
...
initial_for_website = first_name[0]
...
```

Stell dir vor, du hast beschlossen, es zu ändern und die ersten *zwei* Symbole zu verwenden. Wiederum ist die Berechnung nicht kompliziert, verwende einfach `[0]` ersetzen mit `[:2]`. Du musst es aber für *allen* Code tun, der diese Berechnung durchführt. Und du kannst die Option *Alles Ersetzen* nicht verwenden, weil du manchmal das erste Element für andere Zwecke verwenden könntest. Und wenn du den Code bearbeitest, wirst du zwangsläufig einige Stellen vergessen (ich mache das die ganze Zeit), was die Dinge noch inkonsistenter und verwirrender macht. Code in eine Funktion zu verwandeln bedeutet, dass du nur an *einer* Stelle ändern und testen musst. Hier ist der ursprüngliche Code, der über eine Funktion implementiert wurde.

```
def generate_initial(full_string):
    """Erzeugt ein Initial mit dem ersten Symbol.
```

[6] Es ist normal, mehr Code für Tests zu haben als für das eigentliche Programm.

[7] Man braucht trotzdem noch Tests für das integrierte System, aber das Testen einzelner Funktionen ist eine klare Voraussetzung.

```
    Parameter
    ----------
    full_string : str

    Gibt zurück
    ----------
    str : einzelnes Symbol
    """
    return full_string[0]

...
initial = generate_initial("test")
...
initial_for_file = generate_initial(filename)
...
initial_for_website = generate_initial(first_name)
...
```

and here is the "alternative" initial computation. Note that the code that uses the function *stays the same* und hier ist die "alternative" Initialberechnung. Beachte, dass der Code, der die Funktion verwendet, *gleich bleibt*

```
def generate_initial(full_string):
    """Erzeugt ein Initial mit den ersten ZWEI Symbolen.

    Parameter
    ----------
    full_string : str

    Gibt zurück
    ----------
    str : zwei Symbole lang
    """
    return full_string[:2]

...
initial = generate_initial("test")
...
initial_for_file = generate_initial(filename)
...
initial_for_website = generate_initial(first_name)
...
```

Daher ist es besonders nützlich, Code in eine Funktion umzuwandeln, wenn der wiederverwendete Code komplex ist, aber es zahlt sich sogar aus, wenn die Berechnung so einfach und trivial ist wie im obigen Beispiel. Mit einer Funktion

musst du dir nur um einen einzigen Codeblock Gedanken machen und du kannst sicher sein, dass die gleiche Berechnung immer dann ausgeführt wird, wenn du die Funktion aufrufst (und dass dies nicht mehrere Kopien des Codes sind, die möglicherweise identisch sind oder nicht).

Beachte, dass ich den wiederverwendbaren Code als den letzten und den am wenigsten wichtigen Grund zur Verwendung von Funktionen putze. Dies liegt daran, dass die anderen drei Gründe weitaus wichtiger sind. Konzeptuell klaren, isolierten und testbaren Code zu haben, ist vorteilhaft, selbst wenn du diese Funktion nur einmal aufrufst. Es erleichert das Verständnis und das Testen des Codes und hilft dir, seine Komplexität zu reduzieren, indem du Codeblöcke durch deren Bedeutung ersetzt. Sieh dir das folgende Beispiel an. Der erste Code nimmt das erste Symbol, aber diese Aktion (das erste Symbol nehmen) *bedeutet* an sich nichts, es ist nur eine mechanische Berechnung. Es ist nur der ursprüngliche Kontext `initial_for_file = filename[0]` oder zusätzliche Kommentare, die ihm seine Bedeutung geben. Im Gegensatz dazu sagt dir das Aufrufen einer Funktion namens *compute_initial* was passiert, da es den Zweck eindeutig macht. Ich vermute, dass der zukünftige Du sehr pro-Eindeutigkeit und anti-Verwirrung ist.

```
if filename[0] == "A":
    ...

if compute_initial(filename) == "A":
    ...
```

## 7.4 Funktionen in Python

### 7.4.1 Definieren einer Funktion in Python

Eine Funktion in Python sieht so aus (beachte die Einrückung und `:` am Ende der ersten Zeile)

```
def <funktionsname>(param1, param2, ...):
    einige interne Berechnung
    if bedingung:
        return ein Wert
    return ein anderer Wert
```

Die Parameter sind optional, ebenso der Rückgabewert. Die minimalste Funktion wäre daher

```
def minimal_function():
    pass # pass bedeutet "mache nichts"
```

Du musst deine Funktion (einmal!) definieren, bevor du sie aufrufst (ein- oder mehrmals). Du solltest also Funktionen *vor* dem Code erstellen, der sie verwendet.

```
def do_something():
    """
    Das ist eine Funktion namens "do_something". Sie macht eigentlich nichts.
    Sie benötigt keine Eingabe und gibt keinen Wert zurück.
    """
    return

def another_function():
    ...
    # Wir rufen sie in einer anderen Funktion auf.
    do_something()
    ...

# Das ist ein Funktionsaufruf (wir verwenden diese Funktion)
do_something()

# Und wir verwenden sie noch einmal!
do_something()

# Und wieder, aber durch einen another_function Aufruf
another_function()
```

Mache Übung #1.

Du musst auch im Hinterkopf behalten, dass das erneute Definieren einer Funktion (oder das Definieren einer technisch anderen Funktion, die den gleichen Namen hat) die ursprüngliche Definition *überschreibt*, so dass nur die *neueste* Version davon beibehalten und verwendet werden kann.

Mache Übung #2.

Obwohl das Beispiel in der Übung das Problem leicht erkennen lässt, kann es in einem großen Code, der sich über mehrere Dateien erstreckt und verschiedene Bibliotheken verwendet, nicht so einfach sein, das gleiche Problem zu lösen!

### 7.4.2 Funktionsargumente

Einige Funktionen benötigen möglicherweise keine Argumente (auch Parameter genannt), da sie eine feste Aktion ausführen:

```
def ping():
    """
    Maschine, die "ping!" macht.
    """
    print("ping!")
```

Du musst jedoch möglicherweise Informationen an die Funktion über Argumente weitergeben, um zu beeinflussen, wie die Funktion ihre Aktion ausführt. In

Python listest du einfach Argumente innerhalb der runden Klammern nach dem Funktionsnamen auf (es gibt noch mehr Optionen und Funktionen, aber wir halten es zunächst einfach). Beispielsweise könnten wir eine Funktion schreiben, die das Alter einer Person berechnet und ausgibt, basierend auf zwei Parametern 1) ihrem Geburtsjahr, 2) dem aktuellen Jahr:

```
def print_age(birth_year, current_year):
    """
    Gibt das Alter aus, gegeben das Geburtsjahr und das aktuelle Jahr.

    Parameter
    ----------
    birth_year : int
    current_year : int
    """
    print(current_year - birth_year)
```

Es ist eine **sehr gute Idee**, Funktionen, Parametern und Variablen aussagekräftige Namen zu geben. Der folgende Code wird genau das gleiche Ergebnis liefern, aber das Verständnis *warum* und *wofür* es das tut, was es tut, wäre viel schwieriger (also benutze **immer** aussagekräftige Namen!):

```
def x(a, b):
    print(b - a)
```

Wenn du eine Funktion aufrufst, musst du die korrekte Anzahl von Parametern übergeben und sie in der *richtigen Reihenfolge* übergeben, ein weiterer Grund für Funktionen Argumente, aussagekräftige Namen zu haben[8].

Mache Übung #3.

Wenn du eine Funktion aufrufst, werden die Werte, die du an die Funktion *übermittelst*, den Parametern zugewiesen und sie werden als *lokale* Variablen verwendet (mehr darüber später). Es spielt jedoch keine Rolle, *wie* du diese Werte erlangt hast, ob sie in einer Variable waren, hart codiert wurden oder von einer anderen Funktion zurückgegeben wurden. Wenn du numerische, logische oder Zeichenkettenwerte (*unveränderliche* Typen) verwendest, kannst du davon ausgehen, dass jede Verbindung zur ursprünglichen Variablen oder Funktion, die sie erzeugt hat, verschwunden ist (wir werden uns später um *veränderliche* Typen wie Listen kümmern). Wenn du also eine Funktion schreibst oder ihren Code liest, gehst du einfach davon aus, dass sie bei dem Aufruf auf einen bestimmten Wert gesetzt wurde und du kannst den Kontext, in dem dieser Aufruf gemacht wurde, ignorieren.

```
# hart codiert
print_age(1976, 2020)
```

[8] Dies ist auch nicht streng wahr, aber du musst warten, bis du über benannte Parameter und Standardwerte lernst

```
# mit Werten aus Variablen
i_was_born = 1976
today_is = 2023
print_age(i_was_born, today_is)

# mit Wert aus einer Funktion
def get_current_year():
    return 2023

print_age(1976, get_current_year())
```

### 7.4.3 Rückgabewert der Funktionen (Ausgabe)

Deine Funktion kann eine Aktion ausführen, ohne einen Wert an den Aufrufer zurückzugeben (das ist es, was unsere Funktion `print_age` getan hat). Du könntest jedoch den Wert stattdessen zurückgeben müssen. Um die Dinge allgemeiner zu gestalten, möchten wir vielleicht eine neue Funktion namens `compute_age` schreiben, die das Alter zurückgibt, anstatt es zu drucken (wir können es immer selbst drucken).

```
def compute_age(birth_year, current_year):
    """
    Berechnet das Alter, gegeben das Geburtsjahr und das aktuelle Jahr.

    Parameter
    ----------
    birth_year : int
    current_year : int

    Gibt zurück
    ----------
    int : age
    """
    return current_year - birth_year
```

Beachte, dass selbst wenn eine Funktion den Wert zurückgibt, dieser nur beibehalten wird, wenn er tatsächlich verwendet wird (in einer Variablen gespeichert, als Wert verwendet etc.). Ein einfacher Aufruf speichert den zurückgegebenen Wert also nicht irgendwo!

Mache Übung #4.

### 7.4.4 Anwendungsbereiche (für unveränderbare Werte)

Wie wir oben besprochen haben, isoliert das Umwandeln von Code in eine Funktion ihn, sodass er in seinem eigenen *Anwendungsbereich* läuft. In Python ex-

istiert jede Variable in dem *Anwendungsbereich*, in dem sie definiert wurde. Wenn sie im *globalen* Skript definiert wurde, existiert sie in diesem *globalen* Anwendungsbereich als *globale* Variable. Sie ist jedoch nicht zugänglich (zumindest nicht ohne besondere Anstrengungen über einen `global` Operator) innerhalb einer Funktion. Umgekehrt existieren die Parameter einer Funktion und alle *innerhalb einer Funktion* definierten Variablen nur **innerhalb dieser Funktion**. Sie sind für die Außenwelt vollkommen unsichtbar und können nicht von einem globalen Skript oder von einer anderen Funktion aus abgerufen werden. Ebenso haben alle Änderungen, die du am Funktionsparameter oder an der lokalen Variable vornimmst, keinerlei Auswirkungen auf die Außenwelt.

Der Zweck von Anwendungsbereichen besteht darin, einzelne Codeabschnitte voneinander zu isolieren, sodass das Ändern von Variablen innerhalb eines Anwendungsbereichs keine Auswirkungen auf alle anderen Anwendungsbereiche hat. Das bedeutet, dass du beim Schreiben oder Debuggen des Codes keine Angst vor Code in anderen Anwendungsbereichen haben musst und dich nur auf den Code konzentrieren musst, an dem du arbeitest. Da Anwendungsbereiche isoliert sind, können sie *gleichnamige Variablen* haben, die jedoch keine Beziehung zueinander haben, da sie in ihren eigenen parallelen Universen existieren[9]. Wenn du also wissen möchten, welchen Wert eine Variable hat, musst du nur innerhalb des Anwendungsbereichs schauen und alle anderen Anwendungsbereiche ignorieren (auch wenn die Namen übereinstimmen!).

```
# dies ist die Variable `x` im globalen Anwendungsbereich
x  = 5

def f1():
  # Das ist die Variable `x` im Anwendungsbereich der Funktion f1
  # Sie hat den gleichen Namen wie die globale Variable, aber
  # hat keine Beziehung zu ihr: viele Leute heißen Sasha,
  # sind aber immer noch unterschiedliche Personen. Was auch immer
  # mit `x` in f1 passiert, bleibt im Anwendungsbereich von f1.
  x = 3


def f2(x):
  # Dies ist der Parameter `x` im Anwendungsbereich der Funktion f2.
  # Wieder keine Beziehung zu anderen globalen oder lokalen Variablen.
  # Es ist ein völlig getrenntes Objekt, es passiert einfach,
  # dass es den gleichen Namen hat (wieder einfach Namensvettern)
  print(x)
```

Mache Übung #5.

[9] Es ist wie zwei Personen mit identischen Namen, immer noch unterschiedliche Personen.

## 7.5 Spielerantwort als Funktion

Lassen wir all diese Theorie über Funktionen in die Praxis umsetzen. Verwende den Code, den du erstellt hast, um die Spielerantwort zu erhalten und verwandle ihn in eine Funktion. Sie sollte keine Parameter haben (zumindest vorerst) und sollte die Spielerantwort zurückgeben. Ich schlage vor, dass du sie `input_response` (oder so ähnlich) nennst. Überprüfe, ob der Code funktioniert, indem du diese Funktion aus dem Hauptskript aufrufst.

Füge deinen Code in `code02.py` ein.

## 7.6 Eine Funktion debuggen

Jetzt, da du deine erste Funktion hast, kannst du den Sinn von drei Schritt über/Schritt in/Schritt heraus-Tasten verstehen, die dir der Debugger anbietet. Kopiere den folgenden Code in eine separate Datei (nenne sie zum Beispiel `test01.py`).

```
def f1(x, y):
  return x / y

def f2(x, y):
  x = x + 5
  y = y * 2
  return f1(x, y)

z = f2(4, 2)
print(z)
```

Setze zuerst einen Breakpoint auf die Zeile im Hauptskript, die die Funktion `f2()` aufruft. Starte den Debugger über **F5** und das Programm wird an dieser Zeile anhalten. Wenn du nun **F10** drückst (Schritt über), geht das Programm zur nächsten Zeile `print(z)`. Wenn du aber stattdessen **F11** drückst (Schritt hinein), wird das Programm *in* die Funktion hineingehen und zur Zeile `x = x + 5` gehen. Innerhalb der Funktion hast du die gleichen beiden Möglichkeiten, die wir gerade angeschaut haben, aber du kannst auch **Shift+F11** drücken, um aus der Funktion herauszugehen. Hier wird das Programm den gesamten Code durchlaufen, bis du die nächste Zeile *außerhalb* der Funktion erreichst (du solltest wieder bei `print(z)` ankommen). Experimentiere mit dem Setzen von Breakpoints an verschiedenen Zeilen und dem Über-/Hinein-/Herausschreiten, um dich mit diesen nützlichen Debugging-Tools vertraut zu machen.

Setze nun den Breakpoint innerhalb der `f1()` Funktion und führe den Code über **F5** aus. Schau dir die linke Fensterseite an, du wirst einen *Call Stack* Reiter sehen. Während die gelb hervorgehobene Zeile im Editor zeigt, wo du gerade bist (sollte innerhalb der `f1()` Funktion sein), zeigt der *Call Stack* dir, wie du dorthin gekommen bist. In diesem Fall sollte es zeigen:

| | | |
|---|---|---|
| f1 | test01.py | 2:1 |
| f2 | test01.py | 7:1 |
| <module> | test01.py | 9:1 |

Die Aufrufe sind von unten nach oben gestapelt, das bedeutet, dass eine Funktion im Hauptmodul in Zeile 9 aufgerufen wurde, du in der Funktion `f2` in Zeile 7 gelandet bist und dann in der Funktion `f1` und in Zeile 2. Experimentiere mit dem Ein- und Austreten aus Funktionen und behalte dabei den *Call Stack* im Auge. Du wirst diese Informationen vielleicht nicht oft benötigen, aber sie könnten in unseren späteren Projekten mit mehreren verschachtelten Funktionsaufrufen nützlich sein.

## 7.7 Deine Funktion dokumentieren

Eine Funktion zu schreiben, ist nur die halbe Arbeit. Du musst sie dokumentieren! Erinnere dich, das ist eine gute Gewohnheit, die deinen Code einfach zu benutzen und wiederzuverwenden macht. Es gibt verschiedene Möglichkeiten, den Code zu dokumentieren, aber wir werden die NumPy Docstring Konvention verwenden. Hier ist ein Beispiel für eine solche dokumentierte Funktion

```
def generate_initial(full_string):
    """Erzeugt eine Initiale mit dem ersten Symbol.

    Parameter
    ----------
    full_string : str

    Gibt zurück
    ----------
    str : Einzelnes Symbol
    """
    return full_string[0]
```

Schau dir das Handbuch an und dokumentiere die `input_response` Funktion. Du wirst den Abschnitt `Parameter` nicht benötigen, da sie derzeit keine Eingaben akzeptiert.

Aktualisiere deinen Code in `code02.py`.

## 7.8 Verwendung der Aufforderung

In der Zukunft werden wir nach einer bestimmten Zahl fragen, die eine aktuelle Vermutung des Computers ist, daher können wir keine feste Aufforderungsnachricht verwenden. Ändere die `input_response` Funktion,

indem du einen `guess` Parameter hinzufügst. Dann ändere die Aufforderung, die du für die input() verwendet hast, um den Wert in diesem Parameter einzuschließen. Aktualisiere die Dokumentation der Funktionen. Teste es, indem du mit verschiedenen Werten für den `guess` Parameter aufrufst und eine unterschiedliche Aufforderung zur Antwort siehst.

Gib deinen Code in `code03.py` ein.

## 7.9 Aufteilen des Intervalls in der Mitte

Lass uns ein bisschen mehr üben, Funktionen zu schreiben. Denke daran, dass der Computer die Mitte des Intervalls als Vermutung verwenden sollte. Erstelle eine Funktion (nennen wir sie `split_interval()` oder so ähnlich), die zwei Parameter – `lower_limit` und `upper_limit` - entgegennimmt und *eine Ganzzahl* zurückgibt, die der Mitte des Intervalls am nächsten liegt. Der einzige knifflige Teil ist, wie du eine potentiell Gleitkommazahl (z.B. wenn du versuchst, sie für das Intervall 1..10 zu finden) in eine Ganzzahl umwandelst. Du kannst dafür die Funktion int() verwenden. Lies jedoch die Dokumentation sorgfältig durch, da sie *keine* korrekte Rundung durchführt (was tut sie? Lies die Dokumentation!). Du solltest daher die Zahl auf die nächste Ganzzahl round() runden, bevor du sie umwandelst.

Schreibe eine Funktion, dokumentiere sie und teste sie, indem du überprüfst, ob die Zahlen korrekt sind.

Gib deine `split_interval()` Funktion und den Testcode in `code04.py` ein.

## 7.10 Einzelne Runde

Du hast beide Funktionen, die du brauchst, also lass uns den Code schreiben, um das Spiel zu initialisieren und eine einzelne Runde zu spielen. Die Initialisierung läuft darauf hinaus, zwei Variablen zu erstellen, die den unteren und oberen Grenzwerten des Spielbereichs entsprechen (wir haben bisher 1 bis 10 verwendet, aber das kannst du natürlich immer ändern). Als nächstes sollte der Computer eine Vermutung generieren (dafür hast du deine `split_interval()` Funktion) und den Spieler nach der Vermutung fragen (das ist die `input_response()` Funktion). Sobald du die Antwort hast (in einer separaten Variable gespeichert, denke dir selbst einen Namen aus), aktualisiere entweder die obere oder untere Grenze mit einem if..elif..else Ausdruck, basierend auf der Antwort des Spielers (wenn der Spieler gesagt hat, dass seine Zahl höher ist, bedeutet das, dass das neue Intervall von `guess` bis `upper_limit` reicht, und umgekehrt, wenn sie niedriger ist). Drucke eine freudige Nachricht aus, wenn die Vermutung des Computers richtig war.

Gib beide Funktionen und den Skriptcode in `code05.py` ein.

## 7.11 Mehrere Runden

Erweitere das Spiel, so dass der Computer so lange rät, bis er schließlich gewinnt. Du weißt bereits, wie man die while Schleife verwendet, überlege nur, wie du die Antwort des Teilnehmers als eine Schleifenbedingungsvariable verwenden kannst. Denke auch über den Anfangswert dieser Variable nach und wie du sie verwenden kannst, um `input_response()` nur an einer Stelle aufzurufen.

Gib den aktualisierten Code in `code06.py` ein.

## 7.12 Nochmal spielen

Ändere den Code so, dass du dieses Spiel mehrere Male spielen kannst. Du weißt bereits, wie das geht und das Einzige, was du beachten musst, ist, wo genau du die Initialisierung vor jedem Spiel durchführen sollst. Da du das schon für das letzte Spiel implementiert hast, könntest du versucht sein zu sehen, wie du es gemacht hast oder sogar den Code zu kopieren und einzufügen. Allerdings würde ich empfehlen, es von Grund auf neu zu schreiben. Denke daran, dein Ziel ist es nicht, ein Programm zu schreiben, sondern zu lernen, wie man das macht und daher ist der Weg wichtiger als das Ziel.

Gib den aktualisierten Code in `code07.py` ein.

## 7.13 Bester Punktestand

Füge den Code hinzu, um die Anzahl der Versuche zu zählen, die der Computer in jeder Runde benötigt hat, und melde den besten Punktestand (wenigste Anzahl von Versuchen), nachdem das Spiel vorbei ist. Du wirst eine Variable brauchen, um die Anzahl der Versuche zu zählen, und eine, um den besten Punktestand zu speichern. Versuche erneut, es zu schreiben, ohne auf dein vorheriges Spiel zu schauen.

Gib den aktualisierten Code in `code08.py` ein.

## 7.14 Verwenden deiner eigenen Bibliotheken

Du weißt bereits, wie man bestehende Bibliotheken verwendet, aber du kannst auch deine eigenen erstellen und verwenden. Nimm die beiden Funktionen, die du entwickelt hast, und lege sie in eine neue Datei namens `utils.py` (vergiss nicht, einen mehrzeiligen Kommentar am Anfang der Datei hinzuzufügen, um dich daran zu erinnern, was drin ist!). Kopiere den verbleibenden Code (das globale Skript) in `code09.py`. Es wird in seinem aktuellen Zustand nicht funktionieren, da es die beiden Funktionen nicht finden wird (versuche es, um die Fehlermeldung zu sehen), daher musst du aus deinem eigenen Modul `utils` importieren. Das Importieren funktioniert genau so wie bei anderen Bibliotheken.

Beachte, dass der Name deines Moduls `utils` ist (ohne die Endung), auch wenn deine Datei `utils.py` heißt.

Gib die Funktionen in `utils.py` ein, den verbleibenden Code in `code09.py`.

## 7.15 Ordnung muss sein!

Bisher hast du höchstens eine Bibliothek importiert. Da Python jedoch hoch modular ist, ist es sehr verbreitet, viele Importe in einer einzelnen Datei zu haben. Es gibt einige Regeln, die es einfacher machen, die Importe zu verfolgen. Wenn du Bibliotheken importierst, sollten alle Import-Anweisungen oben in deiner Datei stehen und du solltest vermeiden, sie in zufälliger Reihenfolge zu platzieren. Die empfohlene Reihenfolge ist 1) Systembibliotheken, wie `os` oder `random`; 2) Drittanbieter-Bibliotheken, wie `psychopy`; 3) deine Projektmodule. Und innerhalb jedes Abschnitts solltest du die Bibliotheken *alphabetisch* anordnen, sodass beispielsweise

```
import os
import random
```

Das mag für unseren einfachen Code nicht besonders nützlich aussehen, aber wenn deine Projekte wachsen, musst du mehr und mehr Bibliotheken einbeziehen. Wenn du sie in dieser Reihenfolge hältst, ist es einfach zu verstehen, welche Bibliotheken du verwendest und welche nicht standardmäßig sind. Die alphabetische Reihenfolge bedeutet, dass du schnell überprüfen kannst, ob eine Bibliothek enthalten ist, da du schnell den Ort finden kannst, an dem ihre Import-Anweisung erscheinen sollte.

## 7.16 Videospiele mit Video

Reiche deine Dateien ein und mach dich bereit für mehr Aufregung, denn wir wechseln zu “richtigen” Videospielen mit PsychoPy.

# Chapter 8

# Einarmiger Bandit, aber mit Listen und Schleifen

In diesem Kapitel werden wir das Spiel des Einarmigen Banditen mehrmals neu programmieren. Das ist vielleicht nicht die aufregendste Aussicht, aber es gibt uns ein einfaches Spiel, das du bereits zu programmieren weißt, sodass wir uns auf Listen und For-Schleifen konzentrieren können. Hol dir das Übungsnotebook, bevor wir starten.

## 8.1 Konzepte des Kapitels

- Speichern von vielen Elementen in Listen.
- Iterieren über Elemente mit der for Schleife.
- Erstellen eines Bereichs von Werten.
- Schleifen über nummerierte Indizes und Werte.
- Listen erstellen mit Listenabstraktion.

## 8.2 Listen

Bisher haben wir Variablen verwendet, um einzelne Werte zu speichern: die Wahl des Computers, die Vermutung des Spielers, die Anzahl der Versuche, das PsychoPy-Fensterobjekt usw. Aber manchmal müssen wir mehr als einen Wert verarbeiten. Dieses Problem hatten wir bereits im computerbasierten Raten-der-Zahl Spiel, als wir den verbleibenden Zahlenbereich speichern mussten. Wir haben das Problem gelöst, indem wir zwei Variablen verwendet haben, eine für die untere und eine für die obere Grenze. Allerdings ist klar, dass dieser Ansatz nicht gut skalierbar ist und manchmal wissen wir vielleicht nicht einmal, wie viele Werte wir speichern müssen. Die Listen von Python sind die Lösung für dieses Problem.

Eine Liste ist eine veränderbare[1] Sequenz von Elementen, auf die über ihren nullbasierten Index zugegriffen werden kann. Wenn du die Idee der Variablen-als-Box fortschreitest, kannst du dir Listen als eine Box mit nummerierten Steckplätzen vorstellen. Um ein bestimmtes Stück zu speichern und abzurufen, musst du sowohl den *Variablennamen* als auch den *Index des Elements*, an dem du interessiert bist, in dieser Box kennen. Dann arbeitest du mit einer Variable-plus-Index auf genau dieselbe Weise wie mit einer normalen Variable, indem du auf ihren Wert zugreifst oder ihn änderst, genau wie vorher.

Eine Liste wird durch eckige Klammern definiert `<variable> = [<wert1>, <wert2>, ... <wertN>]`. Auf einen einzelnen Platz innerhalb einer Liste wird ebenfalls über eckige Klammern zugegriffen `<variable>[<index>]`, wobei der Index wiederum **nullbasiert** ist[2]. Das bedeutet, dass das *erste* Element `variable[0]` ist und, wenn es $N$ Elemente in der Liste gibt, ist das letzte Element `variable[N-1]`. Du kannst die Gesamtzahl der Elemente in einer Liste herausfinden, indem du ihre Länge über eine spezielle len() Funktion erhältst. So kannst du auf das letzte Element über `variable[len(variable)-1]` zugreifen[3]. Beachte die `-1`: Wenn deine Liste 3 Elemente hat, ist der Index des letzten Elements 2, wenn sie 100 hat, dann 99, usw. Ich verbringe so viel Zeit damit, weil dies eine recht häufige Quelle von Verwirrung ist.

Mache Übung #1 um zu sehen, wie Listen definiert und indiziert werden.

Listen erlauben dir auch, auf mehr als einen Slot/Index gleichzeitig zuzugreifen, und zwar über slicing. Du kannst den Index der Elemente über die Notation `<start>:<stop>` angeben. Zum Beispiel gibt dir `x[1:3]` Zugang zu zwei Elementen mit den Indizes 1 und 2. Ja, *zwei* Elemente: der Slicing-Index geht vom `start` bis **aber nicht einschließlich** zum `stop`. Wenn du also *alle* Elemente einer Liste abrufen willst, musst du `x[0:length(x)]` schreiben, und um nur das letzte Element zu bekommen, schreibst du immer noch `x[len(x)-1]`. Verwirrend? Ich denke schon! Ich verstehe die Logik, aber ich finde dieses stop-is-not-included Konzept gegenintuitiv und ich muss mich immer wieder bewusst daran erinnern. Leider ist dies die Standardmethode, um Zahlenfolgen in Python zu definieren, daher musst du dir dies merken.

Mache Übung #2, um die Intuition zu entwickeln.

Beim Slicing kannst du entweder `start` oder `stop` weglassen. In diesem Fall geht Python davon aus, dass ein fehlendes `start 0` bedeutet (der Index des ersten Elements) und ein fehlendes `stop len(<list>)` bedeutet (also ist das letzte Element eingeschlossen). Wenn du *beide* weglässt, z.B. `meine_schönen_zahlen[:]`, gibt es alle Werte zurück, da dies äquivalent zu

[1] Mehr dazu und zu Tupeln, den unveränderlichen Verwandten der Liste, später.

[2] Das ist typisch für "klassische" Programmiersprachen, aber weniger für solche, die auf lineare Algebra / Data Science ausgerichtet sind. Sowohl Matlab als auch R verwenden einen indexbasierten Index, so dass du vorsichtig sein und überprüfen musst, ob du die richtigen Indizes verwendest.

[3] Es gibt einen einfacheren Weg, dies zu tun, den du in Kürze lernen wirst.

`meine_schönen_zahlen[0:len(meine_schönen_zahlen)]` ist.[4]

Mache Übung #3.

Du kannst auch *negative* Indizes verwenden, die relativ zur Länge der Liste berechnet werden[5]. Wenn du zum Beispiel das *letzte* Element der Liste abrufen willst, kannst du `meine_schönen_zahlen[len(meine_schönen_zahlen)-1]` oder einfach nur `meine_schönen_zahlen[-1]` sagen. Das vorletzte Element wäre `meine_schönen_zahlen[-2]`, usw. Du kannst negative Indizes für das Slicing verwenden, aber vergiss nicht den Haken *einschließlich-dem-Start-aber-ausgenommen-dem-Stopp*: `meine_schönen_zahlen[:-1]` gibt alle Elemente außer dem letzten Element der Liste zurück, nicht die gesamte Liste!

Mache Übung #4.

Das Slicing kann erweitert werden, indem ein `Schritt` über die Notation `start:stop:Schritt` angegeben wird. `Schritt` kann negativ sein, was ermöglicht, Indizes in umgekehrter Reihenfolge zu erstellen:

```
meine_schönen_zahlen = [1, 2, 3, 4, 5, 6, 7]
meine_schönen_zahlen[4:0:-1]
#> [5, 4, 3, 2]
```

Aber du musst auf das Vorzeichen des Schritts achten. Wenn es in die falsche Richtung geht und `stop` nicht erreicht werden kann, gibt Python eine leere Liste zurück.

```
meine_schönen_zahlen = [1, 2, 3, 4, 5, 6, 7]
meine_schönen_zahlen[4:0:1]
#> []
```

Schritte können mit ausgelassenen und negativen Indizes kombiniert werden. Um jedes *ungerade* Element der Liste zu bekommen, schreibst du `meine_schönen_zahlen[::2]`:

```
meine_schönen_zahlen = [1, 2, 3, 4, 5, 6, 7]
meine_schönen_zahlen[::2]
#> [1, 3, 5, 7]
```

Mache Übung #5.

Wenn du versuchst, auf Indizes *außerhalb* eines gültigen Bereichs zuzugreifen, wird Python einen IndexError[6] auslösen. Wenn du also versuchst, das 6$^{\text{te}}$ Element (Index 5) einer fünfelementigen Liste zu holen, wird ein einfacher und

[4] Beachte, dass dies fast, aber nicht ganz das Gleiche ist wie einfach nur `meine_schönen_zahlen` zu schreiben, denn `meine_schönen_zahlen[:]` gibt eine *andere* Liste mit *identischem* Inhalt zurück. Der Unterschied ist subtil, aber wichtig und wir werden später darauf zurückkommen, wenn wir über veränderbare und unveränderbare Typen sprechen.

[5] Wenn du von R kommst, ist das negative Indexing in Python völlig anders.

[6] Wenn du mit R und seiner laxen Haltung gegenüber Indizes vertraut bist, wirst du das sehr befriedigend finden.

unmissverständlicher Fehler generiert. Wenn jedoch deine *Scheibe* größer ist als der Bereich, wird sie ohne zusätzliche Warnung oder einen Fehler abgeschnitten. Daher wird für eine fünfelementige Liste `my_pretty_numbers[:6]` oder `my_pretty_numbers[:600]` beide alle Nummern zurückgeben (effektiv ist dies gleichwertig mit `my_pretty_numbers[:]`). Zudem, wenn die Scheibe leer ist (`2:2`, kann 2 nicht einschließen, weil es ein Stop-Wert ist, obwohl es auch von 2 aus startet) oder die gesamte Scheibe außerhalb des Bereichs liegt, wird Python eine leere Liste zurückgeben, wiederum wird weder eine Warnung noch ein Fehler generiert.

Mache Übung #6.

In Python sind Listen dynamisch, daher kannst du immer Elemente hinzufügen oder entfernen, siehe die Liste der Methoden. Du kannst ein neues Element am Ende der Liste über die Methode `.append(<new_value>)` hinzufügen

```
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers.append(10)
my_pretty_numbers
#> [1, 2, 3, 4, 5, 6, 7, 10]
```

Oder du kannst `insert(<index>, <new_value>)` *vor* einem Element mit diesem Index verwenden. Leider bedeutet dies, dass du einen beliebig großen Index verwenden kannst und es fügt einen neuen Wert als *letztes* Element ein, ohne einen Fehler zu erzeugen.

```
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers.insert(2, 10)
my_pretty_numbers.insert(500, 20)
my_pretty_numbers
#> [1, 2, 10, 3, 4, 5, 6, 7, 20]
```

Du kannst ein Element mit seinem Index über `pop(<index>)` entfernen, beachte, dass das Element auch *zurückgegeben* wird. Wenn du den Index weglässt, entfernt `pop()` das *letzte* Element der Liste. Hier kannst du nur gültige Indizes verwenden.

```
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers.pop(-1)
#> 7
my_pretty_numbers.pop(3)
#> 4
my_pretty_numbers
#> [1, 2, 3, 5, 6]
```

Mache Übung #7.

## 8.3 Neukonzeption des Einarmigen Banditen-Spiels mit Listen

Puh, das war *viel* über Listen[7]. Aber Work hard, play hard! Lass uns zum Einarmigen Banditen-Spiel zurückkehren, das du bereits implementiert hast, und es mithilfe von Listen neu gestalten. Letztere machen viel Sinn, wenn man mit mehreren Slots arbeitet. Erinnere dich an die Regeln: Du hast drei Slots mit zufälligen Zahlen zwischen 1 und 5, wenn alle drei Zahlen übereinstimmen, gibst du "Drei Gleiche!" aus, wenn nur zwei Zahlen übereinstimmen, druckst du "Ein Paar!". Implementiere dieses Spiel mit einer Liste anstelle von drei Variablen. In der ersten Version, verwende die Notation `<variable> = [<value1>, <value2>, ..., <valueN>]`, um sie zu definieren. Beachte außerdem, dass du bei der Verwendung von String-Formatierung alle Werte in einem Tuple (eine eingefrorene Liste, mehr dazu in späteren Kapiteln) oder einer Liste übergibst. Die gute Nachricht: Deine drei Werte befinden sich in der Liste, daher kannst du sie für die String-Formatierung verwenden, denke nur an die Anzahl der Slots, die du innerhalb des Format-Strings definieren musst.

Ich weiß, es wird verlockend sein, einfach den bereits implementierten Code zu kopieren und zu bearbeiten, aber wir sind hier, um zu lernen, daher empfehle ich dringend, ihn von Grund auf neu zu schreiben. Es ist ein sehr einfaches Programm, es neu zu machen ist sehr einfach, aber es hilft dir, mehr zu üben.

Setze deinen Code in *code01.py* um.

Jetzt mache dasselbe, aber beginne mit einer leeren Liste und füge zufällige Zahlen hinzu.

Setze deinen Code in *code02.py* um.

## 8.4 For-Schleife

In dem oben genannten Code mussten wir über drei Maulwürfe (Kreise) iterieren, die wir in einer Liste hatten. Python bietet dafür ein spezielles Werkzeug: eine For-Schleife, die über die Elemente jeder Sequenz iteriert (unsere Liste ist eine Sequenz!). Hier ist ein Beispiel:

```
numbers = [2, 4, 42]
for a_number in numbers:
    print("Der Wert der Variable a_number in dieser Iteration beträgt %d"%(a_number))
    a_number = a_number + 3
    print("  Jetzt haben wir es um 3 erhöht: %d"%(a_number))
    print("  Jetzt verwenden wir es in einer Formel a_number / 10: %g"%(a_number / 10))
#> Der Wert der Variable a_number in dieser Iteration beträgt 2
#>   Jetzt haben wir es um 3 erhöht: 5
#>   Jetzt verwenden wir es in einer Formel a_number / 10: 0.5
```

[7] Und wir haben gerade einmal an der Oberfläche gekratzt!

```
#> Der Wert der Variable a_number in dieser Iteration beträgt 4
#>   Jetzt haben wir es um 3 erhöht: 7
#>   Jetzt verwenden wir es in einer Formel a_number / 10: 0.7
#> Der Wert der Variable a_number in dieser Iteration beträgt 42
#>   Jetzt haben wir es um 3 erhöht: 45
#>   Jetzt verwenden wir es in einer Formel a_number / 10: 4.5
```

Hier wird der Code innerhalb der `for`-Schleife dreimal wiederholt, weil es drei Elemente in der Liste gibt. Bei jeder Iteration wird der nächste Wert aus der Liste einer temporären Variable `a_number` zugewiesen (siehe Ausgabe). Sobald der Wert einer Variablen zugewiesen ist, kannst du ihn wie jede andere Variable verwenden. Du kannst ihn ausgeben lassen (erster `print`), du kannst ihn ändern (zweite Zeile innerhalb der Schleife), seinen Wert verwenden, wenn du andere Funktionen aufrufst, usw. Um das besser zu würdigen, kopiere diesen Code in eine temporäre Datei (nenne sie `test01.py`), setze einen Breakpoint auf die erste `print` Anweisung und dann verwende **F10**, um durch die Schleife zu springen und zu sehen, wie sich der Wert der Variable `a_number` bei jeder Iteration ändert und dann in der zweiten Zeile innerhalb der Schleife modifiziert wird.

Beachte, dass du die gleiche break Anweisung verwenden kannst wie bei der while Schleife.

Mache Übung #8.

## 8.5 Verwendung der For-Schleife zur Erzeugung von Slots

Verwende die For-Schleife zweimal. Zuerst, wenn du die drei Slots erstellst: Beginne mit einer leeren Liste und füge drei zufällige Nummern mit einer For-Schleife hinzu. Zweitens, beim Ausdrucken der Slots. Ich habe oben darauf hingewiesen, dass es einfacher ist, die Formatierung vorzunehmen, wenn man drei Werte in einer Liste hat, aber hier sollst du, um der Übung willen, jeden Slot separat mit einer For-Schleife ausdrucken.

Setze deinen Code in *code03.py* um.

## 8.6 range() Funktion: Code N-mal wiederholen

Manchmal musst du den Code mehrmals wiederholen. Stelle dir zum Beispiel vor, dass du in einem Experiment 40 Versuche hast. Daher musst du den versuchsbezogenen Code 40 Mal wiederholen. Natürlich kannst du von Hand eine 40 Elemente lange Liste erstellen und darüber iterieren, aber Python hat dafür eine praktische range() Funktion. `range(N)` erzeugt N Ganzzahlen von 0 bis N-1 (gleiche Regel wie beim Slicing, dass bis zu, aber nicht einschließlich gilt), über die du in einer For-Schleife iterieren kannst.

```
for x in range(3):
    print("Der Wert von x ist %d"%(x))
#> Der Wert von x ist 0
#> Der Wert von x ist 1
#> Der Wert von x ist 2
```

Du kannst das Verhalten der range() Funktion ändern, indem du einen Startwert und eine Schrittgröße angibst. Aber in ihrer einfachsten Form ist `range(N)` ein praktisches Werkzeug, um den Code so oft zu wiederholen. Beachte, dass du zwar immer eine temporäre Variable in einer `for` Schleife benötigst, sie manchmal aber überhaupt nicht verwendest. In solchen Fällen solltest du `_` (Unterstrich) als Variablennamen verwenden, um anzuzeigen, dass sie nicht verwendet wird.

```
for _ in range(2):
    print("Ich werde zweimal wiederholt!")
#> Ich werde zweimal wiederholt!
#> Ich werde zweimal wiederholt!
```

Alternativ kannst du `range()` verwenden, um durch die Indizes einer Liste zu schleifen (denke daran, du kannst immer auf ein einzelnes Listenelement über `var[index]` zugreifen). Mache genau das[8]! Ändere deinen Code so, dass du die range() Funktion in der For-Schleife verwendest (wie kannst du die Anzahl der Iterationen, die du benötigst, aus der Länge der Liste berechnen?), verwende die temporäre Variable als *Index* für die Liste, um jedes Element zu zeichnen[9]. Wenn du unsicher bist, setze einen Breakpoint innerhalb (oder kurz vor) der Schleife und schreite durch deinen Code, um zu verstehen, welche Werte eine temporäre Schleifenvariable erhält und wie sie verwendet wird.

## 8.7 Informationen zum Slot-Index hinzufügen

In der vorherigen Version des Spiels hatten wir keine Möglichkeit, den Slot-Index anzugeben. Es ist zwar offensichtlich an der Reihenfolge im Ausdruck erkennbar, aber es wäre schöner, wenn wir explizit `"Slot #1: 2"` ausdrucken würden. Verwende die Funktion range() um Indizes von Slots zu generieren, schlaufe über diese Indizes und drucke sie in "Slot # : " aus. Beachte jedoch, dass die Indizes in Python bei Null beginnen, unsere Slots aber bei 1 starten (kein Slot Null!). Überlege dir, wie du dies in einem Ausdruck korrigieren könntest.

Gib deinen Code in *code04.py* ein.

[8] Beachte, das ist nicht eine *bessere*, sondern eine *alternative* Möglichkeit, dies zu tun.

[9] Stilhinweis: Wenn eine Variable ein *Index* von etwas ist, neige ich dazu, sie `isomething` zu nennen. Wenn sie zum Beispiel einen Index zu einem aktuellen Maulwurf hält, würde ich sie `imole` nennen. Das ist *meine* Art, das zu tun. Andere verwenden das `i_` Präfix oder ein `_i` Suffix. Aber so oder so, es ist eine nützliche Benennungskonvention. Denke daran, je einfacher es ist, die Bedeutung einer Variable aus ihrem Namen zu verstehen, desto einfacher ist es für dich, den Code zu lesen und zu ändern.

## 8.8 Über Index und Element gleichzeitig schleifen mit Listenaufzählung

Es passiert ziemlich oft, dass du sowohl über die Indizes als auch über die Elemente einer Liste schleifen musst, daher hat Python eine praktische Funktion dafür: enumerate()! Wenn du anstelle einer Liste über enumerate() iterierst, erhältst du ein Tupel mit `(Index, Wert)`. Hier ist ein Beispiel:

```
letters = ['a', 'b', 'c']
for index, letter in enumerate(letters):
    print('%d: %s'%(index, letter))
#> 0: a
#> 1: b
#> 2: c
```

Verwende enumerate, um gleichzeitig über den Index und das Element zu schleifen und einen Slot nach dem anderen auszudrucken. Sieh dir den `start` Parameter der Funktion an, um sicherzustellen, dass dein Index jetzt bei 1 beginnt.

Gib deinen Code in *code05.py* ein.

## 8.9 List Comprehension

Die List Comprehension bietet eine elegante und leicht zu lesende Möglichkeit, Elemente einer Liste zu erzeugen, zu ändern und/oder zu filtern und dabei eine neue Liste zu erstellen. Die allgemeine Struktur lautet

```
new_list = [<Transformation-des-Elements> for item in old_list if <Bedingung-gegeben-da
```

Schauen wir uns Beispiele an, um zu verstehen, wie es funktioniert. Stell dir vor, du hast eine Liste `numbers = [1, 2, 3]` und du musst jede Zahl um 1 erhöhen[10]. Du kannst es machen, indem du eine neue Liste erstellst und 1 zu jedem Element im Teil hinzufügst:

```
numbers = [1, 2, 3]
numbers_plus_1 = [item + 1 for item in numbers]
```

Beachte, dass dies äquivalent ist zu

```
numbers = [1, 2, 3]
numbers_plus_1 = []
for item in numbers:
    numbers_plus_1.append(item + 1)
```

Oder stelle dir vor, du musst jedes Element in einen String umwandeln. Das kannst du einfach so machen

[10] Ein sehr willkürliches Beispiel!

```
numbers = [1, 2, 3]
numbers_as_strings = [str(item) for item in numbers]
```

Wie wäre die äquivalente Form mit einer normalen for-Schleife? Schreibe beide Versionen des Codes in Jupiter cells und überprüfe, ob die Ergebnisse gleich sind.

Mache Übung #9 im Jupyter Notebook.

Implementiere nun den unten stehenden Code mit Hilfe der List Comprehension. Überprüfe, ob die Ergebnisse übereinstimmen.

```
strings = ['1', '2', '3']
numbers = []
for astring in strings:
    numbers.append(int(astring) + 10)
```

Mache Übung #10 im Jupyter Notebook.

Wie oben bemerkt, kannst du auch eine bedingte Anweisung verwenden, um zu filtern, welche Elemente an die neue Liste weitergegeben werden. In unserem Zahlenbeispiel können wir Zahlen beibehalten, die größer als 1 sind

```
numbers = [1, 2, 3]
numbers_greater_than_1 = [item for item in numbers if item > 1]
```

Manchmal wird die gleiche Aussage in drei Zeilen statt in einer geschrieben, um das Lesen zu erleichtern:

```
numbers = [1, 2, 3]
numbers_greater_than_1 = [item
                          for item in numbers
                          if item > 1]
```

Natürlich kannst du die Transformation und Filterung in einer einzigen Aussage kombinieren. Schreibe einen Code, der alle Elemente unter 2 herausfiltert und 4 zu ihnen hinzufügt.

Mache Übung #11 im Jupyter Notebook.

## 8.10 Verwendung von List Comprehension zur Erstellung von drei Slots

Lass uns List Comprehension verwenden, um in einer einzigen Zeile drei Slots zu erstellen. Die gute Nachricht ist, dass wir uns das Erstellen einer leeren Liste und die Verwendung von `append` ersparen. Die schlechte Nachricht ist, dass du darüber nachdenken musst, über welche Werte du schleifen möchtest, wenn du drei Zufallszahlen erzeugen willst. Tipp: schau dir die Funktion range() erneut

an und überlege, ob du die temporäre Schleifenvariable tatsächlich verwenden wirst.

Gib deinen Code in *code06.py* ein.

## 8.11 Fertig für den Tag

Ausgezeichnet! Verpacke den Ordner mit dem Code und dem Notebook in eine Zip-Datei und reiche sie ein!

# Chapter 9

# Jagd auf den Wumpus

Heute werden wir ein Text-Abenteuer-Computerspiel Jagd auf den Wumpus programmieren: “Im Spiel bewegt sich der Spieler durch eine Reihe von miteinander verbundenen Höhlen, die in einem Dodekaeder angeordnet sind, während sie ein Monster namens Wumpus jagen. Das rundenbasierte Spiel hat den Spieler versucht, tödliche bodenlose Gruben und”super Fledermäuse” zu vermeiden, die sie durch das Höhlensystem bewegen; das Ziel ist es, einen ihrer “krummen Pfeile” durch die Höhlen zu schießen, um den Wumpus zu töten…”

Wie zuvor werden wir mit einem sehr einfachen Programm beginnen und es Schritt für Schritt zur endgültigen Version ausbauen. Der Zweck dieses Kapitels ist es, die Fähigkeiten, die du bereits hast, zu festigen, indem du sie in einem komplexeren Spiel mit mehreren Funktionen, Spielobjekten usw. anwendest. Vergiss nicht, die gesamte Datei zu kommentieren (oberer mehrzeiliger Kommentar, um zu erklären, worum es hier geht), strukturiere deinen Code, kommentiere einzelne Abschnitte und verwende sinnvolle Variablen- und Funktionsnamen. Deine Codekomplexität wird ziemlich hoch sein, daher wirst du den guten Stil benötigen, um dir dabei zu helfen.

Hol dir das Übungsnotebook, bevor wir starten.

## 9.1 Kapitel Konzepte

- Verwenden von sets.

## 9.2 Ein Höhlensystem

In unserem Spiel wird der Spieler durch ein System von Höhlen wandern, wobei jede Höhle mit drei anderen Höhlen verbunden ist. Das Layout der Höhle wird

eine *KONSTANTE* sein, daher definieren wir es zu Beginn des Programms wie folgt.

```
CONNECTED_CAVES = [[1, 4, 5], [2, 0, 7], [3, 1, 9], [4, 2, 11],
                   [0, 3, 13], [6, 14, 0], [7, 5, 15], [8, 6, 1],
                   [9, 7, 16], [10, 8, 2], [11, 9, 17], [12, 10, 3],
                   [13, 11, 18], [14, 12, 4], [5, 13, 19], [16, 19, 6],
                   [17, 15, 8], [18, 16, 10], [19, 17, 12], [15, 18, 14]]
```

Lassen uns das entschlüsseln. Du hast eine Liste von zwanzig Elementen (Höhlen). Innerhalb jeden Elements ist eine Liste von verbundenen Höhlen (Höhlen, zu denen du reisen kannst). Das bedeutet, dass du, wenn du in Höhle Nr. 1 (Index `0`) bist, verbunden bist mit `CONNECTED_CAVES[0]` → `[1, 4, 5]` (beachte, dass auch diese Zahlen im Inneren nullbasierte Indizes sind!). Um also zu sehen, welchen Index die zweite mit der ersten verbundene Höhle hat, würdest du `CONNECTED_CAVES[0][1]` schreiben (du erhältst das erste Element der Liste und dann das zweite Element der Liste aus dem Inneren).

Um dem Spieler das Herumwandern zu ermöglichen, müssen wir zunächst wissen, wo er sich befindet. Lass uns eine neue Variable namens `player_location` definieren und einen zufälligen vordefinierten Index zuweisen. Denke über den niedrigsten gültigen Index nach, den du in Python haben kannst (diesen musst du fest verdrahten). Um den höchstmöglichen gültigen Index zu berechnen, musst du die Gesamtzahl der Höhlen kennen, d.h., die Länge - len() - der Liste. Denke daran, dass die Indizes in Python nullbasiert sind, also überlege, wie du den höchsten gültigen Index aus der Länge berechnest! Nutze diese beiden Indizes und setze den Spieler in eine zufällige Höhle. Hierfür kannst du die Funktion randint verwenden. Schau in den vorherigen Kapiteln nach, falls du vergessen hast, wie man sie benutzt.

Unser Spieler muss wissen, wohin er gehen kann, also müssen wir in jeder Runde die Informationen darüber ausgeben, in welcher Höhle sich der Spieler befindet und über die verbindende Höhle (verwende String-Formatierung, um dies schön aussehen zu lassen). Lass uns das unser erster Code-Ausschnitt für das Spiel sein. Der Code sollte so aussehen

```
# Import randint

# Definiere CONNECTED_CAVES (einfach die Definition kopieren und einfügen)

# Erstelle Variable `player_location` und weise ihr einen zufälligen gültigen Index zu
# Spieler in eine zufällige Höhle setzen

# Ausgabe des aktuellen Höhlenindexes und der Indizes der verbundenen Höhlen. Verwende
```

Füge deinen Code in *code01.py* ein.

## 9.3 Herumwandern

Jetzt, da der Spieler "sehen" kann, wo er sich befindet, lass ihn herumwandern! Verwende die Funktion `input()`, um den Index der Höhle abzufragen, in die der Spieler gehen möchte, und "bewege" den Spieler in diese Höhle (welche Variable musst du ändern?). Denke daran, dass `input()` einen String zurückgibt, daher musst du ihn explizit in eine Ganzzahl umwandeln (siehe das Zahlenraten Spiel, wenn du vergessen hast, wie man das macht). Gib vorerst nur gültige Zahlen ein, da wir später Überprüfungen hinzufügen werden. Um das Herumwandern kontinuierlich zu gestalten, setze es in eine While-Schleife, sodass der Spieler herumwandert, bis er zur Höhle Nr. 5 (Index 4!) gelangt. Wir werden später sinnvollere Spiel-Ende-Bedingungen haben, aber dies ermöglicht es dir, das Spiel zu beenden, ohne es von außen zu unterbrechen. Der Code sollte wie folgt aussehen (achte auf deine Einrückungen!).

```
# import randint function

# define CONNECTED_CAVES (simply copy-paste the definition)

# create `player_location` variable and set it to a random valid cave index

# solange der Spieler nicht in der Höhle Nr. 5 ist (Index 4):
    # Ausgabe des aktuellen Standorts und Liste der verbundenen Höhlen. Verwende String-Formatier
    # Eingabeaufforderung, in welche Höhle der Spieler gehen möchte und "verschiebung" des Spiele

# Ausgabe einer schönen Game-Over-Nachricht
```

Füge deinen Code in *code02.py* ein.

## 9.4 Prüfen, ob ein Wert *in* der Liste ist[#in-collection]

Momentan vertrauen wir dem Spieler (naja, dir), den korrekten Index für die Höhle einzugeben. Daher wird das Programm den Spieler zu einer neuen Höhle versetzen, selbst wenn du einen Index einer Höhle eingibst, die nicht mit der aktuellen verbunden ist. Noch schlimmer ist, dass es versuchen wird, den Spieler zu einer undefinierten Höhle zu versetzen, wenn du einen Index größer als 19 eingibst. Um zu prüfen, ob ein eingegebener Index mit einer der verbundenen Höhlen übereinstimmt, musst du die bedingte Anweisung in verwenden. Die Idee ist einfach, wenn der Wert in der Liste ist, ist die Aussage `True`, wenn nicht, ist sie `False`.

```
x = [1, 2, 3]
print(1 in x)
#> True
print(4 in x)
```

```
#> False
```

Beachte, dass du jeweils *einen* Wert/Objekt prüfen kannst. Da eine Liste auch ein einzelnes Objekt ist, prüfst du, ob sie ein Element der anderen Liste ist, und nicht, ob alle oder einige ihrer Elemente darin enthalten sind.

```
x = [1, 2, [3, 4]]
# Das ist Falsch, weil x kein Element [1, 2] hat, sondern nur 1 und 2 (getrennt voneinander)
print([1, 2] in x)
#> False

# Das ist wahr, weil x das Element [3, 4] hat.
print([3, 4] in x)
#> True
```

Mache Übung #1.

## 9.5 Gültigen Höhlenindex prüfen

Jetzt, da du weißt, wie man prüft, ob ein Wert in der Liste ist, lass uns das verwenden, um den Höhlenindex zu validieren. Bevor du den Spieler bewegst, musst du jetzt prüfen, ob der eingegebene Index in der Liste der verbundenen Höhlen enthalten ist. Wenn dies `True` ist, bewegst du den Spieler wie zuvor. Andernfalls gib eine Fehlermeldung aus, z. B. "Falscher Höhlenindex!" ohne den Spieler zu bewegen. Die Schleife stellt sicher, dass der Spieler erneut zur Eingabe aufgefordert wird, sodass wir uns darüber im Moment keine Sorgen machen müssen. Hier musst du eine temporäre Variable erstellen, um die Eingabe des Spielers zu speichern, da du ihre Gültigkeit prüfen musst *bevor* du entscheidest, ob du den Spieler bewegst. Du machst das Letztere nur, wenn der Wert in dieser temporären Variable ein Index einer der verbundenen Höhlen ist!

Ändere deinen Code, um die Überprüfung der Eingabegültigkeit einzuschließen.

Füge deinen Code in *code03.py* ein.

## 9.6 Überprüfung, ob eine Zeichenfolge in eine ganze Zahl umgewandelt werden kann

Es gibt eine weitere Gefahr bei unserer Eingabe: Der Spieler ist nicht garantiert eine gültige Ganzzahl einzugeben! Bisher haben wir uns darauf verlassen, dass du dich benimmst, aber im wirklichen Leben werden die Leute, selbst wenn sie nicht absichtlich versuchen, dein Programm zu stören, gelegentlich die falsche Taste drücken. Daher müssen wir überprüfen, ob die *Zeichenfolge*, die sie eingegeben haben, in eine *Ganzzahl* umgewandelt werden kann.

Die Python-Zeichenkette ist ein Objekt (mehr dazu in ein paar Kapiteln) mit

verschiedenen Methoden, die es ermöglichen, verschiedene Operationen an ihnen durchzuführen. Eine Teilmenge von Methoden ermöglicht es dir, eine grobe Überprüfung ihres Inhalts durchzuführen. Die Methode, die uns interessiert, ist str.isdigit(), die überprüft, ob alle Symbole Ziffern sind und dass die Zeichenfolge nicht leer ist (sie hat mindestens ein Symbol). Du kannst dem obigen Link folgen, um andere Alternativen wie `str.islower()`, `str.isalpha()`, usw. zu überprüfen.

Mache Übung #2.

## 9.7 Überprüfung der gültigen Ganzzahleneingabe

Ändere den Code, der die Eingabe vom Benutzer erhält. Speichere zuerst die rohe Zeichenkette (nicht in eine Ganzzahl umgewandelt!) in einer Zwischenvariable. Wenn diese Zeichenkette dann nur aus Ziffern besteht, konvertiere sie in eine Ganzzahl und überprüfe dann, ob es sich um einen gültigen verbundenen Höhlenindex handelt (Verschieben des Spielers oder Ausgeben einer Fehlermeldung). Wenn die Eingabezeichenkette jedoch nicht nur aus Ziffern besteht, gib nur die Fehlermeldung aus ("Ungültiger Höhlenindex!"). Dies bedeutet, dass du eine if-Anweisung in der if-Anweisung haben musst!

Füge deinen Code in *code04.py* ein.

## 9.8 Den Code in einer Funktion kapseln

Dein Code ist bereits in seiner Komplexität gewachsen, mit zwei Überprüfungen auf einer Eingabefunktion, daher macht es Sinn, diese Komplexität zu verbergen, indem man ihn in einer Funktion kapselt. Nennen wir es `input_cave`, da es nur eine Eingabe für einen gültigen Index einer verbundenen Höhle sein wird. Du hast bereits den gesamten Code, den du brauchst, aber denke darüber nach, welchen Parameter (oder welche Parameter?) es benötigt: Du kannst nicht direkt auf die globale Konstante `CONNECTED_CAVES` oder die globale Variable `player_location` zugreifen!

Lege die Funktion in eine separate `utils.py` Datei, dokumentiere sowohl die Datei als auch die Funktion! Importiere und verwende sie im Hauptskript. Dein Programm sollte *genau* wie zuvor laufen!

Lege `input_cave` in *utils.py* ab. Aktualisiere deinen Code in *code05.py*.

## 9.9 Mengen

Bisher hatten wir nur den Spieler im Auge und wir haben das getan, indem wir seinen Standort in der Variable `player_location` gespeichert haben. Da wir jedoch mehr Spielobjekte hinzufügen werden (bodenlose Gruben, Fledermäuse, den Wumpus), müssen wir den Überblick behalten, wer-wo-ist, damit wir sie

nicht in einer bereits besetzten Höhle platzieren. Wir werden dies als Gelegenheit nutzen, um etwas über sets zu lernen: eine *ungeordnete* Sammlung von *einzigartigen* Elementen. Diese sind eine Implementierung von mathematischen Mengen und haben Eigenschaften, die für unsere notwendige Buchhaltung nützlich sind. Du erstellst eine Menge über die Funktion set() und es kann entweder eine leere Menge sein (zu der du `.add()` hinzufügen kannst) oder es kann eine Liste (oder ein Tupel) in eine Menge umwandeln, aber es wird alle Duplikate entfernen.

```
# Beginnen mit einer leeren Menge
a_set = set()
a_set.add(1)
print(a_set)
#> {1}
a_set.intersection()
#> {1}

# Umwandlung einer Liste in eine Menge
print(set([1, 2, 2, 3]))
#> {1, 2, 3}
```

Wenn du zwei Mengen hast, kannst du verschiedene Operationen durchführen, um die Vereinigung, Schnittmengeoder Differenz zwischen zwei Mengen zu finden (siehe *Grundlegende Operationen* in der Wikipedia.

Mache Übung #3.

Beachte bitte, dass die Menge *nicht geordnet* ist, so dass du nicht auf ihre einzelnen Elemente zugreifen kannst. Allerdings kannst du genauso wie bei Listen überprüfen, ob ein Wert in der Menge ist.

```
a_set = set([1, 2, 3, 3])
3 in a_set
#> True
```

## 9.10 Die Belegung von Höhlen im Auge behalten

Jetzt, da du Mengen kennst, wird es leicht sein, den Überblick über die belegten Höhlen zu behalten. Im globalen Skript benötigen wir eine Variable, die die Indizes aller belegten Höhlen in einer Menge enthält (du startest mit einer leeren Menge, da alle Höhlen anfangs leer sind). Außerdem benötigen wir eine Funktion, die einen gültigen Index für eine freie Höhle generiert. Dies basiert auf 1) dem Bereich der gültigen Indizes (denke darüber nach, wie du diese Informationen am wirtschaftlichsten an die Funktion weitergeben kannst, du kommst mit nur einer Ganzzahl aus) und 2) den Indizes der bereits belegten Höhlen (wir haben das in der globalen Variable, die du definierst, aber du kannst auf diese

Variable *nicht* direkt zugreifen, also *musst* du diese Information als Parameter übergeben!).

Wir werden diese Funktion zweimal schreiben. Zunächst verwenden wir eine Brute-Force-Methode, um eine freie Höhle zu finden: Erzeuge einfach in einer Schleife einen gültigen zufälligen Index, bis er *nicht* in der Menge der belegten Höhlen ist. Da er es nicht ist, handelt es sich um eine gültige freie Höhle, also solltest du die Schleife verlassen und diesen Wert zurückgeben. Nenne diese Funktion `find_vacant_cave_bruteforce`, implementiere und dokumentiere sie in `utils.py`, teste sie, indem du sie in ein Jupiter-Notebook kopierst und mehrmals mit verschiedenen fest codierten Mengen von belegten Höhlen ausführst. Überprüfen, dass die gültige Höhle nie in der Menge der belegten Höhlen ist.

Lege `find_vacant_cave_bruteforce` in *utils.py* ab. Teste es in Übung 4.

Die Funktion, die du geschrieben hast, funktioniert, ist aber sehr ineffizient und kann ziemlich problematisch sein, wenn unsere Menge an belegten Höhlen lang ist, da du viele Versuche benötigst, bevor du zufällig eine freie findest. Wir können das besser machen, indem wir Mengenoperationen verwenden: Wir können range generieren, es in eine Menge umwandeln, belegte Höhlen davon subtrahieren (das ergibt eine Menge von unbesetzten Höhlen) und dann einen Wert auswählen (beachte aber, dass du dafür eine Menge in eine Liste umwandeln musst!). Im letzten Schritt wählen wir zufällig, aber *nur* aus freien Höhlen, so dass wir keine Schleifen und mehrere Versuche benötigen, um es richtig zu machen.

Implementiere die Funktion `find_vacant_cave` und teste sie mit dem gleichen Code wie zuvor. Du solltest immer nur einen gültigen Index für eine freie Höhle erhalten.

Lege `find_vacant_cave` in *utils.py* ab. Teste es in Übung 5.

## 9.11 Platzieren von bodenlosen Gruben

Jetzt, da wir das Gerüst haben, fügen wir bodenlose Gruben hinzu. Die Idee ist einfach, wir platzieren zwei davon in zufälligen *freien* Höhlen. Wenn der Spieler in eine Höhle mit einer bodenlosen Grube stolpert, fällt er hinein und stirbt (Spielende). Wir werden den Spieler jedoch warnen, dass seine gegenwärtige Höhle neben einer bodenlosen Grube ist, ohne ihm zu sagen, in welcher Höhle sie genau ist.

Zuerst fügen wir sie hinzu. Dazu erstellen wir eine neue Konstante `NUMBER_OF_BOTTOMLESS_PITS` (ich schlage vor, dass wir sie auf `2` setzen, aber du kannst mehr oder weniger davon haben) und eine neue Variable (`bottomless_pits`), die eine Menge von Indizes von Höhlen mit bodenlosen Gruben in ihnen enthält. Füge bodenlose Gruben mit einer for-Schleife hinzu: Bei jeder Iteration erhalte einen Index einer leeren Höhle (über die Funktion

`find_empty_cave`, denke über ihre Parameter nach), füge diesen Index sowohl zu 1) `bottomless_pits` als auch zu 2) `occupied_caves` hinzu, so dass du 1) weißt, wo die bodenlosen Gruben sind und 2) weißt, welche Höhlen belegt sind. Hier ist der Code-Umriss für den Initialisierungsteil (kopiere die Hauptschleife noch nicht). Überprüfe, ob die Zahlen sinnvoll sind (die Anzahl der Höhlen entspricht deinen Erwartungen, der Wert liegt im erwarteten Bereich, es gibt keine Duplikate, usw.)

```
# Erstelle die Variable `occupied_caves` und initialisiere sie als leere Menge
# Erstelle die Variable `bottomless_pits` und initialisiere sie als leere Menge
# Benutze eine for-Schleife und die range-Funktion, um die for-Schleife NUMBER_OF_BOTT
#     Generiere einen neuen Standort für die bodenlose Grube über die Funktion find_em
```

Teste den Code zum Erstellen von bodenlosen Gruben in Übung 6.

## 9.12 In eine bodenlose Grube fallen

Jetzt fügen wir eine der Möglichkeiten hinzu, das Spiel zu beenden: Der Spieler fällt in eine bodenlose Grube. Dafür müssen wir nur überprüfen, ob sich der Spieler in jeder Runde gerade in einer Höhle befindet, in der eine bodenlose Grube ist. Wenn das der Fall ist, ist die Höhle des Spielers tatsächlich in der Liste der bodenlosen Gruben, drucke eine traurige Spiel-Ende-Nachricht und `breche` aus der Schleife aus. Darüber hinaus verändern wir die Bedingung der `while` Schleife zu `while True:`, so dass die einzige Möglichkeit, das Spiel zu beenden, darin besteht, in die Grube zu fallen (nicht ganz fair für den Spieler, aber das werden wir später korrigieren).

Füge den Code für das Platzieren von bodenlosen Gruben und das Hineinfallen in sie in das Hauptskript ein. Drucke die Höhlen mit bodenlosen Gruben zu Beginn des Programs aus und wandere in sie hinein, um sicherzustellen, dass dies das Spiel korrekt beendet.

Aktualisiere deinen Code in *code06.py*.

## 9.13 Warnung vor einer bodenlosen Grube

Wir müssen dem Spieler die Chance geben, das Schicksal zu vermeiden, in eine bodenlose Grube zu fallen, indem wir ihn warnen, dass eine (oder zwei oder mehr) in der Nähe sind. Zu diesem Zweck müssen wir zusätzliche Informationen *vor* ihrer Entscheidung, ihren Zug zu machen, ausdrucken. Deine Aufgabe ist es, zu überprüfen, ob eine Höhle sowohl im Set `bottomless_pits` als auch in der aktuellen Liste der verbundenen Höhlen ist. Du kannst eine for-Schleife verwenden, aber die Verwendung von Mengenoperationen ist viel einfacher, du musst nur überprüfen, ob eine Schnittmenge dieser beiden Mengen leer ist (sie hat null Länge). Wenn mindestens eine der verbundenen Höhlen eine bodenlose Grube in sich hat, drucke “Du fühlst einen Hauch!”.

Füge deinen Code in *code07.py* ein.

## 9.14 Platzierung von Fledermäusen und Warnung vor ihnen

Wir brauchen mehr Nervenkitzel! Lassen wir Fledermäusen hinzu. Sie leben in Höhlen, der Spieler kann sie hören, wenn sie in einer verbundenen Höhle sind ("Du hörst Flügelschläge!"), aber wenn der Spieler unabsichtlich die Höhle mit Fledermäusen betritt, tragen sie den Spieler zu einer *zufälligen* Höhle.

Die Platzierung der Fledermäuse ist analog zur Platzierung der bodenlosen Gruben. Du brauchst eine Konstante, die die Anzahl der Fledermauskolonien bestimmt (z.B. `ANZAHL_DER fledermaeuse` und setzte diese auf `2` oder eine andere von dir bevorzugte Zahl), eine Variable, die ein Set mit Indizes von Höhlen mit Fledermäusen hält (z.B. `fledermaeuse`), und du musst zufällige leere Höhlen auswählen und sie genau so in `bats` speichern, wie du es bei den bodenlosen Gruben getan hast. Drucke den Ort der Fledermäuse zu Diagnosezwecken aus.

Die Warnung vor den Fledermäusen folgt auch der gleichen Logik wie bei den bodenlosen Gruben: Wenn eine der verbundenen Höhlen Fledermäuse in sich hat, druckst du `"Du hörst Flügelschläge!"` aus.

Füge deinen Code in *code08.py* ein.

## 9.15 Der Spieler wird von Fledermäusen in eine zufällige Höhle transportiert

Wenn sich der Spieler in einer Höhle mit Fledermäusen befindet, transportieren diese ihn in eine *zufällige* Höhle, *unabhängig* davon, ob die Höhle bewohnt ist oder nicht (also jede Höhle ist eine gültige Höhle). So können die Fledermäuse den Spieler in eine Höhle tragen:

1. mit einer weiteren Fledermaus-Kolonie, und diese wird den Spieler wieder transportieren.
2. mit einer bodenlosen Grube, und der Spieler wird hineinfallen.
3. später in die Höhle mit dem Wumpus (der Spieler überlebt das möglicherweise nicht und du implementierst das jetzt nicht).

Denk darüber nach:

1. *wann* du die Anwesenheit von Fledermäusen überprüfst (vor oder nach der Überprüfung auf eine bodenlose Grube?),
2. überprüfst du einmal (mit Hilfe von `if`) oder ein-oder-mehrere Male (mit Hilfe von `while`)

Teste deinen Code, indem du in eine Höhle mit Fledermäusen gehst (drucke ihren Standort am Anfang aus, damit du weißt, wo du hingehen musst).

Füge deinen Code in *code09.py* ein.

## 9.16 Hinzufügen von Wumpus (und vom ihm gefressen werden).

Bis jetzt hast du einen Spieler hinzugefügt (Einzelspieler, Standort als Integer gespeichert), bodenlose Gruben (mehrzahl, Standorte in einem Set gespeichert) und Fledermäusen (ebenfalls Mehrzahl). Füge Wumpus hinzu!

1. Erstelle eine neue Variable (`wumpus`?) und platziere Wumpus in einer freien Höhle. Drucke den Standort von Wumpus zu Debugging-Zwecken aus.
2. Warne vor Wumpus im selben Code, der vor Gruben und Fledermäusen warnt. Die Logik ist die gleiche, aber die Überprüfung ist einfacher, da du dir nur um einen einzigen Wumpus-Standort Sorgen machen musst. Der kanonische Warnungstext lautet `"Du riechst einen Wumpus!"`.
3. Überprüfe, ob der Spieler in der gleichen Höhle wie Wumpus ist. Falls dies der Fall ist, ist das Spiel vorbei, da der Spieler von einem hungrigen Wumpus gefressen wird. Dies ähnelt dem *Spielende* aufgrund des Sturzes in eine bodenlose Grube. Überlege, ob die Überprüfung vor oder nach der Überprüfung auf Fledermäuse erfolgen sollte.

Teste deinen Code, indem du in eine Höhle mit Wumpus gehst (drucke ihren Standort am Anfang aus, damit du weißt, wohin du gehen musst).

Füge deinen Code in *code10.py* ein.

## 9.17 Dem Spieler eine Chance geben.

Geben wir dem Spieler eine Chance. Wenn er in die Höhle mit dem Wumpus kommt, erschreckt er ihn. Dann läuft Wumpus entweder weg zu einer zufälligen angrenzenden Höhle (neu) oder bleibt stehen und frisst den Spieler. Erstelle zunächst eine neue Konstante, die die Wahrscheinlichkeit definiert, dass Wumpus wegrennt, zum Beispiel `P_WUMPUS_AENGSTLICH`. In den Implementierungen, die ich gefunden habe, beträgt sie typischerweise 0,25, aber verwende einen Wert, den du für angemessen hältst.

Wenn also der Spieler in der Höhle mit dem Wumpus ist, ziehe eine zufällige Zahl zwischen 0 und 1 (verwende dafür die Funktion uniform). Sie ist Teil der random Bibliothek, daher lautet der Aufruf `random.uniform(...)`. Denke daran, dass du entweder die *gesamte* Bibliothek importieren und dann deren Funktion durch Voranstellen des Bibliotheksnamens aufrufen kannst oder du importierst nur eine bestimmte Funktion über `from ... import ...`. Sobald

du die Zahl zwischen 0 und 1 generiert hast, wenn diese Zahl *kleiner* ist als die Wahrscheinlichkeit, dass der Wumpus Angst hat, bewege ihn zu einer zufälligen angrenzenden Höhle (Fledermäuse ignorieren Wumpus und es klammert sich an die Decke der Höhlen, bodenlose Gruben sind also kein Problem für ihn). Eine nützliche Funktion, die du bereits verwendet hast, ist choice(), auch Teil der random Bibliothek. Andernfalls, wenn Wumpus nicht weggeschreckt wurde, wird der Spieler gefressen und das Spiel ist vorbei (das einzige Ergebnis in *code10*).

Füge deinen Code in *code11.py* ein.

## 9.18 Flug eines krummen Pfeils

Unser Spieler ist mit *krummen* Pfeilen bewaffnet, die durch Höhlen fliegen können. Die Regeln für seinen Flug sind folgende:

- Der Spieler entscheidet, in welche Höhle er einen Pfeil schießt und wie weit der Pfeil fliegt (von 1 bis 5 Höhlen).
- Jedes Mal, wenn der Pfeil in die nächste Höhle fliegen muss, wird diese Höhle zufällig aus den angrenzenden Höhlen *ausgewählt*, mit Ausnahme der Höhle, aus der er kam (also der Pfeil kann keine 180° Wende machen und es stehen nur zwei von drei Höhlen zur Auswahl).
- Wenn der Pfeil in eine Höhle mit Wumpus fliegt, ist er besiegt und das Spiel ist gewonnen.
- Wenn der Pfeil in eine Höhle mit dem Spieler fliegt, dann hat er einen unbeabsichtigten Selbstmord begangen und das Spiel ist verloren.
- Wenn der Pfeil seine letzte Höhle erreicht hat (basierend darauf, wie weit der Spieler schießen wollte) und die Höhle leer ist, fällt er auf den Boden.
- Fledermäuse oder bodenlose Gruben haben keinen Einfluss auf den Pfeil.

Die Gesamtzahl der Pfeile, die der Spieler zu Beginn hat, sollte in der Konstante `PFEIL_ANZAHL` definiert werden (z.B. `5`).

Um den Überblick über den Pfeil zu behalten, benötigst du folgende Variablen:

- `pfeil`: aktuelle Position des Pfeils.
- `pfeil_vorherige_höhle`: Index der Höhle, aus der der Pfeil kam, so dass du weißt, wohin er nicht zurückfliegen kann.
- `verbleibende_schussdistanz`: verbleibende zu reisende Distanz.
- `verbleibende_pfeile`: Anzahl der verbleibenden Pfeile (auf `PFEIL_ANZAHL` gesetzt, wenn das Spiel beginnt).

Behalte dieses Gerüst im Hinterkopf und lass uns anfangen, unsere Pfeile zu programmieren.

## 9.19 Zufällige Höhle, aber keine U-Turn

Du musst eine Funktion programmieren (nennen wir sie `next_arrow_cave()`), die eine zufällige Höhle auswählt, aber nicht die vorherige Höhle, in der der Pfeil war. Es sollte zwei Parameter haben:

- `connected_caves`: eine Liste von verbundenen Höhlen.
- `previous_cave`: Höhle, aus der der Pfeil gekommen ist.

Zuerst, debugge den Code in einer separaten Zelle. Nehme an, dass `connected_caves = CONNECTED_CAVES[1]` (also, der Pfeil ist derzeit in Höhle 1) und `previous_cave = 0` (Pfeil kam aus Höhle 0). Schreibe den Code, der eine der verbleibenden Höhlen zufällig auswählt (in diesem Fall entweder `2` oder `7`), du willst wahrscheinlich Mengenoperationen verwenden (wie bei der effizienten Platzierung von Spielobjekten). Sobald der Code funktioniert, mache daraus eine Funktion, die die nächste Höhle für einen Pfeil zurückgibt. Dokumentiere die Funktion. Teste sie mit anderen Kombinationen von verbundenen und vorherigen Höhlen.

Teste deinen Code in Übung #7. Sobald getestet, packe die Funktion in *utils.py*.

## 9.20 Zurückgelegte Entfernung

Jetzt, da du eine Funktion hast, die zur nächsten zufälligen Höhle fliegt, implementiere das Fliegen mit einer for-Schleife. Ein Pfeil sollte durch `shooting_distance` Höhlen fliegen (setze es für den Test auf `5`, maximale Entfernung, von Hand). Die *erste* Höhle ist gegeben (sie wird vom Spieler ausgewählt), setze daher `arrow` auf `1` und `arrow_previous_cave` auf `0` (Spieler ist in Höhle `0` und hat den Pfeil in Höhle `1` geschossen). Drucke aus Debugging-Zwecken den Standort des Pfeils bei jeder Iteration aus. Teste den Code, indem du die `shooting_distance` änderst. Setzen sie insbesondere auf `1`. Der Pfeil sollte bereits in Höhle `1` "herunterfallen".

Verwende für dieses eine abgespeckte Version des Codes mit der Konstante `CONNECTED_CAVES`, der importierten Funktion `next_arrow_cave()` und allen relevanten Konstanten und Variablen für den Pfeil. Gib die Anfangshöhle, die vorherige Höhle und die Entfernung von Hand ein.

Füge deinen Code in *code12.py* ein.

## 9.21 Ein Ziel treffen

Implementiere die Prüfung zum Treffen des Wumpus oder des Spielers in der Schleife. Sollte die Überprüfung vor oder nach dem Fliegen des Pfeils zur nächsten zufälligen Höhle stattfinden? In beiden Fällen schreibe eine passende "Spiel vorbei" Nachricht und brich aus der Schleife aus. Teste den Code, indem du

den Wumpus von Hand in die Höhle platzierst, auf die der Spieler schießt oder die nächste.

Du kannst auf den Code aus dem vorherigen Abschnitt aufbauen, aber füge `player_location` und `wumpus` Variablen hinzu und kodiere sie von Hand für das Debugging. Führe den Code mehrmals aus, um zu überprüfen, dass er funktioniert.

Füge deinen Code in *code13.py* ein.

## 9.22 Bewegen oder schießen?

Wir sind fast fertig, aber bevor wir anfangen können, den Code zusammenzustellen, benötigen wir noch ein paar weitere Dinge. Zum Beispiel konnte der Spieler zuvor nur ziehen, also haben wir einfach nach der nächsten Höhlennummer gefragt. Jetzt hat der Spieler bei jedem Zug die Wahl, einen Pfeil abzuschießen oder sich zu bewegen. Implementiere eine Funktion `input_shoot_or_move()`, die keine Parameter hat und `"s"` für "Schießen" oder `"m"` für "Bewegen" zurückgibt. Frage darin den Spieler nach seiner Wahl, bis er eine von zwei gültigen Optionen auswählt. Konzeptionell ist dies sehr ähnlich zu deiner anderen Eingabefunktion `input_cave()`, die wiederholt eine Eingabe fordert, bis eine gültige gegeben ist. Teste und dokumentiere!

Füge `input_shoot_or_move()` in *utils.py* einTeste es in Übung 8.

## 9.23 Wie weit?

Implementiere die Funktion `input_distance()`, die keine Parameter hat und die gewünschte Schussentfernung zwischen `1` und `5` zurückgibt. Frage erneut wiederholt nach einer *ganzzahligen* Eingabe (denke daran, du weißt, wie man überprüfen kann, ob die Eingabe eine gültige Ganzzahl ist), wie weit der Pfeil reisen soll, bis eine gültige Eingabe gegeben ist. Dies ist sehr ähnlich zu deinen anderen Eingabefunktionen. Teste und dokumentiere.

Füge `input_distance()` in *utils.py* einTeste sie in Übung 9.

## 9.24 input_cave_with_prompt

Erstellen eine neue Version der Funktion `input_cave()`, nenne sie `input_cave_with_prompt` und füge einen `prompt` Parameter hinzu, damit wir nun entweder nach dem Bewegen zu oder dem Schießen auf die Höhle fragen können (daher die Notwendigkeit des prompts anstelle einer fest codierten Nachricht).

Füge `input_cave_with_prompt()` in *utils.py* einTeste sie in Übung 10.

## 9.25 Alles zusammenfügen

In den letzten Abschnitten hast du alle Teile erstellt, die du für das finale Spiel mit einem krummen Pfeil benötigst. Hier ist ein Pseudocode, wie der finale Code aussehen sollte. Schau dir das an, um besser zu verstehen, wie die neuen Teile in den alten Code integriert werden. Bis jetzt solltest du folgende Konstanten haben (du kannst auch andere Werte haben):

- `CONNECTED_CAVES`
- `NUMBER_OF_BATS` = 2
- `NUMBER_OF_BOTTOMLESS_PITS` = 2
- `P_WUMPUS_SCARED` = 0.25
- `ARROWS_NUMBER` = 5

Folgende Funktionen:

- `find_vacant_cave(...)`, gibt einen Integer-Höhlenindex zurück
- `input_cave_with_prompt(prompt, connected_cave)`, gibt einen Integer-Höhlenindex zurück
- `input_shoot_or_move()`, gibt `"s"` für "schießen" und `"m"` für "bewegen" zurück.
- `input_distance()`, gibt eine Ganzzahl zwischen 1 und 5 zurück
- `next_arrow_cave(connected_caves, previous_cave)`, gibt einen Integer-Höhlenindex zurück

Folgende Variablen:

- `player_location` : Höhlenindex
- `bottomless_pit`: Liste von Höhlenindizes
- `bats`: Liste von Höhlenindizes
- `wumpus`: Höhlenindex
- `remaining_arrows`: Ganzzahl der verbleibenden Pfeile

Dienst-/temporäre Variablen:

- `occupied_caves`: Liste der Höhlenindizes
- `gameover_due_to_arrow` : gibt an, ob das Spiel vorbei ist, weil entweder der Wumpus oder der Spieler von einem Pfeil getroffen wurde
- `arrow`: Standort eines Pfeils, der ursprünglich auf die Wahl des Spielers basiert
- `shooting_distance`: Anzahl der Höhlen, die der Pfeil durchfliegen soll.
- andere pfeilbezogene temporäre Variablen

```
importiere die benötigten Bibliotheken

importiere die benötigten Funktionen aus utils

definiere KONSTANTEN

platziere Spieler, bodenlose Gruben, Fledermäuse und Wumpus
```

```
setze die Anzahl der verbleibenden Pfeile auf ARROWS_NUMBER
setze die Variable gameover_due_to_arrow auf False

während WAHR:
    solange der Spieler schießen will und noch Pfeile hat:
        frage nach der Höhle, auf die der Spieler schießen will  (speichere die Antwort in der Va
        frage, wie weit der Pfeil fliegen soll (speichere die Antwort in der Variable `shooting_c

        lass in einer for-Schleife den Pfeil durch die Höhlen fliegen:
            Wenn den Wumpus getroffen -> Glückwunsch Spiel vorbei Nachricht, gameover_due_to_arro
            Wenn den Spieler getroffen -> Oops Spiel vorbei Nachricht, gameover_due_to_arrow = Tr
            bewege den Pfeil zur nächsten zufälligen Höhle (Funktion next_arrow_cave und Variable
        verringere die Anzahl der verbleibenden Pfeile (Variable remaining_arrows)

    überprüfe, ob das Spiel aufgrund des Pfeils vorbei ist, breche aus der Schleife aus, falls da

    frage den Spieler, welche Höhle er betreten möchte und bewege den Spieler

    solange der Spieler in der Höhle mit den Fledermäusen ist:
        bewege den Spieler zu einer zufälligen Höhle

    überprüfe die bodenlosen Gruben (Spieler stirbt, breche aus der Schleife aus)

    wenn der Spieler in der gleichen Höhle wie der Wumpus ist:
        wenn der Wumpus erschrocken ist
            bewege den Wumpus zu einer zufälligen Höhle
        sonst
            der Spieler ist tot, bricht aus der Schleife aus
```

Füge deinen Code in *code14.py* ein.

## 9.26 Abschluss

Gut gemacht, das war wirklich ein Abenteuer, diese Höhlen zu erkunden! Zippe
und reiche ein.

# Chapter 10

# Guess the animal

Today we will program a game in which computer tries to guess an animal that you thought of and learns from its mistake gradually building it vocabulary. Despite its simplisity, it will give us an opportunity to learn about dictionaries, critical differences between mutable and immutable objects, recursion, and file system. As per usual, grab the exercise notebook before we start.

## 10.1 Chapter concepts

- Dictionaries
- Recursion
- Mutable vs. immutable objects
- Saving/reading objects via pickle and JSON

## 10.2 Game structure

The way the game is played is very simple: On each turn computer asks you whether an animal has a certain property or if it is a specific animal. It starts knowing only about one animal, say, "dog". So, it asks you "Is the animal you are thinking of is a dog?". If it is, game is over and you can do it again. However, if it is not a dog then the computer asks "Who is it?", let say you answer "Duck" and then computer also asks you "What does duck do?" and you answer "quack". The important bit here is that computer uses this information the next time you play the game. It starts by asking "does the animal quack?", if yes, it guess "Duck", if not it falls back on the only animal it has left and guess "Dog". If it is not a dog, it asks you again "Who is it?", you say "Cat". "What does cat do?", "meow". Below you can see the decision tree that the computer can use on each round and how it adds the information it learned from it failures.

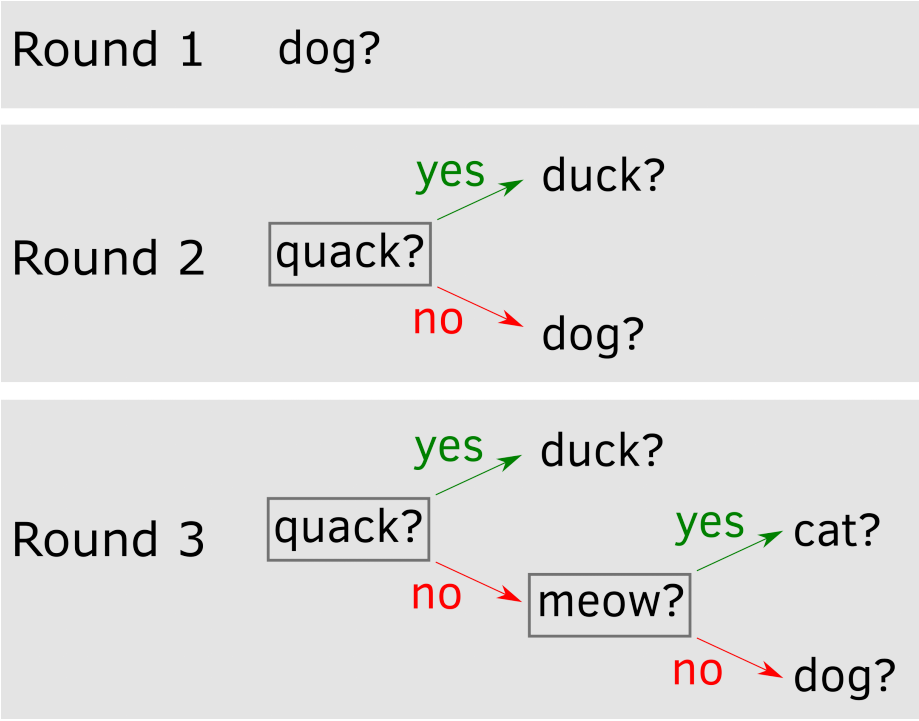



Figure 10.1: Decision tree growing over rounds.

## 10.3 Dictionaries

In the decision tree, we have two kinds nodes: 1) the action decision node ("quack?") that has two edges (*yes* and *no*) that lead to other nodes, and 2) the leaf animal nodes ("duck", "dog", "cat", etc.). Thus each node has a one or two subtrees that in turn include their own decision trees, etc. Thus, each node has following properties:

- `kind`: node *kind* either `"action"` or `"animal"`
- `text`: node *text* which holds either an action or the name of the animal
- `yes` : subtree for answer "yes" (relevant only for *action* nodes)
- `no` : subtree for answer "no" (also relevant only for *action* nodes)

This calls for a container and we *could* put each node with its subtrees into a list and use numerical indexes to access individual elements (e.g., `node[0]` would be the node kind, whereas `node[2]` would hold the yes-subtree) but indexes do not have meaning per se, so figuring out how `node[0]` is different from `node[2]` would be tricky. Python has a solution for cases like this: dictionaries.

A dictionary is a container that stores information using *key : value* pairs. This is similar to how you look up a meaning or a translation (value) of a word (key) in a real dictionary, hence the name. To create a dictionary, you use *curly* brackets `{<key1> : <value1>}, {<key2> : <value2>, ...}` or create it via `dict(<key1>=<value1>, <key2>=<value2>, ...)`. Note that the second version is more restrictive as keys must follow rules for variable names, whereas in curly-brackets version keys can be arbitrary strings.

```
book = {"Author" : "Walter Moers",
        "Title": "Die 13½ Leben des Käpt'n Blaubär"}

# or, equivalently
book = dict(Author="Walter Moers",
            Title="Die 13½ Leben des Käpt'n Blaubär")
```

Once you created a dictionary, you can access or modify each field using its key, e.g. `print(book["Author"])` or `book["Author"] = "Moers, W."`. You can also add new fields by assigning values to them, e.g., `book["Publication year"] = 1999`. In short, you can use a combination of `<dictionary-variable>[<key>]` just like you would use a normal variable. This is similar to using the `list[index]` combination, the only difference is that `index` must be an integer, whereas `key` can be any hashable[1] value.

[1] Immutable values are hashable, whereas mutable ones, like dictionaries and lists, are not. This is because mutable objects can *change* while the program is running and therefore are unusable as a key. I.e., it is hard to match by a key, if the key can be different by the time you need to access the dictionary.

## 10.4 Yes/no input

In our game, we will be asking yes/no question *a lot*, so let us start by programming a `input_yes_no` function that takes a prompt as a single argument and keeps asking for the response until it receives the valid one. It should return `True` if response was "yes" and `False` otherwise. For convenience, it should prepend the prompt with a message `'Type "y" for yes and "n" for no.", e.g., if prompt parameter was`"Is it a pony?" the actual input prompt should be `'Type "y" for yes and "n" for no. Is it a pony?'`. This should be easy for you by now as you implemented several similar functions during the game. Document(!), test it in exercise 7, and put the code in `utils.py`.

Put `input_yes_no` into *utils.py*.Test it in exercise 1.

## 10.5 One-trick pony

Let us start at the beginning by creating a dictionary with a single animal that can be used to ask a question "Is it ?". Create a dictionary following the structure layed out above and think about which field you need (hint, not all four) and what values they should have. In the future, we will modify this tree, so even though you hard code it, it is still a variable, not a constant, so use the appropriate naming style.

Next, you need a simple code that checks if the node `"kind"` is `"animal"`, it asks "Is it ?" (which field you need to use for that?) using the `input_yes_no` function you implemented before. For the moment, congratulate yourself if the answer was "yes" (computer guess it correctly!) but perform no action otherwise.

Put your code into *code01.py*.

## 10.6 Learning a new trick

In the final implementation, our decision tree will grow through trial-and-error but initially, let us hard code a small decision tree by hand. Create a dictionary of dictionaries for the *Round 2* decision tree. It has just three nodes, the top one is an action tree with two subtrees, each subtree is an animal leaf node. Since subtrees a dictionaries, it means that you put a dictionary into a corresponding field, so `decision_tree['yes']` will yield `{"kind" : "animal", "text" : "duck"} and, therefore,`decision_tree['yes']['animal']`will be`"duck". Once you defined it, explore it by hand in a jupiter notebook, trying different field and different levels, as in the example I've shown above.

Now that we have two kinds of nodes, we need to update the query code, so that is asks "Is it ?" for *animal* node (you have this code already) but "Does it ?" for the *action* node. Implement it but take no action for the response to the action node question yet. Test that it works by changing node kind.

Put your code into *code02.py*.

## 10.7 Recursion

Our trees have many nodes at different depths but when we need to act on a node (ask a relevant question), the only thing that matters is the node itself, not the tree it belongs to or where it is within this tree. E.g., look at the figure below for a full and trunkated decision trees. Once we are at the node "meow?", it makes no difference whether we arrived to it from some upper-level node or it was the top node itself, the question we ask and the decision we make is the same. Similarly, once we are at the node `"cat"`, it makes no difference to us whether we ended up where after a long exploration or it was the only node that we have.

This means that we need just one function that will act on the node and that the *same* function will be applied to a relevant subnode for the *action* node. I.e., the function will call itself! This is called a recursion and a classic example[2] to illustrate the concept is computation of a factorial:

$$!n = n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$$

It is easy to see a recursive nature of the computation here as the formula can be rewritten as

$$!n = n \times !(n-1)$$

The only exception is when $n = 1$, so the complete formula is

$$!n = \begin{cases} 1 & \text{if n} = 1 \\ n \times !(n-1) & \text{if n} > 1 \end{cases}$$

Your task is to write a function (document!) that computes a factorial for a given positive integer using the formula above. As long as $n > 1$ it should use itself to compute a factorial of the remaining terms. Test the function to check that computation works correctly.

Implement and test function in exercise 2.

## 10.8 Exploring the decision tree

Let us apply the same idea of a recursion to explore the decision tree while playing the game. For now, hardcode the decision tree shown in the figure below

[2] Actually, it is somewhat misleading, as you do not need recursion for factorial, a for loop will suffice, but this a nice and simple toy example, so we'll walk with the crowd on this one.

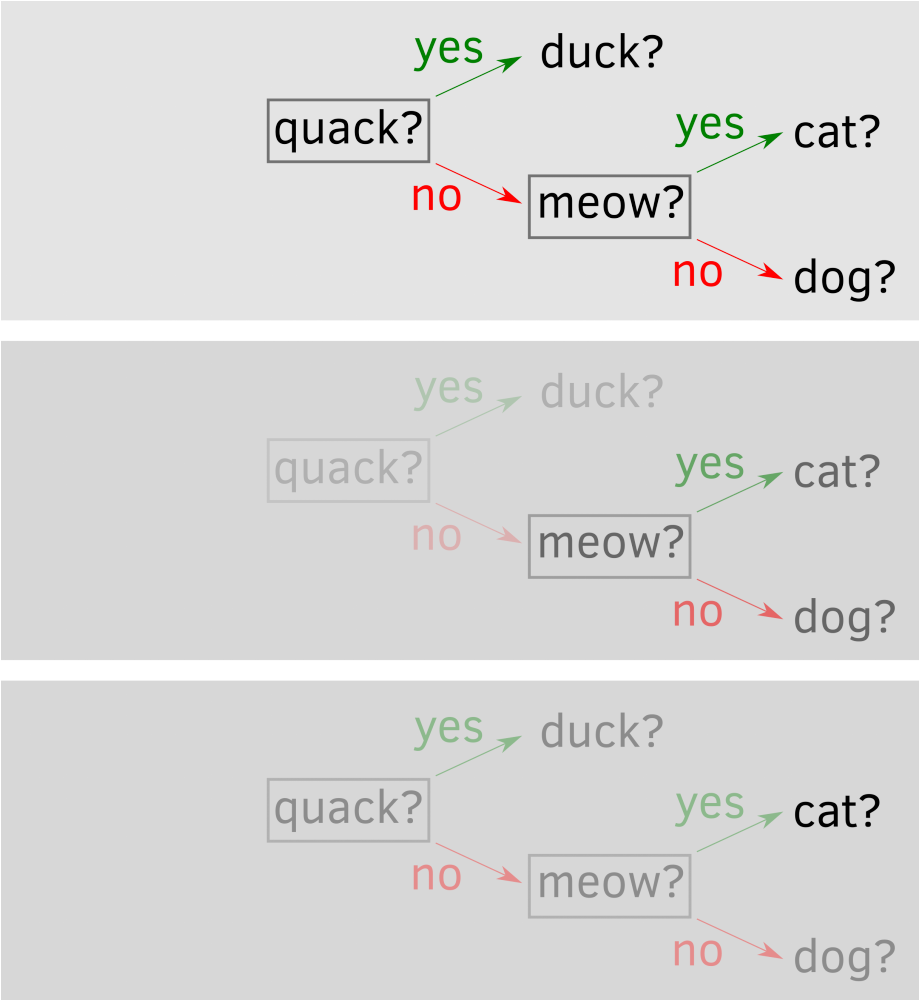



Figure 10.2: Truncated decision tree.

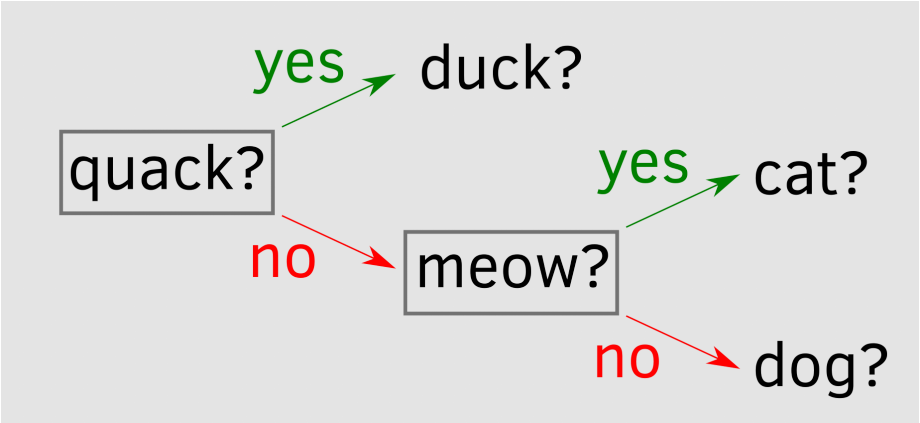



Figure 10.3: Decision tree for three animals.

Next, we need to convert the code from *code02.py* that used a single node to a function. Extend it, so that if the node is an *action* node, the function goes one level deeper by applying itself the corresponding "yes" or "no" subtree (that is determined by player's response). Call it `explore_tree`. It should have just a single parameter (the current node) and it does not need to return any value. Once implemented, call it starting with hardcoded tree and test by giving different answers. The tree is small, so you should be able to quickly try all paths. There should be positive "yay!", if at the end you agree that computer's guess about an animal was correct and no output, if you said "no".

Implement function in *utils.py* Use it in code *code03.py*.

Our next step is to write the code that extends the tree but before that you need to learn about mutable objects and advantages and dangers that they bring.

## 10.9 Variables as boxes (immutable objects)

In this game, you will use dictionaries. These are *mutable*, like lists in contrast to "normal" *immutable* values (integers, floats, strings). You need to learn about this distinction as these two kinds of objects (values) behave very differently under some circumstances, which is both good (power!) and bad (weird unexpected behavior!) news.

You may remember the *variable-as-a-box* metaphor that I used to introduce variables. In short, a variable can be thought of as a "box" with a variable name written on it and a value being stored "inside". When you use this value or assign it to a different variable, you can assume that Python *makes a copy* of it[3] and puts that *copy* into a different variable "box". When you *replace* value of

[3]Not really, but this makes it easier to understand.

a variable, you take out the old value, destroy it (by throwing it into a nearest black hole, I assume), create a new one, and put it into the variable "box". When you *change* a variable based on its current state, the same thing happens. You take out the value, create a new value (by adding to the original one or doing some other operation), destroy the old one, and put the new one back into the variable "box". The important point is that although a *variable* can have different immutable values (we changed `imole` variable on every round), the immutable *value* itself never changes. It gets *replaced* with another immutable value but *never changes*[4].

The box metaphor explains why scopes work the way they do. Each scope has its own set of boxes and whenever you pass information between scopes, e.g., from a global script to a function, a copy of a value (from a variable) is created and put into a new box (e.g., a parameter) inside the function. When a function returns a value, it is copied and put in one of the boxes in the global script (variable you assigned the returned value to), etc.

However, this is true only for *immutable* objects (values) such as numbers, strings, logical values, etc. but also tuples (see below for what these are). As you could have guessed from the name, this means that there are other *mutable* objects and they behave very differently.

## 10.10 Variables as post-it stickers (mutable objects)

Mutable objects are, for example, lists or dictionaries[5], i.e., things that can change. The key difference is that *immutable* objects can be thought as fixed in their size. A number takes up that many bytes to store, same goes for a given string (although a different string would require more or fewer bytes). Still, they do not change, they are created and destroyed when unneeded but never truly updated.

*Mutable* objects can be changed[6]. For example, you can add elements to your list, or remove them, or shuffle them. Same goes for dictionaries. Making such object *immutable* would be computationally inefficient: Every time you add a value a (long) list is destroyed and recreated with just that one additional value. Which is why Python simply *updates* the original object. For further computation efficiency, these objects are not copied when you assign them to a different variable or use as a parameter value but are *passed by reference*. This means that the variable is no longer a "box" you put values into but a "sticker"

[4] A metaphor attempt: You can wear different shirts, so your *look* (variable) changes but each individual shirt (potential values) remains the same (we ignore the wear and tear here) irrespective of whether your are wearing it (value is assigned to a variable) or not.

[5] Coming up shortly!

[6] Building on the looks metaphor: You can change your look by using a different (immutable) shirt or by *changing* your haircut. You hair is mutable, you do not wear a different one on different days to look different, you need to modify it to look different.

you put on an object (a list, a dictionary). And you can put as many stickers on an object as you want *and it still will be the same object*!

What on Earth do I mean? Keeping in mind that a variable is just a sticker (one of many) on a mutable object, try figuring out what will be the output below:

```
x = [1, 2, 3]
y = x
y.append(4)
print(x)
```

Do exercise #3.

Huh? That is precisely what I meant with "stickers on the same object". First, we create a list and put an `x` sticker on it. Then, we assign *the same list* to `y`, in other words, we put a `y` sticker *on the same list*. Since both `x` and `y` are stickers on the *same* object, they are, effectively, synonyms. In that specific situation, once you set `x = y`, it does not matter which variable name you use to change *the* object, they are just two stickers hanging side-by-side on the *same* list. Again, just a reminder, this is *not* what would happen for *immutable* values, like numbers, where things would behave the way you expect them to behave.

This variable-as-a-sticker, a.k.a. "passing value by reference", has very important implications for function calls, as it breaks your scope without ever giving you a warning. Look at the code below and try figuring out what the output will be.

```
def change_it(y):
    y.append(4)

x = [1, 2, 3]
change_it(x)
print(x)
```

Do exercise #4.

How did we manage to modify a *global* variable from inside the function? Didn't we change the *local* parameter of the function? Yep, that is exactly the problem with passing by reference. Your function parameter is yet another sticker on the *same* object, so even though it *looks* like you do not need to worry about global variables (that's why you wrote the function and learned about scopes!), you still do. If you are perplexed by this, you are in a good company. This is one of the most unexpected and confusing bits in Python that routinely catches people[7] by surprise. Let us do a few more exercises, before I show you how to solve the scope problem for mutable objects.

Do exercise #5.

[7] Well, at least me!

## 10.11 Tuple: a frozen list

The wise people who created Python were acutely aware of the problem that the *variable-as-a-sticker* creates. Which is why, they added an **immutable** version of a list, called a tuple. It is a "frozen" list of values, which you can loop over, access its items by index, or figure out how many items it has, but you *cannot modify it*. No appending, removing, replacing values, etc. For you this means that a variable with a frozen list is a box rather than a sticker and that it behaves just like any other "normal" **immutable** object. You can create a `tuple` by using round brackets.

```
i_am_a_tuple = (1, 2, 3)
```

You can loop over it, e.g.,

```
i_am_a_tuple = (1, 2, 3)
for number in i_am_a_tuple:
    print(number)
#> 1
#> 2
#> 3
```

but, as I said, appending will throw a mistake

```
i_am_a_tuple = (1, 2, 3)

# throws AttributeError: 'tuple' object has no attribute 'append'
i_am_a_tuple.append(4)
#> 'tuple' object has no attribute 'append'
```

Same goes for trying to change it

```
i_am_a_tuple = (1, 2, 3)

# throws TypeError: 'tuple' object does not support item assignment
i_am_a_tuple[1] = 1
#> 'tuple' object does not support item assignment
```

This means that when you need to pass a list of values to a function and you want them to have no link to the original variable, you should instead pass *a tuple of values* to the function. The function still has a list of values but the link to the original list object is now broken. You can turn a list into a tuple using `tuple()`. Keeping in mind that `tuple()` creates a frozen copy of the list, what will happen below?

```
x = [1, 2, 3]
y = tuple(x)
x.append(4)
print(y)
```

Do exercise #6.

As you probably figured out, when `y = tuple(x)`, Python creates **a copy** of the list values, freezes them (they are immutable now), and puts them into the "y" box. Hence, whatever you do to the original list, has no effect on the immutable "y".

Conversely, you "unfreeze" a tuple by turning it into a list via `list()`. Please note that it creates **a new list**, which has no relation to any other existing list, even if values are the same or were originally taken from any of them!

Do exercise #7.

Remember I just said that `list()` creates a new list? This means that you can use it to create a copy of a list directly, without an intermediate tuple step. This way you can two *different* lists with *identical* values. You can also achieve the same results by slicing an entire list, e.g. `list(x)`, is the same as `x[:]`.

Do exercise #8.

Here, `y = list(x)` created a new list (which was a carbon copy of the one with the "x" sticker on it) and the "y" sticker was put on that new list, while the "x" remained hanging on the original.

If you feel your head spinning then, unfortunately, I have to tell that it gets even worse. The following paragraph covers fairly advanced scenario but I want you to know about it, as things work extremely counterintuitively and I personally have been caught by this issue a few times and it always took me *forever* to figure out the problem. Thus, I want you to be at least aware of it. What if you have a tuple (immutable!) that contains a list (mutable) inside? As I told you before, you cannot modify the item itself but that item is merely a reference to list (a sticker on a *mutable* object!), so even though tuple is immutable, you can still fiddle with the list itself. Moreover, making a copy of a tuple will merely make a copy of a reference that still points to the same list! So, you could be thinking that since it is all tuples everything is immutatable and well-behaving and be caught out by that[8]. Here is an example of such a mess:

```
tuple_1 = tuple([1, ["A", "B"], 2])
tuple_2 = tuple_1

# This (correctly) does not work
tuple_1[0] = ["C", "D"]
#> 'tuple' object does not support item assignment

# But we can change first element of the list to "C" and second to "D"
# Reference to the list is frozen, but the list itself is mutable!
tuple_1[1][0] = "C"
tuple_2[1][1] = "D"
```

[8] If this makes you want to scream, tell me and will do it together.

```
print(tuple_1)
#> (1, ['C', 'D'], 2)
print(tuple_2)
#> (1, ['C', 'D'], 2)
```

Confusing? You bet! If you feel overwhelmed by this whole immutable/mutable, tuple/list, copy/reference confusion, you are just being a normal human being. I understand the (computational) reasons for doing things this way, I am aware of this difference and how useful this can be but it still catches me by surprise from time to time! So, the word of advice, be careful and double-check your code using debugger whenever you are assigning list or dictionaries, passing them to functions, making copies, having lists inside lists, etc. Be aware that things may not work as you think they should!

## 10.12 Extending the tree

Each node that we have in a tree is a dictionary, which is mutable and that is makes easy if a bit confusing. The good thing is, we can modify tree from inside the function. As I showed you above, you are not passing a dictionary itself to the function, you a passing a *reference* to it, so anything you do to any dictionary within the tree will be applied to the global tree itself. The confusing bit is that we need to keep in mind that we are always with a *reference*, so simple copy won't do us any good. We have two options illustrated in the figure below. The original tree has just one node represented by Dict #1.. When we extend the tree that node must move down and become a leaf of an action node that now becomes the starting point. The can create a new action node dictionary (*Dict #2*), a new animal node for the cat (*Dict #3*) and arrange them as in option #1. However, in this case, we need to make sure that reference in `tree` variable is updated, so that it now points to *Dict #2*. Alternatively, we can keep Dict #1. as the top node but completely replace its content turning it into an action node. The original information will be copied to a new *Dict #3*. Let us try both of these approaches, starting with

## 10.13 Extending the tree via returning new reference

Let us proceed in small steps, as per usual. First, write code that takes an *animal* node (set `tree` variable by hand) and creates a three node tree as in option #1 above. You need to create two additional dictionaries and link them via `"yes"` and `"no"` fields. First, hardcode both new animal and new action. Once your code works, replace hardcoded values with `input` calls asking user "Who is it?" and "What does do?".

Test code in exercise #9.

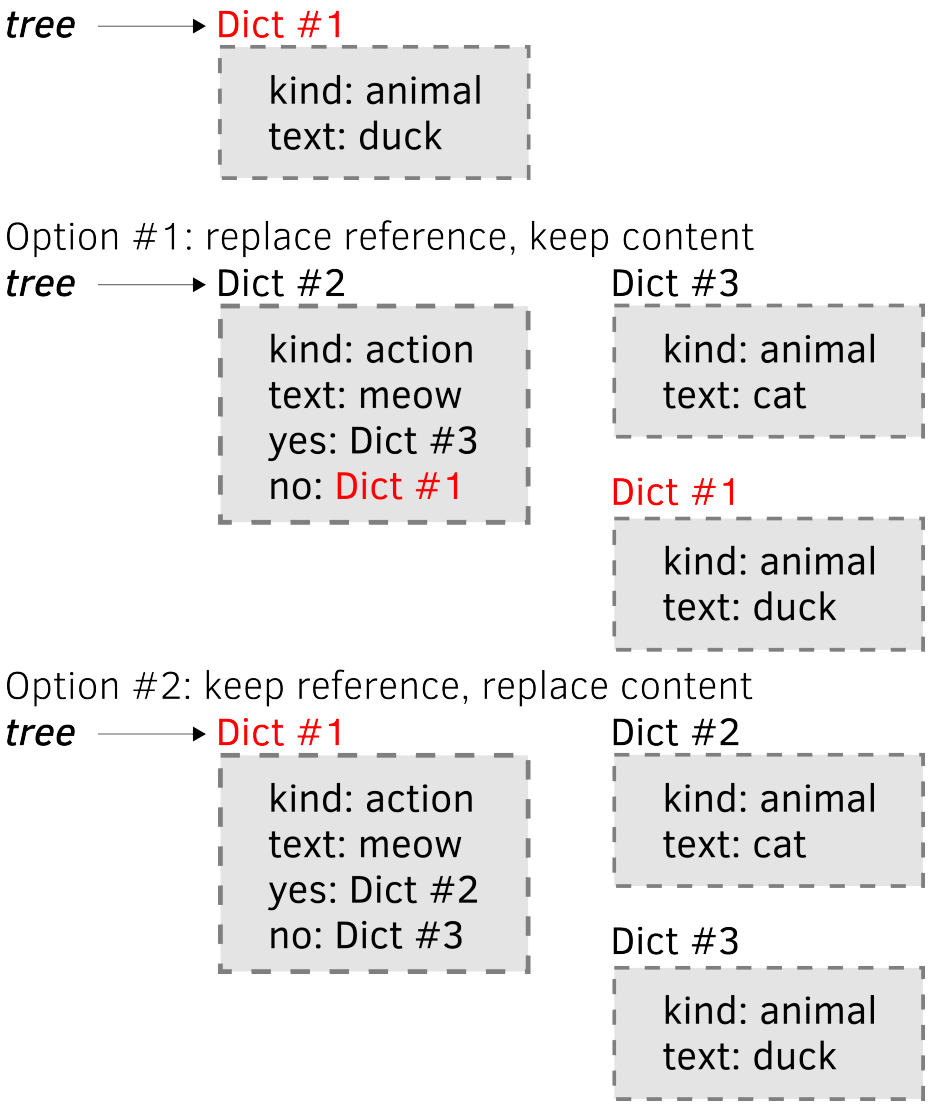



Figure 10.4: Two options for extending tree.

This gives us the code we need for the `explore_and_extend_tree_via_return`[9] function that extends[10] the `explore_tree` function you implemented earlier. Think about where the new code goes to.

A more important modification is that the function must now return a reference to the tree (dictionary that has the tree). It is either the *original* tree or, if you exteded it, the *new* tree. Also, that means for action nodes either `"yes"` or `"no"` dictionaries must be assigned a reference that a recursive call to `explore_and_extend_tree_via_return` returned. E.g., if nothing happened, the same reference to the original dictionary will be assigned. However, if we created a *new* tree, the references to that *new* tree must now be stored in the `"yes"` or `"no"` field. If we do not do this, the field will still point to the original node with our modifications being invisible. Same goes for the top level, as this means not only passing our global `tree` to the function but also assigning the returned reference back to it. Update the function (double check which function you are calling recursively, it should be `explore_and_extend_tree_via_return` not the original `explore_tree`!) and test it by starting with a single animal node tree and calling it in an endless loop (we will use emergency stop via debugger as an exit strategy for now). I would suggest printing out the dictionary after each call (to see how it grows), as well as putting a break point inside or after the function to explore the process, see the calling stack and parameter values for each level.

Implement function in *utils.py* Use it in code *code04.py*.

## 10.14 Extending the tree via modifying the dictionary in-place

The second option (see illustration above) is simpler (we do not need to return anything) but more opaque (we modify things behind the scenes and these modifications are not obvious because there is no return). First, write the code that creates two new animal nodes and replaces the content of the original node with new action (as before, first hardcode the new animal and new action, later repace it with `input` calls as in previous function). You can check that the `tree` references the same object by checking its id. It should remain the same even though the content is different (the `id` for the top node should change in the exercise #9, go back to the code and check).

Important catch, remember, `tree` is a *reference*, so writing `no_animal = tree` to store original information in a new node will do you no good, as it will mean that both `tree` and `no_animal` will refer to the same dictionary. Do `no_animal = tree` and then print out the id for both (the same) and write `tree is no_animal` (`is` checks whether two objects are identical, i.e., same objects,

[9] A mouthful, I know.
[10] Pun intended.

so it will be `True`). Oddly enough, once you write `tree["no"] = no_animal` after that it will reference *itself* (`tree is tree["no"]` will be `True`)!

```
# assignment copies reference but object is the same
dict1 = {"a": 1}
dict2 = dict1
print(id(dict1), id(dict2), dict1 is dict2)
#> 2141494445184 2141494445184 True

# object references itself!
dict1["a"] = dict1
print(id(dict1), id(dict1["a"]), dict1 is dict1["a"])
#> 2141494445184 2141494445184 True
```

There are two ways to solve this problem. You can create a new dictionary assigning *field* values one by one. Since field values are immutable strings, this will create a *different* object with same *content*.

```
dict1 = {"a": 1}
dict2 = {"a" : dict1["a"]}

# same content!
print(dict1 == dict2)
#> True

# different objects
print(id(dict1), id(dict2), dict1 is dict2)
#> 2141494447936 2141494481664 False
```

Alternatively, you can create a either a shallow copy or deep copy of an object using copy library. The former — copy — makes a "shallow" copy by copying the context "as is". In this case, a reference to another object is copied as is and still points to the same object. The deepcopy goes, well, deeper and creates copy for object that the original references. The latter is more computationally expensive (you make copies of *everything*!) but is guaranteed to create a copy with no hidden ties to the original. So, when in doubt, go for the deepcopy. In our case, there is no difference as our original dictionary has just two immutable string, so both copy and deepcopy would do the same.

In our code, use the latter option via copy library.

Test code in exercise #10.

Now that the code is working, you can use it in a new function called `explore_and_modify_tree` which builds upon the `explore_tree` function. Again, use the new code when you are in the *animal* node and the guess was wrong (answer was "no"). Test it the same way in an endless loop, as you did for `explore_and_extend_tree_via_return`.

Implement function in *utils.py* Use it in code *code05.py*.

## 10.15 Can I go home now?

Our program works nicely but the current idea is to play, literally, forever. We should be friendlier than that, so after each round ask the player whether they want to play again and continue only if the answer was "yes" (remember, you already have a function to ask "yes" / "no" questions, use it!).

Update loop in code *code06.py*.

## 10.16 Saving tree for future use via pickle

Our game works, our decision tree grows with every round but the problem is that we start from scratch every time with start the program. This is wasteful and no fun, so we should save our tree at the end of each game and load it again whenever the program starts again. One option is to use pickle library that allows you to dump and load Python objects. Here's how it works (use the unfamiliar `with open("dict1.p", "wb") as pickle_file:` as):

```
import pickle

dict1 = {"a": 1}
print(dict1)
#> {'a': 1}

# dumping dictionary to a file
with open("dict1.p", "wb") as pickle_file:
  pickle.dump(dict1, pickle_file)

# loading a dictionary from a file
with open("dict1.p", "rb") as pickle_file:
  dict2 = pickle.load(pickle_file)
print(dict2)
#> {'a': 1}
```

In our program, we need to load the tree from a file (I called it `animal_tree.p`) at the beginning and dump either at the end (once the player does not want to play anymore) or after each round (this means that the latest version of the tree would be saved *even* if the player exits via emergency interrupt).

Note that we need an *initial* tree, created even before the first run of the program. Create that tree (single animal node) and dump into the file in a separate script or a Jupyter cell.

Implement program in *code07.py*.

## 10.17 Saving tree for future use via JSON

Pickle is a Python-specific serialization format, so you won't be able to use this tree elsewhere (you cannot give to your friend who uses R or C). Plus, it is a *binary* (so, not human readble) and **not a secure** format (see big red warning at the top of official documentation), so you should never trust a pickle file unless it is your own.

An alternative way are JSON files that are widely used in interactive Web (JSON stands for JavaScript Object Notation), are human readable (it is a text file you can open in any editor), and are supported by any other software (any language your friend uses will have a JSON library to work with your file).

Using JSON is very similar to using pickle:

```
import json

dict1 = {"a": 1}
print(dict1)
#> {'a': 1}

# dumping dictionary to a file
with open("dict1.json", "w") as json_file:
  json.dump(dict1, json_file)

# loading a dictionary from a file
with open("dict1.json", "r") as json_file:
  dict2 = json.load(json_file)
print(dict2)
#> {'a': 1}
```

Implement program in *code08.py*.

# Chapter 11

# Brute force Sudoku

In this chapter, we are going to write programs that will generate sudoku puzzle. In this puzzle, you need to complete a $9 \times 9$ grid with number 1 to 9, so that number do not repeat within each row, each column, and each $3 \times 3$ square block. When you play the game, you have an incomplete puzzle and the more gaps you have, the harder the puzzle tends to be. But before you can play the game, some (that is you, today) must 1) generate a complete puzzle, 2) remove some numbers while ensuring that the solution to the puzzle remains unique.

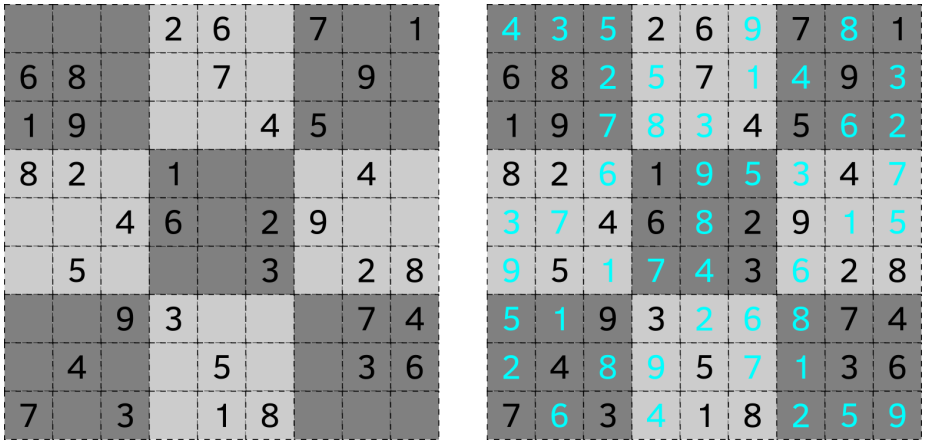



Figure 11.1: An example Sudoku puzzle and the solution

The puzzle is based on a 2D grid and, in principles, you can use list of rows (which are lists themselves) to create this grid. This would be similar to the list of lists that contained information about connecting caves in (Hunt the Wumpus)[#hunt-the-wumpus] game. However, this nested list structure makes it hard to work with columns and blocks, as their elements belong to different

lists, so you need to use for loop instead of simpler list slicing (works, but only for rows).

Instead, we will use it as an opportunity to learn about NumPy library, which is one of the key packages for scientific computing in Python and is a foundation for many data analysis libraries. Note that the material below is by no means complete. If anything, it barely scratches the surface of NumPy. If you need NumPy for your projects, I strongly recommend taking a look at offical Getting Started for aboslute beginners guide and the official user guide.

Grab the exercise notebook to get started.

## 11.1 Importing NumPy

The NumPy is not a core Python library, so you may need to install it. As with all libraries, you must import NumPy before using it in your script. However, this is one of the rare cases where renaming the library during import is a standard and recommended way:

```
import numpy as np
```

## 11.2 1D NumPy arrays versus Python lists

The key data structure that NumPy introduces is a NumPy array that can have any number of dimensions. A one dimensional array, typically called "a vector", is most directly related to a Python list, but with both some limitations and some extra functionality. Unlike Python lists that can hold anything, including other lists, all elements of an array must be of the same numeric type (but character arrays can be created as well). However, the advantage of this restriction is that since all elements are of the same type, it is guaranteed that you can apply the same function to all the elements. Note that there is no such guarantee for potentially heterogenous Python lists, which is why you need to perform operation on each element separately.

You can create a Numpy array from a list via array function:

```
import numpy as np

# A Python list of numbers
a_list = [1, 5, 7]
print(a_list)
#> [1, 5, 7]

# A NumPy array created from the list
an_array = np.array(a_list)
print(an_array)
#> [1 5 7]
```

Note that because of "all values must be of the same type restriction", if the original Python list contained data of various types, all values will be converted to the most flexible one. E.g., a mix of logical values and integers will give you integers, a mix of integers and floats will give you all floats, a mix of anything with strings will give you strings, etc.

```
# logical and integers -> all integers
print(np.array([True, 2, 3, False]))
#> [1 2 3 0]

# integers and floats -> all floats
print(np.array([1.0, 2, 3, 0.0]))
#> [1. 2. 3. 0.]

# logica, integers, floats, and strings -> all strings
print(np.array([False, 1, 2.0, "a"]))
#> ['False' '1' '2.0' 'a']
```

However, note that the type of an array is fixed at the creation type and if you put in a different type value, it will be either converted to that type or, if conversion is tricky, NumPy will throw an error.

```
# array of boolean
array_of_bool = np.array([True, False, True])

# float value is automatically converted to logical
# it is True because only 0.0 is False
array_of_bool[1] = 2.0
print(array_of_bool)
#> [ True  True  True]

# an arbitrary string value that cannot be converted automatically to an integer
array_of_int = np.array([1, 2, 3])
array_of_int[1] = "A text"
#> invalid literal for int() with base 10: 'A text'
```

Do exercise #1.

In general, what you can can do with a list, you can do with a NumPy 1D array. For example, slicing works the same way, you can loop over elements of an array the same way, etc.

```
a_list = [1, 5, 7]
an_array = np.array(a_list)

# slicing
print(a_list[:2])
#> [1, 5]
```

```
print(an_array[1:])
#> [5 7]

# for loop
for value in an_array:
  print(value)
#> 1
#> 5
#> 7
```

However, certain functionality is implemented differently, as the append in the example below. The other functionality, such as a pop, is missing but can be emulated through slicing.

```
# appending values
an_array = np.append(an_array, [4])
print(an_array)
#> [1 5 7 4]
```

The most important practical difference between lists and numpy arrays is that because the latter are homogenous, the operations on them are vectorized. This means that you apply a function to the entire array in one go, making it both easier to use and faster, as most operations on arrays are heavily optimized. Here is an example of multiplying and then adding the same value to all every array element, something that requires a for loop for a normal list

```
a_list = [1, 2, 3]
an_array = np.array(a_list)
2 * an_array + 1
#> array([3, 5, 7])
```

You can also perform elementwise operations on two (or more) arrays at the same time. E.g., here is an example of elementwise addition for two arrays

```
array1 = np.array([1, 2, 4])
array2 = np.array([-1, -3, 5])
array1 + array2
#> array([ 0, -1,  9])
```

Note that it works only if array shapes are the same. In case of the 1D arrays (a.k.a., vectors), it means that their length must be the same.

```
array1 = np.array([1, 2, 4])
array2 = np.array([-1, -3, 5, 7])
array1 + array2
#> operands could not be broadcast together with shapes (3,)
#> (4,)
```

At the same time, you can always use vectors with a single element, which are called "scalars" and this single value is used for every element in the other vector.

```
a_vector = np.array([1, 2, -4])
a_scalar = np.array([-1])
a_vector * a_scalar
#> array([-1, -2,  4])
```

Do exercise #2.

Vectorization also means that you can apply aggregating function – mean, median, min, etc. — to the array instead of computing it by hand.

Do exercise #3.

## 11.3 2D NumPy arrays, a.k.a., matrices

The real power of NumPy is unleashed once your array have two or more dimensions. The 2D arrays are called matrices, whereas arrays with three or more dimensions are known as tensors. The former play key role in classic linear algebra in Python, whereas the latter are required for artifical neural networks (hence, tensor, in TensorFlow).

As with vectors (1D arrays) versus Python lists, the advantage comes from restrictions. Matrices are rectangular, i.e., matrices are composed of multiple rows but each row has the same number of elements. In contrast, you *can* create a rectangular matrix as list of lists (again, our `CONNECTED_CAVES` was $20 \times 3$ rectangular matrix) but this is not guaranteed. On top of that, homogenity of the matrix (all values must of the same type) means you can extract an rectangular part of the matrix and it is guaranteed to be another matrix of the same type. And slicing makes working with 2D arrays much easier. E.g., here is the code to extract a column from a list of lists versus NumPy matrix.

```
matrix_as_list = [[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]]
matrix_as_array = np.array(matrix_as_list)

icolumn = 1 # column index that we want to extract

# extracting column from via for loop
column_as_list = []
for row in matrix_as_list:
  column_as_list.append(row[icolumn])
print(column_as_list)
#> [2, 5, 8]
```

```
# extracting column from a matrix
print(matrix_as_array[:, 1])
#> [2 5 8]
```

Extracting rows, columns, and square block out of a matrix will be key to writing the code for Sudoku, so lets practice!

Do exercise #4.

## 11.4 Creating arrays of a certain shape

There are different ways to create NumPy arrays. Above, we used lists or lists of lists to create them. However, you need create an array of a certain shape filled with zeros or ones. The key parameter in these functions is the *shape* of the array: a list with its dimensions. For a 2D array this means ‘(, ).

```
zeros_matrix_3_by_2 = np.zeros((3, 2))
print(zeros_matrix_3_by_2)
#> [[0. 0.]
#>  [0. 0.]
#>  [0. 0.]]
```

Martix filled with 7. Do exercise #5.

## 11.5 Creating random arrays of a certain shape

NumPy has a module for generating random number — numpy.random — that is conceptually similar to the random but allows you to generate arrays rather than single values. For convenience, the names are kept the same. E.g., function random.randint that generates a single random integer has its twin brother numpy.random.randint that accepts same parameters and, by default also generates a single value. However, you can generate the whole vector / matrix / tensor of random numbers in one go by spezifying its `size`. Confusingly, the parameter is called `size` even though it refers to the “output shape”.

Generate random 4x5 matrix with normally distributed numbers with mean of 3 and standard deviation of 1. Do exercise #6.

## 11.6 Creating arrays with sequences

Similar to creating a range of integer values via range, you can create a vector of integer values via arange[1], although this is equivalent to `np.array(range(...))`:

[1] The name is confusing, but apparently this is a short for “array range”

```
print(np.arange(5))
#> [0 1 2 3 4]
print(np.array(range(5)))
#> [0 1 2 3 4]
```

However, NumPy also has a handy function called linspace that allows you generate sequence of *float* numbers.

Generate random 4x5 matrix with normally distributed numbers with mean of 3 and standard deviation of 1. Do exercise #7.

## 11.7 Stacking array to create a matrix

The arange function will be useful for us to generate a sequence of integers from 1 to 9. However, note that in both cases you can only generate a vector but not a matrix that we need! The solution in this case is to stack individual vectors. When stacking, shapes of individual vectors starts to play a very important role. Unlike lists, that only have length, all arrays including vectors have also shape: information about each dimension.

Things are easy for truly one dimensional arrays. These are arrays created from lists or via functions such as zeros or linspace. If you look at their shape, you will see just one dimension.

```
print(np.array([10, 20, 30]).shape)
#> (3,)
print(np.zeros(5).shape)
#> (5,)
```

These vectors do not have "orientation" (if you know linear algebra you would expect either column or rows vectors), so when you combine these arrays into a matrix you can use them as rows (stacking along `axis=0`, the default, see also vstack) or as columns (stacking along `axis=1`, see also hstack).

```
one_d_vector = np.arange(5)

# Stacking vertically: vectors are used as rows
print(np.stack([one_d_vector, one_d_vector]))
#> [[0 1 2 3 4]
#>  [0 1 2 3 4]]
print(np.stack([one_d_vector, one_d_vector]).shape)
#> (2, 5)

# Stacking horizontally: vectors are used as columns
print(np.stack([one_d_vector, one_d_vector], axis=1))
#> [[0 0]
#>  [1 1]
```

```
#>  [2 2]
#>  [3 3]
#>  [4 4]]
print(np.stack([one_d_vector, one_d_vector], axis=1).shape)
#> (5, 2)
```

Generate single Sudoku ordered row and stack it horizontally and vertically. Do exercise #8.

## 11.8 Repeating and tiling

In the exercise above, you stacked nine arrays but they were all identical. NumPy provides a solution for such repeated values in repeat and tile functions. They both repeat values requested number of times but there are couple of important differences. First, repeat repeats each value N times before going to the next value, whereas tile repeats N times the entire sequence in order.

```
print(np.repeat(np.arange(5), 2))
#> [0 0 1 1 2 2 3 3 4 4]
print(np.tile(np.arange(5), 2))
#> [0 1 2 3 4 0 1 2 3 4]
```

Second, you can decide on the direction the values are repeated similar to how it is determined during stacking. However, the specifics are different for the two functions. For repeat you specify `axis` as in stacking. For tile you specify repetitions (`reps` parameter) and you can specify repetitions *per axis*.

```
# Stack repeated array by row
np.repeat(one_d_vector, 2, axis=0)
#> array([0, 0, 1, 1, 2, 2, 3, 3, 4, 4])

# Tile array twice by row, thrice by column
np.tile(one_d_vector, (2, 3))
#> array([[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4],
#>        [0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]])
```

Generate single Sudoku via repeat and tile. Do exercise #9.

```
a_matrix = np.tile(one_d_vector, (2, 3))
a_matrix
#> array([[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4],
#>        [0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]])
```

## 11.9 Geneating Sudoku via brute force: order and chaos

Our first take on generating Sudoku will be conceptually simple but very inefficient: We will create an ordered matrix (by row or by column), shuffle it (by row or by column), and then check if, accidentally, we ended up with a valid Sudoku. If not, rinse and repeat until we have one.

You already know how to create an ordered matrix that has 1..9 rows or columns. To randomize it, use np.random.shuffle that shuffles an array or a *part of* an array *in place*. The latter part means that the function does not return any value and the array you supplied (by reference, remember mutable objects and functions?).

```
# shuffle the entire array
an_array = np.arange(5)
np.random.shuffle(an_array)
print(an_array)
#> [3 1 0 4 2]

# shuffle part of an array
an_array = np.arange(5)
np.random.shuffle(an_array[:3])
print(an_array)
#> [1 2 0 3 4]
```

Write the code that creates an ordered matrix by row (so, each row goes from 1 till 9 but each column has a single number in it) and write the code that shuffles just one row (put its index in `irow` variable). Change `irow` value to test that your code works.

Write code for creating matrix and shuffling one row in exercise #10.

Once a single row code works, modify that it shuffle all rows but one at a time (you can shuffle the entire matrix but then even rows will most likely be invalid). Once the shuffling code works, turn it into a function `shuffle_by_row`. Just like np.random.shuffle it does not need to return anything, as the shuffling occurs in place. Write the code that generates an ordered-by-row matrix, shuffle it by row and prints it out.

Put `shuffle_by_row` into *utils.py* Write program in *code01.py*

We are here to learn, so to solidify your skills program the same routine by both stacking and shuffling the matrix *by column* (create a separate function `shuffle_by_column` for this).

Put `shuffle_by_column` into *utils.py* Write program in *code02.py*

## 11.10 Is this row even valid?

For a matrix to be a Sudoku, it must adhere to "nine unique numbers in each row, column, and block" rule. Let us implement code for checking the rows (turning it into a column-check will be trivial).

Again, for the row to be valid, it must contain 9 different, i.e., unique numbers. Use unique function to write a comparison for row `irow` (set it to some valid index by hand) that it indeed has 9 unique numbers, i.e., array size is equal to nine. To check that it works correctly, generate an ordered matrix by row (each row goes from 1:9, so any row should be valid) and by column (each row has the same value nine times, so every row should be invalid). Next, shuffle either a single row and check (permutation does not alter the elements, so row should still be valid) or a single column (all or most rows should be invalid).

Write and test single row validation in exercise #11.

Once, we have the code working for a single row, we use (list comprehension)[#list-comprehension] to generate nine logical values (one per row). Matrix is valid (at least with respect to rows) only if *all* rows are valid. Handily, NumPy has a function (all)[https://numpy.org/doc/stable/reference/generated/numpy.all.html] that test whether *all* elements of the array (that we can create from a list that we generated via list comprehension) are `True`. Package this code into function *validate_rows*, think about its inputs and outputs, document.

Put `validate_rows` into *utils.py* Extend program from in *code01.py* in *code03.py*

Now, do the same for columns and use both checks to see if the matrix is a valid Sudoku ::: {.program} Put `validate_columns` into *utils.py* Extend program from in *code03.py* in *code04.py* :::

## 11.11 Blocks

Blocks are a touch trickier to work with as we do not have simple axis to use. However, we can think about each as having a row and column index, which both got from 0 to 2 as we have $3 \times 3$ blocks.

Write the code that uses slicing to extract a single block defined by `i_block_row` and `i_block_col`. For me it helps to compute matrix row for the block top left corner and then get a vertical slice relative to it. Same for the horizontal slice relative to the top-left block column in the matrix. To make debugging easier, generate a $9 \times 9$ matrix of integers stacked either by row or by columns, so that it is easier to see whether you code works. Test it for all combinations for `i_block_row` and `i_block_col`.

Write and test single block extraction code in exercise #12.

Once you have the code to extract block values, the validation logic is very similar to that of the row / column validation. Note that you can use unique

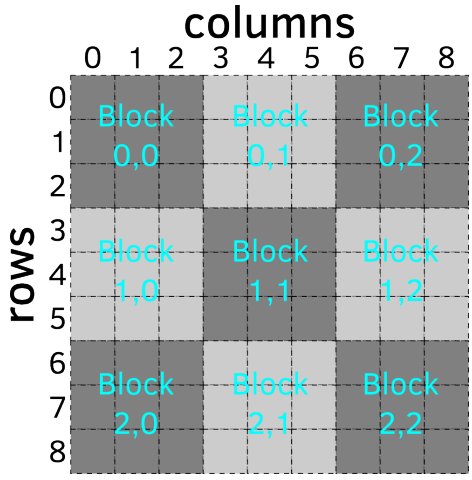



Figure 11.2: Sudoku blocks indexing

directly on the 2D array you extracted or, optionally, you can flatten it to a 1D array first. The main difference is that you need nested loops (one for block row, one for block column and build the list of validation values via append. Once you have a list of nine logical values (one per block), you can check whether they are (all)[https://numpy.org/doc/stable/reference/generated/numpy.all.html] `True`.

Put `validate_blocks` into *utils.py* Extend program from in *code04.py* in *code05.py*

## 11.12 Brute force in action

We have all the building blocks to try to generate Sudoku via brute force. Package three validation functions that you created into a single `validate_matrix` function that returns `True` only if all rows, all columns, and blocks are valid. Then, generate an ordered matrix by row and in a for loop (I would use a for loop and limit it to say 1000 iterations) shuffle this matrix by row and check whether it is valid. If it is, break our of the loop and print out the Sudoku. Write a sad message (without a matrix print out) once you run out of trials (but no sad message otherwise!). Alternatively, you can generated matrix ordered by columns and then shuffle by columns as well (why generating by row and shuffling by columns won't work?)

Put `validate_matrix` into *utils.py* Write program in *code06.py*

## 11.13 Wrap up

Excellent, we have an (occasionally) working program that generates a complete Sudoku matrix. Zip and submit and next time we will write a different, more efficient implementation of the algorithm.

# Chapter 12

# Sudoku backtracking algorithm

In this chapter we will work on Sudoku again using a more efficient backtracking algorithm. In contrast to fully stochastic brute force algorithm that tries to "guess" the entire board in one go, backtracking algorithm makes an educated guess about one cell at at time. As in the previous chapters, we will build the program step by step, but the good news is you know very much everything you need to implement it: matrices, slicing, and recursive functions. We still will used it as opportunity to learn more about lists and NumPy. Grab the exercise notebook and read on.

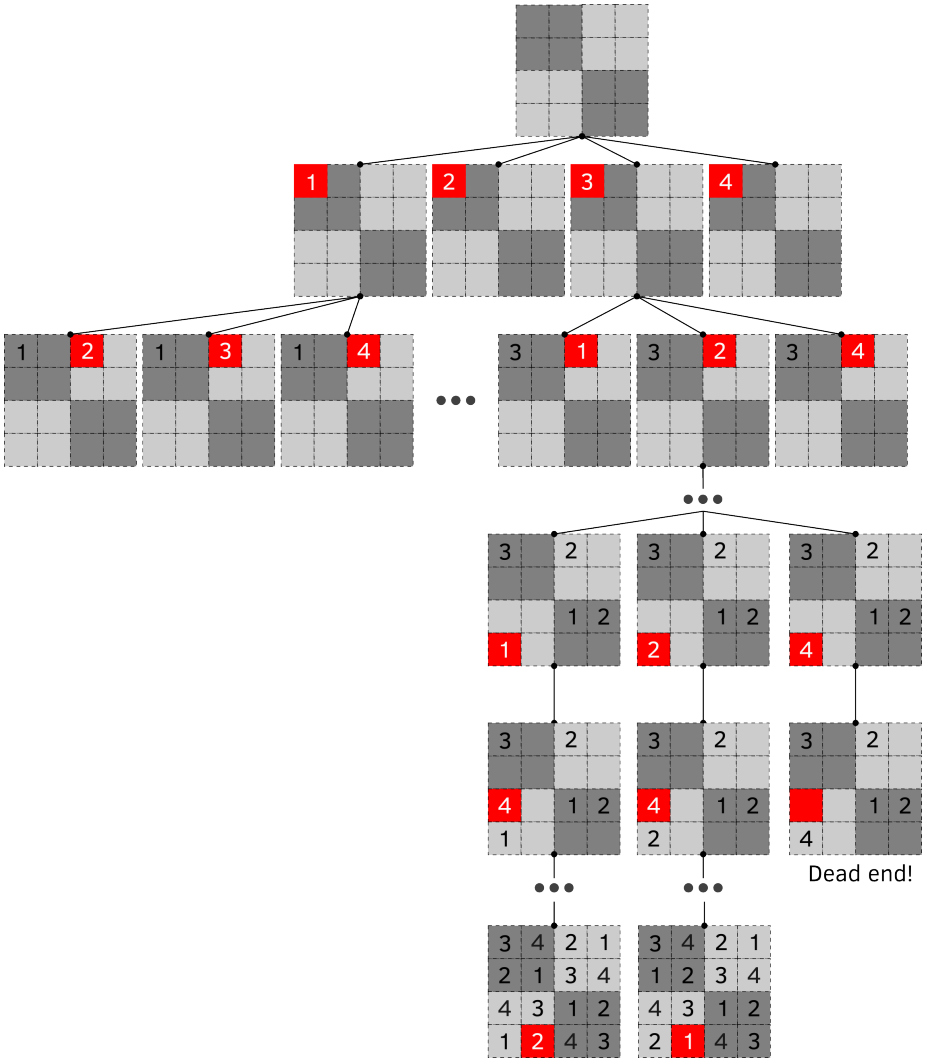
## 12.1 Chapter concepts

- Repetition of lists
- Unpacking of tuples and lists
- Seeding random generator
- Reshaping arrays
- Exceptions

## 12.2 Backtracking algorithm

This algorithm is very intuitive if you think about it as an exploration of a tree. Each node is a cell and our three is 82 levels deep (starting with an empty grid and going through all 81 cells). Each node has *up to* nine possible edges going out of it. Why *up to* nine? We can use numbers 1 to 9 (hence nine edges at most) but unless cells is located in completely empty row, column, and block, some values will be already spoken for which limits valid edges.

An example of such tree for $4 \times 4$ Sudoku is illustrated in the figure below. You start with an empty at the top and you have four possible ways to fill in the first cell (marked in red). However, as you move to a different cell (go one layer down), your choices are limited as both cells belong to the same row. Thus, if you picked 1, you have three choices but 1, if you picked 3 you have three choices but 3. Once you pick on of the available options you move to a next cell. However, you may eventually hit a dead end, as shown at the bottom right: All four numbers were already used, so there is no valid value for the cell and that path through the tree is invalid. The solution is to backtrack: go one level up, pick another number for the cell instead of 4. Here, both 1 and 2 and *in both cases* you may end up with a complete board. This is important as the tree defines *all possible paths* and, therefore, *all possible* $4 \times 4$ valid Sudoku matrices!



If you are patient enough to systematically explore all paths in this tree, you will find all valid Sudoku matrices. However, we do not *all*, we want just *one*. At the same time, we want our program to generate a *different* matrix on each run. If we fix our exploration path (always going through cells in the same order) and our choices (we always go through available numbers in ascending order), we will always end up at the same matrix. So our exploration strategy will include randomness. Initially, we will work with a smaller $4 \times 4$ matrices but once the algorithm works, it will be easy to tweak it to generate the proper $9 \times 9$ ones.

## 12.3 Empty matrix (full of zeros)

The very first matrix in our tree is empty. For simplicity, we denote empty via 0, so you need to generate a $4 \times 4$ matrix filled with zeros. Write the code for this using NumPy: There is a function that you learned about the last time which does exactly the job. Important detail, define size of the matrix (4) as a CONSTANT (let's call it `SUDOKU_SIZE`). This number defines both the matrix size and the range of numbers, so this will be the only change required to generate Sudoku of a different size.

Do exercise #1.

## 12.4 Empty matrix but via list repetitions

The NumPy solution does the job but there is also an alternative way of creating it via repetitions of lists. This does not make things easier here but it is a nice trick to know when for working with list. Namely, you can repeat a list $N$ times via `<list> * N` where `* N` means "repeat N times". Here is an example of repeating three element array four times.

```
[1, 2, 3] * 4
#> [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Given that you can repeat list of lists, generate a $4 \times 4$ list of lists containing all zeros (this can be converted to a $4 \times 4$ matrix via np.array). You need just one line for this, you are not allowed to repeat 0 by hand (so just *one* 0 in your code!) and use `SUDOKU_SIZE` constant.

```
#> [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Do exercise #2.

An important note on use of `*` for lists versus NumPy arrays. In the latter case, `*` means "multiply by", so even if things *look* similar they will behave very differently!

```
a_list = [1, 2, 3]
an_array = np.array(a_list)
```