An Introduction to Python

John C. Lusth

July 10, 2011

Contents

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 *lists*.

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 $\backslash n$ 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'`.

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 Lists

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

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

Lists 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 a list:

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

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

Lists can even contain lists!

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

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

2.6 Indexing into Lists

You can pull out an item from a list by using *bracket notation*. With bracket notation, you specify exactly which element (or elements) you wish to extract from the list. This specification is called an *index*. The first element of the list 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 a list:

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

>>> items[0]
'a'

>>> items[1]
True

>>> items[2]
7

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

Note that extracting an item from a list leaves the list 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.

There is a special notation for extracting *all* the elements of a list *except* the first one. This is known as a *slice*:

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

This particular slice (you can slice a list many different ways) says, start extracting at the second item (which has index one) and go to the end. 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
```

As before, the semicolon 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*. Finally 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*.

In general, we will seperate operators from their operands by spaces, tabs, or newlines, collectively known as *whitespace*.[1] It's not necesary 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.

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] 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!

```
>>> 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 and lists:

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

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

The process of joing strings and lists together is known as *concatenation*.

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 lists 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 lists.

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 opertor; 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*.

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).

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

```
>>> "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 `==` and `!=` comparison operators. If an integer is compared with a real with these operators, the integer is converted into a real before the comparison is made. In other cases, comparing different types with `==` will yield a value of `False`. Conversely, comparing different types with `!=` will yield `True` (the exception, as above, being integers compared with reals). If the types match, `==` will yield true only if the values match as well. The operator `!=` behaves accordingly.

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 `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 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 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 a computer scientist since a cardinal rule is to choose good names 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

>>> dots
```

[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

```
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: *m*, *slope*, or `_e_p_o_l_s_` to represent the slope 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 langue) 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 assignnment `d == 0` returns 0, which is, in turned, assigned to c and so on.

6.2 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.

6.2.1 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?

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*?

The value of *alpha* is 13 and the value of *beta* is 5 .

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

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

What are the final values of *alpha* and *beta*?

The value of *alpha* is 42 and the value of *beta* is 13 .

6.2.2 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. What is the value of counter after the following code is executed?

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

Highlight the following line to see the answer:

The answer is

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.

6.2.3 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.

6.2.4 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 indocrinating you on the use of functions. Repeat this ten times:

> We always do three things with functions: define them (we just did that!), call them, 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 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.

6.3 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.

6.4 Assignment and Lists

You can change a particular element of a list 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 list to a 13? The superior student will experiment with the Python interpreter to verify his or her guess.

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`.

7.2 Logical operators

Python has the following logical operators.

`==`	equal to
`!=`	not equal to
`>=`	greater than or equal to
`>`	greater than
`<`	less than
`<=`	less than or equal to
`and`	and
`or`	or

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

The first five operators are used for comparing two things, while the last two operators are the glue that joins up simpler logical expressions into more complex ones.

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*. 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.

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, 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 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.

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 comput 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 call 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 called.

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 called. You must remember this: *the function body is not evaluated until the function is called.*

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 call it:

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

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 values when called, we do not place any values 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, We can improve this 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 an *argument*[1], in this case, the value of x.

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

Note that we have moved 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:

[1] The information that is passed into a function is collectively known as *arguments*.

```
>>> 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

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 can be called
- functions return values

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 return the result of the computation.

Here is an example of the *procedure* pattern:

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

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

[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.

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. A common mistake 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 function you 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:

```
>>> psquare(4)
>>>

>>>print(psquare(6))
None
```

When you do not specify a return value, but you use the return value anyway (as in the printing example), the return value is set to *None*.

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.

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

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 beastie.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!")

main()
```

This version's output is exactly the same as the previous version. We also can see that *main* implements the *procedure patterrn* since it has no explicit return value.

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)
```

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

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

Don't be alarmed if you don't understand what is going on with this program. But you should be able to type this program into a file and then run it. If you do so, you should see the output:

```
    x is 9
```

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 the file yourself.

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

Import statements often are used include the standard Python libraries.

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

Chapter 10

Input and Output

Python distinguishes between reading from the keyboard and writing to the console as opposed to reading and writing to a file. First we cover reading from the keyboard.

10.1 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 (this message is known as a *prompt*) and waits until a response is typed. 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
```

10.2 Writing to the Console

One uses the *print* function to display text on the console: 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.

10.3 Reading from the Command Line

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)

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 a list (because of the square brackets) and that the program file name, as a string, is found in this list.

Command-line arguments

The second way to pass information to a program is through *command-line arguments*. 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)
```

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 lists, 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])
```

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.

10.3.1 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 as numbers.

10.4 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. Suppose we wish to read from a file named *data*. We would open the file like this:

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

The *open* command takes two arguments, the name of the file and the kind of file pointer to return. In this case, we wish for a *reading* file pointer, so we pass the string `"r"`.

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()
```

10.5 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()
```

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 *optimalParameterList* rule tells us that the list of formal parametes 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 eveness 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 **??** and 8.

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 showed 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

Recursion

In Chapter 7, we learned about conditionals. When we combine conditionals 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. To stop the loop, we use a conditional. Recursive loops are often easier to write and understand, as compared to the iterative loops such as *while*s and *for*s, which you will learn about in the next chapter.

Many mathematical functions are easy to implement recursively, so we will start there. Recall that the factorial of a number n is:

$$n! = n * (n-1) * (n-2) * ... * 2 * 1$$

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.

$$0! = 1$$

$$n! = n * (n-1)!$$

otherwise

This second formulation states that the factorial of zero is one [1] and that the factorial of any other (positive) number is obtained by multiplying the that number by the factorial of one less than that number. After some study, you should be able to see that both the first formulation (with the ellipses ...) and this new formulation are equivalent.[2] The second form is particularly well suited for implementation as a computer program:

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

[1] Mathematicians, being an inclusive bunch, like to invite zero to the factorial party.

[2] The first formulation gets the basic idea of factorial across but is not very precise. For example, how would you compute the factorial of three using the first formulation?

Note how the *factorial* function precisely implements the second formulation.

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. and so the second block of the *if* is evaluated. 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 call to factorial, n does have a value of zero, so we can replace `factorial(0)` with its immediate return value of zero:

```
factorial(3) is 3 * 2 * 1 * 1
```

Thus, `factorial(3)` has a value of six:

```
>>> factorial(3)
6
```

as expected.

13.1 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 $n - 1$ is a little bit simpler 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 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.

13.2 The greatest common divisor

Consider finding the greatest common divisor (gcd) of two numbers. One The Ancient Greek Philosopher Euclid devised a solution involving 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

where 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:
        gcd(divisor,dividend % divisor)
```

We can improve this function slightly, by noting that the remainder is computed 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):
        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 yourself that the routine really works, 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
INTEGER: 6
```

Note, how the first remainder, 24, keeps shifting to the left. In the first recursive call, the remainder becomes *second*, 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.

13.3 The Fibonacci sequence

A third example of recursion is the computation of the n^{th} Fibonacci number. The Fibonacci series looks like this:

```
n       0   1   2   3   4   5   6   7   8    ...
Fib(n)  0   1   1   2   3   5   8  13  21    ...
```

and is found in nature again and again.[3] In general, a Fibonacci number is equal to the sum of the previous two Fibonacci numbers. The exceptions are the zeroeth 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 recursive implementation of a function which computes the nth 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. Unlike our recursive version of *factorial* which recurred about as many times as the size of the number sent to the function, our Fibonacci version 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)
```

[3] Pineapples, the golden ratio, chambered nautilus, etc.

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)
            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 to either `fib(1)` or `fib(0)`.[4] 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 is to cache previously computed values.[5]

13.4 Manipulating lists with recursion

Recursion and lists go hand-in-hand. What follows are a number of recursive patterns involving lists that you should be able to recognize and implement.

For the following discussion, assume the *head* function returns the first item in the given list, while the *tail* function returns a list composed of all the items in the given list except for the first element. If the list is empty, it will have a value of `[]`.

By the way, the head and tail functions are easily implemented in Python:

[4] 13 is 7^{th} Fibonacci number. Curious!

[5] Another way is to use an iterative loop. You will learn about loops in the next chapter.

```
def head(items): return items[0]
def tail(items): return items[1:]  #slicing!
```

13.5 The *counting* pattern

The *counting* pattern is used to count the number of items in a collection. If a list is empty, then its count of items is zero. The following function counts and ultimately returns the number of items in the list:

```
def count(items):
    if (items == []):
        return 0
    else:
        return 1 + tail(items)
```

The functions works on the observation that if you count the number of items in the tail of a list, then the number of items in the entire list is one plus that number. The extra one accounts for the head item that was not counted when the tail was counted.

13.6 The *accumulate* pattern

The *accumulate* pattern is used to combine all the values in a list. The following function performs a summation of the list values:

```
def sum(items):
    if (items == []):
        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 function count and the function sum fact look similar is no coincidence. In fact, most recursive functions, especially those working on lists, look very similar to one another.

13.7 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 == []):
        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 as even (because it is not) 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 == []):
        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 == []):
        return []
    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
```

13.8 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 list. The new list is said to be a *reduction* of the original list.

Suppose we wish to extract the even numbers from a list. The code looks very much like the *sumEvens* function in the previous section, but instead of adding in the desired item, we make a list out of it and concatenate to it the reduction of the tail of the list:

```
def extractEvens(items):
    if (items == []):
        return []
    elif (isEven(head(items))):
        return [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:

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

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

13.9 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 == []):
        return []
    else:
        return [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 a list. 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 a list of numbers:

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

13.10 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 == []):
        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.

13.11 The *shuffle* pattern

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

```
list3 = 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).

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 == []):
        return []
    else:
        return [head(list1),head(list2)] + shuffle(tail(list1),tail(list2))
```

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 take the first elements of each list and make a list out of the two elements, then appending the shuffle of the remaining elements to that list.

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. We can be simply placing the extra cards (list items) at the end of the shuffle. We don't know which list1 will go empty first, so we test for each list becoming empty in turn:

```
def shuffle2(list1,list2):
    if (list1 == []):
        return list2
    elif (list2 == []):
        return list1
    else:
        return [head(list1),head(list2)] + shuffle(tail(list1),tail(list2))
```

If either list is empty, we return the other. Only if both are not empty do we execute the recursive case.

13.12 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 = [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 list is sorted and is based upon the *filtered-accumulate* pattern. the merge. We only accumulate an item *if* it is the smallest item in the two lists:

```
def merge(list1,list2):
    if (list1 == []):
        return list2
    elif (list2 == []):
        return list1
    elif (head(list1) < head(list2)):
        return [head(list1)] + merge(tail(list1),list2)
    else:
        return [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.

13.13 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,list1,pred):
```

where *pred* is the formal parameter that holds a predicate function[6]. Now we replace the < in merge with a call to *pred*.

[6]Recall that a predicate function returns *True* or *False*.

```
def genericMerge(list1,list1,pred):
    if (list1 == []):
        return list2
    elif (list2 == []):
        return list1
    elif (pred(head(list1),head(list2))):
        return [head(list1)] + genericMerge(tail(list1),list2,pred)
    else:
        return [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 = [1,4,6,7,8]
>>> b = [2,3,5,9]

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

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 correct comparison operator.

13.14 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[7] 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.

13.15 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:[8].

[7] The word is *recurs*, not *recurses*!

[8] Yes, division is just repeated subtraction, just like multiplication is repeated division

```
def div2(n):
    if (n == 0):
        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 error.

Chapter 14

Loops

In the previous chapter, you learned how recursion can solve a problem by breaking it in to smaller versions of the same problem. Another approach is to use *iterative loops*. In some programming languages, loops are preferred as they use much less computer memory as compared to recursions. In other languages, this is not the case at all. In general, there is no reason to prefer recursions over loops or vice versa, other than this memory issue. Any loop can be written as a recursion and any recursion can be written as a loop. Use a recursion if that makes the implementation more clear, otherwise, use an iterative loop.

The most basic loop structure in Python is the *while* loop, an example of which is:

```
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 assocated 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".

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)
    }
```

14.1 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 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 ommitted 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 intial value and step of the for loop explicitly.

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

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

Recall the items in a list 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 a list 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. Like recursion and lists, loops and lists go very well together. The next sections detail some common loop patterns involving lists.

14.2 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 lists, but many programming languages do not have lists as part of the language; the programmer must supply lists. For this example, assume that the *start* function gets the first item in the given list, returning `None` if there are no items in the list. The *next* function returns the next item in the given list, returning `None` if there are no more items. This `while` loop counts the number of items in the list:

```
count = 0
i = start(items)
while (i != None)
     count += 1
     i = next(items)
```

When the loop finishes, the variable count holds the number of items in the list.

The counting pattern increments a counter everytime the loop body is evaluated.

14.3 The *filtered-count* pattern

A variation on 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 a list:

```
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 list of items since the count is incremented only when the item of interest is even.

14.4 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 a list:

```
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, total is initialized to zero. When accumulating a product, total is initialized to one.

14.5 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 list, 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 accumulationg, *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 list when accumulating a list (see *filtering* below).

14.6 The *search* pattern

The *search* pattern is a slight variation of *filtered-counting*. Suppose we wish to see if a value is present in 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.

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

This pattern is so common, it is often encapsulated in a function. Moreover, we can improve the efficiency by short-circuiting the search. Once the target item is found, there is no need to search the remainder of the list:

```
def find(target,items):
    found = False:
    i = 0
    while (not(found) and i < len(items)):
        if (items[i] == target):
            found = True
        i += 1
    return found
```

We presume the target item is not in the list and as long as it is not found, we continue to search the list. As soon as we find the item, we set the variable found to True and then the loop condition fails, since not true is false.

Experienced programmers would likely define this function to use an immediate return once the target item is found:

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

As a beginning programmer, however, you should avoid returns 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.

14.7 The *filter* pattern

Recall that a special case of a filtered-accumulation is the *filter* pattern. A loop version of filter starts out by initializing an accumulator variable to an empty list. In the loop body, the accumulator variable gets updated with those items from the original list that pass some test.

Suppose we wish to extract the even numbers from a list. Our test, then, is to see if the current element is even. If so, we add it to our growing list:

```
def extractEvens(items):
```

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

Given a list of integers, *extractEvens* returns a (possibly empty) list 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])
[]
```

14.8 The *extreme* pattern

Often, we wish to find the largest or smallest value in a list. 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 list is greater than or equal to zero. If not, as is the case when all values in the list are negative, the code produces an erroneous result of zero as the largest value.

14.9 The *extreme-index* pattern

Sometimes, we wish to find the index of the most extreme value in a list 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*.

14.10 The *shuffle* pattern

Recall, the *shuffle* pattern from the previous chapter. Instead of using recursion, we can use a version of the loop accumulation pattern instead. As before, let's assume the lists are exactly the same size:

```
def shuffle(list1,list2):
    list3 = []
    for i in range(0,len(list1),1):
        list3 = list3 + [list1[i],list2[i]]
    return list3
```

Note how we initialized the resulting list *list3* to the empty list. Then, as we walked the first list, we pulled elements from both lists, adding them into the resulting list.

When we have walked past the end of *list1* is empty, we know we have also walked past the end of *list2*, since the two lists have the same size.

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

```
def shuffle2(list1,list2):
    list3 = []
    if (len(list1) < len(list2)):
        for i in range(0,len(list1),1):
            list3 = list3 + [list1[i],list2[i]]
        return list3 + list2[i:]
    else:
        for i in range(0,len(list2),1):
            list3 = list3 + [list1[i],list2[i]]
        return list3 + list1[i:]
```

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

```
def shuffle3(list1,list2):
    list3 = []
    i = 0
    while (i < len(list2) and i < len(list2)):
        list3 = [list1[i],list2[i]]
        i = i + 1
    ...
```

When the loop ends, one or both of the lists have been exhausted, but we don't know which one or ones. A simple solution is to add both remainders to *list3* and return.

```
def shuffle3(list1,list2):
    list3 = []
    i = 0
    while (i < len(list2) and i < len(list2)):
        list3 = [list1[i],list2[i]]
        i = i + 1
    return list3 + list1[i:] + list2[i:]
```

Suppose *list1* is empty. Then the expression `list1[i:]` will generate the empty list. Adding the empty list to *list3* will have no effect, as desired. The same is true if *list2* (or both *list1* and *list2* are empty).

14.11 The *merge* pattern

We can also merge using a loop. Suppose we have two ordered lists (we will assume increasing order) and we wish to merge them into one ordered list. We start by keeping two index variables, one pointing to the smallest element in *list1* and one pointing to the smallest element in *list2*. Since the lists are ordered, we know the that the smallest elements are at the head of the lists:

```
i = 0  # index variable for list1
j = 0  # index variable for list2
```

Now, we loop, similar to *shuffle3*:

```
while (i < len(list1) and j < len(list2)):
```

Inside the loop, we test to see if the smallest element in list1 is smaller than the smallest element in list2:

```
    if (list1[i] < list2[j]):
```

If it is, we add the element from *list1* to *list3* and increase the index variable i for *list1* since we have 'used up' the value at index i.

```
        list3 = list3 + [list1[i]]
        i = i + 1
```

Otherwise, *list2* must have the smaller element and we do likewise:

```
        list3 = list3 + [list2[j]]
        j = j + 1
```

Finally, when the loop ends (i or j has gotten too large), we add the remainders of both lists to *list3* and return:

```
return list3 + list1[i:] + list2[j:]
```

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

Putting it all together yields:

```
def merge(list1,list2):
    list3 = []
    i = 0
    j = 0
    while (i < len(list1) and j < len(list2)):
        if (list1[i] < list2[j]):
            list3 = list3 + [list1[i]]
            i = i + 1
        else:
            list3 = list3 + [list2[j]]
            j = j + 1
    return list3 + list1[i:] + list2[j:]
```

14.12 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.

14.13 The *missed-condition* pattern

Related to the bottomless pattern of recursive functions is the missed condtion pattern of loops. With missed condition, the index variable is updated, but it is updated in such a way that the loop condition is 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.

Chapter 15

Comparing Recursion and Looping

In the previous two 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 13, *factorial*, *fibonacci*, and *gcd*, with loops.

15.1 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 intialized 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.

15.2 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 other. 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.

15.3 The Fibonacci sequence

Recall the recursive implementation of Fibonacci:

```
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 So let's try to compute 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 orignal value of b, one. The loop exects 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.

15.4 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 an 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 (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 (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 that exact calculations and number of calulations as the 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 reitierate, use a recursion if that makes the implementation more clear, otherwise, use an iterative loop.

[3]Unfortunately, Python is not a good language in this regard, but the language *Scheme* is.

Chapter 16

More on Input

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

16.0.1 Converting command line arguments en mass

Suppose all the command-line arguments are numbers 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():
    total = []
    # start at 1 to skip over program file name
    for i in range(1,len(sys.argv),1):
        num = eval(sys.argv[i])
        total = total + [num]
    return total
```

The accumulator, total, starts out as the empty list. For each element of sys.argv beyond the program file name, we convert it and store the result in num. We then turn that number into a list (by enclosing it in brackets) and then add it to the growing list.

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)

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 list, signifying that the elements are indeed numbers.

16.1 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.

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 beastie.cs.ua.edu/cs150/projects/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.

16.2 Reading Tokens into a List

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 prinicple of Computer Science is *separation of concerns*. To separate the concerns, we have one function read the tokens, storing them into a list (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 = items + [token]
        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 list of tokens that were read:

```
def  countTokens(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.

16.3 Reading Records into a List

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:

```
Smith    President 32  87000
Jones    Assistant 15  99000
Thompson    Hacker  2 147000
```

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.readtoken()
    if name == "":
        return None                   # no record, returning None
    title = s.readtoken()
    service = eval(s.readtoken())
    salary = eval(s.readtoken())
    return [name,title,service,salary]
```

Note that we return either a record as a list or None if no record was read. Since years of service and salary are numbers, we convert them appropriately with *eval*.

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*:

```
function totalPay(fileName):
    s = Scanner(filename)
    total = 0
    record = readRecord(s)
    while (record != None):
        total += record[3]
        record = readRecord(s)
    s.close()
    print("total salaries:",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 != None:
    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.readtoken()
    if name == "":
        return None                 # no record, returning None
    title = s.readtoken()
    service = eval(s.readtoken())
    salary = eval(s.readtoken())

    # 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.

16.4 Creating a List of Records

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

```
def readTable(fileName):
    s = Scanner(fileName)
    table = []
    record = readRecord(s)
    while record != None:
        table += [record]       #brackets around record!
        record = readRecord(s)
    s.close()
```

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 a list, so we can walk it, accumulate items in each record (as we just did with salary), filter it and so on.

16.5 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

`readstring()` read a string - if a string is not pending, ” is returned

`readint()` read an integer - if an integer is not pending, ” is returned

`readfloat()` read a floating point number - if an float is not pending, ” is returned

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

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

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

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

Chapter 17

Arrays and Lists

In this chapter, we will study arrays and lists, two devices used for bringing related bits of data together underneath one roof, so to speak. These devices serve roughly the same purpose but have different behaviors and it is these behaviors that guide a programmer in using arrays for some applications and lists for others. Unfortunately, current Python immplementations suffer from two problems:

1. Python lists appear to be arrays and thus exhibit array behavior.
2. Python arrays are far less useful that Python lists (even though they appear to be the same thing).

This leaves us in a quandry. We can:

- declare that Python lists are arrays and supply a list module that implements the correct behavior for lists.
- pretend that behavior of Python lists are the expected behaviors (even though they are not) and use the less flexible Python arrays.
- supply list and array modules that conform to expected behaviors.

Since this text is not a 'how to program using Python text' but is rather a 'learning Computer Science with Python' text, we are taking the latter approach.

17.0.1 Getting the array and list modules

If you are running Linux on a 32-bit Intel machine, run these commands to download the modules:

```
wget beastie.cs.ua.edu/cs150/array-i386/nakarray.so
wget beastie.cs.ua.edu/cs150/array-i386/naklist.so
```

If you are running Linux on a 64-bit machine, use this command instead:

```
wget beastie.cs.ua.edu/cs150/array-amd64/nakarray.so
wget beastie.cs.ua.edu/cs150/array-amd64/naklist.so
```

Place the *nakarray.so* and *naklist.so* modules in the same directory as your program that uses arrays and lists. One would import these modules with the following lines:

```
from nakarray import *
from naklist import *
```

17.1 A quick introduction to data structures

A *data structure* is simply a collection of bits of information[1] that are somehow glued together into a single whole. Each of these bits can be be accessed individually. Usually, the bits are somehow related, so the data structure is a convenient way to keep all these related bits nicely packaged together.

At its most basic, a data structure supports the following actions:

- creating the structure
- putting data into the structure
- taking data out of the structure

We will study these actions with an eye to how long each action is expected to take.

There are five main *data structures* built into Python: *tuples*, *lists*, *arrays*, *dictionaries*, and *objects*. As this text focuses on the language features of Python that are common to other programming languages, we will forgo discussion of tuples and dictionaries. Instead, in this chapter, we will focus on arrays and lists. In a later chapter, we will discuss objects.

17.2 Arrays

An array is a data structure with the property that each individual piece of data can be accessed just as quickly as any of the others.

17.2.1 Creating arrays

To create an array, one uses the *Array* function:

```
a = Array(10)
```

The function call `Array(10)` creates an array with room for ten items or *elements*. The variable *a* is created and set to point to the array.

Typically array creation can take anywhere from *constant* time (if elements are not initialized) or *linear* time (if they are). What do we mean by constant time? We mean, in this particular case, the amount of time it takes to create an array of 10 elements takes the same abount of time it takes to create an array of 1,000,000 elements. In other words, the time it takes to create an array is independent of the size of the array. By the same token, linear time means the time it takes to create an array is proportional to the size of the array. If array creation takes linear time, we would expect the time it takes to create a 1,000,000 element array would be, roughly, 100,000,000 times longer than the time to create a 10 element array.

At the end of the chapter, we will attempt to figure out whether array creation takes constant or linear time.

[1] Bits in the informal sense, not zeros and ones.

17.2.2 Setting and getting array elements

Most data structures allow you to both add and remove data. For arrays, adding data means to *set* the value of an array element, while removing data means to *get* the value of an array element. It does not mean that elements themselves are being created and destroyed (although this can be the case in other data structures).

To set an individual element, one uses *square bracket notation*. For example, here's how to set the first item in the array

```
>>> a[0] = 42
```

The number between the square brackets is known as an *index*. Note that the index of the first element is zero. This means that the index of the second element is one, and so on. [2] This business of starting the indices at zero is known as *zero-based* counting.

Once an element has been set, you can retrieve it from the array using a similar notation. The following code illustrates the retrieval of the first (index zero) element.

```
>>> e = a[0]

>>> e
42
```

Of course, there is no need to assign the value of an array element to a variable; you can access it directly:

```
>>> a[0]
42
```

The elements of an array are initialized to `None`, so if you access an element that hasn't been set, you get `None` as a result:

```
>>> print(a[1])
None

>>>print(a)
[42, None, None, None, None, None, None, None, None, None]
```

To find out the number of elements in an array, you can use the *len* function:

```
>>> len(a)
10
```

Because of zero-based counting, the index of the last element is always one less the number of elements in the array. For a ten element array then, the last legal index is nine:

[2] For reasons too complicated to go into here, Computer Scientists love to start counting from zero, as opposed to normal people who generally start counting from one.

```
>>> a[9] = 13

>>>print(a)
[42, None, None, None, None, None, None, None, None, 13]
```

Trying to access an element beyond the 9^{th} results in an error:

```
>>> a[10] = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  IndexError: Index '10' is out of range.
```

17.2.3 Limitations on arrays

Unlike built-in Python strings, arrays, and lists, the *nakarray* module does not allow:

- 'slicing' an array.
- negative indices
- concatenation two arrays using the plus operator

However, unlike built-in Python arrays, which are *homogeneous*, arrays made with the *nakarray* module are *heterogeneous*. Homogeneous, in Computer Science speak, means "all of one type" so a homogeneous array can only hold all integers or all strings, but not both. Heterogeneous arrays can hold a mixture of types.

17.3 Lists

A list is a data structure that has the property that it can be lengthened by placing an element at the beginning in constant time. Here, constant time means adding an element to the front of a list is independent of the number of elements already in the list. Depending on the list implementation, adding an element at the end of a list can take constant or linear time. However, adding an element to the middle of a list takes linear time. In contrast, adding an element to an array is not even allowed.

Retrieving the value of an element takes constant time, if the element is at the beginning of a list, constant or linear time, if the element is at the end of the list, and linear time, if the element is in the middle of the list.

17.3.1 List creation

One creates a list by calling the *List* function.

```
>>> a = List()

>>> print(a)
[ ]
```

The *List* function creates an empty list and this takes a constant amount of time (since the number of elements created is always zero).

```
>>> a.append(64.0)
>>> a.append("hello")
>>> a.append(True)
>>> a.prepend(42)
```

The *append* method adds an element to the end of the list while the prepend method adds the element to the beginning of the list. Once the list is created, however, accessing and setting elements in a list proceeds exactly in the same way as with arrays:

```
>>> a[2]
'hello'

>>> a[3] = 13;

>>> a[3]
13
```

Unlike arrays in which every element can be accessed in the same amount of time, accessing elements further and further towards the back of the list takes more and more time.

We can create a larger list from two smaller lists using the plus operator. This process is known as *list concatenation*.

```
>>> b = a.prepend("apple")
>>> c = a.append("pear")

>>> print(a)
[42, 64.0, 'hello', True]

>>> print(b)
['apple', 0, 42, 64.0, 'hello', True]

>>> print(c)
[42, 64.0, 'hello', True, 'pear', 13]
```

Note that creating a new list leaves the original lists unmodified.

```
>>> a
[42, 64.0, 'hello', True]

>>> c
[42, 64.0, 'hello', True, 'pear']
```

Notice in both instances, we used the plus operator and we enclosed the element to be added in a list. We can conclude that the plus operator can be used to As with joining, the original list is unchanged.

17.4 Mixing arrays and lists

You can concatenate lists together with arrays. If you do so, an array results:

```
>>> array(1,2,3) + list(4,5,6);
ARRAY: [1,2,3,4,5,6]
```

If you join an element to an array, you get this mad-scientist amalgamation of list and array:

```
>>> var d = "zero" join array("one","two","three");
LIST: ("zero" # ["one","two","three"])
```

While the result looks rather strange, you can access each each element as if it were a pure list or pure array:

```
>>> d[0];
STRING: "zero"

>>> d[1];
STRING: "one"
```

17.5 Shallow versus deep copies

Consider the following interaction:

```
>>> var a = list(13,21,34);

>>> var b = 0 join a;

>>> a;
LIST: (13,21,34)

>>> b;
LIST: (0,13,21,34)
```

From the previous discussion, we know that the value of variable *a* is unchanged by the *join* operation that was used to create variable *b*'s value. Look what happens when we change *b*'s first element:

```
>>> b[0] = "zero";

>>> b;
LIST: ("zero",13,21,34)

>>> a;
LIST: (13,21,34)
```

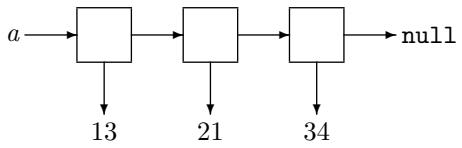
As we expected, we see that *b*'s value changes while *a*'s does not. What happens when we change the second element of *b*?

```
>>> b[1] = :apple;

>>> b;
LIST: ("zero",:apple,21,34)
```

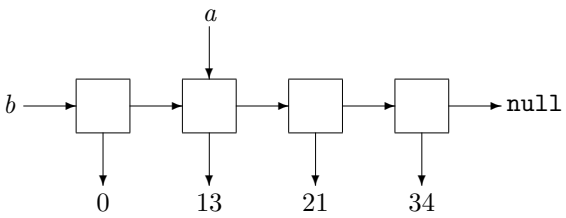
```
>>> a;
LIST: (:apple,21,34)
```

Surprisingly, our change to b was reflected in a as well! To understand why, let's look at a picture of a just after it is created:

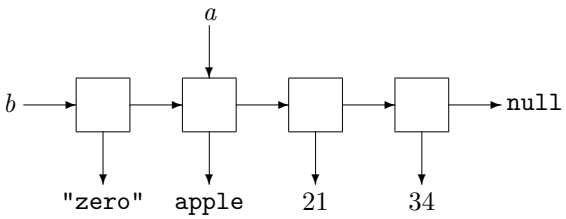


The above image is known as a *box-and-pointer* diagram. The boxes denote *objects* that hold two pieces of information: how to get the next object (the horizontal arrow) and how to get to a value (the vertical arrow). The chain of objects is known as the *backbone* of the list. The variable a points to the first object. When we wish to display the list, we walk from object to object, following the horizontal arrows, and displaying the value associated with each object in turn. The symbol
verb!null! is used to indicate that there are no more following objects.

When we create b, this becomes the state of the world:



We now see that the *join* operator creates a single object that glues a value onto an existing list. We also see why list a appears to be unchanged by the creation of b and by the changing of the first element of b, but is changed when the second element of b is changed:



The reason for this behavior is efficiency. The construction of list b took very little time since only one object was created. Consider using *join* to glue a value onto a list of one million elements. Without this efficiency trick, adding the new value might take a very long time indeed.

Suppose we wish to make a list b that is completely independent of list a. Here is a function named *join** that attempts that very feat:

```
function join*(value,items)
```

```
{
if (items == :null)
    {
    value join :null;
    }
else
    {
    value join (head(items) join* tail(items));
    }
}
```

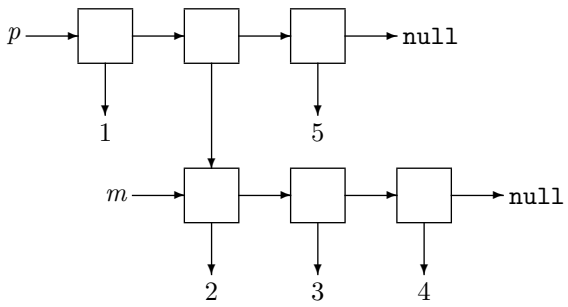
This function walks along the list, rejoining up elements with new objects. When we are done, b will have its own backbone separate from a's.

The problem is that the copy made by *join** is a *shallow* copy. That is, the incoming list was copied, but the elements were not. In the case of a list of symbols, the elements do not need to be copied, but that is not always the case. Consider a list that has a list as one of its elements:

```
var m = list(2,3,4);

>>> var p = list(1,M,5);
LIST: (1,(2,3,4),5)
```

At this point, the second element of list p is list m:



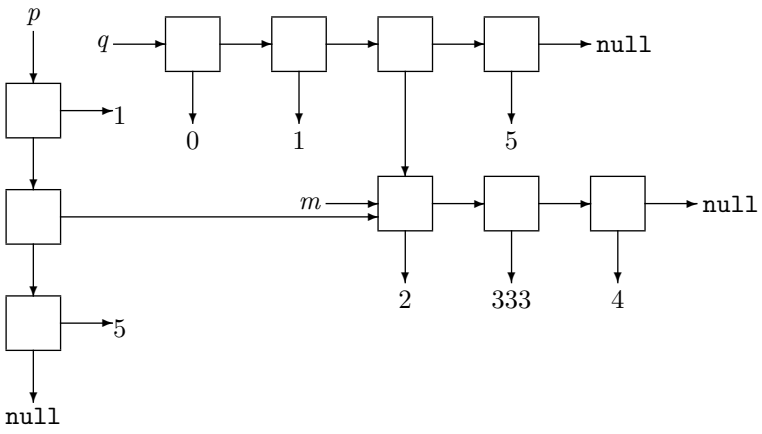
Now, let's add the number 0 to the front of list p using *join**.

```
>>> var q = 0 join* p;       // note, join*, not join
LIST: (0,1,(2,3,4),5)
```

Now we change the second element of third element of q:

```
q[2][1] = 333;
```

generating this situation:



From the illustration, we can see that the change, made through q, will modified both m and the display of p.

To make p, q, and m all independent, one needs to make a *deep* copy, where the elements of the list are copied and the elements of the elements are copied, and so on.

Deep copies are so difficult to do correctly, almost no modern language has deep copying built in.

17.6 Changing the tail of a list

One can use the *tail=* function to change the tail of a list:

```
var a = list(1,2,3);

a tail= list(222,333,444);

>>> a;
LIST: (1,222,333,444);
```

This exchange essentially removes all but the first element of list a, replacing those elements with the elemints 222, 333, and 444.

You can use *tail=* on any kind of list, even those produced by *tail*:

```
var b = list(1,2,3);

tail(b) tail= list(333);

>>> b;
LIST: (1,2,333);
```

Here, the first two elements of list b are preserved.

17.7 Inserting into the middle of a list

One uses a combination of *join*, *tail*, and *tail=* to insert into the middle of a list. Here's how to insert an item after the first element:

```
var c = list(1,3,4);

c tail= (2 join tail(c));

>>> c;
LIST: (1,2,3,4);
```

The parentheses are needed to insure that the tail of c is not set to the number 3 and the result joined with the tail of c. This method of insertion changes the original list.

It is also possible to make a general-purpose insertion function. The following implementation inserts an element into a list of items at the given index (using zero-based counting):

```
function insert(element,items,index)
    {
    if (index == 0)
        {
        element join items; //put the element at the front
        }
    else
        {
        items[0] join insert(element,tail(items),index - 1);
        }
    }
```

Here is the function in action:

```
var d = list(1,3,4);
var e = insert(2,d,1); //insert 2 at index 1 of list d

>>> d;
LIST: (1,3,4);

>>> e;
LIST: (1,2,3,4);
```

Note that this *insert* function does a combination of a shallow copy of the elements preceding the inserted element and creates a sharing of the elements after the inserted element. Thus d is unchanged by this operation (but some changes to e can affect d).

17.8 Objects

Objects, in the simplest view, are like lists and arrays except elements are accessed or updated by name rather than by index. Objects are discussed in the next chapter.

Chapter 18

Sorting

Sorted Tables

Recall that a table is a list of records where each record is a list of the fields incorporating the record.

Sometimes, you need to merge two sorted tables into one table that remains sorted. First, you have to decide which field is used for the sorting. In our example, the records in the data file could be sorted on NAME or on SALARY or any other field.

Suppose we had two data files that are sorted on SALARY, *salaries.1* and *salaries.2*. We wish to merge the data in both files, printing out the merged data, again in sorted order.

First, we need to read the data into tables:

```
table1 = readTable("salaries.1")
table2 = readTable("salaries.2")
```

Our strategy is to compare the first unaccumulated record in *table1* to the first unaccumulated record in *table2*. Let's call these records *r1* and *r2*, respectively. If the salary of *r1* is less than that of *r2*, we accumulate *r1*. Otherwise we accumulate *r2*. We will repeat this process using a loop.

It is clear we need two variables, the first points to the index of the first unaccumulated record in the table1, while the second variable points to the first unaccumulated record in the second table. We start out both variables at zero, meaning no records have been accumulated yet:

```
index1 = 0
index2 = 0
```

How do we know when to stop accumulating? When we run out of records to compare. This happens when *index1* has passed the index of the last record in *table1* or *index2* has passed the index of the last record in *table2*. We reverse that logic for a while loop, because it runs while the test condition is true. The reversed logic is "as long as index1 has not passed the *index* of the last record in *table1* AND *index2* has not passed the index of the last record in *table2*".

```
total = []
while (index1 < len(table1) and index2 < len(table2):
    r1 = table1[index1]
    r2 = table2[index2]
    ...
```

We also must advance *index1* and *index2* to that the loop will finally end. When do we advance *index1*? When we accumulate a record from *table1*. When do we advance *index2*? Likewise, when we accumulate a record from *table2*.

```
total = []
while (index1 < len(table1) and index2 < len(table2):
    r1 = table1[index1]
    r2 = table2[index2]
    if (r1[SALARY] < r2[SALARY]):
       total = total + [r1]
       index1 += 1
    else:
       total = total + [r2]
       index2 += 1
```

When will this loop end? When one of the indices gets too high[1]. This means we will have accumulated all the records from one of the tables, but we don't know which one. So, we add two more loops to accumulate any left over records:

```
for i in range(index1,len(table1),1):
    total = total + [table1[i]]

for i in range(index2,len(table2),1):
    total = total + [table2[i]]
```

Finally, we encapsulate all of our merging code into a function, passing in the index of the field that was used to sort the data. This field is known as the *key*:

```
def merge(table1,table2,key):
    total = []
    while (index1 < len(table1) and index2 < len(table2):
        r1 = table1[index1]
        r2 = table2[index2]
        if (r1[key] < r2[key]):
           total = total + [r1]
           index1 += 1
        else:
           total = total + [r2]
           index2 += 1

    for i in range(index1,len(table1),1):
        total = total + [table1[i]]

    for i in range(index2,len(table2),1):
        total = total + [table2[i]]
```

Finally, we define a main function to tie it all together:

```
def main():
```

[1] Only one will be too high. Why is that?

```
table1 = readTable("salaries.1")
table2 = readTable("salaries.2")
mergedTable = merge(table1,table2,SALARY) #SALARY is the key
printTable(mergedTable)
```

Notice how the main function follows the standard main pattern:

- get the data
- process the data
- write the result

18.1 Merge sort