The Art and Craft of Programming
Python Edition

John C. Lusth

April 29, 2013

Contents

1 Starting Out 9

1.1 Running Python 9

2 Literals 11

2.1 Integers 11

2.2 Real Numbers 12

2.3 Strings 12

2.4 True, False, and None 13

2.5 Collections of literals 13

2.6 Indexing into Arrays 14

3 Combining Literals 17

3.1 Numeric operators 18

3.2 Comparing things 19

3.3 Combining comparisons 20

4 Precedence and Associativity 23

4.1 Precedence 23

4.2 Associativity 23

5 Variables 25

5.1 Variables 25

5.2 Variable naming 27

6 Assignment 29

6.1 Precedence and Associativity of Assignment 30

6.2 Assignment and Arrays 31

6.3 Assignment Patterns 31

6.3.1 The Transfer Pattern . . . 31
6.3.2 The Update Pattern . . . 33
6.3.3 The Throw-away Pattern . . . 34
6.3.4 The Throw-away Pattern and Functions . . . 34
6.4 About Patterns . . . 35
7 Conditionals . . . 37
7.1 Logical expressions . . . 37
7.2 Logical operators . . . 37
7.3 Short circuiting . . . 38
7.4 If expressions . . . 38
7.5 if-elif-else chains . . . 39
8 Functions . . . 41
8.1 Encapsulating a series of operations . . . 41
8.2 Passing arguments . . . 42
8.3 Creating functions on the fly (ADVANCED) . . . 43
8.4 The Function and Procedure Patterns . . . 45
9 Python Programs and Using Files . . . 47
9.1 Your first program . . . 47
9.2 Vim and Python . . . 47
9.3 A Neat Macro . . . 48
9.4 Writing Python Programs . . . 48
9.5 Order of definitions . . . 49
9.6 Importing code . . . 50
10 Input and Output . . . 51
10.1 Input . . . 51
10.1.1 Reading from the keyboard . . . 51
10.1.2 Reading From the command line . . . 52
10.1.3 Reading from files . . . 55
10.2 Output . . . 58
10.2.1 Writing to the console . . . 58

10.3 Writing to a file . . . 59
11 More about Functions . . . 61
11.1 Built-in Functions . . . 61
11.2 Function syntax . . . 62
11.3 Function Objects . . . 62
11.4 Calling Functions . . . 63
11.5 Returning from functions . . . 63
12 Scope . . . 65
12.1 In Scope or Out . . . 65
12.1.1 The Local Variable Pattern . . . 66
12.1.2 The Non-local Variable Pattern . . . 66
12.1.3 The Accessible Variable Pattern . . . 67
12.1.4 The Tinted Windows Pattern . . . 67
12.1.5 Tinted Windows with Parallel Scopes . . . 68
12.2 Alternate terminology . . . 69
12.3 Three Scope Rules . . . 69
12.4 Shadowing . . . 69
12.5 Modules . . . 71
13 Loops . . . 73
13.1 Other loops . . . 74
13.2 The *counting* pattern . . . 75
13.3 The *filtered-count* pattern . . . 75
13.4 The *accumulate* pattern . . . 76
13.5 The *filtered-accumulate* pattern . . . 76
13.6 The *search* pattern . . . 76
13.7 The *extreme* pattern . . . 77
13.8 The *extreme-index* pattern . . . 78
13.9 The *filter* pattern . . . 78
13.10 The *map* pattern . . . 79
13.11 The *shuffle* pattern . . . 79

13.12 The *merge* pattern . . . 81
13.13 The *fossilized* pattern . . . 83
13.14 The *missed-condition* pattern . . . 83
13.15 Code . . . 83
14 More on Input . . . 85
14.1 Converting command line arguments en mass . . . 85
14.2 Reading individual items from files . . . 86
14.3 Reading Tokens into an Array . . . 87
14.4 Reading Records into an Array . . . 88
14.5 Creating an Array of Records . . . 90
14.6 Other Scanner Methods . . . 90
15 Lists . . . 93
15.1 The Node Data Structure . . . 93
15.2 The Linked-List Data Structure . . . 95
15.3 More on Walking lists . . . 101
15.4 A Walking Application . . . 102
15.5 Processing Lists versus Arrays . . . 104
15.5.1 The *filter* and *map* patterns . . . 106
15.5.2 The *shuffle* and *merge* patterns . . . 106
15.5.3 The *fossilized* pattern . . . 107
15.5.4 The *wrong-spot* pattern . . . 107
15.6 Why Lists? . . . 107
15.7 Problems . . . 108
15.8 Code . . . 109
16 Recursion . . . 111
16.1 The parts of a recursive function . . . 113
16.2 The greatest common divisor . . . 113
16.3 The Fibonacci sequence . . . 114
16.4 Manipulating arrays and lists with recursion . . . 116
16.5 The *counting* pattern . . . 117

16.6 The *accumulate* pattern . . . 117
16.7 The *filtered-count* and *filtered-accumulate* patterns . . . 118
16.8 The *filter* pattern . . . 119
16.9 The *map* pattern . . . 119
16.10 The *search* pattern . . . 120
16.11 The *shuffle* pattern . . . 120
16.12 The *merge* pattern . . . 121
16.13 The *generic merge* pattern . . . 122
16.14 The *fossilized* pattern . . . 123
16.15 The *bottomless* pattern . . . 123
16.16 Code . . . 125
17 Comparing Recursion and Looping . . . 127
17.1 Factorial . . . 127
17.2 The greatest common divisor . . . 128
17.3 The Fibonacci sequence . . . 129
17.4 CHALLENGE: Transforming loops into recursions . . . 130
17.5 Code . . . 132

Chapter 1

Starting Out

A word of warning: if you require fancy graphically oriented development environments, you will be sorely disappointed in the Python Programming Language. Python is programming at its simplest: a prompt at which you type in expressions which the Python interpreter evaluates. Of course, you can create Python programs using your favorite text editor (mine is *vim*). Python can be used as a scripting language, as well.

1.1 Running Python

In a terminal window, simply type the command:

```
python3
```

and press the <Enter> key. You should be rewarded with the Python prompt.

```
>>>
```

At this point, you are ready to proceed to next chapter.

Chapter 2

Literals

Python works by figuring out the meaning or value of some code. This is true for the tiniest pieces of code to the largest programs. The process of finding out the meaning of code is known as *evaluation*.

The things whose values are the things themselves are known as *literals*. The literals of Python can be categorized by the following types: *integers*, *real numbers*, *strings*, BOOLEANS, and *arrays*.

Python (or more correctly, the Python interpreter) responds to literals by echoing back the literal itself. Here are examples of each of the types:

```
>>> 3
3

>>> -4.9
-4.9

>>> "hello"
'hello'

>>> True
True

>>>[3, -4.9, "hello"]
[3, -4.9, 'hello']
```

Let's examine the five types in more detail.

2.1 Integers

Integers are numbers without any fractional parts. Examples of integers are:

```
>>> 3
3

>>> -5
-5

>>> 0
0
```

Integers must begin with a digit or a minus sign. The initial minus sign must immediately be followed by a digit.

2.2 Real Numbers

Reals are numbers that do have a fractional part (even if that fractional part is zero!). Examples of real numbers are:

```
>>> 3.2
3.2

>>> 4.0
4.00000000000

>>> 5.
5.00000000000

>>> 0.3
0.30000000000

>>> .3
0.30000000000

>>> 3.0e-4
0.0003

>>> 3e4
30000.0

>>> .000000987654321
9.87654321e-07
```

Real numbers must start with a digit or a minus sign or a decimal point. An initial minus sign must immediately be followed by a digit or a decimal point. An initial decimal point must immediately be followed by a digit. Python accepts real numbers in scientific notation. For example, $3.0 * 10^{-11}$ would be entered as 3.0e-11. The "e" stands for exponent and the 10 is understood, so e-11 means multiply whatever precedes the e by 10^{-11}.

The Python interpreter can hold huge numbers, limited by only the amount of memory available to the interpreter, but holds only 15 digits after the decimal point:

```
>>> 1.2345678987654329
1.234567898765433
```

Note that Python rounds up or rounds down, as necessary.

Numbers greater than 10^6 and less than 10^{-6} are displayed in scientific notation.

2.3 Strings

Strings are sequences of characters delineated by double quotation marks:

```
>>> "hello, world!"
'hello, world!'

>>> "x\nx"
'x\nx'

>>> "\"z\""
'"z"'

>>> ""
''
```

Python accepts both double quotes and single quotes to delineate strings. In this text, we will use the convention that double quotes are used for strings of multiple characters and single quotes for strings consisting of a single character.

Characters in a string can be *escaped* (or quoted) with the backslash character, which changes the meaning of some characters. For example, the character *n*, in a string refers to the letter *n* while the character sequence *n* refers to the *newline* character. A backslash also changes the meaning of the letter *t*, converting it into a tab character. You can also quote single and double quotes with backslashes. When other characters are escaped, it is assumed the backslash is a character of the string and it is escaped (with a backslash) in the result:

```
>>> "\z"
'\\z'
```

Note that Python, when asked the value of strings that contain newline and tab characters, displays them as escaped characters. When newline and tab characters in a string are printed in a program, however, they are displayed as actual newline and tab characters, respectively. As already noted, double and single quotes can be embedded in a string by quoting them with backslashes. A string with no characters between the double quotes is known as an empty string.

Unlike some languages, there is no character type in Python. A single character *a*, for example, is entered as the string `"a"` or `'a'`.

2.4 True, False, and None

There are two special literals, `True` and `False`. These literals are known as the BOOLEAN values and are used to guide the flow of a program. The term BOOLEAN is derived from the last name of George Boole, who, in his 1854 paper *An Investigation of the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities*, laid one of the cornerstones of the modern digital computer. The so-called BOOLEAN logic or BOOLEAN algebra is concerned with the rules of combining truth values (i.e., true or false). As we will see, knowledge of such rules will be important for making Python programs behave properly. In particular, BOOLEAN expressions will be used to control conditionals and loops.

Another special literal is `None`. This literal is used to indicate the end of lists; it also is used to indicates something that has not yet been created. More on `None` when we cover lists and objects.

2.5 Collections of literals

If you read any other text on Python, the basic way of grouping literals together (rather like throwing a bunch of groceries in a shopping bag) is called a *list*. However, in all Python implementations I have tested,

these lists are not lists at all, at least in the Computer Science sense, but are actually *arrays*, specifically dynamic arrays. Because this text is a Computer Science text, rather than a programming text, we will use the term *array* rather than list and introduce true lists in a later chapter[1].

Arrays are just collections of values. One creates an array by enclosing a comma-separated listing of values in square brackets. The simplest array is empty:

```
>>>[]
[]
```

Arrays can contain any values:

```
>>>[2, "help", len]
[2, 'help', <built-in function len>]
```

The first value is an integer, the second a string, and the third is something known as a function. We will learn more about functions later, but the *len* function is used to tell us how many items are in an array:

```
>>> len([2, "help", len])
3
```

As expected, the *len* function tells us that the array `[2, "help", len]` has three items in it.

Arrays can even contain arrays!

```
>>>[0, [3, 2, 1] 4]
[0, [3, 2, 1] 4]
```

An array is something known as a *data structure*; data structures are extremely important in writing sophisticated programs.

2.6 Indexing into Arrays

You can pull out an item from an array by using *bracket notation*. With bracket notation, you specify exactly which element (or elements) you wish to extract from the array. This specification is called an *index*. The first element of the array has index 0, the second index 1, and so on. This concept of having the first element having an index of zero is known as *zero-based counting*, a common concept in Computer Science. Here is some code that extracts the first element of an array. Note that the first interaction creates a *variable* named items that points to an array of three elements.

```
>>> items = ['a', True, 7]

>>> items
['a', True, 7]
```

[1]You might be wondering how I know Python lists are actually arrays. I came to this conclusion by looking at how long it took to access items in a very, very long list. In a list, accessing items at the back should take much longer than accessing items at the front. In an array, the amount of time should be the same. In my tests, the time to access an item was independent of the item's position. Therefore, Python lists must be arrays of some sort. You will learn more about this kind of analysis in your next Computer Science class.

```
>>> items[0]
'a'

>>> items[1]
True

>>> items[2]
7

>>> items
['a', True, 7]
```

Note that extracting an item from an array leaves the array unchanged. What happens if our index is too large?

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

Not surprisingly, we get an error. Note that error message refers to a "list" as opposed to an array. Do not be confused.

There is a special notation for extracting *more than one* element of an array. This notation is known as a *slice*. Here is a slice that extracts all but the first element of an array:

```
>>> items[1:]
[True, 7]
```

This particular slice (you can slice an array many different ways) says, start extracting at the second item (which has index one) and go to the end. This operation is so common, it has a special name, taking the *tail* of an array.

Here is a slice that says, start extracting at the first element (which has index 0) and go up to, but do not include, the element with index 2:

```
>>> items[0:2]
['a', True]
```

We will see more of slicing when we study recursion in a later chapter.

Chapter 3

Combining Literals

Like the literals themselves, combinations of literals are also expressions. For example, suppose you have forgotten your times table and aren't quite sure whether 8 times 7 is 54 or 56. We can ask Python, presenting the interpreter with the expression:

```
>>> 8 * 7
56
>>> 8*7
56
```

Pressing the Enter key signals the end of the expression. The multiplication sign * is known as an *operator*, as it *operates* on the 8 and the 7, producing an equivalent literal value. The 8 and the 7 are known as *operands*. It seems that the actual names of various operands are not being taught anymore, so for nostalgia's sake, here they are. The operand to the left of the multiplication sign (in this case the 8) is known as the *multiplicand*. The operand to the right (in this case the 7) is known as the *multiplier*. The result is known as the *product*.

The operands of the other basic operators have special names too. For addition, the left operand is known as the *augend* and the right operand is known as the *addend*. The result is known as the *sum*. For subtraction, the left operand is the *minuend*, the right the *subtrahend*, and the result as the *difference*. For division (and I think this is still taught), the left operand is the *dividend*, the right operand is the *divisor*, and the result is the *quotient*. Finally, for exponentiation, which is shorthand for repeated multiplication:

```
>>> 3 ** 4
81
>>> 3 * 3 * 3 * 3
81
```

the left operand is the *base* and the right operand is the *exponent*.

In general, we will separate operators from their operands by spaces, tabs, or newlines, collectively known as *whitespace*.[1] It's not necessary to do so, but it makes your code easier to read.

Python always takes in an expression and displays an equivalent literal expression (*e.g.*, integer or real). All Python operators are binary, meaning they operate on exactly two operands. We first look at the numeric operators.

[1] Computer Scientists, when they have to write their annual reports, often refer to the things they are reporting on as *darkspace*. It's always good to have a lot of darkspace in your annual report!

3.1 Numeric operators

If it makes sense to add two things together, you can probably do it in Python using the + operator. For example:

```
>>> 2 + 3
5

>>> 1.9 + 3.1
5.0
```

One can see that if one adds two integers, the result is an integer. If one does the same with two reals, the result is a real.

You can even add strings with strings and arrays with arrays:

```
>>> "hello" + "world"
'helloworld'

>>> [1, 3, 5] + [2, 4, 6]
[1, 3, 5, 2, 4, 6]
```

The process of joining strings and arrays together is known as *concatenation*. Array concatenation is an expensive processive, since you need to the values of both arrays source arrays to build the resulting array.

Things get more interesting when you add things having different types. Adding an integer and a real (in any order) always yields a real.

```
>>> 2 + 3.3
5.3

>>> 3.3 + 2
5.3
```

Adding an string to an integer (with an augend integer) yields an error; the types are not "close" enough, like they are with integers and reals:

```
>>> 2 + "hello"
TypeError: unsupported operand types(s) for +: 'int' and 'str'
```

In general, when adding two things, the types must match or nearly match.

You can multiply strings and arrays with numbers:

```
>>> "hello" * 3
'hellohellohello'

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

Subtraction and division of numbers follow the same rules as addition. However, these operators, as defined, do not work for strings and arrays.

Of special note is the division operator with respect to integer operands. Consider evaluating the following expression:

```
15 / 2
```

If one asked the Python interpreter to perform this task, the result would be 7.5, as expected. However, often we wish for just the quotient without the remainder. In this case, the quotient is 7 and the remainder is 0.5. The double forward slash operator is Python's quotient operator; if we ask the interpreter to evaluate

```
14 // 5
```

the result would be 2, not 2.8. Use of the quotient operator is known as *integer division.*[2]

The complement to integer division is the modulus operator %. While the result of integer division is the quotient, the result of the modulus operator is the remainder. Thus

```
14 % 5
```

evaluates to 4 since 4 is left over when 5 is divided into 14. To check if this is true, one can ask the interpreter to evaluate:

```
(14 // 5 * 5) + (14 % 5) == 14
```

This complicated expression asks the question "is it true that the quotient times the divisor plus the remainder is equal to the original dividend?". The Python interpreter will respond that, indeed, it is true. The reason for the parentheses is delineate the quotient and the remainder within the addition. The parentheses can also be used to change the *precedence* of operators; this is is explained in more detail in the next chapter.

3.2 Comparing things

Remember the BOOLEAN literals, `True` and `False`? We can use the BOOLEAN comparison operators to generate such values. For example, we can ask if 3 is less than 4:

```
>>> 3 < 4
True
```

The interpreters response says that, indeed, 3 is less than 4. If it were not, the interpreter would respond with `False`. Besides `<` (less than), there are other BOOLEAN comparison operators: `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `==` (equal to), and `!=` (not equal to).

Note that two equal signs are used to see if to things are the same. A single equals sign is reserved for the assignment operator, which you will learn about in a later chapter.

Besides integers, we can compare reals with reals, strings with strings, and arrays with arrays using the comparison operators:

[2] In most other modern programming languages, a single / means integer division, so don't forget this when you learn a new language!

```
>>> "apple" < "banana"
True

>>> [1, 2, 3] < [1, 2, 4]
True
```

In general, it is illegal to compare integers or reals with strings.

Any Python type can be compared with any other type with the equality and inequality operators, `==` and `!=`. These operators are used to see if two things are the same or not the same, as the case may be. Usually, for two things to be considered the same, they have to be the same kind of thing or *type*. For example, the expression: `123 == "123"` resolves to `False`, because integers are not strings and vice versa. Likewise, `"False" != False` resolves to `True`, because strings are not BOOLEANS and vice versa. There is an exception, though. If an integer is compared with a real, the integer is converted into a real before the comparison is made. In the case of arrays, `==` will return `False` if the arrays have differing lengths. If the arrays have equal length, then each item in one array is compared using `==` to the respective item in the other array. All individual items must be equal for the two arrays to be equal. For example `[1, [2, 1, 0], 3]` and `[1, [2, 1, 8], 3]` would be considered "not equal", since the middle items would not be considered equal.

3.3 Combining comparisons

We can combine comparisons with the BOOLEAN logical connectives `and` and `or`:

```
>>> 3 < 4 and 4 < 5
True

>>> 3 < 4 or 4 < 5
True

>>> 3 < 4 and 5 < 4
False

>>> 3 < 4 or 5 < 4
True
```

The first interaction asks if both the expression `3 < 4` and the expression `4 < 5` are true. Since both are, the interpreter responds with `True`. The second interaction asks if at least one of the expressions is true. Again, the interpreter responds with `True`. The difference between `and` and `or` is illustrated in the last two interactions. Since only one expression is true (the latter expression being false) only the `or` operator yields a true value.

There is one more BOOLEAN logic operation, called *not*. It simply reverses the value of the expression to which it is attached. The *not* operator can only be called as a function (since it is not a binary operator). Since you do not yet know about functions, I'll show you what it looks like but won't yet explain its actions.

```
>>> not(3 < 4 and 4 < 5)
False

>>> not(3 < 4 or 4 < 5)
False
```

```
>>> not(3 < 4 and 5 < 4)
True

>>> not(3 < 4 or 5 < 4)
False
```

Note that we attached *not* to each of the previous expressions involving the logical connectives. Note also that the response of the interpreter is reversed from before in each case.

Chapter 4

Precedence and Associativity

4.1 Precedence

Precedence (partially) describes the order in which operators, in an expression involving different operators, are evaluated. In Python, the expression

```
3 + 4 < 10 - 2
```

evaluates to true. In particular, `3 + 4` and `10 - 2` are evaluated before the `<`, yielding `7 < 8`, which is indeed true. This implies that `+` and `-` have higher precedence than `<`. If `<` had higher precedence, then `4 < 10` would be evaluated first, yielding `3 + True - 2`, which is nonsensical.

Note that precedence is only a partial ordering. We cannot tell, for example whether `3 + 4` is evaluated before the `10 - 2`, or vice versa. Upon close examination, we see that it does not matter which is performed first as long as both are performed before the expression involving `<` is evaluated.

It is common to assume that the left operand is evaluated before the right operand. For the BOOLEAN connectives `and` and `or`, this is indeed true. But for other operators, such an assumption can lead you into trouble. You will learn why later. For now, remember never, never, never depend on the order in which operands are evaluated!

The lowest precedence operator in Python is the assignment operator which is described later. Next come the BOOLEAN connectives `and` and `or`. At the next higher level are the BOOLEAN comparatives, `<`, `<=`, `>`, `>=`, `==`, and `!=`. After that come the additive arithmetic operators `+` and`-`. Next comes the multiplicative operators `*`, `/` and `%`. Higher still is the exponentiation operator `**`. Finally, at the highest level of precedence is the selection, or *dot*, operator (the dot operator is a period or full-stop). Higher precedence operations are performed before lower precedence operations. Functions which are called with operator syntax have the same precedence level as the mathematical operators.

4.2 Associativity

Associativity describes how multiple expressions connected by operators at the same precedence level are evaluated. All the operators, with the exception of the assignment and exponentiation operators, are left associative. For example the expression 5 - 4 - 3 - 2 - 1 is equivalent to ((((5 - 4) - 3) - 2) - 1). For a left-associative structure, the equivalent, fully parenthesized, structure has open parentheses piling up on the left. If the minus operator was right associative, the equivalent expression would be (5 - (4 - (3 - (2 - 1)))), with the close parentheses piling up on the right. For a commutative operator, it does not matter whether it is left associative or right associative. Subtraction, however, is not commutative, so associativity

does matter. For the given expression, the left associative evaluation is -5. If minus were right associative, the evaluation would be 3.

Chapter 5

Variables

Suppose you found an envelope lying on the street and on the front of the envelope was printed the name *numberOfDogsTeeth*. Suppose further that you opened the envelope and inside was a piece of paper with the number 42 written upon it. What might you conclude from such an encounter? Now suppose you kept walking and found another envelope labeled *meaningOfLifeUniverseEverything* and, again, upon opening it you found a slip of paper with the number 42 on it. Further down the road, you find two more envelopes, entitled *numberOfDotsOnPairOfDice* and *StatuteOfLibertyArmLength*, both of which contain the number 42.

Finally, you find one last envelope labeled *sixTimesNine* and inside of it you, yet again, find the number 42. At this point, you're probably thinking "somebody has an odd affection for the number 42" but then the times table that is stuck somewhere in the dim recesses of your brain begins yelling at you saying "54! It's 54!". After this goes on for an embarrassingly long time, you realize that 6 * 9 is not 42, but 54. So you cross out the 42 in the last envelope and write 54 instead and put the envelope back where you found it.

This strange little story, believe it or not, has profound implications for writing programs that both humans and computers can understand. For programming languages, the envelope is a metaphor for something called a *variable*, which can be thought of as a label for a place in memory where a literal value can reside. In other words, a variable can be thought of as a convenient name for a value. In many programming languages, one can change the value at that memory location, much like replacing the contents of an envelope.[1] A variable is our first encounter with a concept known as *abstraction*, a concept that is fundamental to the whole of computer science.[2]

5.1 Variables

Most likely, you've encountered the term *variable* before. Consider the slope-intercept form of an algebraic equation of a particular line:

$$y = 2x - 3$$

You probably can tell from this equation that the slope of this line is 2 and that it intercepts the y-axis at -3. But what role do the letters y and x actually play? The names x and y are placeholders and stand for the x- and y-coordinates of any conceivable point on that line. Without placeholders, the line would have to be described by listing every point on the line. Since there are an infinite number of points, clearly an exhaustive list is not feasible. As you learned in your algebra class, the common name for a place holder for a specific value is the term *variable*.

[1] Languages that do not allow changes to a variable are called *functional languages*. Python is an "impure" functional language since it is *mostly* functional but allows for variable modification.

[2] Another fundamental concept is *analogy* and if you understand the purpose of the envelope story after reading this section, you're well on your way to being a computer scientist!

One can generalize the above line resulting in an equation that describes every line.[3]

$$y = mx + b$$

Here, the variable m stands for the slope and b stands for the y-intercept. Clearly, this equation was not dreamed up by an English-speaking computer scientist; a cardinal rule is to choose good names or mnemonics for variables, such as s for slope and i for intercept. But alas, for historical reasons, we are stuck with m and b.

The term *variable* is also used in most programming languages, including Python, and the term has roughly the equivalent meaning. The difference is programming languages use the envelope metaphor while algebraic meaning of variable is an equivalence to a value.[4] The difference is purely philosophical and not worth going into at this time. Suppose you found three envelopes, marked m, x, and b, and inside those three envelopes you found the numbers 6, 9, and -12 respectively. If you were asked to make a y envelope, what number should you put inside? If the number 42 in the *sixTimesNine* envelope in the previous story did not bother you (*e.g.*, your internal times table was nowhere to be found), perhaps you might need a little help in completing your task. We can have Python calculate this number with the following dialog:

```
>>> m = 6

>>> x = 9

>>> b = -12

>>> y = m * x + b

>>> y
42
```

The Python interpreter, when asked to compute the value of an expression containing variables, goes to those envelopes (so to speak) and retrieves the values stored there. Note also that Python requires the use of the multiplication sign to multiply the slope m by the x value. In the algebraic equation, the multiplication sign is elided, but is required here.

One creates variables in Python by simply assigning a value to the variable.[5]. If the variable does not exist, it is created; if it does exist, it's value is updated. Note that the interpreter does not give a response when a variable is created or updated.

Here are some more examples of variable creation:

```
>>> dots = 42

>>> bones = 206
```

[3]The third great fundamental concept in computer science is *generalization*. In particular, computer scientists are always trying to make things more abstract and more general (but not overly so). The reason is that software/systems/models exhibiting the proper levels of abstraction and generalization are much much easier to understand and modify. This is especially useful when you are required to make a last second change to the software/system/model.

[4]Even the envelope metaphor can be confusing since it implies that two variables having the same value must each have a copy of that value. Otherwise, how can one value be in two envelopes at the same time? For simple literals, copying is the norm. For more complex objects, the cost of copying would be prohibitive. The solution is to storing the *address* of the object, instead of the object itself, in the envelope. Two variables can now "hold" the same object since it is the address is copied.

[5]In many other languages, you have to declare a variable with a special syntax before you can assign a value to it

```
>>> dots
42

>>> bones
206

>>> CLXIV = bones - dots
164
```

After a variable is created/updated, the variable and its value can be used interchangeably. Thus, one use of variables is to set up constants that will be used over and over again. For example, it is an easy matter to set up an equivalence between the variable PI and the real number 3.14159.

```
PI = 3.14159
radius = 10
area = PI * radius * radius
circumference = 2 * PI * radius
```

Notice how the expressions used to compute the values of the variables area and circumference are more readable than if 3.14159 was used instead of PI. In fact, that is one of the main uses of variables, to make code more readable. The second is if the value of PI should change (e.g. a more accurate value of PI is desired,[6] we would only need to change the definition of PI (this assumes, of course, we can store those definitions for later retrieval and do not need to type them into the interpreter again).

5.2 Variable naming

Like many languages, Python is quite restrictive in regards to legal variable names. A variable name must begin with a letter or an underscore and may be followed by any number of letters, digits, or underscores.

Variables are the next layer in a programming languages, resting on the literal expressions and combinations of expressions (which are expressions themselves). In fact, variables can be thought of as an abstraction of the literals and collections of literals. As an analogy, consider your name. Your name is not you, but it is a convenient (and abstract) way of referring to you. In the same way, variables can be considered as the names of things. A variable isn't the thing itself, but a convenient way to referring to the thing.

While Python lets you name variables in wild ways:

```
>>> _1_2_3_iiiiii__ = 7
```

you should temper your creativity if it gets out of hand. For example, rather than use the variable m for the slope, we could use the name *slope* instead:

```
slope = 6
```

We could have also used a different name:

```
_e_p_o_l_s_ = 6
```

[6]The believed value of PI has changed throughout the centuries and not always to be more accurate (see http://en.wikipedia.org/wiki/History_of_Pi)

The name *_e_p_o_l_s_* is a perfectly good variable name from Python's point of view. It is a particularly poor name from the point of making your Python programs readable by you and others. It is important that your variable names reflect their purpose. In the example above, which is the better name: *b*, *i*, *intercept*, or *_t_p_e_c_r_e_t_n_i_* to represent the intercept of a line?

Chapter 6

Assignment

Once a variable has been created, it is possible to change its value, or *binding*, using the assignment operator. Consider the following interaction with the interpreter:

```
>>> BLACK = 1            # creation
>>> BROWN = 2            # creation
>>> GREEN = 3            # creation

>>> eyeColor = BLACK     # creation

>>> eyeColor             # reference
1

>>> eyeColor = GREEN     # assignment!

>>> eyeColor == BLACK    # equality
False

>>> eyeColor == BROWN    # equality
False

>>> eyeColor == GREEN    # equality
True
```

Note that the # sign is the comment character in Python; it and any following characters on the line are ignored.

The operator/variable = (equals sign) is bound to the *assignment* function. The assignment function, however, is not like the operators + and *. Recall that + and the like evaluate the things on either side (recall that those things on either side are generically known as operands) before combining them. For =, the left operand is not evaluated: (if it were, the assignment

```
eyeColor = GREEN
```

would attempt to assign the value of 1 to be 3. In general, an operator which does not evaluate all its arguments is known as a *special form*.

The last two expressions given to the interpreter in the previous interaction refer to the == (equality) operator. This == operator returns true if its operands refer to the same thing and false otherwise.

Another thing to note in the above interaction is that the variables BLACK, GREEN, and BROWN are not meant to change from their initial values. We denote variables whose values aren't supposed to change by naming the variable using (mostly) capital letters (this convention is borrowed from earlier programming languages). The use of caps emphasizes the constant nature of the (not too) variable.

In the above interaction with the interpreter, we use the integers 1, 2, and 3 to represent the colors black, brown, and green. By abstracting 1, 2, and 3 and giving them meaningful names (i.e., BLACK, BROWN, and GREEN) we find it easy to read code that assigns and tests eye color. We do this because it is difficult to remember which integer is assigned to which color. Without the variables BLACK, BROWN, and GREEN, we have to keep little notes somewhere to remind ourselves what's what. Here is an equivalent interaction with the interpreter without the use of the variables BLACK, GREEN, and BROWN.

```
>>> eyeColor = 1
1

>>> eyeColor
1

>>> eyeColor = 3
3

>>> eyeColor == 2
False

>>> eyeColor == 3
True
```

In this interaction, the meaning of *eyeColor* is not so obvious. We know its a 3, but what eye color does 3 represent? When numbers appear directly in code, they are referred to as *magic numbers* because they obviously mean something and serve some purpose, but how they make the code work correctly is not always readily apparent, much like a magic trick. Magic numbers are to be avoided. Using well-name constants (or variables if constants are not part of the programming language) is considered stylistically superior.

6.1 Precedence and Associativity of Assignment

Assignment has the lowest precedence among the binary operators. It is also right associative. The right associativity allows for statements like

```
a = b = c = d = 0
```

which conveniently assigns a zero to four variables at once and, because of the right associative nature of the operator, is equivalent to:

```
(a = (b = (c = (d = 0))))
```

The resulting value of an assignment operation is the value assigned, so the assignment `d = 0` returns 0, which is, in turned, assigned to *c* and so on.

6.2 Assignment and Arrays

You can change a particular element of a array by assigning a new value to the index of that element by using bracket notation:

```
>>> items = ['a', True, 7]

>>> items[0] = 'b'

>>> items
['b', True, 7]`
```

As expected, assigning to index 0 replaces the first element. In the example, the first element `'a'` is replaced with `'b'`.

What bracket notation would you use to change the 7 in the array to a 13? The superior student will experiment with the Python interpreter to verify his or her guess.

6.3 Assignment Patterns

The art of writing programs lies in the ability to recognize and use patterns that have appeared since the very first programs were written. In this text, we take a pattern approach to teaching how to program. For the topic at hand, we will give a number of patterns that you should be able to recognize to use or avoid as the case may be.

6.3.1 The Transfer Pattern

The *transfer* pattern is used to change the value of a variable based upon the value of another variable. Suppose we have a variable named *alpha* which is initialized to 3 and a variable *beta* which is initialized to 10:

```
alpha = 3
beta = 10
```

Now consider the statement:

```
alpha = beta
```

This statement is read like this: make the new value of *alpha* equal to the value of *beta*, throwing away the old value of *alpha*. What is the value of *alpha* after that statement is executed?

The new value of *alpha* is 10.

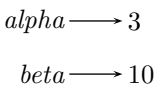
The *transfer* pattern tells us that value of *beta* is imprinted on *alpha* at the moment of assignment but in no case are *alpha* and *beta* conjoined in anyway in the future. Think of it this way. Suppose your friend spray paints her bike neon green. You like the color so much you spray paint your bike neon green as well. This is like assignment: you made the value (color) of your bike the same value (color) as your friend's bike. Does this mean your bike and your friend's bike will always have the same color forever? Suppose your friend

repaints her bike. Will your bike automatically become the new color as well? Or suppose you repaint your bike. Will your friend's bike automatically assume the color of your bike?

A graphical way to see this is to use the idea of a *pointer*. When a variable is assigned a value, we will say that variable *points* to the value. For the variables:

```
alpha = 3
beta = 10
```

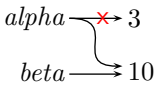
a pointer diagram would look like:



Now when we assign *alpha* the value of *beta*:

```
alpha = beta
```

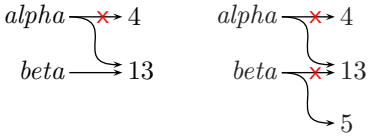
we cross out *alpha*'s pointer and replace it with a pointer to whatever *beta* points:



Note that we *never* point a variable to another variable; we only point variables to values[1]. To test your understanding, what happens if the following code is executed:

```
alpha = 4
beta = 13
alpha = beta
beta = 5
```

What are the final values of *alpha* and *beta*? Here, the pointer diagram would look like:



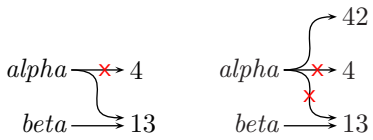
where the picture on the left illustrates what things look like after *alpha* gets *beta*'s value and the picture on the right after *beta* gets the value of 5. So we can see that the final value of *alpha* is 13 and *beta* is 5.

To further test your understanding, what happens if the following code is executed:

[1]There are languages where it is possible to point a variable to another variable. The most famous of these languages are C and C++. Python, however, does not allow you to point a variable to a variable.

```
alpha = 4
beta = 13
alpha = beta
alpha = 42
```

What are the final values of *alpha* and *beta*? Here, the pointer diagram would look like:



where the picture on the left illustrates what things look like after *alpha* gets *beta*'s value and the picture on the right after *alpha* gets the value of 42. So we can see that the final value of *alpha* is 42 and *beta* is 13.

6.3.2 The Update Pattern

The *update* pattern is used to change the value of a variable based upon the original value of the variable. Suppose we have a variable named *counter* which is initialized to zero:

```
counter = 0
```

Now consider the statement:

```
counter = counter + 1
```

This statement is read like this: make the new value of counter equal to the old value of counter plus one. Since the old value is zero, the new value is one. Consider this sequence:

```
counter = 0
counter = counter + 1
counter = counter + 1
counter = counter + 1
counter = counter + 1
counter = counter + 1
```

What is the value of *counter* after the following code is executed?

The value of *counter* is 5.

There is another form of this update:

```
counter = 0
counter += 5
```

The operator `+=` says to update the variable on the left by adding in the value on the right to the current value of the variable.

The *update* pattern can be used to sum a number of variables. Suppose we wish to compute the sum of the variables *a*, *b*, *c*, *d*, and *e*. The obvious way to do this is with one statement:

```
sum = a + b + c + d + e
```

However, we can use the *update* pattern as well:

```
sum = 0
sum = sum + a
sum = sum + b
sum = sum + c
sum = sum + d
sum = sum + e
```

If *a* is 1, *b* is 2, *c* is 3, *d* is 4, and *e* is 5, then the value of sum in both cases is 15. Why would we ever want to use the *update* pattern for computing a sum when the first version is so much more compact and readable? The answer is...you'll have to wait until we cover a programming concept called a *loop*. With loops, the *update* pattern is almost always used to compute sums, products, etc.

6.3.3 The Throw-away Pattern

The *throw-away* pattern is a mistaken attempt to use the *update* pattern. In the *update* pattern, we use the original value of the variable to compute the new value of the variable. Here again is the classic example of incrementing a counter:

```
counter = counter + 1
```

In the *throw-away* pattern, the new value is computed but it the variable is not reassigned, nor is the new value stored anywhere. Many novice programmers attempt to update a counter simply by computing the new value:

```
count + 1       # throw-away!
```

Python does all the work to compute the new value, but since the new value is not assigned to any variable, the new value is thrown away.

6.3.4 The Throw-away Pattern and Functions

The *throw-away* pattern applies to function calls as well. We haven't discussed functions much, but the following example is easy enough to understand. First we define a function that computes some value:

```
def inc(x):
    return x + 1
```

This function returns a value one greater than the value given to it, but what the function actually does is irrelevant to this discussion. That said, we want to start indoctrinating you on the use of functions. Repeat this ten times:

> We always do four things with functions: *define them*, *call them*, *return something*, and *save the return value.*

To call the function, we use the function name followed by a set of parentheses. Inside the parentheses, we place the value we wish to send to the function. Consider this code, which includes a call to the function *inc*:

```
y = 4
y = inc(y)
print("y is",y)
```

If we were to run this code, we would see the following output:

```
y is 5
```

The value of y, 4, is sent to the function which adds one to the given value and returns this new value. This new value, 5, is assigned back to y. Thus we see that y has a new value of 5.

Suppose, we run the following code instead:

```
y = 4
inc(y)
print("y is",y)
```

Note that the return value of the function *inc* is not assigned to any variable. Therefore, the return value is thrown away and the output becomes:

```
y is 4
```

The variable y is unchanged because it was never reassigned.

6.4 About Patterns

As you can see from above, not all patterns are good ones. However, we often mistakenly use bad patterns when programming. If we can recognize those bad patterns more readily, our job of producing a correctly working program is greatly simplified.

Chapter 7

Conditionals

Conditionals implement decision points in a computer program. Suppose you have a program that performs some task on an image. You may well have a point in the program where you do one thing if the image is a JPEG and quite another thing if the image is a GIF file. Likely, at this point, your program will include a conditional expression to make this decision.

Before learning about conditionals, it is important to learn about logical expressions. Such expressions are the core of conditionals and loops.[1]

7.1 Logical expressions

A logical expression evaluates to a truth value, in essence true or false. For example, the expression $x > 0$ resolves to true if x is positive and false if x is negative or zero. In Python, truth is represented by the symbol `True` and falsehood by the symbol `False`. Together, these two symbols are known as BOOLEAN values.

One can assign truth values to variables:

```
>>> c = -1
>>> z = (c > 0);

>>> z
False
```

Here, the variable z would be assigned a value of `True` if c is positive; since c is negative, it is assigned a value of `False`. The parentheses are added to improve readability.

7.2 Logical operators

Python has the following logical operators:

[1] We will learn about loops in the next chapter.

`==`	equal to
`!=`	not equal to
`>=`	greater than or equal to
`>`	greater than
`<`	less than
`<=`	less than or equal to
`and`	both must be true
`or`	one or both must be true
`not`	reverses the truth value

The first five operators are used for comparing two things, while the next two operators are the glue that joins up simpler logical expressions into more complex ones. The last operator is used to reverse a logical value from `True` to `False` and vice versa.

Beginning programmers often confuse the `=` operator, which is the assignment operator, with the `==` operator, which is the equality operator. Remember, assignment is used to change the value of a variable while equality can be used to test the value of a variable (without changing its value). Fortunately, Python does detects many attempts to use assignment when equality was intended and will complain appropriately.

7.3 Short circuiting

When evaluating a logical expression, Python evaluates the expression from left to right and stops evaluating as soon as it finds out that the expression is definitely true or definitely false. For example, when encountering the expression:

```
x != 0 and y / x > 2
```

if x has a value of 0, the subexpression on the left side of the *and* connective resolves to false. At this point, there is no way for the entire expression to be true (since both the left hand side and the right hand side must be true for an *and* expression to be true), so the right hand side of the expression is not evaluated. Note that this expression protects against a divide-by-zero error.

7.4 If expressions

Python's *if* expressions are used to conditionally execute code, depending on the truth value of what is known as the *test* expression. One version of *if* has a block of code following the test expression:

Here is an example:

```
if (name == "John"):
    print("What a great name you have!")
```

In this version, when the test expression is true (*i.e.*, the string `"John"` is bound to the variable *name*), then the code that is indented under the *if* is evaluated (i.e., the compliment is printed). The indented code is known as a *block* and all lines in a block must be indented exactly the same amount[2]. If the test expression is false, however the block is not evaluated. In this text, we will enclose the test expression in parentheses even though it is not required by Python. We do that because some important programming languages require the parentheses and we want to get you into the habit.

[2]In general, it is a bad idea to vary the kinds of characters that make up an indentation. One should use just spaces or just tabs, but should not use an indentation made up of spaces *and* tabs.

Here is another form of *if*:

```
if (major == "Computer Science"):
    print("Smart choice!")
else:
    print("Ever think about changing your major?")
```

In this version, *if* has two blocks, one following the test expression and one following the *else* keyword. Note the colons that follow the test expression and the else; these are required by Python. As before, the first block is evaluated if the test expression is true. If the test expression is false, however, the second block is evaluated instead.

7.5 if-elif-else chains

You can chain *if* statements together using the *elif* keyword, as in:

```
if (bases == 4):
    print("HOME RUN!!!")
elif (bases == 3):
    print("Triple!!")
elif (bases == 2):
    print("double!")
elif (bases == 1)
    print("single")
else:
    print("out")
```

The block that is eventually evaluated is directly underneath the first test expression that is true, reading from top to bottom. If no test expression is true, the block associated with the else is evaluated.

What is the difference between *if-elif-else* chains and a sequence of unchained *if*s? Consider this rewrite of the above code:

```
if (bases == 4):
    print("HOME RUN!!!");
if (bases == 3):
    print("Triple!!");
if (bases == 2):
    print("double!");
if (bases == 1):
    print("single");
else:
    print("out");
```

In the second version, there are four *if* statements and the *else* belongs to the last *if*. Does this behave exactly the same? The answer is, it depends. Suppose the value of the variable *bases* is 0. Then both versions print:

```
out
```

However, if the value of *bases* is 3, for example, the first version prints:

```
triple!!
```

while the second version prints:

```
triple!!
out
```

Why the difference? In the first version, a subsequent test expression is evaluated *only* if all previous test expressions evaluated to false. Once a test expression evaluates to true in an *if-elif-else* chain, the associated block is evaluated and no more processing of the chain is performed. Like the *and* and *or* BOOLEAN connectives, an *if-elif-else* chain short-circuits.

In contrast, the sequences of *if*s in the rewrite are independent; there is no short-circuiting. When the test expression of the first *if* fails, the test expression of the second *if* succeeds and `triple!!` is printed. Now the test expression of the third *if* is tested and fails as well as the test expression of the fourth *if*. But since the fourth *if* has an else, the *else* block is evaluated and `out` is printed.

It is important to know when to use an *if-elif-else* chain and when to use a sequence of independent *if*s. If there should be only one outcome, then use an *if-elif-else* chain. Otherwise, use a sequence of *if*s.

Practice with Boolean expressions

Here is some practice on writing complex Boolean expressions. Your task is to convert the English description into a Python Boolean expression. The variables to be used in the expressions are:

dogSize with string values `"small"`, `"medium"`, and `"large"`
dogHasSpots with Boolean values `True` or `False`
dogAge with positive integer values
dogColor with string values `"white"`, `"black"`, `"red"`, and `"brown"`
catSize with string values `"small"`, `"medium"`, and `"large"`
catHasSpots with Boolean values `True` or `False`
catColor with string values `"white"`, `"black"`, `"orange"`, and `"tabby"`
catEyeColor with string values `"green"`, `"brown"`, `"blue"`
catAge with positive integer values

An old dog is one with an age greater than seven years while an old cat has an age greater than or equal to nine years. A cat and a dog are young if they are younger than three years old.

Example:

Write a boolean expression that captures "Dogs that are large and have spots or cats that are white but do not have blue eyes."

Answer:

```
(dogSize == "large" and dogHasSpots) or (catColor == "white" and catEyeColor != "blue")
```

Chapter 8

Functions

Recall from Chapter 6 the series of expressions we evaluated to find the y-value of a point on the line:

```
y = 5x - 3
```

First, we assigned values to the slope, the x-value, and the y-intercept:

```
>>> m = 5
>>> x = 9
>>> b = -3
```

Once those variables have been assigned, we can compute the value of y:

```
>>> y = m * x + b

>>> y
42
```

Now, suppose we wished to find the y-value corresponding to a different x-value or, worse yet, for a different x-value on a different line. All the work we did would have to be repeated. A *function* is a way to encapsulate all these operations so we can repeat them with a minimum of effort.

8.1 Encapsulating a series of operations

First, we will define a not-too-useful function that calculates y give a slope of 5, a y-intercept of -3, and an x-value of 9 (exactly as above). We do this by wrapping a function around the sequence of operations above. The return value of this function is the computed y value:

```
def y():
    m = 5
    x = 9
    b = -3
    return m * x + b
```

There are a few things to note. The keyword `def` indicates that a function definition is occurring. The name of this particular function is y. The names of the things being sent to the function are given between the

parentheses; since there is nothing between the parentheses, we don't need to send any information to this function when we use it. Together, the first line is known as the *function signature*, which tells you the name of the function and how many values it expects to be sent when it is used.

The stuff indented from the first line of the function definition is called the *function body* and is the code that will be evaluated (or executed) when the function is used. You must remember this: *the function body is not evaluated until the function is actually used.*

Once the function is defined, we can find the value of y repeatedly. Let's assume the function was entered into the file named *line.py*.

First we import the code in *line.py* with the from statement:

```
>>> from line import *   # not line.py!
```

This makes the python interpreter behave as if we had typed in the function definition residing in *line.py* directly into the interpreter. Note, we omit the *.py* extension in the import statement.

After importing the y function, the next thing we do is use it:

```
>>> y()
42
>>> y()
42
```

The use of a function is known as a *function call*. Most function calls look similar: the name of the function to be called followed by a parenthesized list of information to send the function so that it can do its job. Thus, the parentheses after the y indicate that we wish to call the y function and get its value. Because we designed the function to take no information when called, we do not place any information between the parentheses.

Note that when we call the y function again, we get the exact same answer.

The y function, as written, is not too useful in that we cannot use it to compute similar things, such as the y-value for a different value of x. This is because we "hard-wired" the values of b, x, and m in the definition of the function. We can improve the y function by passing in the value of x instead of hard-wiring the value to 9.

8.2 Passing arguments

A hallmark of a good function is that it lets you compute more than one thing. We can modify our y function to *take in* the value of x in which we are interested. In this way, we can compute more than one value of y. We do this by *passing* in some information. This information that is passed to a function in a function call is known as an *argument*[1], in this case, the value of x.

```
def y(x):
    slope = 5
    intercept = -3
    return slope * x + intercept
```

[1]The information that is passed into a function is collectively known as *arguments*. The arguments are then bound to the variables that are found after the function name in the function definition.

Note that we have moved the variable x from the body of the function to between the parentheses. We have also refrained from giving it a value since its value is to be sent to the function when the function is called. What we have done is to *parameterize* the function to make it more general and more useful. The variable x is now called a *formal parameter* since it sits between the parentheses in the first line of the function definition.

Now we can compute y for an infinite number of x's:

```
>>> from line.py import *
>>> y(9)
42

>>> y(0)
-3

>>> y(-2)
-13
```

What if we wish to compute a y-value for a given x for a different line? One approach would be to pass in the *slope* and *intercept* as well as x:

```
def y(x,m,b):
    return m * x + b
```

Now we need to pass even more information to y when we call it:

```
>>> from line.py import *
>>> y(9,5,-3)
42

>>> y(0,5,-3)
-3
```

If we wish to calculate using a different line, we just pass in the new *slope* and *intercept* along with our value of x. This certainly works as intended, but is not the best way. One problem is that we keep on having to type in the slope and intercept even if we are computing y-values on the same line. Anytime you find yourself doing the same tedious thing over and over, be assured that someone has thought of a way to avoid that particular tedium. If so, how do we customize our function so that we only have to enter the slope and intercept once per particular line? We will explore one way for doing this. In reading further, it is not important if you understand all that is going on. What is important is that you know you can use functions to run similar code over and over again.

8.3 Creating functions on the fly (ADVANCED)

Since creating functions is hard work (lots of typing) and Computer Scientists avoid unnecessary work like the plague, somebody early on got the idea of writing a function that itself creates functions! Brilliant! We can do this for our line problem. We will tell our creative function to create a y function for a particular slope and intercept! While we are at it, let's change the variable names m and b to *slope* and *intercept*, respectively:

```
def makeLine(slope,intercept):
    def y(x):
        return slope * x + intercept
    return y    # the value of y is returned, y is NOT CALLED!
```

The *makeLine* function creates a y function and then returns it. Note that this returned function y takes one value when called, the value of x.

So our creative *makeLine* function simply defines a y function and then returns it. Now we can create a bunch of different lines:

```
>>> from line.py import *
>>> a = makeLine(5,-3)
>>> b = makeLine(6,2)

>>> a(9)
42

>>> b(9)
56

>>> a(9)
42
```

Notice how lines a and b remember the slope and intercept supplied when they were created.[2] While this is decidedly cool, the problem is many languages (C, C++, and Java included[3]) do not allow you to define functions that create other functions. Fortunately, Python does allow this.

While this might seem a little mind-boggling, don't worry. The things you should take away from this are:

- functions encapsulate calculations
- functions can be parameterized
- functions are defined so that they may be called
- functions return values

This last point is very important. Whoever calls a function needs to handle the return value either by assigning it to a variable or by passing it immediately to another function (nested function calls). Here is an example of the former:

```
y = square(x)
z = square(y)
```

and here is an example of both the former and the latter:

```
z = square(square(x))
```

Both approaches yield identical results.

[2] The local function y does not really remember these values, but for an introductory course, this is a good enough explanation.

[3] C++ and Java, as well as Python, give you another approach, *objects*. We will not go into objects in this course, but you will learn all about them in your next programming course.

8.4 The Function and Procedure Patterns

When a function calculates (or obtains) a value and returns it, we say that it implements the *function* pattern. If a function does not have a return value, we say it implements the *procedure* pattern.

Here is an example of the *function* pattern:

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

This function takes a value, stores it in x, computes the square of x and returns the result of the computation.

Here is an example of the *procedure* pattern:

```
def greeting(name):
    print("hello,",name)
    return "done"
```

We use a special return value to indicate procedures, the string `"done"`. Returning `"done"` is meant to indicate the return value should not be used.

Almost always, a function that implements the *function* pattern does not print anything, while a function that implements the procedure pattern often does[4]. A common function that implements the procedure pattern is the *main* function[5].

A mistake often made by beginning programmers is to print a calculated value rather than to return it. So, when defining a function, you should ask yourself, should I implement the function pattern or the procedure pattern?

Most of the functions you will implement in this class follow the function pattern.

Another common mistake is to inadvertently implement a *procedure* pattern when a *function* pattern is called for. This happens when the *return* keyword is omitted.

```
def psquare(x):
    x * x
```

While the above code looks like the function pattern, it is actually a procedure pattern. What happens is the value $x * x$ is calculated, but since it is not returned, the newly calculated value is thrown away (remember the *throw-away* pattern?).

Calling this kind of function yields a surprising result:

```
>>> x = psquare(4)
>>>print(x)
None
```

[4]Many times, the printing is done to a file, rather than the console.

[5]This is not strictly true. In most operating systems, a program is expected to return an error code, if the program was not able to run to completion. Thus, raising an exception (see the chapter on error checking), will almost always cause a non-zero error code to be returned to the operating system. What should a program return if no errors are encountered? A zero. You can remember this saying to your self, "zero errors". Thus, in this text, main functions will always return zero. Note that the main's return statement will not be reached if an exception occurs.

When you do not specify a return value, Python will supply one for you anyway. This supplied value is the special symbol `None`. Since *psquare* did not explicitly return a value, `None` was returned and assigned to the variable *x*. However, you should always use an explicit return.

Usually, the procedure pattern causes some side-effect to happen (like printing). A procedure like *psquare*, which has no side-effect, is a useless function.

Chapter 9

Python Programs and Using Files

After a while, it gets rather tedious to cut and paste into the Python interpreter. A more efficient method is to store your program in a text file and then load the file.

I use *vim* as my text editor. *Vim* is an editor that was written by programmers for programmers (*emacs* is another such editor) and serious Computer Scientists and Programmers should learn *vim* (or *emacs*).

9.1 Your first program

Create a text file named *hello.py*. The name really doesn't matter and doesn't have to end in *.py* (the *.py* is a convention to remind us this file contains Python source code). Place in the file:

```
print("hello, world!");
```

Save your work and exit the text editor.

Now execute the following command at the system prompt (not the Python interpreter prompt!):

```
    python3 hello.py
```

You should see the phrase:

```
    hello, world!
```

displayed on your console. Here's a trace using Linux:

```
    lusth@warka:~$ python3 hello.py
    hello, world!
    lusth@warka:~$
```

The `lusth@warka: $` is my system prompt.

9.2 Vim and Python

Move into your home directory and list the files found there with this command:

```
ls -al
```

If you see the file .exrc, then all is well and good. If you do not, then run the following command to retrieve it:

```
(cd ; wget troll.cs.ua.edu/cs150/.exrc)
```

This configures `vim` to understand Python syntax and to color various primitives and keywords in a pleasing manner.

9.3 A Neat Macro

One of the more useful things you can do is set up a *vim* macro. Edit the file *.exrc* in your home directory and find these lines:

```
map @ :!python %^M
map # :!python %
set ai sm sw=4
```

If you were unable to download the file in the previous section, just enter the lines above in the *.exrc* file.

The first line makes the `'@'` key, when pressed, run the Python interpreter on the file you are currently editing (save your work first before tapping the `@` key). The `^M` part of the macro is not a two character sequence (`^` followed by `M`), but a single character made by typing `<Ctrl>-v` followed by `<Ctrl>-m`. It's just when you type `<Ctrl>-v <Ctrl>-m`, it will display as `^M`. The second line defines a similar macro that pauses to let you enter command-line arguments to your Python program. The third line sets some useful parameters: *autoindent* and *showmatch*. The expression `sw=4` sets the indentation to four spaces.

9.4 Writing Python Programs

A typical Python program is composed of two sections. The first section is composed of variable and function definitions. The next section is composed of statements, which are Python expression. Usually the latter section is reduced to a single function call (we'll see an example in a bit).

The *hello.py* file above was a program with no definitions and a single statement. A Python program composed only of definitions will usually run with no output to the screen. Such programs are usually written for the express purpose of being included into other programs.

Typically, one of the function definitions is a function named *main* (by convention); this function takes no arguments. The last line of the program (by convention) is a call to *main*. Here is a rewrite of *hello.py* using that convention.

```
def main():
    println("hello, world!")
    return 0

main()
```

This version's output is exactly the same as the previous version. We also can see that *main* implements the *function pattern* since it has a non-`None` return value. Generally, a zero return value for *main* means that no errors were encountered. Typically, non-zero return values for *main* are used to indicate errors, with each value associated with a specific error.

9.5 Order of definitions

A function (or variable) must be created or defined[1] before it is used. This program will generate an error:

```
main()           #undefined variable error

def main():
    y = 3
    x = y * y
    print("x is",x)
    return 0
```

since *main* can't be called until it is defined. This program is legal, however:

```
def main():
    x = f(3)
    print("x is",x)
    return 0

def f(z):
    return z * z

main()
```

because even though the body of *main* refers to function *f* before function *f* is defined, function *f* is defined by the time function *main* is called (the last statement of the program).

Here is the order of steps taken when this program is run:

1. the function *main* is defined
2. the function *f* is defined
3. the function *main* is called
   (a) the statement `x = f(3)` is evaluated
   (b) the function *f* is called
      i. the formal parameter *z* is set to 3
      ii. the square of *z* is computed
      iii. the function *f* returns the value 9
   (c) x is assigned the value 9
   (d) the statement `print("x is",x)` is evaluated
   (e) the text `"x is 9"` is printed
   (f) the function *main* returns zero
4. the program ends

[1] From now on, we will use the word defined.

At this point, you should type this program into a file and then run it. If you do so, you should see the following output, as expected:

```
x is 9
```

A useful trick to understanding the flow of a program is to place *print* statements throughout the code and try to guess, before running the program, the order in which those added statements display text to the console.

9.6 Importing code

One can include one file of Python code into another. The included file is known as a module. The *import* statement is used to include a module:

```
from moduleX.py import *
```

where *moduleX.py* is the name of the file containing the Python definitions you wish to include. Usually import statements are placed at the top of the file. Including a module imports all the code in that module, as if you had written it in the file yourself.

If *moduleX.py* has import statements, those modules will be included in the file as well.

Import statements often are used to include the standard Python libraries.

Chapter 10

Input and Output

Generally all computational entities, be it variables, functions, or programs, require some sort of input and output. Input roughly corresponds to the process of supplying data and output roughly corresponds to the process of retrieving data. For a variable, assignment is the way values (data) are supplied to the variable, while referencing the variable (as part of an expression) is the way that data is retrieved. For functions, the input is the set of arguments that are supplied in a function call, while the output of a function is its return value. For programs, input refers to pulling in data and output refers to pushing out results. In this chapter, we focus on managing the input and output of programs.

10.1 Input

Input is the process of obtaining data that a program needs to perform its task. There are generally three ways for a program to obtain data:

- interactively, by querying the user for information
- via the command line, by accessing an array of command line arguments
- via a file, by opening a file and reading its contents

We examine these three approaches in turn.

10.1.1 Reading from the keyboard

To read from the keyboard, one uses the *input* function:

```
name = input("What is your name? ")
```

The *input* function prints the given message to the console and waits until a response is typed. In the example above, the message is `"What is your name? "`; this message is known as a *prompt*. The *input* function (as of Python3) always returns a string. If you wish to read an *integer*, you can wrap the call to *input* in a call to *int*:

```
age = int(input("How old are you? "))
```

Other conversion functions similar to *int* are *float*, which converts the string *input* returns to a real number, and *eval*, which converts a string into its Python equivalent. For example, we could substitute *eval* for *int* or

float and we would get the exact same result, *provided* an integer or a real number, respectively, were typed in response to *input* prompt. The *eval* function can even to some math for you:

```
>>> eval("3 + 7")
10
```

10.1.2 Reading From the command line

The second way to pass information to a program is through *command-line arguments*. The command line is the line typed in a terminal window that runs a python program (or any other program). Here is a typical command line on a Linux system:

```
lusth@sprite:~/l1/activities$ python3 prog3.py
```

Everything up to and including the dollar sign is the system prompt. As with all prompts, it is used to signify that the system is waiting for input. The user of the system (me) has typed in the command:

```
python3 prog3.py
```

in response to the prompt. Suppose *prog3.py* is a file with the following code:

```
import sys

def main():
    print("command-line arguments:")
    print("    ",sys.argv)
    return 0

main()
```

In this case, the output of this program would be:

```
command-line arguments:
    ['prog3.py']
```

Note that the program imports the *sys* module and when the value of the variable *sys.argv* is printed, we see its value is:

```
['prog3.py']
```

This tells us that *sys.argv* points to an array (because of the square brackets) and that the program file name, as a string, is found in this array.

Any whitespace-delimited tokens following the program file name are stored in *sys.argv* along with the name of the program being run by the Python interpreter. For example, suppose we run *prog3.py* with the this command:

```
python3 prog3.py 123 123.4 True hello, world
```

Then the output would be:

```
command-line arguments:
    ['prog3.py', '123', '123.4', 'True', 'hello,', 'world']
```

From this result, we can see that all of the tokens are stored in *sys.argv* and that they are stored as strings, *regardless* of whether they look like some other entity, such as integer, real number, or Boolean.

If we wish for `"hello, world"` to be a single token, we would need to enclose the tokens in quotes:

```
python3 prog3.py 123 123.4 True "hello, world"
```

In this case, the output is:

```
command-line arguments:
    ['prog3.py', '123', '123.4', 'True', 'hello, world']
```

There are certain characters that have special meaning to the system. A couple of these are '*' and ';'. To include these characters in a command-line argument, they need to be *escaped* by inserting a backslash prior to the character. Here is an example:

```
python3 prog.py \; \* \\
```

To insert a backslash, one escapes it with a backslash. The output from this command is:

```
command-line arguments:
    ['prog3.py', ';', '*', '\\']
```

Although it looks as if there are two backslashes in the last token, there is but a single backslash. Python uses two backslashes to indicate a single backslash.

Counting the command line arguments

The number of command-line arguments (including the program file name) can be found by using the *len* (length) function. If we modify *prog3.py*'s main function to be:

```
def main():
    print("command-line argument count:",len(sys.argv))
    print("command-line arguments:")
    print("    ",sys.argv)
    return 0
```

and enter the following command at the system prompt:

```
python3 prog3.py 123 123.4 True hello world
```

we get this output:

```
command-line argument count: 6
command-line arguments:
     ['prog3.py', '123', '123.4', 'True', 'hello', 'world']
```

As expected, we see that there are six command-line arguments, including the program file name.

Accessing individual arguments

As with most programming languages and with Python arrays, the individual tokens in *sys.argv* are accessed with zero-based indexing. To access the program file name, we would use the expression:

```
sys.argv[0]
```

To access the first command-line argument after the program file name, we would use the expression:

```
sys.argv[1]
```

Let's now modify *prog3.py* so that it prints out each argument individually. We will using a construct called a loop to do this. You will learn about looping later, but for now, the statement starting with *for* generates all the legal indices for *sys.argv*, from zero to the length of *sys.argv* minus one. Each index, stored in the variable *i*, is then used to print the argument stored at that location in *sys.argv*:

```
def main():
    print("command-line argument count:",len(sys.argv))
    print("command-line arguments:")
    for i in range(0,len(sys.argv),1):
        print("   ",i,":",sys.argv[i])
    return 0;
```

Given this command:

```
python3 prog3.py 123 123.4 True hello world
```

we get the following output:

```
command-line argument count: 6
command-line arguments:
    0 : prog3.py
    1 : 123
    2 : 123.4
    3 : True
    4 : hello
    5 : world
```

This code works no matter how many command line arguments are sent. The superior student will ascertain that this is true.

What command-line arguments are

The command line arguments are stored as strings. Therefore, you must use the *int*, *float*, or *eval* functions if you wish to use any of the command line arguments to integers, real numbers or Booleans, as examples.

10.1.3 Reading from files

The third way to get data to a program is to read the data that has been previously stored in a file.

Python uses a *file pointer* system in reading from a file. To read from a file, the first step is to obtain a pointer to the file. This is known as *opening* a file. The file pointer will always point to the first unread character in a file. When a file is first opened, the file pointer points to the first character in the file.

Reading files using file pointer methods

A Python file pointer has a number of associated functions for reading in parts or all of a file. Suppose we wish to read from a file named *data*. We first obtain a file pointer by opening the file like this:

```
fp = open("data","r")
```

The *open* function takes two arguments, the name of the file and the kind of file pointer to return. We store the file pointer in a variable named *fp* (a variable name commonly used to hold a file pointer). In this case, we wish for a *reading* file pointer, so we pass the string `"r"`. We can also open a file for writing; more on that in the next section.

Next, we can read the entire file into a single string with the *read* method:

```
text = fp.read()
```

After this statement is evaluated, the variable *text* would point to a string containing every character in the file *data*. We call *read* a method, rather than a function (which it is), to indicate that it is a function that belongs to a file pointer object, which *fp* is. You will learn about objects in a later class, but the "dot" operator is a clue that the thing to the left of the dot is an object and the thing to the right is a method (or simple variable) that belongs to the object.

When we are done reading a file, we *close* it:

```
fp.close()
```

Instead of reading all the file in at once using the *read* method, it is sometimes useful to read a file one line at a time. One uses the *readline* method for this task, but the *readline* method is not very useful until we learn about loops in a later chapter.

Using a scanner

A scanner is a reading subsystem that allows you to read whitespace-delimited tokens from a file. Whitespace are those characters in a text file that are generally not visible: spaces, tabs, and newlines. For example, this sentence:

```
I am a sentence!
```

is composed of four whitespace delimited tokens: `I`, `am`, `a`, and `sentence!`.

Typically, a scanner is used just like a file pointer, but is far more flexible. Suppose we wish to read from a file named *data*. We would open the file with a scanner like this:

```
s = Scanner("data")
```

The *Scanner* function takes a single argument, the name of the file, and returns a scanner object, which can be used to read the tokens in the file.

Suppose the file *data* contained:

```
True 3 -4.4
"hello there"
ack!
```

The file contains six whitespace-delimited tokens: `True`, `3`, `-4.4`, `"hello`, followed by `there"`, and `ack!`. We can use the scanner to read each of the tokens using the *readtoken* method.

```
from scanner import *

def main():
    s = Scanner("data")
    b = s.readtoken()
    i = s.readtoken()
    f = s.readtoken()
    str1 = s.readtoken()
    str2 = s.readtoken()
    t = s.readtoken()
    print("The type of",b,"is",type(b))
    print("The type of",i,"is",type(i))
    print("The type of",f,"is",type(f))
    print("The type of",str1,"is",type(str1))
    print("The type of",str2,"is",type(str2))
    print("The type of",t,"is",type(t))
    s.close()
    return 0;

main()
```

To run this program, you will first need to get a scanner for Python. You can obtain a scanner by issuing this command:

```
wget troll.cs.ua.edu/cs150/book/scanner.py
```

The *scanner.py* file needs to reside in the same directory as the program that imports it.

The program, as written, runs fine, yielding the following output:

```
    The type of True is <class 'str'>
    The type of 3 is <class 'str'>
    The type of -4.4 is <class 'str'>
    The type of "hello is <class 'str'>
    The type of there" is <class 'str'>
    The type of ack! is <class 'str'>
```

The *type* function tells us what kind of literal is passed to it. We can see that every token is read in as a string, from the `<class 'str'>` return value from the type function.

Here is a revised version that takes advantage of the full power of the scanner; the program reads in the objects as they appear, a Boolean, integer, float, string, and token:

```
from scanner import *

def main():
    s = Scanner("data")
    b = s.readbool()
    i = s.readint()
    f = s.readfloat()
    str = s.readstring()
    t = s.readtoken()
    print("The type of",b,"is",type(b))
    print("The type of",i,"is",type(i))
    print("The type of",f,"is",type(f))
    print("The type of",str,"is",type(str))
    print("The type of",t,"is",type(t))
    s.close()
    return 0;

main()
```

The output of the revised program is:

```
    The type of True is <class 'bool'>
    The type of 3 is <class 'int'>
    The type of -4.4 is <class 'float'>
    The type of "hello there" is <class 'str'>
    The type of ack! is <class 'str'>
```

The methods *readbool*, *readint*, *readfloat* convert the tokens they read into the appropriate type. Thus, *readbool* returns a Boolean, not a string, *readint* returns an integer, and so on. The *readstring* method will read a string delimited by double quotes as a single token. Note that the double quotes are considered part of the string; to remove them, one can use the following technique:

```
    str = s.readstring()
    str = str[1:-1]         # a slice, all but the first and last characters
```

If any of the reading methods fail (i.e., trying to read an integer when there is no integer at that point), the read methods return the empty string. As a final note, always remember to close a scanner when you are done with it, as in:

```
s.close()
```

10.2 Output

Once a program has processed its input, it needs to make its output known, either by displaying results to the user or by storing the results in a file.

10.2.1 Writing to the console

One uses the *print* function to display text on the console, for benefit of the user of the program. The *print* function is *variadic*, which means it can take a variable number of arguments. The *print* function has lots of options, but we will be interested in only two, *sep* and *end*. The *sep* (for separator) option specifies what is printed between the arguments sent to be printed. If the *sep* option is missing, a space is printed between the values received by *print*:

```
>>> print(1,2,3)
1 2 3
>>>
```

If we wish to use commas as the separator, we would do this:

```
>>> print(1,2,3,sep=",")
1,2,3
>>>
```

If we wish to have no separator, we bind *sep* to an empty string:

```
>>> print(1,2,3,sep="")
123
>>>
```

The *end* option specifies what be printed after the arguments are printed. If you don't supply a *end* option, a newline is printed by default. This call to *print* prints an exclamation point and then a newline at the end:

```
>>> print(1,2,3,end="!\n")
1 2 3!
>>>
```

If you don't want anything printed at the end, bind *end* to an empty string:

```
>>> print(1,2,3,end="")
1 2 3>>>
```

Notice how the Python prompt ends up on the same line as the values printed.

You can combine the *sep* and *end* options.

Printing quote characters

Suppose I have the string:

```
str = "Hello"
```

If I print my string:

```
print(str)
```

The output looks like this:

```
Hello
```

Notice the double quotes are not printed. But suppose I wish to print quotes around my string, so that the output looks like:

```
"Hello"
```

To do this, the print statement becomes:

```
print("\"",str,"\"",sep="")
```

If you need a refresher on what the string `"\""` means, please see Chapters 2.3. The superior student will ponder on the necessity of `sep=""`.

10.3 Writing to a file

Python also requires a file pointer to write to a file. The *open* function is again used to obtain a file pointer, but this time we desire a *writing* file pointer, so we send the string `"w"` as the second argument to *open*:

```
fp = open("data.save","w")
```

Now the variable *fp* points to a writing file object, which has a method named *write*. The only argument *write* takes is a string. That string is then written to the file. Here is a function that copies the text from one file into another:

```
def copyFile(inFile,outFile):
    in = open(inFile,"r")
    out = open(outFile,"w")
    text = in.read();
    out.write(text);
    in.close()
    out.close()
    return "done"
```

Opening a file in order to write to it has the effect of emptying the file of its contents soon as it is opened. The following code deletes the *contents* of a file (which is different than deleting the file):

```
# delete the contents
fp = open(fileName,"w")
fp.close()
```

We can get rid of the variable *fp* by simply treating the call to *open* as the object *open* returns, as in:

```
# delete the contents
open(fileName,"w").close()
```

If you wish to start writing to a file, but save what was there previously, call the *open* function to obtain an *appending* file pointer:

```
fp = open(fileName,"a")
```

Subsequent writes to *fp* will append text to what is already there.

Chapter 11

More about Functions

We have already seen some examples of functions, some user-defined and some built-in. For example, we have used the built-in functions, such as * and defined our own functions, such as *square*. In reality, *square* is not a function, per se, but a variable that is bound to the function that multiplies two numbers together. It is tedious to say "the function bound to the variable *square*", however, so we say the more concise (but technically incorrect) phrase "the *square* function".

11.1 Built-in Functions

Python has many built-in, or *predefined*, functions. No one, however, can anticipate all possible tasks that someone might want to perform, so most programming languages allow the user to define new functions. Python is no exception and provides for the creation of new and novel functions. Of course, to be useful, these functions should be able to call built-in functions as well as other programmer created functions.

For example, a function that determines whether a given number is odd or even is not built into Python but can be quite useful in certain situations. Here is a definition of a function named *isEven* which returns true if the given number is even, false otherwise:

```
>>> def isEven(n):
...     return n % 2 == 0
...
>>>

>>> isEven(42)
True

>>> isEven(3)
False

>>> isEven(3 + 5)
True
```

We could spend days talking about about what's going on in these interactions with the interpreter. First, let's talk about the syntax of a function definition. Later, we'll talk about the purpose of a function definition. Finally, will talk about the mechanics of a function definition and a function call.

11.2 Function syntax

Recall that the words of a programming language include its primitives, keywords and variables. A function definition corresponds to a sentence in the language in that it is built up from the words of the language. And like human languages, the sentences must follow a certain form. This specification of the form of a sentence is known as its *syntax*. Computer Scientists often use a special way of describing syntax of a programming language called the Backus-Naur form (or BNF). Here is a high-level description of the syntax of a Python function definition using BNF:

```
functionDefinition : signature ':' body

signature : 'def' variable '(' optionalParameterList ')'

body : block

optionalParameterList : *EMPTY*
                      | parameterList

parameterList : variable
              | variable ',' parameterList

block:  definition
      | definition block
      | statement
      | statement block
```

The first BNF *rule* says that a function definition is composed of two pieces, a signature and a body, separated by the colon character (parts of the rule that appear verbatim appear within single quotes). The signature starts with the keyword *def* followed by a variable, followed by an open parenthesis, followed by something called an *optionalParameterList*, and finally followed by a close parenthesis. The body of a function something called a *block*, which is composed of *definitions* and *statements*. The *optionalParameterList* rule tells us that the list of formal parameters can possibly be empty, but if not, is composed of a list of variables separated by commas.

As we can see from the BNF rules, parameters are variables that will be bound to the values supplied in the function call. In the particular case of *isEven*, from the previous section, the variable x will be bound to the number whose evenness is to be determined. As noted earlier, it is customary to call x a *formal parameter* of the function *isEven*. In function calls, the values to be bound to the formal parameters are called *arguments*.

11.3 Function Objects

Let's look more closely at the body of *isEven*:

```
def isEven(x):
    return x % 2 == 0
```

The % operator is bound to the remainder or modulus function. The == operator is bound to the equality function and determines whether the value of the left operand expression is equal to the value of the right operand expression, yielding true or false as appropriate. The BOOLEAN value produced by == is then immediately returned as the value of the function.

When given a function definition like that above, Python performs a couple of tasks. The first is to create the internal form of the function, known as a *function object*, which holds the function's signature and body. The second task is to bind the function name to the function object so that it can be called at a later time. Thus, the name of the function is simply a variable that happens to be bound to a function object. As noted before, we often say 'the function *isEven*' even though we really mean 'the function object bound to the variable *even*?'.

The value of a function definition is the function object; you can see this by printing out the value of *isEven*:

```
>>> print(isEven)
<function isEven at 0x9cbf2ac>

>>> isEven = 4
>>> print(isEven)
4
```

Further interactions with the interpreter provide evidence that *isEven* is indeed a variable; we can reassign its value, even though it is considered in poor form to do so.

11.4 Calling Functions

Once a function is created, it is used by *calling* the function with *arguments*. A function is called by supplying the name of the function followed by a parenthesized, comma separated, list of expressions. The arguments are the values that the formal parameters will receive. In Computer Science speak, we say that the values of the arguments are to be bound to the formal parameters. In general, if there are n formal parameters, there should be n arguments.[1] Furthermore, the value of the first argument is bound to the first formal parameter, the second argument is bound to the second formal parameter, and so on. Moreover, all the arguments are evaluated before being bound to any of the parameters.

Once the evaluated arguments are bound to the parameters, then the body of the function is evaluated. Most times, the expressions in the body of the function will reference the parameters. If so, how does the interpreter find the values of those parameters? That question is answered in the next chapter.

11.5 Returning from functions

The return value of a function is the value of the expression following the `return` keyword. For a function to return this expression, however, the return has to be *reached*. Look at this example:

```
def test(x,y):
    if (y == 0):
        return 0
    else:
        print("good value for y!")
        return x / y

    print("What?")
    return 1
```

[1] For *variadic* functions, which Python allows for, the number of arguments may be more or less than the number of formal parameters.

Note that the == operator returns true if the two operands have the same value. In the function, if y is zero, then the

```
return 0
```

statement is reached. This causes an immediate return from the function and no other expressions in the function body are evaluated. The return value, in this case, is zero. If y is not equal to zero, a message is printed and the second return is reached, again causing an immediate return. In this case, a quotient is returned.

Since both parts of the if statement have returns, then the last two lines of the function:

```
print("What?")
return 1
```

are *unreachable*. Since they are unreachable, they cannot be executed under any conditions and thus serve no purpose and can be deleted.

Chapter 12

Scope

A *scope* holds the current set of variables and their values. In Python, there is something called the *global scope*. The global scope holds all the values of the built-in variables and functions (remember, a function name is just a variable).

When you enter the Python interpreter, either by running it interactively or by using it to evaluate a program in a file, you start out in the global scope. As you define variables, they and their values are added to the global scope.

This interaction adds the variable x to the global scope:

```
>>> x = 3
```

This interaction adds two more variables to the global scope:

```
>>> y = 4
>>> def negate(z):
...     return -z;
...
>>>
```

What are the two variables? The two variables added to the global scope are y and *negate*.

Indeed, since the name of a function is a variable and the variable *negate* is being bound to a function object, it becomes clear that this binding is occurring in the global scope, just like y being bound to 4.

Scopes in Python can be identified by their indentation level. The global scope holds all variables defined with an indentation level of zero. Recall that when functions are defined, the body of the function is indented. This implies that variables defined in the function body belong to a different scope and this is indeed the case. Thus we can identify to which scope a variable belongs by looking at the pattern of indentations. In particular, we can label variables as either *local* or *non-local* with respect to a particular scope. Moreover, non-local variables may be *in scope* or or *out of scope*.

12.1 In Scope or Out

The indentation pattern of a program can tells us where variables are visible (in scope) and where they are not (out of scope). We begin by first learning to recognizing the scopes in which variables are defined.

12.1.1 The Local Variable Pattern

All variables *defined* at a particular indentation level or scope are considered *local* to that indentation level or scope. In Python, if one assigns a value to a variable, that variable must be local to that scope. The only exception is if the variable was explicitly declared *global* (more on that later). Moreover, the formal parameters of a function definition belong to the scope that is identified with the function body. So within a function body, the local variables are the formal parameters plus any variables defined in the function body.

Let's look at an example. Note, you do not need to completely understand the examples presented in the rest of the chapter in order to identify the local and non-local variables.

```
def f(a,b):
    c = a + b
    c = g(c) + X
    d = c * c + a
    return d * b
```

In this example, we can immediately say the formal parameters, a and b, are local with respect to the scope of the body of function f. Furthermore, variables c and d are defined in the function body so they are local as well, with respect to the scope of the body of function f. It is rather wordy to say "local with respect to the scope of the body of the function f", so Computer Scientists will almost always shorten this to "local with respect to f" or just "local" if it is clear the discussion is about a particular function or scope. We will use this shortened phrasing from here on out. Thus a, b, c, and d are local with respect to f. The variable f is local to the global scope since the function f is defined in the global scope.

12.1.2 The Non-local Variable Pattern

In the previous section, we determined the local variables of the function. By the process of elimination, that means the variables g, and X are non-local. The name of function itself is non-local with respect to its body, f is non-local as well.

Another way of making this determination is that neither `g` nor `X` are assigned values in the function body. Therefore, they must be non-local. In addition, should a variable be explicitly declared *global*, it is non-local even if it is assigned a value. Here again is an example:

```
def h(a,b):
    global c
    c = a + b
    c = g(c) + X
    d = c * c + a
    return d * b
```

In this example, variables a, b, and d are local with respect to h while c, g, and X are non-local. Even though c is assigned a value, the declaration:

```
    global c
```

means that c belongs to a different scope (the global scope) and thus is non-local.

12.1.3 The Accessible Variable Pattern

A variable is accessible with respect to a particular scope if it is *in scope*. A variable is in scope if it is local or was defined in a scope that *encloses* the particular scope. Some scope A encloses some other scope B if, by moving (perhaps repeatedly) leftward from scope B, scope A can be reached. Here is example:

```
Z = 5

def f(x):
   return x + Z

print(f(3))
```

The variable Z is local with respect to the global scope and is non-local with respect to f. However, we can move leftward from the scope of f one indentation level and reach the global scope where Z is defined. Therefore, the global scope encloses the scope of f and thus Z is accessible from f. Indeed, the global scope encloses all other scopes and this is why the built-in functions are accessible at any indentation level.

Here is another example that has two enclosing scopes:

```
X = 3
def g(a)
   def m(b)
      return a + b + X + Y
   Y = 4
   return m(a % 2)

print(g(5))
```

If we look at function m, we see that there is only one local variable, b, and that m references three non-local variables, a, X, and Y. Are these non-local variables accessible? Moving leftward from the body of m, we reach the body of g, so the scope of g encloses the scope of m. The local variables of g are a, m, and Y, so both a and Y are accessible in the scope of m. If we move leftward again, we reach the global scope, so the global scope encloses the scope of g, which in turn encloses the scope of m. By transitivity, the global scope encloses the scope of m, so X, which is defined in the global scope is accessible to the scope of m. So, all the non-locals of m are accessible to m.

In the next section, we explore how a variable can be inaccessible.

12.1.4 The Tinted Windows Pattern

The scope of local variables is like a car with tinted windows, with the variables defined within riding in the back seat. If you are outside the scope, you cannot peer through the car windows and see those variables. You might try and buy some x-ray glasses, but they probably wouldn't work. Here is an example:

```
    z = 3

    def f(a):
        c = a + g(a)
        return c * c
```

```
print("the value of a is",a) #x-ray!
f(z);
```

The print statement causes an error:

```
Traceback (most recent call last):
  File "xray.py", line 7, in <module>
      print("the value of a is",a) #x-ray!
      NameError: name 'a' is not defined
```

If we also tried to print the value of *c*, which is a local variable of function *f*, at that same point in the program, we would get a similar error.

The rule for figuring out which variables are in scope and which are not is: *you* **cannot** *see into an enclosed scope.* Contrast this with the non-local pattern: *you* **can** *see variables declared in enclosing outer scopes.*

12.1.5 Tinted Windows with Parallel Scopes

The tinted windows pattern also applies to parallel scopes. Consider this code:

```
z = 3

def f(a):
    return a + g(a)

def g(x):
    # starting point 1
    print("the value of a is",a) #x-ray!
    return x + 1

f(z);
```

Note that the global scope encloses both the scope of *f* and the scope of *g*. However, the scope of *f* does not enclose the scope of *g*. Neither does the scope of *g* enclose the scope of *f*.

One of these functions references a variable that is not in scope. Can you guess which one? The function *g* references a variable not in scope.

Let's see why by first examining *f* to see whether or not its non-local references are in scope. The only local variable of function *f* is *a*. The only referenced non-local is *g*. Moving leftward from the body of *f*, we reach the global scope where where both *f* and *g* are defined. Therefore, *g* is visible with respect to *f* since it is defined in a scope (the global scope) that encloses *f*.

Now to investigate *g*. The only local variable of *g* is *x* and the only non-local that *g* references is *a*. Moving outward to the global scope, we see that there is no variable *a* defined there, therefore the variable *a* is not in scope with respect to *g*.

When we actually run the code, we get an error similar to the following when running this program:

```
Traceback (most recent call last):
  File "xray.py", line 11, in <module>
```

```
    f(z);
  File "xray.py", line 4, in f
    return a + g(a)
  File "xray.py", line 8, in g
    print("the value of a is",a) #x-ray!
NameError: global name 'a' is not defined
```

The lesson to be learned here is that we cannot see into the local scope of the body of function *f*, *even if we are at a similar nesting level*. Nesting level doesn't matter. We can only see variables in our own scope and those in *enclosing* scopes. All other variables cannot be seen.

Therefore, if you ever see a variable-not-defined error, you either have spelled the variable name wrong, you haven't yet created the variable, or you are trying to use x-ray vision to see somewhere you can't.

12.2 Alternate terminology

Sometimes, enclosed scopes are referred to as *inner* scopes while enclosing scopes are referred to as *outer* scopes. In addition, both locals and any non-locals found in enclosing scopes are considered *visible* or *in scope*, while non-locals that are not found in an enclosing scope are considered *out of scope*. We will use all these terms in the remainder of the text book.

12.3 Three Scope Rules

Here are three simple rules you can use to help you figure out the scope of a particular variable:

- Formal parameters belong in
- The function name belongs out
- You can see out but you can't see in (tinted windows).

The first rule is shorthand for the fact that formal parameters belong to the scope of the function body. Since the function body is "inside" the function definition, we can say the formal parameters belong in.

The second rule reminds us that as we move outward from a function body, we find the enclosing scope holds the function definition. That is to say, the function name is bound to a function object in the scope enclosing the function body.

The third rule tells us all the variables that belong to ever-enclosing scopes are accessible and therefore can be referenced by the innermost scope. The opposite is not true. A variable in an enclosed scope can not be referenced by an enclosing scope. If you forget the directions of this rule, think of tinted windows. You can see out of a tinted window, but you can't see in.

12.4 Shadowing

The formal parameters of a function can be thought of as variable definitions that are only in effect when the body of the function is being evaluated. That is, those variables are only visible in the body and no where else. This is why formal parameters are considered to be *local* variable definitions, since they only have local effect (with the locality being the function body). Any direct reference to those particular variables outside the body of the function is not allowed (Recall that you can't see in). Consider the following interaction with the interpreter:

```
>>> def square(a):
...     return a * a
...
>>>

>>> square(4)
16

>>> a
NameError: name 'a' is not defined
```

In the above example, the scope of variable *a* is restricted to the body of the function *square*. Any reference to *a* other than in the context of *square* is invalid. Now consider a slightly different interaction with the interpreter:

```
>>> a = 10
>>> def almostSquare(a):
...     return a * a + b
...
>>> b = 1

>>> almostSquare(4)
17

>>> a
10
```

In this dialog, the global scope has three variables added, *a*, *almostSquare* and *b*. In addition, the variable serving as the formal parameter of *almostSquare* has the same name as the first variable defined in the dialog. Moreover, the body of *almostSquare* refers to both variables *a* and *b*. To which *a* does the body of almostSquare refer? The global *a* or the local *a*? Although it seems confusing at first, the Python interpreter has no difficulty in figuring out what's what. From the responses of the interpreter, the *b* in the body must refer to the variable that was defined with an initial value of one. This is consistent with our thinking, since *b* belongs to the enclosing scope and is accessible within the body of *almostSquare*. The *a* in the function body must refer to the formal parameter whose value was set to 4 by the call to the function (given the output of the interpreter).

When a local variable has the same name as a non-local variable that is also in scope, the local variable is said to *shadow* the non-local version. The term shadowed refers to the fact that the other variable is in the shadow of the local variable and cannot be seen. Thus, when the interpreter needs the value of the variable, the value of the local variable is retrieved. It is also possible for a non-local variable to shadow another non-local variable. In this case, the variable in the nearest outer scope shadows the variable in the scope further away.

In general, when a variable is referenced, Python first looks in the local scope. If the variable is not found there, Python looks in the enclosing scope. If the variable is not there, it looks in the scope enclosing the enclosing scope, and so on.

In the particular example, a reference to *a* is made when the body of *almostSquare* is executed. The value of *a* is immediately found in the local scope. When the value of *b* is required, it is not found in the local scope. The interpreter then searches the enclosing scope (which in this case happens to be the global scope). The global scope does hold *b* and its value, so a value of 1 is retrieved.

Since *a* has a value of 4 and *b* has a value of 1, the value of 17 is returned by the function. Finally, the last interaction with the interpreter illustrates the fact that the initial binding of *a* was unaffected by the function call.

12.5 Modules

Often, we wish to use code that has already been written. Usually, such code contains handy functions that have utility for many different projects. In Python, such collections of functions are known as modules. We can include modules into our current project with the *import* statement, which we saw in Chapters 8 and 9.

The import statement has two forms. The first is:

```
from ModuleX import *
```

This statement imports all the definitions from ModuleX and places them into the global scope. At this point, those definitions look the same as the built-ins, but if any of those definitions have the same name as a built-in, the built-in is shadowed.

The second form looks like this:

```
import ModuleX
```

This creates a new scope that is separate from the global scope (but is enclosed by the global scope). Suppose *ModuleX* has a definition for variable *a*, with a value of 1. Since *a* is in a scope enclosed by the global scope, it is inaccessible from the global scope (you can't see in):

```
>>> import ModuleX
>>> a
NameError: name 'a' is not defined
```

The direct reference to *a* failed, as expected. However, one can get to *a* and its value *indirectly*:

```
>>> import ModuleX
>>> ModuleX . a
1
```

This new notation is known as *dot* notation and is commonly used in object-oriented programming systems to references pieces of an object. For our purposes, *ModuleX* can be thought of as a *named* scope and the *dot* operator is used to look up variable *a* in the scope named ModuleX.

This second form of import is used when the possibility that some of your functions or variables have the same name as those in the included module. Here is an example:

```
>>> a = 0
>>> from ModuleX import *

>>> a
1
```

Note that the *ModuleX*'s variable *a* has shadowed the previously defined *a*. With the second form of import, the two versions of *a* can each be referenced:

```
>>> a = 0
>>> import ModuleX

>>> a
0
>>> ModuleX . a
1
```

Chapter 13

Loops

Loops are used to repeatedly execute some code. The most basic loop structure in Python is the *while* loop, an example of which is:

```
i = 0
while (i < 10):
    print(i,end="")
    i = i + 1
```

We see a `while` loop looks much like an `if` statement. The difference is that blocks belonging to `if`s are evaluated at most once whereas blocks associated with loops may be evaluated many many times. Another difference in nomenclature is that the block of a loop is known as the *body* (like blocks associated with function definitions). Furthermore, the loop test expression is known as the *loop condition*.

As Computer Scientists hate to type extra characters if they can help it, you will often see:

```
i = i + 1
```

written as

```
i += 1
```

The latter version is read as "increment i" and saves a whopping two characters of typing.

A *while* loop tests its condition before the body of the loop is executed. If the initial test fails, the body is not executed at all. For example:

```
i = 10
while (i < 10):
    print(i,end="")
    i += 1
```

never prints out anything since the test immediately fails. In this example, however:

```
i = 0;
```

```
while (i < 10):
    print(i,end="")
    i += 1
```

the loop prints out the digits 0 through 9:

```
0123456789
```

A `while` loop repeatedly evaluates its body as long as the loop condition remains true.

To write an infinite loop, use `True` as the condition:

```
while (True):
    i = getInput()
    print("input is",i)
    process(i)
```

13.1 Other loops

There are many kinds of loops in Python, in this text we will only refer to `while` loops and `for` loops that count, as these are commonly found in other programming languages. The `while` loop we have seen; here is an example of a counting `for` loop:

```
for i in range(0,10,1):
    print(i)
```

This loop is exactly equivalent to:

```
i = 0
while (i < 10):
    print(i)
    i += 1
```

In fact, a while loop of the general form:

```
i = INIT
while (i < LIMIT):
    # body
    ...
    i += STEP
```

can be written as a counting `for` loop:

```
for i in range(INIT,LIMIT,STEP):
    # body
    ...
```

The *range* function counts from `INIT` to `LIMIT` (non-inclusive) by `STEP` and these values are assigned to i, in turn. After each assignment to i, the loop body is evaluated. After the last value is assigned to i and the loop body evaluated on last time, the `for` loop ends.

In Python, the *range* function assumes 1 for the step if the step is omitted and assumes 0 for the initial value and 1 for the step if both the initial value and step are omitted. However, in this text, we will always give the initial value and step of the `for` loop explicitly.

For loops are commonly used to sweep through each element of an array:

```
for i in range(0,len(items),1):
    print(items[i])
```

Recall the items in an array of n elements are located at indices 0 through $n-1$. These are exactly the values produced by the *range* function. So, this loop accesses each element, by its index, in turn, and thus prints out each element, in turn. Since using an index of n in an array of n items produces an error, the *range* function conveniently makes its given limit non-inclusive.

As stated earlier, there are other kinds of loops in Python, some of which, at times, are more convenient to use than a `while` loop or a counting `for` loop. However, anything that can be done with those other loops can be done with the loops presented here. Loops and arrays go very well together; the next sections detail some common loop patterns involving arrays.

13.2 The *counting* pattern

The counting pattern is used to count the number of items in a collection. Note that the built-in function *len* already does this for Python arrays, so counting the elements in an array is a bit of a waste of time. Still, it is about the easiest loop to write, so let's forge ahead[1]:

```
count = 0
for i in range(0,len(items),1):
     count += 1
```

When the loop finishes, the variable *count* holds the number of items in the array. In general, the counting pattern increments a counter every time the loop body is evaluated.

13.3 The *filtered-count* pattern

A variation on the counting pattern involves filtering. When *filtering*, we use an `if` statement to decide whether we should count an item or not. Suppose we wish to count the number of even items in an array:

```
count = 0
for i in range(0,len(items),1):
    if (items[i] % 2 == 0):
        count += 1
```

When this loop terminates, the variable *count* will hold the number of even integers in the array of items. The `if` makes sure the count is incremented only when the item of interest is even.

[1] We will see the utility of the counting pattern for *lists*, for which there is no equivalent *len* function, in a future chapter.

13.4 The *accumulate* pattern

Similar to the counting pattern, the *accumulate* pattern updates a variable, not by increasing its value by one, but by the value of an item. This loop, sums all the values in an array:

```
total = 0
for i in range(0,len(items),1):
    total += items[i]
```

By convention, the variable *total* is used to accumulate the item values. When accumulating a sum, the total is initialized to zero. When accumulating a product, the total is initialized to one[2].

13.5 The *filtered-accumulate* pattern

Similar to the *accumulate* pattern, the *filtered-accumulate* pattern updates a variable only if some test is passed. This function sums all the even values in a given array, returning the final sum:

```
def sumEvens(items):
    total = 0
    for i in range(0,len(items),1):
        if (items[i] % 2 == 0):
            total += items[i]
    return total
```

As before, the variable *total* is used to accumulate the item values. As with a regular accumulating, *total* is initialized to zero when accumulating a sum. The initialization value is one when accumulating a product and the initialization value is the empty array when accumulating an array (see *filtering* below).

13.6 The *search* pattern

The *search* pattern is a slight variation of *filtered-counting*. Suppose we wish to see if a value is present in an array. We can use a filtered-counting approach; if the count is greater than zero, we know that the item was indeed in the array.

```
def find(target,items): # is target in items?
    return occurrences(target,items) > 0

def occurrences(target,items):
    count = 0
    for i in range(0,len(items),1):
        if (items[i] == target):
            count = count + 1
    return count
```

In this case, *occurrences* implements the filtered-count pattern and helps *find* do its job. We designate such functions as *occurrences* and the like, *helper functions*. However, we can improve the efficiency of *find* by having it partially implement the filtered-count, but short-circuiting the search once the target item is found. We do this by returning from the body of the loop:

[2]The superior student will ascertain why this is so.

```
def find(target,items):
    for i in range(0,len(items),1):
        if (items[i] == target):
            return True             # short-circuit!
    return False
```

When the array is empty, we return false, as indicated by the last line of the function, because the loop never runs and thus the return inside the loop can never be reached. In the same vein, if no item matches the target, we cannot return true, and eventually the loop exits and false is returned.

As a beginning programmer, however, you should be wary of returning from the body of a loop. The reason is most beginners end up defining the function this way instead:

```
def find(target,items):
    for i in range(0,len(items),1):
        if (items[i] == target):
            return True
        else:
            return False
```

The behavior of this latter version of *find* is incorrect, but unfortunately, it appears to work correctly under some conditions. If you cannot figure out why this version fails under some conditions and appears to succeed under others, you most definitely should stay away from placing returns in loop bodies.

13.7 The *extreme* pattern

Often, we wish to find the largest or smallest value in an array. Here is one approach, which assumes that the first item is the largest and then corrects that assumption if need be:

```
largest = items[0]
for i in range(0,len(items),1):
    if (items[i] > largest):
        largest = items[i]
```

When this loop terminates, the variable *largest* holds the largest value. We can improve the loop slightly by noting that the first time the loop body evaluates, we compare the putative largest value against itself, which is a worthless endeavor. To fix this, we can start the index variable i at 1 instead:

```
largest = items[0]
for i in range(1,len(items),1): #start comparing at index 1
    if (items[i] > largest):
        largest = items[i]
```

Novice programmers often make the mistake of initialing setting *largest* to zero and then comparing all values against *largest*, as in:

```
largest = 0
for i in range(0,len(items),1):
    if (items[i] > largest):
        largest = items[i]
```

This code appears to work in some cases, namely if the largest value in the array is greater than or equal to zero. If not, as is the case when all values in the array are negative, the code produces an erroneous result of zero as the largest value.

13.8 The *extreme-index* pattern

Sometimes, we wish to find the index of the most extreme value in an array rather than the actual extreme value. In such cases, we assume index zero holds the extreme value:

```
ilargest = 0
for i in range(1,len(items),1):
    if (items[i] > items[ilargest]):
        ilargest = i
```

Here, we successively store the index of the largest value see so far in the variable *ilargest*.

13.9 The *filter* pattern

A special case of a filtered-accumulation is called *filter*. Instead of summing the filtered items (for example), we collect the filtered items into an array. The new array is said to be a *reduction* of the original array.

Suppose we wish to extract the even numbers from an array. The code looks very much like the *sumEvens* function, but instead of adding in the desired item, we make an array out of it and add it to the back of the growing array of even numbers:

```
def extractEvens(items):
    evens = []
    for i in range(0,len(items),1):
        if (isEven(items[i])):
            evens = evens + [items[i]] # array concatenation
    return evens
```

Given an array of integers, *extractEvens* returns a (possibly empty) array of the even numbers:

```
>>> extractEvens([4,2,5,2,7,0,8,3,7])
[4, 2, 2, 0, 8]

>>> extractEvens([1,3,5,7,9])
[]
```

Note that our use of array concatenation makes our *extractEvens* function run very slowly for very large arrays[3]. To speed up *extractEvens*, we will use the object nature of arrays and invoke the *append* method to add an new element to the end of the existing array[4]. In Python, one invokes a method on an object by using the *dot operator*. Generically, the expression:

```
o.f()
```

[3]When you take a course in data structures and algorithm analysis, you will see that the run time of *extractEvens* grows in proportion to the *square* of the number of even elements in the list.

[4]A function that works on and is bundled with an object is often called a *method*. You will learn more about objects and methods later in your academic career.

means run method *f* on object *o*. Using the *append* method, our *extractEvens* function becomes:

```
def extractEvens(items):
    evens = []
    for i in range(0,len(items),1):
        if (isEven(items[i])):
            evens.append(items[i]) # method invocation
    return evens
```

We can see that the *append* method implements the procedure pattern in that it modifies the *evens* array, not returning anything. With this change, the *extractEvens* function will run much faster[5].

13.10 The *map* pattern

Mapping is a task closely coupled with that of filtering, but rather than collecting certain items, as with the *filter* pattern, we collect all the items. As we collect, however, we transform each item as we collect it. The basic pattern looks like this:

```
def map(f,items):
    result = []
    for i in range(0,len(items),1):
        transformed = f(items[i])
        result.append(transformed)
    return result
```

Here, function *f* is used to transform each item in the before it is added to the growing result.

Suppose we wish to subtract one from each element in an array. First we need a transforming function that reduces its argument by one:

```
def decrement(x): return x - 1
```

Now we can "map" the *decrement* function over an array of numbers:

```
>>> map(decrement,[4,3,7,2,4,3,1])
[3, 2, 6, 1, 3, 2,  0]
```

13.11 The *shuffle* pattern

Sometimes, we wish to combine two arrays into a third array, This is easy to do with the concatenation operator, `+`.

```
array3 = array1 + array2
```

[5]The reason is Python arrays are dynamic arrays, in that they grow larger as needed. If dynamic arrays are implemented properly, then the total cost of all appends divided by the number of appends is a small constant value. The upshot is that *extractEvens* now will in time proportional to the number of even values in the array, rather than the square of that number. This is quite an improvement! Suppose the input array is composed solely of a million even numbers. We would expect our new version of *extractEvens* to run, roughly, a million times faster than the original version.

This places the first element in the second array after the last element in the first array. However, many times we wish to intersperse the elements from the first array with the elements in the second array. This is known as a *shuffle*, so named since it is similar to shuffling a deck of cards. When a deck of cards is shuffled, the deck is divided in two halves (one half is akin to the first array and the other half is akin to the second array). Next the two halves are interleaved back into a single deck (akin to the resulting third array).

We can use a loop to shuffle two arrays. If both arrays are exactly the same length, a shuffling function is easy to implement using the *accumulate* pattern. Let's first assume the arrays are exactly the same size, because that makes things easier:

```
def shuffle(array1,array2):
    array3 = []
    for i in range(0,len(array1),1):
        array3.append(array1[i])
        array3.append(array2[i])
    return array3
```

Note how we initialized the resulting array *array3* to the empty array. Then, as we walked the first array, we pulled elements from both arrays, adding them into the resulting array.

When we have walked past the end of *array1* is empty, we know we have also walked past the end of *array2*, since the two arrays have the same size.

If the incoming arrays do not have the same length, life gets more complicated:

```
def shuffle2(array1,array2):
    array3 = []
    if (len(array1) < len(array2)):
        for i in range(0,len(array1),1):
            array3.append(array1[i])
            array3.append(array2[i])
        return array3 + array2[i+1:]
    else:
        for i in range(0,len(array2),1):
            array3.append(array1[i])
            array3.append(array2[i])
        return array3 + array1[i+1:]
```

Note the need to add the remaining elements in the longer array to the items that made it into *array3*.

We can also use a *while* loop that goes until one of the arrays is empty. This has the effect of removing the redundant code in *shuffle2*:

```
def shuffle3(array1,array2):
    array3 = []
    i = 0
    while (i < len(array1) and i < len(array2)):
        array3.append(array1[i])
        array3.append(array2[i])
        i = i + 1
    ...
```

When the loop ends, one or both of the arrays have been exhausted, but we don't know which one or ones. A simple solution is to add both remainders to *array3* and return.

```
def shuffle3(array1,array2):
    array3 = []
    i = 0
    while (i < len(array1) and i < len(array2)):
        array3.append(array1[i])
        array3.append(array2[i])
        i = i + 1
    return array3 + array1[i:] + array2[i:]
```

Suppose *array1* is empty. Then the expression `array1[i:]` will generate the empty array. Adding the empty array to *array3* will have no effect, as desired. The same is true if *array2* (or both *array1* and *array2* are empty).

Looking back at shuffle2, we see that the code would be much simpler if we could guarantee that one array was shorter than another. Suppose we write the code so that the assumption is that the first array is the shorter. Our code becomes:

```
def shuffle4(array1,array2):
    array3 = []
    for i in range(0,len(array1),1):
        array3.append(array1[i])
        array3.append(array2[i])
    return array3 + array2[i+1:]
```

But what happens when the second array is shorter? The code will fail (how?) unless we handle it. Here's one way to handle it:

```
def shuffle4(array1,array2):
    if (len(array2) < len(array1)):
        return shuffle4(array2,array1) # arrays are flipped
    else:
        array3 = []
        for i in range(0,len(array1),1):
            array3.append(array1[i])
            array3.append(array2[i])
        return array3 + array2[i+1:]
```

Note that we re-call *shuffle4* with *array2* as the first array if the length of *array2* is less than the length of *array1*. On this second call to *shuffle4*, we guarantee that the shorter array is the first array.

Although it may seem, at first, rather strange to call *shuffle4* from within *shuffle4*, Computer Scientists do this trick all the time. It's called recursion and you will learn much more about recursion in a later chapter.

13.12 The *merge* pattern

With the *shuffle* pattern, we always took the head elements from both arrays at each step in the shuffling process. Sometimes, we wish to place a constraint of the choice of elements. For example, suppose the two arrays to be combined are sorted and we wish the resulting array to be sorted as well. The following example shows that shuffling does not always work:

```
>>> a = [1,4,6,7,8]
>>> b = [2,3,5,9]

>>> c = shuffle2(a,b)
[1, 2, 4, 3, 6, 5, 7, 9, 8]
```

The *merge* pattern is used to ensure the resulting array is sorted and is based upon the *filtered-accumulate* pattern. The twist is we only accumulate an item *if* it is the smallest item in the two arrays that has not already been accumulated. We start by keeping two index variables, one pointing to the smallest element in *array1* and one pointing to the smallest element in *array2*. Since the arrays are ordered, we know the that the smallest elements are at the head of the arrays:

```
i = 0  # index variable for array1
j = 0  # index variable for array2
```

Now, we loop, similar to *shuffle3*:

```
while (i < len(array1) and j < len(array2)):
```

Inside the loop, we test to see if the smallest element in *array1* is smaller than the smallest element in *array2*:

```
    if (array1[i] < array2[j]):
```

If it is, we add the element from *array1* to *array3* and increase the index variable i for *array1* since we have "used up" the value at index i.

```
        array3.append(array1[i])
        i = i + 1
```

Otherwise, *array2* must have the smaller element and we do likewise:

```
        array3.append(array2[j])
        j = j + 1
```

Finally, when the loop ends (i or j has gotten too large), we add the remainders of both arrays to *array3* and return:

```
return array3 + array1[i:] + array2[j:]
```

In the case of merging, one of the arrays will be exhausted and the other will not. As with *shuffle3*, we really don't care which array was exhausted.

Putting it all together yields:

```
def merge(array1,array2):
    array3 = []
```

```
    i = 0
    j = 0
    while (i < len(array1) and j < len(array2)):
        if (array1[i] < array2[j]):
            array3.append(array1[i])
            i = i + 1
        else:
            array3.append(array2[j])
            j = j + 1
    return array3 + array1[i:] + array2[j:]
```

13.13 The *fossilized* pattern

Sometimes, a loop is so ill-specified that it never ends. This is known as an *infinite loop*. Of the two loops we are investigating, the *while* loop is the most susceptible to infinite loop errors. One common mistake is the *fossilized* pattern, in which the index variable never changes so that the loop condition never becomes false:

```
i = 0
while (i < n):
    print(i)
```

This loop keeps printing until you terminate the program with prejudice. The reason is that i never changes; presumably a statement to increment i at the bottom of the loop body has been omitted.

13.14 The *missed-condition* pattern

With the missed condition pattern, the index variable is updated, but it is updated in such a way that the loop condition never evaluates to false.

```
i = n
while (i > 0):
    print(i)
    i += 1
```

Here, the index variable i needs to be decremented rather than incremented. If i has an initial value greater than zero, the increment pushes i further and further above zero. Thus, the loop condition never fails and the loop becomes infinite.

13.15 Code

For Linux and Mac users, the code from this chapter can be found here

For Windows users, look here (you will need to save the file as *loops.py*).

Chapter 14

More on Input

Now that we have learned how to loop, we can perform more sophisticated types of input.

14.1 Converting command line arguments en mass

Suppose all the command-line arguments are integers that need to be converted from their string versions stored in *sys.argv*. We can use a loop and the accumulate pattern to accumulate the converted string elements:

```
def convertArgsToNumbers():
    result = []
    # start at 1 to skip over program file name
    for i in range(1,len(sys.argv),1):
        num = int(sys.argv[i])
        result.append(num)
    return result
```

The accumulator, *result*, starts out as the empty array. For each element of *sys.argv* beyond the program file name, we convert it and store the result in *num*. We then append that number to the growing array.

With a program file named *convert.py* as follows:

```
import sys

def main():
    ints = convertArgsToNumbers()
    print("original args are",sys.argv[1:])
    print("converted args are",ints)
    return 0

def convertArgsToNumbers():
    ...

main()
```

we get the following behavior:

```
$ python convert.py 1 34 -2
```

```
original args are ['1', '34', '-2']
converted args are [1, 34, -2]
```

Note the absence of quotation marks in the converted array, signifying that the elements are indeed numbers.

14.2 Reading individual items from files

Instead of reading all of the file at once using the *read* function, we can read it one item at a time. When we read an item at a time, we always follow this pattern:

```
open the file
read the first item
while the read was good
    process the item
    read the next item
close the file
```

In Python, we tell if the read was good by checking the value of the variable that points to the value read. Usually, the empty string is used to indicate the read failed.

Processing files a line at a time

Here is another version of the *copyFile* function from Chapter 10. This version reads and writes one line at a time. In addition, the function returns the number of lines processed:

```
def copyFile(inFile,outFile):
    in = open(inFile,"r")
    out = open(outFile,"w")
    count = 0
    line = in.readline()
    while (line != ""):
        out.write(line)
        count += 1
        line = in.readline()
    in.close()
    out.close()
    return count
```

Notice we used the counting pattern, augmented by printing out the current line every time the count was incremented.

Using a Scanner

A scanner is a reading subsystem that allows you to read whitespace-delimited tokens from a file. To get a scanner for Python, issue this command:

```
wget troll.cs.ua.edu/cs150/book/scanner.py
```

To use a scanner, you will need to import it into your program:

```
from scanner import *
```

Typically, a scanner is used with a loop. Suppose we wish to count the number of short tokens (a token is a series of characters surrounded by empty space) in a file. Let's assume a short token is one whose length is less than or equal to some limit. Here is a loop that does that:

```
def countShortTokens(fileName):
    s = Scanner(fileName)              #create the scanner
    count = 0
    token = s.readtoken()              #read the first token
    while (token != ""):               #check if the read was good
        if (len(token) <= SHORT_LIMIT):
            count += 1
        token = s.readtoken()          #read the next token
    s.close()                          #always close the scanner when done
    return count
```

Note that the use of the scanner follows the standard reading pattern: opening (creating the scanner), making the first read, testing if the read was good, processing the item read (by counting it), reading the next item, and finally closing the file (by closing the scanner) after the loop terminates. Using a scanner always means performing the five steps as given in the comments. This code also incorporates the filtered-counting pattern, as expected.

14.3 Reading Tokens into an Array

Note that the *countShortTokens* function is doing two things, reading the tokens and also counting the number of short tokens. It is said that this function has two *concerns*, reading and counting. A fundamental principle of Computer Science is *separation of concerns*. To separate the concerns, we have one function read the tokens, storing them into an array (reading and storing is considered to be a single concern). We then have another function count the tokens. Thus, we will have separated the two concerns into separate functions, each with its own concern. Here is the reading (and storing) function, which implements the accumulation pattern:

```
def readTokens(fileName):
    s = Scanner(fileName)              #create the scanner
    items = []
    token = s.readtoken()              #read the first token
    while (token != ""):               #check if the read was good
        items.append(token)            #add the token to the items array
        token = s.readtoken()          #read the next token
    s.close()                          #always close the scanner when done
    return items
```

Next, we implement the filtered-counting function. Instead of passing the file name, as before, we pass the array of tokens that were read:

```
def countShortTokens(items):
    count = 0
    for i in range(0,len(items),1)
        if (len(items[i]) <= SHORT_LIMIT):
```

```
            count += 1
    return count
```

Each function is now simpler than the original function. This makes it easier to fix any errors in a function since you can concentrate on the single concern implemented by that function.

14.4 Reading Records into an Array

Often, data in a file is organized as *records*, where a record is just a collection of consecutive tokens. Each token in a record is known as a *field*. Suppose every four tokens in a file comprises a record:

```
"Amber Smith"       President   32   87000.05
"Thad Jones"        Assistant   15   99000.42
"Ellen Thompson"    Hacker       2  147000.99
```

Typically, we define a function to read one collection of tokens at a time. Here is a function that reads a single record:

```
def readRecord(s):                  # we pass the scanner in
    name = s.readstring()
    if (name == ""):
        return ""                   # no record, returning the empty string
    name = name[1:-1]               # strip the quotes from the name
    title = s.readtoken()
    years = s.readint()
    salary = s.readfloat()
    return [name,title,years,salary]
```

Note that we return either a record as an array or the empty string if no record was read. Since name is a string, years of service is an integer, and salary is a real number, we read them appropriately with the *readstring*, *readinteger*, and *readfloat* methods, respectively.

To total up all the salaries, for example, we can use an accumulation loop (assuming the salary data resides in a file named *salaries*). We do so by repeatedly calling *readrecord*:

```
def totalPay(fileName):
    s = Scanner(fileName)
    total = 0
    record = readRecord(s)
    while (record != ""):
        total += record[3]
        record = readRecord(s)
    s.close()
    return total
```

Note that it is the job of the caller of *readRecord* to create the scanner, repeatedly send the scanner to *readRecord*, and close the scanner when done. Also note that we tell if the read was good by checking to see if *readRecord* return `None`.

The above function has two stylistic flaws. It uses those magic numbers we read about in Chapter 6. It is not clear from the code that the field at index three is the salary. To make the code more readable, we can

set up some "constants" in the global scope (so that they will be visible everywhere): The second issue is that that the function has two concerns (reading and accumulating). We will fix the magic number problem first.

```
NAME = 0
TITLE = 1
SERVICE = 2
SALARY = 3
```

Our accumulation loop now becomes:

```
total = 0
record = readRecord(s)
while (record != ""):
    total += record[SALARY]
    record = readRecord(s)
```

We can also rewrite our *readRecord* function so that it only needs to know the number of fields:

```
def readRecord(s):                  # we pass the scanner in
    name = s.readstring()
    if (name == ""):
        return ""                   # no record, returning None
    name = name[1:-1]               # strip the quotes from the name
    title = s.readtoken()
    years = s.readint()
    salary = s.readfloat()

    # create an empty record

    result = [0,0,0,0]

    # fill out the elements

    result[NAME] = name
    result[TITLE] = title
    result[SERVICE] = service
    result[SALARY] = salary

    return result
```

Even if someone changes the constants to:

```
NAME = 3
TITLE = 2
SERVICE = 1
SALARY = 0
```

The code still works correctly. Now, however, the salary resides at index 0, but the accumulation loop is still accumulating the salary due to its use of the constant to access the salary.

14.5 Creating an Array of Records

We can separate the two concerns of the *totalPay* function by having one function read the records into an array and having another total up the salaries. An array of records is known as a *table*. Creating the table is just like accumulating the salary, but instead we accumulate the entire record into an array:

```
def readTable(fileName):
    s = Scanner(fileName)
    table = []
    record = readRecord(s)
    while (record != ""):
        table.append(record)
        record = readRecord(s)
    s.close()
    return table
```

Now the table holds all the records in the file. We must remember to enclose the record in square brackets before we accumulate it into the growing table. The superior student will try this code without the brackets and ascertain the difference.

The accumulation function is straightforward:

```
def totalPay(fileName):
    table = readTable(fileName)
    total = 0
    for i in range(0,len(table),1):
        record = table[i]
        total += record[SALARY]
    return total
```

We can simply this function by removing the temporary variable *record*:

```
def totalPay(fileName):
    table = readTable(fileName)
    total = 0
    for i in range(0,len(table),1):
        total += table[i][SALARY]
    return total
```

Since a table is just an array, so we can walk it, accumulate items in each record (as we just did with salary), filter it and so on.

14.6 Other Scanner Methods

A scanner object has other methods for reading. They are

`readline()` read a line from a file, like Python's *readline*.

`readchar()` read the next non-whitespace character

`readrawchar()` read the next character, whitespace or no

You can also use a scanner to read from the keyboard. Simply pass an empty string as the file name:

```
s = Scanner("")
```

Finally, you can scan tokens and such from a string by first creating a keyboard scanner, and then setting the input to the string you wish to scan:

```
s = Scanner("")
s.fromstring(str)
```

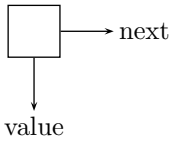
Chapter 15

Lists

Recall that what we are calling arrays are called lists by the Python community[1]. Now we will introduce true lists; these lists will be our first attempt at building a *data structure*. A data structure is simply a bundling of pieces of data together in a single entity along with a set of operations (implemented using functions) to manipulate such entities. Common operations for data structures include putting data into the structure and taking data out. Moreover, the data structure and the functions which implement the operations are constructed so that the operations do not take an inordinate amount of time as more and more data is placed into the structure.

15.1 The Node Data Structure

Our lists will use an intermediate data structure known as a *node*. A node is a simple structure that holds two pieces of data. The first data item is some arbitrary value, while the second is another node. Graphically, nodes are represented as a box with two arrows. The downward pointing arrow typically points to the value while the rightward pointing arrow points to other node.



In actuality, the data items are the addresses in memory where the values and nodes reside; such memory addresses are known as *pointers*. Thus, the graphical representation of the node is quite accurate, with the arrows "pointing" to the data items. The downward arrow is known as the *value pointer* and the rightward arrow is known as the *next node pointer* or simply *next pointer*.

All data structures are built upon other data structures, so there must be some low-level data structure that is built into the language. Python has two such built-in structures, arrays and objects[2]. Objects would be a good choice as a structure upon which to build nodes, but you don't know about objects yet, so we will use arrays instead. You will learn more about objects in a later chapter and in subsequent classes.

Each of our nodes will actually be an array of length two. The first slot in the array will hold the value pointer and the second will hold the *next* pointer. Now that we have our structure, we will need to define a set of operations for nodes. The first one we define is a function to create a node. Such a function is known as a *constructor*:

[1] We can't help if other folks are sloppy in their terminology.

[2] Python has other built-in data structures as well. But even if Python had no built-in structures, one can use functions themselves as data structures!

```
def NodeCreate(value,next):     #this is the constructor
    return [value,next]
```

Note that the constructor takes two arguments, the *value* and the *next* pointer and simply returns an array holding those two values. Next, we define two operations for retrieving the data items stored in the node:

```
def NodeValue(n):
    return n[0]

def NodeNext(n):
    return n[1]
```

The node from which we wish to extract information is passed as an argument in calls to these two functions. The definitions rely on the fact that the *value* is stored in the first slot and the *next* pointer is stored in the second slot of the array that constitutes a node. Typically, functions that retrieve values or other information from a data structure are known as *accessors* or *getters*.

Finally, we define operations for changing the data in a node:

```
def NodeSetValue(n,value):
    n[0] = value
    return

def NodeSetNext(n,next):
    n[1] = next
    return
```

Such functions are typically known as *mutators* or *setters*, since the change the data inside the structure. Note that they implement the procedure pattern, as they do not compute any new values.

With these definitions in place, we can now make nodes and change their values. We will use the value `None` to indicated that the next pointer of a node should not be followed. Such pointers-which-must-not-be-followed are generically known as *null* pointers.

```
a = NodeCreate(3,None)
b = NodeCreate(4,None)
print("a's value is",NodeValue(a))
print("b's value is",NodeValue(b))
NodeSetValue(a,"two")
print("a's value now is",NodeValue(a))
```

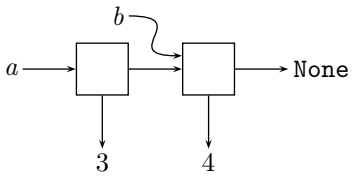
Executing this code yields the following output:

```
a's value is 3
b's value is 4
a's value now is two
```

Continuing on from the previous bit of code, we can even join nodes together:

```
NodeSetNext(a,b)        # link!
print("b's value, through a is",NodeValue(NodeNext(a)))
print("a is",a)
```

The first statement sets the *next* pointer of a to b; in other words, the null pointer of node a was replaced by a pointer to node b. Afterwards this statement, a and b are said to be *linked* together, as in a chain.



Indeed, a group of nodes linked together via *next* pointers is known as a *linked list*. The next pointer of the last node in a list of nodes is always null. The output of the first print statement conforms to expectation; we see the value of the node a's *next* pointer points to, namely b, is 4:

```
b's value, through a is 4
```

The output of the second print statement reveals that nodes are indeed arrays:

```
a is [2, [4, None]]
```

In fact, if we replace b's *next* pointer, currently `None`, with a new node and then print a again:

```
NodeSetNext(b,NodeCreate(6,None))
print("a is",a)
```

we start to see why nodes can be used to form lists:

```
a is [2, [4, [6, None]]]
```

Note that the value 6 can only be reached through a or through b. There is no variable like a or b that points directly to that node.

15.2 The Linked-List Data Structure

Now we could stop at this point and just use nodes and their operations to make true lists, but note that the node operators freed us from knowing that nodes were based upon arrays. If you look closely at the above code where we tested our node operations, you will see that there is absolutely no clue that nodes are built from arrays. In fact, we only got an inkling of the internal structure of a node when we printed a node directly using *print*, which is *not* a node operator. Likewise, we can build a list data structure based upon nodes and be freed from both knowing lists are built from nodes and that nodes are built from arrays.

The first list operation we will define is the constructor:

```
def ListCreate(*args):
    items = None
    for i in range(len(args)-1,-1,-1): # work from right to left
        items = join(args[i],items)
    return items
```

This function has a bit of Python voodoo in it. The first bit is the asterisk before the single formal parameter, *arg*. This means that any number of arguments can be passed to *ListCreate* and all of them will be bundled up into an array that is bound to the variable *args*. The second bit is the funny looking `range(len(args)-1,-1,-1)`, which produces the indices of all the elements in *args* in reverse order. This causes the elements of *args* to be joined up in reverse. The reversal is necessary because the first value that is joined ends up at the back of items[3].

When *ListCreate* is called with no arguments, the constructor returns an empty list:

```
>>> ListCreate()
None
```

Thus an empty list is represented by `None`. If we pass a number of arguments, we see output similar to when we joined a bunch of nodes together:

```
>>> ListCreate(2,"six",False)
[2, ['six', [False, None]]]
```

Therefore, *ListCreate* can take any number of values to populate a newly created list[4].

The last bit of new stuff in the body of the *ListCreate* function is the reference to a *join* function:

```
def join(value,list):
    return NodeCreate(value,list)
```

The *join* function takes a value and a list as arguments and returns a new node that glues the two arguments together. The result is a list one item larger than the list that is passed in.

We can see from the definition of *ListCreate* that a list either has the value `None`, if there were no arguments, or the result of *join*, if there were. Since *join* returns a node, then a list is actually a node (or `None` if the list is empty). You may be wondering at this point why we bother to distinguish lists and nodes. One reason is that lists in other programming languages aren't quite so easy to implement as in Python and, in those languages, nodes are quite distinct from lists. Thus, we want you to be ready for those languages. Another reason is it is important to practice the concept of abstraction. Modern software systems exhibit a large number of abstraction layers. With our lists, we can see three layers of abstractions: `arrays`, which abstract some underlying machine code, `nodes`, which abstract `arrays` of length two, and `lists`, which abstracts a chain of `nodes`. Each level of abstraction frees us from thinking about the details of the underlying layers.

Let us now look at the *join* operation more closely. Here is a rewrite that uses a temporary local variable, *n*, to hold the new node that will join the given value and list together:

[3] If you are unclear about the need for reversing the arguments to *ListCreate*, you should instrument the loop by printing out the value of items each time the loop executes. Then all will become clear.

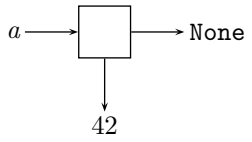
[4] A function that can take a different number of arguments from call to call is called a *variadic* function.

```
def join(value,list):
    #before
    n = NodeCreate(value,None)
    #during
    NodeSetNext(n,list)
    #after
    return n
```

First, let us create a list to which we will join a value:

```
a = ListCreate(42)
```

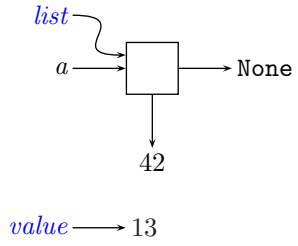
Graphically, the situation looks like this:



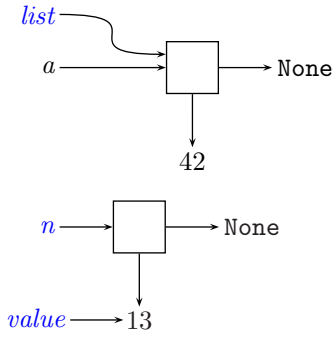
Now, we call *join* to join the value of 13 to the list *a*.

```
a = join(13,a)
```

At the point of the `#before` comment in join, the formal parameters *value* and *list* have been set to 13 and *a*, respectively. The situation looks like this:

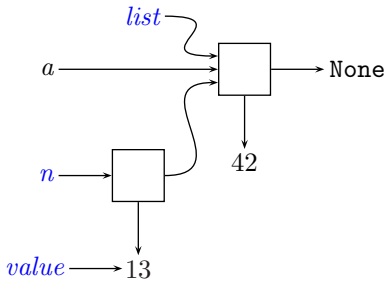


The variables *list* and *value*, since they are local to the function *join*, are shown in blue[5]. After the step `n = NodeCreate(value,None)`, marked with the comment `#during`, a new node with the given value is created and bound to the variable *n*. The situation changes to:

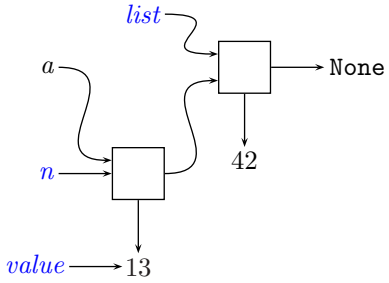


[5]When two variables, in this case *a* and *list*, point to the same thing, they are said to be *aliases* of one another.

The next step sets the *next* pointer of n to point to the given list, via the statement `NodeSetNext(n,list)`:



Finally, the value of n is returned and a is assigned this new value:



From the drawing, we can see that list a now includes the value 13 at the front of the list, as intended. Of course, at this point the variables local to *join*, shown in blue, are no longer in scope.

Adding values to the front of a list is known as *prepending*, so *join*[6] prepends a value to a list. Instead of adding new values to the front, we can also add values to the back of the list. Such a procedure is termed *appending*, but we will restrict appending to lists proper. In other words, we will only append a list to an existing list:

```
def append(listA,listB): #listA length must be > 0
    node = listA
    while (NodeNext(node) != None):
        node = NodeNext(node)
    NodeSetNext(node,listB)
    return
```

Here, we set the variable *node* to the beginning of *listA*. Then, as long as *node*'s *next* pointer points to a node and not `None`[7], we set *node* to the next node in the list. We repeat this process until we reach the last node in the list, whose *next* pointer does indeed point to `None`. At this point, we drop out of the loop and set the next pointer of this last node to `listB`[8]. Generally, you can quickly tell if a function is destructive or non-destructive by looking at the return value. If the function implements the *procedure* pattern, it is likely destructive. If it implements the *function* pattern, it is likely non-destructive[9].

Here is an example:

[6] *Prepend* is rather an awkward term, so we will use *join* as the function name, rather than *prepend*.

[7] The test condition of the while loop is the reason for the comment that the length of *listA* must be greater than zero.

[8] Note that while *NodeValue* and *NodeNext* are non-destructive operations, *append* is a destructive operation, since a pointer in the list is actually changed.

[9] Sometimes, destructive functions return a value for convenience to the caller, in which case this generalization fails.

```
>>> a = ListCreate(2,4,6)
>>> b = ListCreate(1,3,5)

>>> append(a,b)
[2, [4, [6, [1, [3, [5, None]]]]]]

>>> NodeNext(a)
[4, [6, [1, [3, [5, None]]]]]
```

If we wish to append a single value to a list, we must turn the single value into a list first:

```
>>> a = ListCreate(2,4,6)

>>> append(a,ListCreate(8))
[2, [4, [6, [8, None]]]]
```

The process of moving from one node to the next in a list is known as *walking* the list. Obviously, the longer the list, the longer it takes to walk it. Thus, appending to a list can take much longer than prepending to the same list, since prepending takes the same amount of time regardless of how long the list is. However, as you will learn in your data structures class, there is a much faster way to append to a list. If you cannot wait to find out this clever implementation of *append*, search for the phrase "linked list tail pointer" on the interwebs.

Once we are able to insert values into a list, we now need functions to look at the values in the list. Two common operations of a list are *head*, which retrieves the first item in the list, and *tail*, which returns a the list made up of the all the nodes in the original list *except* the first node:

```
def head(items):
    return NodeValue(items)

def tail(items):
    return NodeNext(items)
```

As you can see, these functions are *wrappers* to the underlying node functions, but are used frequently because they are smaller names with generally universally understood semantics. Often, wrappers to the functions that change node values and pointers are defined as well:

```
def setHead(items,value):
    NodeSetValue(items,value)
    return

def setTail(items,next):
    NodeSetNext(items,next)
    return
```

Here is an example:

```
>>> a = ListCreate(1,8,7,2)
```

```
>>> a
[1, [8, [7, [2, None]]]]
>>> head(a)
1
>>> tail(a)
[8, [7, [2, None]]]
>>> head(a)
1
>>> head(tail(a))
8
>>> setHead(tail(a),0)
>>> a
[1, [0, [7, [2, None]]]]
```

Note that taking the tail of a list does not alter the original list in any way[10].

Finally, the last two operations we will discuss retrieve and set the values of nodes anywhere in the list. To specify which node is of interest, the caller of these functions will supply an index. This index will function exactly like the zero-based index of arrays; index zero will refer to the first value, index one will refer to the second value, and so on. The function *ListIndex* gets a value at the given index and *ListSetIndex* replaces the old value at the given index with the given value. Both of these functions will use a helper function named *ListIndexNode*, which will return the node at a given index[11]:

```
def ListIndexNode(items,index):
    node = items
    for i in range(0,index,1):
        node = tail(node)
    return node

def ListIndex(items,index):
    node = ListIndexNode(items,index)
    return head(node)

def ListSetIndex(items,index,value):
    node = ListIndexNode(items,index)
    NodeSetValue(node,value)
    return
```

Note that the *ListIndexNode* function walks the list for *ListIndex* and *ListSetIndex*. Note further that this implementation does no error checking. What happens if the index is greater than or equal to the number of nodes in the list?

Rather than have you type all this code in, the above *node* and *list* operations are bundled into a single module, named *List.py*. You can get the module with this command:

```
wget troll.cs.ua.edu/cs150/book/List.py
```

on Linux and this command:

[10] Again, when an operation does not affect its operands, it is said to be non-destructive. So, like the functions they wrap, *head* and *tail* are both non-destructive

[11] This is one of the reasons objects would make a better base structure than arrays for nodes and lists. Ideally, nobody should call *ListIndexNode* except *ListIndex* and *ListSetIndex*. If objects are used, *ListIndexNode* can be made *private*, so that only other list functions can call it.

```
curl -O troll.cs.ua.edu/cs150/book/List.py
```

on a Mac. For Windows, browse to:

```
http://troll.cs.ua.edu/cs150/book/ms/List.py
```

and save the page.

These versions of *List.py* generates better error messages for *ListIndex* and *ListSetIndex*.

15.3 More on Walking lists

This idea of walking lists is an extremely important concept; so important that we will review the two "walks" that are commonly needed when manipulating lists. The first is walking to a certain location in list. Usually, one walks to index (covered previously) or to a value. Of course, walking to a value is simply the search pattern, implemented for lists. Here is a search function that works on lists:

```
def find(value,items):
    spot = items;
    while (spot != None and head(spot) != value):
        spot = tail(spot)
    return spot
```

Note that this version returns the node containing the value if the value is found and `None` otherwise. Because of the possibility that a value may not be in the list, we must add the condition `spot != None` to the while loop test. If we didn't, then *spot* would eventually reach `None` and the *head* operation on `None` would generate an error. Sometimes, *find* would be written with a simpler loop test, but complicated by a return from the body of the loop:

```
def find(value,items):
    spot = items;
    while (spot != None):
        if (head(spot) == value): return spot
        spot = tail(spot)
    return None
```

These two implementations are semantically equivalent; the former is usually preferred stylistically, but the latter is likely more common.

The second walk one is likely to encounter is a walk to the next-to-the-last node in a list, which is useful in many situations: removing the last item, adding a value to the end of a list quickly (again, search for "linked list tail pointer" for details), and so on.

One must walk the list to get to the penultimate node. Here is one attempt; it keeps two pointers when walking the list, a leading pointer and a trailing pointer that stays one node behind the leader. The walk ends when the *next* pointer of the leading node becomes null. We call this pattern the *trailing value pattern*:

```
def getPenultimateNode(items):
    trailing = None;
```

```
    leading = items;
    while (tail(leading) != None): #when to stop walking
        trailing = leading
        leading = tail(leading)
    return trailing
```

If we walked until the lead node became null, then the trailing node would be the last node in the list, not the next-to-the-last node. However, checking the next pointer of a node is a dangerous proposition. What happens in the case of an empty list? How many nodes must be in a list for this function to work properly?

The above approach can be simplified by checking if the trailing pointer's next pointer points to a node whose next pointer is null. Although the check is a bit more complicated, it obviates the need for the leading node:

```
def getPenultimateNode(items):
    trailing = items;
    while (tail(tail(trailing)) != None):
        trailing = tail(trailing)
    return trailing
```

The two versions of *getPenultimateNode* are similar, but not exactly equivalent. How many nodes must be in a list for the second version to work properly?

15.4 A Walking Application

An application of the trailing value pattern is to insert a value into an ordered list so that the list remains ordered[12]. For example, if the ordered list is:

```
[1,[3,[8,[13,[14,[17,None]]]]]]
      ^  ^
```

and we wish to insert 10 into the list, we must place the 10 between the 8 and the 13 so that the list values remain in increasing order (from left to right). The ^ marks show where the trailing and leading pointers should end up when the walk is finished. If the 10 is inserted between the trailing and leading pointers, we end up with:

```
[1,[3,[8,[10,[13,[14,[17,None]]]]]]]
```

as desired. Using *getPenultimateNode* as a starting point, we have:

```
def insertInOrder(value,items):
    trailing = None;
    leading = items;
    while (...): #when to stop walking
        trailing = leading
        leading = tail(leading)
```

[12]This idea of repeatedly inserting items in a list with the list remaining ordered is the basis for an important data structure, the *priority queue*. Priority queues are used in network connectivity algorithms and discrete-event simulations. You will learn about priority queues in a subsequent class

```
    # insert new value in between trailing and leading
    ...
    return "done"
```

The ellipses mark where changes to the trailing value pattern need to be made. The first ellipsis (the while test) should force the walk to stop when the correct insertion point is found. Clearly, that is when leading value is greater than the value to be inserted. The second ellipsis concerns how the actual insertion is to be performed. We know we will need to:

- create a new node containing the new value
- point the new node's next pointer to the leading node
- point the trailing node's next pointer to the new node

All that's left then is to fill in the blanks. Here is the result:

```
def insertInOrder(value,items):
    trailing = None;
    leading = items;
    while (head(leading) < value): #when to stop walking
        trailing = leading
        leading = tail(leading)
    # insert new value in between trailing and leading
    n = NodeCreate(value,None)
    NodeSetNext(n,leading)
    NodeSetNext(trailing,n)
    return "done"
```

Note that we wanted to *stop* when the leading value is *greater* than the value to be inserted. Since we are working with a while loop, we need to reverse the logic so that the walk *continues* as long as the leading value is *less than* the new value[13].

Testing our code with the above example yields:

```
>>> a = ListCreate(1,3,8,13,14,17)
>>> insertInOrder(10,a)
>>> a
[1,[3,[8,[10,[13,[14,[17,None]]]]]]]
```

It appears our code works correctly! Very exciting! Unfortunately, the excitement of getting code to work seduces both novice and experienced Computer Scientists alike into thinking the task is finished. Many times, including this case in particular, initial success only means the basic idea is correct, not that the code is correct, for all possible inputs. Indeed, for our implementation of *insertInOrder*, there are *edge cases* for which this code fails. An edge case is a set of inputs that force the code to do as much (or as little) work as possible. For *insertInOrder*, what inputs causes the function to do as much work as possible? The only place where a variable amount of work is performed is the loop that walks the list. In order to make the loop go as many times as possible, it is clear that we would need to insert a value at the very end of the list. Doing so yields:

[13] Actually, reversing the logic of greater than yields less than *or equal*, but we are ignoring situations when values match exactly. What happens in those cases is left as an exercise.

```
>>> a = ListCreate(1,3,8,13,14,17)
>>> insertInOrder(20,a)
Traceback (most recent call last):
  File "test.py", line 15, in <module>
    print("inserting 18:",insertInOrder(20,a))
  File "test.py", line 6, in insertInOrder
    while (head(leading) < value): #when to stop walking
  File "/home/lusth/l1/book/List.py", line 34, in head
    return NodeValue(items)
  File "/home/lusth/l1/book/List.py", line 12, in NodeValue
    def NodeValue(n): return n[0]
TypeError: 'NoneType' object is not subscriptable
```

Likewise, making the loop do as little work as possible yields errors in the code. What inputs would cause the loop run very little or not at all? Obviously, if the value to be inserted is smaller than any value in the list, the loop does not need to be run. Less obviously, what if the given list is empty? Fixing *insertInOrder* to handle these edge cases is left as an exercise.

15.5 Processing Lists versus Arrays

Lists can be processed with loops, much like arrays. In fact, there is a direct translation from an array loop to a list loop. Suppose you have the following array loop:

```
for i in range(0,len(items),1):     # items is an array
    # loop body here
    ...
```

The corresponding list loop would be:

```
spot = items                        # items is an list
while (spot != None):
    # loop body here
    ...
    spot = tail(spot)
```

In addition, the empty array `[]` is replaced by the empty list `None` and the array reference `items[i]` is replaced by `head(items)`. Finally, references to the *append* method of arrays is replaced by calls to the *join* function. For example:

```
results.append(items[i])
```

is replaced by:

```
result = join(head(spot),results)
```

Let's look a some example translations. The counting pattern is very important for lists, since there is no built-in function like *len* for determining the length of a list. The array version of this pattern is:

```
count = 0
for i in range(0,len(items),1):
     count += 1
```

Translating the code to work with arrays using the above substitutions yields:

```
count = 0
spot = items                          #typical list walking initialization
while (spot != None):                 #typical list walking condition
    count += 1
    spot = tail(spot)                 #typical list walking update
```

As with arrays, each "step" taken results in a counter being incremented. For another example, let's look at a filtered accumulation. First, the array pattern:

```
def sumEvens(items):
    total = 0
    for i in range(0,len(items),1):
        if (items[i] % 2 == 0):
            total += items[i]
    return total
```

Now the list translation:

```
def sumEvens(items):
    total = 0
    spot = items
    while (spot != None):
        if (head(spot) % 2 == 0):
            total += head(spot)
        spot = tail(spot)
    return total
```

Finally, let's look at the *extreme index* pattern, since it adds a slight wrinkle. Here is the array version:

```
ilargest = 0
for i in range(1,len(items),1):
    if (items[i] > items[ilargest]):
        ilargest = i
```

Note that we have no instructions for translating `items[ilargest]`. We can, however, save both the largest value seen so far as well as the current index:

```
index = 0
spot = items
largest = head(spot)
ilargest = 0
while (spot != None):
```

```
        if (head(spot) > largest):
            ilargest = index
            largest = head(spot)
        index += 1
        spot = tail(spot)
```

In general, any processing that requires finding an index will be more complicated with a list.

15.5.1 The *filter* and *map* patterns

Filtering and mapping lists with loops is problematic, since using join instead of append ends up reversing the elements in the resulting list. This is because join puts an element at the beginning and thus the last element joined ends up at the front of the list and the next-to-the-last element ends up in the second position, and so on. Still, if the order of the result doesn't matter, then lists and loops go well together for this kind of processing. Here is an array loop that filters out the odd elements, leaving the even ones in the result:

```
def extractEvens(items):
    evens = []
    for i in range(0,len(items),1):
        if (isEven(items[i])):
            evens.append(items[i])
    return evens
```

Translating this to list notation yields:

```
def extractEvens(items):
    evens = None
    spot = items
    while (spot != None):
        if (isEven(head(spot))):
            evens = join(head(spot),evens)
        spot = tail(spot)
    return evens
```

Because of the reversal:

```
extractEvens(ListCreate(2,6,5,3,9,8,4))
```

returns:

```
[4,[8,[6,[2,None]]]]
```

15.5.2 The *shuffle* and *merge* patterns

As with arrays, shuffling and merging lists using loops is rather complicated. So much so that we will not even bother discussing loop-with-list versions. However, using recursion with lists greatly simplifies this type of processing, as we shall see in the next chapter.

15.5.3 The *fossilized* pattern

The primary reason for accidentally implementing the fossilized pattern is to forget the while loop update. For example, while:

```
spot = items
while (spot != None):
    # loop body here
    ...
    spot = tail(spot)
```

may have been intended, what gets coded is:

```
spot = items
while (spot != None):
    # loop body here
    ...
```

Since spot is never updated, it never becomes *None* (unless, of course, it was *None* to begin with). This leads to an infinite loop.

15.5.4 The *wrong-spot* pattern

The wrong-spot pattern is another common error. Here instead of referring to *spot*, one mistakenly refers to *items* instead. An example is this attempt at finding the smallest value in a list:

```
smallest = head(items)
spot = items
while (spot != None):
    if (head(spot) < smallest):
        smallest = head(items)  # ERROR: should use spot here
    spot = tail(spot)
```

This loop never sets smallest to anything except the first value in *items*.

15.6 Why Lists?

You may be asking, why go through all this trouble to implement lists when arrays seem to do every thing lists can do. The reason comes down to the fact that no data structure does all things well. For example, a data structure may be very fast when putting values in, but very slow in taking values out. Another kind of data structure may be very slow putting values in but very fast taking values out. This is why there are quite a number of different data structures, each with their own strengths.

In the case of lists and arrays, adding an item to a list can be much quicker than adding an item to an array. Consider four scenarios:

1. prepending a value to a list containing one item
2. prepending a value to a list containing one million items

3. prepending a value to an array containing one item
4. prepending a value to an array containing one million items

Scenarios 1, 2, and 3 all take about the same amount of time, whereas scenario 4 takes about a million times longer than the other scenarios. This is because when a value is joined to an array, a new array is created and all the elements in the original array are copied into the new array. If the old array has one value, then one value is copied. If the old array has a million values, then a million values are copied.

Another reason is that, in other programming languages besides Python, arrays have a fixed size and you cannot put more values in the array than there is room. A list, on the other hand, can grow without bound, subject to the memory available to the program. That said, sometimes arrays are the better data structure to use. Therefore, the List module contains two functions for converting between arrays and lists:

```
ArrayToList(items)  # where items is an array; a list is returned
ListToArray(items)  # where items is a list; an array is returned
```

Use these functions when the data is in one form, but you need it in another.

We will use lists heavily in the next chapter on recursion.

15.7 Problems

1. Assume a list holds the values 3, 8 and 10. Draw a box and pointer diagram of this list.
2. Suppose the loop in *ListCreate* is replaced with:

   ```
   for i in args: #an alternate loop
       items = join(i,items)
   ```

   Explain what happens and why when more than one argument is passed to *ListCreate*.
3. Assume a list holds the values 3 and 8. Consider appending the value 5. Draw a set of box and pointer diagrams, similar to those used in the discussion of *join*, that illustrate the action of *append*. You should have a drawing of the situation before the while loop begins, one for each update of the variable node while the loop is running, one after the new node is created, and one showing the list just before the function returns. Label your drawings appropriately.
4. Define a function named *appendValue*, which takes a list and a value and appends the value to the list. It should do so by setting the *next* pointer of the last node in the first list to a new node containing the given value. Your function should look very similar to *append*.
5. The module *List.py* exhibits an important principle in Computer Science, *abstraction*. The list operations treat nodes as abstract objects; that is to say, lists do not know nor care about how nodes are constructed. Verify this is true by modifying the node operations so that *value* pointer is stored in the second slot of the underlying two-element array and the *next* pointer is stored in the first slot. Now test the list operations to see that they work exactly as before.
6. The list operation *ListIndexNode* exhibits another important concept in Computer Science, *generalization*. Since both *ListIndex* and *ListSetIndex* need to walk the list, that list-waking code was generalized into the function *ListIndexNode*. Modify the *List.py* module so that *ListIndex* and *ListSetIndex* each incorporate the code of *ListNodeIndex*. Then comment out *ListIndexNode*. Both operations should function as before, but you should notice how similar they look. When you see such similarity, the similar code is a candidate for generalization.

7. Define a function named *length*, which takes a list as an argument and returns the number of items in the list.

8. Define a function named *reverse*, which takes a list as an argument and returns a new list with the same items as the given list, but in the opposite order. For example `(reverse (ListCreate 1 2 3 4))` should return the list `[4, [3, [2, [1, None]]]]`.

9. Define a function named *chop*, which takes a list as an argument and destructively removes the last element of a list. Your function should look similar to *getPenultimateNode*.

10. Define a function named *equals*, which takes two lists as arguments and returns whether or not the two lists have the same elements in the same order.

11. The linked lists developed in this chapter are known as *singly*-linked lists, a class of lists with a single pointer links a node to the next node in the list. Another class of lists, *doubly*-linked lists, provide an additional pointer from a node to its predecessor. Define a constructor for nodes upon which doubly-linked lists could be based.

12. Fix the given version of *insertInOrder* so that it correctly handles edge cases.

13. Consider the version of *insertInOrder* given in this chapter. If a value to be inserted matches a value already in the list, does the new value end up before or after the old value? How would you change the code to get the opposite behavior?

15.8 Code

For Linux and Mac users, the code from this chapter can be found here.

For Windows users, look here (you will need to save the file as *lists.py*).

Chapter 16

Recursion

In Chapter 7, we learned about *if* statements. When we combine *if* statements with functions that call themselves, we obtain a powerful programming paradigm called *recursion*.

Recursion is a form of looping; when we loop, we evaluate code over and over again. We use a conditional to decide whether to continue the loop or to stop. Recursive loops are often easier to write and understand, as compared to the iterative loops such as *while*s and *for*s, which you learned about in a previous chapter. In some programming languages, iterative loops are preferred as they use much less computer memory and are slightly faster as compared to recursive loops. In other languages, this is not the case at all. In general, there is no reason to prefer iterative loops over recursive ones, other than this memory issue and slight loss of performance. Any iterative loop can be written as a recursion and any recursion can be written as an iterative loop. Use a recursion if that makes the implementation more clear, otherwise, use an iterative loop.

Many mathematical functions are easy to implement recursively, so we will start there. Recall that the factorial of a number n is:

$$n! = n * (n-1) * (n-2) * ... * 2 * 1 \tag{16.1}$$

Consider writing a function which computes the factorial of a positive integer. For example, if the function were passed the value of 4, it should return the value of 24 since 4! is 4*3*2*1 or 24. To apply recursion to solve this problem or any problem, for that matter, it must be possible to state the solution of a problem so that it references a simpler version of the problem. For factorial, the factorial of a number can be stated in terms of a simpler factorial. Consider the two equations below:

$$0! = 1 \tag{16.2}$$

$$n! = n * (n-1)! \quad \text{otherwise} \tag{16.3}$$

Equation 16.2 says that the factorial of zero is one[1]. Equation 16.3 states that the factorial of any other (positive) number is obtained by multiplying the number by the factorial of one less than that number. Together, these two equations form a *recurrence equation* that describes the computation of factorial.

After some study, you should be able to see that this new way of describing factorial is equivalent to Equation 16.1, the one that that used ellipses[2]. Equation 16.1 gets the basic idea of factorial across but is not very precise. For example, how would you compute the factorial of three using Equation 16.1?

[1] Mathematicians, being an inclusive bunch, like to invite zero to the factorial party.
[2] Ellipses are the three dots in a row and are stand-ins for stuff that has been omitted.

The second form with the two equations is particularly well suited for implementation as a function in a computer program:

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

Note how the *factorial* function precisely implements the recurrence equation. Convince yourself that the function really works by tracing the function call:

```
factorial(3)
```

Decomposing the call, we find that:

```
factorial(3) is 3 * factorial(2)
```

since *n*, having a value of 3, is not equal to 0. Thus the second block of the *if* is evaluated and we can replace *n* with 3 and *n-1* with 2. Likewise, we can replace `factorial(2)` by `2 * factorial(1)`, yielding:

```
factorial(3) is 3 * 2 * factorial(1)
```

since *n*, now having a value of 2, is still not zero. Continuing along this vein, we can replace `factorial(1)` by `1 * factorial(0)`, yielding:

```
factorial(3) is 3 * 2 * 1 * factorial(0)
```

Now in this last call to factorial, *n* does have a value of zero, so we can replace `factorial(0)` with its immediate return value of one:

```
factorial(3) is 3 * 2 * 1 * 1
```

Thus, `factorial(3)` has a value of six:

```
>>> factorial(3)
6
```

as expected.

In contrast, here is a function which computes factorial using a *for* loop:

```
def factorial(n):
    total = 1;
    for i in range(1,n+1):
       total = total * i
    return total
```

We can see from this version that factorial is an accumulation. This version also more closely follows Equation 16.1. Note that we have to extend the upper end of the range by one in order to get n included in the accumulation, since the upper limit of the *range* function is exclusive.

16.1 The parts of a recursive function

Recursive approaches rely on the fact that it is usually simpler to solve a smaller problem than a larger one. In the factorial problem, trying to find the factorial of $n-1$ is a little bit simpler[3] than finding the factorial of n. If finding the factorial of $n-1$ is still too hard to solve easily, then find the factorial of $n-2$ and so on until we find a case where the solution is dead easy. With regards to factorial, this is when n is equal to zero. The *easy-to-solve* code (and the values that get you there) is known as the *base* case. The *find-the-solution-using-a-simpler-problem* code (and the values that get you there) is known as the *recursive* case. The recursive case usually contains a call to the very function being executed. This call is known as a *recursive* call.

Most well-formed recursive functions are composed of at least one *base* case and at least one *recursive* case.

16.2 The greatest common divisor

Consider finding the greatest common divisor, the *gcd*, of two numbers. For example, the *gcd* of 30 and 70 is 10, since 10 is the largest number that divides both 30 and 70 evenly. The ancient Greek philosopher Euclid devised a solution for this problem that involves repeated division. The first division divides the two numbers in question, saving the remainder. Now the divisor becomes the dividend and the remainder becomes the divisor. This process is repeated until the remainder becomes zero. At that point, the current divisor is the *gcd*. We can specify this as a recurrence equation, with this last bit about the remainder becoming zero as our base case:

gcd(a,b)	is	b	if a divided by b has a remainder of zero
gcd(a,b)	is	*gcd*(b,a % b)	otherwise

In this recurrence equation, a and b are the dividend and the divisor, respectively. Recall that the modulus operator `%` returns the remainder. Using the recurrence equation as a guide, we can easily implement a function for computing the *gcd* of two numbers.

```
def gcd(dividend,divisor):
    if (dividend % divisor == 0):
        return divisor
    else:
        return gcd(divisor,dividend % divisor)
```

Note that in our implementation of *gcd*, we used more descriptive variables than a and b. We can improve this function further, by noting that the remainder is computed twice, once for the base case and once again to make the recursive call. Rather than compute it twice, we compute it straight off and save it in an aptly named variable:

```
def gcd(dividend,divisor):
    remainder = dividend % divisor
    if (remainder == 0):
```

[3] If one views more multiplications as more complex, then, clearly, computing the factorial of $n-1$ is simpler than computing the factorial of n.

```
        return divisor
    else:
        return gcd(divisor,remainder)
```

Look at how the recursive version turns the *divisor* into the *dividend* by passing *divisor* as the first argument in the recursive call. By the same token, *remainder* becomes *divisor* by nature of being the second argument in the recursive call. To convince one's self that the routine really works, one can modify *gcd* to "visualize" the arguments. On simple way of visualizing the action of a function is to add a print statement:

```
def gcd(dividend,divisor):
    remainder = dividend % divisor
    print("gcd:",dividend,divisor,remainder)
    if (remainder == 0):
        return divisor
    else:
        return gcd(divisor,remainder)
```

After doing so, we get the following output:

```
>>> gcd(66,42)
gcd: 66 42 24
gcd: 42 24 18
gcd: 24 18 6
gcd: 18 6 0
6
```

Note, how the first remainder, 24, keeps shifting to the left. In the first recursive call, the remainder becomes *divisor*, so the 24 shifts one spot to the left. On the second recursive call, the current *divisor*, which is 24, becomes the *dividend*, so the 24 shifts once again to the left.

We can also write a iterative version of *gcd*:

```
def gcd(dividend,divisor):
    while (divisor != 0):
        print(divisor,dividend)
        temp = dividend % divisor
        dividend = divisor
        divisor = temp
    return dividend
```

While the iterative version of factorial was only slightly more complicated than the recursive version, with *gcd*, we start to see more of an advantage using the recursive formulation. For instance, where did the *temp* variable come from and why is it necessary[4]?

16.3 The Fibonacci sequence

A third example of recursion is the computation of the n^{th} Fibonacci number. The Fibonacci series looks like this:

[4]We'll see why the variable *temp* is needed in the next chapter.

```
n            :  0   1   2   3   4   5   6   7   8    ...
Fibonacci(n) :  0   1   1   2   3   5   8  13  21    ...
```

and is found in nature again and again[5]. From this table, we can see that the 7^{th} Fibonacci number is 13. In general, a Fibonacci number is equal to the sum of the previous two Fibonacci numbers. The exceptions are the zeroth and the first Fibonacci numbers which are equal to 0 and 1 respectively. Voila! The recurrence case and the two base cases have jumped right out at us! Here, then, is a recurrence equation which describes the computation of the n^{th} Fibonacci number.

fib(n)	is	0	if n is zero
fib(n)	is	1	if n is one
fib(n)	is	*fib*(n - 1) + *fib*(n - 2)	otherwise

Again, it is straightforward to convert the recurrence equation into a working function:

```
# compute the nth Fibonacci number
# n must be non-negative!

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

Our implementation is straightforward and elegant. Unfortunately, it's horribly inefficient in Python. Unlike our recursive version of *factorial* which recurred about as many times as the size of the number sent to the function, our recursive version of Fibonacci will recur many, many more times than the size of its input. Here's why.

Consider the call to `fib(6)`. Tracing all the recursive calls to *fib*, we get:

```
fib(6) is fib(5) + fib(4)
```

Replacing `fib(5)` with `fib(4) + fib(3)`, we get:

```
fib(6) is fib(4) + fib(3) + fib(4)
```

We can already see a problem, we will compute `fib(4)` twice, once from the original call to `fib(6)` and again when we try to find `fib(5)`. If we write down all the recursive calls generated by `fib(6)` with each recursive call indented from the previous, we get a structure that looks like this:

```
fib(6)
    fib(5)
        fib(4)
```

[5]A pineapple, the golden ratio, a chambered nautilus, etc.

```
            fib(3)
                fib(2)
                    fib(1)
                    fib(0)
                fib(1)
            fib(2)
                fib(1)
                fib(0)
        fib(3)
            fib(2)
                fib(1)
                fib(0)
            fib(1)
    fib(4)
        fib(3)
            fib(2)
                fib(1)
                fib(0)
            fib(1)
        fib(2)
            fib(1)
            fib(0)
```

We would expect, based on how the Fibonacci sequence is generated, to take about six "steps" to calculate `fib(6)`. Instead, ultimately there were 13 calls[6] to either `fib(1)` or `fib(0)`. There was a tremendous amount of duplicated and, therefore, wasted effort. An important part of Computer Science is how to reduce the wasted effort. One way to keep the recursive nature without the penalty of redundant computations is to *cache* previously computed values. Another way is to use an iterative loop:

```
def fib(n):
    previous = 0
    current = 1
    for i in range(0,n,1):
        temp = previous
        previous = current
        current = previous + temp
    return previous
```

Here, the recursive version, for all its faults, is much, much easier to understand. Complications of the iterative version include, why is *previous* returned and not *current*? And where did the variable *temp* come from? In the Chapter 17, we will see a way to combine the clarity of recursion with the efficieny of loops.

16.4 Manipulating arrays and lists with recursion

Recursion, arrays, and lists go hand-in-hand. What follows are a number of recursive patterns involving arrays and lists that you should be able to recognize and implement.

For the following discussion, assume the *ahead* function returns the first item of the given array, while the *atail* function returns an array composed of all the items in the given array except for the first element. If the array is empty, it will have a value of `[]`.

[6] 13 is 7^{th} Fibonacci number and seven is one more than six. A coincidence? Maybe...or maybe not!

By the way, the *ahead* and *atail* functions are easily implemented in Python:

```
def ahead(items): return items[0]
def atail(items): return items[1:]  #slicing, which copies!
```

Note that the *ahead* and *atail* functions are analogous to the *ListHead* and *ListTail* functions of the previous chapter on lists. Note also that we will use the term *collection* to stand in for either a list or an array.

16.5 The *counting* pattern

The *counting* pattern is used to count the number of items in a collection. If the collection is empty, then its count of items is zero. The following function counts and ultimately returns the number of items in an array:

```
def count(items):
    if (items == []): # base case
        return 0
    else:
        return 1 + count(atail(items))
```

The functions works on the observation that if you count the number of items in the tail of a array, then the number of items in the entire array is one plus that number. The extra one accounts for the head item that was not counted when the tail was counted.

The list version is similar:

```
def count(items):
    if (items == None): # base case
        return 0
    else:
        return 1 + count(tail(items))
```

The only code difference in the list version are the use of `None` instead of `[]` to signify empty and *tail* instead of *atail*. However, the performance difference between the two versions can be huge! For a list or an array with n items, the list version will run n times faster. So if your collection holds a million items, the array version will work on the order of one million times slower. This is due to how *atail* works versus how *tail* works, as discussed in the previous chapter. For small numbers of items, however, the array version will run quickly enough.

16.6 The *accumulate* pattern

The *accumulate* pattern is used to combine all the values in a collection. The following function performs a summation of the list values:

```
def sum(items):
    if (items == None): #base case
        return 0
    else:
        return head(items) + sum(tail(items))
```

Note that the only difference between the *count* function and the *sum* function is the recursive case adds in the value of the head item, rather than just counting the head item. That the functions *count* and *sum* look similar is no coincidence. In fact, most recursive functions, especially those working on collections, look very similar to one another.

16.7 The *filtered-count* and *filtered-accumulate* patterns

A variation on the *counting* and *accumulate* patterns involves *filtering*. When filtering, we use an additional `if` statement to decide whether or not we should count the item, or in the case of accumulating, whether or not the item ends up in the accumulation.

Suppose we wish to count the number of even items in a list:

```
def countEvens(items):
    if (items == None): #base case
        return 0
    elif (head(items) % 2 == 0):
        return 1 + countEvens(tail(items))
    else:
        return 0 + countEvens(tail(items))
```

The base case states that there are zero even numbers in an empty list. The first recursive case simply counts the head item if it is even and so adds 1 to the count of even items in the remainder of the list. The second recursive case does not count the head item (because it is not even) and so adds in a 0 to the count of the remaining items. Of course, the last return would almost always be written as:

```
return countEvens(tail(items))
```

As another example of *filtered counting*, we can pass in a value and count how many times that value occurs:

```
def occurrences(target,items):
    if (items == None):
        return 0
    elif (head(items) == target):
        return 1 + occurrences(target,tail(items))
    else:
        return occurrences(target,tail(items))
```

An example of a *filtered-accumulation* would be to sum the even-numbered integers in a list:

```
def sumEvens(items):
    if (items == None):
        return 0
    elif (isEven(head(items))):
        return head(items) + sumEvens(tail(items))
    else:
        return sumEvens(tail(items))
```

where the *isEven* function is defined as:

```
def isEven(x):
    return x % 2 == 0
```

16.8 The *filter* pattern

A special case of a filtered-accumulation is called *filter*. Instead of summing the filtered items (for example), we collect the filtered items into a collection. The new collection is said to be a *reduction* of the original collection.

Suppose we wish to extract the even numbers from a list. The structure of the code looks very much like the *sumEvens* function in the previous section, but instead of adding in the desired item, we join the item to the reduction of the tail of the array:

```
def extractEvens(items):
    if (items == None):
        return None
    elif (isEven(head(items))):
        return join(head(items),extractEvens(tail(items)))
    else:
        return extractEvens(tail(items))
```

Given a list of integers, *extractEvens* returns a (possibly empty) list of the even numbers:

```
>>> a = ListCreate(4,2,5,2,7,0,8,3,7)
>>> extractEvens(a)
[4, [2, [2, [0, [8, None]]]]]

>>> b = ListCreate(1,3,5,7,9)
>>> extractEvens(b)
[]
```

We see again that our list representation depends on arrays.

16.9 The *map* pattern

Mapping is a task closely coupled with that of reduction, but rather than collecting certain items, as with the *filter* pattern, we collect all the items. As we collect, however, we transform each item as we collect it. The basic pattern looks like this:

```
def map(f,items):
    if (items == None):
        return None
    else:
        return join(f(head(items)),map(f,tail(items)))
```

Here, function *f* is used to transform each item in the recursive step.

Suppose we wish to subtract one from each element in an array. First we need a transforming function that reduces its argument by one:

```
def decrement(x): return x - 1
```

Now we can "map" the *decrement* function over an array of numbers:

```
>>> a = ListCreate(4,3,7,2,4,3,1)
>>> map(decrement,a)
[3, [2, [6, [1, [3, [2, [0, None]]]]]]]
```

Both map and filtering (reduce) are used heavily by Google in programming their search strategies.

16.10 The *search* pattern

The *search* pattern is a slight variation of *filtered-counting*. Suppose we wish to see if a value is present in a list. We can use a filtered-counting approach and if the count is greater than zero, we know that the item was indeed in the list.

```
def find(target,items):
    return occurrences(target,items) > 0
```

In this case, *occurrences* helps *find* do its job. We call such functions, naturally, *helper functions*. We can improve the efficiency of *find* by having it perform the search, but short-circuiting the search once the target item is found. We do this by turning the first recursive case into a second base case:

```
def find(target,items):
    if (items == None):
        return False
    elif (head(items) == target):    # short-circuit!
        return True
    else:
        return find(target,tail(items))
```

When the list is empty, we return false because if the item had been list, we would have hit the second base case (and returned true) before hitting the first. If neither base case hits, we simple search the remainder of the list (the recursive case). If the second base case never hits, the first base case eventually will.

16.11 The *shuffle* pattern

Sometimes, we wish to combine two lists. This is easy to do with the append operator:

```
append(list1,list2)
```

This places the first element in the second list after the last element in the first list. However, many times we wish to intersperse the elements from the first list with the elements in the second list. This is known as a *shuffle*, so named since it is similar to shuffling a deck of cards. When a deck of cards is shuffled, the deck is divided in two halves (one half is akin to the first list and the other half is akin to the second list). Next the two halves are interleaved back into a single deck (akin to the resulting third list). Note that while appending is destructive, in that it changes the first list, shuffling is non-destructive, in the neither of the shuffled lists are modified.

We can use recursion to shuffle two lists. If both lists are exactly the same length, the recursive function is easy to implement using the *accumulate* pattern:

```
def shuffle(list1,list2):
    if (list1 == None):
        return None
    else:
        rest = shuffle(tail(list1),tail(list2))
        return join(head(list1),join(head(list2),rest))
```

If *list1* is empty (which means *list2* is empty since they have the same number of elements), the function returns the empty, since shuffling nothing together yields nothing. Otherwise, we shuffle the tails of the lists (the result is stored in the temporary variable *rest*), then join the the first elements of each list to the shuffled remainders.

If you have ever shuffled a deck of cards, you will know that it is rare for the deck to be split exactly in half prior to the shuffle. Can we amend our shuffle function to deal with this problem? Indeed, we can, by simply placing the extra cards (list items) at the end of the shuffle. We don't know which list (*list1* or *list2*) will go empty first, so we test for each list becoming empty in turn:

```
def shuffle2(list1,list2):
    if (list1 == None):
        return list2
    elif (list2 == None):
        return list1
    else:
        rest = shuffle2(tail(list1),tail(list2))
        return join(head(list1),join(head(list2),rest))
```

If either list is empty, we return the other. Only if both are not empty do we execute the recursive case.

One can make shuffling even simpler by manipulating the recursive call that shuffles the remainder of the lists. If we flip the order of the lists in the recursive call, we only need to deal with the first list becoming empty:

```
def shuffle3(list1,list2):
    if (list1 == None):
        return list2
    else:
        return join(head(list1),shuffle3(list2,tail(list1)))
```

Note that in the recursive call, we take the tail of *list1* since we joined the head of *list1* to the resulting shuffle. The base case returns *list2* because if *list1* is empty, the "shuffle" of the remaining elements is just *list2*. Also, even if *list1* and *list2* are both empty, the base case test returns the correct value, `None`.

Finally, note how much simpler it is to shuffle two lists using recursion as compared to shuffling two arrays with loops.

16.12 The *merge* pattern

With the *shuffle* pattern, we always took the head elements from both lists at each step in the shuffling process. Sometimes, we wish to place a constraint of the choice of elements. For example, suppose the two

lists to be combined are sorted and we wish the resulting list to be sorted as well. The following example shows that shuffling does not always work:

```
>>> a = ListCreate(1,4,6,7,8,11)
>>> b = ListCreate(2,3,5,9)

>>> c = shuffle3(a,b)
[1, [2, [4, [3, [6, [5, [7, [9, [8, [11, None]]]]]]]]]]
```

The *merge* pattern is used to ensure the resulting list is sorted and is based upon the *filtered-accumulate* pattern. We only accumulate an item *if* it is the smallest item in the two lists:

```
def merge(list1,list2):
    if (list1 == None):
        return list2
    elif (list2 == None):
        return list1
    elif (head(list1) < head(list2)):
        return join(head(list1),merge(tail(list1),list2))
    else:
        return join(head(list2),merge(list1,tail(list2)))
```

As with *shuffle2*, we don't know which list will become empty first, so we check both in turn.

In the first recursive case, the first element of the first list is smaller than the first element of the second list. So we accumulate the first element of the first list and recur, sending the tail of the first list because we have used/accumulated the head of that list. The second list we pass unmodified, since we did not use/accumulate an element from the second list.

In the second recursive case, we implement the symmetric version of the first recursive case, focusing on the second list rather than the first.

16.13 The *generic merge* pattern

The *merge* function in the previous section hard-wired the comparison operator to `<`. Many times, the elements of the lists to be merged cannot be compared with `<` or, if they can, a different operator, such as `>`, might be desired. The generic merge solves this problem by allowing the caller to pass in a comparison function as a third argument:

```
def genericMerge(list1,list2,pred):
```

where *pred* is the formal parameter that holds a predicate function[7]. Now we replace the `<` in merge with a call to *pred*.

```
def genericMerge(list1,list2,pred):
    if (list1 == None):
        return list2
    elif (list2 == None):
```

[7]Recall that a predicate function returns *True* or *False*.

```
        return list1
    elif (pred(head(list1),head(list2))):
        return join(head(list1),genericMerge(tail(list1),list2,pred))
    else:
        return join(head(list2),genericMerge(list1,tail(list2),pred))
```

The *pred* function, which is passed the two head elements, returns `True`, if the first element should be accumulated, and `False`, otherwise.

We can still use *genericMerge* to merge two sorted lists of numbers (which can be compared with ¡) by using the *operator* module. The *operator* module provides function forms of the operators `+`, `-`, `<`, and so on.

```
>>> import operator
>>> a = ListCreate(1,4,6,7,8,11)
>>> b = ListCreate(2,3,5,9)

>>> c = genericMerge(a,b,operator.lt)
[1, [2, [3, [4, [5, [6, [7, [8, [9, [11, None]]]]]]]]]]
```

The *genericMerge* function is a *generalization* of *merge*. When we generalize a function, we modify it so it can do what it did before plus new things that it could not. Here, we can still have it (genericMerge) do what it (merge) did before, by passing in the less than comparison operator. We can use it to merge two lists sorted in increasing order by passing in the greater than comparison operator.

16.14 The *fossilized* pattern

If a recursive function mistakenly never makes the problem smaller, the problems is said to be *fossilized*. Without ever smaller problems, the base case is never reached and the function recurs[8] forever. This condition is known as an *infinite recursive loop*. Here is an example:

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

Since factorial is solving the same problem over and over, n never gets smaller so it never reaches zero. Fossilizing the problem is a common error made by both novice and expert programmers alike.

16.15 The *bottomless* pattern

Related to the *fossilized* pattern is the *bottomless* pattern. With the *bottomless* pattern, the problem gets smaller, but the base case is never reached. Here is a function that attempts to divide a positive number by two, by seeing how many times you can subtract two from the number:[9]

```
def div2(n):
    if (n == 0):
```

[8] The word is *recurs*, not *recurses*!

[9] Yes, division is just repeated subtraction, just like multiplication is repeated division.

```
        return 0
    else:
        return 1 + div2(n - 2)
```

Things work great for a while:

```
>>> div2(16)
8

>>> div2(6)
3

>>> div2(134)
67
```

But then, something goes terribly wrong:

```
>>> div2(7)
RuntimeError: maximum recursion depth exceeded in cmp
```

What happened? To see, let's *visualize* our function, as we did with the *gcd* function previously, by adding a *print* statement:

```
def div2(n):
    print("div2: n is",n)
    if (n == 0):
        return 0
    else:
        return 1 + div2(n - 2)
```

Now every time the function is called, both originally and recursively, we can see how the value of n is changing:

```
>>div2(7)
div2: n is 7
div2: n is 5
div2: n is 3
div2: n is 1
div2: n is -1
div2: n is -3
div2: n is -5
div2: n is -7
...
RuntimeError: maximum recursion depth exceeded in cmp
```

Now we can see why things went wrong, the value of n skipped over the value of zero and just kept on going. The solution is to change the base case to catch odd (and even) numbers:

```
def div2(n):
    if (n < 2):
        return 0
    else:
        return 1 + div2(n - 2)
```

Remember, when you see a recursion depth exceeded error, you likely have implemented either the fossilized or the bottomless pattern.

16.16 Code

For Linux and Mac users, the code from this chapter can be found here.

For Windows users, look here (you will need to save the file as *recursion.py*).

Chapter 17

Comparing Recursion and Looping

In the previous chapters, we learned about repeatedly evaluating the same code using both recursion and loops. Now we compare and contrast the two techniques by implementing the three mathematical functions from Chapter 16: *factorial*, *fibonacci*, and *gcd*, with loops.

17.1 Factorial

Recall that the factorial function, written recursively, looks like this:

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

We see that is a form of the *accumulate* pattern. So our factorial function using a loop should look something like this:

```
def factorial(n):
    total = ???
    for i in range(???):
        total *= ???
    return total
```

Since we are accumulating a product, total should be initialized to 1.

```
def factorial(n):
    total = 1
    for i in range(???):
        total *= ???
    return total
```

Also, the loop variable should take on all values in the factorial, from 1 to n:

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

```
    total = 1
    for i in range(1,n+1,1):
        total *= ???
    return total
```

Finally, we accumulate i into the total:

```
def factorial(n):
    total = 1
    for i in range(1,n+1,1):
        total *= i
    return total
```

The second argument to range is set to $n + 1$ instead of n because we want n to be included in the total.

Now, compare the loop version to the recursive version. Both contain about the same amount of code, but the recursive version is easier to ascertain as correct.

17.2 The greatest common divisor

Here is a slightly different version of the *gcd* function, built using the following recurrence:

$gcd(a,b)$	is	a	if b is zero
$gcd(a,b)$	is	$gcd(b,a \% b)$	otherwise

The function allows one more recursive call than the 16.2. By doing so, we eliminate the need for the local variable *remainder*. Here is the implementation:

```
def gcd(a,b):
    if (b == 0):
        return a
    else:
        return gcd(b,a % b)
```

Let's turn it into a looping function. This style of recursion doesn't fit any of the patterns we know, so we'll have to start from scratch. We do know that b becomes the new value of a and $a \% b$ becomes the new value of b on every recursive call, so the same thing must happen on every evaluation of the loop body. We stop when b is equal to zero so we should continue looping while b is not equal to zero. These observations lead us to this implementation:

```
def gcd(a,b):
    while (b != 0):
        a = b
        b = a % b
    return a
```

Unfortunately, this implementation is faulty, since we've lost the original value of a by the time we perform the modulus operation. Reversing the two statements in the body of the loop:

```
def gcd(a,b):
    while (b != 0):
        b = a % b
        a = b
    return a
```

is no better; we lose the original value of b by the time we assign it to a. What we need to do is temporarily save the original value of b before we assign a's value. Then we can assign the saved value to a after b has been reassigned:

```
def gcd(a,b):
    while (b != 0):
        temp = b
        b = a % b
        a = temp
    return a
```

Now the function is working correctly. But why did we temporarily need to save a value in the loop version and not in the recursive version? The answer is that the recursive call does not perform any assignments so no values were lost. On the recursive call, new versions of the formal parameters a and b received the computations performed for the function call. The old versions were left untouched.

It should be noted that Python allows simultaneous assignment that obviates the need for the temporary variable:

```
def gcd(a,b):
    while (b != 0):
        a,b = b,a % b
    return a
```

While this code is much shorter, it is a little more difficult to read. Moreover, other common languages do not share this feature and you are left using a temporary variable to preserve needed values when using those languages.

17.3 The Fibonacci sequence

Recall the recursive implementation for finding the n^{th} Fibonacci number:

```
def fib(n):
    if (n < 2):
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

For brevity, we have collapsed the two base cases into a single base case. If n is zero, zero is returned and if n is one, one is returned, as before.

Let's try to implement *fib* using an iterative loop. As before, this doesn't seem to fit a pattern, so we start by reasoning about this. If we let a be the first Fibonacci number, zero, and b be the second Fibonacci number, one, then the third fibonacci number would be $a + b$, which we can save in a variable named c. At

this point, the fourth Fibonacci number would be $b + c$, but since we are using a loop, we need to have the code be the same for each iteration of the loop. If we let a have the value of b and b have the value of c, then the fourth Fibonacci number would be $a + b$ again. This leads to our implementation:

```
def fib(n):
    a = 0    # the first Fibonacci number
    b = 1    # the second Fibonacci number
    for i in range(0,n,1):
        c = a + b
        a = b
        b = c
    return a
```

In the loop body, we see that *fib* is much like *gcd*; the second number becomes the first number and some combination of the first and second number becomes the second number. In the case of *gcd*, the combination was the remainder and, in the case of *fib*, the combination is sum. A rather large question remains, why does the function return a instead of b or c? The reason is, suppose *fib* was called with a value of 0, which is supposed to generate the first Fibonacci number. The loop does not run in this case and the value of a is returned, zero, as required. If a value of 1 is passed to *fib*, then the loop runs exactly once and a gets the original value of b, one. The loop expects and this time, one is returned, as required. So, empirically, it appears that the value of a is the correct choice of return value. As with factorial, hitting on the right way to proceed iteratively is not exactly straightforward, while the recursive version practically wrote itself.

17.4 CHALLENGE: Transforming loops into recursions

To transform an iterative loop into a recursive loop, one first identifies those variables that exist outside the loop but are changing in the loop body; these variable will become formal parameters in the recursive function. For example, the *fib* loop above has three (not two!) variables that are being changed during each iteration of the loop: a, b, and i[1]. The variable c is used only inside the loop and thus is ignored.

Given this, we start out our recursive function like so:

```
def loop(a,b,i):
    ...
```

The loop test becomes an *if* test in the body of the *loop* function:

```
def loop(a,b,i)
    if (i < n):
        ...
    else:
        ...
```

The *if-true* block becomes the recursive call. The arguments to the recursive call encode the updates to the loop variables The *if-false* block becomes the value the loop attempted to calculate:

```
def loop(a,b,i):
    if (i < n):
```

[1]The loop variable is considered a outside variable changed by the loop.

```
            return loop(b,a + b,i + 1)
        else:
            return a
```

Remember, a gets the value of b and b gets the value of c which is $a + b$. Since we are performing recursion with no assignments, we don't need the variable c anymore. The loop variable i is incremented by one each time.

Next, we replace the loop with the the *loop* function in the function containing the original loop. That way, any non-local variables referenced in the test or body of the original loop will be visible to the *loop* function:

```
def fib(n):
    def loop(a,b,i):
        if (i < n):
            return loop(b,a + b,i + 1)
        else:
            return a
    ...
```

Finally, we call the *loop* function with the initial values of the loop variables:

```
def fib(n):
    def loop(a,b,i):
        if (i < n):
            return loop(b,a + b,i + 1)
        else:
            return a
    return loop(0,1,0)
```

Note that this recursive function looks nothing like our original *fib*. However, it suffers from none of the inefficiencies of the original version and yet it performs no assignments.[2] The reason for its efficiency is that it performs the exact calculations and number of calculations as the iterative loop-based function.

For more practice, let's convert the iterative version of *factorial* into a recursive function using this method. We'll again end up with a different recursive function than before. For convenience, here is the loop version:

```
def fact(n):
    total = 1
    for i in range(1,n+1,1):
        total *= i
    return total
```

We start, as before, by working on the *loop* function. In this case, only two variables are changing in the loop: *total* and *i*.

```
def loop(total,i):
    ...
```

[2] A style of programming that uses no assignments is called *functional* programming and is very important in theorizing about the nature of computation.

Next, we write the *if* expression:

```
def loop(total,i):
    if (i < n + 1):
        return loop(total * i,i + 1)
    else:
        return total
```

Next, we embed the *loop* function and call it:

```
def fact(n):
    def loop(total,i):
        if (i < n + 1):
            return loop(total * i,i + 1)
        else:
            return total
    return loop(1,1)
```

The moral of this story is that any iterative loop can be rewritten as a recursion and any recursion can be rewritten as an iterative loop. Moreover, in *good* languages,[3] there is no reason to prefer one way over the other, either in terms of the time it takes or the space used in execution. To reiterate, use a recursion if that makes the implementation more clear, otherwise, use an iterative loop.

17.5 Code

For Linux and Mac users, the code from this chapter can be found here.

For Windows users, look here (you will need to save the file as *recursion-vs-loops.py*).

[3]Unfortunately, Python is not a good language in this regard, but the language *Scheme* is. When the value of a recursive call is immediately returned (i.e., not combined with some other value), a function is said to be *tail recursive*. The Scheme programming language optimizes tail recursive functions so that they are just as efficient as loops both in time and in space utilization.

Chapter 18

Two-dimensional Arrays

Matrices, commonly used in mathematics, are easily represented by two-dimensional arrays in Python. A two-dimensional array is an array whose elements are arrays themselves[1]. Matrices, like tables (the ones that contain data, not the dining room kind), can be divided into rows and columns. A matrix with p rows and q columns is said to be an pxq matrix. First, we delve into creating matrices and then we will study using them to solve some problems.

18.1 Creating a matrix

To make a 3x3 matrix m, one starts by making an array that is 3 elements long. One could simply specify the array directly:

```
m = [None,None,None]    #attempt 1
```

Note we have initialized the array so that each slot in the array holds the value `None`. This approach works fine if the size of the array is small and known in advance. In other cases, we can use a Python trick to quickly construct an array of a given size:

```
m = [None] * 3          #attempt 2
```

We have hardwired the 3 in the above code fragment, but we could use a variable instead, unlike the first attempt. In fact, we use this trick to define a general-purpose function that makes an array of a given size:

```
def createArray(size):
    return [None] * size

m = createArray(3)          # attempt 3
```

We designate this initial array as the *backbone* of the matrix. Pictorially, m looks like this:

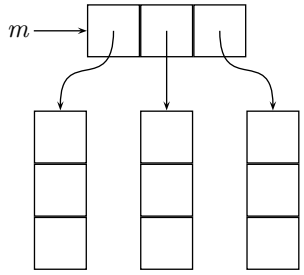
[1] A three-dimensional array would be an array whose elements are arrays whose elements are arrays. Arrays of two dimensions and higher are known as multi-dimensional arrays.

The next step is to assign each slot in the backbone to point to an array of size 3:

```
m = createArray(3)          #attempt 4

m[0] = createArray(3)
m[1] = createArray(3)
m[2] = createArray(3)
```

Now m looks like this:



Of course, creating the matrix this way requires we know the size in advance. If we do not, we can use a loop to set the slots of the backbone:

```
size = 3

m = createArray(size)           #attempt 5

for i in range(0,size,1):
    m[i] = createArray(size)
```

This code works great if the matrix is *square* (i.e. the number of rows equals the number of columns). If not, we replace size in the code above with explicit values for the number of rows and the number of columns:

```
rows = 3
cols = 4

m = createArray(rows)           #attempt 6

for i in range(0,rows,1):
    m[i] = createArray(cols)
```

We can see from this code that the length of the backbone corresponds to the number of rows while the length of the arrays to which the backbone slots point represents the number of columns[2]. Wrapping this code in function that creates a matrix with a given number of rows and columns yields:

```
def createMatrix(rows,cols):
```

[2]This choice of the backbone representing the rows is purely arbitrary, but when this choice is made, the matrix is said to be *row-ordered*. If the opposite choice is made, that the backbone represents the columns, then the matrix is said to be *column-ordered*. Most programming languages provide row-ordered matrices and higher dimensional arrays.

```
    m = createArray(rows)
    for i in range(rows):
        matrix[i] = createArray(cols)
    return m
```

18.2 Retrieving and modifying values in a matrix

To retrieve a value at row r and column c from a matrix m, one uses an expression similar to:

```
value = m[r][c]
```

One can set the value of the element at row r and column c with an expression similar to:

```
m[r][c] = value
```

Because matrices are built from arrays, the first row has index 0 and the first column has index 0 as well. Thus the first value in the first row in matrix m can be found at:

```
m[0][0]
```

Where is the last element in the last row found in a matrix m with x rows and y columns?

18.3 Working with matrices

Matrices are typically processed with with two nested *for* loops. The outer loop runs over the rows, while the inner loop runs over the columns of each row. Here is a generic function for working with a matrix:

```
def processMatrix(m):
    rows = len(m)
    cols = len(m[0])

    for r in range(0,rows,1):
        for c in range(0,cols,1):
            # do something with m[r][c]

    return
```

Note how we retrieve the number of rows by determining the length of the backbone and the number of columns be determining the length of one of the rows.

One useful thing to do with a matrix is visualize it. In other words, we might wish to print out its contents. Using the general purpose template *processMatrix* as a guide, we simply substitute a print statement for the processing comment:

```
def displayMatrix(m):
    rows = len(m)
    cols = len(m[0])
```

```
    for r in range(0,rows,1):
        for c in range(0,cols,1):
            print(m[r][c])

    return
```

If we were to print this matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

we would see this output:

```
1
2
3
4
5
6
```

The good news is we printed every element as desired. The bad news is our output looks nothing like a matrix; every element is printed on a line by itself. We can fix this problem by using the `end` directive in our print statement:

```
print(m[r][c],end=" ")
```

Recall that setting the end character to a space will convert all the newlines into spaces. Our output becomes:

```
1 2 3 4 5 6
```

Better, but we need a newline after each row is printed. The easiest way to do this is to print the newline after the inner loop completes:

```
def displayMatrix(m):
    rows = len(m)
    cols = len(m[0])

    for r in range(0,rows,1):
        for c in range(0,cols,1):
            print(m[r][c],end="")
        print()

    return
```

Now our output becomes:

```
1 2 3
4 5 6
```

which is a reasonable approximation to the way a matrix is supposed to look.

18.4 Reading a 2D array from a file

Reading a 2-dimensional array from a file is similar to reading a simple array. As a reminder, here is a function for doing just that:

```
def readArrayOfIntegers(fileName):
    array = []
    s = Scanner(fileName)
    value = s.readint()
    while (value != ""):
        array.append(value)
        value = s.readint()
    s.close()
    return array
```

This function creates an array, grows it using *append*, and then returns the final version of the array. If the array already exists, things become much simpler; one can read into the array by using a variation of the counting pattern:

```
def readArrayOfIntegers2(array,fileName):
    s = Scanner(fileName)
    for i in range(0,len(array),1):
        array[i] = s.readint()
    s.close()
    return # procedure pattern, return value not needed
```

This second version assumes there is enough data in the file to fill the array. A read error would be generated otherwise. Note that a function that fills a previously constructed array does not require a return value, since the data read from the file is stored in the given array.

If a matrix already exists, then a reading function very similar to *readArrayOfIntegers2* is easily written by replacing the single loop with two nested loops:

```
def readMatrixOfIntegers(matrix,fileName):
    s = Scanner(fileName)
    for r in range(0,len(matrix),1): # loop over rows
        for c in range(0,len(matrix[0]),1): # loop over columns
            matrix[r][c] = s.readint()
    s.close()
    return
```

As a final note, use of the scanner means that the data in the input file does not need to be organized in matrix form; all the matrix elements could be on a single line, for example.

18.5 2D patterns

Matrix patterns are very similar to patterns that loop over simple arrays. For example, here is a filtered-count implementation for a simple array:

```
def countEvens(items):
```

```
    count = 0
    for i in range(0,len(items),1):
        if (isEven(items[i])):
            count += 1
    return count
```

The analogous function for 2D arrays is similar:

```
def countEvens2D(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    count = 0
    for r in range(0,rows,1):
        for c in range(0,cols,1):
            if (isEven(matrix[r][c])):
                count += 1
    return count
```

Indeed, most of the patterns for 2D arrays follow directly from the simple array versions. Here is an instance of the extreme pattern:

```
def minValue2D(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    min = matrix[0][0]
    for r in range(0,rows,1):
        for c in range(0,cols,1):
            if (matrix[r][c] < min):
                min = matrix[r][c]
    return min
```

Finding the extreme index is a bit trickier since one needs to save two indices, the row *and* the column of the extreme value:

```
def minValueIndex2D(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    minRow = 0
    minCol = 0
    for r in range(0,rows,1):
        for c in range(0,cols,1):
            if (matrix[r][c] < matrix[minRow][minCol]):
                minRow = r
                minCol = c
    return minRow,MinCol
```

One can return from the innermost loop, if one knows what one is doing:

```
def find2D(value,matrix):
```

```
    rows = len(matrix)
    cols = len(matrix[0])
    for r in range(0,rows,1):
        for c in range(0,cols,1):
            if (matrix[r][c] == value):
                return True
    return False # this return must be outside the outside loop!
```

Be careful where you place the `return False`, though.

Although the above examples are rather simple, things can get complicated. Suppose you wished to return the index of the row with the largest sum.

```
def largestRow(matrix):
    rows = len(matrix)
    ilargestSum = 0;
    largestSum = sum(matrix[0]) ; sum the 1st row with a helper
    for r in range(1,rows,1):
        s = sum(matrix[r])
        if (s > largestSum):
            ilargestSum = r
            largestSum = s
    return ilargestSum

def sum(items):
    total = 0
    for i in range(0,len(items),1):
        total += items[i]
    return total
```

Curiously, the *largestRow* function does not appear to follow the pattern of other matrix functions; it only has one loop over the rows. However, it calls a helper function from within its loop and that helper function loops over the columns, so we still have our nested loops.

18.6 Simulating 2D arrays with simple arrays

Although Python and most other programming languages allow for two-dimensional arrays, some restricted languages do not. If you are programming in such a language and need a two-dimensional array, not to worry; two-dimensional arrays can be faked with simple arrays. The trick is to convert a two-dimensional address (row index and column index) into a one dimensional address (simple index).

Consider this matrix:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

Using our second attempt at displaying this matrix, we would have obtained the following output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

which looks remarkably like a simple, one-dimensional array! In fact, we can think of this output representing a simple array in which the rows of our two-dimensional array have been stored sequentially.

Now consider extracting the 6 from the 2D version. If our matrix was bound to the variable *m*, the location of the 6 would be:

```
m[1][2]
```

If our simple array version were bound to the variable n, then the 6 can be found at:

```
n[6]
```

since the numbers in the array reflect the indices of the array. It turns out that there is a very simple relationship between the 2D address and the 1D address. Since the rows are laid out sequentially in the 1D case and since the row index in the 2D case is 1, that means there is an entire row preceding the row of interest in the 1D case. Since the rows are 4 columns long, then four elements precede the row of interest. The column of interest has two columns preceding it, so summing the preceding row elements and the preceding column elements yields 6, exactly the address in the the simple array!

In general, the index of a 2D location in a 1D array can be computed by the following function:

```
def simple2DIndex(row,col,rowLength):
    return row * rowLength + col
```

Note that this function must know the length of each row in order to make its computation. Note also that the row length is simply the number of columns. If we wished to display a 1D simulation of a 2D array, our display function might look like this:

```
def displaySimpleMatrix(n,cols):
    rows = len(n) / cols;

    for r in range(0,rows,1):
        for c in range(0,cols,1):
            print(n[simple2DIndex(r,c)])
        print()

    return
```

In general, when simulating a 2D array, one replaces all indexing of the form:

```
m[j][k]
```

with

```
m[j * cols + k]
```

18.7 Ragged 2D arrays

TBW

18.8 Problems

- Modify the *displayMatrix* function to print out the vertical bars that delineate a matrix. That is to say, the display function should output something like:

```
| 1 2 3 |
| 4 5 6 |
```

- Modify the *displayMatrix* function to make the displayed matrix even fancier:

```
+       +
| 1 2 3 |
| 4 5 6 |
+       +
```

- It is relatively easy to transpose a square matrix (a square matrix has the number of rows and the number of columns equal). Here is a such a routine:

```
def transposeSquare(m,size):
    for r in range(????,???,???):
        for c in range(???,???,???):
             m[r][c],m[c][r] = m[???][???],m[???][???]
    return
```