Appendix B. Solutions to End-of-Part Exercises

This appendix provides solutions for the book's end-of-part exercises. Code files named by captions or narrative in these solutions are available in the book examples package's *AppendixB* folder, which has one subfolder per part (e.g., *AppendixB/Part1* is the first part's files). See the Preface for more info on the examples package.

Part I, Getting Started

See "Test Your Knowledge: Part I Exercises" in Chapter 3 for the exercises.

1. *Interaction*: Assuming Python is configured properly, the interaction should look something like the following. You can run this any way you like—in IDLE, a console, an app, a notebook's page, and so on:

   ```
   $ python3
   …information lines…
   >>> 'Hello World!'
   'Hello World!'
   >>>                  # Use ctrl+D/ctrl+Z to exit on U
   ```

2. *Programs*: Your code (i.e., module) file should look something like Example B-1:

   Example B-1. Part1/module1.py

   ```
   print('Hello module world!')
   ```

   And here is the sort of interaction you should have; for console launches, be sure to use your platform's version of the "python3" command (e.g., try "py -3" on Windows):

```
$ python3 module1.py
Hello module world!
```

Again, feel free to run this other ways—by clicking or tapping the file's icon, by using IDLE's *Run→Run Module* menu option, by UI options in web notebooks or other IDEs, and so on.

3. *Modules*: The following interaction listing illustrates running a module file by importing it:

```
$ python3
>>> import module1
Hello module world!
>>>
```

Remember that you will need to *reload* the module to run it again without stopping and restarting the interactive interpreter (i.e., REPL). Moving the *.py* file to a different directory and importing it normally fails: Python likely generated a *module1.*.pyc* file in the *__pycache__* subdirectory of the source code file's folder, but it won't use it when you import the module there if the source code (*.py*) file has been moved elsewhere and to a folder not in Python's import search path.

The *.pyc* file is written automatically if Python has access to the source file's directory; it contains the compiled bytecode version of a module. See Chapter 3 for more on modules, Chapter 2 for more on bytecode, and Chapter 22 ahead for more on both. To really use the saved *.pyc* sans *.py*, as of Python 3.2, you must move it up one level and rename it without the "*" part in the middle, or generate it from and alongside the source code file with the Python `compileall` module's "legacy" (`-b`) mode. For example, the following compiles all source code files in the current directory into directly usable bytecode files (you can also list specific files or recurse into subfolders, per Python library docs):

```
$ python3 -m compileall -b -l .
```

4. *Scripts*: Assuming your platform supports the `#!` trick, your solution will look like Example B-2, although your `#!` line may need to list a different path to Python on your machine. This line is significant under the Windows launcher shipped and installed with Python, where it is parsed to

select a version of Python to run the script, despite the Unix path syntax, and subject to a default setting; see Appendix A and Python's docs for more details. This launching scheme is optional and generally less portable than others.

Example B-2. Part1/script1.py

```
#!/usr/local/bin/python3
print('Hello module world!')
```

Running this as a program by console command line:

```
$ chmod +x script1.py                 # See also: #!/usr
$ ./script1.py                        # "./" needed only
Hello module world!

$ python3 script1.py                  # Or run normally
Hello module world!
```

5. *Errors and debugging*: The following interaction demonstrates the sorts of error messages you'll get when you complete this exercise. Really, you're triggering Python exceptions; the default exception-handling behavior terminates the running Python program and prints an error message and stack trace on the screen. The stack trace shows where you were in a program when the exception occurred (if function calls are active when the error happens, the "Traceback" section displays all active call levels). In Chapter 10 and Part VII, you will learn that you can catch exceptions using `try` statements and process them arbitrarily. You'll also learn that Python includes a full-blown source code debugger (module `pdb`) for special error-detection requirements. For now, notice that Python gives meaningful messages when programming errors occur, instead of crashing silently:

```
$ python3
>>> 2 ** 500
3273390607896141870013189696827599152216642046043064
5488327009232590415715088668412756007100921725654588

>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: division by zero

>>> oops
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'oops' is not defined
```

6. *Breaks and cycles*: When you type this code:

```
$ python3
>>> L = [1, 2]
>>> L.append(L)
>>> L
[1, 2, [...]]
```

you create a *cyclic* data structure in Python. In Python releases before 1.5.1, the Python printer wasn't smart enough to detect cycles in objects, and it would print an unending stream of `[1, 2, [1, 2, [1, 2, [1, 2` , and so on until you hit the Ctrl+C break-key combination on your machine (which, technically, raises a keyboard-interrupt exception that prints a default message). Beginning with Python 1.5.1, the printer is clever enough to detect cycles, prints `[[...]]` instead to let you know that it has detected a loop in the object's structure, and avoids getting stuck printing forever.

The reason for the cycle is subtle and requires information you will glean in Part II, so this is something of a preview. But in short, assignments in Python always generate *references* to objects, not copies of them. You can think of objects as chunks of memory and of references as implicitly followed pointers. When you run the first assignment in the preceding code, the name `L` becomes a named reference to a two-item list object—a pointer to a piece of memory. Python lists are really arrays of object references, with an `append` method that changes the array in place by tacking on another object reference at the end. Here, the `append` call adds a reference to the front of `L` at the end of `L` , which leads to the cycle illustrated in Figure B-1: a pointer at the end of the list that points back to the front of the list.

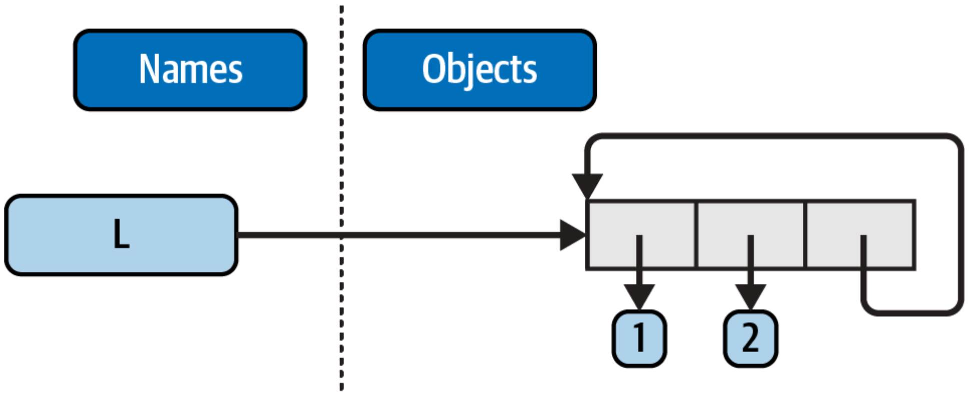

Figure B-1. A cyclic object, created by appending a list to itself

Besides being printed specially, as you'll learn in Chapter 6, cyclic objects must also be handled specially by Python's garbage collector, or their space will remain unreclaimed even when they are no longer in use. Though rare in practice, in some programs that traverse arbitrary objects or structures, you might have to detect such cycles yourself by keeping track of where you've been to avoid looping. Believe it or not, cyclic data structures can sometimes be useful, despite their special-case printing.

Part II, Objects and Operations

See "Test Your Knowledge: Part II Exercises" in Chapter 9 for the exercises.

1. *The basics*: Here are the sorts of results you should get, along with a few comments about their meaning. Again, note that `;` is used in a few of these to squeeze more than one statement onto a single line (the `;` is a statement separator), and commas build up tuples displayed in parentheses. See file *Part2/basics.txt* for copy/paste sans emedia, though typing these manually is a good way to practice syntax:

```
$ python3
# Numbers

>>> 2 ** 16                                   # 2 raised to
65536
>>> 2 / 5, 2 / 5.0                            # Division kee
(0.4, 0.4)

# Strings

>>> 'hack' + 'code'                           # Concatenatio
'hackcode'
```

```
>>> S = 'Python'
>>> 'grok ' + S
'grok Python'
>>> S * 5                              # Repetition
'PythonPythonPythonPythonPython'
>>> S[0], S[:0], S[1:]                 # An empty sli
('P', '', 'ython')                     # Empty of sam

>>> how = 'fun'
>>> 'coding %s is %s!' % (S, how)      # Formatting:
'coding Python is fun!'
>>> 'coding {} is {}!'.format(S, how)
'coding Python is fun!'
>>> f'coding {S} is {how}!'
'coding Python is fun!'

# Tuples

>>> ('x',)[0]                          # Indexing a s
'x'
>>> ('x', 'y')[1]                      # Indexing a t
'y'

# Lists

>>> L = [1, 2, 3] + [4, 5, 6]          # List operati
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5,
>>> ([1, 2, 3] + [4, 5, 6])[2:4]
[3, 4]
>>> [L[2], L[3]]                       # Fetch from o
[3, 4]
>>> L.reverse(); L                     # Method: reve
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                        # Method: sort
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                         # Method: offs
3

# Dictionaries

>>> {'a': 1, 'b': 2}['b']              # Index a dict
2
>>> D = {'x': 1, 'y': 2, 'z': 3}
>>> D['w'] = 0                         # Create a new
>>> D['x'] + D['w']
```

```
1
>>> D[(1, 2, 3)] = 4                  # A tuple usea

>>> D
{'x': 1, 'y': 2, 'z': 3, 'w': 0, (1, 2, 3): 4}

>>> list(D.keys()), list(D.values()), (1, 2, 3) in D
(['x', 'y', 'z', 'w', (1, 2, 3)], [1, 2, 3, 0, 4], T

# Empties

>>> [[]], ["", [], (), {}, None]      # Lots of noth
([[]], ['', [], (), {}, None])
```

2. *Indexing and slicing*: Indexing out of bounds (e.g., `L[4]`) raises an error; Python always checks to make sure that all offsets are within the bounds of a sequence.
   On the other hand, slicing out of bounds (e.g., `L[-1000:100]`) works because Python scales out-of-bounds slices so that they always fit (the limits are set to zero and the sequence length, if required).
   Extracting a sequence in reverse, with the lower bound greater than the higher bound (e.g., `L[3:1]`), doesn't really work. You get back an empty slice (`[]`) because Python scales the slice limits to make sure that the lower bound is always less than or equal to the upper bound (e.g., `L[3:1]` is scaled to `L[3:3]` , the empty insertion point at offset `3`). Python slices are always extracted from left to right, even if you use negative indexes (they are first converted to positive indexes by adding the sequence length). Note that Python's three-limit slices modify this behavior somewhat. For instance, `L[3:1:-1]` does extract from right to left:

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]

>>> L
```

```
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. *Indexing, slicing, and* `del` : Your interaction with the interpreter should look something like the following. Note that assigning an empty list to an offset stores an empty list object there, but assigning an empty list to a slice deletes the slice. Slice assignment expects another sequence, or you'll get a type error; it inserts items *inside* the sequence assigned, not the sequence itself:

```
>>> L = [1, 2, 3, 4]
>>> L[2] = []
>>> L
[1, 2, [], 4]

>>> L[2:3] = []
>>> L
[1, 2, 4]

>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]

>>> L[1:2] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
```

4. *Tuple assignment*: The values of `X` and `Y` are swapped. When tuples appear on the left and right of an assignment symbol (`=`), Python assigns objects on the right to targets on the left according to their positions. This is probably easiest to understand by noting that the targets on the left aren't a real tuple, even though they look like one; they are simply a set of independent assignment targets. The items on the right are a tuple, which gets unpacked during the assignment (this tuple provides the temporary assignment needed to achieve the swap effect):

```
>>> X = 'code'
>>> Y = 'hack'
>>> X, Y = Y, X
>>> X
'hack'
>>> Y
'code'
```

5. *Dictionary keys*: Any *immutable* (technically, “hashable”) object can be used as a dictionary key, including integers, tuples, strings, and so on. This really is a dictionary, even though some of its keys look like integer offsets. Mixed-type keys work fine, too:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Dictionary indexing*: Indexing a nonexistent key (`D['d']`) raises an error; assigning to a nonexistent key (`D['d']='hack'`) creates a new dictionary entry. On the other hand, out-of-bounds indexing for lists raises an error, too, but so do out-of-bounds assignments. Variable names work like dictionary keys; they must have already been assigned when referenced, but they are created when first assigned. In fact, variable names can be processed as dictionary keys if you wish (they’re made visible in the dictionaries of stack frames or module [or other object] namespaces):

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D['a']
1
>>> D['d']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'

>>> D['d'] = 4
>>> D
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

```
>>> L = [0, 1]
>>> L[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> L[2] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

7. *Generic operations*: Question answers (with some error text omitted in listings):
   a. The `+` operator doesn't work on different/mixed types (e.g., string `+` list, list `+` tuple).
   b. `+` doesn't work for dictionaries, as they aren't sequences (though `|` does).
   c. The `append` method works only for lists, not strings, and `keys` works only on dictionaries. `append` assumes its target is mutable, since it's an in-place extension; strings are immutable. Dictionary `keys` is similarly type specific.
   d. Slicing and concatenation always return a new object of the same type as the objects processed:

```
>>> 'x' + 1
TypeError: illegal argument type for built-in ope

>>> {} + {}
TypeError: bad operand type(s) for +

>>> [].append(9)
>>> ''.append('s')
AttributeError: attribute-less object

>>> list({}.keys())
[]
>>> [].keys()
AttributeError: keys

>>> [][:]
[]
>>> ''[:]
''
```

8. *String indexing*: This is a bit of a trick question—because strings are collections of one-character strings, every time you index a string, you get back a string that can be indexed again. `S[0][0][0][0][0]` just keeps indexing the first character over and over. This generally doesn't work for lists (lists can hold arbitrary objects) unless the list contains strings:

```
>>> S = 'hack'
>>> S[0][0][0][0][0]
'h'
>>> L = ['h', 'a']
>>> L[0][0][0]
'h'
```

9. *Immutable types*: Either of the following solutions works. Index assignment doesn't because strings are immutable:

```
>>> S = 'hack'
>>> S = S[0] + 'e' + S[2:]
>>> S
'heck'
>>> S = S[0] + 'i' + S[2] + S[3]
>>> S
'hick'
```

   (See also the `bytearray` string type in Chapter 37—it's a mutable sequence of small integers that is essentially processed the same as a string, especially when its bytes are ASCII character code points.)

10. *Nesting*: Here is a sample (your specs will vary):

```
>>> pat = {'name': ('Pat', 'Q', 'Jones'), 'age': Non
>>> pat['job']
'engineer'
>>> pat['name'][2]
'Jones'
```

11. *Files*: Examples B-3 and B-4 show one way to create and read back a text file in Python using Unicode encoding defaults on the host (which are generally moot for simple ASCII text like this):

Example B-3. Part2/maker.py

```
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')          # Or: open(
file.close()                               # close not
```

Example B-4. Part2/reader.py

```
file = open('myfile.txt')                  # 'r' is de
print(file.read())                         # Or print(
```

When run (here, from a console command line), the file shows up in the directory you're working in because its name has no path prefix. The `ls` here is a Unix command; use `dir` on Windows:

```
$ python3 maker.py
$ python3 reader.py
Hello file world!

$ ls -l myfile.txt
-rw-r--r--  1 me  staff  18 Aug 11 19:34 myfile.txt
```

Part III, Statements and Syntax

See "Test Your Knowledge: Part III Exercises" in Chapter 15 for the exercises.

1. *Coding basic loops*: As you work through this exercise, you'll wind up with code that looks like the following:

```
>>> S = 'hack'
>>> for c in S:
...     print(ord(c))
...
104
97
99
107
```

```
>>> x = 0
>>> for c in S: x += ord(c)            # Or: x = x
...
>>> x
407
>>> chr(x)                             # Extra crea
'Ɨ'

>>> x = []
>>> for c in S: x.append(ord(c))       # Manual lis
...
>>> x
[104, 97, 99, 107]

>>> list(map(ord, S))
[115, 112, 97, 109]
>>> [ord(c) for c in S]                # map and li
[115, 112, 97, 109]
```

2. *Coding basic selections*: Here is the sort of code expected. To handle out-of-range numbers, add an `else` for `if`, a `case _` for `match`, a `get` method call or `in` test for the dictionary, and a `try` handler for the list. For versions of this code that are easier to copy/paste, see file *Part3/selections.txt* in the examples package:

```
>>> month = 3
>>> if month == 1:
...     print('January')
... elif month == 2:
...     print('February')
... elif month == 3:
...     print('March')
...
March

>>> match month:
...     case 1:
...         print('January')
...     case 2:
...         print('February')
...     case 3:
...         print('March')
...
March
```

```
>>> {1: 'January', 2: 'February', 3: 'March'}[month]
'March'
>>> ['January', 'February', 'March'][month - 1]
'March'
```

3. *Backslash characters*: The example prints the bell character (`\a`) 50 times. Assuming your machine can handle it, and when it's run outside of some interfaces like IDLE, you may get a series of beeps (or one sustained tone if your machine is fast enough). Hey—you were warned.
4. *Sorting dictionaries*: Here's one way to work through this exercise (see Chapter 8 or Chapter 14 if this doesn't make sense). You really do have to split off the `keys` and `sort` calls like this because `sort` returns `None` . You can iterate through dictionary keys directly without calling `keys` (e.g., `for key in D:`), but the keys list will not be sorted like it is by this code. The `sorted` built-in is simpler but creates a new list object:

```
>>> D = {'a': 1, 'c': 3, 'e': 5, 'g': 7, 'f': 6, 'd'
>>> D
{'a': 1, 'c': 3, 'e': 5, 'g': 7, 'f': 6, 'd': 4, 'b'

>>> keys = list(D.keys())                    # Keys view
>>> keys.sort()                              # Sort list
>>> for key in keys:                         # Iterate o
...     print(key, '=>', D[key])
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7

>>> D
{'a': 1, 'c': 3, 'e': 5, 'g': 7, 'f': 6, 'd': 4, 'b'
>>>
>>> for key in sorted(D):                    # Simpler a
...     print(key, '=>', D[key])
...
…same output…
```

5. *Program logic alternatives*: Here's some sample code for the solutions, available in the examples package's *Part3/power*.py*. For step `e`, assign the result of `2 ** X` to a variable outside the loops of steps `a` and `b` and use it inside the loop. Your results may vary; this exercise is mostly designed to get you playing with code alternatives, so anything reasonable gets full credit:

```
# a

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('at index', i)
        break
    i += 1
else:
    print(X, 'not found')

# b

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'was found at', L.index(p))
        break
else:
    print(X, 'not found')

# c

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# d
```

```
X = 5
L = []
for i in range(7): L.append(2 ** i)
print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')


# "Deeper thoughts"

X = 5
L = list(map(lambda x: 2 ** x, range(7)))      # Or
print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')
```

Part IV, Functions and Generators

See "Test Your Knowledge: Part IV Exercises" in Chapter 21 for the exercises.

1. *The basics*: There's not much to this one, but notice that using `print` (and hence your function) is technically a *polymorphic* operation, which does the right thing for each type of object:

```
$ python3
>>> def echo(x):
        print(x)

>>> echo('hack')
hack
>>> echo(3.12)
3.12
>>> echo([1, 2, 3])
[1, 2, 3]
```

```
>>> echo({'edition': 6})
{'edition': 6}
```

2. *Arguments*: Example B-5 gives a sample solution. Remember that you have to use `print` to see results in the test calls because a file isn't the same as code typed interactively; Python doesn't normally echo the results of expression statements in files:

   Example B-5. Part4/adder1.py

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

   print(adder(5, 1.0))
   print(adder('hack', 'code'))
   print(adder(['a', 'b'], ['c', 'd']))
   ```

   And the output:

   ```
   $ python3 adder1.py
   6.0
   hackcode
   ['a', 'b', 'c', 'd']
   ```

3. *Arbitrary arguments*: Two alternative `adder` functions are shown in Example B-6. The hard part here is figuring out how to initialize an accumulator to an empty value of whatever type is passed in. The first solution uses manual type testing to look for an integer and an empty slice of the first argument (assumed to be a sequence) if the argument is determined not to be an integer. The second solution uses the first argument to initialize and scan items 2 and beyond, much like one of the `min` function variants shown in Chapter 18.
   The second solution may be better. Both of these assume all arguments are of the same type, and neither works on dictionaries (as we saw in Part II, `+` doesn't work on mixed types or dictionaries). You could add a type test and special code using `for`, `update`, `**`, or `|` to support dictionaries combos, too, but that's extra credit; see solutions 5 and 6 ahead for related notes. And yes, there is a `sum(*iterable*)` built-in in Python that would make this even simpler, but the point here is to write code of your own; you'll have to eventually:

Example B-6. Part4/adder2.py

```
def adder1(*args):
    print('adder1:', end=' ')
    if type(args[0]) == type(0):                    # Inte
         sum = 0                                    # Init
    else:                                           # else
         sum = args[0][:0]                          # Use
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print('adder2:', end=' ')
    sum = args[0]                                   # Init
    for next in args[1:]:
        sum += next                                 # Add
    return sum

for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('hack', 'code', 'well'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))
```

Here's the sort of output you should get:

```
$ python3 adder2.py
adder1: 9
adder1: hackcodewell
adder1: ['a', 'b', 'c', 'd', 'e', 'f']
adder2: 9
adder2: hackcodewell
adder2: ['a', 'b', 'c', 'd', 'e', 'f']
```

4. *Keywords*: Example B-7 gives a solution to the first part of this exercise, along with its output in a console.

Example B-7. Part4/adder3.py

```
def adder(red=1, green=2, blue=3):
    return red + green + blue

print(adder())
print(adder(5))
print(adder(5, 6))
print(adder(5, 6, 7))
print(adder(blue=7, red=6, green=5))
print(adder(blue=1, red=2))

$ python3 adder3.py
6
10
14
18
18
5
```

Example B-8 gives the second part's solution and its output. To iterate over keyword arguments, use the `**args` form in the function header and use a loop (e.g., `for x in args.keys(): use args[x]`), or use `args.values()` to make this the same as summing `*args` positionals in exercise number 3:

Example B-8. Part4/adder4.py

```
def adder1(*args):                  # Sum any number
    tot = args[0]                   # Same as #3, fo
    for arg in args[1:]:
        tot += arg
    return tot

def adder2(**args):                 # Sum any number
    argskeys = list(args.keys())    # list required
    tot = args[argskeys[0]]
    for key in argskeys[1:]:
        tot += args[key]
    return tot

def adder3(**args):                 # Same, but conv
    args = list(args.values())      # List needed to
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):                 # Same, but reus
    return adder1(*args.values())

print(adder1(1, 2, 3),       adder1('aa', 'bb', 'cc'
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb',
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb',
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb',

$ python3 adder4.py
6 aabbcc
…repeated 4 times…
```

5. (5 and 6) *Dictionary tools*: Solutions for exercises 5 and 6 are combined and listed in Example B-9. These are just coding exercises because Python now provides dictionary methods `D.copy()` and `D1.update(D2)` to handle things like copying and adding (merging) dictionaries. In fact, there are *four* ways to merge dictionaries today, as hinted in solution 3: `for` loops like those here, `D1.update(D2)`, `{**D1,**D2}`, and `D1|D2`. See Chapter 8 for more info on and examples of these tools. `X[:]` doesn't work for dictionaries, as they're not sequences (see Chapter 8 for details). Also, remember that if you assign (`e = d`) rather than copying, you generate a reference to a *shared* dictionary object; changing `d` changes `e`, too:

Example B-9. Part4/dicttools.py

```
def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new
```

Here is the expected behavior of this code demoed in a REPL:

```
$ python3
>>> from dicttools import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1: 1}
>>> y = {2: 2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}
```

6. See #5 (where solutions were combined).
7. *More argument-matching examples*: Here is the sort of interaction you should get, along with comments that explain the matching that goes on. It may be easiest to paste the functions into a file and import them all with a `*` for testing in a REPL; they're repeated in Example B-10 for reference (and in the examples package for copying):

   Example B-10. Part4/testfuncs.py

```
def f1(a, b): print(a, b)            # Normal args

def f2(a, *b): print(a, b)           # Positional co

def f3(a, **b): print(a, b)          # Keyword colle

def f4(a, *b, **c): print(a, b, c)   # Mixed modes

def f5(a, b=2, c=3): print(a, b, c)  # Defaults

def f6(a, b=2, *c): print(a, b, c)   # Defaults and
```

   The expected REPL interaction:

```
$ python3
>>> from testfuncs import *
>>> f1(1, 2)                         # Matched by po
1 2
>>> f1(b=2, a=1)                     # Matched by na
1 2

>>> f2(1, 2, 3)                      # Extra positio
1 (2, 3)
```

```
>>> f3(1, x=2, y=3)                 # Extra keyword
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, **dict(x=2, y=3))   # Extras of bot
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                           # Both defaults
1 2 3
>>> f5(1, 4)                        # Only one defa
1 4 3
>>> f5(1, c=4)                      # Middle defaul
1 2 4

>>> f6(1)                           # One argument:
1 2 ()
>>> f6(1, *[3, 4])                  # Extra positio
1 3 (4,)
```

8. *Primes revisited*: Example B-11 is the primes example, wrapped up in a function and a module (file *primes.py*) so it can be run multiple times. An `if` test was added to trap negatives, `0`, and `1`. It's crucial to use `//` floor division instead of the `/` true division we studied in Chapter 5 to avoid fractional remainders (5 / 2 would yield a false factor 2.5, but 5 / 2 truncates down to 2). Change `//` to `/` to see the difference for yourself:

Example B-11. Part4/primes.py

```
def prime(y):
    if y <= 1:
        print(y, 'is nonprime')
    else:
        x = y // 2
        while x > 1:
            if y % x == 0:
                print(y, 'has factor', x)
                break
            x -= 1
        else:
            print(y, 'is prime')

tests = (27, 24, 13, 13.0, 15, 15.0, 3, 2, 1, -3)
for test in tests:
    prime(test)
```

Here is the module in action; the `//` operator also allows it to work for floating-point numbers by truncating to the floor (5.0 // 2 is 2.0, not 2.5):

```
$ python3 primes.py
27 has factor 9
24 has factor 12
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
3 is prime
2 is prime
1 is nonprime
-3 is nonprime
```

This function still isn't very reusable—it could *return* values, instead of printing—but it's enough to run experiments. It's also not a strict mathematical prime (floating-point numbers work, but shouldn't), and it's still perhaps inefficient. Improvements are left as exercises for more mathematically minded readers. (Hint: a `for` loop over `range(x, 1, -1)` may be a bit quicker than the `while`, but the algorithm may be the real bottleneck here.) To time alternatives, use the homegrown `timer` or standard-library `timeit` modules and coding patterns like those used in Chapter 21's benchmarking sections (and solution 10 ahead).

9. *Iterations and comprehensions*: Here is the sort of code you should write; coding alternatives are notoriously subjective, so there's no right or wrong preference (though see the next solution for an objective factor):

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(map(math.sqrt, values))
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

```
>>> list(math.sqrt(x) for x in values)
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. *Timing tools*: The code file in Example B-13 times the three square root options. Each test takes the best of 5 runs; each run takes the total time required to call the test function 1,000 times; and each test function iterates 10,000 times. The last result of each function is printed to verify that all three do the same work.

    This code also uses a preview (really, cheat) to remotely access the *timer2.py* module in Chapter 21's code folder (Example B-12) with an `import` run its own folder, assumed to be the examples' *AppendixB/Part4*. Appending `sys.path` is one way to augment the search path used to find imported modules, along with `PYTHONPATH` environment settings. This avoids a file copy; we'll explore it in this book's next part, so take it on faith for now.

Example B-12. ../../Chapter21/timer2.py

```
…Example 21-7 in Chapter 21…
```

Example B-13. Part4/timesqrt.py

```
import sys                            # Add timer2.py'
sys.path.append('../../Chapter21')    # Assuming runni
import timer2                         # A cheat! - see

reps = 10_000
repslist = list(range(reps))          # Pull out range

from math import sqrt                 # Not math.sqrt:
def mathMod():
    for i in repslist:
        res = sqrt(i)
    return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
    for i in repslist:
        res = i ** .5
    return res

print(sys.version)
for test in (mathMod, powCall, powExpr):
    elapsed, result = timer2.bestoftotal(test, _reps
    print (f'{test.__name__}: {elapsed:.5f} => {resu
```

Following are the test results for CPython 3.12 (the standard) and PyPy 7.3 (which implements Python 3.10) on macOS. In short, the `math` module is quicker than the `**` expression on both Pythons, and `**` is quicker than the `pow` built-in function in CPython but the same in PyPy:

```
$ python3 timesqrt.py
3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [
mathMod: 0.40860 => 99.99499987499375
powCall: 0.68245 => 99.99499987499375
powExpr: 0.57762 => 99.99499987499375

$ pypy3 timesqrt.py
3.10.14 (75b3de9d9035, Apr 21 2024, 10:56:19)
[PyPy 7.3.16 with GCC Apple LLVM 15.0.0 (clang-1500.
mathMod: 0.05246 => 99.99499987499375
powCall: 0.33288 => 99.99499987499375
powExpr: 0.33244 => 99.99499987499375
```

PyPy is also some 8X to 2X faster than CPython on floating-point math and iterations here, but CPython may sprout a JIT, which evens the gap (see Chapter 2). The results for CPython jive with the prior edition’s tests for CPython 3.3 that follow, which used different repeat counts and hosts but were relatively similar. As always, you should try this with your code and on your own machine and version of Python for more definitive results:

```
c:\code> py -3 timesqrt.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [
mathMod: 2.04481 => 99.99499987499375
powCall: 3.40973 => 99.99499987499375
powExpr: 2.56458 => 99.99499987499375
```

To time the relative speeds of *dictionary comprehensions* and equivalent `for` loops interactively, you can run a session like the following. At least on this test in CPython 3.12, the two are roughly the same in speed, with a slight advantage to comprehensions—though the difference isn't exactly earth-shattering. As verification, these results relatively match those we obtained from a `pybench` test in Example 21-10 (sans the slower `dict` call). Do similar to vet the speed of comprehensions with `if` and `for`. And again, rather than taking any of these results as gospel, you should investigate further on your own with your computer and your Python:

```
$ python3
>>> def dictcomp(I):
        return {i: i for i in range(I)}

>>> def dictloop(I):
        new = {}
        for i in range(I): new[i] = i
        return new

>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8:
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8:

>>> import sys; sys.path.append('../../Chapter21')
>>> from timer2 import bestoftotal

>>> bestoftotal(dictcomp, 10_000, _reps1=5, _reps=50
0.17137739405734465
>>> bestoftotal(dictloop, 10_000, _reps1=5, _reps=50
0.18112968490459025

>>> len(bestoftotal(dictcomp, 10_000, _reps1=5, _rep
10000
>>> len(bestoftotal(dictloop, 10_000, _reps1=5, _rep
10000
```

11. *Recursive functions*: One way to code this function follows (typed in a REPL here, but also coded in file *Part4/countdown.py* of the examples package). A simple `range`, comprehension, or `map` will do the job here as well, of course, but recursion is useful enough to warrant the experimentation here:

```
>>> def countdown(N):
        if N == 0:
            print('stop')
        else:
            print(N, end=' ')
            countdown(N - 1)

>>> countdown(5)
5 4 3 2 1 stop
>>> countdown(20)
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 s

# Nonrecursive options

>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
>>> t = [print(i, end=' ') for i in range(5, 0, -1)]
5 4 3 2 1
>>> t = list(map(lambda x: print(x, end=' '), range(
5 4 3 2 1
```

A *generator*-based solution isn't required for this exercise, but one is listed next; all the other techniques seem much simpler in this case—a good example of contexts where generators should probably be avoided. Remember that generators produce no results until iterated, so we need a `for` loop or `yield from` here (yielding `countdown2(N-1)` directly simply returns a generator, not its products):

```
>>> def countdown2(N):                           #
        if N == 0:
            yield 'stop'
        else:
            yield N
            for x in countdown2(N - 1): yield x   #

>>> list(countdown2(5))
[5, 4, 3, 2, 1, 'stop']

# Nonrecursive options

>>> def countdown3():                         # Genera
        yield from range(5, 0, -1)            # Or: fo
```

```
>>> list(countdown3())
[5, 4, 3, 2, 1]
>>> list(x for x in range(5, 0, -1))          # Equiva
[5, 4, 3, 2, 1]
```

12. *Computing factorials*: Example B-14 shows one way to code this exercise, using Python's standard-library `timeit` module of Chapter 21. Naturally, there are many possible variations on its code; its ranges, for instance, could run from `2..N+1` to skip an iteration, and `fact2` could use `reduce(operator.mul, range(N, 1, -1))` to avoid a `lambda`. Improve freely.

    Example B-14. Part4/factorials.py

```
from functools import reduce
from timeit import repeat
import math

def fact0(N):
    if N == 1:
        return N
    else:
        return N * fact0(N - 1)

def fact1(N):
    return N if N == 1 else N * fact1(N - 1)

def fact2(N):
    return reduce(lambda x, y: x * y, range(1, N + 1

def fact3(N):
    res = 1
    for i in range(1, N + 1): res *= i
    return res

def fact4(N):
    return math.factorial(N)

# Tests
print(fact0(6), fact1(6), fact2(6), fact3(6), fact4(
print(fact0(500) == fact1(500) == fact2(500) == fact

for test in (fact0, fact1, fact2, fact3, fact4):
    print(test.__name__, min(repeat(stmt=lambda: tes
```

This code uses Python's `timeit` module to benchmark alternatives. Its results for CPython 3.12 on macOS:

```
$ python3 factorials.py
720 720 720 720 720
True
fact0 0.08720566902775317
fact1 0.08635473699541762
fact2 0.06704489700496197
fact3 0.05152398400241509
fact4 0.00873392098583281
```

Conclusions: recursion is slowest on this Python and machine and fails once `N` reaches the maximum stack-size setting in `sys`. Per Chapter 19, this limit can be increased, but simple loops or the standard-library tool seem the best route here in any event, and the built-in wins soundly. This general finding holds true often. For instance, `''.join(reversed(S))` may be the preferred way to reverse a string, even though recursive solutions are possible. Time the code in Example B-15 to see for yourself:

Example B-15. Part4/reverses.py

```
def rev1(S):
    if len(S) == 1:
        return S
    else:
        return S[-1] + rev1(S[:-1])        # Recursi

def rev2(S):
    return ''.join(reversed(S))            # Nonrecu

def rev3(S):
    return S[::-1]                         # Sequenc
```

Part V, Modules and Packages

See "Test Your Knowledge: Part V Exercises" in Chapter 25 for the exercises.

1. *Import basics*: When you're done, your file and REPL interaction with it should look similar to Example B-16. Remember that Python can read a whole file into a list of line strings, and the `len` built-in returns the lengths of strings and lists:

   Example B-16. Part5/mymod.py (initial code, mymod_start.py)

   ```
   def countLines(name):
       file = open(name)
       return len(file.readlines())

   def countChars(name):
       return len(open(name).read())

   def test(name):                                  # C
       return countLines(name), countChars(name)    # C
   ```

   ```
   $ python3
   >>> import mymod
   >>> mymod.test('mymod.py')
   (10, 281)
   ```

   Your counts may vary for comments, an extra line at the end, and so on, and you don't need to set `PYTHONPATH` if the module is in the automatically searched current working directory. Note that these functions load the entire file in memory all at once, so they won't work for pathologically large files that are too big for your device's memory. To be more robust, you could read line by line with iterators instead and count as you go (see *Part5/mymod_lines.py* in the examples package):

   ```
   def countLines(name):
       tot = 0
       for line in open(name): tot += 1
       return tot

   def countChars(name):
       tot = 0
       for line in open(name): tot += len(line)
       return tot
   ```

   A generator expression can have the same effect (though the excessive magic may cost you some points):

```
def countLines(name): return sum(+1 for line in open
def countChars(name): return sum(len(line) for line
```

On Unix, you can verify your output with a `wc` command; on Windows, right-click on your file to view its properties. Note that your script may report fewer characters than Windows does—for portability, Python converts Windows `\r\n` line-end markers to `\n`, thereby dropping one byte (character) per line. To match byte counts with Windows exactly, you must open in binary mode (`'rb'`) or add the number of bytes corresponding to the number of lines. See Chapters 9 and 37 for more on end-of-line translations in text files.

The "ambitious" part of this exercise (passing in a file object so you only open the file once) will require you to use the `seek` method of the built-in file object. It works like C's `fseek` call (and may call it behind the scenes): `seek` resets the current position in the file to a passed-in offset. After a `seek`, future input/output operations are relative to the new position. To rewind to the start of a file without closing and reopening it, call `file.seek(0)`; the file `read` methods all pick up at the current position in the file, so you need to rewind to reread. Example B-17 shows what this tweak would look like, along with its output in a REPL:

Example B-17. Part5/mymod2.py

```
def countLines(file):
    file.seek(0)                                # R
    return len(file.readlines())

def countChars(file):
    file.seek(0)                                # D
    return len(file.read())

def test(name):
    file = open(name)                           # P
    return countLines(file), countChars(file)   # C

$ python3
>>> import mymod2
>>> mymod2.test('mymod2.py')
(12, 414)
```

2. `from` / `from *` : Here's the `from *` part; replace `*` with `countChars` to do the rest:

```
$ python3
>>> from mymod import *
>>> countChars('mymod.py')
281
```

3. `__main__` : If you code it properly, this file works in either mode—program run or module import, as Example B-18 and the REPL session following it demo:

Example B-18. Part5/mymod.py (edited)

```
def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):                                    # C
    return countLines(name), countChars(name)      # C

if __name__ == '__main__':                         # A
    print(test('mymod.py'))                        # W
```

```
$ python3 mymod.py
(13, 434)
```

This is where you would probably begin to consider using command-line arguments or user input to provide the filename to be counted instead of hardcoding it in the script. Examples B-19 and B-20 show the required mods (see Chapters 21 and 25 for more on `sys.argv`, and Chapter 10 for more on `input`):

Example B-19. Part5/mymod_argv.py (changed parts)

```
…
if __name__ == '__main__':
    import sys                                     # C
    print(test(sys.argv[1]))
```

```
$ python3 mymod_argv.py mymod.py
(13, 434)
```

Example B-20. Part5/mymod_input.py (changed parts)

```
…
if __name__ == '__main__':
    print(test(input('Enter file name: ')))       # C

$ python3 mymod_input.py
Enter file name: mymod.py
(13, 434)
```

4. *Nested imports*: It's not much, but Example B-21 gives one solution and its results (the point here is to experiment with importing one module from another in a variety of ways):

   Example B-21. Part5/myclient.py

   ```
   from mymod import countLines, countChars
   print(countLines('mymod.py'), countChars('mymod.py')

   $ python3 myclient.py
   13 434
   ```

   As for the rest of this question, `mymod`'s functions are accessible (that is, importable) from the top level of `myclient`, since `from` simply assigns to names in the importer (it works as if `mymod`'s `def`s appeared in `myclient`). For example, another file can say:

   ```
   import myclient
   myclient.countLines(…)

   from myclient import countChars
   countChars(…)
   ```

   If `myclient` used `import` instead of `from`, you'd need to use a path to get to the functions in `mymod` through `myclient`:

```
import myclient
myclient.mymod.countLines(…)

from myclient import mymod
mymod.countChars(…)
```

In general, you can define *collector* modules that import all the names from other modules so they're available in a single convenience module. The following hypothetical code, for example, creates three different copies of the name `somename` — `mod1.somename`, `collector.somename`, and `__main__.somename`; all three share the same integer object initially, and only the name `somename` exists at the interactive prompt as is:

```
# File mod1.py (hypothetical)
somename = 99

# File collector.py (hypothetical)
from mod1 import *                                   # C
from mod2 import *                                   # "
from mod3 import *

>>> from collector import somename
```

5. *Package imports*: For this, copy the *mymod.py* solution file listed for exercise 3 (Example B-18) into a directory package. The following commands run in a Unix console set up the directory and an optional *__init__.py* file; you'll need to interpolate for other platforms and tools (e.g., use `copy` and `notepad` on Windows instead of `cp` and `vi`). This works in any directory, and you can do some of this from a file-explorer GUI, too.
   When finished, you'll have a *mypkg* subdirectory that contains the files *__init__.py* and *mymod.py*. Technically, *mypkg* is located in the "home" directory component of the module search path. Notice how a `print` statement coded in the directory's initialization file fires only the first time it is imported, not the second. Raw strings (`r'…'`) can also avoid `\` escape issues in the file paths if you're working on Windows, but `/` works there too:

```
$ mkdir mypkg                           # Windows: same
$ cp mymod.py mypkg/mymod.py            # Windows: copy
$ vi mypkg/__init__.py                  # Windows: notep
…code a print statement…

$ python3                                              #
>>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines('mypkg/mymod.py')           #
13
>>> from mypkg.mymod import countChars
>>> countChars('mypkg/mymod.py')                       #
434
```

If you copy the module to *__main__.py*, the copy will run if you run the directory as a whole (though there may be no reason to do so in practice, as the original module can be run directly too):

```
$ cp mypkg/mymod.py mypkg/__main__.py                  #
$ python3 mypkg
(13, 434)
$ python3 mypkg/mymod.py
(13, 434)
```

6. *Reloads*: This exercise just asks you to experiment with changing the *changer.py* example in the book's Example 23-10, so there's nothing to show here.
7. *Circular imports*: The short story is that importing `recur2` first works because the recursive import then happens at the import in `recur1`, not at a `from` in `recur2`.

   The long story goes like this: importing `recur2` first works because the recursive import from `recur1` to `recur2` fetches `recur2` as a whole instead of getting specific names. `recur2` is incomplete when it's imported from `recur1`, but because it uses `import` instead of `from`, you're safe: Python finds and returns the already created `recur2` module object and continues to run the rest of `recur1` without a glitch. When the `recur2` import resumes, the second `from` finds the name `Y` in `recur1` (it's been run completely), so no error is reported.

   Running a file as a *script* is not the same as importing it as a module; these cases are the same as running the first `import` or `from` in the script interactively. For instance, running `recur1` as a script works because it is the same as importing `recur2` interactively, as `recur2` is the first module imported in `recur1`. Running `recur2` as a script fails for the same reason—it's the same as running its first import interactively.

Part VI, Classes and OOP

See "Test Your Knowledge: Part VI Exercises" in Chapter 32 for the exercises.

1. *Inheritance*: Example B-22 lists a solution for this exercise, along with some interactive tests. The `__add__` overload has to appear only once, in the superclass, as it invokes type-specific `add` methods in subclasses:

   Example B-22. Part6/adder.py

```
class Adder:
    def add(self, x, y):
        print('not implemented!')
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                             # C
        return self.add(self.data, other)                 # C

class ListAdder(Adder):
    def add(self, x, y):
        return x + y

class DictAdder(Adder):
    def add(self, x, y):
        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new
```

```
$ python3
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
```

```
>>> x = DictAdder()
>>> x.add({1: 1}, {2: 2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
TypeError: can only concatenate list (not "ListAdder
```

Notice in the last test that you get an error for expressions where a class instance appears on the right of a `+`; if you want to fix this, use `__radd__` methods, as described in Chapter 30.

If you are saving a value in the instance anyhow, you might as well rewrite the `add` method to take just one argument, in the spirit of other examples in this part of the book. Example B-23 sketches this mutation:

Example B-23. Part6/adder2.py

```
class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                   # Pass a
        return self.add(other)                  # The lef
    def add(self, y):
        print('not implemented!')

class ListAdder(Adder):
    def add(self, y):
        return self.data + y

class DictAdder(Adder):
    def add(self, y):
        d = self.data.copy()                    # Change
        d.update(y)                             # Or "che
        return d

x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)                                        # Prints
```

```
z = DictAdder(dict(name='x')) + {'a': 1}
print(z)                                        # Prints
```

Because values are attached to objects rather than passed around, this version is arguably more object-oriented. And, once you've gotten to this point, you'll probably find that you can get rid of `add` altogether and simply define type-specific `__add__` methods in the two subclasses.

2. *Operator overloading*: The solution code and its REPL results in Example B-24 demo a handful of operator-overloading methods we explored in Chapter 30. Copying the initial value in the constructor is important because it may be mutable; you don't want to change or have a reference to an object that's possibly shared somewhere outside the class. The `__getattr__` method routes calls to the wrapped list. For tips on a possibly easier way to code this, See "Extending Types by Subclassing" in Chapter 32:

Example B-24. Part6/mylist.py

```
class MyList:
    def __init__(self, start):
        #self.wrapped = start[:]                     #
        self.wrapped = list(start)                   #
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
    def __getitem__(self, offset):                   #
        return self.wrapped[offset]                  #
    def __len__(self):
        return len(self.wrapped)                     #
    def append(self, node):
        self.wrapped.append(node)
    def __getattr__(self, name):                     #
        return getattr(self.wrapped, name)
    def __repr__(self):                              #
        return repr(self.wrapped)

if __name__ == '__main__':
    x = MyList('hack')
    print(x)
    print(x[2])
    print(x[1:])
```

```
        print(x + ['code'])
        print(x * 3)
        x.append('1'); x.extend(['z'])
        x.sort()
        print(' '.join(c for c in x))

    $ python3 mylist.py
    ['h', 'a', 'c', 'k']
    c
    ['a', 'c', 'k']
    ['h', 'a', 'c', 'k', 'code']
    ['h', 'a', 'c', 'k', 'h', 'a', 'c', 'k', 'h', 'a', '
    1 a c h k z
```

Note that it's also important to copy the start value by calling `list` instead of slicing here, because otherwise the result may not be a true *list*, and so will not respond to expected list methods, such as `append` (e.g., slicing a string returns another string, not a list). You would be able to copy a `MyList` start value by slicing because its class overloads the slicing operation and provides the expected list interface; however, you need to avoid slice-based copying for objects such as strings.

3. *Subclassing*: One solution appears in Example B-25; your solution will be similar. You can also use `super` here instead of explicit superclass names for methods and attributes, as partly noted in the code's comments:

Example B-25. Part6/mysub.py

```
from mylist import MyList

class MyListSub(MyList):
    calls = 0                                           #
    def __init__(self, start):
        self.adds = 0                                   #
        MyList.__init__(self, start)                    #

    def __add__(self, other):
        print('add: ' + str(other))
        MyListSub.calls += 1                            #
        self.adds += 1                                  #
        return MyList.__add__(self, other)              #

    def stats(self):
        return self.calls, self.adds                    #
```

```
if __name__ == '__main__':
    x = MyListSub('read')
    y = MyListSub('code')
    print(x[2])
    print(x[1:])
    print(x + ['lp6e'])
    print(x + ['book'])
    print(y + ['py312'])
    print(x.stats())

$ python3 mysub.py
a
['e', 'a', 'd']
add: ['lp6e']
['r', 'e', 'a', 'd', 'lp6e']
add: ['book']
['r', 'e', 'a', 'd', 'book']
add: ['py312']
['c', 'o', 'd', 'e', 'py312']
(3, 2)
```

4. *Attribute methods*: The following works through this exercise. As noted in Chapter 28 and elsewhere, `__getattr__` is *not* called for built-in operations in Python 3.X, so the expressions aren't intercepted at all here; a class like this must somehow redefine `__X__` operator-overloading methods explicitly. You can find more on this limitation in Chapters 28, 31, 32, and 38, as well as *workarounds* for it in Chapter 39 and its *inheritance* special case in Chapter 40. Its impacts are potentially broad but can be addressed with code:

```
$ python3
>>> class Attrs:
        def __getattr__(self, name):
            print('get:', name)
        def __setattr__(self, name, value):
            print('set:', name, value)

>>> x = Attrs()
>>> x.append
get append
>>> x.lang = 'py312'
set: lang py312
>>> x + 2
TypeError: unsupported operand type(s) for +: 'Attrs
```

```
>>> x[1]
TypeError: 'Attrs' object is not subscriptable
>>> x[1:5]
TypeError: 'Attrs' object is not subscriptable
```

5. *Set objects*: Here's the sort of interaction you should get. To make the import of *Chapter32/setwrapper.py* work, either run this in the folder where this file resides, copy this file to your working directory, or add this file's folder to your import search path per Part V. Comments explain which methods are called. Also, bear in mind that sets are a built-in type in Python, so this is mostly just a coding exercise (see Chapter 5 for more on sets):

```
$ python3
>>> from setwrapper import Set      # Run there, copy
>>> x = Set([1, 2, 3, 4])           # Runs __init__
>>> y = Set([3, 4, 5])

>>> x & y                           # __and__, inters
Set:[3, 4]
>>> x | y                           # __or__, union,
Set:[1, 2, 3, 4, 5]

>>> z = Set('hello')                # __init__ remove
>>> z[0], z[-1], z[2:]              # __getitem__
('h', 'o', ['l', 'o'])

>>> for c in z: print(c, end=' ')  # __iter__ (else
...
h e l o
>>> ''.join(c.upper() for c in z)  # __iter__ (else
'HELO'
>>> len(z), z                       # __len__, __repr
(4, Set:['h', 'e', 'l', 'o'])

>>> z & 'mello', z | 'mello'
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])
```

A solution to the multiple-operand extension subclass looks like the class in Example B-26. It needs to replace only two methods in the original set. The class's documentation string explains how it works:

Example B-26. Part6/multiset.py

```
from setwrapper import Set

class MultiSet(Set):
    """
    Inherits all Set names, but extends intersect an
    multiple operands.  Note that "self" is still th
    (stored in the *args argument now).  Also note t
    & and | operators call the new methods here with
    processing more than 2 requires a method call, n
    intersect doesn't remove duplicates here: the Se
    """
    def intersect(self, *others):
        res = []
        for x in self:                              # Sca
            for other in others:                    # For
                if x not in other: break            # Ite
            else:                                   # No:
                res.append(x)                       # Yes
        return Set(res)

    def union(*args):                               # sel
        res = []
        for seq in args:                            # For
            for x in seq:                           # For
                if not x in res:
                    res.append(x)                   # Ada
        return Set(res)
```

Your interaction with this extension will look something like the following. Note that you can intersect by using `&` or calling `intersect`, but you must call `intersect` for three or more operands; `&` is a binary (two-sided) operator. Also, note that we could have called `MultiSet` simply `Set` to make this change more transparent if we used `setwrapper.Set` to refer to the original within `multiset` (the `as` clause in an import could rename the class too if desired):

```
>>> from multiset import *
>>> x = MultiSet([1, 2, 3, 4])
>>> y = MultiSet([3, 4, 5])
>>> z = MultiSet([0, 1, 2])
>>> x & y, x | y                                  # Two
(Set:[3, 4], Set:[1, 2, 3, 4, 5])

>>> x.intersect(y, z)                             # Thr
Set:[]
>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]
>>> x.intersect([1,2,3], [2,3,4], [1,2,3])        # Fou
Set:[2, 3]
>>> x.union(range(10))                            # Non
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]

>>> w = MultiSet('soap')                          # Str
>>> w
Set(['s', 'o', 'a', 'p'])
>>> ''.join(w | 'super')
'soapuer''
>>> (w | 'super') & MultiSet('slots')
Set(['s', 'o'])
```

6. *Class tree links*: Example B-27 lists one way to change the lister class in Example 31-10, along with a rerun of the associated tester to show its augmented format. For full credit, do the same for the `dir` -based version, and also do this when formatting class objects in the tree-climber variant. To import *testmixin.py* as a test, either copy it over from the Chapter 31 examples folder or add that folder to `sys.path` as we did earlier in Part IV's solutions. It was copied here for variety:

Example B-27. Part6/listinstance-mod.py

```
class ListInstance:
    def __attrnames(self):
        …unchanged…

    def __str__(self):
        return (f'<Instance of {self.__class__.__nam
                f'({self.__supers()}), '
                f'address {id(self):#x}:'
                f'{self.__attrnames()}>')

    def __supers(self):
        names = []
        for super in self.__class__.__bases__:
            names.append(super.__name__)
        return ', '.join(names)
        # Or: ', '.join(super.__name__ for super in

if __name__ == '__main__':
    import testmixin                    # Assume tes
    testmixin.tester(ListInstance)      # Test class

$ python3 listinstance-mod.py
<Instance of Sub(Super, ListInstance), address 0x10e
    data1='code'
    data2='Python'
    data3=3.12
>
```

7. *Composition*: A full-points solution is coded in Example B-28, with comments from the description mixed in with the code. This is one case where it's probably easier to express a problem in code than it is in narrative:

Example B-28. Part6/lunch.py

```
class Lunch:
    def __init__(self):                         # M
        self.cust = Customer()
        self.empl = Employee()
    def order(self, foodName):                  # S
        self.cust.placeOrder(foodName, self.empl)
    def result(self):                           # A
        self.cust.printFood()

class Customer:
    def __init__(self):                         # I
        self.food = None
    def placeOrder(self, foodName, employee):   # P
        self.food = employee.takeOrder(foodName)
    def printFood(self):                        # P
        print(self.food.name)

class Employee:
    def takeOrder(self, foodName):              # R
        return Food(foodName)

class Food:
    def __init__(self, name):                   # S
        self.name = name
```

```
if __name__ == '__main__':
    x = Lunch()                                      # S
    x.order('burritos')                              # I
    x.result()
    x.order('pizza')
    x.result()
```

When run, customers place orders and get food from employees. This could be much more involved, but it suffices to demo the routing of messages between objects that's typical in OOP code:

```
$ python3 lunch.py
burritos
pizza
```

8. *Zoo animal hierarchy*: Example B-29 shows one way to code the taxonomy in Python; it's artificial, but the general coding pattern applies to many real structures, from GUIs to employee databases to spacecraft. Notice that the `self.speak` call in `Animal` triggers an independent inheritance search, which generally finds `speak` in a subclass. Test this interactively by calling the `reply` method for instances per the exercise description. Try extending this hierarchy with new classes and making instances of various classes in the tree:

   Example B-29. Part6/zoo.py

```
class Animal:
    def reply(self):   self.speak()                  # B
    def speak(self):   print('blah')                 # C

class Mammal(Animal):
    def speak(self):   print('huh?')

class Cat(Mammal):
    def speak(self):   print('meow')

class Dog(Mammal):
    def speak(self):   print('bark')

class Primate(Mammal):
    def speak(self):   print('Hello world!')
```

```
class Hacker(Primate): pass                         # I
```

Part VII, Exceptions

See “Test Your Knowledge: Part VII Exercises” in Chapter 36 for the exercises.

1. `try` / `except` : One possible coding of the `oops` function is listed in Example B-30. As for the noncoding questions, changing `oops` to raise a `KeyError` instead of an `IndexError` means that the `try` handler won’t catch the exception—it “percolates” to the top level and triggers Python’s default error message. The names `KeyError` and `IndexError` come from the outermost built-in names scope (the *B* in “LEGB”). Import `builtins` and pass it as an argument to the `dir` function to see this for yourself, per Chapter 17.

   Example B-30. Part7/oops.py

   ```
   def oops():
       raise IndexError()

   def doomed():
       try:
           oops()
       except IndexError:
           print('caught an index error!')
       else:
           print('no error caught...')

   if __name__ == '__main__': doomed()

   $ python3 oops.py
   caught an index error!
   ```

2. *Exception objects and lists*: Example B-31 is one way to extend this module for an exception of its own:

Example B-31. Part7/oops2.py

```
class MyError(Exception): pass

def oops():
    raise MyError('Hack!')

def doomed():
    try:
        oops()
    except IndexError:
        print('caught an index error!')
    except MyError as exc:
        print('caught error:', MyError, exc)
    else:
        print('no error caught...')

if __name__ == '__main__':
    doomed()

$ python3 oops2.py
caught error: <class '__main__.MyError'> Hack!
```

Like all class exceptions, the raised instance is accessible via the `as` variable `data`; the error message shows both the class's (`<...>`) and its instance's (`Hack!`) displays. The instance must be inheriting both an `__init__` and a `__repr__` or `__str__` from Python's `Exception` class, or it would print much as the class does. See Chapter 35 for details on how these defaults work in built-in exception classes.

3. *Error handling*: Example B-32 is one way to solve this exercise. It codes tests in a file rather than interactively, but the results are similar enough for full credit. Notice that the empty `except` and `sys.exc_info` approach used here will catch exit-related exceptions that listing `Exception` with an `as` variable won't; that's probably not ideal in most applications code but might be useful in a tool like this designed to work as a sort of exceptions firewall.

Example B-32. Part7/exctools.py

```
import sys, traceback

def safe(callee, *pargs, **kargs):
    try:
        callee(*pargs, **kargs)           # Catch e
    except:                               # Or "exc
        traceback.print_exc()
        print(f'Got {sys.exc_info()[0]} {sys.exc_inf

if __name__ == '__main__':
    import oops2
    safe(oops2.oops)
```

```
$ python3 exctools.py
Traceback (most recent call last):
  File "/…/LP6E/AppendixB/Part7/exctools.py", line 5
    callee(*pargs, **kargs)           # Catch every
    ^^^^^^^^^^^^^^^^^^^^^^^
  File "/…/LP6E/AppendixB/Part7/oops2.py", line 4, i
    raise MyError('Hack!')
oops2.MyError: Hack!
Got <class 'oops2.MyError'> Hack!
```

Bonus points: the sort of code in Example B-33 could turn this into a *function decorator* that could wrap and catch exceptions raised by any function, using techniques introduced briefly in Chapter 32, but covered more fully in Chapter 39—it augments a function, rather than expecting it to be passed in explicitly, and produces similar output when run (there's an extra call level, and filenames differ):

Example B-33. Part7/exctools_deco.py

```
import sys, traceback

def safe(callee):
    def callproxy(*pargs, **kargs):
        try:
            return callee(*pargs, **kargs)
        except Exception as E:
            traceback.print_exc()
            print(f'Got {E.__class__} {E}')
    return callproxy

if __name__ == '__main__':
    import oops2

    @safe
    def test():                    # test = safe(test)
```

```
            oops2.oops()

        test()
```

4. *Self-study examples*: In closing, Examples B-34 through B-43 are 10 examples for you to study on your own. Their code and supporting files are in the *Self-Study-Demos* subfolder of the examples package's *AppendixB/Part7* folder. These require no extra installs as they use standard-library tools, though `tkinter` is sketchy on phones (see Appendix A). For more examples, see follow-up books and resources for the application domains you'll be exploring next:

Example B-34. Part7/Self-Study-Demos/largest-dir.py

```
# Find the largest Python source file in a single di

import os, glob
dirname = '/Users/me/Downloads'    # Edit me to use

allsizes = []
allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])
```

Example B-35. Part7/Self-Study-Demos/largest-tree.py

```
# Find the largest Python source file in an entire d

import sys, os, pprint
if sys.platform[:3] == 'win':
    dirname = r'C:\Users\me\Downloads'    # Edit me
else:
    dirname = '/Users/me/Downloads'

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirnam
    for filename in filesHere:
        if filename.endswith('.py'):
            fullname = os.path.join(thisDir, filenam
                fullsize = os.path.getsize(fullname)
                allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])
```

Example B-36. Part7/Self-Study-Demos/largest-import.py

```
# Find the largest Python source file on the module

import sys, os, pprint
visited  = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(sr
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath  = os.path.join(thisDir, file
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('skipping', pypath)
                allsizes.append((pysize, pypath))

allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])
```

Example B-37. Part7/Self-Study-Demos/summer1.py

```
# Sum columns in a text file separated by commas

filename = 'data.txt'    # Edit me for others
sums = {}

for line in open(filename):
    cols = line.split(',')
```

```
        nums = [int(col) for col in cols]
        for (ix, num) in enumerate(nums):
            sums[ix] = sums.get(ix, 0) + num

for key in sorted(sums):
    print(key, '=', sums[key])
```

Example B-38. Part7/Self-Study-Demos/summer2.py

```
# Similar to summer1, but using lists instead of dic

import sys
filename = sys.argv[1]            # "python3 summer2
numcols  = int(sys.argv[2])
totals   = [0] * numcols

for line in open(filename):
    cols = line.split(',')
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums

print(totals)
```

Example B-39. Part7/Self-Study-Demos/regrtest.py

```
# Simple test for regressions in the output of a set

import os
testscripts = [dict(script='test1.py', args=''),
               dict(script='test2.py', args='-opt')]

for testcase in testscripts:
    commandline = '%(script)s %(args)s' % testcase
    output = os.popen(commandline).read()
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Created:', result)
    else:
        priorresult = open(result).read()
        if output != priorresult:
            print('FAILED:', testcase['script'])
            print(output)
        else:
            print('Passed:', testcase['script'])
```

Example B-40. Part7/Self-Study-Demos/gui1.py

```
"""
Build a GUI with tkinter having buttons that change
Caution: this GUI may grow until you close its windo
"""

from tkinter import *
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange'

def reply(text):
    print(text)
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color)
    L.config(fg=color)

def cycle():
    L.config(fg=random.choice(colors))
    win.after(250, cycle)

def grow():
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)

win = Tk()
L = Label(win, text='Hack',
          font=('arial', fontsize, 'italic'), fg='ye
          relief=RAISED)
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='popup', command=(lambda: reply('ne
Button(win, text='cycle', command=cycle).pack(side=B
Button(win, text='grow', command=grow).pack(side=BOT
win.mainloop()
```

Example B-41. Part7/Self-Study-Demos/gui2.py

```
"""
Similar to gui1, but use classes so each window has
Caution: this GUI may grow until you press Stop or k
"""

from tkinter import *
import random

class MyGui:
    """
    A GUI with buttons that change color and make th
    """
    colors = ['blue', 'green', 'orange', 'red', 'bro

    def __init__(self, parent, title='popup'):
        parent.title(title)
        self.growing = False
        self.fontsize = 10
        self.lab = Label(parent, text='Hack2', fg='w
        self.lab.pack(expand=YES, fill=BOTH)
        Button(parent, text='Hack', command=self.rep
        Button(parent, text='Grow', command=self.gro
        Button(parent, text='Stop', command=self.sto

    def reply(self):
        "change the button's color at random on Hack
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color,
                font=('courier', self.fontsize, 'bol

    def grow(self):
        "start making the label grow on Grow presses
        self.growing = True
        self.grower()

    def grower(self):
        "multiple presses schedule multiple growers"
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fo
            self.lab.after(500, self.grower)
    def stop(self):
        "stop all button grower loops on Stop presse
        self.growing = False

class MySubGui(MyGui):
    colors = ['black', 'purple']           # Customi

MyGui(Tk(), 'main')
MyGui(Toplevel())
MySubGui(Toplevel())
mainloop()
```

Example B-42. Part7/Self-Study-Demos/popmail.py

```
"""
POP email inbox scanning and deletion utility.
Scan pop email box, fetching just headers, allowing
deletions without downloading the complete message.
"""

import poplib, getpass, sys

mailserver = 'your pop email server name here'   # E
mailuser   = 'your pop email user name here'     # E
mailpasswd = getpass.getpass(f'Password for {mailser

print('Connecting...')
server = poplib.POP3(mailserver)
server.user(mailuser)
server.pass_(mailpasswd)

try:
    print(server.getwelcome())
    msgCount, mboxSize = server.stat()
    print('There are', msgCount, 'mail messages, siz
    msginfo = server.list()
    print(msginfo)
    for i in range(msgCount):
        msgnum  = i+1
        msgsize = msginfo[1][i].split()[1]
        resp, hdrlines, octets = server.top(msgnum,
        print('-'*80)
        print('[%d: octets=%d, size=%s]' % (msgnum,
        for line in hdrlines: print(line)
        if input('Print?') in ['y', 'Y']:
            for line in server.retr(msgnum)[1]: prin
        if input('Delete?') in ['y', 'Y']:
            print('deleting')
            server.dele(msgnum)
        else:
            print('skipping')
finally:
    server.quit()                                   #
input('Bye.')                                       #
```

Example B-43. Part7/Self-Study-Demos/sqldbase.py

```
# Database script to populate and query an SQLite da

import sqlite3, time
conn = sqlite3.connect('people.db')     # Filename fo
curs = conn.cursor()                    # Submit SQL

# Make+fill table if doesn't yet exist
tbl = curs.execute('select name from sqlite_master w
if tbl.fetchone() is None:
    print('Making table anew')
    curs.execute('create table people (name, job, pa

    recs = [('Pat', 'mgr', 40000), ('Sue', 'dev', 60
    for rec in recs:
        curs.execute('insert into people values (?,
    conn.commit()

# Show all rows
print('Rows:')
curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

# Show just devs
print('Devs:')
curs.execute("select name, pay from people where job
colnames = [desc[0] for desc in curs.description]
while row := curs.fetchone():
    print('-' * 30)
    for (name, value) in zip(colnames, row):
        print(f'{name:<4} => {value}')
```

```
# Update devs' pay: shown on next run
secs = int(time.time())  # UTC!
curs.execute('update people set pay = ? where job =
conn.commit()
```