Joomla 4 Extensions

The Road to

Astrid Günther

1. Juli 2020

Zusammenfassung

Text des Abstracts hier.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Vorwort

In — diesem[1] Buch lernst(https://github.com/dabeaz-course/practical-python 2020) du die Donaudaumpfschiffahrtsgesellschaftskapitän. Grundlagen zur Erstellung einer Joomla 4 Erweiterung. Du erstellst eine beispiel

Anwendung ohne komplizierte Werkzeuge. Ich erkläre dir alles Notwendige — von der Projekteinrichtung bis zur Veröffentlichung der Anwendung auf einem Webserver. Das Buch enthält Hinweise zu weiterführendem Lesematerial und Übungen am Ende jedes Kapitels. Nachdem du das Buch gelesen hast, hast du die Grundlagen, um deine eigene Erweiterung für Joomla 4 zu erstellen. Und, was heutzutage nicht unwichtig ist: Das Lernmaterial halte ich auf dem neuesten Stand. Querverweis interner link hierhin

Mit diesem Buch biete[2] ich dir eine Basis, bevor du in die vielen Möglichkeiten, die cryptographycryptographycryptographycryptographycryptographycryptographycryptographycryptograp die Community und das Ökosystem bereitstellen, eintauchst. Meine Erklärungen beinhalten nur wenige spezielle Werkzeuge, dafür aber viele Informationen über React selbst. Ich erkläre allgemeine Konzepte, Muster und Best Practice anhand einer realen Anwendung.

Wenn
du die URL ansiehst, während du eine Komponente im Administrationsbereich verwendest, bemerkst du gegebenenfalls die Ansichts- und Layoutvariablen. Beispiel: `index.php?option=com_foos &view=foos&layout=default` weist uns an, die foos-Ansicht mit dem Standardlayout zu laden, sodass
`compon ents/com_foos/tmpl/foos/default.php` aufgerufen wird, wenn du dich im Front-End und `administrator/components/com_foos/tmpl/foos/default.php`, wenn du dich im Backend befindest.

Im Wesentlichen lernst du, eine eigene React-Anwendung von Grund auf neu zu erstellen, mit Funktionen wie Paginierung, clientseitiger und serverseitiger Suche und erweiterten Interaktionen wie Sortieren. Ich hoffe, dass meine Begeisterung für React und JavaScript dich ansteckt und dir so den Einstieg erleichtert.

1.1 Über Astrid Günther

Dich interessiert wer ich bin? Darüber freue ich mich! Ich weiß nicht, welche Informationen du erwartest. Ich fange einmal an und hoffe, dass ich das Passende von mir preisgebe.

Seit 2017 arbeite ich selbständig. Vorher war ich 30 Jahre wohlbehütet im öffentlichen Dienst beschäftigt. Die letzten 20 in einer Sparkasse. Das war — mit Abstand betrach-

[1] longnote1
[2] longnote2

tet — nie das Richtige für mich. Hat mir aber, in einer Zeit, in der meine Tochter aufgewachsen ist, das Leben vereinfacht. Alles hat sein Gutes!

Ich fange vorne an: Ich habe 1969 das Licht der Welt erblickt und hatte eine unspektakuläre Kindheit. Nach dem Realschulabschluss habe ich von 1986 bis 1992 im mittleren Justizdienst gearbeitet. Während dieser Zeit habe ich nebenberuflich das Telekolleg II besucht und mit der Fachhochschulreife abgeschlossen. Von 1992 bis 1997 arbeitete ich im gehobenen Postdienst. Hier kam ich bei meiner Arbeit im IT – Betriebs- und Servicezentrum der Postdirektion Köln zum ersten Mal mit Computern in Berührung. 1997 wechselte ich in die EDV-Abteilung einer Sparkasse. Hier war ich bis 2017 angestellt — unterbrochen durch Erziehungsurlaub — erst als Systembetreuerin im Second Level Support und später als Programmiererin. Programmiert habe ich als Einzelkämpferin überwiegend in Java und PHP.

Um im IT-Bereich einen Berufsabschluss zu erlangen, habe ich von 1997 bis 2000 abends die berufsbildende Schule Wirtschaft besucht. Am Ende hatte ich die Erlaubnis, mich **staatlich geprüfte Betriebswirtin für Informationsverarbeitung** zu nennen. Im Anschluss nahm ich 2000 bei der FernUniversität Hagen ein Informatik-Studium in Angriff, welches ich im März 2006 mit dem Abschluss **Master of Computer Science** erfolgreich beendete. Im Studium habe ich hauptsächlich die objektorientierte Programmierung mit *Java* gelernt.

2007 habe ich die erste Website für eine Bekannte erstellt. Diese Arbeit forderte mich im Positiven. Ich habe daraufhin bei der Studiengemeinschaft Darmstadt den Kurs **Grafik-Designer/in PC (SGD)** belegt und mit einer Prüfung beendet. Zunächst programmierte ich alles in PHP, später nutze ich Content Management System. *Wordpress* war kurz im Einsatz. Hängen geblieben sind wir bei *Joomla!*. Zur Zeit lege ich meinen Schwerpunkt auf statische System. Hier kommen *Gatsby* und *React* ins Spiel.

Außerdem keimte in mir die Lust, mein Wissen selbst weiterzugeben. Ich habe erst für den *KnowWareVerlag*, später für *BookBoon* und heute als **SelfPublisherin** Bücher geschrieben und veröffentlicht. Seit 2017 arbeite ich ausschließlich selbständig. Ich programmiere individuelle Webanwendungen, schreibe Fachliteratur im IT-Bereich und erstelle Websites.

Warum schreibe und übersetzte ich?

Dafür gibt es mehrere Gründe. Einer meiner Antriebe ist, dass das Erstellen von Texten mich bereichert. Ja, ich dokumentiere für mich. Ich finde, dass das Aufschreiben von Gedanken mir hilft, den Wirbelwind der Informationen im Kopf zu ordnen. Außerdem bringe ich Sachverhalte zu Papier, weil ich weiß, dass andere Menschen davon profitieren — so wie ich beim Lesen von Text fremder Autoren Nutzen ziehe. Egal ob Belletristik oder Fachliteratur.

Weitere Informationen über mich, Möglichkeiten zur Unterstützung oder Infos zu einer

Zusammenarbeit findest du auf meiner Website[3].

1.2 FAQ

Wie bekomme ich Updates?

Ich informiere auf zwei Arten über Aktualisierungen meiner Inhalte. Erfahre Neuigkeiten per E-Mail, indem du den Newsletter abonnierst oder folge mir auf Twitter. Unabhängig vom Kanal ist es mein Ziel, qualitativ hochwertige Inhalte zu teilen. Sobald du eine Benachrichtigung über eine Änderung erhalten hast, ist eine neue Version auf meiner Website verfügbar.

Ist das Lernmaterial aktuell?

Programmierbücher sind oft kurz nach ihrer Veröffentlichung schon veraltet. Da ich dieses Buch als Selfpublisher veröffentliche, ist es mir möglich, es bei Bedarf kurzfristig zu aktualisiere. Immer dann, wenn sich etwas ändert, werde ich das Buch überarbeiten und eine neue Version veröffentlichen.

Kann ich eine digitale Kopie des Buches erhalten, wenn ich es bei Amazon gekauft habe?

Erst nachdem du das Buch bei Amazon gekauft hast, stelltest du fest, dass das Buch auf meiner Website in einer digitalen Version verfügbar ist. Da ich Amazon als eine Möglichkeit verwende, für mich zu werben und Inhalte zu monetarisieren, danke ich dir für deine Unterstützung und lade dich ein, dich auf meiner Website anzumelden. Nachdem du dort ein Konto erstellt hast, schreibe mir eine E-Mail und füge eine Quittung von Amazon bei. Ich werde dann ein digitales Buch für dich freischalten. Mit einem Konto auf meiner Plattform hast du in Zukunft weiterhin Zugriff auf die neueste Version des Buches.

Wenn du ein gedrucktes Buch gekauft hast, notiere bitte deine Lernschritte im Buch. Ich habe mit Absicht die Printausgabe so gestaltet, dass größere Codefragmente genügend Platz bieten, um dir ausreichend Spielraum zum individuellen Arbeiten zu bieten.

Wie kann ich beim Lesen des Buches Hilfe bekommen?

Das Buch verbindet eine Gemeinschaft von Lernenden, die sich gegenseitig helfen und Menschen, die mitlesen. Tritt dieser Community gerne bei. So erhältst du Hilfe. Oder du hilfst anderen. Das gegenseitige Unterstützen hilft dir und anderen dabei, Wissen zu verinnerlichen. Folge der Navigation zu den Kursen auf meiner Website, melde dich dort an und navigiere zum Menüpunkt Community.

Kann ich helfen, den Inhalt zu verbessern?

[3] https://www.astrid-guenther.de

Wenn du Feedback hast, schreibe mir gerne eine E-Mail und ich werde deine Vorschläge berücksichtigen. Erwarte bitte keine direkte Antwort von mir, denn das ist mir zeitlich nicht immer möglich. Wenn du dir ein Feedback wünschst, dann frage in der Community, siehe "Wie kann ich beim Lesen des Buches Hilfe bekommen?".

Wie und wo melde ich einen Fehler?

Wenn du einen Fehler im Code findest, melde dies bitte über Github. Am Ende jedes Abschnitts findest du eine URL zum aktuellen GitHub-Projekt. Bitte eröffne hier ein Issue. Ich bin dankbar für deine Hilfe!

Wie unterstütze ich das Projekt idealerweise?

Du findest meine Lektionen nützlich und möchtest einen Beitrag leisten? Dann suche bitte auf der About-Seite meiner Website nach Informationen darüber, welche Möglichkeiten es gibt, mich zu unterstützen. In jedem Fall ist hilfreich für potentielle Leser, wenn du darüber informierst, wie meine Bücher dir geholfen haben. Nur mit Unterstützung ist es mir möglich, weiterhin kostenloses Lernmaterial anzubieten.

Was ist deine Motivation hinter dem Buch?

Mir ist es wichtig, über aktuelle Themen zu berichten. Ich stoße oft online auf Websites, die nicht aktualisiert werden oder nur einen kleinen Teil eines Themas abdecken. Viele Menschen haben Schwierigkeiten, geeignetes Lernmaterial zu finden. Ich biete aktuelle Inhalte und hoffe, dass ich andere mit meinen Projekten unterstütze, indem ich ihnen Lernmaterial kostenlos zur Verfügung stelle und etwas zurückgebe.

1.3 Für wen ist dieses Buch?

JavaScript-Anfänger

JavaScript-Anfänger mit Grundkenntnissen in CSS und HTML: Wenn du die Webentwicklung während einer Ausbildung lernst und ein grundlegendes Verständnis für CSS und HTML hast, biete dir dieses Buch alles, was du zum Erlernen von React benötigst. Wenn du dich wackelig fühlst und der Meinung bist, dass dein JavaScript-Wissen lückenhaft ist, dann schließe diese Lücke, bevor du mit dem Buch fortfährst. Im Buch wirst du zusätzlich viele Hinweise und Links zu grundlegendem Wissen finden.

JavaScript-Veteranen

jQuery-JavaScript-Veteranen: Wenn du JavaScript früher ausgiebig mit jQuery, MooTools und Dojo verwendet hast, scheint die neue JavaScript-Ära überwältigend zu sein. Das grundlegende Wissen hat sich nicht geändert, es ist nach wie vor JavaScript und HTML unter der Haube — daher hilft dieses Buch dir beim Einstieg in React.

JavaScript-Enthusiasten

JavaScript-Enthusiasten mit Kenntnissen in anderen modernen SPA-Frameworks: Wenn du Erfahrungen mit Angular oder Vue gesammelt hast, wirst du zu Beginn auf viele Dinge stoßen, die anders sind. Aber: Alle diese Frameworks bauen auf derselben Grundlage auf — JavaScript und HTML. Nach kurzem Umlernen wirst du dich schnell in React zurechtfinden.

Nicht-JavaScript-Entwickler

Wenn du eine andere Programmiersprache gelernt hast, bist du mit den verschiedenen Aspekten der Programmierung vertraut. Nachdem du dir die Grundlagen zu JavaScript, CSS und HTML angeeignet hast, wirst du React zusammen mit mir schnell lernen.

Designer und UI/UX-Enthusiasten

Arbeitest du im Bereich Design, Benutzerinteraktion oder Benutzererfahrung? Dann zögere nicht, dieses Buch in die Hand zu nehmen. Wenn du mit HTML und CSS vertraut bist, ist dies vorteilhaft. Nachdem du einige JavaScript-Grundlagen durchgearbeitet hast, wirst du die Inhalte dieses Buches verstehen. Heutzutage rücken UI und UX näher an die Implementierungsdetails heran. Es bringt dir Vorteile, wenn du weißt, wie die Dinge im Code funktionieren.

Teamleiter oder Produktmanager

Wenn du Teamleiter oder Produktmanager einer Entwicklungsabteilung bist, vermittelt dir dieses Buch eine Übersicht über alle wesentlichen Teile einer React-Anwendung. In jedem Abschnitt wird ein Konzept, ein Muster oder eine Technik erläutert. So wird Schritt für Schritt die Gesamtarchitektur aufgebaut und verbessert. Ergebnis ist eine fertige Anwendung, die alle wesentlichen Aspekte von React berücksichtigt.

2 Orientierung Kursaufbau und Übersicht

Willkommen! Diese Seite enthält einige wichtige Informationen über dieses Buch.

2.1 Course Duration and Time Requirements

This course was originally given as an instructor-led in-person training that spanned 3 to 4 days. To complete the course in its entirety, you should minimally plan on committing 25-35 hours of work. Most participants find the material to be quite challenging without peeking at solution code (see below).

2.2 Setup and Python Installation

https://www.python.org/downloads/

https://askubuntu.com/questions/1086649/how-to-update-python-to-the-latest-version-on-ubuntu-18-04 https://stackoverflow.com/questions/62496378/version-numbers-in-python https://askubuntu.com/questions/1252373/versioning-in-python sudo apt-get update && sudo apt-get upgrade sudo apt-get install python3.8

You need nothing more than a basic Python 3.6 installation or newer. There is no dependency on any particular operating system, editor, IDE, or extra Python-related tooling. There are no third-party dependencies.

That said, most of this course involves learning how to write scripts and small programs that involve data read from files. Therefore, you need to make sure you're in an environment where you can easily work with files. This includes using an editor to create Python programs and being able to run those programs from the shell/terminal.

You might be inclined to work on this course using a more interactive environment such as Jupyter Notebooks. **I DO NOT ADVISE THIS!** Although notebooks are great for experimentation, many of the exercises in this course teach concepts related to program organization. This includes working with functions, modules, import statements, and refactoring of programs whose source code spans multiple files. In my experience, it is hard to replicate this kind of working environment in notebooks.

2.3 Forking/Cloning the Course Repository

To prepare your environment for the course, I recommend creating your own fork of the course GitHub repo at https://github.com/dabeaz-course/practical-python. Once you are done, you can clone it to your local machine:

```
bash % git clone https://github.com/yourname/practical-python
bash % cd practical-python
bash %
```

Do all of your work within the `practical-python/` directory. If you commit your solution code back to your fork of the repository, it will keep all of your code together in one place and you'll have a nice historical record of your work when you're done.

If you don't want to create a personal fork or don't have a GitHub account, you can still clone the course directory to your machine:

```
bash % git clone https://github.com/dabeaz-course/practical-python
bash % cd practical-python
bash %
```

With this option, you just won't be able to commit code changes except to the local copy on your machine.

2.4 Coursework Layout

Do all of your coding work in the `Work/` directory. Within that directory, there is a `Data/` directory. The `Data/` directory contains a variety of datafiles and other scripts used during the course. You will frequently have to access files located in `Data/`. Course exercises are written with the assumption that you are creating programs in the `Work/` directory.

2.5 Course Order

Course material should be completed in section order, starting with section 1. Course exercises in later sections build upon code written in earlier sections. Many of the later exercises involve minor refactoring of existing code.

2.6 Solution Code

The `Solutions/` directory contains full solution code to selected exercises. Feel free to look at this if you need a hint. To get the most out of the course however, you should try to create your own solutions first.

Contents

3 Practical Python Programming

3.1 Table of Contents

Please see the Instructor Notes if you plan on teaching the course.

Home

Contents | Next (2 Working With Data)

3.2 1. Python stelllt sich vor

Das Ziel dieses ersten Abschnitts ist es, einige Python-Grundlagen von Grund auf anzusehen. Du lernst, wie du kleine Programme bearbeitest, ausführst und debuggst. Letztendlich schreibst du selbst ein kurzes Skript, das eine CSV-Datendatei liest und eine einfache Berechnung durchführt. Hört sich gut an, oder?

Contents | Next (2 Working With Data)

Contents | Next (1.2 Ein erstes Programm)

4 1.1 Python

4.0.1 What is Python?

Python is an interpreted high level programming language. It is often classified as a "scripting language" and is considered similar to languages such as Perl, Tcl, or Ruby. The syntax of Python is loosely inspired by elements of C programming.

Python was created by Guido van Rossum around 1990 who named it in honor of Monty Python.

4.0.2 Where to get Python?

Python.org is where you obtain Python. For the purposes of this course, you only need a basic installation. I recommend installing Python 3.6 or newer. Python 3.6 is used in the notes and solutions.

4.0.3 Why was Python created?

In the words of Python's creator:

> My original motivation for creating Python was the perceived need for a higher level language in the Amoeba [Operating Systems] project. I realized that the development of system administration utilities in C was taking too long. Moreover, doing these things in the Bourne shell wouldn't work for a variety of reasons. … So, there was a need for a language that would bridge the gap between C and the shell.
>
> - Guido van Rossum

4.0.4 Where is Python on my Machine?

Although there are many environments in which you might run Python, Python is typically installed on your machine as a program that runs from the terminal or command shell. From the terminal, you should be able to type `python` like this:

```
bash $ python
Python 3.8.1 (default, Feb 20 2020, 09:29:22)
[Clang 10.0.0 (clang-1000.10.44.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hallo Python")
Hallo Python
>>>
```

If you are new to using the shell or a terminal, you should probably stop, finish a short tutorial on that first, and then return here.

Although there are many non-shell environments where you can code Python, you will be a stronger Python programmer if you are able to run, debug, and interact with Python at the terminal. This is Python's native environment. If you are able to use Python here, you will be able to use it everywhere else.

4.1 Exercises

4.1.1 Exercise 1.1: Using Python as a Calculator

On your machine, start Python and use it as a calulator to solve the following problem.

Lucky Larry bought 75 shares of Google stock at a price of $235.14 per share. Today, shares of Google are priced at $711.25. Using Python's interactive mode as a calculator, figure out how much profit Larry would make if he sold all of his shares.

```
>>> (711.25 - 235.14) * 75
35708.25
>>>
```

Pro-tip: Use the underscore (_) variable to use the result of the last calculation. For example, how much profit does Larry make after his evil broker takes their 20% cut?

```
>>> _ * 0.80
28566.600000000002
>>>
```

4.1.2 Exercise 1.2: Getting help

Use the `help()` command to get help on the `abs()` function. Then use `help()` to get help on the `round()` function. Type `help()` just by itself with no value to enter the interactive help viewer.

One caution with `help()` is that it doesn't work for basic Python statements such as `for`, `if`, `while`, and so forth (i.e., if you type `help(for)` you'll get a syntax error). You can try putting the help topic in quotes such as `help("for")` instead. If that doesn't work, you'll have to turn to an internet search.

Followup: Go to http://docs.python.org and find the documentation for the `abs()` function (hint: it's found under the library reference related to built-in functions).

4.1.3 Exercise 1.3: Cutting and Pasting

This course is structured as a series of traditional web pages where you are encouraged to try interactive Python code samples **by typing them out by hand.** If you are learning Python for the first time, this "slow approach" is encouraged. You will get a better feel for the language by slowing down, typing things in, and thinking about what you are doing.

If you must "cut and paste" code samples, select code starting after the `>>>` prompt and going up to, but not any further than the first blank line or the next `>>>` prompt (whichever appears first). Select "copy" from the browser, go to the Python window, and select "paste" to copy it into the Python shell. To get the code to run, you may have to hit "Return" once after you've pasted it in.

Use cut-and-paste to execute the Python statements in this session:

```
>>> 12 + 20
32
>>> (3 + 4
```

```
         + 5 + 6)
18
>>> for i in range(5):
        print(i)

0
1
2
3
4
>>>
```

Warning: It is never possible to paste more than one Python command (statements that appear after >>>) to the basic Python shell at a time. You have to paste each command one at a time.

Now that you've done this, just remember that you will get more out of the class by typing in code slowly and thinking about it–not cut and pasting.

4.1.4 Exercise 1.4: Where is My Bus?

Try something more advanced and type these statements to find out how long people waiting on the corner of Clark street and Balmoral in Chicago will have to wait for the next northbound CTA #22 bus:

```
>>> import urllib.request
>>> u = urllib.request.urlopen('http://ctabustracker.com/bustime/map/getStopPredictio
>>> from xml.etree.ElementTree import parse
>>> doc = parse(u)
>>> for pt in doc.findall('.//pt'):
        print(pt.text)

6 MIN
18 MIN
28 MIN
>>>
```

Yes, you just downloaded a web page, parsed an XML document, and extracted some useful information in about 6 lines of code. The data you accessed is actually feeding the website http://ctabustracker.com/bustime/home.jsp. Try it again and watch the predictions change.

Note: This service only reports arrival times within the next 30 minutes. If you're in a different timezone and it happens to be 3am in Chicago, you might not get any output.

You use the tracker link above to double check.

If the first import statement `import urllib.request` fails, you're probably using Python 2. For this course, you need to make sure you're using Python 3.6 or newer. Go to https://www.python.org to download it if you need it.

If your work environment requires the use of an HTTP proxy server, you may need to set the `HTTP_PROXY` environment variable to make this part of the exercise work. For example:

```
>>> import os
>>> os.environ['HTTP_PROXY'] = 'http://yourproxy.server.com'
>>>
```

If you can't make this work, don't worry about it. The rest of this course has nothing to do with parsing XML.

Contents | Next (1.2 Ein erstes Programm)

Contents | Previous (1.1 Python) | Next (1.3 Numbers)

5 1.2 Ein erstes Programm

In diesem Abschnitt erstellst du dein erstes Programms, du lernst den Interpreter und das Debuggen kennen.

5.0.1 Python ausführen

Python-Programme laufen immer in einem Interpreter.

Der Interpreter ist eine "konsolenbasierte" Anwendung, die über eine Befehlszeile ausgeführt wird.

```
python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Erfahrene Programmierer haben kein Problem damit, die Befehlszeile zu nutzen. Für Anfänger ist dies oft ein No-Go. Verwendest du lieber eine Umgebung, die eine benutzerfreundlichere Schnittstelle zu Python bietet? Das ist in Ordnung. Trotzdem ist es sinnvoll, sich mit dem Terminal anzufreunden. So schließt du Probleme, die außerhalb von Python begründet sind, schnell aus. Ich lerne aus diesem Grund mit der Befehlszeile.

5.0.2 Interaktiver Modus

Wenn du Python aufrufst, öffnet sich der *interaktive* Modus, in dem du experimentierst.

Wenn du eine Anweisung eingibst, wird diese sofort abgeschlossen. Es gibt keinen Kreislauf in der Form von bearbeiten/kompilieren/ausführen/debuggen oder in gewohntem Englisch: edit/compile/run/debug. Probiere es aus, es macht Spaß:

```
>>> print('Hallo Python')
Hallo Python
>>> 37*2
74
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

Dieser sogenannte *Direkt-Modus oder REPL*[4] ist nützlich für das Debuggen und Experimentieren.

> **Was ist REPL?** Der Begriff REPL ist eine Abkürzung für *Read, Evaluate, Print and Loop* oder *Lesen, Auswerten, Drucken und Wiederholen*. REPL ist eine interaktive Möglichkeit, mit dem Computer in Python zu kommunizieren. Damit dies funktioniert, führt der Computer vier Dinge aus:
> - Lesen der Benutzereingaben oder des Python-Befehles.
> - Bewerten des Codes, um herauszufinden, was gemeint ist.
> - Ausdruck aller Ergebnisse damit du die Antwort siehst.
> - Rückkehr zu Schritt 1 um das Gespräch mit dir fortzusetzten.

STOP: Wenn du bisher Probleme hast, löse diese erst. Im Vorwort findest du Links Hilfsangeboten. Verwendest du eine IDE[5]? Ich gehe hier davon aus, dass deine Installation korrekt ist und du vertraut mit ihr bist.

Schauen wir uns die Elemente der REPL genauer an:

- `>>>` ist die Interpreter-Eingabeaufforderung. Sie erwartet eine Anweisung.
- `...` ist die Interpreter-Eingabeaufforderung zum Fortsetzen einer Anweisung.

Die drei Punkte `...` werden nicht in jeder Umgebung angezeigt. In diesem Text er-

[4] https://de.wikipedia.org/wiki/Direct_mode
[5] https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

setzt ich diese mit Leerzeichen. Das hat gleichzeitig den Vorteil, dass Codebeispiele unkompliziert kopiert werden können.

Mit dem Unterstrich _ greifst du auf das letzte gespeicherte Ergebnis zu.

```
>>> 37 * 42
1554
>>> _ * 2
3108
>>> _ + 50
3158
>>>
```

Das gibt nur im interaktiven Modus. Nutze den Unterstrich nicht _ in einem Programm.

5.0.3 Programme erstellen

Programme werden in `.py`-Dateien gespeichert.

```
# hello.py
print('Hallo Python')
```

Erstelle diese Dateien mit dem Texteditor deiner Wahl.

5.0.4 Ausführen von Programmen

Um ein Programm auszuführen, führe es mit dem Befehl `python` aus. Zum Beispiel über die Befehlzeile mit `python hello.py`, wenn du dich im gleichen Verzeichnis wie die Datei `hello.py` befindest:

```
python hello.py
Hallo Python
```

Oder über die Shell in Windows:

```
C:\SomeFolder>hello.py
Hallo Python

C:\SomeFolder>c:\python36\python hello.py
Hallo Python
```

Hinweis: Unter Umständen ist erforderlich, den vollständigen Pfad zum Python-Interpreter anzugeben. Beispielsweise `c:\python36\python`.

5.0.5 Ein Beispielprogramm

Lösen wir das folgende Problem:

> Eines Morgens legst einen Dollarschein auf den Bürgersteig neben dem Sears Tower in Chicago ab. An jedem darauffolgenden Tag verdoppelst du die Anzahl an Scheinen und legst diese dazu. Wie lange dauert es, bis der Geldscheinstapel die Höhe des Turms überschreitet?

Hier ist eine Lösung:

```
# sears.py
bill_thickness = 0.11 * 0.001 # Meter (0.11 mm)
sears_height = 442 # Höhe (Meter)
num_bills = 1
day = 1

while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2

print('Anzahl der Tage', day)
print('Anzahl der Geldscheine', num_bills)
print('Endhöhe', num_bills * bill_thickness)
```

Beim Aufruf siehst du die folgende Ausgabe:

```
python3 sears.py
1 1 0.00011
2 2 0.00022
3 4 0.00044
4 8 0.00088
5 16 0.00176
...
21 1048576 115.34336
22 2097152 230.68672
Anzahl der Tage 23
Anzahl der Geldscheine 4194304
Endhöhe 461.37344
```

Lerne anhnad dieses Programms eine Reihe wichtiger Kernkonzepte.

5.0.6 Anweisungen

Ein Python-Programm ist eine Folge von Anweisungen:

```
a = 3 + 4
b = a * 2
print(b)
```

Each statement is terminated by a newline. Statements are executed one after the other until control reaches the end of the file.

5.0.7 Comments

Comments are text that will not be executed.

```
a = 3 + 4
# This is a comment
b = a * 2
print(b)
```

Comments are denoted by `#` and extend to the end of the line.

5.0.8 Variables

A variable is a name for a value. You can use letters (lower and upper-case) from a to z. As well as the character underscore `_`. Numbers can also be part of the name of a variable, except as the first character.

```
height = 442 # valid
_height = 442 # valid
height2 = 442 # valid
2height = 442 # invalid
```

5.0.9 Types

Variables do not need to be declared with the type of the value. The type is associated with the value on the right hand side, not name of the variable.

```
height = 442           # An integer
height = 442.0         # Floating point
height = 'Really tall' # A string
```

Python is dynamically typed. The perceived "type" of a variable might change as a program executes depending on the current value assigned to it.

5.0.10 Case Sensitivity

Python is case sensitive. Upper and lower-case letters are considered different letters. These are all different variables:

```
name = 'Jake'
Name = 'Elwood'
NAME = 'Guido'
```

Language statements are always lower-case.

```
while x < 0:   # OK
WHILE x < 0:   # ERROR
```

5.0.11 Looping

The `while` statement executes a loop.

```
while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2

print('Number of days', days)
```

The statements indented below the `while` will execute as long as the expression after the `while` is `true`.

5.0.12 Indentation

Indentation is used to denote groups of statements that go together. Consider the previous example:

```
while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2

print('Number of days', days)
```

Indentation groups the following statements together as the operations that repeat:

```
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2
```

Because the `print()` statement at the end is not indented, it does not belong to the loop. The empty line is just for readability. It does not affect the execution.

5.0.13 Indentation best practices

- Use spaces instead of tabs.
- Use 4 spaces per level.
- Use a Python-aware editor.

Python's only requirement is that indentation within the same block be consistent. For example, this is an error:

```
while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
        day = day + 1 # ERROR
    num_bills = num_bills * 2
```

5.0.14 Conditionals

The `if` statement is used to execute a conditional:

```
if a > b:
    print('Computer says no')
else:
    print('Computer says yes')
```

You can check for multiple conditions by adding extra checks using `elif`.

```
if a > b:
    print('Computer says no')
elif a == b:
    print('Computer says yes')
else:
    print('Computer says maybe')
```

5.0.15 Printing

The `print` function produces a single line of text with the values passed.

```
print('Hallo Python!') # Prints the text 'Hallo Python!'
```

You can use variables. The text printed will be the value of the variable, not the name.

```
x = 100
print(x) # Prints the text '100'
```

If you pass more than one value to `print` they are separated by spaces.

```
name = 'Jake'
print('My name is', name) # Print the text 'My name is Jake'
```

`print()` always puts a newline at the end.

```
print('Hello')
print('My name is', 'Jake')
```

This prints:

```
Hello
My name is Jake
```

The extra newline can be suppressed:

```
print('Hello', end=' ')
print('My name is', 'Jake')
```

This code will now print:

```
Hello My name is Jake
```

5.0.16 User input

To read a line of typed user input, use the `input()` function:

```
name = input('Enter your name:')
print('Your name is', name)
```

`input` prints a prompt to the user and returns their response. This is useful for small programs, learning exercises or simple debugging. It is not widely used for real programs.

5.0.17 pass statement

Sometimes you need to specify an empty code block. The keyword `pass` is used for it.

```
if a > b:
    pass
else:
    print('Computer says false')
```

This is also called a "no-op" statement. It does nothing. It serves as a placeholder for statements, possibly to be added later.

5.1 Exercises

This is the first set of exercises where you need to create Python files and run them. From this point forward, it is assumed that you are editing files in the `practical-python/Work/` directory. To help you locate the proper place, a number of empty starter files have been created with the appropriate filenames. Look for the file `Work/bounce.py` that's used in the first exercise.

5.1.1 Exercise 1.5: The Bouncing Ball

A rubber ball is dropped from a height of 100 meters and each time it hits the ground, it bounces back up to 3/5 the height it fell. Write a program `bounce.py` that prints a table showing the height of the first 10 bounces.

Your program should make a table that looks something like this:

```
1 60.0
2 36.0
3 21.599999999999998
4 12.959999999999999
5 7.775999999999999
6 4.6655999999999995
7 2.7993599999999996
8 1.6796159999999998
9 1.0077695999999998
10 0.6046617599999998
```

Note: You can clean up the output a bit if you use the round() function. Try using it to round the output to 4 digits.

```
1 60.0
2 36.0
3 21.6
4 12.96
5 7.776
6 4.6656
7 2.7994
8 1.6796
9 1.0078
10 0.6047
```

5.1.2 Exercise 1.6: Debugging

The following code fragment contains code from the Sears tower problem. It also has a bug in it.

```
# sears.py

bill_thickness = 0.11 * 0.001    # Meters (0.11 mm)
sears_height   = 442             # Height (meters)
num_bills      = 1
day            = 1

while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = days + 1
    num_bills = num_bills * 2

print('Number of days', day)
print('Number of bills', num_bills)
print('Final height', num_bills * bill_thickness)
```

Copy and paste the code that appears above in a new program called `sears.py`. When you run the code you will get an error message that causes the program to crash like this:

```
Traceback (most recent call last):
  File "sears.py", line 10, in <module>
    day = days + 1
NameError: name 'days' is not defined
```

Reading error messages is an important part of Python code. If your program crashes, the very last line of the traceback message is the actual reason why the the program crashed. Above that, you should see a fragment of source code and then an identifying filename and line number.

- Which line is the error?
- What is the error?
- Fix the error
- Run the program successfully

Contents | Previous (1.1 Python) | Next (1.3 Numbers)

Contents | Previous (1.2 Ein erstes Programm) | Next (1.4 Strings)

6 1.3 Numbers

This section discusses mathematical calculations.

6.0.1 Types of Numbers

Python has 4 types of numbers:

- Booleans
- Integers
- Floating point
- Complex (imaginary numbers)

6.0.2 Booleans (bool)

Booleans have two values: `True`, `False`.

```
a = True
b = False
```

Numerically, they're evaluated as integers with value `1`, `0`.

```
c = 4 + True # 5
d = False
if d == 0:
    print('d is False')
```

But, don't write code like that. It would be odd.

6.0.3 Integers (int)

Signed values of arbitrary size and base:

```
a = 37
b = -29939299372771662737712848181224123 1
c = 0x7fa8      # Hexadecimal
d = 0o253       # Octal
e = 0b10001111  # Binary
```

Common operations:

```
x + y      Add
x - y      Subtract
x * y      Multiply
```

```
x / y       Divide (produces a float)
x // y      Floor Divide (produces an integer)
x % y       Modulo (remainder)
x ** y      Power
x << n      Bit shift left
x >> n      Bit shift right
x & y       Bit-wise AND
x | y       Bit-wise OR
x ^ y       Bit-wise XOR
~x          Bit-wise NOT
abs(x)      Absolute value
```

6.0.4 Floating point (float)

Use a decimal or exponential notation to specify a floating point value:

```
a = 37.45
b = 4e5 # 4 x 10**5 or 400,000
c = -1.345e-10
```

Floats are represented as double precision using the native CPU representation IEEE 754. This is the same as the `double` type in the programming language C.

17 digits or precision Exponent from -308 to 308

Be aware that floating point numbers are inexact when representing decimals.

```
>>> a = 2.1 + 4.2
>>> a == 6.3
False
>>> a
6.300000000000001
>>>
```

This is **not a Python issue**, but the underlying floating point hardware on the CPU.

Common Operations:

```
x + y       Add
x - y       Subtract
x * y       Multiply
x / y       Divide
x // y      Floor Divide
x % y       Modulo
x ** y      Power
abs(x)      Absolute Value
```

These are the same operators as Integers, except for the bit-wise operators. Additional math functions are found in the `math` module.

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
d = math.tan(x)
e = math.log(x)
```

6.0.5 Comparisons

The following comparison / relational operators work with numbers:

```
x < y      Less than
x <= y     Less than or equal
x > y      Greater than
x >= y     Greater than or equal
x == y     Equal to
x != y     Not equal to
```

You can form more complex boolean expressions using

`and`, `or`, `not`

Here are a few examples:

```
if b >= a and b <= c:
    print('b is between a and c')

if not (b < a or b > c):
    print('b is still between a and c')
```

6.0.6 Converting Numbers

The type name can be used to convert values:

```
a = int(x)    # Convert x to integer
b = float(x)  # Convert x to float
```

Try it out.

```
>>> a = 3.14159
>>> int(a)
3
```

```
>>> b = '3.14159' # It also works with strings containing numbers
>>> float(b)
3.14159
>>>
```

6.1 Exercises

Reminder: These exercises assume you are working in the `practical-python/Work` directory. Look for the file `mortgage.py`.

6.1.1 Exercise 1.7: Dave's mortgage

Dave has decided to take out a 30-year fixed rate mortgage of $500,000 with Guido's Mortgage, Stock Investment, and Bitcoin trading corporation. The interest rate is 5% and the monthly payment is $2684.11.

Here is a program that calculates the total amount that Dave will have to pay over the life of the mortgage:

```
# mortgage.py

principal = 500000.0
rate = 0.05
payment = 2684.11
total_paid = 0.0

while principal > 0:
    principal = principal * (1+rate/12) - payment
    total_paid = total_paid + payment

print('Total paid', total_paid)
```

Enter this program and run it. You should get an answer of `966,279.6`.

6.1.2 Exercise 1.8: Extra payments

Suppose Dave pays an extra $1000/month for the first 12 months of the mortgage?

Modify the program to incorporate this extra payment and have it print the total amount paid along with the number of months required.

When you run the new program, it should report a total payment of `929,965.62` over 342 months.

6.1.3 Exercise 1.9: Making an Extra Payment Calculator

Modify the program so that extra payment information can be more generally handled. Make it so that the user can set these variables:

```
extra_payment_start_month = 60
extra_payment_end_month = 108
extra_payment = 1000
```

Make the program look at these variables and calculate the total paid appropriately.

How much will Dave pay if he pays an extra $1000/month for 4 years starting in year 5 of the mortgage?

6.1.4 Exercise 1.10: Making a table

Modify the program to print out a table showing the month, total paid so far, and the remaining principal. The output should look something like this:

```
1 2684.11 499399.22
2 5368.22 498795.94
3 8052.33 498190.15
4 10736.44 497581.83
5 13420.55 496970.98
...
308 874705.88 2971.43
309 877389.99 299.7
310 880074.1 -2383.16
Total paid 880074.1
Months 310
```

6.1.5 Exercise 1.11: Bonus

While you're at it, fix the program to correct for the overpayment that occurs in the last month.

6.1.6 Exercise 1.12: A Mystery

`int()` and `float()` can be used to convert numbers. For example,

```
>>> int("123")
123
>>> float("1.23")
1.23
>>>
```

With that in mind, can you explain this behavior?

```
>>> bool("False")
True
>>>
```

Contents | Previous (1.2 Ein erstes Programm) | Next (1.4 Strings)

Contents | Previous (1.3 Numbers) | Next (1.5 Lists)

7 1.4 Strings

This section introduces ways to work with text.

7.0.1 Representing Literal Text

String literals are written in programs with quotes.

```
# Single quote
a = 'Yeah but no but yeah but...'

# Double quote
b = "computer says no"

# Triple quotes
c = '''
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes,
don't look around the eyes,
look into my eyes, you're under.
'''
```

Normally strings may only span a single line. Triple quotes capture all text enclosed across multiple lines including all formatting.

There is no difference between using single (') versus double (") quotes. The same type of quote used to start a string must be used to terminate it.

7.0.2 String escape codes

Escape codes are used to represent control characters and characters that can't be easily typed directly at the keyboard. Here are some common escape codes:

```
'\n'      Line feed
'\r'      Carriage return
'\t'      Tab
'\''      Literal single quote
'\"'      Literal double quote
'\\'      Literal backslash
```

7.0.3 String Representation

Each character in a string is stored internally as a so-called Unicode "code-point" which is an integer. You can specify an exact code-point value using the following escape sequences:

```
a = '\xf1'          # a = 'ñ'
b = '\u2200'        # b = ' '
c = '\U0001D122'    # c = ' '
d = '\N{FOR ALL}'   # d = ' '
```

The Unicode Character Database is a reference for all available character codes.

7.0.4 String Indexing

Strings work like an array for accessing individual characters. You use an integer index, starting at 0. Negative indices specify a position relative to the end of the string.

```
a = 'Hallo Python'
b = a[0]          # 'H'
c = a[4]          # 'o'
d = a[-1]         # 'd' (end of string)
```

You can also slice or select substrings specifying a range of indices with `:`.

```
d = a[:5]     # 'Hello'
e = a[6:]     # 'world'
f = a[3:8]    # 'lo wo'
g = a[-5:]    # 'world'
```

The character at the ending index is not included. Missing indices assume the beginning or ending of the string.

7.0.5 String operations

Concatenation, length, membership and replication.

```
# Concatenation (+)
a = 'Hello' + 'World'   # 'HelloWorld'
b = 'Say ' + a          # 'Say HelloWorld'

# Length (len)
s = 'Hello'
len(s)                  # 5

# Membership test (`in`, `not in`)
t = 'e' in s            # True
f = 'x' in s            # False
g = 'hi' not in s       # True

# Replication (s * n)
rep = s * 5             # 'HelloHelloHelloHelloHello'
```

7.0.6 String methods

Strings have methods that perform various operations with the string data.

Example: stripping any leading / trailing white space.

```
s = '  Hello '
t = s.strip()     # 'Hello'
```

Example: Case conversion.

```
s = 'Hello'
l = s.lower()     # 'hello'
u = s.upper()     # 'HELLO'
```

Example: Replacing text.

```
s = 'Hallo Python'
t = s.replace('Hello' , 'Hallo')   # 'Hallo world'
```

More string methods:

Strings have a wide variety of other methods for testing and manipulating the text data. This is a small sample of methods:

```
s.endswith(suffix)     # Check if string ends with suffix
s.find(t)              # First occurrence of t in s
s.index(t)             # First occurrence of t in s
s.isalpha()            # Check if characters are alphabetic
s.isdigit()            # Check if characters are numeric
s.islower()            # Check if characters are lower-case
s.isupper()            # Check if characters are upper-case
s.join(slist)          # Join a list of strings using s as delimiter
s.lower()              # Convert to lower case
s.replace(old,new)     # Replace text
s.rfind(t)             # Search for t from end of string
s.rindex(t)            # Search for t from end of string
s.split([delim])       # Split string into list of substrings
s.startswith(prefix)   # Check if string starts with prefix
s.strip()              # Strip leading/trailing space
s.upper()              # Convert to upper case
```

7.0.7 String Mutability

Strings are "immutable" or read-only. Once created, the value can't be changed.

```
>>> s = 'Hallo Python'
>>> s[1] = 'a'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

All operations and methods that manipulate string data, always create new strings.

7.0.8 String Conversions

Use `str()` to convert any value to a string. The result is a string holding the same text that would have been produced by the `print()` statement.

```
>>> x = 42
>>> str(x)
'42'
>>>
```

7.0.9 Byte Strings

A string of 8-bit bytes, commonly encountered with low-level I/O, is written as follows:

```
data = b'Hallo Python\r\n'
```

By putting a little b before the first quotation, you specify that it is a byte string as opposed to a text string.

Most of the usual string operations work.

```
len(data)                          # 13
data[0:5]                          # b'Hello'
data.replace(b'Hello', b'Cruel')   # b'Cruel World\r\n'
```

Indexing is a bit different because it returns byte values as integers.

```
data[0]   # 72 (ASCII code for 'H')
```

Conversion to/from text strings.

```
text = data.decode('utf-8') # bytes -> text
data = text.encode('utf-8') # text -> bytes
```

The `'utf-8'` argument specifies a character encoding. Other common values include `'ascii'` and `'latin1'`.

7.0.10 Raw Strings

Raw strings are string literals with an uninterpreted backslash. They are specified by prefixing the initial quote with a lowercase "r".

```
>>> rs = r'c:\newdata\test' # Raw (uninterpreted backslash)
>>> rs
'c:\\newdata\\test'
```

The string is the literal text enclosed inside, exactly as typed. This is useful in situations where the backslash has special significance. Example: filename, regular expressions, etc.

7.0.11 f-Strings

A string with formatted expression substitution.

```
>>> name = 'IBM'
>>> shares = 100
>>> price = 91.1
```

```
>>> a = f'{name:>10s} {shares:10d} {price:10.2f}'
>>> a
'       IBM        100      91.10'
>>> b = f'Cost = ${shares*price:0.2f}'
>>> b
'Cost = $9110.00'
>>>
```

Note: This requires Python 3.6 or newer. The meaning of the format codes is covered later.

7.1 Exercises

In these exercises, you'll experiment with operations on Python's string type. You should do this at the Python interactive prompt where you can easily see the results. Important note:

> In exercises where you are supposed to interact with the interpreter, `>>>` is the interpreter prompt that you get when Python wants you to type a new statement. Some statements in the exercise span multiple lines–to get these statements to run, you may have to hit 'return' a few times. Just a reminder that you *DO NOT* type the `>>>` when working these examples.

Start by defining a string containing a series of stock ticker symbols like this:

```
>>> symbols = 'AAPL,IBM,MSFT,YHOO,SCO'
>>>
```

7.1.1 Exercise 1.13: Extracting individual characters and substrings

Strings are arrays of characters. Try extracting a few characters:

```
>>> symbols[0]
?
>>> symbols[1]
?
>>> symbols[2]
?
>>> symbols[-1]        # Last character
?
>>> symbols[-2]        # Negative indices are from end of string
?
>>>
```

In Python, strings are read-only.

Verify this by trying to change the first character of `symbols` to a lower-case 'a'.

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

7.1.2 Exercise 1.14: String concatenation

Although string data is read-only, you can always reassign a variable to a newly created string.

Try the following statement which concatenates a new symbol "GOOG" to the end of `symbols`:

```
>>> symbols = symbols + 'GOOG'
>>> symbols
'AAPL,IBM,MSFT,YHOO,SCOGOOG'
>>>
```

Oops! That's not what you wanted. Fix it so that the `symbols` variable holds the value `'AAPL,IBM,MSFT,YHOO,SCO,GOOG'`.

```
>>> symbols = ?
>>> symbols
'AAPL,IBM,MSFT,YHOO,SCO,GOOG'
>>>
```

Add `'HPQ'` to the front the string:

```
>>> symbols = ?
>>> symbols
'HPQ,AAPL,IBM,MSFT,YHOO,SCO,GOOG'
>>>
```

In these examples, it might look like the original string is being modified, in an apparent violation of strings being read only. Not so. Operations on strings create an entirely new string each time. When the variable name `symbols` is reassigned, it points to the newly created string. Afterwards, the old string is destroyed since it's not being used anymore.

7.1.3 Exercise 1.15: Membership testing (substring testing)

Experiment with the `in` operator to check for substrings. At the interactive prompt, try these operations:

```
>>> 'IBM' in symbols
?
>>> 'AA' in symbols
True
>>> 'CAT' in symbols
?
>>>
```

Why did the check for `'AA'` return `True`?

7.1.4 Exercise 1.16: String Methods

At the Python interactive prompt, try experimenting with some of the string methods.

```
>>> symbols.lower()
?
>>> symbols
?
>>>
```

Remember, strings are always read-only. If you want to save the result of an operation, you need to place it in a variable:

```
>>> lowersyms = symbols.lower()
>>>
```

Try some more operations:

```
>>> symbols.find('MSFT')
?
>>> symbols[13:17]
?
>>> symbols = symbols.replace('SCO','DOA')
>>> symbols
?
>>> name = '   IBM   \n'
>>> name = name.strip()    # Remove surrounding whitespace
>>> name
?
>>>
```

7.1.5 Exercise 1.17: f-strings

Sometimes you want to create a string and embed the values of variables into it.

To do that, use an f-string. For example:

```
>>> name = 'IBM'
>>> shares = 100
>>> price = 91.1
>>> f'{shares} shares of {name} at ${price:0.2f}'
'100 shares of IBM at $91.10'
>>>
```

Modify the `mortgage.py` program from Exercise 1.10 to create its output using f-strings. Try to make it so that output is nicely aligned.

7.1.6 Exercise 1.18: Regular Expressions

One limitation of the basic string operations is that they don't support any kind of advanced pattern matching. For that, you need to turn to Python's `re` module and regular expressions. Regular expression handling is a big topic, but here is a short example:

```
>>> text = 'Today is 3/27/2018. Tomorrow is 3/28/2018.'
>>> # Find all occurrences of a date
>>> import re
>>> re.findall(r'\d+/\d+/\d+', text)
['3/27/2018', '3/28/2018']
>>> # Replace all occurrences of a date with replacement text
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2018-3-27. Tomorrow is 2018-3-28.'
>>>
```

For more information about the `re` module, see the official documentation at https://docs.python.org/library/re.html.

7.1.7 Commentary

As you start to experiment with the interpreter, you often want to know more about the operations supported by different objects. For example, how do you find out what operations are available on a string?

Depending on your Python environment, you might be able to see a list of available methods via tab-completion. For example, try typing this:

```
>>> s = 'Hallo Python'
>>> s.<tab key>
>>>
```

If hitting tab doesn't do anything, you can fall back to the builtin-in `dir()` function. For example:

```
>>> s = 'hello'
>>> dir(s)
['__add__', '__class__', '__contains__', ..., 'find', 'format',
'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>>
```

`dir()` produces a list of all operations that can appear after the `(.)`. Use the `help()` command to get more information about a specific operation:

```
>>> help(s.upper)
Help on built-in function upper:

upper(...)
    S.upper() -> string

    Return a copy of the string S converted to uppercase.
>>>
```

Contents | Previous (1.3 Numbers) | Next (1.5 Lists)

Contents | Previous (1.4 Strings) | Next (1.6 Files)

8 1.5 Lists

This section introduces lists, Python's primary type for holding an ordered collection of values.

8.0.1 Creating a List

Use square brackets to define a list literal:

```
names = [ 'Elwood', 'Jake', 'Curtis' ]
nums = [ 39, 38, 42, 65, 111]
```

Sometimes lists are created by other methods. For example, a string can be split into a list using the `split()` method:

```
>>> line = 'GOOG,100,490.10'
>>> row = line.split(',')
>>> row
['GOOG', '100', '490.10']
>>>
```

8.0.2 List operations

Lists can hold items of any type. Add a new item using `append()`:

```
names.append('Murphy')    # Adds at end
names.insert(2, 'Aretha') # Inserts in middle
```

Use `+` to concatenate lists:

```
s = [1, 2, 3]
t = ['a', 'b']
s + t           # [1, 2, 3, 'a', 'b']
```

Lists are indexed by integers. Starting at 0.

```
names = [ 'Elwood', 'Jake', 'Curtis' ]

names[0]  # 'Elwood'
names[1]  # 'Jake'
names[2]  # 'Curtis'
```

Negative indices count from the end.

```
names[-1] # 'Curtis'
```

You can change any item in a list.

```
names[1] = 'Joliet Jake'
names                    # [ 'Elwood', 'Joliet Jake', 'Curtis' ]
```

Length of the list.

```
names = ['Elwood','Jake','Curtis']
len(names)  # 3
```

Membership test (`in`, `not in`).

```
'Elwood' in names       # True
'Britney' not in names  # True
```

Replication (`s * n`).

```
s = [1, 2, 3]
s * 3   # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

8.0.3 List Iteration and Search

Use `for` to iterate over the list contents.

```
for name in names:
    # use name
    # e.g. print(name)
    ...
```

This is similar to a `foreach` statement from other programming languages.

To find the position of something quickly, use `index()`.

```
names = ['Elwood','Jake','Curtis']
names.index('Curtis')   # 2
```

If the element is present more than once, `index()` will return the index of the first occurrence.

If the element is not found, it will raise a `ValueError` exception.

8.0.4 List Removal

You can remove items either by element value or by index:

```
# Using the value
names.remove('Curtis')

# Using the index
del names[1]
```

Removing an item does not create a hole. Other items will move down to fill the space vacated. If there are more than one occurrence of the element, `remove()` will remove only the first occurrence.

8.0.5 List Sorting

Lists can be sorted "in-place".

```
s = [10, 1, 7, 3]
s.sort()                    # [1, 3, 7, 10]

# Reverse order
s = [10, 1, 7, 3]
s.sort(reverse=True)        # [10, 7, 3, 1]

# It works with any ordered data
s = ['foo', 'bar', 'spam']
s.sort()                    # ['bar', 'foo', 'spam']
```

Use `sorted()` if you'd like to make a new list instead:

```
t = sorted(s)               # s unchanged, t holds sorted values
```

8.0.6 Lists and Math

Caution: Lists were not designed for math operations.

```
>>> nums = [1, 2, 3, 4, 5]
>>> nums * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> nums + [10, 11, 12, 13, 14]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
```

Specifically, lists don't represent vectors/matrices as in MATLAB, Octave, R, etc. However, there are some packages to help you with that (e.g. numpy).

8.1 Exercises

In this exercise, we experiment with Python's list datatype. In the last section, you worked with strings containing stock symbols.

```
>>> symbols = 'HPQ,AAPL,IBM,MSFT,YHOO,DOA,GOOG'
```

Split it into a list of names using the `split()` operation of strings:

```
>>> symlist = symbols.split(',')
```

8.1.1 Exercise 1.19: Extracting and reassigning list elements

Try a few lookups:

```
>>> symlist[0]
'HPQ'
>>> symlist[1]
'AAPL'
>>> symlist[-1]
'GOOG'
>>> symlist[-2]
'DOA'
>>>
```

Try reassigning one value:

```
>>> symlist[2] = 'AIG'
>>> symlist
['HPQ', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'DOA', 'GOOG']
>>>
```

Take a few slices:

```
>>> symlist[0:3]
['HPQ', 'AAPL', 'AIG']
>>> symlist[-2:]
['DOA', 'GOOG']
>>>
```

Create an empty list and append an item to it.

```
>>> mysyms = []
>>> mysyms.append('GOOG')
>>> mysyms
['GOOG']
```

You can reassign a portion of a list to another list. For example:

```
>>> symlist[-2:] = mysyms
>>> symlist
['HPQ', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'GOOG']
>>>
```

When you do this, the list on the left-hand-side (`symlist`) will be resized as appropriate to make the right-hand-side (`mysyms`) fit. For instance, in the above example, the last two items of `symlist` got replaced by the single item in the list `mysyms`.

8.1.2 Exercise 1.20: Looping over list items

The `for` loop works by looping over data in a sequence such as a list. Check this out by typing the following loop and watching what happens:

```
>>> for s in symlist:
        print('s =', s)
# Look at the output
```

8.1.3 Exercise 1.21: Membership tests

Use the `in` or `not in` operator to check if `'AIG'`,`'AA'`, and `'CAT'` are in the list of symbols.

```
>>> # Is 'AIG' IN the `symlist`?
True
>>> # Is 'AA' IN the `symlist`?
False
>>> # Is 'CAT' NOT IN the `symlist`?
True
>>>
```

8.1.4 Exercise 1.22: Appending, inserting, and deleting items

Use the `append()` method to add the symbol `'RHT'` to end of `symlist`.

```
>>> # append 'RHT'
>>> symlist
['HPQ', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'GOOG', 'RHT']
>>>
```

Use the `insert()` method to insert the symbol `'AA'` as the second item in the list.

```
>>> # Insert 'AA' as the second item in the list
>>> symlist
['HPQ', 'AA', 'AAPL', 'AIG', 'MSFT', 'YHOO', 'GOOG', 'RHT']
>>>
```

Use the `remove()` method to remove `'MSFT'` from the list.

```
>>> # Remove 'MSFT'
>>> symlist
['HPQ', 'AA', 'AAPL', 'AIG', 'YHOO', 'GOOG', 'RHT']
>>>
```

Append a duplicate entry for `'YHOO'` at the end of the list.

Note: it is perfectly fine for a list to have duplicate values.

```
>>> # Append 'YHOO'
>>> symlist
['HPQ', 'AA', 'AAPL', 'AIG', 'YHOO', 'GOOG', 'RHT', 'YHOO']
>>>
```

Use the `index()` method to find the first position of `'YHOO'` in the list.

```
>>> # Find the first index of 'YHOO'
4
>>> symlist[4]
'YHOO'
>>>
```

Count how many times `'YHOO'` is in the list:

```
>>> symlist.count('YHOO')
2
>>>
```

Remove the first occurrence of `'YHOO'`.

```
>>> # Remove first occurrence 'YHOO'
>>> symlist
['HPQ', 'AA', 'AAPL', 'AIG', 'GOOG', 'RHT', 'YHOO']
>>>
```

Just so you know, there is no method to find or remove all occurrences of an item. However, we'll see an elegant way to do this in section 2.

8.1.5 Exercise 1.23: Sorting

Want to sort a list? Use the `sort()` method. Try it out:

```
>>> symlist.sort()
>>> symlist
['AA', 'AAPL', 'AIG', 'GOOG', 'HPQ', 'RHT', 'YHOO']
>>>
```

Want to sort in reverse? Try this:

```
>>> symlist.sort(reverse=True)
>>> symlist
['YHOO', 'RHT', 'HPQ', 'GOOG', 'AIG', 'AAPL', 'AA']
>>>
```

Note: Sorting a list modifies its contents 'in-place'. That is, the elements of the list are shuffled around, but no new list is created as a result.

8.1.6 Exercise 1.24: Putting it all back together

Want to take a list of strings and join them together into one string? Use the `join()` method of strings like this (note: this looks funny at first).

```
>>> a = ','.join(symlist)
>>> a
'YHOO,RHT,HPQ,GOOG,AIG,AAPL,AA'
>>> b = ':'.join(symlist)
>>> b
'YHOO:RHT:HPQ:GOOG:AIG:AAPL:AA'
>>> c = ''.join(symlist)
>>> c
'YHOORHTHPQGOOGAIGAAPLAA'
>>>
```

8.1.7 Exercise 1.25: Lists of anything

Lists can contain any kind of object, including other lists (e.g., nested lists). Try this out:

```
>>> nums = [101, 102, 103]
>>> items = ['spam', symlist, nums]
>>> items
['spam', ['YHOO', 'RHT', 'HPQ', 'GOOG', 'AIG', 'AAPL', 'AA'], [101, 102, 103]]
```

Pay close attention to the above output. `items` is a list with three elements. The first element is a string, but the other two elements are lists.

You can access items in the nested lists by using multiple indexing operations.

```
>>> items[0]
'spam'
>>> items[0][0]
's'
>>> items[1]
['YHOO', 'RHT', 'HPQ', 'GOOG', 'AIG', 'AAPL', 'AA']
>>> items[1][1]
'RHT'
>>> items[1][1][2]
```

```
'T'
>>> items[2]
[101, 102, 103]
>>> items[2][1]
102
>>>
```

Even though it is technically possible to make very complicated list structures, as a general rule, you want to keep things simple. Usually lists hold items that are all the same kind of value. For example, a list that consists entirely of numbers or a list of text strings. Mixing different kinds of data together in the same list is often a good way to make your head explode so it's best avoided.

Contents | Previous (1.4 Strings) | Next (1.6 Files)

Contents | Previous (1.5 Lists) | Next (1.7 Functions)

9 1.6 File Management

Most programs need to read input from somewhere. This section discusses file access.

9.0.1 File Input and Output

Open a file.

```
f = open('foo.txt', 'rt')     # Open for reading (text)
g = open('bar.txt', 'wt')     # Open for writing (text)
```

Read all of the data.

```
data = f.read()

# Read only up to 'maxbytes' bytes
data = f.read([maxbytes])
```

Write some text.

```
g.write('some text')
```

Close when you are done.

```
f.close()
g.close()
```

Files should be properly closed and it's an easy step to forget. Thus, the preferred approach is to use the `with` statement like this.

```
with open(filename, 'rt') as file:
    # Use the file `file`
    ...
    # No need to close explicitly
...statements
```

This automatically closes the file when control leaves the indented code block.

9.0.2 Common Idioms for Reading File Data

Read an entire file all at once as a string.

```
with open('foo.txt', 'rt') as file:
    data = file.read()
    # `data` is a string with all the text in `foo.txt`
```

Read a file line-by-line by iterating.

```
with open(filename, 'rt') as file:
    for line in file:
        # Process the line
```

9.0.3 Common Idioms for Writing to a File

Write string data.

```
with open('outfile', 'wt') as out:
    out.write('Hallo Python\n')
    ...
```

Redirect the print function.

```
with open('outfile', 'wt') as out:
    print('Hallo Python', file=out)
    ...
```

9.1 Exercises

These exercises depend on a file `Data/portfolio.csv`. The file contains a list of lines with information on a portfolio of stocks. It is assumed that you are working in the

`practical-python/Work/` directory. If you're not sure, you can find out where Python thinks it's running by doing this:

```
>>> import os
>>> os.getcwd()
'/Users/beazley/Desktop/practical-python/Work' # Output vary
>>>
```

9.1.1 Exercise 1.26: File Preliminaries

First, try reading the entire file all at once as a big string:

```
>>> with open('Data/portfolio.csv', 'rt') as f:
        data = f.read()

>>> data
'name,shares,price\n"AA",100,32.20\n"IBM",50,91.10\n"CAT",150,83.44\n"MSFT",200,51.23\
>>> print(data)
name,shares,price
"AA",100,32.20
"IBM",50,91.10
"CAT",150,83.44
"MSFT",200,51.23
"GE",95,40.37
"MSFT",50,65.10
"IBM",100,70.44
>>>
```

In the above example, it should be noted that Python has two modes of output. In the first mode where you type `data` at the prompt, Python shows you the raw string representation including quotes and escape codes. When you type `print(data)`, you get the actual formatted output of the string.

Although reading a file all at once is simple, it is often not the most appropriate way to do it—especially if the file happens to be huge or if contains lines of text that you want to handle one at a time.

To read a file line-by-line, use a for-loop like this:

```
>>> with open('Data/portfolio.csv', 'rt') as f:
        for line in f:
            print(line, end='')

name,shares,price
"AA",100,32.20
```

```
"IBM",50,91.10
...
>>>
```

When you use this code as shown, lines are read until the end of the file is reached at which point the loop stops.

On certain occasions, you might want to manually read or skip a *single* line of text (e.g., perhaps you want to skip the first line of column headers).

```
>>> f = open('Data/portfolio.csv', 'rt')
>>> headers = next(f)
>>> headers
'name,shares,price\n'
>>> for line in f:
    print(line, end='')

"AA",100,32.20
"IBM",50,91.10
...
>>> f.close()
>>>
```

`next()` returns the next line of text in the file. If you were to call it repeatedly, you would get successive lines. However, just so you know, the `for` loop already uses `next()` to obtain its data. Thus, you normally wouldn't call it directly unless you're trying to explicitly skip or read a single line as shown.

Once you're reading lines of a file, you can start to perform more processing such as splitting. For example, try this:

```
>>> f = open('Data/portfolio.csv', 'rt')
>>> headers = next(f).split(',')
>>> headers
['name', 'shares', 'price\n']
>>> for line in f:
    row = line.split(',')
    print(row)

['"AA"', '100', '32.20\n']
['"IBM"', '50', '91.10\n']
...
>>> f.close()
```

Note: In these examples, `f.close()` is being called explicitly because the `with` statement

isn't being used.

9.1.2 Exercise 1.27: Reading a data file

Now that you know how to read a file, let's write a program to perform a simple calculation.

The columns in `portfolio.csv` correspond to the stock name, number of shares, and purchase price of a single stock holding. Write a program called `pcost.py` that opens this file, reads all lines, and calculates how much it cost to purchase all of the shares in the portfolio.

Hint: to convert a string to an integer, use `int(s)`. To convert a string to a floating point, use `float(s)`.

Your program should print output such as the following:

```
Total cost 44671.15
```

9.1.3 Exercise 1.28: Other kinds of "files"

What if you wanted to read a non-text file such as a gzip-compressed datafile? The builtin `open()` function won't help you here, but Python has a library module `gzip` that can read gzip compressed files.

Try it:

```
>>> import gzip
>>> with gzip.open('Data/portfolio.csv.gz', 'rt') as f:
    for line in f:
        print(line, end='')

... look at the output ...
>>>
```

Note: Including the file mode of `'rt'` is critical here. If you forget that, you'll get byte strings instead of normal text strings.

9.1.4 Commentary: Shouldn't we being using Pandas for this?

Data scientists are quick to point out that libraries like Pandas already have a function for reading CSV files. This is true–and it works pretty well. However, this is not a course on learning Pandas. Reading files is a more general problem than the specifics of CSV files. The main reason we're working with a CSV file is that it's a familiar format

to most coders and it's relatively easy to work with directly–illustrating many Python features in the process. So, by all means use Pandas when you go back to work. For the rest of this course however, we're going to stick with standard Python functionality.

Contents | Previous (1.5 Lists) | Next (1.7 Functions)

Contents | Previous (1.6 Files) | Next (2.0 Working with Data)

10 1.7 Functions

As your programs start to get larger, you'll want to get organized. This section briefly introduces functions and library modules. Error handling with exceptions is also introduced.

10.0.1 Custom Functions

Use functions for code you want to reuse. Here is a function definition:

```
def sumcount(n):
    '''
    Returns the sum of the first n integers
    '''
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```

To call a function.

```
a = sumcount(100)
```

A function is a series of statements that perform some task and return a result. The `return` keyword is needed to explicitly specify the return value of the function.

10.0.2 Library Functions

Python comes with a large standard library. Library modules are accessed using `import`. For example:

```
import math
x = math.sqrt(10)
```

```
import urllib.request
u = urllib.request.urlopen('http://www.python.org/')
data = u.read()
```

We will cover libraries and modules in more detail later.

10.0.3 Errors and exceptions

Functions report errors as exceptions. An exception causes a function to abort and may cause your entire program to stop if unhandled.

Try this in your python REPL.

```
>>> int('N/A')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

For debugging purposes, the message describes what happened, where the error occurred, and a traceback showing the other function calls that led to the failure.

10.0.4 Catching and Handling Exceptions

Exceptions can be caught and handled.

To catch, use the `try - except` statement.

```
for line in f:
    fields = line.split()
    try:
        shares = int(fields[1])
    except ValueError:
        print("Couldn't parse", line)
    ...
```

The name `ValueError` must match the kind of error you are trying to catch.

It is often difficult to know exactly what kinds of errors might occur in advance depending on the operation being performed. For better or for worse, exception handling often gets added *after* a program has unexpectedly crashed (i.e., "oh, we forgot to catch that error. We should handle that!").

10.0.5 Raising Exceptions

To raise an exception, use the `raise` statement.

```
raise RuntimeError('What a kerfuffle')
```

This will cause the program to abort with an exception traceback. Unless caught by a `try-except` block.

```
% python3 foo.py
Traceback (most recent call last):
  File "foo.py", line 21, in <module>
    raise RuntimeError("What a kerfuffle")
RuntimeError: What a kerfuffle
```

10.1 Exercises

10.1.1 Exercise 1.29: Defining a function

Try defining a simple function:

```
>>> def greeting(name):
        'Issues a greeting'
        print('Hello', name)

>>> greeting('Guido')
Hello Guido
>>> greeting('Paula')
Hello Paula
>>>
```

If the first statement of a function is a string, it serves as documentation. Try typing a command such as `help(greeting)` to see it displayed.

10.1.2 Exercise 1.30: Turning a script into a function

Take the code you wrote for the `pcost.py` program in Exercise 1.27 and turn it into a function `portfolio_cost(filename)`. This function takes a filename as input, reads the portfolio data in that file, and returns the total cost of the portfolio as a float.

To use your function, change your program so that it looks something like this:

```
def portfolio_cost(filename):
    ...
```

```
    # Your code here
    ...

cost = portfolio_cost('Data/portfolio.csv')
print('Total cost:', cost)
```

When you run your program, you should see the same output as before. After you've run your program, you can also call your function interactively by typing this:

```
bash $ python3 -i pcost.py
```

This will allow you to call your function from the interactive mode.

```
>>> portfolio_cost('Data/portfolio.csv')
44671.15
>>>
```

Being able to experiment with your code interactively is useful for testing and debugging.

10.1.3 Exercise 1.31: Error handling

What happens if you try your function on a file with some missing fields?

```
>>> portfolio_cost('Data/missing.csv')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "pcost.py", line 11, in portfolio_cost
    nshares    = int(fields[1])
ValueError: invalid literal for int() with base 10: ''
>>>
```

At this point, you're faced with a decision. To make the program work you can either sanitize the original input file by eliminating bad lines or you can modify your code to handle the bad lines in some manner.

Modify the `pcost.py` program to catch the exception, print a warning message, and continue processing the rest of the file.

10.1.4 Exercise 1.32: Using a library function

Python comes with a large standard library of useful functions. One library that might be useful here is the `csv` module. You should use it whenever you have to work with CSV data files. Here is an example of how it works:

```
>>> import csv
>>> f = open('Data/portfolio.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> headers
['name', 'shares', 'price']
>>> for row in rows:
        print(row)

['AA', '100', '32.20']
['IBM', '50', '91.10']
['CAT', '150', '83.44']
['MSFT', '200', '51.23']
['GE', '95', '40.37']
['MSFT', '50', '65.10']
['IBM', '100', '70.44']
>>> f.close()
>>>
```

One nice thing about the `csv` module is that it deals with a variety of low-level details such as quoting and proper comma splitting. In the above output, you'll notice that it has stripped the double-quotes away from the names in the first column.

Modify your `pcost.py` program so that it uses the `csv` module for parsing and try running earlier examples.

10.1.5 Exercise 1.33: Reading from the command line

In the `pcost.py` program, the name of the input file has been hardwired into the code:

```
# pcost.py

def portfolio_cost(filename):
    ...
    # Your code here
    ...

cost = portfolio_cost('Data/portfolio.csv')
print('Total cost:', cost)
```

That's fine for learning and testing, but in a real program you probably wouldn't do that.

Instead, you might pass the name of the file in as an argument to a script. Try changing

the bottom part of the program as follows:

```
# pcost.py
import sys

def portfolio_cost(filename):
    ...
    # Your code here
    ...

if len(sys.argv) == 2:
    filename = sys.argv[1]
else:
    filename = 'Data/portfolio.csv'

cost = portfolio_cost(filename)
print('Total cost:', cost)
```

`sys.argv` is a list that contains passed arguments on the command line (if any).

To run your program, you'll need to run Python from the terminal.

For example, from bash on Unix:

```
bash % python3 pcost.py Data/portfolio.csv
Total cost: 44671.15
bash %
```

Contents | Previous (1.6 Files) | Next (2.0 Working with Data) Contents | Prev (1 Introduction to Python) | Next (3 Program Organization)

11 2. Working With Data

To write useful programs, you need to be able to work with data. This section introduces Python's core data structures of tuples, lists, sets, and dictionaries and discusses common data handling idioms. The last part of this section dives a little deeper into Python's underlying object model.

- 2.7 Object model

12 2.1 Datatypes and Data structures

This section introduces data structures in the form of tuples and dictionaries.

12.0.1 Primitive Datatypes

Python has a few primitive types of data:

- Integers
- Floating point numbers
- Strings (text)

We learned about these in the introduction.

12.0.2 None type

```
email_address = None
```

`None` is often used as a placeholder for optional or missing value. It evaluates as `False` in conditionals.

```
if email_address:
    send_email(email_address, msg)
```

12.0.3 Data Structures

Real programs have more complex data. For example information about a stock holding:

```
100 shares of GOOG at $490.10
```

This is an "object" with three parts:

- Name or symbol of the stock ("GOOG", a string)
- Number of shares (100, an integer)
- Price (490.10 a float)

12.0.4 Tuples

A tuple is a collection of values grouped together.

Example:

```
s = ('GOOG', 100, 490.1)
```

Sometimes the `()` are omitted in the syntax.

```
s = 'GOOG', 100, 490.1
```

Special cases (0-tuple, 1-tuple).

```
t = ()            # An empty tuple
w = ('GOOG', )    # A 1-item tuple
```

Tuples are often used to represent *simple* records or structures. Typically, it is a single *object* of multiple parts. A good analogy: *A tuple is like a single row in a database table.*

Tuple contents are ordered (like an array).

```
s = ('GOOG', 100, 490.1)
name = s[0]                 # 'GOOG'
shares = s[1]               # 100
price = s[2]                # 490.1
```

However, the contents can't be modified.

```
>>> s[1] = 75
TypeError: object does not support item assignment
```

You can, however, make a new tuple based on a current tuple.

```
s = (s[0], 75, s[2])
```

12.0.5 Tuple Packing

Tuples are more about packing related items together into a single *entity*.

```
s = ('GOOG', 100, 490.1)
```

The tuple is then easy to pass around to other parts of a program as a single object.

12.0.6 Tuple Unpacking

To use the tuple elsewhere, you can unpack its parts into variables.

```
name, shares, price = s
print('Cost', shares * price)
```

The number of variables on the left must match the tuple structure.

```
name, shares = s     # ERROR
Traceback (most recent call last):
...
ValueError: too many values to unpack
```

12.0.7 Tuples vs. Lists

Tuples look like read-only lists. However, tuples are most often used for a *single item* consisting of multiple parts. Lists are usually a collection of distinct items, usually all of the same type.

```
record = ('GOOG', 100, 490.1)       # A tuple representing a record in a portfolio

symbols = [ 'GOOG', 'AAPL', 'IBM' ]  # A List representing three stock symbols
```

12.0.8 Dictionaries

A dictionary is mapping of keys to values. It's also sometimes called a hash table or associative array. The keys serve as indices for accessing values.

```
s = {
    'name': 'GOOG',
    'shares': 100,
    'price': 490.1
}
```

12.0.9 Common operations

To get values from a dictionary use the key names.

```
>>> print(s['name'], s['shares'])
GOOG 100
>>> s['price']
490.10
>>>
```

To add or modify values assign using the key names.

```
>>> s['shares'] = 75
>>> s['date'] = '6/6/2007'
>>>
```

To delete a value use the `del` statement.

```
>>> del s['date']
>>>
```

12.0.10 Why dictionaries?

Dictionaries are useful when there are *many* different values and those values might be modified or manipulated. Dictionaries make your code more readable.

```
s['price']
# vs
s[2]
```

12.1 Exercises

In the last few exercises, you wrote a program that read a datafile `Data/portfolio.csv`. Using the `csv` module, it is easy to read the file row-by-row.

```
>>> import csv
>>> f = open('Data/portfolio.csv')
>>> rows = csv.reader(f)
>>> next(rows)
['name', 'shares', 'price']
>>> row = next(rows)
>>> row
['AA', '100', '32.20']
>>>
```

Although reading the file is easy, you often want to do more with the data than read it. For instance, perhaps you want to store it and start performing some calculations on it. Unfortunately, a raw "row" of data doesn't give you enough to work with. For example, even a simple math calculation doesn't work:

```
>>> row = ['AA', '100', '32.20']
>>> cost = row[1] * row[2]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
TypeError: can't multiply sequence by non-int of type 'str'
>>>
```

To do more, you typically want to interpret the raw data in some way and turn it into a more useful kind of object so that you can work with it later. Two simple options are tuples or dictionaries.

12.1.1 Exercise 2.1: Tuples

At the interactive prompt, create the following tuple that represents the above row, but with the numeric columns converted to proper numbers:

```
>>> t = (row[0], int(row[1]), float(row[2]))
>>> t
('AA', 100, 32.2)
>>>
```

Using this, you can now calculate the total cost by multiplying the shares and the price:

```
>>> cost = t[1] * t[2]
>>> cost
3220.0000000000005
>>>
```

Is math broken in Python? What's the deal with the answer of 3220.0000000000005?

This is an artifact of the floating point hardware on your computer only being able to accurately represent decimals in Base-2, not Base-10. For even simple calculations involving base-10 decimals, small errors are introduced. This is normal, although perhaps a bit surprising if you haven't seen it before.

This happens in all programming languages that use floating point decimals, but it often gets hidden when printing. For example:

```
>>> print(f'{cost:0.2f}')
3220.00
>>>
```

Tuples are read-only. Verify this by trying to change the number of shares to 75.

```
>>> t[1] = 75
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Although you can't change tuple contents, you can always create a completely new tuple that replaces the old one.

```
>>> t = (t[0], 75, t[2])
>>> t
('AA', 75, 32.2)
>>>
```

Whenever you reassign an existing variable name like this, the old value is discarded. Although the above assignment might look like you are modifying the tuple, you are actually creating a new tuple and throwing the old one away.

Tuples are often used to pack and unpack values into variables. Try the following:

```
>>> name, shares, price = t
>>> name
'AA'
>>> shares
75
>>> price
32.2
>>>
```

Take the above variables and pack them back into a tuple

```
>>> t = (name, 2*shares, price)
>>> t
('AA', 150, 32.2)
>>>
```

12.1.2 Exercise 2.2: Dictionaries as a data structure

An alternative to a tuple is to create a dictionary instead.

```
>>> d = {
        'name' : row[0],
        'shares' : int(row[1]),
        'price'  : float(row[2])
    }
>>> d
{'name': 'AA', 'shares': 100, 'price': 32.2 }
>>>
```

Calculate the total cost of this holding:

```
>>> cost = d['shares'] * d['price']
>>> cost
3220.0000000000005
>>>
```

Compare this example with the same calculation involving tuples above. Change the number of shares to 75.

```
>>> d['shares'] = 75
>>> d
{'name': 'AA', 'shares': 75, 'price': 75}
>>>
```

Unlike tuples, dictionaries can be freely modified. Add some attributes:

```
>>> d['date'] = (6, 11, 2007)
>>> d['account'] = 12345
>>> d
{'name': 'AA', 'shares': 75, 'price':32.2, 'date': (6, 11, 2007), 'account': 12345}
>>>
```

12.1.3 Exercise 2.3: Some additional dictionary operations

If you turn a dictionary into a list, you'll get all of its keys:

```
>>> list(d)
['name', 'shares', 'price', 'date', 'account']
>>>
```

Similarly, if you use the `for` statement to iterate on a dictionary, you will get the keys:

```
>>> for k in d:
        print('k =', k)

k = name
k = shares
k = price
k = date
k = account
>>>
```

Try this variant that performs a lookup at the same time:

```
>>> for k in d:
        print(k, '=', d[k])
```

```
name = AA
shares = 75
price = 32.2
date = (6, 11, 2007)
account = 12345
>>>
```

You can also obtain all of the keys using the `keys()` method:

```
>>> keys = d.keys()
>>> keys
dict_keys(['name', 'shares', 'price', 'date', 'account'])
>>>
```

`keys()` is a bit unusual in that it returns a special `dict_keys` object.

This is an overlay on the original dictionary that always gives you the current keys—even if the dictionary changes. For example, try this:

```
>>> del d['account']
>>> keys
dict_keys(['name', 'shares', 'price', 'date'])
>>>
```

Carefully notice that the `'account'` disappeared from `keys` even though you didn't call `d.keys()` again.

A more elegant way to work with keys and values together is to use the `items()` method. This gives you `(key, value)` tuples:

```
>>> items = d.items()
>>> items
dict_items([('name', 'AA'), ('shares', 75), ('price', 32.2), ('date', (6, 11, 2007))])
>>> for k, v in d.items():
        print(k, '=', v)

name = AA
shares = 75
price = 32.2
date = (6, 11, 2007)
>>>
```

If you have tuples such as `items`, you can create a dictionary using the `dict()` function. Try it:

```
>>> items
dict_items([('name', 'AA'), ('shares', 75), ('price', 32.2), ('date', (6, 11, 2007))])
```

```
>>> d = dict(items)
>>> d
{'name': 'AA', 'shares': 75, 'price':32.2, 'date': (6, 11, 2007)}
>>>
```

Contents | Previous (1.6 Files) | Next (2.2 Containers)

Contents | Previous (2.1 Datatypes) | Next (2.3 Formatting)

13 2.2 Containers

This section discusses lists, dictionaries, and sets.

13.0.1 Overview

Programs often have to work with many objects.

- A portfolio of stocks
- A table of stock prices

There are three main choices to use.

- Lists. Ordered data.
- Dictionaries. Unordered data.
- Sets. Unordered collection of unique items.

13.0.2 Lists as a Container

Use a list when the order of the data matters. Remember that lists can hold any kind of object. For example, a list of tuples.

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.3),
    ('CAT', 150, 83.44)
]

portfolio[0]            # ('GOOG', 100, 490.1)
portfolio[2]            # ('CAT', 150, 83.44)
```

13.0.3 List construction

Building a list from scratch.

```
records = []  # Initial empty list

# Use .append() to add more items
records.append(('GOOG', 100, 490.10))
records.append(('IBM', 50, 91.3))
...
```

An example when reading records from a file.

```
records = []  # Initial empty list

with open('Data/portfolio.csv', 'rt') as f:
    next(f) # Skip header
    for line in f:
        row = line.split(',')
        records.append((row[0], int(row[1]), float(row[2])))
```

13.0.4 Dicts as a Container

Dictionaries are useful if you want fast random lookups (by key name). For example, a dictionary of stock prices:

```
prices = {
   'GOOG': 513.25,
   'CAT': 87.22,
   'IBM': 93.37,
   'MSFT': 44.12
}
```

Here are some simple lookups:

```
>>> prices['IBM']
93.37
>>> prices['GOOG']
513.25
>>>
```

13.0.5 Dict Construction

Example of building a dict from scratch.

```
prices = {} # Initial empty dict

# Insert new items
prices['GOOG'] = 513.25
prices['CAT'] = 87.22
prices['IBM'] = 93.37
```

An example populating the dict from the contents of a file.

```
prices = {} # Initial empty dict

with open('Data/prices.csv', 'rt') as f:
    for line in f:
        row = line.split(',')
        prices[row[0]] = float(row[1])
```

Note: If you try this on the `Data/prices.csv` file, you'll find that it almost works–there's a blank line at the end that causes it to crash. You'll need to figure out some way to modify the code to account for that (see Exercise 2.6).

13.0.6 Dictionary Lookups

You can test the existence of a key.

```
if key in d:
    # YES
else:
    # NO
```

You can look up a value that might not exist and provide a default value in case it doesn't.

```
name = d.get(key, default)
```

An example:

```
>>> prices.get('IBM', 0.0)
93.37
>>> prices.get('SCOX', 0.0)
0.0
>>>
```

13.0.7 Composite keys

Almost any type of value can be used as a dictionary key in Python. A dictionary key must be of a type that is immutable. For example, tuples:

```
holidays = {
  (1, 1) : 'New Years',
  (3, 14) : 'Pi day',
  (9, 13) : "Programmer's day",
}
```

Then to access:

```
>>> holidays[3, 14]
'Pi day'
>>>
```

Neither a list, a set, nor another dictionary can serve as a dictionary key, because lists and dictionaries are mutable.

13.0.8 Sets

Sets are collection of unordered unique items.

```
tech_stocks = { 'IBM','AAPL','MSFT' }
# Alternative syntax
tech_stocks = set(['IBM', 'AAPL', 'MSFT'])
```

Sets are useful for membership tests.

```
>>> tech_stocks
set(['AAPL', 'IBM', 'MSFT'])
>>> 'IBM' in tech_stocks
True
>>> 'FB' in tech_stocks
False
>>>
```

Sets are also useful for duplicate elimination.

```
names = ['IBM', 'AAPL', 'GOOG', 'IBM', 'GOOG', 'YHOO']

unique = set(names)
# unique = set(['IBM', 'AAPL','GOOG','YHOO'])
```

Additional set operations:

```
names.add('CAT')          # Add an item
names.remove('YHOO')      # Remove an item

s1 | s2                   # Set union
s1 & s2                   # Set intersection
s1 - s2                   # Set difference
```

13.1 Exercises

In these exercises, you start building one of the major programs used for the rest of this course. Do your work in the file `Work/report.py`.

13.1.1 Exercise 2.4: A list of tuples

The file `Data/portfolio.csv` contains a list of stocks in a portfolio. In Exercise 1.30, you wrote a function `portfolio_cost(filename)` that read this file and performed a simple calculation.

Your code should have looked something like this:

```
# pcost.py

import csv

def portfolio_cost(filename):
    '''Computes the total cost (shares*price) of a portfolio file'''
    total_cost = 0.0

    with open(filename, 'rt') as f:
        rows = csv.reader(f)
        headers = next(rows)
        for row in rows:
            nshares = int(row[1])
            price = float(row[2])
            total_cost += nshares * price
    return total_cost
```

Using this code as a rough guide, create a new file `report.py`. In that file, define a function `read_portfolio(filename)` that opens a given portfolio file and reads it into a list of tuples. To do this, you're going to make a few minor modifications to the above code.

First, instead of defining `total_cost = 0`, you'll make a variable that's initially set to an empty list. For example:

```
portfolio = []
```

Next, instead of totaling up the cost, you'll turn each row into a tuple exactly as you just did in the last exercise and append it to this list. For example:

```
for row in rows:
    holding = (row[0], int(row[1]), float(row[2]))
    portfolio.append(holding)
```

Finally, you'll return the resulting `portfolio` list.

Experiment with your function interactively (just a reminder that in order to do this, you first have to run the `report.py` program in the interpreter):

Hint: Use `-i` when executing the file in the terminal

```
>>> portfolio = read_portfolio('Data/portfolio.csv')
>>> portfolio
[('AA', 100, 32.2), ('IBM', 50, 91.1), ('CAT', 150, 83.44), ('MSFT', 200, 51.23),
    ('GE', 95, 40.37), ('MSFT', 50, 65.1), ('IBM', 100, 70.44)]
>>>
>>> portfolio[0]
('AA', 100, 32.2)
>>> portfolio[1]
('IBM', 50, 91.1)
>>> portfolio[1][1]
50
>>> total = 0.0
>>> for s in portfolio:
        total += s[1] * s[2]

>>> print(total)
44671.15
>>>
```

This list of tuples that you have created is very similar to a 2-D array. For example, you can access a specific column and row using a lookup such as `portfolio[row][column]` where `row` and `column` are integers.

That said, you can also rewrite the last for-loop using a statement like this:

```
>>> total = 0.0
>>> for name, shares, price in portfolio:
            total += shares*price
```

```
>>> print(total)
44671.15
>>>
```

13.1.2 Exercise 2.5: List of Dictionaries

Take the function you wrote in Exercise 2.4 and modify to represent each stock in the portfolio with a dictionary instead of a tuple. In this dictionary use the field names of "name", "shares", and "price" to represent the different columns in the input file.

Experiment with this new function in the same manner as you did in Exercise 2.4.

```
>>> portfolio = read_portfolio('Data/portfolio.csv')
>>> portfolio
[{'name': 'AA', 'shares': 100, 'price': 32.2}, {'name': 'IBM', 'shares': 50, 'price':
    {'name': 'CAT', 'shares': 150, 'price': 83.44}, {'name': 'MSFT', 'shares': 200, 'p
    {'name': 'GE', 'shares': 95, 'price': 40.37}, {'name': 'MSFT', 'shares': 50, 'pric
    {'name': 'IBM', 'shares': 100, 'price': 70.44}]
>>> portfolio[0]
{'name': 'AA', 'shares': 100, 'price': 32.2}
>>> portfolio[1]
{'name': 'IBM', 'shares': 50, 'price': 91.1}
>>> portfolio[1]['shares']
50
>>> total = 0.0
>>> for s in portfolio:
        total += s['shares']*s['price']

>>> print(total)
44671.15
>>>
```

Here, you will notice that the different fields for each entry are accessed by key names instead of numeric column numbers. This is often preferred because the resulting code is easier to read later.

Viewing large dictionaries and lists can be messy. To clean up the output for debugging, consider using the `pprint` function.

```
>>> from pprint import pprint
>>> pprint(portfolio)
[{'name': 'AA', 'price': 32.2, 'shares': 100},
    {'name': 'IBM', 'price': 91.1, 'shares': 50},
```

```
    {'name': 'CAT', 'price': 83.44, 'shares': 150},
    {'name': 'MSFT', 'price': 51.23, 'shares': 200},
    {'name': 'GE', 'price': 40.37, 'shares': 95},
    {'name': 'MSFT', 'price': 65.1, 'shares': 50},
    {'name': 'IBM', 'price': 70.44, 'shares': 100}]
>>>
```

13.1.3 Exercise 2.6: Dictionaries as a container

A dictionary is a useful way to keep track of items where you want to look up items using an index other than an integer. In the Python shell, try playing with a dictionary:

```
>>> prices = { }
>>> prices['IBM'] = 92.45
>>> prices['MSFT'] = 45.12
>>> prices
... look at the result ...
>>> prices['IBM']
92.45
>>> prices['AAPL']
... look at the result ...
>>> 'AAPL' in prices
False
>>>
```

The file `Data/prices.csv` contains a series of lines with stock prices. The file looks something like this:

```
"AA",9.22
"AXP",24.85
"BA",44.85
"BAC",11.27
"C",3.72
...
```

Write a function `read_prices(filename)` that reads a set of prices such as this into a dictionary where the keys of the dictionary are the stock names and the values in the dictionary are the stock prices.

To do this, start with an empty dictionary and start inserting values into it just as you did above. However, you are reading the values from a file now.

We'll use this data structure to quickly lookup the price of a given stock name.

A few little tips that you'll need for this part. First, make sure you use the `csv` module just as you did before—there's no need to reinvent the wheel here.

```
>>> import csv
>>> f = open('Data/prices.csv', 'r')
>>> rows = csv.reader(f)
>>> for row in rows:
        print(row)


['AA', '9.22']
['AXP', '24.85']
...
[]
>>>
```

The other little complication is that the `Data/prices.csv` file may have some blank lines in it. Notice how the last row of data above is an empty list—meaning no data was present on that line.

There's a possibility that this could cause your program to die with an exception. Use the `try` and `except` statements to catch this as appropriate. Thought: would it be better to guard against bad data with an `if`-statement instead?

Once you have written your `read_prices()` function, test it interactively to make sure it works:

```
>>> prices = read_prices('Data/prices.csv')
>>> prices['IBM']
106.28
>>> prices['MSFT']
20.89
>>>
```

13.1.4 Exercise 2.7: Finding out if you can retire

Tie all of this work together by adding a few additional statements to your `report.py` program that computes gain/loss. These statements should take the list of stocks in Exercise 2.5 and the dictionary of prices in Exercise 2.6 and compute the current value of the portfolio along with the gain/loss.

Contents | Previous (2.1 Datatypes) | Next (2.3 Formatting)

Contents | Previous (2.2 Containers) | Next (2.4 Sequences)

14 2.3 Formatting

This section is a slight digression, but when you work with data, you often want to produce structured output (tables, etc.). For example:

```
      Name      Shares        Price
----------  ----------  -----------
        AA         100        32.20
       IBM          50        91.10
       CAT         150        83.44
      MSFT         200        51.23
        GE          95        40.37
      MSFT          50        65.10
       IBM         100        70.44
```

14.0.1 String Formatting

One way to format string in Python 3.6+ is with `f-strings`.

```
>>> name = 'IBM'
>>> shares = 100
>>> price = 91.1
>>> f'{name:>10s} {shares:>10d} {price:>10.2f}'
'       IBM        100      91.10'
>>>
```

The part `{expression:format}` is replaced.

It is commonly used with `print`.

```
print(f'{name:>10s} {shares:>10d} {price:>10.2f}')
```

14.0.2 Format codes

Format codes (after the `:` inside the `{}`) are similar to C `printf()`. Common codes include:

```
d       Decimal integer
b       Binary integer
x       Hexadecimal integer
f       Float as [-]m.dddddd
e       Float as [-]m.dddddde+-xx
g       Float, but selective use of E notation s String
c       Character (from integer)
```

Common modifiers adjust the field width and decimal precision. This is a partial list:

```
:>10d   Integer right aligned in 10-character field
:<10d   Integer left aligned in 10-character field
:^10d   Integer centered in 10-character field
:0.2f   Float with 2 digit precision
```

14.0.3 Dictionary Formatting

You can use the `format_map()` method to apply string formatting to a dictionary of values:

```
>>> s = {
    'name': 'IBM',
    'shares': 100,
    'price': 91.1
}
>>> '{name:>10s} {shares:10d} {price:10.2f}'.format_map(s)
'       IBM        100      91.10'
>>>
```

It uses the same codes as `f-strings` but takes the values from the supplied dictionary.

14.0.4 format() method

There is a method `format()` that can apply formatting to arguments or keyword arguments.

```
>>> '{name:>10s} {shares:10d} {price:10.2f}'.format(name='IBM', shares=100, price=91.1
'       IBM        100      91.10'
>>> '{:10s} {:10d} {:10.2f}'.format('IBM', 100, 91.1)
'       IBM        100      91.10'
>>>
```

Frankly, `format()` is a bit verbose. I prefer f-strings.

14.0.5 C-Style Formatting

You can also use the formatting operator `%`.

```
>>> 'The value is %d' % 3
'The value is 3'
>>> '%5d %-5d %10d' % (3,4,5)
```

```
'    3 4              5'
>>> '%0.2f' % (3.1415926,)
'3.14'
```

This requires a single item or a tuple on the right. Format codes are modeled after the C `printf()` as well.

Note: This is the only formatting available on byte strings.

```
>>> b'%s has %n messages' % (b'Dave', 37)
b'Dave has 37 messages'
>>>
```

14.1 Exercises

14.1.1 Exercise 2.8: How to format numbers

A common problem with printing numbers is specifying the number of decimal places. One way to fix this is to use f-strings. Try these examples:

```
>>> value = 42863.1
>>> print(value)
42863.1
>>> print(f'{value:0.4f}')
42863.1000
>>> print(f'{value:>16.2f}')
        42863.10
>>> print(f'{value:<16.2f}')
42863.10
>>> print(f'{value:*>16,.2f}')
*******42,863.10
>>>
```

Full documentation on the formatting codes used f-strings can be found here. Formatting is also sometimes performed using the `%` operator of strings.

```
>>> print('%0.4f' % value)
42863.1000
>>> print('%16.2f' % value)
        42863.10
>>>
```

Documentation on various codes used with `%` can be found here.

Although it's commonly used with `print`, string formatting is not tied to printing. If you want to save a formatted string. Just assign it to a variable.

```
>>> f = '%0.4f' % value
>>> f
'42863.1000'
>>>
```

14.1.2 Exercise 2.9: Collecting Data

In Exercise 2.7, you wrote a program called `report.py` that computed the gain/loss of a stock portfolio. In this exercise, you're going to start modifying it to produce a table like this:

```
      Name     Shares      Price     Change
---------- ---------- ---------- ----------
        AA        100       9.22     -22.98
       IBM         50     106.28      15.18
       CAT        150      35.46     -47.98
      MSFT        200      20.89     -30.34
        GE         95      13.48     -26.89
      MSFT         50      20.89     -44.21
       IBM        100     106.28      35.84
```

In this report, "Price" is the current share price of the stock and "Change" is the change in the share price from the initial purchase price.

In order to generate the above report, you'll first want to collect all of the data shown in the table. Write a function `make_report()` that takes a list of stocks and dictionary of prices as input and returns a list of tuples containing the rows of the above table.

Add this function to your `report.py` file. Here's how it should work if you try it interactively:

```
>>> portfolio = read_portfolio('Data/portfolio.csv')
>>> prices = read_prices('Data/prices.csv')
>>> report = make_report(portfolio, prices)
>>> for r in report:
        print(r)

('AA', 100, 9.22, -22.980000000000004)
('IBM', 50, 106.28, 15.180000000000007)
('CAT', 150, 35.46, -47.98)
('MSFT', 200, 20.89, -30.339999999999996)
('GE', 95, 13.48, -26.889999999999997)
```

```
...
>>>
```

14.1.3 Exercise 2.10: Printing a formatted table

Redo the for-loop in Exercise 2.9, but change the print statement to format the tuples.

```
>>> for r in report:
        print('%10s %10d %10.2f %10.2f' % r)

        AA        100       9.22     -22.98
       IBM         50     106.28      15.18
       CAT        150      35.46     -47.98
      MSFT        200      20.89     -30.34
...
>>>
```

You can also expand the values and use f-strings. For example:

```
>>> for name, shares, price, change in report:
        print(f'{name:>10s} {shares:>10d} {price:>10.2f} {change:>10.2f}')

        AA        100       9.22     -22.98
       IBM         50     106.28      15.18
       CAT        150      35.46     -47.98
      MSFT        200      20.89     -30.34
...
>>>
```

Take the above statements and add them to your `report.py` program. Have your program take the output of the `make_report()` function and print a nicely formatted table as shown.

14.1.4 Exercise 2.11: Adding some headers

Suppose you had a tuple of header names like this:

```
headers = ('Name', 'Shares', 'Price', 'Change')
```

Add code to your program that takes the above tuple of headers and creates a string where each header name is right-aligned in a 10-character wide field and each field is separated by a single space.

```
'      Name      Shares      Price      Change'
```

Write code that takes the headers and creates the separator string between the headers and data to follow. This string is just a bunch of "-" characters under each field name. For example:

```
'---------- ---------- ---------- ----------'
```

When you're done, your program should produce the table shown at the top of this exercise.

```
      Name      Shares      Price      Change
---------- ---------- ---------- ----------
        AA         100       9.22     -22.98
       IBM          50     106.28      15.18
       CAT         150      35.46     -47.98
      MSFT         200      20.89     -30.34
        GE          95      13.48     -26.89
      MSFT          50      20.89     -44.21
       IBM         100     106.28      35.84
```

14.1.5 Exercise 2.12: Formatting Challenge

How would you modify your code so that the price includes the currency symbol ($) and the output looks like this:

```
      Name      Shares      Price      Change
---------- ---------- ---------- ----------
        AA         100      $9.22     -22.98
       IBM          50    $106.28      15.18
       CAT         150     $35.46     -47.98
      MSFT         200     $20.89     -30.34
        GE          95     $13.48     -26.89
      MSFT          50     $20.89     -44.21
       IBM         100    $106.28      35.84
```

Contents | Previous (2.2 Containers) | Next (2.4 Sequences)

Contents | Previous (2.3 Formatting) | Next (2.5 Collections)

15 2.4 Sequences

15.0.1 Sequence Datatypes

Python has three *sequence* datatypes.

- String: `'Hello'`. A string is a sequence of characters.
- List: `[1, 4, 5]`.
- Tuple: `('GOOG', 100, 490.1)`.

All sequences are ordered, indexed by integers, and have a length.

```
a = 'Hello'                 # String
b = [1, 4, 5]               # List
c = ('GOOG', 100, 490.1)    # Tuple

# Indexed order
a[0]                        # 'H'
b[-1]                       # 5
c[1]                        # 100

# Length of sequence
len(a)                      # 5
len(b)                      # 3
len(c)                      # 3
```

Sequences can be replicated: `s * n`.

```
>>> a = 'Hello'
>>> a * 3
'HelloHelloHello'
>>> b = [1, 2, 3]
>>> b * 2
[1, 2, 3, 1, 2, 3]
>>>
```

Sequences of the same type can be concatenated: `s + t`.

```
>>> a = (1, 2, 3)
>>> b = (4, 5)
>>> a + b
(1, 2, 3, 4, 5)
>>>
>>> c = [1, 5]
>>> a + c
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
```

15.0.2 Slicing

Slicing means to take a subsequence from a sequence. The syntax is `s[start:end]`. Where `start` and `end` are the indexes of the subsequence you want.

```
a = [0,1,2,3,4,5,6,7,8]

a[2:5]    # [2,3,4]
a[-5:]    # [4,5,6,7,8]
a[:3]     # [0,1,2]
```

- Indices `start` and `end` must be integers.
- Slices do *not* include the end value. It is like a half-open interval from math.
- If indices are omitted, they default to the beginning or end of the list.

15.0.3 Slice re-assignment

On lists, slices can be reassigned and deleted.

```
# Reassignment
a = [0,1,2,3,4,5,6,7,8]
a[2:4] = [10,11,12]       # [0,1,10,11,12,4,5,6,7,8]
```

Note: The reassigned slice doesn't need to have the same length.

```
# Deletion
a = [0,1,2,3,4,5,6,7,8]
del a[2:4]                # [0,1,4,5,6,7,8]
```

15.0.4 Sequence Reductions

There are some common functions to reduce a sequence to a single value.

```
>>> s = [1, 2, 3, 4]
>>> sum(s)
10
>>> min(s)
1
>>> max(s)
4
```

```
>>> t = ['Hello', 'World']
>>> max(t)
'World'
>>>
```

15.0.5 Iteration over a sequence

The for-loop iterates over the elements in a sequence.

```
>>> s = [1, 4, 9, 16]
>>> for i in s:
...     print(i)
...
1
4
9
16
>>>
```

On each iteration of the loop, you get a new item to work with. This new value is placed into the iteration variable. In this example, the iteration variable is `x`:

```
for x in s:          # `x` is an iteration variable
    ...statements
```

On each iteration, the previous value of the iteration variable is overwritten (if any). After the loop finishes, the variable retains the last value.

15.0.6 break statement

You can use the `break` statement to break out of a loop early.

```
for name in namelist:
    if name == 'Jake':
        break
    ...
    ...
statements
```

When the `break` statement executes, it exits the loop and moves on the next `statements`. The `break` statement only applies to the inner-most loop. If this loop is within another loop, it will not break the outer loop.

15.0.7 continue statement

To skip one element and move to the next one, use the `continue` statement.

```
for line in lines:
    if line == '\n':    # Skip blank lines
        continue
    # More statements
    ...
```

This is useful when the current item is not of interest or needs to be ignored in the processing.

15.0.8 Looping over integers

If you need to count, use `range()`.

```
for i in range(100):
    # i = 0,1,...,99
```

The syntax is `range([start,] end [,step])`

```
for i in range(100):
    # i = 0,1,...,99
for j in range(10,20):
    # j = 10,11,..., 19
for k in range(10,50,2):
    # k = 10,12,...,48
    # Notice how it counts in steps of 2, not 1.
```

- The ending value is never included. It mirrors the behavior of slices.
- `start` is optional. Default `0`.
- `step` is optional. Default `1`.
- `range()` computes values as needed. It does not actually store a large range of numbers.

15.0.9 enumerate() function

The `enumerate` function adds an extra counter value to iteration.

```
names = ['Elwood', 'Jake', 'Curtis']
for i, name in enumerate(names):
    # Loops with i = 0, name = 'Elwood'
```

```
    # i = 1, name = 'Jake'
    # i = 2, name = 'Curtis'
```

The general form is `enumerate(sequence [, start = 0])`. `start` is optional. A good example of using `enumerate()` is tracking line numbers while reading a file:

```
with open(filename) as f:
    for lineno, line in enumerate(f, start=1):
        ...
```

In the end, `enumerate` is just a nice shortcut for:

```
i = 0
for x in s:
    statements
    i += 1
```

Using `enumerate` is less typing and runs slightly faster.

15.0.10 For and tuples

You can iterate with multiple iteration variables.

```
points = [
  (1, 4),(10, 40),(23, 14),(5, 6),(7, 8)
]
for x, y in points:
    # Loops with x = 1, y = 4
    #            x = 10, y = 40
    #            x = 23, y = 14
    #            ...
```

When using multiple variables, each tuple is *unpacked* into a set of iteration variables. The number of variables must match the of items in each tuple.

15.0.11 zip() function

The `zip` function takes multiple sequences and makes an iterator that combines them.

```
columns = ['name', 'shares', 'price']
values = ['GOOG', 100, 490.1 ]
pairs = zip(columns, values)
# ('name','GOOG'), ('shares',100), ('price',490.1)
```

To get the result you must iterate. You can use multiple variables to unpack the tuples as shown earlier.

```
for column, value in pairs:
    ...
```

A common use of `zip` is to create key/value pairs for constructing dictionaries.

```
d = dict(zip(columns, values))
```

15.1 Exercises

15.1.1 Exercise 2.13: Counting

Try some basic counting examples:

```
>>> for n in range(10):             # Count 0 ... 9
        print(n, end=' ')

0 1 2 3 4 5 6 7 8 9
>>> for n in range(10,0,-1):        # Count 10 ... 1
        print(n, end=' ')

10 9 8 7 6 5 4 3 2 1
>>> for n in range(0,10,2):         # Count 0, 2, ... 8
        print(n, end=' ')

0 2 4 6 8
>>>
```

15.1.2 Exercise 2.14: More sequence operations

Interactively experiment with some of the sequence reduction operations.

```
>>> data = [4, 9, 1, 25, 16, 100, 49]
>>> min(data)
1
>>> max(data)
100
>>> sum(data)
204
>>>
```

Try looping over the data.

```
>>> for x in data:
        print(x)

4
9
...
>>> for n, x in enumerate(data):
        print(n, x)

0 4
1 9
2 1
...
>>>
```

Sometimes the `for` statement, `len()`, and `range()` get used by novices in some kind of horrible code fragment that looks like it emerged from the depths of a rusty C program.

```
>>> for n in range(len(data)):
        print(data[n])

4
9
1
...
>>>
```

Don't do that! Not only does reading it make everyone's eyes bleed, it's inefficient with memory and it runs a lot slower. Just use a normal `for` loop if you want to iterate over data. Use `enumerate()` if you happen to need the index for some reason.

15.1.3 Exercise 2.15: A practical enumerate() example

Recall that the file `Data/missing.csv` contains data for a stock portfolio, but has some rows with missing data. Using `enumerate()`, modify your `pcost.py` program so that it prints a line number with the warning message when it encounters bad input.

```
>>> cost = portfolio_cost('Data/missing.csv')
Row 4: Couldn't convert: ['MSFT', '', '51.23']
Row 7: Couldn't convert: ['IBM', '', '70.44']
>>>
```

To do this, you'll need to change a few parts of your code.

```
...
for rowno, row in enumerate(rows, start=1):
    try:
        ...
    except ValueError:
        print(f'Row {rowno}: Bad row: {row}')
```

15.1.4 Exercise 2.16: Using the zip() function

In the file `Data/portfolio.csv`, the first line contains column headers. In all previous code, we've been discarding them.

```
>>> f = open('Data/portfolio.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> headers
['name', 'shares', 'price']
>>>
```

However, what if you could use the headers for something useful? This is where the `zip()` function enters the picture. First try this to pair the file headers with a row of data:

```
>>> row = next(rows)
>>> row
['AA', '100', '32.20']
>>> list(zip(headers, row))
[ ('name', 'AA'), ('shares', '100'), ('price', '32.20') ]
>>>
```

Notice how `zip()` paired the column headers with the column values. We've used `list()` here to turn the result into a list so that you can see it. Normally, `zip()` creates an iterator that must be consumed by a for-loop.

This pairing is an intermediate step to building a dictionary. Now try this:

```
>>> record = dict(zip(headers, row))
>>> record
{'price': '32.20', 'name': 'AA', 'shares': '100'}
>>>
```

This transformation is one of the most useful tricks to know about when processing a lot of data files. For example, suppose you wanted to make the `pcost.py` program work with various input files, but without regard for the actual column number where the name, shares, and price appear.

Modify the `portfolio_cost()` function in `pcost.py` so that it looks like this:

```
# pcost.py

def portfolio_cost(filename):
    ...
        for rowno, row in enumerate(rows, start=1):
            record = dict(zip(headers, row))
            try:
                nshares = int(record['shares'])
                price = float(record['price'])
                total_cost += nshares * price
            # This catches errors in int() and float() conversions above
            except ValueError:
                print(f'Row {rowno}: Bad row: {row}')
        ...
```

Now, try your function on a completely different data file `Data/portfoliodate.csv` which looks like this:

```
name,date,time,shares,price
"AA","6/11/2007","9:50am",100,32.20
"IBM","5/13/2007","4:20pm",50,91.10
"CAT","9/23/2006","1:30pm",150,83.44
"MSFT","5/17/2007","10:30am",200,51.23
"GE","2/1/2006","10:45am",95,40.37
"MSFT","10/31/2006","12:05pm",50,65.10
"IBM","7/9/2006","3:15pm",100,70.44
```

```
>>> portfolio_cost('Data/portfoliodate.csv')
44671.15
>>>
```

If you did it right, you'll find that your program still works even though the data file has a completely different column format than before. That's cool!

The change made here is subtle, but significant. Instead of `portfolio_cost()` being hardcoded to read a single fixed file format, the new version reads any CSV file and picks the values of interest out of it. As long as the file has the required columns, the code will work.

Modify the `report.py` program you wrote in Section 2.3 so that it uses the same technique to pick out column headers.

Try running the `report.py` program on the `Data/portfoliodate.csv` file and see that it produces the same answer as before.

15.1.5 Exercise 2.17: Inverting a dictionary

A dictionary maps keys to values. For example, a dictionary of stock prices.

```
>>> prices = {
        'GOOG' : 490.1,
        'AA' : 23.45,
        'IBM' : 91.1,
        'MSFT' : 34.23
    }
>>>
```

If you use the `items()` method, you can get `(key,value)` pairs:

```
>>> prices.items()
dict_items([('GOOG', 490.1), ('AA', 23.45), ('IBM', 91.1), ('MSFT', 34.23)])
>>>
```

However, what if you wanted to get a list of `(value, key)` pairs instead? *Hint: use* ***`zip()`***.

```
>>> pricelist = list(zip(prices.values(),prices.keys()))
>>> pricelist
[(490.1, 'GOOG'), (23.45, 'AA'), (91.1, 'IBM'), (34.23, 'MSFT')]
>>>
```

Why would you do this? For one, it allows you to perform certain kinds of data processing on the dictionary data.

```
>>> min(pricelist)
(23.45, 'AA')
>>> max(pricelist)
(490.1, 'GOOG')
>>> sorted(pricelist)
[(23.45, 'AA'), (34.23, 'MSFT'), (91.1, 'IBM'), (490.1, 'GOOG')]
>>>
```

This also illustrates an important feature of tuples. When used in comparisons, tuples are compared element-by-element starting with the first item. Similar to how strings are compared character-by-character.

`zip()` is often used in situations like this where you need to pair up data from different places. For example, pairing up the column names with column values in order to make a dictionary of named values.

Note that `zip()` is not limited to pairs. For example, you can use it with any number of input lists:

```
>>> a = [1, 2, 3, 4]
>>> b = ['w', 'x', 'y', 'z']
>>> c = [0.2, 0.4, 0.6, 0.8]
>>> list(zip(a, b, c))
[(1, 'w', 0.2), (2, 'x', 0.4), (3, 'y', 0.6), (4, 'z', 0.8))]
>>>
```

Also, be aware that `zip()` stops once the shortest input sequence is exhausted.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = ['x', 'y', 'z']
>>> list(zip(a,b))
[(1, 'x'), (2, 'y'), (3, 'z')]
>>>
```

Contents | Previous (2.3 Formatting) | Next (2.5 Collections)

Contents | Previous (2.4 Sequences) | Next (2.6 List Comprehensions)

16 2.5 collections module

The `collections` module provides a number of useful objects for data handling. This part briefly introduces some of these features.

16.0.1 Example: Counting Things

Let's say you want to tabulate the total shares of each stock.

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]
```

There are two `IBM` entries and two `GOOG` entries in this list. The shares need to be combined together somehow.

16.0.2 Counters

Solution: Use a `Counter`.

```
from collections import Counter
total_shares = Counter()
for name, shares, price in portfolio:
    total_shares[name] += shares

total_shares['IBM']     # 150
```

16.0.3 Example: One-Many Mappings

Problem: You want to map a key to multiple values.

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]
```

Like in the previous example, the key `IBM` should have two different tuples instead.

Solution: Use a `defaultdict`.

```
from collections import defaultdict
holdings = defaultdict(list)
for name, shares, price in portfolio:
    holdings[name].append((shares, price))
holdings['IBM'] # [ (50, 91.1), (100, 45.23) ]
```

The `defaultdict` ensures that every time you access a key you get a default value.

16.0.4 Example: Keeping a History

Problem: We want a history of the last N things. Solution: Use a `deque`.

```
from collections import deque

history = deque(maxlen=N)
with open(filename) as f:
```

```
    for line in f:
        history.append(line)
        ...
```

16.1 Exercises

The `collections` module might be one of the most useful library modules for dealing with special purpose kinds of data handling problems such as tabulating and indexing.

In this exercise, we'll look at a few simple examples. Start by running your `report.py` program so that you have the portfolio of stocks loaded in the interactive mode.

```
bash % python3 -i report.py
```

16.1.1 Exercise 2.18: Tabulating with Counters

Suppose you wanted to tabulate the total number of shares of each stock. This is easy using `Counter` objects. Try it:

```
>>> portfolio = read_portfolio('Data/portfolio.csv')
>>> from collections import Counter
>>> holdings = Counter()
>>> for s in portfolio:
        holdings[s['name']] += s['shares']

>>> holdings
Counter({'MSFT': 250, 'IBM': 150, 'CAT': 150, 'AA': 100, 'GE': 95})
>>>
```

Carefully observe how the multiple entries for `MSFT` and `IBM` in `portfolio` get combined into a single entry here.

You can use a Counter just like a dictionary to retrieve individual values:

```
>>> holdings['IBM']
150
>>> holdings['MSFT']
250
>>>
```

If you want to rank the values, do this:

```
>>> # Get three most held stocks
>>> holdings.most_common(3)
```

```
[('MSFT', 250), ('IBM', 150), ('CAT', 150)]
>>>
```

Let's grab another portfolio of stocks and make a new Counter:

```
>>> portfolio2 = read_portfolio('Data/portfolio2.csv')
>>> holdings2 = Counter()
>>> for s in portfolio2:
        holdings2[s['name']] += s['shares']

>>> holdings2
Counter({'HPQ': 250, 'GE': 125, 'AA': 50, 'MSFT': 25})
>>>
```

Finally, let's combine all of the holdings doing one simple operation:

```
>>> holdings
Counter({'MSFT': 250, 'IBM': 150, 'CAT': 150, 'AA': 100, 'GE': 95})
>>> holdings2
Counter({'HPQ': 250, 'GE': 125, 'AA': 50, 'MSFT': 25})
>>> combined = holdings + holdings2
>>> combined
Counter({'MSFT': 275, 'HPQ': 250, 'GE': 220, 'AA': 150, 'IBM': 150, 'CAT': 150})
>>>
```

This is only a small taste of what counters provide. However, if you ever find yourself needing to tabulate values, you should consider using one.

16.1.2 Commentary: collections module

The `collections` module is one of the most useful library modules in all of Python. In fact, we could do an extended tutorial on just that. However, doing so now would also be a distraction. For now, put `collections` on your list of bedtime reading for later.

Contents | Previous (2.4 Sequences) | Next (2.6 List Comprehensions) Contents | Previous (2.5 Collections) | Next (2.7 Object Model)

17 2.6 List Comprehensions

A common task is processing items in a list. This section introduces list comprehensions, a powerful tool for doing just that.

17.0.1 Creating new lists

A list comprehension creates a new list by applying an operation to each element of a sequence.

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a ]
>>> b
[2, 4, 6, 8, 10]
>>>
```

Another example:

```
>>> names = ['Elwood', 'Jake']
>>> a = [name.lower() for name in names]
>>> a
['elwood', 'jake']
>>>
```

The general syntax is: `[ <expression> for <variable_name> in <sequence> ]`.

17.0.2 Filtering

You can also filter during the list comprehension.

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0 ]
>>> b
[2, 8, 4, 20]
>>>
```

17.0.3 Use cases

List comprehensions are hugely useful. For example, you can collect values of a specific dictionary fields:

```
stocknames = [s['name'] for s in stocks]
```

You can perform database-like queries on sequences.

```
a = [s for s in stocks if s['price'] > 100 and s['shares'] > 50 ]
```

You can also combine a list comprehension with a sequence reduction:

```
cost = sum([s['shares']*s['price'] for s in stocks])
```

17.0.4 General Syntax

```
[ <expression> for <variable_name> in <sequence> if <condition>]
```

What it means:

```
result = []
for variable_name in sequence:
    if condition:
        result.append(expression)
```

17.0.5 Historical Digression

List comprehensions come from math (set-builder notation).

```
a = [ x * x for x in s if x > 0 ] # Python

a = { x^2 | x  s, x > 0 }         # Math
```

It is also implemented in several other languages. Most coders probably aren't thinking about their math class though. So, it's fine to view it as a cool list shortcut.

17.1 Exercises

Start by running your `report.py` program so that you have the portfolio of stocks loaded in the interactive mode.

```
bash % python3 -i report.py
```

Now, at the Python interactive prompt, type statements to perform the operations described below. These operations perform various kinds of data reductions, transforms, and queries on the portfolio data.

17.1.1 Exercise 2.19: List comprehensions

Try a few simple list comprehensions just to become familiar with the syntax.

```
>>> nums = [1,2,3,4]
>>> squares = [ x * x for x in nums ]
>>> squares
[1, 4, 9, 16]
>>> twice = [ 2 * x for x in nums if x > 2 ]
>>> twice
```

```
[6, 8]
>>>
```

Notice how the list comprehensions are creating a new list with the data suitably transformed or filtered.

17.1.2 Exercise 2.20: Sequence Reductions

Compute the total cost of the portfolio using a single Python statement.

```
>>> portfolio = read_portfolio('Data/portfolio.csv')
>>> cost = sum([ s['shares'] * s['price'] for s in portfolio ])
>>> cost
44671.15
>>>
```

After you have done that, show how you can compute the current value of the portfolio using a single statement.

```
>>> value = sum([ s['shares'] * prices[s['name']] for s in portfolio ])
>>> value
28686.1
>>>
```

Both of the above operations are an example of a map-reduction. The list comprehension is mapping an operation across the list.

```
>>> [ s['shares'] * s['price'] for s in portfolio ]
[3220.0000000000005, 4555.0, 12516.0, 10246.0, 3835.1499999999996, 3254.9999999999995,
>>>
```

The `sum()` function is then performing a reduction across the result:

```
>>> sum(_)
44671.15
>>>
```

With this knowledge, you are now ready to go launch a big-data startup company.

17.1.3 Exercise 2.21: Data Queries

Try the following examples of various data queries.

First, a list of all portfolio holdings with more than 100 shares.

```
>>> more100 = [ s for s in portfolio if s['shares'] > 100 ]
>>> more100
[{'price': 83.44, 'name': 'CAT', 'shares': 150}, {'price': 51.23, 'name': 'MSFT', 'sha
>>>
```

All portfolio holdings for MSFT and IBM stocks.

```
>>> msftibm = [ s for s in portfolio if s['name'] in {'MSFT','IBM'} ]
>>> msftibm
[{'price': 91.1, 'name': 'IBM', 'shares': 50}, {'price': 51.23, 'name': 'MSFT', 'share
  {'price': 65.1, 'name': 'MSFT', 'shares': 50}, {'price': 70.44, 'name': 'IBM', 'shar
>>>
```

A list of all portfolio holdings that cost more than $10000.

```
>>> cost10k = [ s for s in portfolio if s['shares'] * s['price'] > 10000 ]
>>> cost10k
[{'price': 83.44, 'name': 'CAT', 'shares': 150}, {'price': 51.23, 'name': 'MSFT', 'sha
>>>
```

17.1.4 Exercise 2.22: Data Extraction

Show how you could build a list of tuples `(name, shares)` where `name` and `shares` are taken from `portfolio`.

```
>>> name_shares =[ (s['name'], s['shares']) for s in portfolio ]
>>> name_shares
[('AA', 100), ('IBM', 50), ('CAT', 150), ('MSFT', 200), ('GE', 95), ('MSFT', 50), ('IB
>>>
```

If you change the the square brackets (`[,]`) to curly braces (`{, }`), you get something known as a set comprehension. This gives you unique or distinct values.

For example, this determines the set of unique stock names that appear in `portfolio`:

```
>>> names = { s['name'] for s in portfolio }
>>> names
{ 'AA', 'GE', 'IBM', 'MSFT', 'CAT'] }
>>>
```

If you specify `key:value` pairs, you can build a dictionary. For example, make a dictionary that maps the name of a stock to the total number of shares held.

```
>>> holdings = { name: 0 for name in names }
>>> holdings
```

```
{'AA': 0, 'GE': 0, 'IBM': 0, 'MSFT': 0, 'CAT': 0}
>>>
```

This latter feature is known as a **dictionary comprehension**. Let's tabulate:

```
>>> for s in portfolio:
        holdings[s['name']] += s['shares']

>>> holdings
{ 'AA': 100, 'GE': 95, 'IBM': 150, 'MSFT':250, 'CAT': 150 }
>>>
```

Try this example that filters the `prices` dictionary down to only those names that appear in the portfolio:

```
>>> portfolio_prices = { name: prices[name] for name in names }
>>> portfolio_prices
{'AA': 9.22, 'GE': 13.48, 'IBM': 106.28, 'MSFT': 20.89, 'CAT': 35.46}
>>>
```

17.1.5 Exercise 2.23: Extracting Data From CSV Files

Knowing how to use various combinations of list, set, and dictionary comprehensions can be useful in various forms of data processing. Here's an example that shows how to extract selected columns from a CSV file.

First, read a row of header information from a CSV file:

```
>>> import csv
>>> f = open('Data/portfoliodate.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> headers
['name', 'date', 'time', 'shares', 'price']
>>>
```

Next, define a variable that lists the columns that you actually care about:

```
>>> select = ['name', 'shares', 'price']
>>>
```

Now, locate the indices of the above columns in the source CSV file:

```
>>> indices = [ headers.index(colname) for colname in select ]
>>> indices
[0, 3, 4]
```

```
>>>
```

Finally, read a row of data and turn it into a dictionary using a dictionary comprehension:

```
>>> row = next(rows)
>>> record = { colname: row[index] for colname, index in zip(select, indices) }   # di
>>> record
{'price': '32.20', 'name': 'AA', 'shares': '100'}
>>>
```

If you're feeling comfortable with what just happened, read the rest of the file:

```
>>> portfolio = [ { colname: row[index] for colname, index in zip(select, indices) } f
>>> portfolio
[{'price': '91.10', 'name': 'IBM', 'shares': '50'}, {'price': '83.44', 'name': 'CAT',
  {'price': '51.23', 'name': 'MSFT', 'shares': '200'}, {'price': '40.37', 'name': 'GE'
  {'price': '65.10', 'name': 'MSFT', 'shares': '50'}, {'price': '70.44', 'name': 'IBM'
>>>
```

Oh my, you just reduced much of the `read_portfolio()` function to a single statement.

17.1.6 Commentary

List comprehensions are commonly used in Python as an efficient means for transforming, filtering, or collecting data. Due to the syntax, you don't want to go overboard—try to keep each list comprehension as simple as possible. It's okay to break things into multiple steps. For example, it's not clear that you would want to spring that last example on your unsuspecting co-workers.

That said, knowing how to quickly manipulate data is a skill that's incredibly useful. There are numerous situations where you might have to solve some kind of one-off problem involving data imports, exports, extraction, and so forth. Becoming a guru master of list comprehensions can substantially reduce the time spent devising a solution. Also, don't forget about the `collections` module.

Contents | Previous (2.5 Collections) | Next (2.7 Object Model)

Contents | Previous (2.6 List Comprehensions) | Next (3 Program Organization)

18 2.7 Objects

This section introduces more details about Python's internal object model and discusses some matters related to memory management, copying, and type checking.

18.0.1 Assignment

Many operations in Python are related to *assigning* or *storing* values.

```
a = value          # Assignment to a variable
s[n] = value       # Assignment to an list
s.append(value)    # Appending to a list
d['key'] = value   # Adding to a dictionary
```

A caution: assignment operations ***never make a copy*** *of the value being assigned.* All assignments are merely reference copies (or pointer copies if you prefer).

18.0.2 Assignment example

Consider this code fragment.

```
a = [1,2,3]
b = a
c = [a,b]
```

A picture of the underlying memory operations. In this example, there is only one list object `[1,2,3]`, but there are four different references to it.

References

This means that modifying a value affects *all* references.

```
>>> a.append(999)
>>> a
[1,2,3,999]
>>> b
[1,2,3,999]
>>> c
[[1,2,3,999], [1,2,3,999]]
>>>
```

Notice how a change in the original list shows up everywhere else (yikes!). This is because no copies were ever made. Everything is pointing to the same thing.

18.0.3 Reassigning values

Reassigning a value *never* overwrites the memory used by the previous value.

```
a = [1,2,3]
b = a
```

```
a = [4,5,6]

print(a)      # [4, 5, 6]
print(b)      # [1, 2, 3]    Holds the original value
```

Remember: **Variables are names, not memory locations.**

18.0.4 Some Dangers

If you don't know about this sharing, you will shoot yourself in the foot at some point. Typical scenario. You modify some data thinking that it's your own private copy and it accidentally corrupts some data in some other part of the program.

Comment: This is one of the reasons why the primitive datatypes (int, float, string) are immutable (read-only).

18.0.5 Identity and References

Use the `is` operator to check if two values are exactly the same object.

```
>>> a = [1,2,3]
>>> b = a
>>> a is b
True
>>>
```

`is` compares the object identity (an integer). The identity can be obtained using `id()`.

```
>>> id(a)
3588944
>>> id(b)
3588944
>>>
```

Note: It is almost always better to use `==` for checking objects. The behavior of `is` is often unexpected:

```
>>> a = [1,2,3]
>>> b = a
>>> c = [1,2,3]
>>> a is b
True
>>> a is c
False
```

```
>>> a == c
True
>>>
```

18.0.6 Shallow copies

Lists and dicts have methods for copying.

```
>>> a = [2,3,[100,101],4]
>>> b = list(a) # Make a copy
>>> a is b
False
```

It's a new list, but the list items are shared.

```
>>> a[2].append(102)
>>> b[2]
[100,101,102]
>>>
>>> a[2] is b[2]
True
>>>
```

For example, the inner list `[100, 101, 102]` is being shared. This is known as a shallow copy. Here is a picture.

Shallow copy

18.0.7 Deep copies

Sometimes you need to make a copy of an object and all the objects contained within it. You can use the `copy` module for this:

```
>>> a = [2,3,[100,101],4]
>>> import copy
>>> b = copy.deepcopy(a)
>>> a[2].append(102)
>>> b[2]
[100,101]
>>> a[2] is b[2]
False
>>>
```

18.0.8 Names, Values, Types

Variable names do not have a *type*. It's only a name. However, values *do* have an underlying type.

```
>>> a = 42
>>> b = 'Hallo Python'
>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>
```

`type()` will tell you what it is. The type name is usually used as a function that creates or converts a value to that type.

18.0.9 Type Checking

How to tell if an object is a specific type.

```
if isinstance(a, list):
    print('a is a list')
```

Checking for one of many possible types.

```
if isinstance(a, (list,tuple)):
    print('a is a list or tuple')
```

Caution: Don't go overboard with type checking. It can lead to excessive code complexity. Usually you'd only do it if doing so would prevent common mistakes made by others using your code.

18.0.10 Everything is an object

Numbers, strings, lists, functions, exceptions, classes, instances, etc. are all objects. It means that all objects that can be named can be passed around as data, placed in containers, etc., without any restrictions. There are no *special* kinds of objects. Sometimes it is said that all objects are "first-class".

A simple example:

```
>>> import math
>>> items = [abs, math, ValueError ]
>>> items
[<built-in function abs>,
  <module 'math' (builtin)>,
```

```
  <type 'exceptions.ValueError'>]
>>> items[0](-45)
45
>>> items[1].sqrt(2)
1.4142135623730951
>>> try:
        x = int('not a number')
    except items[2]:
        print('Failed!')
Failed!
>>>
```

Here, `items` is a list containing a function, a module and an exception. You can directly use the items in the list in place of the original names:

```
items[0](-45)       # abs
items[1].sqrt(2)    # math
except items[2]:    # ValueError
```

With great power comes responsibility. Just because you can do that doesn't mean you should.

18.1 Exercises

In this set of exercises, we look at some of the power that comes from first-class objects.

18.1.1 Exercise 2.24: First-class Data

In the file `Data/portfolio.csv`, we read data organized as columns that look like this:

```
name,shares,price
"AA",100,32.20
"IBM",50,91.10
...
```

In previous code, we used the `csv` module to read the file, but still had to perform manual type conversions. For example:

```
for row in rows:
    name   = row[0]
    shares = int(row[1])
    price  = float(row[2])
```

This kind of conversion can also be performed in a more clever manner using some list basic operations.

Make a Python list that contains the names of the conversion functions you would use to convert each column into the appropriate type:

```
>>> types = [str, int, float]
>>>
```

The reason you can even create this list is that everything in Python is *first-class*. So, if you want to have a list of functions, that's fine. The items in the list you created are functions for converting a value `x` into a given type (e.g., `str(x)`, `int(x)`, `float(x)`).

Now, read a row of data from the above file:

```
>>> import csv
>>> f = open('Data/portfolio.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> row = next(rows)
>>> row
['AA', '100', '32.20']
>>>
```

As noted, this row isn't enough to do calculations because the types are wrong. For example:

```
>>> row[1] * row[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
>>>
```

However, maybe the data can be paired up with the types you specified in `types`. For example:

```
>>> types[1]
<type 'int'>
>>> row[1]
'100'
>>>
```

Try converting one of the values:

```
>>> types[1](row[1])      # Same as int(row[1])
100
>>>
```

Try converting a different value:

```
>>> types[2](row[2])     # Same as float(row[2])
32.2
>>>
```

Try the calculation with converted values:

```
>>> types[1](row[1])*types[2](row[2])
3220.0000000000005
>>>
```

Zip the column types with the fields and look at the result:

```
>>> r = list(zip(types, row))
>>> r
[(<type 'str'>, 'AA'), (<type 'int'>, '100'), (<type 'float'>,'32.20')]
>>>
```

You will notice that this has paired a type conversion with a value. For example, `int` is paired with the value `'100'`.

The zipped list is useful if you want to perform conversions on all of the values, one after the other. Try this:

```
>>> converted = []
>>> for func, val in zip(types, row):
          converted.append(func(val))
...
>>> converted
['AA', 100, 32.2]
>>> converted[1] * converted[2]
3220.0000000000005
>>>
```

Make sure you understand what's happening in the above code. In the loop, the `func` variable is one of the type conversion functions (e.g., `str`, `int`, etc.) and the `val` variable is one of the values like `'AA'`, `'100'`. The expression `func(val)` is converting a value (kind of like a type cast).

The above code can be compressed into a single list comprehension.

```
>>> converted = [func(val) for func, val in zip(types, row)]
>>> converted
['AA', 100, 32.2]
>>>
```

18.1.2 Exercise 2.25: Making dictionaries

Remember how the `dict()` function can easily make a dictionary if you have a sequence of key names and values? Let's make a dictionary from the column headers:

```
>>> headers
['name', 'shares', 'price']
>>> converted
['AA', 100, 32.2]
>>> dict(zip(headers, converted))
{'price': 32.2, 'name': 'AA', 'shares': 100}
>>>
```

Of course, if you're up on your list-comprehension fu, you can do the whole conversion in a single step using a dict-comprehension:

```
>>> { name: func(val) for name, func, val in zip(headers, types, row) }
{'price': 32.2, 'name': 'AA', 'shares': 100}
>>>
```

18.1.3 Exercise 2.26: The Big Picture

Using the techniques in this exercise, you could write statements that easily convert fields from just about any column-oriented datafile into a Python dictionary.

Just to illustrate, suppose you read data from a different datafile like this:

```
>>> f = open('Data/dowstocks.csv')
>>> rows = csv.reader(f)
>>> headers = next(rows)
>>> row = next(rows)
>>> headers
['name', 'price', 'date', 'time', 'change', 'open', 'high', 'low', 'volume']
>>> row
['AA', '39.48', '6/11/2007', '9:36am', '-0.18', '39.67', '39.69', '39.45', '181800']
>>>
```

Let's convert the fields using a similar trick:

```
>>> types = [str, float, str, str, float, float, float, float, int]
>>> converted = [func(val) for func, val in zip(types, row)]
>>> record = dict(zip(headers, converted))
>>> record
{'volume': 181800, 'name': 'AA', 'price': 39.48, 'high': 39.69,
'low': 39.45, 'time': '9:36am', 'date': '6/11/2007', 'open': 39.67,
```

```
'change': -0.18}
>>> record['name']
'AA'
>>> record['price']
39.48
>>>
```

Bonus: How would you modify this example to additionally parse the `date` entry into a tuple such as `(6, 11, 2007)`?

Spend some time to ponder what you've done in this exercise. We'll revisit these ideas a little later.

Contents | Previous (2.6 List Comprehensions) | Next (3 Program Organization)

Contents | Prev (2 Working With Data) | Next (4 Classes and Objects)

19 3. Program Organization

So far, we've learned some Python basics and have written some short scripts. However, as you start to write larger programs, you'll want to get organized. This section dives into greater details on writing functions, handling errors, and introduces modules. By the end you should be able to write programs that are subdivided into functions across multiple files. We'll also give some useful code templates for writing more useful scripts.

Contents | Prev (2 Working With Data) | Next (4 Classes and Objects)

Contents | Prev (3 Program Organization) | Next (5 Inner Workings of Python Objects)

20 4. Classes and Objects

So far, our programs have only used built-in Python datatypes. In this section, we introduce the concept of classes and objects. You'll learn about the `class` statement that allows you to make new objects. We'll also introduce the concept of inheritance, a tool that is commonly use to build extensible programs. Finally, we'll look at a few other

features of classes including special methods, dynamic attribute lookup, and defining new exceptions.

- 4.1 Introducing Classes
- 4.2 Inheritance
- 4.3 Special Methods
- 4.4 Defining new Exception

Contents | Prev (3 Program Organization) | Next (5 Inner Workings of Python Objects)

Contents | Prev (4 Classes and Objects) | Next (6 Generators)

21 5. Inner Workings of Python Objects

This section covers some of the inner workings of Python objects. Programmers coming from other programming languages often find Python's notion of classes lacking in features. For example, there is no notion of access-control (e.g., private, protected), the whole `self` argument feels weird, and frankly, working with objects sometimes feel like a "free for all." Maybe that's true, but we'll find out how it all works as well as some common programming idioms to better encapsulate the internals of objects.

It's not necessary to worry about the inner details to be productive. However, most Python coders have a basic awareness of how classes work. So, that's why we're covering it.

- 5.1 Dictionaries Revisited (Object Implementation)
- 5.2 Encapsulation Techniques

Contents | Prev (4 Classes and Objects) | Next (6 Generators)

Contents | Prev (5 Inner Workings of Python Objects) | Next (7 Advanced Topics)

22 6. Generators

Iteration (the `for`-loop) is one of the most common programming patterns in Python. Programs do a lot of iteration to process lists, read files, query databases, and more. One of the most powerful features of Python is the ability to customize and redefine iteration in the form of a so-called "generator function." This section introduces this topic. By the end, you'll write some programs that process some real-time streaming data in an interesting way.

- 6.1 Iteration Protocol
- 6.2 Customizing Iteration with Generators
- 6.3 Producer/Consumer Problems and Workflows

- 6.4 Generator Expressions

Contents | Prev (5 Inner Workings of Python Objects) | Next (7 Advanced Topics)

Contents | Prev (6 Generators) | Next (8 Testing and Debugging

23 7. Advanced Topics

In this section, we look at a small set of somewhat more advanced Python features that you might encounter in your day-to-day coding. Many of these topics could have been covered in earlier course sections, but weren't in order to spare you further head-explosion at the time.

It should be emphasized that the topics in this section are only meant to serve as a very basic introduction to these ideas. You will need to seek more advanced material to fill out details.

- 7.1 Variable argument functions
- 7.2 Anonymous functions and lambda
- 7.3 Returning function and closures
- 7.4 Function decorators
- 7.5 Static and class methods

Contents | Prev (6 Generators) | Next (8 Testing and Debugging)

Contents | Prev (7 Advanced Topics) | Next (9 Packages)

24 8. Overview

This section introduces a few basic topics related to testing, logging, and debugging.

- 8.1 Testing
- 8.2 Logging, error handling and diagnostics
- 8.3 Debugging

Contents | Prev (7 Advanced Topics) | Next (9 Packages)

Contents | Prev (8 Testing and Debugging)

25 9 Packages

We conclude the course with a few details on how to organize your code into a package structure. We'll also discuss the installation of third party packages and preparing to

give your own code away to others.

The subject of packaging is an ever-evolving, overly complex part of Python development. Rather than focus on specific tools, the main focus of this section is on some general code organization principles that will prove useful no matter what tools you later use to give code away or manage dependencies.

Contents | Prev (8 Testing and Debugging)

26 Ausblick

Wir sind am Ende von *The Road to React* angekommen. Ich hoffe, du hattest Vergnügen beim Lesen und hast dir dabei gleichzeitig die Grundlagen zum Arbeiten mit React angeeignet. Wenn dir das Buch gefallen hat, freue ich mich, wenn du es mit deinen Freunden teilst — insbesondere mit denen, die wie du an React interessiert sind. Eine konstruktive Rezension bei Amazon oder Goodreads hilft mir, in Zukunft bessere Inhalte basierend auf deinem Feedback anzubieten.

Von hier aus empfehle ich dir, die Beispiel-Anwendung zu erweitern. Erstelle dein eigenes React-Projekt bevor du ein weiteres Buch, einen anderen Kurs oder ein zusätzliches Tutorial in Angriff nimmst. Probieren das Gelernte eine Woche lang aus und teile es mit anderen, indem du es zum Beispiel auf Github veröffentlichst. Wende dich gerne an mich oder andere. Ich bin immer daran interessiert zu sehen, was meine Leser entwickelt haben und wie ich sie bestmöglich unterstütze.

Nachdem du die Grundlagen beherrschst, empfehle ich dir das Folgende um deine Anwendung und dein Wissen sinnvoll zu erweitern:

- **Herstellen einer Verbindung zu einer Datenbank und/oder Authentifizierung:** In wachsende React-Anwendungen sind persistente Daten in der Regel unumgänglich. Die Daten werden in der Regel in einer Datenbank gespeichert, damit sie nach Browsersitzungen erhalten bleiben und für verschiedene Benutzer zugänglich sind. Firebase ist eine der einfachsten Möglichkeiten, eine Datenbank zu integrieren, ohne eine eigene Backend-Anwendung zu schreiben. Mein Buch mit dem Titel "The Road to Firebase" bietet dir eine schrittweise Anleitung zur Verwendung der Firebase-Authentifizierung und -Datenbank in React.
- **Die Verbindung zu einem Administrationsbereich/Backend:** React bietet ein Grundgerüst für Frontend-Anwendungen, und wir haben in unsere Beispielanwendung ausschließlich Daten von der API eines Drittanbieters im Frontend

angezeigt. Erstelle selbst eine API mit einer Backend-Anwendung, die eine Verbindung zu einer Datenbank herstellt und die Authentifizierung und Autorisierung bietet. In "The Road to GraphQL" erkläre ich dir, wie du GraphQL für die Client-Server-Kommunikation verwendest. Du erfährst, wie du dein eigenes Backend mit einer Datenbank verbindest, Benutzersitzungen verwaltest und wie du über eine GraphQL-API dein Frontend mit der Backend-Anwendung verknüpfst.

- **Statusverwaltung:** Du hast React verwendet, um den lokalen Komponentenstatus zu verwalten. Dies ist eine solide Grundlage für die meisten Anwendungen. Zusätzlich gibt es externe Statusverwaltungslösungen. Ich behandele die beliebteste in meinem Buch "The Road to Redux".
- **Tooling mit Webpack und Babel:** Wir haben die *Create React App* verwendet, um die Anwendung in diesem Buch einzurichten. Dein Ziel ist es sicher, die Einrichtung einmal selbst in die Hand zu nehmen und die Werkzeuge zu erlernen, um Projekte unabhängig von der *Create React App* zu erstellen. Ich empfehle dir ein minimales Setup mit Webpack, wobei du nach und nach je nach Anforderung zusätzliche Werkzeuge einbindest.
- **Code-Organisation:** Öffne das Kapitel über die Code-Organisation und wende die dort beschriebenen Änderungen an, falls du dies bisher nicht getan hast. Es hilft dir dabei, deine Komponenten in strukturierten Dateien und Ordnern zu organisieren und die Prinzipien der Codeaufteilung, Wiederverwendbarkeit, Wartbarkeit und des Modul-API-Designs zu verstehen. Deine Anwendung wird wachsen und strukturierte Module benötigen. Deshalb ist es besser, schon jetzt die Grundlagen zu legen.
- **Testen:** Wir haben die Oberfläche der Möglichkeiten im Bereich Tests angekratzt. Wenn du mit dem Testen von Webanwendungen nicht vertraut bist, vertiefe dein Wissen im Bereich Unit-Tests und Integrationstests mit React-Anwendungen. Cypress ist ein nützliches Tool für End-to-End-Tests in React.
- **Typprüfung:** In der Vergangenheit wurde TypeScript in React verwendet. Dies ist eine bewährte Methode, um Fehler zu vermeiden, und die Benutzbarkeit für Entwickler zu verbessern. Tauche tiefer in dieses Thema ein, und erstelle deine JavaScript-Anwendungen robuster. Wer weiß? Unter Umständen verwendest du in Zukunft überwiegend TypeScript anstelle von JavaScript.
- **UI-Komponenten:** Viele Anfänger führen Frameworks oder UI-Komponentenbibliotheken wie Bootstrap meiner Meinung nach zu früh in ihre Projekte ein. Auf den ersten Blick erscheint es praktisch, ein Dropdown-Menü, ein Kontrollkästchen oder einen Dialog mit Standard-HTML-Elementen zu integrieren. Beachte dabei: Die meisten dieser Komponenten verwalten ihren eigenen lokalen Status. Ein Kontrollkästchen weiß, ob es aktiviert oder deaktiviert ist. Implementiere dieses daher als gesteuerte Komponenten. Nachdem du die grundlegenden Implementierungen der wichtigen UI-Komponente erarbeitet hast, ist die Einführung einer

UI-Komponentenbibliothek unkomplizierter.

- **Routing:** Implementiere das Routing für deine Anwendung mit React Router. Es gibt bisher nur eine Seite in der Beispiel-Anwendung, aber diese wird wachsen. Mit React Router verwaltest du weitere Seiten über mehrere URLs hinweg. Wenn du das Routing in deine Anwendung integrierst, werden für neue Seitenaufrufe keine Anforderungen an den Webserver gesandt. Der Router übernimmt diese clientseitig.
- **React Native:** React Native ermöglicht es dir, native Apps plattformübergreifend und parallel für Android und iOS zu programmieren. Sobald du React beherrschst, ist die Lernkurve für React Native nicht steil, da beide dieselben Prinzipien teilen. Einige wenige Unterschiede gibt es bei den Layoutkomponenten, den Build-Tools und den APIs deines Mobilgeräts.

Last but not least lade ich dich ein, meine Website zu besuchen, um weitere Informationen zu aktuellen Themen im Bereich Webentwicklung und Softwareentwicklung zu lesen. Abonniere gerne meinen Newsletter oder folge mir auf Twitter, um Updates zu Artikeln, Büchern und Kursen zu erhalten.

Vielen Dank dafür, dass du mein Buch gelesen hast.

Viele Grüße,

Robin Wieruch

27 Stichwortverzeichnis

y

Z

28 Literaturverzeichnis

https://github.com/dabeaz-course/practical-python, David Beazley; 2020. „Python“. Chicago: @dabeaz.