Part IV

Additional Topics

Chapter 16

Recursion

16.1 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
    if n <= 0:
        print 'Blastoff!'
    else:
        print n
        countdown(n-1)
```

If `n` is 0 or negative, it outputs the word, "Blastoff!" Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

The execution of `countdown` begins with `n=3`, and since `n` is greater than 0, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is not greater than 0, it outputs the word, "Blastoff!" and then returns.

The `countdown` that got `n=1` returns.

The `countdown` that got `n=2` returns.

The `countdown` that got `n=3` returns.

And then you're back in `__main__`. So, the total output looks like this:

```
3
2
1
Blastoff!
```

A function that calls itself is **recursive**; the process is called **recursion**.

As another example, we can write a function that prints a string `n` times.

```
def print_n(s, n):
    if n <= 0:
        return
    print s
    print_n(s, n-1)
```

If `n <= 0` the `return` statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

The rest of the function is similar to `countdown`: if `n` is greater than 0, it displays `s` and then calls itself to display `s` $n-1$ additional times. So the number of lines of output is `1 + (n - 1)` which, if you do your algebra right, comes out to `n`.

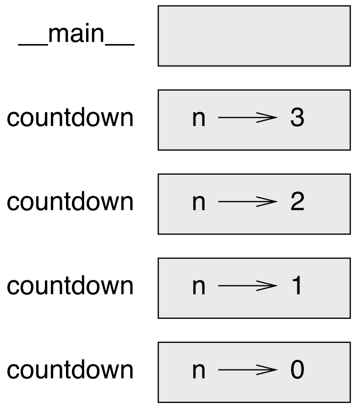
For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

16.2 Stack diagrams for recursive functions

In Section 6.6, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

This figure shows a stack diagram for `countdown` called with `n = 3`:



As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames.

> Draw a stack diagram for `print_n` called with `s = 'Hello'` and `n=4`.

16.3 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
    recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                .
                .
                .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 `recurse` frames on the stack!

16.4 More recursion

We have only covered a small subset of Python, but you might be interested to know that this subset is a *complete* programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

frabjuous: An adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol !, you might get something like this:

$$\begin{aligned} 0! &= 1 \\ n! &= n(n-1)! \end{aligned}$$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$.

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` has a single parameter:

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):
    if n == 0:
        return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$ and then multiply it by n:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

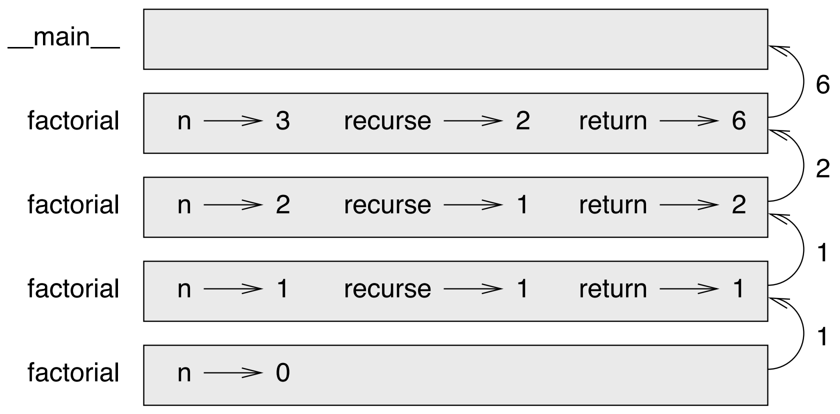
The flow of execution for this program is similar to the flow of `countdown` in Section 16.1. If we call `factorial` with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of `n-1`...

> Since 2 is not 0, we take the second branch and calculate the factorial of `n-1`...
>
> > Since 1 is not 0, we take the second branch and calculate the factorial of `n-1`...
> >
> > > Since 0 *is* 0, we take the first branch and return 1 without making any more recursive calls.
> >
> > The return value (1) is multiplied by *n*, which is 1, and the result is returned.
>
> The return value (1) is multiplied by *n*, which is 2, and the result is returned.

The return value (2) is multiplied by *n*, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

Here is what the stack diagram looks like for this sequence of function calls:



The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them did not execute.

16.5 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. An alternative is what I call the "leap of faith." When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `math.cos` or `math.exp`, you don't examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

The same is true when you call one of your own functions. For example, in Section 6.10, we wrote a function called `is_divisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—examining the code and testing—we can use the function without looking at the code again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (yields the correct result) and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

16.6 One more example

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition:

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2);\end{aligned}$$

Translated into Python, it looks like this:

```
def fibonacci (n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for fairly small values of *n*, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

16.7 Checking types

What happens if we call `factorial` and give it 1.5 as an argument?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. But how can that be? There is a base case—when `n == 0`. But if `n` is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of `n` is 0.5. In the next, it is -0.5. From there, it gets smaller and smaller, but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

```
def factorial (n):
    if not isinstance(n, int):
        print 'Factorial is only defined for integers.'
        return None
    elif n < 0:
        print 'Factorial is only defined for positive integers.'
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Now we have three base cases. The first catches nonintegers and the second catches negative integers. In both cases, the program prints an error message and returns `None` to indicate that something went wrong:

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is only defined for positive integers.
None
```

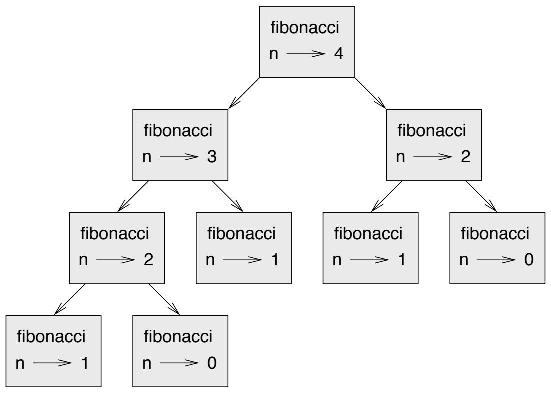
If we get past both checks, then we know that *n* is a positive integer, and we can prove that the recursion terminates.

This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

16.8 Hints

If you played with the `fibonacci` function from Section 16.6, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly.

To understand why, consider this **call graph** for `fibonacci` with `n=4`:



A call graph shows a set function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **hint**. Here is an implementation of `fibonacci` using hints:

```
previous = {0:0, 1:1}

def fibonacci(n):
    if n in previous:
        return previous[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    previous[n] = res
    return res
```

`previous` keeps track of the Fibonacci numbers we already know. We start with only two items: 0 maps to 0 and 1 maps to 1.

Whenever `fibonacci` is called, it checks `previous`. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it.

`previous` is created outside the function, so it belongs to the special frame called `__main__`. Variables in `__main__` are sometimes called **global** because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

Using this version of `fibonacci`, you can compute `fibonacci(40)` in an eyeblink. But if you compute `fibonacci(50)`, you get:

```
>>> fibonacci(50)
12586269025L
```

The `L` at the end of the result indicates that the result is too big to fit into a Python integer. Python converted it to a long integer.

16.9 Debugging

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

- There is something wrong with the arguments the function is getting.
- There is something wrong with the function.
- There is something wrong with the return value or the way it is being used.

To rule out the first possibility, you can add a `print` statement at the beginning of the function and display the values of the parameters (and maybe their types).

If the parameters look good, add a `print` statement before each `return` statement that displays the return value. If possible, check the result by hand. If necessary, call the function with special values where you know what the result should be (as in Section 6.11).

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!).

Adding print statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of `factorial` with print statements:

```
def factorial(n):
    space = ' ' * (4 * n)
    print space, 'factorial', n
    if n == 0:
        print space, 'returning 1'
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print space, 'returning', result
        return result
```

`space` is a string of space characters that controls the indentation of the output. Here is the result of `factorial(5)` :

```
                     factorial 5
                 factorial 4
             factorial 3
         factorial 2
     factorial 1
 factorial 0
 returning 1
     returning 1
         returning 2
             returning 6
                 returning 24
                     returning 120
```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but according to the Fifth Theorem of Debugging:

> A little bit of scaffolding can save a lot of debugging.

16.10 Glossary

recursion: The process of calling the function that is currently executing.

base case: A conditional branch in a recursive function that does not make a recursive call.

infinite recursion: A function that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a runtime error.

16.11 Exercises

Exercise 16.1. *Draw a stack diagram for the following program. What does the program print?*

```
def b(z):
    prod = a(z, z)
    print z, prod
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    sum = x + y + z
    pow = b(sum)**2
    return pow

x = 1
y = x + 1
print c(x, y+3, x+y)
```

Exercise 16.2.

Chapter 17

Tuples

17.1 Tuples are immutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma, Python treats `('a')` as a string in parentheses:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Since `tuple` is the name of a built-in function, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

And the slice operator selects a range of elements.

```
>>> print t[1:3]
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can’t modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

17.2 Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap `a` and `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>>> print uname
monty
>>> print domain
python.org
```

17.3 Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x/y` and then `x%y`. It is better to compute them both at the same time.

The built-in function `divmod` takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)
>>> print t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

`max` and `min` are built-in functions that find the largest and smallest elements of a sequence. `max_min` computes both and returns a tuple of two values.

17.4 Lists and tuples

zip is a built-in function that takes two or more sequences and “zips” them into a list of tuples, where each tuple contains one element from each sequence.

This example zips a string and a list:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples where each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result gets the length of the shorter one.

```
>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print number, letter
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letter and number. The output of this loop is:

```
0 a
1 b
2 c
```

If you combine zip, for and tuple assignment, you get a standard idiom for traversing two (or more) sequences at the same time. For example, has_match takes two sequences, t1 and t2, and returns True if there is an index i such that t1[i] == t2[i]:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate:

```
for index, element in enumerate('abc'):
    print index, element
```

The output of this loop is:

```
0 a
1 b
2 c
```

Again.

17.5 Dictionaries and tuples

Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

As you should expect from a dictionary, the items are in no particular order.

Conversely, you can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

Combining this feature with `zip` yields a concise way to create a dictionary:

```
>>> d = dict(zip('abc', range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method `update` also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

Combining `items`, tuple assignment and `for`, you get the idiom for traversing the keys and values of a dictionary:

```
for key, value in d.items():
    print value, key
```

The output of this loop is:

```
0 a
2 c
1 b
```

Again.

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined `last`, `first` and `number`, we could write:

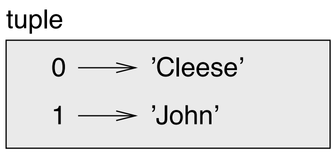
```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

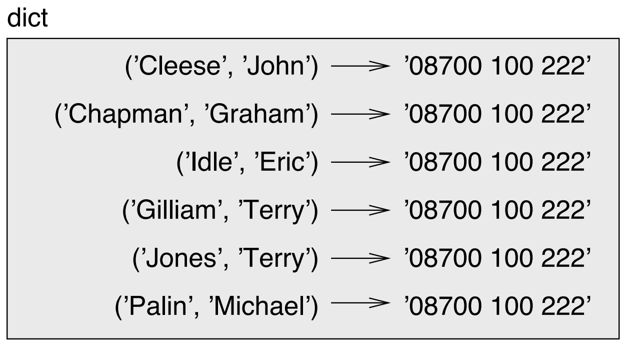
```
for last, first in directory:
    print first, last, directory[last,first]
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple `('Cleese', 'John')` would appear:



But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear:


dict
('Cleese', 'John') ⟶ '08700 100 222'
('Chapman', 'Graham') ⟶ '08700 100 222'
('Idle', 'Eric') ⟶ '08700 100 222'
('Gilliam', 'Terry') ⟶ '08700 100 222'
('Jones', 'Terry') ⟶ '08700 100 222'
('Palin', 'Michael') ⟶ '08700 100 222'

Here the tuples are shown using Python syntax as a graphical shorthand.

The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

17.6 Sorting tuples

The comparison operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on. Here is an example that sorts and prints the key-value pairs of a dictionary:

```
>>> d = {'a': 0, 'c': 2, 'b': 1}
>>> t = d.items()
>>> t.sort()
>>> print t
[('a', 0), ('b', 1), ('c', 2)]
```

To sort by value (rather than key), you can build a list of value-key pairs. One way to do that is to traverse the dictionary items and append tuples onto a list:

```
def value_key_pairs(d):
    res = []
    for key, value in d.items():
        res.append((value, key))
    return res
```

The argument for `append` has two sets of parentheses: one because its an argument and the other because it is a tuple.
Exercise 17.1. *Draw a diagram that shows the final state of* `value_key_pairs` *with* `d = {'a': 0, 'c': 2, 'b': 1}`.

17.7 Sequences of sequences

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how and why do you choose one over the others.

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

- In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list.
- If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
- If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. But Python provides the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new list with the same elements in a different order.

17.8 Glossary

tuple: An immutable sequence of elements.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

17.9 Exercises

Exercise 17.2. *Write a function called* `most_frequent` *that takes a string and prints the 3 most common letters in the string.*
Exercise 17.3. *Write a program that reads a word list from a file (see Section 19.1) and prints all the sets of words that are anagrams.*

Here is an example of what the output might look like:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

Hint: you might want to build a dictionary that maps from a set of letters to a list of words that can be spelled with those letters. The question is, how can you represent the set of letters in a way that can be used as a key?
Exercise 17.4. *Modify the previous program so that it prints the largest set of anagrams first, followed by the second largest set, and so on.*