Course Setup and Overview

Welcome to Practical Python Programming! This page has some important information about course setup and logistics.

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).

Setup and Python Installation

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.

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.

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.

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.

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 | Next (1 Introduction to Python)

Practical Python Programming

Table of Contents

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

Home

Contents | Next (2 Working With Data)

1. Introduction to Python

The goal of this first section is to introduce some Python basics from the ground up. Starting with nothing, you'll learn how to edit, run, and debug small programs. Ultimately, you'll write a short script that reads a CSV data file and performs a simple calculation.

Contents | Next (2 Working With Data)

Contents | Next (1.2 A First Program)

1.1 Python

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.

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.

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

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("hello world")
hello world
>>>
```

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.

Exercises

Exercise 1.1: Using Python as a Calculator

On your machine, start Python and use it as a calculator 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
>>>
```

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).

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.

Exercise 1.4: Where is My Bus?

Note: This was a whimsical example that was a real crowd-pleaser when I taught this course in my office. You could query the bus and then literally watch it pass by the window out front. Sadly, APIs rarely live forever and it seems that this one has now ridden off into the sunset. --Dave

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/getStopPredictions.jsp?
stop=14791&route=22')
>>> 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 A First Program)

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

1.2 A First Program

This section discusses the creation of your first program, running the interpreter, and some basic debugging.

Running Python

Python programs always run inside an interpreter.

The interpreter is a "console-based" application that normally runs from a command shell.

```
python3
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Expert programmers usually have no problem using the interpreter in this way, but it's not so user-friendly for beginners. You may be using an environment that provides a different interface to Python. That's fine, but learning how to run Python terminal is still a useful skill to know.

Interactive Mode

When you start Python, you get an *interactive* mode where you can experiment.

If you start typing statements, they will run immediately. There is no edit/compile/run/debug cycle.

```
>>> print('hello world')
hello world
>>> 37*42
1554
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

This so-called *read-eval-print-loop* (or REPL) is very useful for debugging and exploration.

STOP: If you can't figure out how to interact with Python, stop what you're doing and figure out how to do it. If you're using an IDE, it might be hidden behind a menu option or other window. Many parts of this course assume that you can interact with the interpreter.

Let's take a closer look at the elements of the REPL:

- `>>>` is the interpreter prompt for starting a new statement.
- `...` is the interpreter prompt for continuing a statement. Enter a blank line to finish typing and run what you've entered.

The `...` prompt may or may not be shown depending on your environment. For this course, it is shown as blanks to make it easier to cut/paste code samples.

The underscore `_` holds the last result.

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

This is only true in the interactive mode. You never use `_` in a program.

Creating programs

Programs are put in `.py` files.

```
# hello.py
print('hello world')
```

You can create these files with your favorite text editor.

Running Programs

To execute a program, run it in the terminal with the `python` command. For example, in command-line Unix:

```
bash % python hello.py
hello world
bash %
```

Or from the Windows shell:

```
C:\SomeFolder>hello.py
hello world

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

Note: On Windows, you may need to specify a full path to the Python interpreter such as `c:\python36\python`. However, if Python is installed in its usual way, you might be able to just type the name of the program such as `hello.py`.

A Sample Program

Let's solve the following problem:

> One morning, you go out and place a dollar bill on the sidewalk by the Sears tower in Chicago. Each day thereafter, you go out double the number of bills. How long does it take for the stack of bills to exceed the height of the tower?

Here's a solution:

```
# 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 = day + 1
    num_bills = num_bills * 2

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

When you run it, you get the following output:

```
bash % python3 sears.py
1 1 0.00011
2 2 0.00022
3 4 0.00044
4 8 0.00088
5 16 0.00176
6 32 0.00352
...
21 1048576 115.34336
22 2097152 230.68672
Number of days 23
Number of bills 4194304
Final height 461.37344
```

Using this program as a guide, you can learn a number of important core concepts about Python.

Statements

A python program is a sequence of statements:

```
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.

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.

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
```

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.

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
```

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', day)
```

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

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', day)
```

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.

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
```

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')
```

Printing

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

```
print('Hello world!') # Prints the text 'Hello world!'
```

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
```

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.

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.

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.

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
```

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

1.3 Numbers

This section discusses mathematical calculations.

Types of Numbers

Python has 4 types of numbers:

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

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.

Integers (int)

Signed values of arbitrary size and base:

```
a = 37
b = -299392993727716627377128481812241231
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
```

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 of 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)
```

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')
```

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
>>>
```

Exercises

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

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`.

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.

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 = 61
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 after the first five years have already been paid?

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 3478.83
309 877389.99 809.21
310 880074.1 -1871.53
Total paid 880074.1
Months 310
```

Exercise 1.11: Bonus

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

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 A First Program) | Next (1.4 Strings)

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

1.4 Strings

This section introduces ways to work with text.

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. *However, the same type of quote used to start a string must be used to terminate it*.

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
```

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.

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 = 'Hello world'
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.

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'
```

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 = 'Hello world'
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
```

String Mutability

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

```
>>> s = 'Hello World'
>>> 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.

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'
>>>
```

Byte Strings

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

```
data = b'Hello World\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'`.

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.

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.

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'
>>>
```

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
>>>
```

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.

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`?

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
?
>>>
```

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.

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.

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 = 'hello world'
>>> 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)

1.5 Lists

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

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']
>>>
```

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]
```

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.

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.

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
```

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).

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(',')
```

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`.

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
```

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
>>>
```

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.

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.

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'
>>>
```

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)

1.6 File Management

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

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.

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
```

Common Idioms for Writing to a File

Write string data.

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

Redirect the print function.

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

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
>>>
```

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\n"GE",
```

```
95,40.37\n"MSFT",50,65.10\n"IBM",100,70.44\n'
>>> 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.

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
```

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.

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)

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.

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.

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.

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.

Catching and Handling Exceptions

Exceptions can be caught and handled.

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

```
for line in file:
    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!").

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
```

Exercises

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.

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.

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.

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.

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 %
```

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.

Contents | Prev (1 Introduction to Python) | Next (3 Program Organization)

2.1 Datatypes and Data structures

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

Primitive Datatypes

Python has a few primitive types of data:

- Integers
- Floating point numbers
- Strings (text)

We learned about these in the introduction.

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)
```

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)

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])
```

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.

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
```

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
```

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
}
```

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']
>>>
```

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]
```

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.

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)
>>>
```

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': 32.2 }
>>>
```

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}
>>>
```

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)

2.2 Containers

This section discusses lists, dictionaries, and sets.

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.

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)
```

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])))
```

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
>>>
```

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).

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
>>>
```

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.

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:

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

s1 = { 'a', 'b', 'c'}
s2 = { 'c', 'd' }
s1 | s2                 # Set union { 'a', 'b', 'c', 'd' }
s1 & s2                 # Set intersection { 'c' }
s1 - s2                 # Set difference { 'a', 'b' }
```

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`.

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
>>>
```

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': 91.1},
    {'name': 'CAT', 'shares': 150, 'price': 83.44}, {'name': 'MSFT', 'shares': 200, 'price':
51.23},
    {'name': 'GE', 'shares': 95, 'price': 40.37}, {'name': 'MSFT', 'shares': 50, 'price':
65.1},
    {'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}]
>>>
```

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
>>>
```

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)

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
```

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}')
```

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
```

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.

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.

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 %d messages' % (b'Dave', 37)
b'Dave has 37 messages'
>>> b'%b has %d messages' % (b'Dave', 37)  # %b may be used instead of %s
b'Dave has 37 messages'
>>>
```

Exercises

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'
>>>
```

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)
...
>>>
```

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.

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
```

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)

2.4 Sequences

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
```

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.

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]
```

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'
>>>
```

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.

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.

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.

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.

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.

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 number of items in each tuple.

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))
```

Exercises

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
>>>
```

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.

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}')
```

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.

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)

2.5 collections module

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

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.

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
```

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.

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)
        ...
```

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
```

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.

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)

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.

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> ]`.

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]
>>>
```

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])
```

General Syntax

```
[ <expression> for <variable_name> in <sequence> if <condition>]
```

What it means:

```
result = []
for variable_name in sequence:
    if condition:
        result.append(expression)
```

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.

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.

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.

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,
7044.0]
>>>
```

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.

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', 'shares':
200}]
>>>
```

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', 'shares':
200},
  {'price': 65.1, 'name': 'MSFT', 'shares': 50}, {'price': 70.44, 'name': 'IBM', 'shares':
100}]
>>>
```

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', 'shares':
200}]
>>>
```

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), ('IBM',
100)]
>>>
```

If you change 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}
>>>
```

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) }   # dict-
comprehension
>>> 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) } for row
in rows ]
>>> portfolio
[{'price': '91.10', 'name': 'IBM', 'shares': '50'}, {'price': '83.44', 'name': 'CAT',
'shares': '150'},
  {'price': '51.23', 'name': 'MSFT', 'shares': '200'}, {'price': '40.37', 'name': 'GE',
'shares': '95'},
  {'price': '65.10', 'name': 'MSFT', 'shares': '50'}, {'price': '70.44', 'name': 'IBM',
'shares': '100'}]
>>>
```

Oh my, you just reduced much of the `read_portfolio()` function to a single statement.

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)

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.

Assignment

Many operations in Python are related to *assigning* or *storing* values.

```
a = value          # Assignment to a variable
s[n] = value       # Assignment to a list
s.append(value)    # Appending to a list
d['key'] = value   # Adding to a dictionary
```

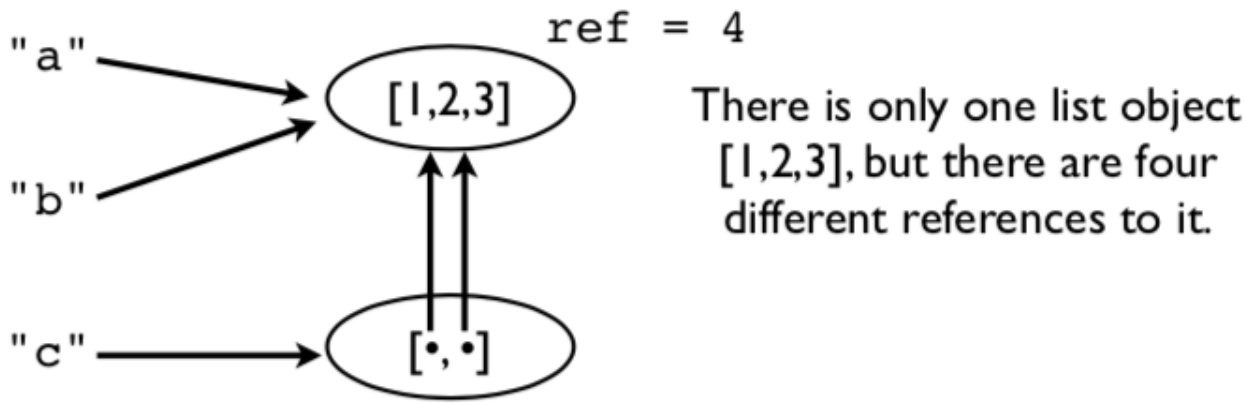
*A caution: assignment operations **never make a copy** of the value being assigned.* All assignments are merely reference copies (or pointer copies if you prefer).

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.



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.

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.**

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).

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
>>>
```

Shallow copies

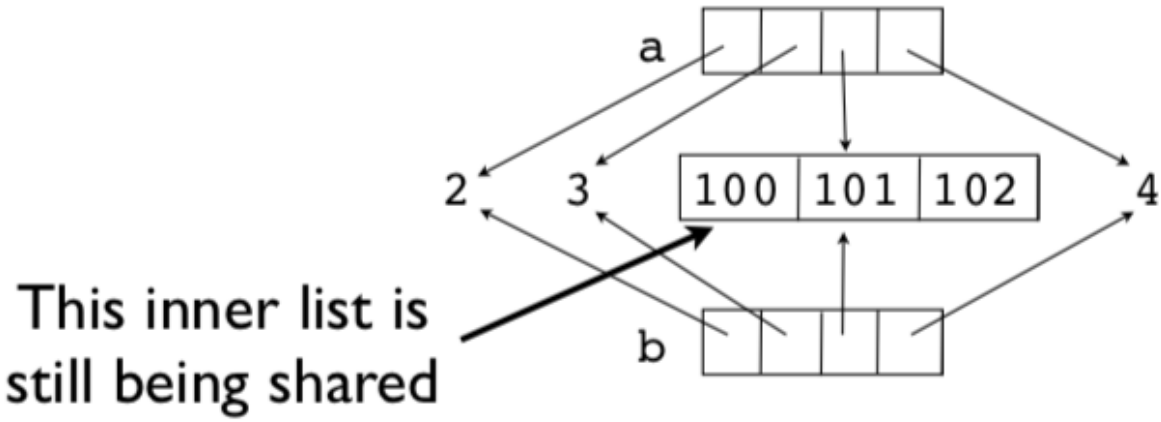
Lists and dicts have methods for copying.

```
>>> 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.



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
>>>
```

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 = 'Hello World'
>>> 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.

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. *

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.

Exercises

In this set of exercises, we look at some of the power that comes from first-class objects.

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]
>>>
```

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}
>>>
```

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)

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 | Previous (2.7 Object Model) | Next (3.2 More on Functions)

3.1 Scripting

In this part we look more closely at the practice of writing Python scripts.

What is a Script?

A *script* is a program that runs a series of statements and stops.

```
# program.py

statement1
statement2
statement3
...
```

We have mostly been writing scripts to this point.

A Problem

If you write a useful script, it will grow in features and functionality. You may want to apply it to other related problems. Over time, it might become a critical application. And if you don't take care, it might turn into a huge tangled mess. So, let's get organized.

Defining Things

Names must always be defined before they get used later.

```
def square(x):
    return x*x

a = 42
b = a + 2     # Requires that `a` is defined

z = square(b) # Requires `square` and `b` to be defined
```

The order is important. You almost always put the definitions of variables and functions near the top.

Defining Functions

It is a good idea to put all of the code related to a single *task* all in one place. Use a function.

```
def read_prices(filename):
    prices = {}
    with open(filename) as f:
        f_csv = csv.reader(f)
        for row in f_csv:
            prices[row[0]] = float(row[1])
    return prices
```

A function also simplifies repeated operations.

```
oldprices = read_prices('oldprices.csv')
newprices = read_prices('newprices.csv')
```

What is a Function?

A function is a named sequence of statements.

```
def funcname(args):
  statement
  statement
  ...
  return result
```

Any Python statement can be used inside.

```
def foo():
    import math
    print(math.sqrt(2))
    help(math)
```

There are no *special* statements in Python (which makes it easy to remember).

Function Definition

Functions can be *defined* in any order.

```
def foo(x):
    bar(x)

def bar(x):
    statements

# OR
def bar(x):
    statements

def foo(x):
    bar(x)
```

Functions must only be defined prior to actually being *used* (or called) during program execution.

```
foo(3)        # foo must be defined already
```

Stylistically, it is probably more common to see functions defined in a *bottom-up* fashion.

Bottom-up Style

Functions are treated as building blocks. The smaller/simpler blocks go first.

```
# myprogram.py
def foo(x):
    ...

def bar(x):
    ...
    foo(x)          # Defined above
    ...

def spam(x):
    ...
    bar(x)          # Defined above
    ...

spam(42)            # Code that uses the functions appears at the end
```

Later functions build upon earlier functions. Again, this is only a point of style. The only thing that matters in the above program is that the call to `spam(42)` go last.

Function Design

Ideally, functions should be a *black box*. They should only operate on passed inputs and avoid global variables and mysterious side-effects. Your main goals: *Modularity* and *Predictability*.

Doc Strings

It's good practice to include documentation in the form of a doc-string. Doc-strings are strings written immediately after the name of the function. They feed `help()`, IDEs and other tools.

```
def read_prices(filename):
    '''
    Read prices from a CSV file of name,price data
    '''
    prices = {}
    with open(filename) as f:
        f_csv = csv.reader(f)
        for row in f_csv:
            prices[row[0]] = float(row[1])
    return prices
```

A good practice for doc strings is to write a short one sentence summary of what the function does. If more information is needed, include a short example of usage along with a more detailed description of the arguments.

Type Annotations

You can also add optional type hints to function definitions.

```
def read_prices(filename: str) -> dict:
    '''
    Read prices from a CSV file of name,price data
    '''
    prices = {}
    with open(filename) as f:
        f_csv = csv.reader(f)
        for row in f_csv:
            prices[row[0]] = float(row[1])
    return prices
```

The hints do nothing operationally. They are purely informational. However, they may be used by IDEs, code checkers, and other tools to do more.

Exercises

In section 2, you wrote a program called `report.py` that printed out a report showing the performance of a stock portfolio. This program consisted of some functions. For example:

```
# report.py
import csv

def read_portfolio(filename):
    '''
    Read a stock portfolio file into a list of dictionaries with keys
    name, shares, and price.
    '''
    portfolio = []
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)

        for row in rows:
            record = dict(zip(headers, row))
            stock = {
                'name' : record['name'],
                'shares' : int(record['shares']),
                'price' : float(record['price'])
            }
            portfolio.append(stock)
    return portfolio
...
```

However, there were also portions of the program that just performed a series of scripted calculations. This code appeared near the end of the program. For example:

```
...

# Output the report

headers = ('Name', 'Shares', 'Price', 'Change')
print('%10s %10s %10s %10s'  % headers)
print(('-' * 10 + ' ') * len(headers))
for row in report:
    print('%10s %10d %10.2f %10.2f' % row)
...
```

In this exercise, we're going take this program and organize it a little more strongly around the use of functions.

Exercise 3.1: Structuring a program as a collection of functions

Modify your `report.py` program so that all major operations, including calculations and output, are carried out by a collection of functions. Specifically:

- Create a function `print_report(report)` that prints out the report.
- Change the last part of the program so that it is nothing more than a series of function calls and no other computation.

Exercise 3.2: Creating a top-level function for program execution

Take the last part of your program and package it into a single function `portfolio_report(portfolio_filename, prices_filename)`. Have the function work so that the following function call creates the report as before:

```
portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
```

In this final version, your program will be nothing more than a series of function definitions followed by a single function call to `portfolio_report()` at the very end (which executes all of the steps involved in the program).

By turning your program into a single function, it becomes easy to run it on different inputs. For example, try these statements interactively after running your program:

```
>>> portfolio_report('Data/portfolio2.csv', 'Data/prices.csv')
... look at the output ...
>>> files = ['Data/portfolio.csv', 'Data/portfolio2.csv']
>>> for name in files:
        print(f'{name:-^43s}')
        portfolio_report(name, 'Data/prices.csv')
        print()

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

Commentary

Python makes it very easy to write relatively unstructured scripting code where you just have a file with a sequence of statements in it. In the big picture, it's almost always better to utilize functions whenever you can. At some point, that script is going to grow and you'll wish you had a bit more organization. Also, a little known fact is that Python runs a bit faster if you use functions.

Contents | Previous (2.7 Object Model) | Next (3.2 More on Functions)

Contents | Previous (3.1 Scripting) | Next (3.3 Error Checking)

3.2 More on Functions

Although functions were introduced earlier, very few details were provided on how they actually work at a deeper level. This section aims to fill in some gaps and discuss matters such as calling conventions, scoping rules, and more.

Calling a Function

Consider this function:

```
def read_prices(filename, debug):
    ...
```

You can call the function with positional arguments:

```
prices = read_prices('prices.csv', True)
```

Or you can call the function with keyword arguments:

```
prices = read_prices(filename='prices.csv', debug=True)
```

Default Arguments

Sometimes you want an argument to be optional. If so, assign a default value in the function definition.

```
def read_prices(filename, debug=False):
    ...
```

If a default value is assigned, the argument is optional in function calls.

```
d = read_prices('prices.csv')
e = read_prices('prices.dat', True)
```

Note: Arguments with defaults must appear at the end of the arguments list (all non-optional arguments go first).

Prefer keyword arguments for optional arguments

Compare and contrast these two different calling styles:

```
parse_data(data, False, True) # ?????

parse_data(data, ignore_errors=True)
parse_data(data, debug=True)
parse_data(data, debug=True, ignore_errors=True)
```

In most cases, keyword arguments improve code clarity--especially for arguments that serve as flags or which are related to optional features.

Design Best Practices

Always give short, but meaningful names to functions arguments.

Someone using a function may want to use the keyword calling style.

```
d = read_prices('prices.csv', debug=True)
```

Python development tools will show the names in help features and documentation.

Returning Values

The `return` statement returns a value

```
def square(x):
    return x * x
```

If no return value is given or `return` is missing, `None` is returned.

```
def bar(x):
    statements
    return

a = bar(4)      # a = None

# OR
def foo(x):
    statements  # No `return`

b = foo(4)      # b = None
```

Multiple Return Values

Functions can only return one value. However, a function may return multiple values by returning them in a tuple.

```
def divide(a,b):
    q = a // b      # Quotient
    r = a % b       # Remainder
    return q, r     # Return a tuple
```

Usage example:

```
x, y = divide(37,5) # x = 7, y = 2

x = divide(37, 5)   # x = (7, 2)
```

Variable Scope

Programs assign values to variables.

```
x = value # Global variable

def foo():
    y = value # Local variable
```

Variables assignments occur outside and inside function definitions. Variables defined outside are "global". Variables inside a function are "local".

Local Variables

Variables assigned inside functions are private.

```
def read_portfolio(filename):
    portfolio = []
    for line in open(filename):
        fields = line.split(',')
        s = (fields[0], int(fields[1]), float(fields[2]))
        portfolio.append(s)
    return portfolio
```

In this example, `filename`, `portfolio`, `line`, `fields` and `s` are local variables. Those variables are not retained or accessible after the function call.

```
>>> stocks = read_portfolio('portfolio.csv')
>>> fields
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'fields' is not defined
>>>
```

Locals also can't conflict with variables found elsewhere.

Global Variables

Functions can freely access the values of globals defined in the same file.

```
name = 'Dave'

def greeting():
    print('Hello', name)  # Using `name` global variable
```

However, functions can't modify globals:

```
name = 'Dave'

def spam():
  name = 'Guido'

spam()
print(name) # prints 'Dave'
```

Remember: All assignments in functions are local.

Modifying Globals

If you must modify a global variable you must declare it as such.

```
name = 'Dave'

def spam():
    global name
    name = 'Guido' # Changes the global name above
```

The global declaration must appear before its use and the corresponding variable must exist in the same file as the function. Having seen this, know that it is considered poor form. In fact, try to avoid `global` entirely if you can. If you need a function to modify some kind of state outside of the function, it's better to use a class instead (more on this later).

Argument Passing

When you call a function, the argument variables are names that refer to the passed values. These values are NOT copies (see section 2.7). If mutable data types are passed (e.g. lists, dicts), they can be modified *in-place*.

```
def foo(items):
    items.append(42)    # Modifies the input object

a = [1, 2, 3]
foo(a)
print(a)                # [1, 2, 3, 42]
```

Key point: Functions don't receive a copy of the input arguments.

Reassignment vs Modifying

Make sure you understand the subtle difference between modifying a value and reassigning a variable name.

```
def foo(items):
    items.append(42)    # Modifies the input object

a = [1, 2, 3]
foo(a)
print(a)                # [1, 2, 3, 42]

# VS
def bar(items):
    items = [4,5,6]    # Changes local `items` variable to point to a different object

b = [1, 2, 3]
bar(b)
print(b)                # [1, 2, 3]
```

Reminder: Variable assignment never overwrites memory. The name is merely bound to a new value.

Exercises

This set of exercises have you implement what is, perhaps, the most powerful and difficult part of the course. There are a lot of steps and many concepts from past exercises are put together all at once. The final solution is only about 25 lines of code, but take your time and make sure you understand each part.

A central part of your `report.py` program focuses on the reading of CSV files. For example, the function `read_portfolio()` reads a file containing rows of portfolio data and the function `read_prices()` reads a file containing rows of price data. In both of those functions, there are a lot of low-level "fiddly" bits and similar features. For example, they both open a file and wrap it with the `csv` module and they both convert various fields into new types.

If you were doing a lot of file parsing for real, you'd probably want to clean some of this up and make it more general purpose. That's our goal.

Start this exercise by opening the file called `Work/fileparse.py`. This is where we will be doing our work.

Exercise 3.3: Reading CSV Files

To start, let's just focus on the problem of reading a CSV file into a list of dictionaries. In the file `fileparse.py`, define a function that looks like this:

```
# fileparse.py
import csv

def parse_csv(filename):
    '''
    Parse a CSV file into a list of records
    '''
    with open(filename) as f:
        rows = csv.reader(f)

        # Read the file headers
        headers = next(rows)
        records = []
```

```
        for row in rows:
            if not row:    # Skip rows with no data
                continue
            record = dict(zip(headers, row))
            records.append(record)

    return records
```

This function reads a CSV file into a list of dictionaries while hiding the details of opening the file, wrapping it with the `csv` module, ignoring blank lines, and so forth.

Try it out:

Hint: `python3 -i fileparse.py`.

```
>>> portfolio = parse_csv('Data/portfolio.csv')
>>> portfolio
[{'price': '32.20', 'name': 'AA', 'shares': '100'}, {'price': '91.10', 'name': 'IBM',
'shares': '50'}, {'price': '83.44', 'name': 'CAT', 'shares': '150'}, {'price': '51.23',
'name': 'MSFT', 'shares': '200'}, {'price': '40.37', 'name': 'GE', 'shares': '95'},
{'price': '65.10', 'name': 'MSFT', 'shares': '50'}, {'price': '70.44', 'name': 'IBM',
'shares': '100'}]
>>>
```

This is good except that you can't do any kind of useful calculation with the data because everything is represented as a string. We'll fix this shortly, but let's keep building on it.

Exercise 3.4: Building a Column Selector

In many cases, you're only interested in selected columns from a CSV file, not all of the data. Modify the `parse_csv()` function so that it optionally allows user-specified columns to be picked out as follows:

```
>>> # Read all of the data
>>> portfolio = parse_csv('Data/portfolio.csv')
>>> portfolio
[{'price': '32.20', 'name': 'AA', 'shares': '100'}, {'price': '91.10', 'name': 'IBM',
'shares': '50'}, {'price': '83.44', 'name': 'CAT', 'shares': '150'}, {'price': '51.23',
'name': 'MSFT', 'shares': '200'}, {'price': '40.37', 'name': 'GE', 'shares': '95'},
{'price': '65.10', 'name': 'MSFT', 'shares': '50'}, {'price': '70.44', 'name': 'IBM',
'shares': '100'}]

>>> # Read only some of the data
>>> shares_held = parse_csv('Data/portfolio.csv', select=['name','shares'])
>>> shares_held
[{'name': 'AA', 'shares': '100'}, {'name': 'IBM', 'shares': '50'}, {'name': 'CAT', 'shares':
'150'}, {'name': 'MSFT', 'shares': '200'}, {'name': 'GE', 'shares': '95'}, {'name': 'MSFT',
'shares': '50'}, {'name': 'IBM', 'shares': '100'}]
>>>
```

An example of a column selector was given in Exercise 2.23. However, here's one way to do it:

```
# fileparse.py
import csv

def parse_csv(filename, select=None):
    '''
    Parse a CSV file into a list of records
    '''
    with open(filename) as f:
        rows = csv.reader(f)

        # Read the file headers
        headers = next(rows)

        # If a column selector was given, find indices of the specified columns.
        # Also narrow the set of headers used for resulting dictionaries
        if select:
            indices = [headers.index(colname) for colname in select]
            headers = select
        else:
            indices = []

        records = []
        for row in rows:
            if not row:    # Skip rows with no data
                continue
            # Filter the row if specific columns were selected
            if indices:
                row = [ row[index] for index in indices ]

            # Make a dictionary
            record = dict(zip(headers, row))
            records.append(record)

    return records
```

There are a number of tricky bits to this part. Probably the most important one is the mapping of the column selections to row indices. For example, suppose the input file had the following headers:

```
>>> headers = ['name', 'date', 'time', 'shares', 'price']
>>>
```

Now, suppose the selected columns were as follows:

```
>>> select = ['name', 'shares']
>>>
```

To perform the proper selection, you have to map the selected column names to column indices in the file. That's what this step is doing:

```
>>> indices = [headers.index(colname) for colname in select ]
>>> indices
[0, 3]
>>>
```

In other words, "name" is column 0 and "shares" is column 3. When you read a row of data from the file, the indices are used to filter it:

```
>>> row = ['AA', '6/11/2007', '9:50am', '100', '32.20' ]
>>> row = [ row[index] for index in indices ]
>>> row
['AA', '100']
>>>
```

Exercise 3.5: Performing Type Conversion

Modify the `parse_csv()` function so that it optionally allows type-conversions to be applied to the returned data. For example:

```
>>> portfolio = parse_csv('Data/portfolio.csv', types=[str, int, float])
>>> portfolio
[{'price': 32.2, 'name': 'AA', 'shares': 100}, {'price': 91.1, 'name': 'IBM', 'shares': 50},
{'price': 83.44, 'name': 'CAT', 'shares': 150}, {'price': 51.23, 'name': 'MSFT', 'shares':
200}, {'price': 40.37, 'name': 'GE', 'shares': 95}, {'price': 65.1, 'name': 'MSFT',
'shares': 50}, {'price': 70.44, 'name': 'IBM', 'shares': 100}]

>>> shares_held = parse_csv('Data/portfolio.csv', select=['name', 'shares'], types=[str,
int])
>>> shares_held
[{'name': 'AA', 'shares': 100}, {'name': 'IBM', 'shares': 50}, {'name': 'CAT', 'shares':
150}, {'name': 'MSFT', 'shares': 200}, {'name': 'GE', 'shares': 95}, {'name': 'MSFT',
'shares': 50}, {'name': 'IBM', 'shares': 100}]
>>>
```

You already explored this in Exercise 2.24. You'll need to insert the following fragment of code into your solution:

```
...
if types:
    row = [func(val) for func, val in zip(types, row) ]
...
```

Exercise 3.6: Working without Headers

Some CSV files don't include any header information. For example, the file `prices.csv` looks like this:

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

Modify the `parse_csv()` function so that it can work with such files by creating a list of tuples instead. For example:

```
>>> prices = parse_csv('Data/prices.csv', types=[str,float], has_headers=False)
>>> prices
[('AA', 9.22), ('AXP', 24.85), ('BA', 44.85), ('BAC', 11.27), ('C', 3.72), ('CAT', 35.46),
('CVX', 66.67), ('DD', 28.47), ('DIS', 24.22), ('GE', 13.48), ('GM', 0.75), ('HD', 23.16),
('HPQ', 34.35), ('IBM', 106.28), ('INTC', 15.72), ('JNJ', 55.16), ('JPM', 36.9), ('KFT',
26.11), ('KO', 49.16), ('MCD', 58.99), ('MMM', 57.1), ('MRK', 27.58), ('MSFT', 20.89),
('PFE', 15.19), ('PG', 51.94), ('T', 24.79), ('UTX', 52.61), ('VZ', 29.26), ('WMT', 49.74),
('XOM', 69.35)]
>>>
```

To make this change, you'll need to modify the code so that the first line of data isn't interpreted as a header line. Also, you'll need to make sure you don't create dictionaries as there are no longer any column names to use for keys.

Exercise 3.7: Picking a different column delimiter

Although CSV files are pretty common, it's also possible that you could encounter a file that uses a different column separator such as a tab or space. For example, the file `Data/portfolio.dat` looks like this:

```
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
```

The `csv.reader()` function allows a different column delimiter to be given as follows:

```
rows = csv.reader(f, delimiter=' ')
```

Modify your `parse_csv()` function so that it also allows the delimiter to be changed.

For example:

```
>>> portfolio = parse_csv('Data/portfolio.dat', types=[str, int, float], delimiter=' ')
>>> portfolio
[{'price': '32.20', 'name': 'AA', 'shares': '100'}, {'price': '91.10', 'name': 'IBM',
'shares': '50'}, {'price': '83.44', 'name': 'CAT', 'shares': '150'}, {'price': '51.23',
'name': 'MSFT', 'shares': '200'}, {'price': '40.37', 'name': 'GE', 'shares': '95'},
{'price': '65.10', 'name': 'MSFT', 'shares': '50'}, {'price': '70.44', 'name': 'IBM',
'shares': '100'}]
>>>
```

Commentary

If you've made it this far, you've created a nice library function that's genuinely useful. You can use it to parse arbitrary CSV files, select out columns of interest, perform type conversions, without having to worry too much about the inner workings of files or the `csv` module.

Contents | Previous (3.1 Scripting) | Next (3.3 Error Checking)

Contents | Previous (3.2 More on Functions) | Next (3.4 Modules)

3.3 Error Checking

Although exceptions were introduced earlier, this section fills in some additional details about error checking and exception handling.

How programs fail

Python performs no checking or validation of function argument types or values. A function will work on any data that is compatible with the statements in the function.

```
def add(x, y):
    return x + y

add(3, 4)               # 7
add('Hello', 'World')   # 'HelloWorld'
add('3', '4')           # '34'
```

If there are errors in a function, they appear at run time (as an exception).

```
def add(x, y):
    return x + y

>>> add(3, '4')
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +:
'int' and 'str'
>>>
```

To verify code, there is a strong emphasis on testing (covered later).

Exceptions

Exceptions are used to signal errors. To raise an exception yourself, use `raise` statement.

```
if name not in authorized:
    raise RuntimeError(f'{name} not authorized')
```

To catch an exception use `try-except`.

```
try:
    authenticate(username)
except RuntimeError as e:
    print(e)
```

Exception Handling

Exceptions propagate to the first matching `except`.

```
def grok():
    ...
    raise RuntimeError('Whoa!')   # Exception raised here

def spam():
    grok()                        # Call that will raise exception

def bar():
    try:
       spam()
    except RuntimeError as e:     # Exception caught here
        ...

def foo():
    try:
         bar()
    except RuntimeError as e:     # Exception does NOT arrive here
        ...

foo()
```

To handle the exception, put statements in the `except` block. You can add any statements you want to handle the error.

```
def grok(): ...
    raise RuntimeError('Whoa!')

def bar():
    try:
      grok()
    except RuntimeError as e:   # Exception caught here
        statements              # Use this statements
        statements
        ...

bar()
```

After handling, execution resumes with the first statement after the `try-except`.

```
def grok(): ...
    raise RuntimeError('Whoa!')

def bar():
    try:
      grok()
    except RuntimeError as e:   # Exception caught here
        statements
        statements
        ...
    statements                  # Resumes execution here
    statements                  # And continues here
    ...

bar()
```

Built-in Exceptions

There are about two-dozen built-in exceptions. Usually the name of the exception is indicative of what's wrong (e.g., a `ValueError` is raised because you supplied a bad value). This is not an exhaustive list. Check the documentation for more.

```
ArithmeticError
AssertionError
EnvironmentError
EOFError
ImportError
IndexError
KeyboardInterrupt
KeyError
MemoryError
NameError
ReferenceError
RuntimeError
SyntaxError
SystemError
```

```
TypeError
ValueError
```

Exception Values

Exceptions have an associated value. It contains more specific information about what's wrong.

```
raise RuntimeError('Invalid user name')
```

This value is part of the exception instance that's placed in the variable supplied to `except`.

```
try:
    ...
except RuntimeError as e:   # `e` holds the exception raised
    ...
```

`e` is an instance of the exception type. However, it often looks like a string when printed.

```
except RuntimeError as e:
    print('Failed : Reason', e)
```

Catching Multiple Errors

You can catch different kinds of exceptions using multiple `except` blocks.

```
try:
  ...
except LookupError as e:
  ...
except RuntimeError as e:
  ...
except IOError as e:
  ...
except KeyboardInterrupt as e:
  ...
```

Alternatively, if the statements to handle them is the same, you can group them:

```
try:
  ...
except (IOError,LookupError,RuntimeError) as e:
  ...
```

Catching All Errors

To catch any exception, use `Exception` like this:

```
try:
    ...
except Exception:       # DANGER. See below
    print('An error occurred')
```

In general, writing code like that is a bad idea because you'll have no idea why it failed.

Wrong Way to Catch Errors

Here is the wrong way to use exceptions.

```
try:
    go_do_something()
except Exception:
    print('Computer says no')
```

This catches all possible errors and it may make it impossible to debug when the code is failing for some reason you didn't expect at all (e.g. uninstalled Python module, etc.).

Somewhat Better Approach

If you're going to catch all errors, this is a more sane approach.

```
try:
    go_do_something()
except Exception as e:
    print('Computer says no. Reason :', e)
```

It reports a specific reason for failure. It is almost always a good idea to have some mechanism for viewing/reporting errors when you write code that catches all possible exceptions.

In general though, it's better to catch the error as narrowly as is reasonable. Only catch the errors you can actually handle. Let other errors pass by--maybe some other code can handle them.

Reraising an Exception

Use `raise` to propagate a caught error.

```
try:
    go_do_something()
except Exception as e:
    print('Computer says no. Reason :', e)
    raise
```

This allows you to take action (e.g. logging) and pass the error on to the caller.

Exception Best Practices

Don't catch exceptions. Fail fast and loud. If it's important, someone else will take care of the problem. Only catch an exception if you are *that* someone. That is, only catch errors where you can recover and sanely keep going.

`finally` statement

It specifies code that must run regardless of whether or not an exception occurs.

```
lock = Lock()
...
lock.acquire()
try:
    ...
finally:
    lock.release()  # this will ALWAYS be executed. With and without exception.
```

Commonly used to safely manage resources (especially locks, files, etc.).

`with` statement

In modern code, `try-finally` is often replaced with the `with` statement.

```
lock = Lock()
with lock:
    # lock acquired
    ...
# lock released
```

A more familiar example:

```
with open(filename) as f:
    # Use the file
    ...
# File closed
```

`with` defines a usage *context* for a resource. When execution leaves that context, resources are released. `with` only works with certain objects that have been specifically programmed to support it.

Exercises

Exercise 3.8: Raising exceptions

The `parse_csv()` function you wrote in the last section allows user-specified columns to be selected, but that only works if the input data file has column headers.

Modify the code so that an exception gets raised if both the `select` and `has_headers=False` arguments are passed. For example:

```
>>> parse_csv('Data/prices.csv', select=['name','price'], has_headers=False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fileparse.py", line 9, in parse_csv
    raise RuntimeError("select argument requires column headers")
RuntimeError: select argument requires column headers
>>>
```

Having added this one check, you might ask if you should be performing other kinds of sanity checks in the function. For example, should you check that the filename is a string, that types is a list, or anything of that nature?

As a general rule, it's usually best to skip such tests and to just let the program fail on bad inputs. The traceback message will point at the source of the problem and can assist in debugging.

The main reason for adding the above check is to avoid running the code in a non-sensical mode (e.g., using a feature that requires column headers, but simultaneously specifying that there are no headers).

This indicates a programming error on the part of the calling code. Checking for cases that "aren't supposed to happen" is often a good idea.

Exercise 3.9: Catching exceptions

The `parse_csv()` function you wrote is used to process the entire contents of a file. However, in the real-world, it's possible that input files might have corrupted, missing, or dirty data. Try this experiment:

```
>>> portfolio = parse_csv('Data/missing.csv', types=[str, int, float])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fileparse.py", line 36, in parse_csv
    row = [func(val) for func, val in zip(types, row)]
ValueError: invalid literal for int() with base 10: ''
>>>
```

Modify the `parse_csv()` function to catch all `ValueError` exceptions generated during record creation and print a warning message for rows that can't be converted.

The message should include the row number and information about the reason why it failed. To test your function, try reading the file `Data/missing.csv` above. For example:

```
>>> portfolio = parse_csv('Data/missing.csv', types=[str, int, float])
Row 4: Couldn't convert ['MSFT', '', '51.23']
Row 4: Reason invalid literal for int() with base 10: ''
Row 7: Couldn't convert ['IBM', '', '70.44']
Row 7: Reason invalid literal for int() with base 10: ''
>>>
>>> portfolio
[{'price': 32.2, 'name': 'AA', 'shares': 100}, {'price': 91.1, 'name': 'IBM', 'shares': 50},
{'price': 83.44, 'name': 'CAT', 'shares': 150}, {'price': 40.37, 'name': 'GE', 'shares':
95}, {'price': 65.1, 'name': 'MSFT', 'shares': 50}]
>>>
```

Exercise 3.10: Silencing Errors

Modify the `parse_csv()` function so that parsing error messages can be silenced if explicitly desired by the user. For example:

```
>>> portfolio = parse_csv('Data/missing.csv', types=[str,int,float], silence_errors=True)
>>> portfolio
[{'price': 32.2, 'name': 'AA', 'shares': 100}, {'price': 91.1, 'name': 'IBM', 'shares': 50},
{'price': 83.44, 'name': 'CAT', 'shares': 150}, {'price': 40.37, 'name': 'GE', 'shares':
95}, {'price': 65.1, 'name': 'MSFT', 'shares': 50}]
>>>
```

Error handling is one of the most difficult things to get right in most programs. As a general rule, you shouldn't silently ignore errors. Instead, it's better to report problems and to give the user an option to the silence the error message if they choose to do so.

Contents | Previous (3.2 More on Functions) | Next (3.4 Modules)

Contents | Previous (3.3 Error Checking) | Next (3.5 Main Module)

3.4 Modules

This section introduces the concept of modules and working with functions that span multiple files.

Modules and import

Any Python source file is a module.

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

The `import` statement loads and *executes* a module.

```
# program.py
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...
```

Namespaces

A module is a collection of named values and is sometimes said to be a *namespace*. The names are all of the global variables and functions defined in the source file. After importing, the module name is used as a prefix. Hence the *namespace*.

```
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...
```

The module name is directly tied to the file name (foo -> foo.py).

Global Definitions

Everything defined in the *global* scope is what populates the module namespace. Consider two modules that define the same variable `x`.

```
# foo.py
x = 42
def grok(a):
    ...
```

```
# bar.py
x = 37
def spam(a):
    ...
```

In this case, the `x` definitions refer to different variables. One is `foo.x` and the other is `bar.x`. Different modules can use the same names and those names won't conflict with each other.

Modules are isolated.

Modules as Environments

Modules form an enclosing environment for all of the code defined inside.

```
# foo.py
x = 42

def grok(a):
    print(x)
```

Global variables are always bound to the enclosing module (same file). Each source file is its own little universe.

Module Execution

When a module is imported, *all of the statements in the module execute* one after another until the end of the file is reached. The contents of the module namespace are all of the *global* names that are still defined at the end of the execution process. If there are scripting statements that carry out tasks in the global scope (printing, creating files, etc.) you will see them run on import.

`import as` statement

You can change the name of a module as you import it:

```
import math as m
def rectangular(r, theta):
    x = r * m.cos(theta)
    y = r * m.sin(theta)
    return x, y
```

It works the same as a normal import. It just renames the module in that one file.

`from` module import

This picks selected symbols out of a module and makes them available locally.

```
from math import sin, cos

def rectangular(r, theta):
    x = r * cos(theta)
    y = r * sin(theta)
    return x, y
```

This allows parts of a module to be used without having to type the module prefix. It's useful for frequently used names.

Comments on importing

Variations on import do *not* change the way that modules work.

```
import math
# vs
import math as m
# vs
from math import cos, sin
...
```

Specifically, `import` always executes the *entire* file and modules are still isolated environments.

The `import module as` statement is only changing the name locally. The `from math import cos, sin` statement still loads the entire math module behind the scenes. It's merely copying the `cos` and `sin` names from the module into the local space after it's done.

Module Loading

Each module loads and executes only *once*. *Note: Repeated imports just return a reference to the previously loaded module.*

`sys.modules` is a dict of all loaded modules.

```
>>> import sys
>>> sys.modules.keys()
['copy_reg', '__main__', 'site', '__builtin__', 'encodings', 'encodings.encodings',
'posixpath', ...]
>>>
```

Caution: A common confusion arises if you repeat an `import` statement after changing the source code for a module. Because of the module cache `sys.modules`, repeated imports always return the previously loaded module--even if a change was made. The safest way to load modified code into Python is to quit and restart the interpreter.

Locating Modules

Python consults a path list (sys.path) when looking for modules.

```
>>> import sys
>>> sys.path
[
  '',
  '/usr/local/lib/python36/python36.zip',
  '/usr/local/lib/python36',
  ...
]
```

The current working directory is usually first.

Module Search Path

As noted, `sys.path` contains the search paths. You can manually adjust if you need to.

```
import sys
sys.path.append('/project/foo/pyfiles')
```

Paths can also be added via environment variables.

```
% env PYTHONPATH=/project/foo/pyfiles python3
Python 3.6.0 (default, Feb 3 2017, 05:53:21)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)]
>>> import sys
>>> sys.path
['','/project/foo/pyfiles', ...]
```

As a general rule, it should not be necessary to manually adjust the module search path. However, it sometimes arises if you're trying to import Python code that's in an unusual location or not readily accessible from the current working directory.

Exercises

For this exercise involving modules, it is critically important to make sure you are running Python in a proper environment. Modules often present new programmers with problems related to the current working directory or with Python's path settings. For this course, it is assumed that you're writing all of your code in the `Work/` directory. For best results, you should make sure you're also in that directory when you launch the interpreter. If not, you need to make sure `practical-python/Work` is added to `sys.path`.

Exercise 3.11: Module imports

In section 3, we created a general purpose function `parse_csv()` for parsing the contents of CSV datafiles.

Now, we're going to see how to use that function in other programs. First, start in a new shell window. Navigate to the folder where you have all your files. We are going to import them.

Start Python interactive mode.

```
bash % python3
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Once you've done that, try importing some of the programs you previously wrote. You should see their output exactly as before. Just to emphasize, importing a module runs its code.

```
>>> import bounce
... watch output ...
>>> import mortgage
... watch output ...
>>> import report
... watch output ...
>>>
```

If none of this works, you're probably running Python in the wrong directory. Now, try importing your `fileparse` module and getting some help on it.

```
>>> import fileparse
>>> help(fileparse)
... look at the output ...
>>> dir(fileparse)
... look at the output ...
>>>
```

Try using the module to read some data:

```
>>> portfolio = fileparse.parse_csv('Data/portfolio.csv',select=['name','shares','price'],
types=[str,int,float])
>>> portfolio
... look at the output ...
>>> pricelist = fileparse.parse_csv('Data/prices.csv',types=[str,float], has_headers=False)
>>> pricelist
... look at the output ...
>>> prices = dict(pricelist)
>>> prices
... look at the output ...
>>> prices['IBM']
106.11
>>>
```

Try importing a function so that you don't need to include the module name:

```
>>> from fileparse import parse_csv
>>> portfolio = parse_csv('Data/portfolio.csv', select=['name','shares','price'], types=
[str,int,float])
>>> portfolio
... look at the output ...
>>>
```

Exercise 3.12: Using your library module

In section 2, you wrote a program `report.py` that produced a stock report 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
```

Take that program and modify it so that all of the input file processing is done using functions in your `fileparse` module. To do that, import `fileparse` as a module and change the `read_portfolio()` and `read_prices()` functions to use the `parse_csv()` function.

Use the interactive example at the start of this exercise as a guide. Afterwards, you should get exactly the same output as before.

Exercise 3.13: Intentionally left blank (skip)

Exercise 3.14: Using more library imports

In section 1, you wrote a program `pcost.py` that read a portfolio and computed its cost.

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

Modify the `pcost.py` file so that it uses the `report.read_portfolio()` function.

Commentary

When you are done with this exercise, you should have three programs. `fileparse.py` which contains a general purpose `parse_csv()` function. `report.py` which produces a nice report, but also contains `read_portfolio()` and `read_prices()` functions. And finally, `pcost.py` which computes the portfolio cost, but makes use of the `read_portfolio()` function written for the `report.py` program.

Contents | Previous (3.3 Error Checking) | Next (3.5 Main Module)

Contents | Previous (3.4 Modules) | Next (3.6 Design Discussion)

3.5 Main Module

This section introduces the concept of a main program or main module.

Main Functions

In many programming languages, there is a concept of a *main* function or method.

```
// c / c++
int main(int argc, char *argv[]) {
    ...
}
```

```
// java
class myprog {
    public static void main(String args[]) {
        ...
    }
}
```

This is the first function that executes when an application is launched.

Python Main Module

Python has no *main* function or method. Instead, there is a *main* module. The *main module* is the source file that runs first.

```
bash % python3 prog.py
...
```

Whatever file you give to the interpreter at startup becomes *main*. It doesn't matter the name.

`__main__` check

It is standard practice for modules that run as a main script to use this convention:

```
# prog.py
...
if __name__ == '__main__':
    # Running as the main program ...
    statements
    ...
```

Statements enclosed inside the `if` statement become the *main* program.

Main programs vs. library imports

Any Python file can either run as main or as a library import:

```
bash % python3 prog.py # Running as main
```

```
import prog   # Running as library import
```

In both cases, `__name__` is the name of the module. However, it will only be set to `__main__` if running as main.

Usually, you don't want statements that are part of the main program to execute on a library import. So, it's common to have an `if-` check in code that might be used either way.

```
if __name__ == '__main__':
    # Does not execute if loaded with import ...
```

Program Template

Here is a common program template for writing a Python program:

```
# prog.py
# Import statements (libraries)
import modules

# Functions
def spam():
    ...

def blah():
    ...

# Main function
def main():
    ...
```

```
if __name__ == '__main__':
    main()
```

Command Line Tools

Python is often used for command-line tools

```
bash % python3 report.py portfolio.csv prices.csv
```

It means that the scripts are executed from the shell / terminal. Common use cases are for automation, background tasks, etc.

Command Line Args

The command line is a list of text strings.

```
bash % python3 report.py portfolio.csv prices.csv
```

This list of text strings is found in `sys.argv`.

```
# In the previous bash command
sys.argv # ['report.py, 'portfolio.csv', 'prices.csv']
```

Here is a simple example of processing the arguments:

```
import sys

if len(sys.argv) != 3:
    raise SystemExit(f'Usage: {sys.argv[0]} ' 'portfile pricefile')
portfile = sys.argv[1]
pricefile = sys.argv[2]
...
```

Standard I/O

Standard Input / Output (or stdio) are files that work the same as normal files.

```
sys.stdout
sys.stderr
sys.stdin
```

By default, print is directed to `sys.stdout`. Input is read from `sys.stdin`. Tracebacks and errors are directed to `sys.stderr`.

Be aware that *stdio* could be connected to terminals, files, pipes, etc.

```
bash % python3 prog.py > results.txt
# or
bash % cmd1 | python3 prog.py | cmd2
```

Environment Variables

Environment variables are set in the shell.

```
bash % setenv NAME dave
bash % setenv RSH ssh
bash % python3 prog.py
```

`os.environ` is a dictionary that contains these values.

```
import os

name = os.environ['NAME'] # 'dave'
```

Changes are reflected in any subprocesses later launched by the program.

Program Exit

Program exit is handled through exceptions.

```
raise SystemExit
raise SystemExit(exitcode)
raise SystemExit('Informative message')
```

An alternative.

```
import sys
sys.exit(exitcode)
```

A non-zero exit code indicates an error.

The `#!` line

On Unix, the `#!` line can launch a script as Python. Add the following to the first line of your script file.

```
#!/usr/bin/env python3
# prog.py
...
```

It requires the executable permission.

```
bash % chmod +x prog.py
# Then you can execute
bash % prog.py
... output ...
```

Note: The Python Launcher on Windows also looks for the `#!` line to indicate language version.

Script Template

Finally, here is a common code template for Python programs that run as command-line scripts:

```
#!/usr/bin/env python3
# prog.py

# Import statements (libraries)
import modules

# Functions
def spam():
    ...

def blah():
    ...

# Main function
def main(argv):
    # Parse command line args, environment, etc.
    ...

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

Exercises

Exercise 3.15: `main()` functions

In the file `report.py` add a `main()` function that accepts a list of command line options and produces the same output as before. You should be able to run it interactively like this:

```
>>> import report
>>> report.main(['report.py', 'Data/portfolio.csv', 'Data/prices.csv'])
      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
>>>
```

Modify the `pcost.py` file so that it has a similar `main()` function:

```
>>> import pcost
>>> pcost.main(['pcost.py', 'Data/portfolio.csv'])
Total cost: 44671.15
>>>
```

Exercise 3.16: Making Scripts

Modify the `report.py` and `pcost.py` programs so that they can execute as a script on the command line:

```
bash $ python3 report.py Data/portfolio.csv Data/prices.csv
      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

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

Contents | Previous (3.4 Modules) | Next (3.6 Design Discussion)

Contents | Previous (3.5 Main module) | Next (4 Classes)

3.6 Design Discussion

In this section we reconsider a design decision made earlier.

Filenames versus Iterables

Compare these two programs that return the same output.

```
# Provide a filename
def read_data(filename):
    records = []
    with open(filename) as f:
        for line in f:
            ...
            records.append(r)
    return records

d = read_data('file.csv')
```

```
# Provide lines
def read_data(lines):
    records = []
    for line in lines:
        ...
        records.append(r)
    return records

with open('file.csv') as f:
    d = read_data(f)
```

- Which of these functions do you prefer? Why?
- Which of these functions is more flexible?

Deep Idea: "Duck Typing"

Duck Typing is a computer programming concept to determine whether an object can be used for a particular purpose. It is an application of the duck test.

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

In the second version of `read_data()` above, the function expects any iterable object. Not just the lines of a file.

```
def read_data(lines):
    records = []
    for line in lines:
        ...
        records.append(r)
    return records
```

This means that we can use it with other *lines*.

```
# A CSV file
lines = open('data.csv')
data = read_data(lines)

# A zipped file
lines = gzip.open('data.csv.gz','rt')
data = read_data(lines)
```

```
# The Standard Input
lines = sys.stdin
data = read_data(lines)

# A list of strings
lines = ['ACME,50,91.1','IBM,75,123.45', ... ]
data = read_data(lines)
```

There is considerable flexibility with this design.

Question: Should we embrace or fight this flexibility?

Library Design Best Practices

Code libraries are often better served by embracing flexibility. Don't restrict your options. With great flexibility comes great power.

Exercise

Exercise 3.17: From filenames to file-like objects

You've now created a file `fileparse.py` that contained a function `parse_csv()`. The function worked like this:

```
>>> import fileparse
>>> portfolio = fileparse.parse_csv('Data/portfolio.csv', types=[str,int,float])
>>>
```

Right now, the function expects to be passed a filename. However, you can make the code more flexible. Modify the function so that it works with any file-like/iterable object. For example:

```
>>> import fileparse
>>> import gzip
>>> with gzip.open('Data/portfolio.csv.gz', 'rt') as file:
...      port = fileparse.parse_csv(file, types=[str,int,float])
...
>>> lines = ['name,shares,price', 'AA,100,34.23', 'IBM,50,91.1', 'HPE,75,45.1']
>>> port = fileparse.parse_csv(lines, types=[str,int,float])
>>>
```

In this new code, what happens if you pass a filename as before?

```
>>> port = fileparse.parse_csv('Data/portfolio.csv', types=[str,int,float])
>>> port
... look at output (it should be crazy) ...
>>>
```

Yes, you'll need to be careful. Could you add a safety check to avoid this?

Exercise 3.18: Fixing existing functions

Fix the `read_portfolio()` and `read_prices()` functions in the `report.py` file so that they work with the modified version of `parse_csv()`. This should only involve a minor modification. Afterwards, your `report.py` and `pcost.py` programs should work the same way they always did.

Contents | Previous (3.5 Main module) | Next (4 Classes)

Contents | Prev (3 Program Organization) | Next (5 Inner Workings of Python Objects)

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.

Contents | Prev (3 Program Organization) | Next (5 Inner Workings of Python Objects)

Contents | Previous (3.6 Design discussion) | Next (4.2 Inheritance)

4.1 Classes

This section introduces the class statement and the idea of creating new objects.

Object Oriented (OO) programming

A Programming technique where code is organized as a collection of *objects*.

An *object* consists of:

- Data. Attributes
- Behavior. Methods which are functions applied to the object.

You have already been using some OO during this course.

For example, manipulating a list.

```
>>> nums = [1, 2, 3]
>>> nums.append(4)      # Method
>>> nums.insert(1,10)   # Method
>>> nums
[1, 10, 2, 3, 4]        # Data
>>>
```

`nums` is an *instance* of a list.

Methods (`append()` and `insert()`) are attached to the instance (`nums`).

The `class` statement

Use the `class` statement to define a new object.

```
class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.health = 100

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def damage(self, pts):
        self.health -= pts
```

In a nutshell, a class is a set of functions that carry out various operations on so-called *instances*.

Instances

Instances are the actual *objects* that you manipulate in your program.

They are created by calling the class as a function.

```
>>> a = Player(2, 3)
>>> b = Player(10, 20)
>>>
```

`a` and `b` are instances of `Player`.

Emphasize: The class statement is just the definition (it does nothing by itself). Similar to a function definition.

Instance Data

Each instance has its own local data.

```
>>> a.x
2
>>> b.x
10
```

This data is initialized by the `__init__()`.

```
class Player:
    def __init__(self, x, y):
        # Any value stored on `self` is instance data
        self.x = x
        self.y = y
        self.health = 100
```

There are no restrictions on the total number or type of attributes stored.

Instance Methods

Instance methods are functions applied to instances of an object.

```
class Player:
    ...
    # `move` is a method
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

The object itself is always passed as first argument.

```
>>> a.move(1, 2)

# matches `a` to `self`
# matches `1` to `dx`
# matches `2` to `dy`
def move(self, dx, dy):
```

By convention, the instance is called `self`. However, the actual name used is unimportant. The object is always passed as the first argument. It is merely Python programming style to call this argument `self`.

Class Scoping

Classes do not define a scope of names.

```
class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def left(self, amt):
        move(-amt, 0)       # NO. Calls a global `move` function
        self.move(-amt, 0)  # YES. Calls method `move` from above.
```

If you want to operate on an instance, you always refer to it explicitly (e.g., `self`).

Exercises

Starting with this set of exercises, we start to make a series of changes to existing code from previous sections. It is critical that you have a working version of Exercise 3.18 to start. If you don't have that, please work from the solution code found in the `Solutions/3_18` directory. It's fine to copy it.

Exercise 4.1: Objects as Data Structures

In section 2 and 3, we worked with data represented as tuples and dictionaries. For example, a holding of stock could be represented as a tuple like this:

```
s = ('GOOG',100,490.10)
```

or as a dictionary like this:

```
s = { 'name'   : 'GOOG',
      'shares' : 100,
      'price'  : 490.10
}
```

You can even write functions for manipulating such data. For example:

```
def cost(s):
    return s['shares'] * s['price']
```

However, as your program gets large, you might want to create a better sense of organization. Thus, another approach for representing data would be to define a class. Create a file called `stock.py` and define a class `Stock` that represents a single holding of stock. Have the instances of `Stock` have `name`, `shares`, and `price` attributes. For example:

```
>>> import stock
>>> a = stock.Stock('GOOG',100,490.10)
>>> a.name
'GOOG'
>>> a.shares
100
>>> a.price
490.1
>>>
```

Create a few more `Stock` objects and manipulate them. For example:

```
>>> b = stock.Stock('AAPL', 50, 122.34)
>>> c = stock.Stock('IBM', 75, 91.75)
>>> b.shares * b.price
6117.0
>>> c.shares * c.price
6881.25
>>> stocks = [a, b, c]
>>> stocks
[<stock.Stock object at 0x37d0b0>, <stock.Stock object at 0x37d110>, <stock.Stock object at
```

```
0x37d050>]
>>> for s in stocks:
     print(f'{s.name:>10s} {s.shares:>10d} {s.price:>10.2f}')

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

One thing to emphasize here is that the class `Stock` acts like a factory for creating instances of objects. Basically, you call it as a function and it creates a new object for you. Also, it must be emphasized that each object is distinct---they each have their own data that is separate from other objects that have been created.

An object defined by a class is somewhat similar to a dictionary--just with somewhat different syntax. For example, instead of writing `s['name']` or `s['price']`, you now write `s.name` and `s.price`.

Exercise 4.2: Adding some Methods

With classes, you can attach functions to your objects. These are known as methods and are functions that operate on the data stored inside an object. Add a `cost()` and `sell()` method to your `Stock` object. They should work like this:

```
>>> import stock
>>> s = stock.Stock('GOOG', 100, 490.10)
>>> s.cost()
49010.0
>>> s.shares
100
>>> s.sell(25)
>>> s.shares
75
>>> s.cost()
36757.5
>>>
```

Exercise 4.3: Creating a list of instances

Try these steps to make a list of Stock instances from a list of dictionaries. Then compute the total cost:

```
>>> import fileparse
>>> with open('Data/portfolio.csv') as lines:
...     portdicts = fileparse.parse_csv(lines, select=['name','shares','price'], types=
[str,int,float])
...
>>> portfolio = [ stock.Stock(d['name'], d['shares'], d['price']) for d in portdicts]
>>> portfolio
[<stock.Stock object at 0x10c9e2128>, <stock.Stock object at 0x10c9e2048>, <stock.Stock
object at 0x10c9e2080>,
 <stock.Stock object at 0x10c9e25f8>, <stock.Stock object at 0x10c9e2630>, <stock.Stock
object at 0x10ca6f748>,
 <stock.Stock object at 0x10ca6f7b8>]
>>> sum([s.cost() for s in portfolio])
44671.15
>>>
```

Exercise 4.4: Using your class

Modify the `read_portfolio()` function in the `report.py` program so that it reads a portfolio into a list of `Stock` instances as just shown in Exercise 4.3. Once you have done that, fix all of the code in `report.py` and `pcost.py` so that it works with `Stock` instances instead of dictionaries.

Hint: You should not have to make major changes to the code. You will mainly be changing dictionary access such as `s['shares']` into `s.shares`.

You should be able to run your functions the same as before:

```
>>> import pcost
>>> pcost.portfolio_cost('Data/portfolio.csv')
44671.15
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
      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 (3.6 Design discussion) | Next (4.2 Inheritance)

Contents | Previous (4.1 Classes) | Next (4.3 Special methods)

4.2 Inheritance

Inheritance is a commonly used tool for writing extensible programs. This section explores that idea.

Introduction

Inheritance is used to specialize existing objects:

```
class Parent:
    ...

class Child(Parent):
    ...
```

The new class `Child` is called a derived class or subclass. The `Parent` class is known as base class or superclass. `Parent` is specified in `()` after the class name, `class Child(Parent):`.

Extending

With inheritance, you are taking an existing class and:

- Adding new methods
- Redefining some of the existing methods
- Adding new attributes to instances

In the end you are **extending existing code**.

Example

Suppose that this is your starting class:

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    def cost(self):
        return self.shares * self.price

    def sell(self, nshares):
        self.shares -= nshares
```

You can change any part of this via inheritance.

Add a new method

```
class MyStock(Stock):
    def panic(self):
        self.sell(self.shares)
```

Usage example.

```
>>> s = MyStock('GOOG', 100, 490.1)
>>> s.sell(25)
>>> s.shares
75
>>> s.panic()
>>> s.shares
0
>>>
```

Redefining an existing method

```
class MyStock(Stock):
    def cost(self):
        return 1.25 * self.shares * self.price
```

Usage example.

```
>>> s = MyStock('GOOG', 100, 490.1)
>>> s.cost()
61262.5
>>>
```

The new method takes the place of the old one. The other methods are unaffected. It's tremendous.

Overriding

Sometimes a class extends an existing method, but it wants to use the original implementation inside the redefinition. For this, use `super()`:

```
class Stock:
    ...
    def cost(self):
        return self.shares * self.price
    ...

class MyStock(Stock):
    def cost(self):
        # Check the call to `super`
        actual_cost = super().cost()
        return 1.25 * actual_cost
```

Use `super()` to call the previous version.

Caution: In Python 2, the syntax was more verbose.

```
actual_cost = super(MyStock, self).cost()
```

`__init__` and inheritance

If `__init__` is redefined, it is essential to initialize the parent.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class MyStock(Stock):
    def __init__(self, name, shares, price, factor):
        # Check the call to `super` and `__init__`
        super().__init__(name, shares, price)
        self.factor = factor

    def cost(self):
        return self.factor * super().cost()
```

You should call the `__init__()` method on the `super` which is the way to call the previous version as shown previously.

Using Inheritance

Inheritance is sometimes used to organize related objects.

```
class Shape:
    ...

class Circle(Shape):
    ...

class Rectangle(Shape):
    ...
```

Think of a logical hierarchy or taxonomy. However, a more common (and practical) usage is related to making reusable or extensible code. For example, a framework might define a base class and instruct you to customize it.

```
class CustomHandler(TCPHandler):
    def handle_request(self):
        ...
        # Custom processing
```

The base class contains some general purpose code. Your class inherits and customized specific parts.

"is a" relationship

Inheritance establishes a type relationship.

```
class Shape:
    ...

class Circle(Shape):
    ...
```

Check for object instance.

```
>>> c = Circle(4.0)
>>> isinstance(c, Shape)
True
>>>
```

Important: Ideally, any code that worked with instances of the parent class will also work with instances of the child class.

`object` base class

If a class has no parent, you sometimes see `object` used as the base.

```
class Shape(object):
    ...
```

`object` is the parent of all objects in Python.

*Note: it's not technically required, but you often see it specified as a hold-over from it's required use in Python 2. If omitted, the class still implicitly inherits from `object`.

Multiple Inheritance

You can inherit from multiple classes by specifying them in the definition of the class.

```
class Mother:
    ...

class Father:
    ...

class Child(Mother, Father):
    ...
```

The class `Child` inherits features from both parents. There are some rather tricky details. Don't do it unless you know what you are doing. Some further information will be given in the next section, but we're not going to utilize multiple inheritance further in this course.

Exercises

A major use of inheritance is in writing code that's meant to be extended or customized in various ways--especially in libraries or frameworks. To illustrate, consider the `print_report()` function in your `report.py` program. It should look something like this:

```
def print_report(reportdata):
    '''
    Print a nicely formatted table from a list of (name, shares, price, change) tuples.
    '''
    headers = ('Name','Shares','Price','Change')
    print('%10s %10s %10s %10s' % headers)
    print(('-'*10 + ' ')*len(headers))
    for row in reportdata:
        print('%10s %10d %10.2f %10.2f' % row)
```

When you run your report program, you should be getting output like this:

```
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
      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
```

Exercise 4.5: An Extensibility Problem

Suppose that you wanted to modify the `print_report()` function to support a variety of different output formats such as plain-text, HTML, CSV, or XML. To do this, you could try to write one gigantic function that did everything. However, doing so would likely lead to an unmaintainable mess. Instead, this is a perfect opportunity to use inheritance instead.

To start, focus on the steps that are involved in a creating a table. At the top of the table is a set of table headers. After that, rows of table data appear. Let's take those steps and put them into their own class. Create a file called `tableformat.py` and define the following class:

```
# tableformat.py

class TableFormatter:
    def headings(self, headers):
        '''
        Emit the table headings.
        '''
    raise NotImplementedError()

    def row(self, rowdata):
        '''
        Emit a single row of table data.
```

```
        '''
    raise NotImplementedError()
```

This class does nothing, but it serves as a kind of design specification for additional classes that will be defined shortly. A class like this is sometimes called an "abstract base class."

Modify the `print_report()` function so that it accepts a `TableFormatter` object as input and invokes methods on it to produce the output. For example, like this:

```
# report.py
...

def print_report(reportdata, formatter):
    '''
    Print a nicely formatted table from a list of (name, shares, price, change) tuples.
    '''
    formatter.headings(['Name','Shares','Price','Change'])
    for name, shares, price, change in reportdata:
        rowdata = [ name, str(shares), f'{price:0.2f}', f'{change:0.2f}' ]
        formatter.row(rowdata)
```

Since you added an argument to print_report(), you're going to need to modify the `portfolio_report()` function as well. Change it so that it creates a `TableFormatter` like this:

```
# report.py

import tableformat

...
def portfolio_report(portfoliofile, pricefile):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.TableFormatter()
    print_report(report, formatter)
```

Run this new code:

```
>>> ================================ RESTART ================================
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
... crashes ...
```

It should immediately crash with a `NotImplementedError` exception. That's not too exciting, but it's exactly what we expected. Continue to the next part.

Exercise 4.6: Using Inheritance to Produce Different Output

The `TableFormatter` class you defined in part (a) is meant to be extended via inheritance. In fact, that's the whole idea. To illustrate, define a class `TextTableFormatter` like this:

```
# tableformat.py
...
class TextTableFormatter(TableFormatter):
    '''
    Emit a table in plain-text format
    '''
    def headings(self, headers):
        for h in headers:
            print(f'{h:>10s}', end=' ')
        print()
        print(('-'*10 + ' ')*len(headers))

    def row(self, rowdata):
        for d in rowdata:
            print(f'{d:>10s}', end=' ')
        print()
```

Modify the `portfolio_report()` function like this and try it:

```
# report.py
...
def portfolio_report(portfoliofile, pricefile):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.TextTableFormatter()
    print_report(report, formatter)
```

This should produce the same output as before:

```
>>> ================================ RESTART ================================
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
      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
>>>
```

However, let's change the output to something else. Define a new class `CSVTableFormatter` that produces output in CSV format:

```
# tableformat.py
...
class CSVTableFormatter(TableFormatter):
    '''
    Output portfolio data in CSV format.
    '''
    def headings(self, headers):
        print(','.join(headers))

    def row(self, rowdata):
        print(','.join(rowdata))
```

Modify your main program as follows:

```
def portfolio_report(portfoliofile, pricefile):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.CSVTableFormatter()
    print_report(report, formatter)
```

You should now see CSV output like this:

```
>>> ================================ RESTART ================================
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
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
```

Using a similar idea, define a class `HTMLTableFormatter` that produces a table with the following output:

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

Test your code by modifying the main program to create a `HTMLTableFormatter` object instead of a `CSVTableFormatter` object.

Exercise 4.7: Polymorphism in Action

A major feature of object-oriented programming is that you can plug an object into a program and it will work without having to change any of the existing code. For example, if you wrote a program that expected to use a `TableFormatter` object, it would work no matter what kind of `TableFormatter` you actually gave it. This behavior is sometimes referred to as "polymorphism."

One potential problem is figuring out how to allow a user to pick out the formatter that they want. Direct use of the class names such as `TextTableFormatter` is often annoying. Thus, you might consider some simplified approach. Perhaps you embed an `if` statement into the code like this:

```
def portfolio_report(portfoliofile, pricefile, fmt='txt'):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    if fmt == 'txt':
        formatter = tableformat.TextTableFormatter()
```

```
    elif fmt == 'csv':
        formatter = tableformat.CSVTableFormatter()
    elif fmt == 'html':
        formatter = tableformat.HTMLTableFormatter()
    else:
        raise RuntimeError(f'Unknown format {fmt}')
    print_report(report, formatter)
```

In this code, the user specifies a simplified name such as `'txt'` or `'csv'` to pick a format. However, is putting a big `if`-statement in the `portfolio_report()` function like that the best idea? It might be better to move that code to a general purpose function somewhere else.

In the `tableformat.py` file, add a function `create_formatter(name)` that allows a user to create a formatter given an output name such as `'txt'`, `'csv'`, or `'html'`. Modify `portfolio_report()` so that it looks like this:

```
def portfolio_report(portfoliofile, pricefile, fmt='txt'):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.create_formatter(fmt)
    print_report(report, formatter)
```

Try calling the function with different formats to make sure it's working.

Exercise 4.8: Putting it all together

Modify the `report.py` program so that the `portfolio_report()` function takes an optional argument specifying the output format. For example:

```
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv', 'txt')
      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
>>>
```

Modify the main program so that a format can be given on the command line:

```
bash $ python3 report.py Data/portfolio.csv Data/prices.csv csv
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
bash $
```

Discussion

Writing extensible code is one of the most common uses of inheritance in libraries and frameworks. For example, a framework might instruct you to define your own object that inherits from a provided base class. You're then told to fill in various methods that implement various bits of functionality.

Another somewhat deeper concept is the idea of "owning your abstractions." In the exercises, we defined *our own class* for formatting a table. You may look at your code and tell yourself "I should just use a formatting library or something that someone else already made instead!" No, you should use BOTH your class and a library. Using your own class promotes loose coupling and is more flexible. As long as your application uses the programming interface of your class, you can change the internal implementation to work in any way that you want. You can write all-custom code. You can use someone's third party package. You swap out one third-party package for a different package when you find a better one. It doesn't matter--none of your application code will break as long as you preserve the interface. That's a powerful idea and it's one of the reasons why you might consider inheritance for something like this.

That said, designing object oriented programs can be extremely difficult. For more information, you should probably look for books on the topic of design patterns (although understanding what happened in this exercise will take you pretty far in terms of using objects in a practically useful way).

Contents | Previous (4.1 Classes) | Next (4.3 Special methods)

Contents | Previous (4.2 Inheritance) | Next (4.4 Exceptions)

4.3 Special Methods

Various parts of Python's behavior can be customized via special or so-called "magic" methods. This section introduces that idea. In addition dynamic attribute access and bound methods are discussed.

Introduction

Classes may define special methods. These have special meaning to the Python interpreter. They are always preceded and followed by `__`. For example `__init__`.

```
class Stock(object):
    def __init__(self):
        ...
    def __repr__(self):
        ...
```

There are dozens of special methods, but we will only look at a few specific examples.

Special methods for String Conversions

Objects have two string representations.

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> print(d)
2012-12-21
>>> d
datetime.date(2012, 12, 21)
>>>
```

The `str()` function is used to create a nice printable output:

```
>>> str(d)
'2012-12-21'
>>>
```

The `repr()` function is used to create a more detailed representation for programmers.

```
>>> repr(d)
'datetime.date(2012, 12, 21)'
>>>
```

Those functions, `str()` and `repr()`, use a pair of special methods in the class to produce the string to be displayed.

```
class Date(object):
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # Used with `str()`
    def __str__(self):
        return f'{self.year}-{self.month}-{self.day}'

    # Used with `repr()`
    def __repr__(self):
        return f'Date({self.year},{self.month},{self.day})'
```

Note: The convention for `__repr__()` is to return a string that, when fed to `eval()`, will recreate the underlying object. If this is not possible, some kind of easily readable representation is used instead.

Special Methods for Mathematics

Mathematical operators involve calls to the following methods.

```
a + b       a.__add__(b)
a - b       a.__sub__(b)
a * b       a.__mul__(b)
a / b       a.__truediv__(b)
a // b      a.__floordiv__(b)
a % b       a.__mod__(b)
a << b      a.__lshift__(b)
a >> b      a.__rshift__(b)
a & b       a.__and__(b)
a | b       a.__or__(b)
a ^ b       a.__xor__(b)
a ** b      a.__pow__(b)
-a          a.__neg__()
~a          a.__invert__()
abs(a)      a.__abs__()
```

Special Methods for Item Access

These are the methods to implement containers.

```
len(x)      x.__len__()
x[a]        x.__getitem__(a)
x[a] = v    x.__setitem__(a,v)
del x[a]    x.__delitem__(a)
```

You can use them in your classes.

```
class Sequence:
    def __len__(self):
        ...
    def __getitem__(self,a):
        ...
    def __setitem__(self,a,v):
        ...
    def __delitem__(self,a):
        ...
```

Method Invocation

Invoking a method is a two-step process.

1. Lookup: The `.` operator
2. Method call: The `()` operator

```
>>> s = Stock('GOOG',100,490.10)
>>> c = s.cost  # Lookup
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()         # Method call
49010.0
>>>
```

Bound Methods

A method that has not yet been invoked by the function call operator `()` is known as a *bound method*. It operates on the instance where it originated.

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s
<Stock object at 0x590d0>
>>> c = s.cost
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()
49010.0
>>>
```

Bound methods are often a source of careless non-obvious errors. For example:

```
>>> s = Stock('GOOG', 100, 490.10)
>>> print('Cost : %0.2f' % s.cost)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float argument required
>>>
```

Or devious behavior that's hard to debug.

```
f = open(filename, 'w')
...
f.close     # Oops, Didn't do anything at all. `f` still open.
```

In both of these cases, the error is cause by forgetting to include the trailing parentheses. For example, `s.cost()` or `f.close()`.

Attribute Access

There is an alternative way to access, manipulate and manage attributes.

```
getattr(obj, 'name')          # Same as obj.name
setattr(obj, 'name', value)   # Same as obj.name = value
delattr(obj, 'name')          # Same as del obj.name
hasattr(obj, 'name')          # Tests if attribute exists
```

Example:

```
if hasattr(obj, 'x'):
    x = getattr(obj, 'x'):
else:
    x = None
```

*Note: `getattr()` also has a useful default value *arg*.

```
x = getattr(obj, 'x', None)
```

Exercises

Exercise 4.9: Better output for printing objects

Modify the `Stock` object that you defined in `stock.py` so that the `__repr__()` method produces more useful output. For example:

```
>>> goog = Stock('GOOG', 100, 490.1)
>>> goog
Stock('GOOG', 100, 490.1)
>>>
```

See what happens when you read a portfolio of stocks and view the resulting list after you have made these changes. For example:

```
>>> import report
>>> portfolio = report.read_portfolio('Data/portfolio.csv')
>>> portfolio
... see what the output is ...
>>>
```

Exercise 4.10: An example of using getattr()

`getattr()` is an alternative mechanism for reading attributes. It can be used to write extremely flexible code. To begin, try this example:

```
>>> import stock
>>> s = stock.Stock('GOOG', 100, 490.1)
>>> columns = ['name', 'shares']
>>> for colname in columns:
        print(colname, '=', getattr(s, colname))

name = GOOG
shares = 100
>>>
```

Carefully observe that the output data is determined entirely by the attribute names listed in the `columns` variable.

In the file `tableformat.py`, take this idea and expand it into a generalized function `print_table()` that prints a table showing user-specified attributes of a list of arbitrary objects. As with the earlier `print_report()` function, `print_table()` should also accept a `TableFormatter` instance to control the output format. Here's how it should work:

```
>>> import report
>>> portfolio = report.read_portfolio('Data/portfolio.csv')
>>> from tableformat import create_formatter, print_table
>>> formatter = create_formatter('txt')
>>> print_table(portfolio, ['name','shares'], formatter)
      name     shares
---------- ----------
        AA        100
       IBM         50
       CAT        150
      MSFT        200
        GE         95
      MSFT         50
       IBM        100

>>> print_table(portfolio, ['name','shares','price'], formatter)
      name     shares      price
---------- ---------- ----------
        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
>>>
```

Contents | Previous (4.2 Inheritance) | Next (4.4 Exceptions)

Contents | Previous (4.3 Special methods) | Next (5 Object Model)

4.4 Defining Exceptions

User defined exceptions are defined by classes.

```
class NetworkError(Exception):
    pass
```

Exceptions always inherit from `Exception`.

Usually they are empty classes. Use `pass` for the body.

You can also make a hierarchy of your exceptions.

```
class AuthenticationError(NetworkError):
     pass

class ProtocolError(NetworkError):
    pass
```

Exercises

Exercise 4.11: Defining a custom exception

It is often good practice for libraries to define their own exceptions.

This makes it easier to distinguish between Python exceptions raised in response to common programming errors versus exceptions intentionally raised by a library to a signal a specific usage problem.

Modify the `create_formatter()` function from the last exercise so that it raises a custom `FormatError` exception when the user provides a bad format name.

For example:

```
>>> from tableformat import create_formatter
>>> formatter = create_formatter('xls')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "tableformat.py", line 71, in create_formatter
    raise FormatError('Unknown table format %s' % name)
FormatError: Unknown table format xls
>>>
```

Contents | Previous (4.3 Special methods) | Next (5 Object Model)

Contents | Prev (4 Classes and Objects) | Next (6 Generators)

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.

Contents | Prev (4 Classes and Objects) | Next (6 Generators)

Contents | Previous (4.4 Exceptions) | Next (5.2 Encapsulation)

5.1 Dictionaries Revisited

The Python object system is largely based on an implementation involving dictionaries. This section discusses that.

Dictionaries, Revisited

Remember that a dictionary is a collection of named values.

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

Dictionaries are commonly used for simple data structures. However, they are used for critical parts of the interpreter and may be the *most important type of data in Python*.

Dicts and Modules

Within a module, a dictionary holds all of the global variables and functions.

```
# foo.py

x = 42
def bar():
    ...

def spam():
    ...
```

If you inspect `foo.__dict__` or `globals()`, you'll see the dictionary.

```
{
    'x' : 42,
    'bar' : <function bar>,
    'spam' : <function spam>
}
```

Dicts and Objects

User defined objects also use dictionaries for both instance data and classes. In fact, the entire object system is mostly an extra layer that's put on top of dictionaries.

A dictionary holds the instance data, `__dict__`.

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.__dict__
{'name' : 'GOOG', 'shares' : 100, 'price': 490.1 }
```

You populate this dict (and instance) when assigning to `self`.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

The instance data, `self.__dict__`, looks like this:

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

Each instance gets its own private dictionary.

```
s = Stock('GOOG', 100, 490.1)     # {'name' : 'GOOG','shares' : 100, 'price': 490.1 }
t = Stock('AAPL', 50, 123.45)     # {'name' : 'AAPL','shares' : 50, 'price': 123.45 }
```

If you created 100 instances of some class, there are 100 dictionaries sitting around holding data.

Class Members

A separate dictionary also holds the methods.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    def cost(self):
        return self.shares * self.price

    def sell(self, nshares):
        self.shares -= nshares
```

The dictionary is in `Stock.__dict__`.

```
{
    'cost': <function>,
    'sell': <function>,
    '__init__': <function>
}
```

Instances and Classes

Instances and classes are linked together. The `__class__` attribute refers back to the class.

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.__dict__
{ 'name': 'GOOG', 'shares': 100, 'price': 490.1 }
>>> s.__class__
<class '__main__.Stock'>
>>>
```

The instance dictionary holds data unique to each instance, whereas the class dictionary holds data collectively shared by *all* instances.

Attribute Access

When you work with objects, you access data and methods using the `.` operator.

```
x = obj.name          # Getting
obj.name = value      # Setting
del obj.name          # Deleting
```

These operations are directly tied to the dictionaries sitting underneath the covers.

Modifying Instances

Operations that modify an object update the underlying dictionary.

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.__dict__
{ 'name':'GOOG', 'shares': 100, 'price': 490.1 }
>>> s.shares = 50       # Setting
>>> s.date = '6/7/2007' # Setting
>>> s.__dict__
{ 'name': 'GOOG', 'shares': 50, 'price': 490.1, 'date': '6/7/2007' }
>>> del s.shares        # Deleting
>>> s.__dict__
{ 'name': 'GOOG', 'price': 490.1, 'date': '6/7/2007' }
>>>
```

Reading Attributes

Suppose you read an attribute on an instance.

```
x = obj.name
```

The attribute may exist in two places:

- Local instance dictionary.
- Class dictionary.

Both dictionaries must be checked. First, check in local `__dict__`. If not found, look in `__dict__` of class through `__class__`.

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```

This lookup scheme is how the members of a *class* get shared by all instances.

How inheritance works

Classes may inherit from other classes.

```
class A(B, C):
    ...
```

The base classes are stored in a tuple in each class.

```
>>> A.__bases__
(<class '__main__.B'>, <class '__main__.C'>)
>>>
```

This provides a link to parent classes.

Reading Attributes with Inheritance

Logically, the process of finding an attribute is as follows. First, check in local `__dict__`. If not found, look in `__dict__` of the class. If not found in class, look in the base classes through `__bases__`. However, there are some subtle aspects of this discussed next.

Reading Attributes with Single Inheritance

In inheritance hierarchies, attributes are found by walking up the inheritance tree in order.

```
class A: pass
class B(A): pass
class C(A): pass
class D(B): pass
class E(D): pass
```

With single inheritance, there is single path to the top. You stop with the first match.

Method Resolution Order or MRO

Python precomputes an inheritance chain and stores it in the *MRO* attribute on the class. You can view it.

```
>>> E.__mro__
(<class '__main__.E'>, <class '__main__.D'>,
 <class '__main__.B'>, <class '__main__.A'>,
 <type 'object'>)
>>>
```

This chain is called the **Method Resolution Order**. To find an attribute, Python walks the MRO in order. The first match wins.

MRO in Multiple Inheritance

With multiple inheritance, there is no single path to the top. Let's take a look at an example.

```
class A: pass
class B: pass
class C(A, B): pass
class D(B): pass
class E(C, D): pass
```

What happens when you access an attribute?

```
e = E()
e.attr
```

An attribute search process is carried out, but what is the order? That's a problem.

Python uses *cooperative multiple inheritance* which obeys some rules about class ordering.

- Children are always checked before parents
- Parents (if multiple) are always checked in the order listed.

The MRO is computed by sorting all of the classes in a hierarchy according to those rules.

```
>>> E.__mro__
(
  <class 'E'>,
  <class 'C'>,
  <class 'A'>,
  <class 'D'>,
  <class 'B'>,
  <class 'object'>)
>>>
```

The underlying algorithm is called the "C3 Linearization Algorithm." The precise details aren't important as long as you remember that a class hierarchy obeys the same ordering rules you might follow if your house was on fire and you had to evacuate--children first, followed by parents.

An Odd Code Reuse (Involving Multiple Inheritance)

Consider two completely unrelated objects:

```
class Dog:
    def noise(self):
        return 'Bark'

    def chase(self):
        return 'Chasing!'

class LoudDog(Dog):
    def noise(self):
        # Code commonality with LoudBike (below)
        return super().noise().upper()
```

And

```
class Bike:
    def noise(self):
        return 'On Your Left'

    def pedal(self):
        return 'Pedaling!'

class LoudBike(Bike):
    def noise(self):
        # Code commonality with LoudDog (above)
        return super().noise().upper()
```

There is a code commonality in the implementation of `LoudDog.noise()` and `LoudBike.noise()`. In fact, the code is exactly the same. Naturally, code like that is bound to attract software engineers.

The "Mixin" Pattern

The *Mixin* pattern is a class with a fragment of code.

```
class Loud:
    def noise(self):
        return super().noise().upper()
```

This class is not usable in isolation. It mixes with other classes via inheritance.

```
class LoudDog(Loud, Dog):
    pass

class LoudBike(Loud, Bike):
    pass
```

Miraculously, loudness was now implemented just once and reused in two completely unrelated classes. This sort of trick is one of the primary uses of multiple inheritance in Python.

Why `super()`

Always use `super()` when overriding methods.

```
class Loud:
    def noise(self):
        return super().noise().upper()
```

`super()` delegates to the *next class* on the MRO.

The tricky bit is that you don't know what it is. You especially don't know what it is if multiple inheritance is being used.

Some Cautions

Multiple inheritance is a powerful tool. Remember that with power comes responsibility. Frameworks / libraries sometimes use it for advanced features involving composition of components. Now, forget that you saw that.

Exercises

In Section 4, you defined a class `Stock` that represented a holding of stock. In this exercise, we will use that class. Restart the interpreter and make a few instances:

```
>>> ================================ RESTART ================================
>>> from stock import Stock
>>> goog = Stock('GOOG',100,490.10)
>>> ibm  = Stock('IBM',50, 91.23)
>>>
```

Exercise 5.1: Representation of Instances

At the interactive shell, inspect the underlying dictionaries of the two instances you created:

```
>>> goog.__dict__
... look at the output ...
>>> ibm.__dict__
... look at the output ...
>>>
```

Exercise 5.2: Modification of Instance Data

Try setting a new attribute on one of the above instances:

```
>>> goog.date = '6/11/2007'
>>> goog.__dict__
... look at output ...
>>> ibm.__dict__
... look at output ...
>>>
```

In the above output, you'll notice that the `goog` instance has a attribute `date` whereas the `ibm` instance does not. It is important to note that Python really doesn't place any restrictions on attributes. For example, the attributes of an instance are not limited to those set up in the `__init__()` method.

Instead of setting an attribute, try placing a new value directly into the `__dict__` object:

```
>>> goog.__dict__['time'] = '9:45am'
>>> goog.time
'9:45am'
>>>
```

Here, you really notice the fact that an instance is just a layer on top of a dictionary. Note: it should be emphasized that direct manipulation of the dictionary is uncommon--you should always write your code to use the (.) syntax.

Exercise 5.3: The role of classes

The definitions that make up a class definition are shared by all instances of that class. Notice, that all instances have a link back to their associated class:

```
>>> goog.__class__
... look at output ...
>>> ibm.__class__
... look at output ...
>>>
```

Try calling a method on the instances:

```
>>> goog.cost()
49010.0
>>> ibm.cost()
4561.5
>>>
```

Notice that the name 'cost' is not defined in either `goog.__dict__` or `ibm.__dict__`. Instead, it is being supplied by the class dictionary. Try this:

```
>>> Stock.__dict__['cost']
... look at output ...
>>>
```

Try calling the `cost()` method directly through the dictionary:

```
>>> Stock.__dict__['cost'](goog)
49010.0
>>> Stock.__dict__['cost'](ibm)
4561.5
>>>
```

Notice how you are calling the function defined in the class definition and how the `self` argument gets the instance.

Try adding a new attribute to the `Stock` class:

```
>>> Stock.foo = 42
>>>
```

Notice how this new attribute now shows up on all of the instances:

```
>>> goog.foo
42
>>> ibm.foo
42
>>>
```

However, notice that it is not part of the instance dictionary:

```
>>> goog.__dict__
... look at output and notice there is no 'foo' attribute ...
>>>
```

The reason you can access the `foo` attribute on instances is that Python always checks the class dictionary if it can't find something on the instance itself.

Note: This part of the exercise illustrates something known as a class variable. Suppose, for instance, you have a class like this:

```
class Foo(object):
     a = 13                  # Class variable
     def __init__(self,b):
         self.b = b          # Instance variable
```

In this class, the variable `a`, assigned in the body of the class itself, is a "class variable." It is shared by all of the instances that get created. For example:

```
>>> f = Foo(10)
>>> g = Foo(20)
>>> f.a          # Inspect the class variable (same for both instances)
13
>>> g.a
13
>>> f.b          # Inspect the instance variable (differs)
10
>>> g.b
20
>>> Foo.a = 42   # Change the value of the class variable
>>> f.a
42
>>> g.a
```

```
42
>>>
```

Exercise 5.4: Bound methods

A subtle feature of Python is that invoking a method actually involves two steps and something known as a bound method. For example:

```
>>> s = goog.sell
>>> s
<bound method Stock.sell of Stock('GOOG', 100, 490.1)>
>>> s(25)
>>> goog.shares
75
>>>
```

Bound methods actually contain all of the pieces needed to call a method. For instance, they keep a record of the function implementing the method:

```
>>> s.__func__
<function sell at 0x10049af50>
>>>
```

This is the same value as found in the `Stock` dictionary.

```
>>> Stock.__dict__['sell']
<function sell at 0x10049af50>
>>>
```

Bound methods also record the instance, which is the `self` argument.

```
>>> s.__self__
Stock('GOOG',75,490.1)
>>>
```

When you invoke the function using `()` all of the pieces come together. For example, calling `s(25)` actually does this:

```
>>> s.__func__(s.__self__, 25)    # Same as s(25)
>>> goog.shares
50
>>>
```

Exercise 5.5: Inheritance

Make a new class that inherits from `Stock`.

```
>>> class NewStock(Stock):
        def yow(self):
            print('Yow!')

>>> n = NewStock('ACME', 50, 123.45)
>>> n.cost()
6172.50
>>> n.yow()
Yow!
>>>
```

Inheritance is implemented by extending the search process for attributes. The `__bases__` attribute has a tuple of the immediate parents:

```
>>> NewStock.__bases__
(<class 'stock.Stock'>,)
>>>
```

The `__mro__` attribute has a tuple of all parents, in the order that they will be searched for attributes.

```
>>> NewStock.__mro__
(<class '__main__.NewStock'>, <class 'stock.Stock'>, <class 'object'>)
>>>
```

Here's how the `cost()` method of instance `n` above would be found:

```
>>> for cls in n.__class__.__mro__:
        if 'cost' in cls.__dict__:
            break

>>> cls
<class '__main__.Stock'>
>>> cls.__dict__['cost']
<function cost at 0x101aed598>
>>>
```

Contents | Previous (4.4 Exceptions) | Next (5.2 Encapsulation)

Contents | Previous (5.1 Dictionaries Revisited) | Next (6 Generators)

5.2 Classes and Encapsulation

When writing classes, it is common to try and encapsulate internal details. This section introduces a few Python programming idioms for this including private variables and properties.

Public vs Private.

One of the primary roles of a class is to encapsulate data and internal implementation details of an object. However, a class also defines a *public* interface that the outside world is supposed to use to manipulate the object. This distinction between implementation details and the public interface is important.

A Problem

In Python, almost everything about classes and objects is *open*.

- You can easily inspect object internals.
- You can change things at will.
- There is no strong notion of access-control (i.e., private class members)

That is an issue when you are trying to isolate details of the *internal implementation*.

Python Encapsulation

Python relies on programming conventions to indicate the intended use of something. These conventions are based on naming. There is a general attitude that it is up to the programmer to observe the rules as opposed to having the language enforce them.

Private Attributes

Any attribute name with leading `_` is considered to be *private*.

```
class Person(object):
    def __init__(self, name):
        self._name = 0
```

As mentioned earlier, this is only a programming style. You can still access and change it.

```
>>> p = Person('Guido')
>>> p._name
'Guido'
>>> p._name = 'Dave'
>>>
```

As a general rule, any name with a leading `_` is considered internal implementation whether it's a variable, a function, or a module name. If you find yourself using such names directly, you're probably doing something wrong. Look for higher level functionality.

Simple Attributes

Consider the following class.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

A surprising feature is that you can set the attributes to any value at all:

```
>>> s = Stock('IBM', 50, 91.1)
>>> s.shares = 100
>>> s.shares = "hundred"
>>> s.shares = [1, 0, 0]
>>>
```

You might look at that and think you want some extra checks.

```
s.shares = '50'     # Raise a TypeError, this is a string
```

How would you do it?

Managed Attributes

One approach: introduce accessor methods.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.set_shares(shares)
        self.price = price

    # Function that layers the "get" operation
    def get_shares(self):
        return self._shares

    # Function that layers the "set" operation
    def set_shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        self._shares = value
```

Too bad that this breaks all of our existing code. `s.shares = 50` becomes `s.set_shares(50)`

Properties

There is an alternative approach to the previous pattern.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
```

```
            raise TypeError('Expected int')
        self._shares = value
```

Normal attribute access now triggers the getter and setter methods under `@property` and `@shares.setter`.

```
>>> s = Stock('IBM', 50, 91.1)
>>> s.shares         # Triggers @property
50
>>> s.shares = 75    # Triggers @shares.setter
>>>
```

With this pattern, there are *no changes* needed to the source code. The new *setter* is also called when there is an assignment within the class, including inside the `__init__()` method.

```
class Stock:
    def __init__(self, name, shares, price):
        ...
        # This assignment calls the setter below
        self.shares = shares
        ...

    ...
    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected int')
        self._shares = value
```

There is often a confusion between a property and the use of private names. Although a property internally uses a private name like `_shares`, the rest of the class (not the property) can continue to use a name like `shares`.

Properties are also useful for computed data attributes.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def cost(self):
        return self.shares * self.price
    ...
```

This allows you to drop the extra parentheses, hiding the fact that it's actually a method:

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.shares # Instance variable
100
>>> s.cost   # Computed Value
49010.0
>>>
```

Uniform access

The last example shows how to put a more uniform interface on an object. If you don't do this, an object might be confusing to use:

```
>>> s = Stock('GOOG', 100, 490.1)
>>> a = s.cost() # Method
49010.0
>>> b = s.shares # Data attribute
100
>>>
```

Why is the `()` required for the cost, but not for the shares? A property can fix this.

Decorator Syntax

The `@` syntax is known as "decoration". It specifies a modifier that's applied to the function definition that immediately follows.

```
...
@property
def cost(self):
    return self.shares * self.price
```

More details are given in Section 7.

`__slots__` Attribute

You can restrict the set of attributes names.

```
class Stock:
    __slots__ = ('name','_shares','price')
    def __init__(self, name, shares, price):
        self.name = name
        ...
```

It will raise an error for other attributes.

```
>>> s.price = 385.15
>>> s.prices = 410.2
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: 'Stock' object has no attribute 'prices'
```

Although this prevents errors and restricts usage of objects, it's actually used for performance and makes Python use memory more efficiently.

Final Comments on Encapsulation

Don't go overboard with private attributes, properties, slots, etc. They serve a specific purpose and you may see them when reading other Python code. However, they are not necessary for most day-to-day coding.

Exercises

Exercise 5.6: Simple Properties

Properties are a useful way to add "computed attributes" to an object. In `stock.py`, you created an object `Stock`. Notice that on your object there is a slight inconsistency in how different kinds of data are extracted:

```
>>> from stock import Stock
>>> s = Stock('GOOG', 100, 490.1)
>>> s.shares
100
>>> s.price
490.1
>>> s.cost()
49010.0
>>>
```

Specifically, notice how you have to add the extra () to `cost` because it is a method.

You can get rid of the extra () on `cost()` if you turn it into a property. Take your `Stock` class and modify it so that the cost calculation works like this:

```
>>> ================================ RESTART ================================
>>> from stock import Stock
>>> s = Stock('GOOG', 100, 490.1)
>>> s.cost
49010.0
>>>
```

Try calling `s.cost()` as a function and observe that it doesn't work now that `cost` has been defined as a property.

```
>>> s.cost()
... fails ...
>>>
```

Making this change will likely break your earlier `pcost.py` program. You might need to go back and get rid of the `()` on the `cost()` method.

Exercise 5.7: Properties and Setters

Modify the `shares` attribute so that the value is stored in a private attribute and that a pair of property functions are used to ensure that it is always set to an integer value. Here is an example of the expected behavior:

```
>>> ================================ RESTART ================================
>>> from stock import Stock
>>> s = Stock('GOOG',100,490.10)
>>> s.shares = 50
>>> s.shares = 'a lot'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected an integer
>>>
```

Exercise 5.8: Adding slots

Modify the `Stock` class so that it has a `__slots__` attribute. Then, verify that new attributes can't be added:

```
>>> ================================ RESTART ================================
>>> from stock import Stock
>>> s = Stock('GOOG', 100, 490.10)
>>> s.name
'GOOG'
>>> s.blah = 42
... see what happens ...
>>>
```

When you use `__slots__`, Python uses a more efficient internal representation of objects. What happens if you try to inspect the underlying dictionary of `s` above?

```
>>> s.__dict__
... see what happens ...
>>>
```

It should be noted that `__slots__` is most commonly used as an optimization on classes that serve as data structures. Using slots will make such programs use far-less memory and run a bit faster. You should probably avoid `__slots__` on most other classes however.

Contents | Previous (5.1 Dictionaries Revisited) | Next (6 Generators)

Contents | Prev (5 Inner Workings of Python Objects) | Next (7 Advanced Topics)

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.

Contents | Prev (5 Inner Workings of Python Objects) | Next (7 Advanced Topics)

Contents | Previous (5.2 Encapsulation) | Next (6.2 Customizing Iteration)

6.1 Iteration Protocol

This section looks at the underlying process of iteration.

Iteration Everywhere

Many different objects support iteration.

```
a = 'hello'
for c in a: # Loop over characters in a
    ...

b = { 'name': 'Dave', 'password':'foo'}
for k in b: # Loop over keys in dictionary
    ...

c = [1,2,3,4]
for i in c: # Loop over items in a list/tuple
    ...

f = open('foo.txt')
for x in f: # Loop over lines in a file
    ...
```

Iteration: Protocol

Consider the `for`-statement.

```
for x in obj:
    # statements
```

What happens under the hood?

```
_iter = obj.__iter__()        # Get iterator object
while True:
    try:
        x = _iter.__next__()  # Get next item
        # statements ...
    except StopIteration:     # No more items
        break
```

All the objects that work with the `for-loop` implement this low-level iteration protocol.

Example: Manual iteration over a list.

```
>>> x = [1,2,3]
>>> it = x.__iter__()
>>> it
<listiterator object at 0x590b0>
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
3
>>> it.__next__()
Traceback (most recent call last):
File "<stdin>", line 1, in ? StopIteration
>>>
```

Supporting Iteration

Knowing about iteration is useful if you want to add it to your own objects. For example, making a custom container.

```
class Portfolio:
    def __init__(self):
        self.holdings = []

    def __iter__(self):
        return self.holdings.__iter__()
    ...

port = Portfolio()
for s in port:
    ...
```

Exercises

Exercise 6.1: Iteration Illustrated

Create the following list:

```
a = [1,9,4,25,16]
```

Manually iterate over this list. Call `__iter__()` to get an iterator and call the `__next__()` method to obtain successive elements.

```
>>> i = a.__iter__()
>>> i
<listiterator object at 0x64c10>
>>> i.__next__()
1
>>> i.__next__()
9
>>> i.__next__()
4
>>> i.__next__()
25
>>> i.__next__()
16
>>> i.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

The `next()` built-in function is a shortcut for calling the `__next__()` method of an iterator. Try using it on a file:

```
>>> f = open('Data/portfolio.csv')
>>> f.__iter__()    # Note: This returns the file itself
<_io.TextIOWrapper name='Data/portfolio.csv' mode='r' encoding='UTF-8'>
>>> next(f)
'name,shares,price\n'
>>> next(f)
'"AA",100,32.20\n'
>>> next(f)
'"IBM",50,91.10\n'
>>>
```

Keep calling `next(f)` until you reach the end of the file. Watch what happens.

Exercise 6.2: Supporting Iteration

On occasion, you might want to make one of your own objects support iteration--especially if your object wraps around an existing list or other iterable. In a new file `portfolio.py`, define the following class:

```
# portfolio.py

class Portfolio:

    def __init__(self, holdings):
```

```
        self._holdings = holdings

    @property
    def total_cost(self):
        return sum([s.cost for s in self._holdings])

    def tabulate_shares(self):
        from collections import Counter
        total_shares = Counter()
        for s in self._holdings:
            total_shares[s.name] += s.shares
        return total_shares
```

This class is meant to be a layer around a list, but with some extra methods such as the `total_cost` property. Modify the `read_portfolio()` function in `report.py` so that it creates a `Portfolio` instance like this:

```
# report.py
...

import fileparse
from stock import Stock
from portfolio import Portfolio

def read_portfolio(filename):
    '''
    Read a stock portfolio file into a list of dictionaries with keys
    name, shares, and price.
    '''
    with open(filename) as file:
        portdicts = fileparse.parse_csv(file,
                                        select=['name','shares','price'],
                                        types=[str,int,float])

    portfolio = [ Stock(d['name'], d['shares'], d['price']) for d in portdicts ]
    return Portfolio(portfolio)
...
```

Try running the `report.py` program. You will find that it fails spectacularly due to the fact that `Portfolio` instances aren't iterable.

```
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
... crashes ...
```

Fix this by modifying the `Portfolio` class to support iteration:

```
class Portfolio:

    def __init__(self, holdings):
        self._holdings = holdings
```

```
    def __iter__(self):
        return self._holdings.__iter__()

    @property
    def total_cost(self):
        return sum([s.shares*s.price for s in self._holdings])

    def tabulate_shares(self):
        from collections import Counter
        total_shares = Counter()
        for s in self._holdings:
            total_shares[s.name] += s.shares
        return total_shares
```

After you've made this change, your `report.py` program should work again. While you're at it, fix up your `pcost.py` program to use the new `Portfolio` object. Like this:

```
# pcost.py

import report

def portfolio_cost(filename):
    '''
    Computes the total cost (shares*price) of a portfolio file
    '''
    portfolio = report.read_portfolio(filename)
    return portfolio.total_cost
...
```

Test it to make sure it works:

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

Exercise 6.3: Making a more proper container

If making a container class, you often want to do more than just iteration. Modify the `Portfolio` class so that it has some other special methods like this:

```
class Portfolio:
    def __init__(self, holdings):
        self._holdings = holdings

    def __iter__(self):
        return self._holdings.__iter__()

    def __len__(self):
        return len(self._holdings)
```

```
    def __getitem__(self, index):
        return self._holdings[index]

    def __contains__(self, name):
        return any([s.name == name for s in self._holdings])

    @property
    def total_cost(self):
        return sum([s.shares*s.price for s in self._holdings])

    def tabulate_shares(self):
        from collections import Counter
        total_shares = Counter()
        for s in self._holdings:
            total_shares[s.name] += s.shares
        return total_shares
```

Now, try some experiments using this new class:

```
>>> import report
>>> portfolio = report.read_portfolio('Data/portfolio.csv')
>>> len(portfolio)
7
>>> portfolio[0]
Stock('AA', 100, 32.2)
>>> portfolio[1]
Stock('IBM', 50, 91.1)
>>> portfolio[0:3]
[Stock('AA', 100, 32.2), Stock('IBM', 50, 91.1), Stock('CAT', 150, 83.44)]
>>> 'IBM' in portfolio
True
>>> 'AAPL' in portfolio
False
>>>
```

One important observation about this--generally code is considered "Pythonic" if it speaks the common vocabulary of how other parts of Python normally work. For container objects, supporting iteration, indexing, containment, and other kinds of operators is an important part of this.

Contents | Previous (5.2 Encapsulation) | Next (6.2 Customizing Iteration)

Contents | Previous (6.1 Iteration Protocol) | Next (6.3 Producer/Consumer)

6.2 Customizing Iteration

This section looks at how you can customize iteration using a generator function.

A problem

Suppose you wanted to create your own custom iteration pattern.

For example, a countdown.

```
>>> for x in countdown(10):
...   print(x, end=' ')
...
10 9 8 7 6 5 4 3 2 1
>>>
```

There is an easy way to do this.

Generators

A generator is a function that defines iteration.

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

For example:

```
>>> for x in countdown(10):
...   print(x, end=' ')
...
10 9 8 7 6 5 4 3 2 1
>>>
```

A generator is any function that uses the `yield` statement.

The behavior of generators is different than a normal function. Calling a generator function creates a generator object. It does not immediately execute the function.

```
def countdown(n):
    # Added a print statement
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1
```

```
>>> x = countdown(10)
# There is NO PRINT STATEMENT
>>> x
# x is a generator object
<generator object at 0x58490>
>>>
```

The function only executes on `__next__()` call.

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> x.__next__()
Counting down from 10
10
>>>
```

`yield` produces a value, but suspends the function execution. The function resumes on next call to `__next__()`.

```
>>> x.__next__()
9
>>> x.__next__()
8
```

When the generator finally returns, the iteration raises an error.

```
>>> x.__next__()
1
>>> x.__next__()
Traceback (most recent call last):
File "<stdin>", line 1, in ? StopIteration
>>>
```

Observation: A generator function implements the same low-level protocol that the for statements uses on lists, tuples, dicts, files, etc.

Exercises

Exercise 6.4: A Simple Generator

If you ever find yourself wanting to customize iteration, you should always think generator functions. They're easy to write---make a function that carries out the desired iteration logic and use `yield` to emit values.

For example, try this generator that searches a file for lines containing a matching substring:

```
>>> def filematch(filename, substr):
        with open(filename, 'r') as f:
            for line in f:
                if substr in line:
                    yield line

>>> for line in open('Data/portfolio.csv'):
        print(line, end='')

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
>>> for line in filematch('Data/portfolio.csv', 'IBM'):
        print(line, end='')

"IBM",50,91.10
"IBM",100,70.44
>>>
```

This is kind of interesting--the idea that you can hide a bunch of custom processing in a function and use it to feed a for-loop. The next example looks at a more unusual case.

Exercise 6.5: Monitoring a streaming data source

Generators can be an interesting way to monitor real-time data sources such as log files or stock market feeds. In this part, we'll explore this idea. To start, follow the next instructions carefully.

The program `Data/stocksim.py` is a program that simulates stock market data. As output, the program constantly writes real-time data to a file `Data/stocklog.csv`. In a separate command window go into the `Data/` directory and run this program:

```
bash % python3 stocksim.py
```

If you are on Windows, just locate the `stocksim.py` program and double-click on it to run it. Now, forget about this program (just let it run). Using another window, look at the file `Data/stocklog.csv` being written by the simulator. You should see new lines of text being added to the file every few seconds. Again, just let this program run in the background---it will run for several hours (you shouldn't need to worry about it).

Once the above program is running, let's write a little program to open the file, seek to the end, and watch for new output. Create a file `follow.py` and put this code in it:

```
# follow.py
import os
import time

f = open('Data/stocklog.csv')
f.seek(0, os.SEEK_END)   # Move file pointer 0 bytes from end of file

while True:
    line = f.readline()
    if line == '':
        time.sleep(0.1)   # Sleep briefly and retry
        continue
    fields = line.split(',')
    name = fields[0].strip('"')
    price = float(fields[1])
    change = float(fields[4])
    if change < 0:
        print(f'{name:>10s} {price:>10.2f} {change:>10.2f}')
```

If you run the program, you'll see a real-time stock ticker. Under the hood, this code is kind of like the Unix `tail -f` command that's used to watch a log file.

Note: The use of the `readline()` method in this example is somewhat unusual in that it is not the usual way of reading lines from a file (normally you would just use a `for`-loop). However, in this case, we are using it to repeatedly probe the end of the file to see if more data has been added (`readline()` will either return new data or an empty string).

Exercise 6.6: Using a generator to produce data

If you look at the code in Exercise 6.5, the first part of the code is producing lines of data whereas the statements at the end of the `while` loop are consuming the data. A major feature of generator functions is that you can move all of the data production code into a reusable function.

Modify the code in Exercise 6.5 so that the file-reading is performed by a generator function `follow(filename)`. Make it so the following code works:

```
>>> for line in follow('Data/stocklog.csv'):
          print(line, end='')

... Should see lines of output produced here ...
```

Modify the stock ticker code so that it looks like this:

```
if __name__ == '__main__':
    for line in follow('Data/stocklog.csv'):
        fields = line.split(',')
        name = fields[0].strip('"')
        price = float(fields[1])
        change = float(fields[4])
        if change < 0:
            print(f'{name:>10s} {price:>10.2f} {change:>10.2f}')
```

Exercise 6.7: Watching your portfolio

Modify the `follow.py` program so that it watches the stream of stock data and prints a ticker showing information for only those stocks in a portfolio. For example:

```
if __name__ == '__main__':
    import report

    portfolio = report.read_portfolio('Data/portfolio.csv')

    for line in follow('Data/stocklog.csv'):
        fields = line.split(',')
        name = fields[0].strip('"')
        price = float(fields[1])
        change = float(fields[4])
        if name in portfolio:
            print(f'{name:>10s} {price:>10.2f} {change:>10.2f}')
```

Note: For this to work, your `Portfolio` class must support the `in` operator. See Exercise 6.3 and make sure you implement the `__contains__()` operator.

Discussion

Something very powerful just happened here. You moved an interesting iteration pattern (reading lines at the end of a file) into its own little function. The `follow()` function is now this completely general purpose utility that you can use in any program. For example, you could use it to watch server logs, debugging logs, and other similar data sources. That's kind of cool.

Contents | Previous (6.1 Iteration Protocol) | Next (6.3 Producer/Consumer)

Contents | Previous (6.2 Customizing Iteration) | Next (6.4 Generator Expressions)

6.3 Producers, Consumers and Pipelines

Generators are a useful tool for setting various kinds of producer/consumer problems and dataflow pipelines. This section discusses that.

Producer-Consumer Problems

Generators are closely related to various forms of *producer-consumer* problems.

```
# Producer
def follow(f):
    ...
    while True:
        ...
        yield line        # Produces value in `line` below
        ...

# Consumer
for line in follow(f):    # Consumes value from `yield` above
    ...
```

`yield` produces values that `for` consumes.

Generator Pipelines

You can use this aspect of generators to set up processing pipelines (like Unix pipes).

producer → processing → processing → consumer

Processing pipes have an initial data producer, some set of intermediate processing stages and a final consumer.

producer → *processing → processing → consumer*

```
def producer():
    ...
    yield item
    ...
```

The producer is typically a generator. Although it could also be a list of some other sequence. `yield` feeds data into the pipeline.

producer → *processing* → *processing* → **consumer**

```
def consumer(s):
    for item in s:
        ...
```

Consumer is a for-loop. It gets items and does something with them.

producer → **processing** → **processing** → *consumer*

```
def processing(s):
    for item in s:
        ...
        yield newitem
        ...
```

Intermediate processing stages simultaneously consume and produce items. They might modify the data stream. They can also filter (discarding items).

producer → *processing* → *processing* → *consumer*

```
def producer():
    ...
    yield item          # yields the item that is received by the `processing`
    ...

def processing(s):
    for item in s:      # Comes from the `producer`
        ...
        yield newitem   # yields a new item
        ...

def consumer(s):
    for item in s:      # Comes from the `processing`
        ...
```

Code to setup the pipeline

```
a = producer()
b = processing(a)
c = consumer(b)
```

You will notice that data incrementally flows through the different functions.

Exercises

For this exercise the `stocksim.py` program should still be running in the background. You're going to use the `follow()` function you wrote in the previous exercise.

Exercise 6.8: Setting up a simple pipeline

Let's see the pipelining idea in action. Write the following function:

```
>>> def filematch(lines, substr):
        for line in lines:
            if substr in line:
                yield line

>>>
```

This function is almost exactly the same as the first generator example in the previous exercise except that it's no longer opening a file--it merely operates on a sequence of lines given to it as an argument. Now, try this:

```
>>> from follow import follow
>>> lines = follow('Data/stocklog.csv')
>>> ibm = filematch(lines, 'IBM')
>>> for line in ibm:
        print(line)

... wait for output ...
```

It might take awhile for output to appear, but eventually you should see some lines containing data for IBM.

Exercise 6.9: Setting up a more complex pipeline

Take the pipelining idea a few steps further by performing more actions.

```
>>> from follow import follow
>>> import csv
>>> lines = follow('Data/stocklog.csv')
>>> rows = csv.reader(lines)
>>> for row in rows:
        print(row)

['BA', '98.35', '6/11/2007', '09:41.07', '0.16', '98.25', '98.35', '98.31', '158148']
['AA', '39.63', '6/11/2007', '09:41.07', '-0.03', '39.67', '39.63', '39.31', '270224']
['XOM', '82.45', '6/11/2007', '09:41.07', '-0.23', '82.68', '82.64', '82.41', '748062']
['PG', '62.95', '6/11/2007', '09:41.08', '-0.12', '62.80', '62.97', '62.61', '454327']
...
```

Well, that's interesting. What you're seeing here is that the output of the `follow()` function has been piped into the `csv.reader()` function and we're now getting a sequence of split rows.

Exercise 6.10: Making more pipeline components

Let's extend the whole idea into a larger pipeline. In a separate file `ticker.py`, start by creating a function that reads a CSV file as you did above:

```
# ticker.py

from follow import follow
import csv

def parse_stock_data(lines):
    rows = csv.reader(lines)
    return rows

if __name__ == '__main__':
    lines = follow('Data/stocklog.csv')
    rows = parse_stock_data(lines)
    for row in rows:
        print(row)
```

Write a new function that selects specific columns:

```
# ticker.py
...
def select_columns(rows, indices):
    for row in rows:
        yield [row[index] for index in indices]
...
def parse_stock_data(lines):
    rows = csv.reader(lines)
    rows = select_columns(rows, [0, 1, 4])
    return rows
```

Run your program again. You should see output narrowed down like this:

```
['BA', '98.35', '0.16']
['AA', '39.63', '-0.03']
['XOM', '82.45','-0.23']
['PG', '62.95', '-0.12']
...
```

Write generator functions that convert data types and build dictionaries. For example:

```
# ticker.py
...

def convert_types(rows, types):
    for row in rows:
        yield [func(val) for func, val in zip(types, row)]

def make_dicts(rows, headers):
    for row in rows:
        yield dict(zip(headers, row))
```

```
...
def parse_stock_data(lines):
    rows = csv.reader(lines)
    rows = select_columns(rows, [0, 1, 4])
    rows = convert_types(rows, [str, float, float])
    rows = make_dicts(rows, ['name', 'price', 'change'])
    return rows
...
```

Run your program again. You should now a stream of dictionaries like this:

```
{ 'name':'BA', 'price':98.35, 'change':0.16 }
{ 'name':'AA', 'price':39.63, 'change':-0.03 }
{ 'name':'XOM', 'price':82.45, 'change': -0.23 }
{ 'name':'PG', 'price':62.95, 'change':-0.12 }
...
```

Exercise 6.11: Filtering data

Write a function that filters data. For example:

```
# ticker.py
...

def filter_symbols(rows, names):
    for row in rows:
        if row['name'] in names:
            yield row
```

Use this to filter stocks to just those in your portfolio:

```
import report
portfolio = report.read_portfolio('Data/portfolio.csv')
rows = parse_stock_data(follow('Data/stocklog.csv'))
rows = filter_symbols(rows, portfolio)
for row in rows:
    print(row)
```

Exercise 6.12: Putting it all together

In the `ticker.py` program, write a function `ticker(portfile, logfile, fmt)` that creates a real-time stock ticker from a given portfolio, logfile, and table format. For example::

```
>>> from ticker import ticker
>>> ticker('Data/portfolio.csv', 'Data/stocklog.csv', 'txt')
      Name      Price     Change
---------- ---------- ----------
        GE      37.14      -0.18
      MSFT      29.96      -0.09
       CAT      78.03      -0.49
```

```
        AA      39.34      -0.32
...

>>> ticker('Data/portfolio.csv', 'Data/stocklog.csv', 'csv')
Name,Price,Change
IBM,102.79,-0.28
CAT,78.04,-0.48
AA,39.35,-0.31
CAT,78.05,-0.47
...
```

Discussion

Some lessons learned: You can create various generator functions and chain them together to perform processing involving data-flow pipelines. In addition, you can create functions that package a series of pipeline stages into a single function call (for example, the `parse_stock_data()` function).

Contents | Previous (6.2 Customizing Iteration) | Next (6.4 Generator Expressions)

Contents | Previous (6.3 Producer/Consumer) | Next (7 Advanced Topics)

6.4 More Generators

This section introduces a few additional generator related topics including generator expressions and the itertools module.

Generator Expressions

A generator version of a list comprehension.

```
>>> a = [1,2,3,4]
>>> b = (2*x for x in a)
>>> b
<generator object at 0x58760>
>>> for i in b:
...   print(i, end=' ')
...
2 4 6 8
>>>
```

Differences with List Comprehensions.

- Does not construct a list.
- Only useful purpose is iteration.
- Once consumed, can't be reused.

General syntax.

```
(<expression> for i in s if <conditional>)
```

It can also serve as a function argument.

```
sum(x*x for x in a)
```

It can be applied to any iterable.

```
>>> a = [1,2,3,4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c:
...   print(i, end=' ')
...
-1 -4 -9 -16
>>>
```

The main use of generator expressions is in code that performs some calculation on a sequence, but only uses the result once. For example, strip all comments from a file.

```
f = open('somefile.txt')
lines = (line for line in f if not line.startswith('#'))
for line in lines:
    ...
f.close()
```

With generators, the code runs faster and uses little memory. It's like a filter applied to a stream.

Why Generators

- Many problems are much more clearly expressed in terms of iteration.
  - Looping over a collection of items and performing some kind of operation (searching, replacing, modifying, etc.).
  - Processing pipelines can be applied to a wide range of data processing problems.
- Better memory efficiency.
  - Only produce values when needed.
  - Contrast to constructing giant lists.
  - Can operate on streaming data
- Generators encourage code reuse
  - Separates the *iteration* from code that uses the iteration
  - You can build a toolbox of interesting iteration functions and *mix-n-match*.

`itertools` module

The `itertools` is a library module with various functions designed to help with iterators/generators.

```
itertools.chain(s1,s2)
itertools.count(n)
itertools.cycle(s)
itertools.dropwhile(predicate, s)
itertools.groupby(s)
itertools.ifilter(predicate, s)
itertools.imap(function, s1, ... sN)
itertools.repeat(s, n)
itertools.tee(s, ncopies)
itertools.izip(s1, ... , sN)
```

All functions process data iteratively. They implement various kinds of iteration patterns.

More information at Generator Tricks for Systems Programmers tutorial from PyCon '08.

Exercises

In the previous exercises, you wrote some code that followed lines being written to a log file and parsed them into a sequence of rows. This exercise continues to build upon that. Make sure the `Data/stocksim.py` is still running.

Exercise 6.13: Generator Expressions

Generator expressions are a generator version of a list comprehension. For example:

```
>>> nums = [1, 2, 3, 4, 5]
>>> squares = (x*x for x in nums)
>>> squares
<generator object <genexpr> at 0x109207e60>
>>> for n in squares:
...     print(n)
...
1
4
9
16
25
```

Unlike a list a comprehension, a generator expression can only be used once. Thus, if you try another for-loop, you get nothing:

```
>>> for n in squares:
...     print(n)
...
>>>
```

Exercise 6.14: Generator Expressions in Function Arguments

Generator expressions are sometimes placed into function arguments. It looks a little weird at first, but try this experiment:

```
>>> nums = [1,2,3,4,5]
>>> sum([x*x for x in nums])    # A list comprehension
55
>>> sum(x*x for x in nums)      # A generator expression
55
>>>
```

In the above example, the second version using generators would use significantly less memory if a large list was being manipulated.

In your `portfolio.py` file, you performed a few calculations involving list comprehensions. Try replacing these with generator expressions.

Exercise 6.15: Code simplification

Generators expressions are often a useful replacement for small generator functions. For example, instead of writing a function like this:

```
def filter_symbols(rows, names):
    for row in rows:
        if row['name'] in names:
            yield row
```

You could write something like this:

```
rows = (row for row in rows if row['name'] in names)
```

Modify the `ticker.py` program to use generator expressions as appropriate.

Contents | Previous (6.3 Producer/Consumer) | Next (7 Advanced Topics)

Contents | Prev (6 Generators) | Next (8 Testing and Debugging)

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.

Contents | Prev (6 Generators) | Next (8 Testing and Debugging)

Contents | Previous (6.4 Generator Expressions) | Next (7.2 Anonymous Functions)

7.1 Variable Arguments

This section covers variadic function arguments, sometimes described as `*args` and `**kwargs`.

Positional variable arguments (*args)

A function that accepts *any number* of arguments is said to use variable arguments. For example:

```
def f(x, *args):
    ...
```

Function call.

```
f(1,2,3,4,5)
```

The extra arguments get passed as a tuple.

```
def f(x, *args):
    # x -> 1
    # args -> (2,3,4,5)
```

Keyword variable arguments (**kwargs)

A function can also accept any number of keyword arguments. For example:

```
def f(x, y, **kwargs):
    ...
```

Function call.

```
f(2, 3, flag=True, mode='fast', header='debug')
```

The extra keywords are passed in a dictionary.

```
def f(x, y, **kwargs):
    # x -> 2
    # y -> 3
    # kwargs -> { 'flag': True, 'mode': 'fast', 'header': 'debug' }
```

Combining both

A function can also accept any number of variable keyword and non-keyword arguments.

```
def f(*args, **kwargs):
    ...
```

Function call.

```
f(2, 3, flag=True, mode='fast', header='debug')
```

The arguments are separated into positional and keyword components

```
def f(*args, **kwargs):
    # args = (2, 3)
    # kwargs -> { 'flag': True, 'mode': 'fast', 'header': 'debug' }
    ...
```

This function takes any combination of positional or keyword arguments. It is sometimes used when writing wrappers or when you want to pass arguments through to another function.

Passing Tuples and Dicts

Tuples can be expanded into variable arguments.

```
numbers = (2,3,4)
f(1, *numbers)      # Same as f(1,2,3,4)
```

Dictionaries can also be expanded into keyword arguments.

```
options = {
    'color' : 'red',
    'delimiter' : ',',
    'width' : 400
}
f(data, **options)
# Same as f(data, color='red', delimiter=',', width=400)
```

Exercises

Exercise 7.1: A simple example of variable arguments

Try defining the following function:

```
>>> def avg(x,*more):
        return float(x+sum(more))/(1+len(more))

>>> avg(10,11)
10.5
>>> avg(3,4,5)
4.0
>>> avg(1,2,3,4,5,6)
3.5
>>>
```

Notice how the parameter `*more` collects all of the extra arguments.

Exercise 7.2: Passing tuple and dicts as arguments

Suppose you read some data from a file and obtained a tuple such as this:

```
>>> data = ('GOOG', 100, 490.1)
>>>
```

Now, suppose you wanted to create a `Stock` object from this data. If you try to pass `data` directly, it doesn't work:

```
>>> from stock import Stock
>>> s = Stock(data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 4 arguments (2 given)
>>>
```

This is easily fixed using `*data` instead. Try this:

```
>>> s = Stock(*data)
>>> s
Stock('GOOG', 100, 490.1)
>>>
```

If you have a dictionary, you can use `**` instead. For example:

```
>>> data = { 'name': 'GOOG', 'shares': 100, 'price': 490.1 }
>>> s = Stock(**data)
Stock('GOOG', 100, 490.1)
>>>
```

Exercise 7.3: Creating a list of instances

In your `report.py` program, you created a list of instances using code like this:

```
def read_portfolio(filename):
    '''
    Read a stock portfolio file into a list of dictionaries with keys
    name, shares, and price.
    '''
    with open(filename) as lines:
        portdicts = fileparse.parse_csv(lines,
                                select=['name','shares','price'],
                                types=[str,int,float])

    portfolio = [ Stock(d['name'], d['shares'], d['price'])
                  for d in portdicts ]
    return Portfolio(portfolio)
```

You can simplify that code using `Stock(**d)` instead. Make that change.

Exercise 7.4: Argument pass-through

The `fileparse.parse_csv()` function has some options for changing the file delimiter and for error reporting. Maybe you'd like to expose those options to the `read_portfolio()` function above. Make this change:

```
def read_portfolio(filename, **opts):
    '''
    Read a stock portfolio file into a list of dictionaries with keys
    name, shares, and price.
    '''
    with open(filename) as lines:
        portdicts = fileparse.parse_csv(lines,
                                        select=['name','shares','price'],
                                        types=[str,int,float],
                                        **opts)

    portfolio = [ Stock(**d) for d in portdicts ]
    return Portfolio(portfolio)
```

Once you've made the change, trying reading a file with some errors:

```
>>> import report
>>> port = report.read_portfolio('Data/missing.csv')
Row 4: Couldn't convert ['MSFT', '', '51.23']
Row 4: Reason invalid literal for int() with base 10: ''
Row 7: Couldn't convert ['IBM', '', '70.44']
Row 7: Reason invalid literal for int() with base 10: ''
>>>
```

Now, try silencing the errors:

```
>>> import report
>>> port = report.read_portfolio('Data/missing.csv', silence_errors=True)
>>>
```

Contents | Previous (6.4 Generator Expressions) | Next (7.2 Anonymous Functions)

Contents | Previous (7.1 Variable Arguments) | Next (7.3 Returning Functions)

7.2 Anonymous Functions and Lambda

List Sorting Revisited

Lists can be sorted *in-place*. Using the `sort` method.

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

You can sort in reverse order.

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

It seems simple enough. However, how do we sort a list of dicts?

```
[{'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}]
```

By what criteria?

You can guide the sorting by using a *key function*. The *key function* is a function that receives the dictionary and returns the value of interest for sorting.

```
def stock_name(s):
    return s['name']

portfolio.sort(key=stock_name)
```

Here's the result.

```
# Check how the dictionaries are sorted by the `name` key
[
  {'name': 'AA', 'price': 32.2, 'shares': 100},
  {'name': 'CAT', 'price': 83.44, 'shares': 150},
  {'name': 'GE', 'price': 40.37, 'shares': 95},
  {'name': 'IBM', 'price': 91.1, 'shares': 50},
  {'name': 'IBM', 'price': 70.44, 'shares': 100},
  {'name': 'MSFT', 'price': 51.23, 'shares': 200},
  {'name': 'MSFT', 'price': 65.1, 'shares': 50}
]
```

Callback Functions

In the above example, the key function is an example of a callback function. The `sort()` method "calls back" to a function you supply. Callback functions are often short one-line functions that are only used for that one operation. Programmers often ask for a short-cut for specifying this extra processing.

Lambda: Anonymous Functions

Use a lambda instead of creating the function. In our previous sorting example.

```
portfolio.sort(key=lambda s: s['name'])
```

This creates an *unnamed* function that evaluates a *single* expression. The above code is much shorter than the initial code.

```
def stock_name(s):
    return s['name']

portfolio.sort(key=stock_name)

# vs lambda
portfolio.sort(key=lambda s: s['name'])
```

Using lambda

- lambda is highly restricted.
- Only a single expression is allowed.
- No statements like `if`, `while`, etc.
- Most common use is with functions like `sort()`.

Exercises

Read some stock portfolio data and convert it into a list:

```
>>> import report
>>> portfolio = list(report.read_portfolio('Data/portfolio.csv'))
>>> for s in portfolio:
        print(s)

Stock('AA', 100, 32.2)
Stock('IBM', 50, 91.1)
Stock('CAT', 150, 83.44)
Stock('MSFT', 200, 51.23)
Stock('GE', 95, 40.37)
Stock('MSFT', 50, 65.1)
Stock('IBM', 100, 70.44)
>>>
```

Exercise 7.5: Sorting on a field

Try the following statements which sort the portfolio data alphabetically by stock name.

```
>>> def stock_name(s):
       return s.name

>>> portfolio.sort(key=stock_name)
>>> for s in portfolio:
           print(s)

... inspect the result ...
>>>
```

In this part, the `stock_name()` function extracts the name of a stock from a single entry in the `portfolio` list. `sort()` uses the result of this function to do the comparison.

Exercise 7.6: Sorting on a field with lambda

Try sorting the portfolio according the number of shares using a `lambda` expression:

```
>>> portfolio.sort(key=lambda s: s.shares)
>>> for s in portfolio:
        print(s)

... inspect the result ...
>>>
```

Try sorting the portfolio according to the price of each stock

```
>>> portfolio.sort(key=lambda s: s.price)
>>> for s in portfolio:
        print(s)

... inspect the result ...
>>>
```

Note: `lambda` is a useful shortcut because it allows you to define a special processing function directly in the call to `sort()` as opposed to having to define a separate function first.

Contents | Previous (7.1 Variable Arguments) | Next (7.3 Returning Functions)

Contents | Previous (7.2 Anonymous Functions) | Next (7.4 Decorators)

7.3 Returning Functions

This section introduces the idea of using functions to create other functions.

Introduction

Consider the following function.

```
def add(x, y):
    def do_add():
        print('Adding', x, y)
        return x + y
    return do_add
```

This is a function that returns another function.

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
Adding 3 4
7
```

Local Variables

Observe how the inner function refers to variables defined by the outer function.

```
def add(x, y):
    def do_add():
        # `x` and `y` are defined above `add(x, y)`
        print('Adding', x, y)
        return x + y
    return do_add
```

Further observe that those variables are somehow kept alive after `add()` has finished.

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
Adding 3 4      # Where are these values coming from?
7
```

Closures

When an inner function is returned as a result, that inner function is known as a *closure*.

```
def add(x, y):
    # `do_add` is a closure
    def do_add():
        print('Adding', x, y)
        return x + y
    return do_add
```

Essential feature: A closure retains the values of all variables needed for the function to run properly later on. Think of a closure as a function plus an extra environment that holds the values of variables that it depends on.

Using Closures

Closure are an essential feature of Python. However, their use if often subtle. Common applications:

- Use in callback functions.
- Delayed evaluation.
- Decorator functions (later).

Delayed Evaluation

Consider a function like this:

```
def after(seconds, func):
    import time
    time.sleep(seconds)
    func()
```

Usage example:

```
def greeting():
    print('Hello Guido')

after(30, greeting)
```

`after` executes the supplied function... later.

Closures carry extra information around.

```
def add(x, y):
    def do_add():
        print(f'Adding {x} + {y} -> {x+y}')
    return do_add

def after(seconds, func):
    import time
    time.sleep(seconds)
    func()

after(30, add(2, 3))
# `do_add` has the references x -> 2 and y -> 3
```

Code Repetition

Closures can also be used as technique for avoiding excessive code repetition. You can write functions that make code.

Exercises

Exercise 7.7: Using Closures to Avoid Repetition

One of the more powerful features of closures is their use in generating repetitive code. If you refer back to Exercise 5.7, recall the code for defining a property with type checking.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
    ...
    @property
```

```
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected int')
        self._shares = value
    ...
```

Instead of repeatedly typing that code over and over again, you can automatically create it using a closure.

Make a file `typedproperty.py` and put the following code in it:

```
# typedproperty.py

def typedproperty(name, expected_type):
    private_name = '_' + name
    @property
    def prop(self):
        return getattr(self, private_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
            raise TypeError(f'Expected {expected_type}')
        setattr(self, private_name, value)

    return prop
```

Now, try it out by defining a class like this:

```
from typedproperty import typedproperty

class Stock:
    name = typedproperty('name', str)
    shares = typedproperty('shares', int)
    price = typedproperty('price', float)

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Try creating an instance and verifying that type-checking works.

```
>>> s = Stock('IBM', 50, 91.1)
>>> s.name
'IBM'
>>> s.shares = '100'
... should get a TypeError ...
>>>
```

Exercise 7.8: Simplifying Function Calls

In the above example, users might find calls such as `typedproperty('shares', int)` a bit verbose to type--especially if they're repeated a lot. Add the following definitions to the `typedproperty.py` file:

```
String = lambda name: typedproperty(name, str)
Integer = lambda name: typedproperty(name, int)
Float = lambda name: typedproperty(name, float)
```

Now, rewrite the `Stock` class to use these functions instead:

```
class Stock:
    name = String('name')
    shares = Integer('shares')
    price = Float('price')

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Ah, that's a bit better. The main takeaway here is that closures and `lambda` can often be used to simplify code and eliminate annoying repetition. This is often good.

Exercise 7.9: Putting it into practice

Rewrite the `Stock` class in the file `stock.py` so that it uses typed properties as shown.

Contents | Previous (7.2 Anonymous Functions) | Next (7.4 Decorators)

Contents | Previous (7.3 Returning Functions) | Next (7.5 Decorated Methods)

7.4 Function Decorators

This section introduces the concept of a decorator. This is an advanced topic for which we only scratch the surface.

Logging Example

Consider a function.

```
def add(x, y):
    return x + y
```

Now, consider the function with some logging added to it.

```
def add(x, y):
    print('Calling add')
    return x + y
```

Now a second function also with some logging.

```
def sub(x, y):
    print('Calling sub')
    return x - y
```

Observation

Observation: It's kind of repetitive.

Writing programs where there is a lot of code replication is often really annoying. They are tedious to write and hard to maintain. Especially if you decide that you want to change how it works (i.e., a different kind of logging perhaps).

Code that makes logging

Perhaps you can make a function that makes functions with logging added to them. A wrapper.

```
def logged(func):
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Now use it.

```
def add(x, y):
    return x + y

logged_add = logged(add)
```

What happens when you call the function returned by `logged`?

```
logged_add(3, 4)      # You see the logging message appear
```

This example illustrates the process of creating a so-called *wrapper function*.

A wrapper is a function that wraps around another function with some extra bits of processing, but otherwise works in the exact same way as the original function.

```
>>> logged_add(3, 4)
Calling add   # Extra output. Added by the wrapper
7
>>>
```

Note: The `logged()` function creates the wrapper and returns it as a result.

Decorators

Putting wrappers around functions is extremely common in Python. So common, there is a special syntax for it.

```
def add(x, y):
    return x + y
add = logged(add)

# Special syntax
@logged
def add(x, y):
    return x + y
```

The special syntax performs the same exact steps as shown above. A decorator is just new syntax. It is said to *decorate* the function.

Commentary

There are many more subtle details to decorators than what has been presented here. For example, using them in classes. Or using multiple decorators with a function. However, the previous example is a good illustration of how their use tends to arise. Usually, it's in response to repetitive code appearing across a wide range of function definitions. A decorator can move that code to a central definition.

Exercises

Exercise 7.10: A decorator for timing

If you define a function, its name and module are stored in the `__name__` and `__module__` attributes. For example:

```
>>> def add(x,y):
        return x+y

>>> add.__name__
'add'
>>> add.__module__
'__main__'
>>>
```

In a file `timethis.py`, write a decorator function `timethis(func)` that wraps a function with an extra layer of logic that prints out how long it takes for a function to execute. To do this, you'll surround the function with timing calls like this:

```
start = time.time()
r = func(*args,**kwargs)
end = time.time()
print('%s.%s: %f' % (func.__module__, func.__name__, end-start))
```

Here is an example of how your decorator should work:

```
>>> from timethis import timethis
>>> @timethis
def countdown(n):
    while n > 0:
        n -= 1

>>> countdown(10000000)
__main__.countdown : 0.076562
>>>
```

Discussion: This `@timethis` decorator can be placed in front of any function definition. Thus, you might use it as a diagnostic tool for performance tuning.

Contents | Previous (7.3 Returning Functions) | Next (7.5 Decorated Methods)

Contents | Previous (7.4 Decorators) | Next (8 Testing and Debugging)

7.5 Decorated Methods

This section discusses a few built-in decorators that are used in combination with method definitions.

Predefined Decorators

There are predefined decorators used to specify special kinds of methods in class definitions.

```
class Foo:
    def bar(self,a):
        ...

    @staticmethod
    def spam(a):
        ...

    @classmethod
    def grok(cls,a):
        ...

    @property
    def name(self):
        ...
```

Let's go one by one.

Static Methods

`@staticmethod` is used to define a so-called *static* class methods (from C++/Java). A static method is a function that is part of the class, but which does *not* operate on instances.

```
class Foo(object):
    @staticmethod
    def bar(x):
        print('x =', x)

>>> Foo.bar(2) x=2
>>>
```

Static methods are sometimes used to implement internal supporting code for a class. For example, code to help manage created instances (memory management, system resources, persistence, locking, etc). They're also used by certain design patterns (not discussed here).

Class Methods

`@classmethod` is used to define class methods. A class method is a method that receives the *class* object as the first parameter instead of the instance.

```
class Foo:
    def bar(self):
        print(self)

    @classmethod
    def spam(cls):
        print(cls)

>>> f = Foo()
>>> f.bar()
<__main__.Foo object at 0x971690>   # The instance `f`
>>> Foo.spam()
<class '__main__.Foo'>              # The class `Foo`
>>>
```

Class methods are most often used as a tool for defining alternate constructors.

```
class Date:
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        # Notice how the class is passed as an argument
```

```
        tm = time.localtime()
        # And used to create a new instance
        return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)

d = Date.today()
```

Class methods solve some tricky problems with features like inheritance.

```
class Date:
    ...
    @classmethod
    def today(cls):
        # Gets the correct class (e.g. `NewDate`)
        tm = time.localtime()
        return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)

class NewDate(Date):
    ...

d = NewDate.today()
```

Exercises

Exercise 7.11: Class Methods in Practice

In your `report.py` and `portfolio.py` files, the creation of a `Portfolio` object is a bit muddled. For example, the `report.py` program has code like this:

```
def read_portfolio(filename, **opts):
    '''
    Read a stock portfolio file into a list of dictionaries with keys
    name, shares, and price.
    '''
    with open(filename) as lines:
        portdicts = fileparse.parse_csv(lines,
                                        select=['name','shares','price'],
                                        types=[str,int,float],
                                        **opts)

    portfolio = [ Stock(**d) for d in portdicts ]
    return Portfolio(portfolio)
```

and the `portfolio.py` file defines `Portfolio()` with an odd initializer like this:

```
class Portfolio:
    def __init__(self, holdings):
        self.holdings = holdings
    ...
```

Frankly, the chain of responsibility is all a bit confusing because the code is scattered. If a `Portfolio` class is supposed to contain a list of `Stock` instances, maybe you should change the class to be a bit more clear. Like this:

```
# portfolio.py

import stock

class Portfolio:
    def __init__(self):
        self.holdings = []

    def append(self, holding):
        if not isinstance(holding, stock.Stock):
            raise TypeError('Expected a Stock instance')
        self.holdings.append(holding)
    ...
```

If you want to read a portfolio from a CSV file, maybe you should make a class method for it:

```
# portfolio.py

import fileparse
import stock

class Portfolio:
    def __init__(self):
        self.holdings = []

    def append(self, holding):
        if not isinstance(holding, stock.Stock):
            raise TypeError('Expected a Stock instance')
        self.holdings.append(holding)

    @classmethod
    def from_csv(cls, lines, **opts):
        self = cls()
        portdicts = fileparse.parse_csv(lines,
                                        select=['name','shares','price'],
                                        types=[str,int,float],
                                        **opts)

        for d in portdicts:
            self.append(stock.Stock(**d))

        return self
```

To use this new Portfolio class, you can now write code like this:

```
>>> from portfolio import Portfolio
>>> with open('Data/portfolio.csv') as lines:
...     port = Portfolio.from_csv(lines)
...
>>>
```

Make these changes to the `Portfolio` class and modify the `report.py` code to use the class method.

Contents | Previous (7.4 Decorators) | Next (8 Testing and Debugging)

Contents | Prev (7 Advanced Topics) | Next (9 Packages)

8. Testing and debugging

This section introduces a few basic topics related to testing, logging, and debugging.

Contents | Prev (7 Advanced Topics) | Next (9 Packages)

Contents | Previous (7.5 Decorated Methods) | Next (8.2 Logging)

8.1 Testing

Testing Rocks, Debugging Sucks

The dynamic nature of Python makes testing critically important to most applications. There is no compiler to find your bugs. The only way to find bugs is to run the code and make sure you try out all of its features.

Assertions

The `assert` statement is an internal check for the program. If an expression is not true, it raises a `AssertionError` exception.

`assert` statement syntax.

```
assert <expression> [, 'Diagnostic message']
```

For example.

```
assert isinstance(10, int), 'Expected int'
```

It shouldn't be used to check the user-input (i.e., data entered on a web form or something). It's purpose is more for internal checks and invariants (conditions that should always be true).

Contract Programming

Also known as Design By Contract, liberal use of assertions is an approach for designing software. It prescribes that software designers should define precise interface specifications for the components of the software.

For example, you might put assertions on all inputs of a function.

```
def add(x, y):
    assert isinstance(x, int), 'Expected int'
    assert isinstance(y, int), 'Expected int'
    return x + y
```

Checking inputs will immediately catch callers who aren't using appropriate arguments.

```
>>> add(2, 3)
5
>>> add('2', '3')
Traceback (most recent call last):
...
AssertionError: Expected int
>>>
```

Inline Tests

Assertions can also be used for simple tests.

```
def add(x, y):
    return x + y

assert add(2,2) == 4
```

This way you are including the test in the same module as your code.

Benefit: If the code is obviously broken, attempts to import the module will crash.

This is not recommended for exhaustive testing. It's more of a basic "smoke test". Does the function work on any example at all? If not, then something is definitely broken.

`unittest` Module

Suppose you have some code.

```
# simple.py

def add(x, y):
    return x + y
```

Now, suppose you want to test it. Create a separate testing file like this.

```
# test_simple.py

import simple
import unittest
```

Then define a testing class.

```
# test_simple.py

import simple
import unittest

# Notice that it inherits from unittest.TestCase
class TestAdd(unittest.TestCase):
    ...
```

The testing class must inherit from `unittest.TestCase`.

In the testing class, you define the testing methods.

```
# test_simple.py

import simple
import unittest

# Notice that it inherits from unittest.TestCase
class TestAdd(unittest.TestCase):
    def test_simple(self):
        # Test with simple integer arguments
        r = simple.add(2, 2)
        self.assertEqual(r, 5)
    def test_str(self):
        # Test with strings
        r = simple.add('hello', 'world')
        self.assertEqual(r, 'helloworld')
```

*Important: Each method must start with `test`.

Using `unittest`

There are several built in assertions that come with `unittest`. Each of them asserts a different thing.

```
# Assert that expr is True
self.assertTrue(expr)

# Assert that x == y
self.assertEqual(x,y)

# Assert that x != y
self.assertNotEqual(x,y)
```

```
# Assert that x is near y
self.assertAlmostEqual(x,y,places)

# Assert that callable(arg1,arg2,...) raises exc
self.assertRaises(exc, callable, arg1, arg2, ...)
```

This is not an exhaustive list. There are other assertions in the module.

Running `unittest`

To run the tests, turn the code into a script.

```
# test_simple.py

...

if __name__ == '__main__':
    unittest.main()
```

Then run Python on the test file.

```
bash % python3 test_simple.py
F.
======================================================
FAIL: test_simple (__main__.TestAdd)
------------------------------------------------------
Traceback (most recent call last):
  File "testsimple.py", line 8, in test_simple
    self.assertEqual(r, 5)
AssertionError: 4 != 5
------------------------------------------------------
Ran 2 tests in 0.000s
FAILED (failures=1)
```

Commentary

Effective unit testing is an art and it can grow to be quite complicated for large applications.

The `unittest` module has a huge number of options related to test runners, collection of results and other aspects of testing. Consult the documentation for details.

Third Party Test Tools

The built-in `unittest` module has the advantage of being available everywhere--it's part of Python. However, many programmers also find it to be quite verbose. A popular alternative is pytest. With pytest, your testing file simplifies to something like the following:

```
# test_simple.py
import simple

def test_simple():
    assert simple.add(2,2) == 4

def test_str():
    assert simple.add('hello','world') == 'helloworld'
```

To run the tests, you simply type a command such as `python -m pytest`. It will discover all of the tests and run them.

There's a lot more to `pytest` than this example, but it's usually pretty easy to get started should you decide to try it out.

Exercises

In this exercise, you will explore the basic mechanics of using Python's `unittest` module.

In earlier exercises, you wrote a file `stock.py` that contained a `Stock` class. For this exercise, it assumed that you're using the code written for Exercise 7.9 involving typed-properties. If, for some reason, that's not working, you might want to copy the solution from `Solutions/7_9` to your working directory.

Exercise 8.1: Writing Unit Tests

In a separate file `test_stock.py`, write a set a unit tests for the `Stock` class. To get you started, here is a small fragment of code that tests instance creation:

```
# test_stock.py

import unittest
import stock

class TestStock(unittest.TestCase):
    def test_create(self):
        s = stock.Stock('GOOG', 100, 490.1)
        self.assertEqual(s.name, 'GOOG')
        self.assertEqual(s.shares, 100)
        self.assertEqual(s.price, 490.1)

if __name__ == '__main__':
    unittest.main()
```

Run your unit tests. You should get some output that looks like this:

```
.
----------------------------------------------------------------------
Ran 1 tests in 0.000s

OK
```

Once you're satisfied that it works, write additional unit tests that check for the following:

- Make sure the `s.cost` property returns the correct value (49010.0)
- Make sure the `s.sell()` method works correctly. It should decrement the value of `s.shares` accordingly.
- Make sure that the `s.shares` attribute can't be set to a non-integer value.

For the last part, you're going to need to check that an exception is raised. An easy way to do that is with code like this:

```
class TestStock(unittest.TestCase):
    ...
    def test_bad_shares(self):
         s = stock.Stock('GOOG', 100, 490.1)
         with self.assertRaises(TypeError):
             s.shares = '100'
```

Contents | Previous (7.5 Decorated Methods) | Next (8.2 Logging)

Contents | Previous (8.1 Testing) | Next (8.3 Debugging)

8.2 Logging

This section briefly introduces the logging module.

logging Module

The `logging` module is a standard library module for recording diagnostic information. It's also a very large module with a lot of sophisticated functionality. We will show a simple example to illustrate its usefulness.

Exceptions Revisited

In the exercises, we wrote a function `parse()` that looked something like this:

```
# fileparse.py
def parse(f, types=None, names=None, delimiter=None):
    records = []
    for line in f:
        line = line.strip()
        if not line: continue
        try:
            records.append(split(line,types,names,delimiter))
        except ValueError as e:
            print("Couldn't parse :", line)
            print("Reason :", e)
    return records
```

Focus on the `try-except` statement. What should you do in the `except` block?

Should you print a warning message?

```
try:
    records.append(split(line,types,names,delimiter))
except ValueError as e:
    print("Couldn't parse :", line)
    print("Reason :", e)
```

Or do you silently ignore it?

```
try:
    records.append(split(line,types,names,delimiter))
except ValueError as e:
    pass
```

Neither solution is satisfactory because you often want *both* behaviors (user selectable).

Using logging

The `logging` module can address this.

```
# fileparse.py
import logging
log = logging.getLogger(__name__)

def parse(f,types=None,names=None,delimiter=None):
    ...
    try:
        records.append(split(line,types,names,delimiter))
    except ValueError as e:
        log.warning("Couldn't parse : %s", line)
        log.debug("Reason : %s", e)
```

The code is modified to issue warning messages or a special `Logger` object. The one created with `logging.getLogger(__name__)`.

Logging Basics

Create a logger object.

```
log = logging.getLogger(name)   # name is a string
```

Issuing log messages.

```
log.critical(message [, args])
log.error(message [, args])
log.warning(message [, args])
log.info(message [, args])
log.debug(message [, args])
```

Each method represents a different level of severity.

All of them create a formatted log message. `args` is used with the `%` operator to create the message.

```
logmsg = message % args # Written to the log
```

Logging Configuration

The logging behavior is configured separately.

```
# main.py

...

if __name__ == '__main__':
    import logging
    logging.basicConfig(
        filename  = 'app.log',      # Log output file
        level     = logging.INFO,   # Output level
    )
```

Typically, this is a one-time configuration at program startup. The configuration is separate from the code that makes the logging calls.

Comments

Logging is highly configurable. You can adjust every aspect of it: output files, levels, message formats, etc. However, the code that uses logging doesn't have to worry about that.

Exercises

Exercise 8.2: Adding logging to a module

In `fileparse.py`, there is some error handling related to exceptions caused by bad input. It looks like this:

```
# fileparse.py
import csv

def parse_csv(lines, select=None, types=None, has_headers=True, delimiter=',',
silence_errors=False):
    '''
    Parse a CSV file into a list of records with type conversion.
    '''
    if select and not has_headers:
        raise RuntimeError('select requires column headers')

    rows = csv.reader(lines, delimiter=delimiter)

    # Read the file headers (if any)
    headers = next(rows) if has_headers else []

    # If specific columns have been selected, make indices for filtering and set output
```

```
columns
    if select:
        indices = [ headers.index(colname) for colname in select ]
        headers = select

    records = []
    for rowno, row in enumerate(rows, 1):
        if not row:     # Skip rows with no data
            continue

        # If specific column indices are selected, pick them out
        if select:
            row = [ row[index] for index in indices]

        # Apply type conversion to the row
        if types:
            try:
                row = [func(val) for func, val in zip(types, row)]
            except ValueError as e:
                if not silence_errors:
                    print(f"Row {rowno}: Couldn't convert {row}")
                    print(f"Row {rowno}: Reason {e}")
                continue

        # Make a dictionary or a tuple
        if headers:
            record = dict(zip(headers, row))
        else:
            record = tuple(row)
        records.append(record)

    return records
```

Notice the print statements that issue diagnostic messages. Replacing those prints with logging operations is relatively simple. Change the code like this:

```
# fileparse.py
import csv
import logging
log = logging.getLogger(__name__)

def parse_csv(lines, select=None, types=None, has_headers=True, delimiter=',',
silence_errors=False):
    '''
    Parse a CSV file into a list of records with type conversion.
    '''
    if select and not has_headers:
        raise RuntimeError('select requires column headers')

    rows = csv.reader(lines, delimiter=delimiter)

    # Read the file headers (if any)
    headers = next(rows) if has_headers else []
```

```
    # If specific columns have been selected, make indices for filtering and set output
columns
    if select:
        indices = [ headers.index(colname) for colname in select ]
        headers = select

    records = []
    for rowno, row in enumerate(rows, 1):
        if not row:     # Skip rows with no data
            continue

        # If specific column indices are selected, pick them out
        if select:
            row = [ row[index] for index in indices]

        # Apply type conversion to the row
        if types:
            try:
                row = [func(val) for func, val in zip(types, row)]
            except ValueError as e:
                if not silence_errors:
                    log.warning("Row %d: Couldn't convert %s", rowno, row)
                    log.debug("Row %d: Reason %s", rowno, e)
                continue

        # Make a dictionary or a tuple
        if headers:
            record = dict(zip(headers, row))
        else:
            record = tuple(row)
        records.append(record)

    return records
```

Now that you've made these changes, try using some of your code on bad data.

```
>>> import report
>>> a = report.read_portfolio('Data/missing.csv')
Row 4: Bad row: ['MSFT', '', '51.23']
Row 7: Bad row: ['IBM', '', '70.44']
>>>
```

If you do nothing, you'll only get logging messages for the `WARNING` level and above. The output will look like simple print statements. However, if you configure the logging module, you'll get additional information about the logging levels, module, and more. Type these steps to see that:

```
>>> import logging
>>> logging.basicConfig()
>>> a = report.read_portfolio('Data/missing.csv')
WARNING:fileparse:Row 4: Bad row: ['MSFT', '', '51.23']
WARNING:fileparse:Row 7: Bad row: ['IBM', '', '70.44']
>>>
```

You will notice that you don't see the output from the `log.debug()` operation. Type this to change the level.

```
>>> logging.getLogger('fileparse').setLevel(logging.DEBUG)
>>> a = report.read_portfolio('Data/missing.csv')
WARNING:fileparse:Row 4: Bad row: ['MSFT', '', '51.23']
DEBUG:fileparse:Row 4: Reason: invalid literal for int() with base 10: ''
WARNING:fileparse:Row 7: Bad row: ['IBM', '', '70.44']
DEBUG:fileparse:Row 7: Reason: invalid literal for int() with base 10: ''
>>>
```

Turn off all, but the most critical logging messages:

```
>>> logging.getLogger('fileparse').setLevel(logging.CRITICAL)
>>> a = report.read_portfolio('Data/missing.csv')
>>>
```

Exercise 8.3: Adding Logging to a Program

To add logging to an application, you need to have some mechanism to initialize the logging module in the main module. One way to do this is to include some setup code that looks like this:

```
# This file sets up basic configuration of the logging module.
# Change settings here to adjust logging output as needed.
import logging
logging.basicConfig(
    filename = 'app.log',            # Name of the log file (omit to use stderr)
    filemode = 'w',                  # File mode (use 'a' to append)
    level    = logging.WARNING,      # Logging level (DEBUG, INFO, WARNING, ERROR, or
CRITICAL)
)
```

Again, you'd need to put this someplace in the startup steps of your program. For example, where would you put this in your `report.py` program?

Contents | Previous (8.1 Testing) | Next (8.3 Debugging)

Contents | Previous (8.2 Logging) | Next (9 Packages)

8.3 Debugging

Debugging Tips

So, your program has crashed...

```
bash % python3 blah.py
Traceback (most recent call last):
  File "blah.py", line 13, in ?
    foo()
  File "blah.py", line 10, in foo
    bar()
  File "blah.py", line 7, in bar
    spam()
  File "blah.py", 4, in spam
    line x.append(3)
AttributeError: 'int' object has no attribute 'append'
```

Now what?!

Reading Tracebacks

The last line is the specific cause of the crash.

```
bash % python3 blah.py
Traceback (most recent call last):
  File "blah.py", line 13, in ?
    foo()
  File "blah.py", line 10, in foo
    bar()
  File "blah.py", line 7, in bar
    spam()
  File "blah.py", 4, in spam
    line x.append(3)
# Cause of the crash
AttributeError: 'int' object has no attribute 'append'
```

However, it's not always easy to read or understand.

PRO TIP: Paste the whole traceback into Google.

Using the REPL

Use the option `-i` to keep Python alive when executing a script.

```
bash % python3 -i blah.py
Traceback (most recent call last):
  File "blah.py", line 13, in ?
    foo()
  File "blah.py", line 10, in foo
    bar()
  File "blah.py", line 7, in bar
    spam()
  File "blah.py", 4, in spam
    line x.append(3)
AttributeError: 'int' object has no attribute 'append'
>>>
```

It preserves the interpreter state. That means that you can go poking around after the crash. Checking variable values and other state.

Debugging with Print

`print()` debugging is quite common.

Tip: Make sure you use `repr()`

```
def spam(x):
    print('DEBUG:', repr(x))
    ...
```

`repr()` shows you an accurate representation of a value. Not the *nice* printing output.

```
>>> from decimal import Decimal
>>> x = Decimal('3.4')
# NO `repr`
>>> print(x)
3.4
# WITH `repr`
>>> print(repr(x))
Decimal('3.4')
>>>
```

The Python Debugger

You can manually launch the debugger inside a program.

```
def some_function():
    ...
    breakpoint()      # Enter the debugger (Python 3.7+)
    ...
```

This starts the debugger at the `breakpoint()` call.

In earlier Python versions, you did this. You'll sometimes see this mentioned in other debugging guides.

```
import pdb
...
pdb.set_trace()       # Instead of `breakpoint()`
...
```

Run under debugger

You can also run an entire program under debugger.

```
bash % python3 -m pdb someprogram.py
```

It will automatically enter the debugger before the first statement. Allowing you to set breakpoints and change the configuration.

Common debugger commands:

```
(Pdb) help            # Get help
(Pdb) w(here)         # Print stack trace
(Pdb) d(own)          # Move down one stack level
(Pdb) u(p)            # Move up one stack level
(Pdb) b(reak) loc     # Set a breakpoint
(Pdb) s(tep)          # Execute one instruction
(Pdb) c(ontinue)      # Continue execution
(Pdb) l(ist)          # List source code
(Pdb) a(rgs)          # Print args of current function
(Pdb) !statement      # Execute statement
```

For breakpoints location is one of the following.

```
(Pdb) b 45            # Line 45 in current file
(Pdb) b file.py:45    # Line 45 in file.py
(Pdb) b foo           # Function foo() in current file
(Pdb) b module.foo    # Function foo() in a module
```

Exercises

Exercise 8.4: Bugs? What Bugs?

It runs. Ship it!

Contents | Previous (8.2 Logging) | Next (9 Packages)

Contents | Prev (8 Testing and Debugging)

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)

Contents | Previous (8.3 Debugging) | Next (9.2 Third Party Packages)

9.1 Packages

If writing a larger program, you don't really want to organize it as a large of collection of standalone files at the top level. This section introduces the concept of a package.

Modules

Any Python source file is a module.

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

An `import` statement loads and *executes* a module.

```
# program.py
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...
```

Packages vs Modules

For larger collections of code, it is common to organize modules into a package.

```
# From this
pcost.py
report.py
fileparse.py

# To this
porty/
    __init__.py
    pcost.py
    report.py
    fileparse.py
```

You pick a name and make a top-level directory. `porty` in the example above (clearly picking this name is the most important first step).

Add an `__init__.py` file to the directory. It may be empty.

Put your source files into the directory.

Using a Package

A package serves as a namespace for imports.

This means that there are now multilevel imports.

```
import porty.report
port = porty.report.read_portfolio('port.csv')
```

There are other variations of import statements.

```
from porty import report
port = report.read_portfolio('portfolio.csv')

from porty.report import read_portfolio
port = read_portfolio('portfolio.csv')
```

Two problems

There are two main problems with this approach.

- imports between files in the same package break.
- Main scripts placed inside the package break.

So, basically everything breaks. But, other than that, it works.

Problem: Imports

Imports between files in the same package *must now include the package name in the import*. Remember the structure.

```
porty/
    __init__.py
    pcost.py
    report.py
    fileparse.py
```

Modified import example.

```
# report.py
from porty import fileparse

def read_portfolio(filename):
    return fileparse.parse_csv(...)
```

All imports are *absolute*, not relative.

```
# report.py
import fileparse    # BREAKS. fileparse not found

...
```

Relative Imports

Instead of directly using the package name, you can use `.` to refer to the current package.

```
# report.py
from . import fileparse

def read_portfolio(filename):
    return fileparse.parse_csv(...)
```

Syntax:

```
from . import modname
```

This makes it easy to rename the package.

Problem: Main Scripts

Running a package submodule as a main script breaks.

```
bash $ python porty/pcost.py # BREAKS
...
```

Reason: You are running Python on a single file and Python doesn't see the rest of the package structure correctly (`sys.path` is wrong).

All imports break. To fix this, you need to run your program in a different way, using the `-m` option.

```
bash $ python -m porty.pcost # WORKS
...
```

`__init__.py` files

The primary purpose of these files is to stitch modules together.

Example: consolidating functions

```
# porty/__init__.py
from .pcost import portfolio_cost
from .report import portfolio_report
```

This makes names appear at the *top-level* when importing.

```
from porty import portfolio_cost
portfolio_cost('portfolio.csv')
```

Instead of using the multilevel imports.

```
from porty import pcost
pcost.portfolio_cost('portfolio.csv')
```

Another solution for scripts

As noted, you now need to use `-m package.module` to run scripts within your package.

```
bash % python3 -m porty.pcost portfolio.csv
```

There is another alternative: Write a new top-level script.

```
#!/usr/bin/env python3
# pcost.py
import porty.pcost
import sys
porty.pcost.main(sys.argv)
```

This script lives *outside* the package. For example, looking at the directory structure:

```
pcost.py       # top-level-script
porty/         # package directory
    __init__.py
    pcost.py
    ...
```

Application Structure

Code organization and file structure is key to the maintainability of an application.

There is no "one-size fits all" approach for Python. However, one structure that works for a lot of problems is something like this.

```
porty-app/
  README.txt
  script.py         # SCRIPT
  porty/
    # LIBRARY CODE
    __init__.py
    pcost.py
    report.py
    fileparse.py
```

The top-level `porty-app` is a container for everything else--documentation, top-level scripts, examples, etc.

Again, top-level scripts (if any) need to exist outside the code package. One level up.

```
#!/usr/bin/env python3
# porty-app/script.py
import sys
import porty

porty.report.main(sys.argv)
```

Exercises

At this point, you have a directory with several programs:

```
pcost.py          # computes portfolio cost
report.py         # Makes a report
ticker.py         # Produce a real-time stock ticker
```

There are a variety of supporting modules with other functionality:

```
stock.py          # Stock class
portfolio.py      # Portfolio class
fileparse.py      # CSV parsing
tableformat.py    # Formatted tables
follow.py         # Follow a log file
typedproperty.py  # Typed class properties
```

In this exercise, we're going to clean up the code and put it into a common package.

Exercise 9.1: Making a simple package

Make a directory called `porty/` and put all of the above Python files into it. Additionally create an empty `__init__.py` file and put it in the directory. You should have a directory of files like this:

```
porty/
    __init__.py
    fileparse.py
    follow.py
    pcost.py
    portfolio.py
    report.py
    stock.py
    tableformat.py
    ticker.py
    typedproperty.py
```

Remove the file `__pycache__` that's sitting in your directory. This contains pre-compiled Python modules from before. We want to start fresh.

Try importing some of package modules:

```
>>> import porty.report
>>> import porty.pcost
>>> import porty.ticker
```

If these imports fail, go into the appropriate file and fix the module imports to include a package-relative import. For example, a statement such as `import fileparse` might change to the following:

```
# report.py
from . import fileparse
...
```

If you have a statement such as `from fileparse import parse_csv`, change the code to the following:

```
# report.py
from .fileparse import parse_csv
...
```

Exercise 9.2: Making an application directory

Putting all of your code into a "package" isn't often enough for an application. Sometimes there are supporting files, documentation, scripts, and other things. These files need to exist OUTSIDE of the `porty/` directory you made above.

Create a new directory called `porty-app`. Move the `porty` directory you created in Exercise 9.1 into that directory. Copy the `Data/portfolio.csv` and `Data/prices.csv` test files into this directory. Additionally create a `README.txt` file with some information about yourself. Your code should now be organized as follows:

```
porty-app/
    portfolio.csv
    prices.csv
    README.txt
    porty/
```

```
        __init__.py
        fileparse.py
        follow.py
        pcost.py
        portfolio.py
        report.py
        stock.py
        tableformat.py
        ticker.py
        typedproperty.py
```

To run your code, you need to make sure you are working in the top-level `porty-app/` directory. For example, from the terminal:

```
shell % cd porty-app
shell % python3
>>> import porty.report
>>>
```

Try running some of your prior scripts as a main program:

```
shell % cd porty-app
shell % python3 -m porty.report portfolio.csv prices.csv txt
      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

shell %
```

Exercise 9.3: Top-level Scripts

Using the `python -m` command is often a bit weird. You may want to write a top level script that simply deals with the oddities of packages. Create a script `print-report.py` that produces the above report:

```
#!/usr/bin/env python3
# print-report.py
import sys
from porty.report import main
main(sys.argv)
```

Put this script in the top-level `porty-app/` directory. Make sure you can run it in that location:

```
shell % cd porty-app
shell % python3 print-report.py portfolio.csv prices.csv txt
      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

shell %
```

Your final code should now be structured something like this:

```
porty-app/
    portfolio.csv
    prices.csv
    print-report.py
    README.txt
    porty/
        __init__.py
        fileparse.py
        follow.py
        pcost.py
        portfolio.py
        report.py
        stock.py
        tableformat.py
        ticker.py
        typedproperty.py
```

Contents | Previous (8.3 Debugging) | Next (9.2 Third Party Packages)

Contents | Previous (9.1 Packages) | Next (9.3 Distribution)

9.2 Third Party Modules

Python has a large library of built-in modules (*batteries included*).

There are even more third party modules. Check them in the Python Package Index or PyPi. Or just do a Google search for a specific topic.

How to handle third-party dependencies is an ever-evolving topic with Python. This section merely covers the basics to help you wrap your brain around how it works.

The Module Search Path

`sys.path` is a directory that contains the list of all directories checked by the `import` statement. Look at it:

```
>>> import sys
>>> sys.path
... look at the result ...
>>>
```

If you import something and it's not located in one of those directories, you will get an `ImportError` exception.

Standard Library Modules

Modules from Python's standard library usually come from a location such as `/usr/local/lib/python3.6'. You can find out for certain by trying a short test:

```
>>> import re
>>> re
<module 're' from '/usr/local/lib/python3.6/re.py'>
>>>
```

Simply looking at a module in the REPL is a good debugging tip to know about. It will show you the location of the file.

Third-party Modules

Third party modules are usually located in a dedicated `site-packages` directory. You'll see it if you perform the same steps as above:

```
>>> import numpy
>>> numpy
<module 'numpy' from '/usr/local/lib/python3.6/site-packages/numpy/__init__.py'>
>>>
```

Again, looking at a module is a good debugging tip if you're trying to figure out why something related to `import` isn't working as expected.

Installing Modules

The most common technique for installing a third-party module is to use `pip`. For example:

```
bash % python3 -m pip install packagename
```

This command will download the package and install it in the `site-packages` directory.

Problems

- You may be using an installation of Python that you don't directly control.
    - A corporate approved installation
    - You're using the Python version that comes with the OS.
- You might not have permission to install global packages in the computer.

- There might be other dependencies.

Virtual Environments

A common solution to package installation issues is to create a so-called "virtual environment" for yourself. Naturally, there is no "one way" to do this--in fact, there are several competing tools and techniques. However, if you are using a standard Python installation, you can try typing this:

```
bash % python -m venv mypython
bash %
```

After a few moments of waiting, you will have a new directory `mypython` that's your own little Python install. Within that directory you'll find a `bin/` directory (Unix) or a `Scripts/` directory (Windows). If you run the `activate` script found there, it will "activate" this version of Python, making it the default `python` command for the shell. For example:

```
bash % source mypython/bin/activate
(mypython) bash %
```

From here, you can now start installing Python packages for yourself. For example:

```
(mypython) bash % python -m pip install pandas
...
```

For the purposes of experimenting and trying out different packages, a virtual environment will usually work fine. If, on the other hand, you're creating an application and it has specific package dependencies, that is a slightly different problem.

Handling Third-Party Dependencies in Your Application

If you have written an application and it has specific third-party dependencies, one challenge concerns the creation and preservation of the environment that includes your code and the dependencies. Sadly, this has been an area of great confusion and frequent change over Python's lifetime. It continues to evolve even now.

Rather than provide information that's bound to be out of date soon, I refer you to the Python Packaging User Guide.

Exercises

Exercise 9.4 : Creating a Virtual Environment

See if you can recreate the steps of making a virtual environment and installing pandas into it as shown above.

Contents | Previous (9.1 Packages) | Next (9.3 Distribution)

Contents | Previous (9.2 Third Party Packages) | Next (The End)

9.3 Distribution

At some point you might want to give your code to someone else, possibly just a co-worker. This section gives the most basic technique of doing that. For more detailed information, you'll need to consult the Python Packaging User Guide.

Creating a setup.py file

Add a `setup.py` file to the top-level of your project directory.

```
# setup.py
import setuptools

setuptools.setup(
    name="porty",
    version="0.0.1",
    author="Your Name",
    author_email="you@example.com",
    description="Practical Python Code",
    packages=setuptools.find_packages(),
)
```

Creating MANIFEST.in

If there are additional files associated with your project, specify them with a `MANIFEST.in` file. For example:

```
# MANIFEST.in
include *.csv
```

Put the `MANIFEST.in` file in the same directory as `setup.py`.

Creating a source distribution

To create a distribution of your code, use the `setup.py` file. For example:

```
bash % python setup.py sdist
```

This will create a `.tar.gz` or `.zip` file in the directory `dist/`. That file is something that you can now give away to others.

Installing your code

Others can install your Python code using `pip` in the same way that they do for other packages. They simply need to supply the file created in the previous step. For example:

```
bash % python -m pip install porty-0.0.1.tar.gz
```

Commentary

The steps above describe the absolute most minimal basics of creating a package of Python code that you can give to another person. In reality, it can be much more complicated depending on third-party dependencies, whether or not your application includes foreign code (i.e., C/C++), and so forth. Covering that is outside the scope of this course. We've only taken a tiny first step.

Exercises

Exercise 9.5: Make a package

Take the `porty-app/` code you created for Exercise 9.3 and see if you can recreate the steps described here. Specifically, add a `setup.py` file and a `MANIFEST.in` file to the top-level directory. Create a source distribution file by running `python setup.py sdist`.

As a final step, see if you can install your package into a Python virtual environment.

Contents | Previous (9.2 Third Party Packages) | Next (The End)

The End!

You've made it to the end of the course. Thanks for your time and your attention. May your future Python hacking be fun and productive!

I'm always happy to get feedback. You can find me at https://dabeaz.com or on Twitter at @dabeaz. - David Beazley.

Contents | Home