How to Think Like a Computer Scientist
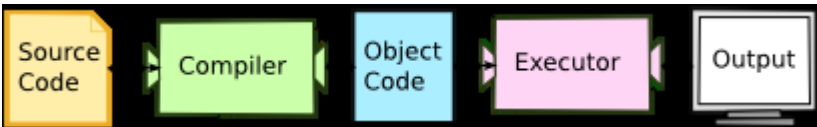
Learning with Python

2nd Edition (Using Python 2.x)

1. The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specific y computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives.

Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skil for a computer scientist is problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excel ent opportunity to practice problem-solving skil s. That's why this chapter is cal ed, The way of the program.

On one level, you wil be learning to program, a useful skil by itself. On another level, you wil use programming as a means to an end. As we go along, that end wil become clearer.

1.1. The Python programming language

The programming language you wil be learning is Python. Python is an example of a high-level language; other high-level languages you might have heard of are C++, PHP, and Java.

As you might infer from the name high-level language, there are also low-level languages, sometimes referred to as machine languages or assembly languages. Loosely speaking, computers can only execute

programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a smal disadvantage of high-level languages.

But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, almost al programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

Two kinds of programs process high-level languages into low-level languages: interpreters and compilers. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is cal ed the source code, and the translated program is cal ed the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.

Many modern languages use both processes. They are first compiled into a lower level language, cal ed byte code, and then interpreted by a program cal ed a virtual machine. Python uses both processes, but

because of the way programmers interact with it, it is usual y considered an interpreted language.

There are two ways to use the Python interpreter: shell mode and script mode. In shel mode, you type Python statements into the Python shell and the interpreter immediately prints the result: $ python

Python 2.5.1 (r251:54863, May 2 2007, 16:56:35)

[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> print 1 + 1

2

The first line of this example is the command that starts the Python interpreter at a Unix command prompt. The next three lines are messages from the interpreter. The fourth line starts with >>>, which is the Python prompt. The interpreter uses the prompt to indicate that it is ready for instructions. We typed print 1 + 1, and the interpreter replied 2.

Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file.

Such a file is cal ed a script. For example, we used a text editor to create a file named firstprogram.py with the fol owing contents:

print 1 + 1

By convention, files that contain Python programs have names that end with .py.

To execute the program, we have to tel the interpreter the name of the script:

$ python firstprogram.py

These examples show Python being run from a Unix command line. In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

The examples in this book use both the Python interpreter and scripts. You wil be able to tel which is intended since shel mode examples wil always start with the Python prompt.

Working in shel mode is convenient for testing short bits of code because you get immediate feedback.

Think of it as scratch paper used to help you work out problems. Anything longer than a few lines should be put into a script.

1.2. What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

input

Get data from the keyboard, a file, or some other device.

output

Display data on the screen or send data to a file or other device.

math

Perform basic mathematical operations like addition and multiplication.

conditional execution

Check for certain conditions and execute the appropriate sequence of statements.

repetition

Perform some action repeatedly, usual y with some variation.

Believe it or not, that's pretty much al there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look more or less like these. Thus, we can describe programming as the process of breaking a large, complex task into smal er and smal er subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

That may be a little vague, but we wil come back to this topic later when we talk about algorithms.

1.3. What is debugging?

Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are cal ed bugs and the process of tracking them down and correcting them is cal ed debugging.

Three kinds of errors can occur in a program: [syntax errors](), [runtime errors](), and [semantic errors.]() It is useful to distinguish between them in order to track them down more quickly.

1.4. Syntax errors

Python can only execute a program if the program is syntactical y correct; otherwise, the process fails and returns an error message.

syntax refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python wil print an error message and quit, and you wil not be able to run your program. During the first few weeks of your programming career, you wil probably spend a lot of time tracking down syntax errors. As you gain experience, though, you wil make fewer errors and find them faster.

1.5. Runtime errors

The second type of error is a runtime error, so cal ed because the error does not appear until you run the program. These errors are also cal ed exceptions because they usual y indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you wil see in the first few chapters, so it might be a while before you encounter one.

1.6. Semantic errors

The third type of error is the semantic error. If there is a semantic error in your program, it wil run successful y, in the sense that the computer wil not generate any error messages, but it wil not do the right thing. It wil do something else. Specifical y, it wil do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires

you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.7. Experimental debugging

One of the most important skil s you wil acquire is debugging. Although it can be frustrating, debugging is one of the most intel ectual y rich, chal enging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, When you have eliminated the impossible, whatever remains, however improbable, must be the truth. (A. Conan Doyle, The Sign of Four) For some people, programming and debugging are the same thing. That is, programming is the process of gradual y debugging a program until it does what you want. The idea is that you should start with a program that does something and make smal modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system kernel that contains mil ions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, one of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux (The Linux Users' Guide Beta Version 1).

Later chapters wil make more suggestions about debugging and other programming practices.

1.8. Formal and natural languages

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved natural y.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express

computations.

Formal languages tend to have strict rules about syntax. For example, $3+3=6$ is a syntactical y correct mathematical statement, but $3=+6\$$ is not. $H2O$ is a syntactical y correct chemical name, but $2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3=+6\$$ is that $\$$

is not a legal token in mathematics (at least as far as we know). Similarly, $2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax rule pertains to the structure of a statement— that is, the way the tokens are arranged. The statement $3=+6\$$ is structural y il egal because you can't place a plus sign

immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is cal ed parsing.

For example, when you hear the sentence, The other shoe fel , you understand that the other shoe is the subject and fel is the verb. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a shoe is and what it means to fal , you wil understand the general implication of this sentence.

Although formal and natural languages have many features in common — tokens, structure, syntax, and semantics — there are many differences:

ambiguity

Natural languages are ful of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy

In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness

Natural languages are ful of idiom and metaphor. If someone says, The other shoe fel , there is probably no shoe and nothing fal ing.

Formal languages mean exactly what they say.

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry

Words are used for their sounds as wel as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose

The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but stil often ambiguous.

Programs

The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usual y not a good idea to read from top to bottom, left to right.

Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure.

Final y, the details matter. Little things like spel ing errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.9. The first program

Traditional y, the first program written in a new language is cal ed Hel o, World! because al it does is display the words, Hel o, World! In Python, it looks like this:

print "Hello, World!"

This is an example of a print statement, which doesn't actual y print anything on paper. It displays a value on the screen. In this case, the result is the words

Hello, World!

The quotation marks in the program mark the beginning and end of the value; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the Hel o, World!

program. By this standard, Python does about as wel as possible.

1.10. Glossary

algorithm

A general process for solving a category of problems.

bug

An error in a program.

byte code

An intermediate language between source code and object code. Many modern languages first compile source code into byte code and then interpret the byte code with a program cal ed a virtual machine.

compile

To translate a program written in a high-level language into a low-level language al at once, in preparation for later execution.

debugging

The process of finding and removing any of the three kinds of programming errors.

exception

Another name for a runtime error.

executable

Another name for object code that is ready to be executed.

formal language

Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; al programming languages are formal languages.

high-level language

A programming language like Python that is designed to be easy for humans to read and write.

interpret

To execute a program in a high-level language by translating it one line at a time.

low-level language

A programming language that is designed to be easy for a computer to execute; also cal ed machine language or assembly language.

natural language

Any one of the languages that people speak that evolved natural y.

object code

The output of the compiler after it translates the program.

parse

To examine a program and analyze the syntactic structure.

portability

A property of a program that can run on more than one kind of computer.

print statement

An instruction that causes the Python interpreter to display a value on the screen.

problem solving

The process of formulating a problem, finding a solution, and expressing the solution.

program

a sequence of instructions that specifies to a computer actions and computations to be performed.

Python shell

An interactive user interface to the Python interpreter. The user of a Python shel types commands at the prompt (>>>), and presses the return key to send these commands immediately to the interpreter for processing.

runtime error

An error that does not occur until the program has started to execute but that prevents the program from continuing.

script

A program stored in a file (usual y one that wil be interpreted).

semantic error

An error in a program that makes it do something other than what the programmer intended.

semantics

The meaning of a program.

source code

A program in a high-level language before being compiled.

syntax

The structure of a program.

syntax error

An error in a program that makes it impossible to parse — and therefore impossible to interpret.

token

One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

1.11. Exercises

1. Write an English sentence with understandable semantics but incorrect syntax. Write another English sentence which has correct syntax but has semantic errors.

2. Start a Python shel . Type 1 + 2 and then hit return. Python evaluates this expression, prints the result, and then prints another prompt. * is the multiplication operator, and ** is the exponentiation operator. Experiment by entering different expressions and recording what is printed by the Python interpreter. What happens if you use the / operator? Are the results what you expect? Explain.

3. Type 1 2 and then hit return. Python tries to evaluate the expression, but it can't because the expression is not syntactical y legal. Instead, it prints the error message:

File "<stdin>", line 1

1 2

^

SyntaxError: invalid syntax

In many cases, Python indicates where the syntax error occurred, but it is not always right, and it doesn't give you much information about what is wrong.

So, for the most part, the burden is on you to learn the syntax rules.

In this case, Python is complaining because there is no operator between the numbers.

See if you can find a few more examples of things that wil produce error messages when you enter them at the Python prompt. Write down what you enter at the prompt and the last line of the error message that Python reports back to you.

4. Type *print 'hello'*. Python executes this statement, which has the effect of printing the letters h-e-l-l-o. Notice that the quotation marks that you used to enclose the string are not part of the output. Now type "hello" and describe your result. Make note of when you see the quotation marks and when you don't.

5. Type print cheese without the quotation marks. The output wil look something like this: Traceback (most recent call last):

File "<stdin>", line 1, in ?

NameError: name 'cheese' is not defined

This is a run-time error; specifical y, it is a NameError, and even more specifical y, it is an error because the name *cheese* is not defined. If you don't know what that means yet, you wil soon.

6. Type 'This is a test...' at the Python prompt and hit enter. Record what happens.

Now create a python script named test1.py with the fol owing contents (be sure to save it before you try to run it):

'This is a test...'

What happens when you run this script? Now change the contents to:

print 'This is a test...'

and run it again.

What happened this time?

Whenever an expression is typed at the Python prompt, it is evaluated and the result is printed on the line below. 'This is a test...' is an expression, which evaluates to 'This is a test...'

(just like the expression 42 evaluates to 42). In a script, however, evaluations of expressions are not sent to the program output, so it is necessary to explicitly print them.

2. Variables, expressions and statements

2.1. Values and data types

A value is one of the fundamental things — like a letter or a number — that a program manipulates. The values we have seen so far are 2 (the result when we added 1 + 1), and "Hello, World!".

These values belong to different data types: 2 is an integer, and "Hello, World!" is a string, so-cal ed because it contains a string of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

>>> print 4

4

If you are not sure what type a value has, the interpreter can tel you.

>>> type("Hello, World!")

>>> type(17)

<type 'int'>

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type cal ed float, because these numbers are represented in a format cal ed floating-point.

>>> type(3.2)

<type 'float'>

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

>>> type("17")

<type 'str'>

>>> type("3.2")

<type 'str'>

They're strings.

Strings in Python can be enclosed in either single quotes (') or double quotes ("):

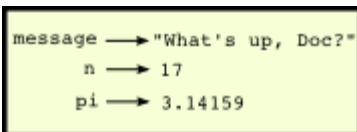**>>>** type('This is a string.')

<type 'str'>

>>> type("And so is this.")

Double quoted strings can contain single quotes inside them, as in "Bruce's beard", and single quoted strings can have double quotes inside them, as in 'The knights who say "Ni!"'.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

>>> print 1,000,000

1 0 0

```
message ——→ "What's up, Doc?"
       n ——→ 17
      pi ——→ 3.14159
```

Wel , that's not what we expected at al ! Python interprets 1,000,000 as a list of three items to be printed. So remember not to put commas in your integers.

2.2. Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

The assignment statement creates new variables and gives them values:

>>> message = "What's up, Doc?"

>>> n = 17

>>> pi = 3.14159

This example makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named message. The second

gives the integer 17 to n, and the third gives the floating-point number 3.14159 to pi.

The assignment operator, =, should not be confused with an equals sign (even though it uses the same character). Assignment operators link a name, on the left hand side of the operator, with a value, on the right hand side. This is why you wil get an error if you enter:

>>> 17 = n

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is cal ed a state diagram because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the assignment statements:

The print statement also works with variables.

>>> print message

What's up, Doc?

>>> print n

17

>>> print pi

3.14159

In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

>>> type(message)

>>> type(n)

<type 'int'>

>>> type(pi)

<type 'float'>

The type of a variable is the type of the value it refers to.

2.3. Variable names and keywords

Programmers general y choose names for their variables that are meaningful — they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. Bruce and bruce are different variables.

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or price_of_tea_in_china.

If you give a variable an il egal name, you get a syntax error:

>>> 76trombones = "big parade"

SyntaxError: invalid syntax

>>> more$ = 1000000

SyntaxError: invalid syntax

>>> class = "Computer Science 101"

SyntaxError: invalid syntax

76trombones is il egal because it does not begin with a letter. more$ is il egal because it contains an il egal character, the dol ar sign. But what's wrong with class?

It turns out that class is one of the Python keywords. Keywords define the language's rules and structure, and they cannot be used as variable names.

Python has thirty-one keywords:

and

as

assert

break

class

continue

def

del

elif

else

except

exec

final y

for

from

global

if

import

in

is

lambda

not

or

pass

print

raise

return

try

while

with

yield

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

2.4. Statements

A statement is an instruction that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.

A script usual y contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

print 1

x = 2

print x

produces the output:

12

Again, the assignment statement produces no output.

2.5. Evaluating expressions

An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter evaluates it and displays the result:

>>> 1 + 1

2

The evaluation of an expression produces a value, which is why expressions can appear on the right hand side of assignment

statements. A value al by itself is a simple expression, and so is a variable.

>>> 17

17

>>> x

2

Confusingly, evaluating an expression is not quite the same thing as printing a value.

>>> message = "What's up, Doc?"

>>> message

"What's up, Doc?"

>>> print message

What's up, Doc?

When the Python shel displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But the print statement prints the value of the expression, which in this case is the contents of the string.

In a script, an expression al by itself is a legal statement, but it doesn't do anything. The script 17

3.2

"Hello, World!"

1 + 1

produces no output at al . How would you change the script to display the values of these four expressions?

2.6. Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are cal ed operands.

The fol owing are al legal Python expressions whose meaning is more or less clear: 20+32 hour-1 hour*60+minute minute/60 5**2 (5+9)*(15-7)

The symbols +, -, and /, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (*) is the symbol for multiplication, and ** is the symbol for exponentiation.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

Addition, subtraction, multiplication, and exponentiation al do what you expect, but you might be surprised by division. The fol owing operation has an unexpected result:

>>> minute = 59

>>> minute/60

0

The value of minute is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing integer division.

When both of the operands are integers, the result must also be an integer, and by convention, integer division always rounds down, even in cases like this where the next integer is very close.

A possible solution to this problem is to calculate a percentage rather than a fraction:

>>> minute*100/60

98

Again the result is rounded down, but at least now the answer is approximately correct. Another alternative is to use floating-point division. We'l see in the chapter 4 how to convert integer values and variables to floating-point values.

2.7. Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python fol ows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even though it doesn't change the result.

2. Exponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.

3. Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2*3-1 yields 5 rather than 4, and 2/3-1 is

-1, not 1 (remember that in integer division, 2/3=0).

4. Operators with the same precedence are evaluated from left to right. So in the expression minute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59*1, which is 59, which is wrong.

2.8. Operations on strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The fol owing are il egal (assuming that message has type string):

message-1 "Hello"/123 message*"Hello" "15"+2

Interestingly, the + operator does work with strings, although it does not do exactly what you might expect. For strings, the + operator represents concatenation, which means joining the two operands by linking them end-to-end. For example:

fruit = "banana"

baked_good = " nut bread"

print fruit + baked_good

The output of this program is banana nut bread. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings.

The * operator also works on strings; it performs repetition. For example, 'Fun'*3 is 'FunFunFun'. One of the operands has to be a string; the other has to be an integer.

On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication.

Just as 4*3 is equivalent to 4+4+4, we expect "Fun"*3 to be the same as "Fun"+"Fun"+"Fun", and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

2.9. Input

There are two built-in functions in Python for getting keyboard input:

n = raw_input("Please enter your name: ")

print n

n = input("Enter a numerical expression: ")

print n

A sample run of this script would look something like this:

$ python tryinput.py

Please enter your name: Arthur, King of the Britons

Arthur, King of the Britons

Enter a numerical expression: 7 * 3

21

Each of these functions al ows a prompt to be given to the function between the parentheses.

2.10. Composition

So far, we have looked at the elements of a program — variables, expressions, and statements — in isolation, without talking about

how to combine them.

One of the most useful features of programming languages is their ability to take smal building blocks and compose them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

>>> print 17 + 3

20

In reality, the addition has to happen before the printing, so the actions aren't actual y happening at the same time. The point is that any expression involving numbers, strings, and variables can be used inside a print statement. You've already seen an example of this:

print "Number of minutes since midnight: ", hour*60+minute

You can also put arbitrary expressions on the right-hand side of an assignment statement: percentage = (minute * 100) / 60

This ability may not seem impressive now, but you wil see other examples where composition makes it possible to express complex computations neatly and concisely.

Warning: There are limits on where you can use certain expressions. For example, the left-hand side of an assignment statement has to be a variable name, not an expression. So, the fol owing is il egal: minute+1 = hour.

2.11. Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are cal ed comments, and they are marked with the # symbol:

 # compute the percentage of the hour that has elapsed

percentage = (minute * 100) / 60

In this case, the comment appears on a line by itself. You can also put comments at the end of a line: percentage = (minute * 100) / 60 *# caution: integer division*

Everything from the # to the end of the line is ignored — it has no effect on the program. The message is intended for the programmer or for future programmers who might use this code. In this case, it reminds the reader about the ever-surprising behavior of integer division.

2.12. Glossary

assignment operator

= is Python's assignment operator, which should not be confused with the mathematical comparison operator using the same symbol.

assignment statement

A statement that assigns a value to a name (variable). To the left of the assignment operator, =, is a name. To the right of the assignment operator is an expression which is evaluated by the Python interpreter and then assigned to the name. The difference between the left and right hand sides of the assignment statement is often confusing to new programmers. In the fol owing assignment: n = n + 1

n plays a very different role on each side of the =. On the right it is a value and makes up part of the expression which wil be evaluated by

the Python interpreter before assigning it to the name on the left.

comment

Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

composition

The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

concatenate

To join two strings end-to-end.

data type

A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (type int), floating-point numbers (type float), and strings (type str).

evaluate

To simplify an expression by performing the operations in order to yield a single value.

expression

A combination of variables, operators, and values that represents a single result value.

float

A Python data type which stores floating-point numbers. Floating-point numbers are stored internal y in two parts: a base and an

exponent. When printed in the standard format, they look like decimal numbers. Beware of rounding errors when you use floats, and remember that they are only approximate values.

int

A Python data type that holds positive and negative whole numbers.

integer division

An operation that divides one integer by another and yields an integer. Integer division yields only the whole number of times that the numerator is divisible by the denominator and discards any remainder.

keyword

A reserved word that is used by the compiler to parse program; you cannot use keywords like if, def, and while as variable names.

operand

One of the values on which an operator operates.

operator

A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

rules of precedence

The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

state diagram

A graphical representation of a set of variables and the values to which they refer.

statement

An instruction that the Python interpreter can execute. Examples of statements include the assignment statement and the print statement.

str

A Python data type that holds a string of characters.

value

A number or string (or other things to be named later) that can be stored in a variable or computed in an expression.

variable

A name that refers to a value.

variable name

A name given to a variable. Variable names in Python consist of a sequence of letters (a..z, A..Z, and _) and digits (0..9) that begins with a letter. In best programming practice, variable names should be chosen so that they describe their use in the program, making the program self documenting.

2.13. Exercises

1. Record what happens when you print an assignment statement:

>>> print n = 7

How about this?

>>> print 7 + 5

Or this?

>>> print 5.2, "this", 4 - 2, "that", 5/2.0

Can you think of a general rule for what can fol ow the print statement? What does the print statement return?

2. Take the sentence: All work and no play makes Jack a dull boy. Store each word in a separate variable, then print out the sentence on one line using print.

3. Add parenthesis to the expression 6 * 1 - 2 to change its value from 4 to -6.

4. Place a comment before a line of code that previously worked, and record what happens when you rerun the program.

5. The difference between input and raw_input is that input evaluates the input string and raw_input does not. Try the fol owing in the interpreter and record what happens:

>>> x = input()

3.14

>>> type(x)

>>> x = raw_input()

3.14

>>> type(x)

>>> x = input()

'The knights who say "ni!"'

>>> x

What happens if you try the example above without the quotation marks?

>>> x = input()

The knights who say "ni!"

>>> x

>>> x = raw_input()

'The knights who say "ni!"'

>>> x

Describe and explain each result.

6. Start the Python interpreter and enter bruce + 4 at the prompt. This wil give you an error: NameError: name 'bruce' **is not** defined

Assign a value to bruce so that bruce + 4 evaluates to 10.

7. Write a program (Python script) named madlib.py, which asks the user to enter a series of nouns, verbs, adjectives, adverbs, plural nouns, past tense verbs, etc., and then generates a paragraph which is syntactical y correct but semantical y ridiculous (see http://madlibs.org for examples).

3. Functions

3.1. Definitions and use

In the context of programming, a function is a named sequence of statements that performs a desired operation. This operation is specified in a function definition. In Python, the syntax for a function definition is:

def NAME(LIST OF PARAMETERS):

STATEMENTS

You can make up any names you want for the functions you create, except that you can't use a name that is a Python keyword. The list of parameters specifies what information, if any, you have to provide in order to use the new function.

There can be any number of statements inside the function, but they have to be indented from the def.

In the examples in this book, we wil use the standard indentation of four spaces. Function definitions are the first of several compound statements we wil see, al of which have the same pattern: 1. A header, which begins with a keyword and ends with a colon.

2. A body consisting of one or more Python statements, each indented the same amount – 4 spaces is the Python standard – from the header.

In a function definition, the keyword in the header is def, which is fol owed by the name of the function and a list of parameters enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters. In either case, the parentheses are required.

The first couple of functions we are going to write have no parameters, so the syntax looks like this: **def** new_line():

print # *a print statement with no arguments prints a new line*

This function is named new_line. The empty parentheses indicate that it has no parameters. Its body contains only a single statement, which outputs a newline character. (That's what happens when you use a print command without any arguments.)

Defining a new function does not make the function run. To do that we need a function call. Function cal s contain the name of the function being executed fol owed by a list of values, cal ed arguments, which are assigned to the parameters in the function definition. Our first examples have an empty parameter list, so the function cal s do not take any arguments. Notice, however, that the parentheses are required in the function call:

print "First Line."

new_line()

print "Second Line."

The output of this program is:

First line.

Second line.

The extra space between the two lines is a result of the new_line() function cal . What if we wanted more space between the lines? We could cal the same function repeatedly:

print "First Line."

new_line()

new_line()

new_line()

print "Second Line."

Or we could write a new function named three_lines that prints three new lines:

def three_lines():

new_line()

new_line()

new_line()

print "First Line."

three_lines()

print "Second Line."

This function contains three statements, al of which are indented by four spaces. Since the next statement is not indented, Python knows that it is not part of the function.

You should notice a few things about this program:

You can cal the same procedure repeatedly. In fact, it is quite common and useful to do so.

You can have one function cal another function; in this case three_lines cal s new_line.

So far, it may not be clear why it is worth the trouble to create al of these new functions. Actual y, there are a lot of reasons, but this example demonstrates two:

1. Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code.

2. Creating a new function can make a program smal er by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to cal three_lines three times.

Pul ing together the code fragments from the previous section into a script named tryme1.py, the whole program looks like this:

def new_line():

print

def three_lines():

new_line()

new_line()

new_line()

print "First Line."

three_lines()

print "Second Line."

This program contains two function definitions: new_line and three_lines. Function definitions get executed just like other statements, but the effect is to create the new function. The statements inside the function do not get executed until the function is cal ed, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed

before the first time it is cal ed.

3.2. Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is cal ed the flow of execution.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is cal ed. Although it is not common, you can define one function inside another. In this case, the inner definition isn't executed until the outer function is cal ed.

Function cal s are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the cal ed function, executes al the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can cal another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that cal ed it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, fol ow the flow of execution.

3.3. Parameters, arguments, and the **import** statement

Most functions require arguments, values that control how the function does its job. For example, if you want to find the absolute value of a number, you have to indicate what the number is. Python has a built-in function for computing the absolute value:

>>> abs(5)

5**>>>** abs(-5)

5

In this example, the arguments to the abs function are 5 and -5.

Some functions take more than one argument. For example the built-in function pow takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables cal ed parameters.

>>> pow(2, 3)

8**>>>** pow(7, 4)

2401

Another built-in function that takes more than one argument is max.

>>> max(7, 11)

11

>>> max(4, 1, 17, 2, 12)

17

>>> max(3 * 11, 5**3, 512 - 9, 1024**0)

503

max can be sent any number of arguments, separated by commas, and wil return the maximum value sent. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1.

Here is an example of a user-defined function that has a parameter:

def print_twice(param):

print param, param

This function takes a single argument and assigns it to the parameter named param. The value of the parameter (at this point we have no idea what it wil be) is printed twice, fol owed by a newline. The name param was chosen to reinforce the idea that it is a parameter, but in general, you wil want to choose a name for your parameters that describes their use in the function.

The interactive Python shel provides us with a convenient way to test our functions. We can use the import statement to bring the functions we have defined in a script into the interpreter session. To see how this works, assume the print_twice function is defined in a script named chap03.py. We can now test it interactively by importing it into our Python shel session:

>>> from chap03 import *

>>> print_twice('Spam')

Spam Spam

>>> print_twice(5)

5 5

>>> print_twice(3.14159)

3.14159 3.14159

In a function cal , the value of the argument is assigned to the corresponding parameter in the function definition. In effect, it is if param = 'Spam' is executed when print_twice('Spam') is cal ed, param = 5 in print_twice(5), and param = 3.14159 in print_twice(3.14159).

Any type of argument that can be printed can be sent to print_twice In the first function cal , the argument is a string. In the second, it's an integer. In the third, it's a float.

As with built-in functions, we can use an expression as an argument for print_twice:

>>> print_twice('Spam' * 4)

SpamSpamSpamSpam SpamSpamSpamSpam

'Spam'*4 is first evaluated to 'SpamSpamSpamSpam', which is then passed as an argument to print_twice.

3.4. Composition

Just as with mathematical functions, Python functions can be composed, meaning that you use the result of one function as the input to another.

>>> print_twice(abs(-7))

7 7

>>> print_twice(max(3, 1, abs(-11), 7))

11 11

In the first example, abs(-7) evaluates to 7, which then becomes the argument to print_twice. In the second example we have two levels
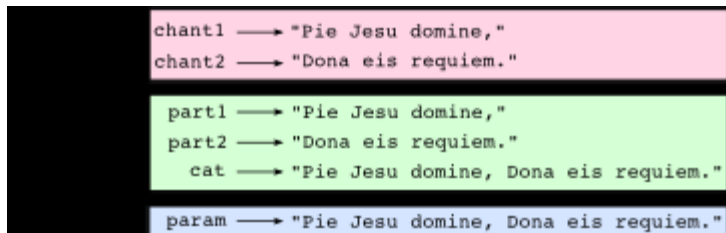
of composition, since abs(-11) is first evaluated to 11 before max(3, 1, 11, 7) is evaluated to 11 and print_twice(11) then displays the result.

We can also use a variable as an argument:

>>> sval = 'Eric, the half a bee.'

>>> print_twice(sval)

Eric, the half a bee. Eric, the half a bee.

Notice something very important here. The name of the variable we pass as an argument (sval) has nothing to do with the name of the parameter (param). Again, it is as if param = sval is executed when print_twice(sval) is cal ed. It doesn't matter what the value was named in the cal er, in print_twice its name is param.

3.5. Variables and parameters are local

When you create a local variable inside a function, it only exists inside the function, and you cannot use it outside. For example:

def cat_twice(part1, part2):

cat = part1 + part2

print_twice(cat)

This function takes two arguments, concatenates them, and then prints the result twice. We can cal the function with two strings:

>>> chant1 = "Pie Jesu domine, "

>>> chant2 = "Dona eis requiem."

>>> cat_twice(chant1, chant2)

Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.

When cat_twice terminates, the variable cat is destroyed. If we try to print it, we get an error:

>>> print cat

NameError: name 'cat' is not defined

Parameters are also local. For example, outside the function print_twice, there is no such thing as param. If you try to use it, Python wil complain.

3.6. Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram.

Like state diagrams, stack diagrams show the value of each variable, but they also show the function to which each variable belongs.

Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:

The order of the stack shows the flow of execution. print_twice was cal ed by cat_twice, and cat_twice was cal ed by __main__, which is a special name for the topmost function. When you create a variable outside of any function, it belongs to __main__.

Each parameter refers to the same value as its corresponding argument. So, part1 has the same value as chant1, part2 has the same value as chant2, and param has the same value as cat.

If an error occurs during a function cal , Python prints the name of the function, and the name of the function that cal ed it, and the name of the function that cal ed that, al the way back to the top most function.

To see how this works, create a Python script named tryme2.py that looks like this: **def** print_twice(param):

print param, param

print cat

def cat_twice(part1, part2):

cat = part1 + part2

print_twice(cat)

chant1 = "Pie Jesu domine, "

chant2 = "Dona eis requim."

cat_twice(chant1, chant2)

We've added the statement, print cat inside the print_twice function, but cat is not defined there.

Running this script wil produce an error message like this:

Traceback (innermost last):

File "tryme2.py", line 11, **in** <module>

cat_twice(chant1, chant2)

File "tryme2.py", line 7, **in** cat_twice

print_twice(cat)

File "tryme2.py", line 3, **in** print_twice

print cat

NameError: **global** name 'cat' **is not** defined

This list of functions is cal ed a traceback. It tel s you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

Notice the similarity between the traceback and the stack diagram. It's not a coincidence. In fact, another common name for a traceback is a stack trace.

3.7. Glossary

argument

A value provided to a function when the function is cal ed. This value is assigned to the corresponding parameter in the function.

body

The second part of a compound statement. The body consists of a sequence of statements al indented the same amount from the beginning of the header. The standard amount of indentation used within the Python community is 4 spaces.

compound statement

A statement that consists of two parts:

1. header - which begins with a keyword determining the statement type, and ends with a colon.

2. body - containing one or more statements indented the same amount from the header.

The syntax of a compound statement looks like this:

keyword expression:

statement

statement ...

flow of execution

The order in which statements are executed during a program run.

frame

A box in a stack diagram that represents a function cal . It contains the local variables and parameters of the function.

function

A named sequence of statements that performs some useful operation. Functions may or may not take parameters and may or may not produce a result.

function call

A statement that executes a function. It consists of the name of the function fol owed by a list of arguments enclosed in parentheses.

function composition

Using the output from one function cal as the input to another.

function definition

A statement that creates a new function, specifying its name, parameters, and the statements it executes.

header

The first part of a compound statement. Headers begin with a keyword and end with a colon (:) import

A statement which permits functions and variables defined in a Python script to be brought into the environment of another script or a running Python shel . For example, assume the fol owing is in a script named tryme.py:

def print_thrice(thing):

print thing, thing, thing

n = 42

s = "And now for something completely different..."

Now begin a python shel from within the same directory where tryme.py is located:

$ ls

tryme.py

$ python

>>>

Three names are defined in tryme.py: print_thrice, n, and s. If we try to access any of these in the shel without first importing, we get an error:

>>> n

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'n' is not defined

>>> print_thrice("ouch!")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'print_thrice' is not defined

If we import everything from tryme.py, however, we can use everything defined in it:

>>> from tryme import *

>>> n

42

>>> s

'And now for something completely different...'

>>> print_thrice("Yipee!")

Yipee! Yipee! Yipee!

>>>

Note that you do not include the .py from the script name in the import statement.

local variable

A variable defined inside a function. A local variable can only be used inside its function.

parameter

A name used inside a function to refer to the value passed as an argument.

stack diagram

A graphical representation of a stack of functions, their variables, and the values to which they refer.

traceback

A list of the functions that are executing, printed when a runtime error occurs. A traceback is also commonly refered to as a stack trace, since it lists the functions in the order in which they are stored in the [runtime stack](#).

3.8. Exercises

1. Using a text editor, create a Python script named tryme3.py . Write a function in this file cal ed nine_lines that uses three_lines to print nine blank lines. Now add a function named clear_screen that prints out twenty-five blank lines. The last line of your program should be a call to clear_screen.

2. Move the last line of tryme3.py to the top of the program, so the function call to clear_screen appears before the function definition. Run the program and record what error message you get.

Can you state a rule about function definitions and function calls which describes where they can appear relative to each other in a program?

3. Starting with a working version of tryme3.py , move the definition of new_line after the definition of three_lines. Record what happens when you run this program. Now move the definition of new_line below a cal to three_lines(). Explain how this is an example of the rule you stated in the previous exercise.

4. Fil in the body of the function definition for cat_n_times so that it wil print the string, s, n times: **def** cat_n_times(s, n):

<fill **in** your code here>

Save this function in a script named import_test.py. Now at a unix prompt, make sure you are in the same directory where the import_test.py is located (ls should show import_test.py). Start a Python shel and try the fol owing:

>>> from import_test import *

>>> cat_n_times('Spam', 7)

SpamSpamSpamSpamSpamSpamSpam

If al is wel , your session should work the same as this one. Experiment with other cal s to cat_n_times until you feel comfortable with how it works.

4. Conditionals

4.1. The modulus operator

The modulus operator works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

>>> quotient = 7 / 3

>>> print quotient

2**>>>** remainder = 7 % 3

>>> print remainder

1

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another —if x % y is zero, then x is divisible by y.

Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

4.2. Boolean values and expressions

The Python type for storing true and false values is cal ed bool, named after the British mathematician, George Boole. George Boole created Boolean algebra, which is the basis of al modern computer arithmetic.

There are only two boolean values: True and False. Capitalization is important, since true and false are not boolean values.

>>> type(True)

<type 'bool'>

>>> type(true)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'true' is not defined

A boolean expression is an expression that evaluates to a boolean value. The operator == compares two values and produces a boolean value:

>>> 5 == 5

True

>>> 5 == 6

False

In the first statement, the two operands are equal, so the expression evaluates to True; in the second statement, 5 is not equal to 6, so we get False.

The == operator is one of the comparison operators; the others are:

x != y # *x is not equal to y*

x > y # *x is greater than y*

x < y # *x is less than y*

x >= y # *x is greater than or equal to y*

x <= y # *x is less than or equal to y*

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. Also, there is no such thing as =< or =>.

4.3. Logical operators

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 and less than 10.

n % 2 == 0 or n % 3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3.

Final y, the not operator negates a boolean expression, so not(x > y) is true if (x > y) is false, that is, if x is less than or equal to y.

4.4. Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the ** if statement**:

if x > 0:

print "x is positive"

The boolean expression after the if statement is cal ed the condition. If it is true, then the indented statement gets executed. If not, nothing happens.

The syntax for an if statement looks like this:

if BOOLEAN EXPRESSION:

STATEMENTS

As with the function definition from last chapter and other compound statements, the if statement consists of a header and a body. The header begins with the keyword if fol owed by a boolean expression and ends with a colon (:).

The indented statements that fol ow are cal ed a block. The first unindented statement marks the end of the block. A statement block inside a compound statement is cal ed the body of the statement.

Each of the statements inside the body are executed in order if the boolean expression evaluates to True. The entire block is skipped if the boolean expression evaluates to False.

There is no limit on the number of statements that can appear in the body of an if statement, but there has to be at least one. Occasional y, it is useful to have a body with no statements (usual y as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

if True: # *This is always true*

pass # *so this is always executed, but it does nothing*

4.5. Alternative execution

A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

if x % 2 == 0:

print x, "is even"

else:

print x, "is odd"

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives wil be executed. The alternatives are cal ed branches, because they are branches in the flow of execution.

As an aside, if you need to check the parity (evenness or oddness) of numbers often, you might wrap this code in a function:

```
def print_parity(x):

if x % 2 == 0:

print x, "is even"

else:

print x, "is odd"
```

For any value of x, print_parity displays an appropriate message. When you cal it, you can provide any integer expression as an argument.

```
>>> print_parity(17)

17 is odd.

>>> y = 41

>>> print_parity(y+1)

42 is even.
```

4.6. Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a

chained conditional:

if x < y:

print x, "is less than", y

elif x > y:

print x, "is greater than", y

else:

print x, "and", y, "are equal"

elif is an abbreviation of else if . Again, exactly one branch wil be executed. There is no limit of the number of elif statements but only a single (and optional) else statement is al owed and it must be the last branch in the statement:

if choice == 'a':

function_a()

elif choice == 'b':

function_b()

elif choice == 'c':

function_c()

else:

print "Invalid choice."

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch

executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

4.7. Nested conditionals

One conditional can also be nested within another. We could have written the trichotomy example as fol ows:

if x == y:

print x, "and", y, "are equal"

else:

if x < y:

print x, "is less than", y

else:

print x, "is greater than", y

The outer conditional contains two branches. The first branch contains a simple output statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both output statements, although they could have been conditional statements as wel .

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the fol owing code using a single conditional:

if 0 < x:

```python
if x < 10:
```

```python
print "x is a positive single digit."
```

The print statement is executed only if we make it past both the conditionals, so we can use the and operator:

```python
if 0 < x and x < 10:
```

```python
print "x is a positive single digit."
```

These kinds of conditions are common, so Python provides an alternative syntax that is similar to mathematical notation:

```python
if 0 < x < 10:
```

```python
print "x is a positive single digit."
```

This condition is semantical y the same as the compound boolean expression and the nested conditional.

4.8. The **return** statement

The return statement al ows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```python
def print_square_root(x):
```

```python
if x <= 0:
```

```python
print "Positive numbers only, please."
```

```python
return
```

```python
result = x**0.5
```

```python
print "The square root of", x, "is", result
```

The function print_square_root has a parameter named x. The first thing it does is check whether x is less than or equal to 0, in which case it displays an error message and then uses return to exit the function. The flow of execution immediately returns to the cal er, and the remaining lines of the function are not executed.

4.9. Keyboard input

In [Input](#) we were introduced to Python's built-in functions that get input from the keyboard: raw_input and input. Now let's look at these again in greater depth.

When either of these functions are cal ed, the program stops and waits for the user to type something.

When the user presses Return or the Enter key, the program resumes and raw_input returns what the user typed as a string:

>>> my_input = raw_input()

What are you waiting for?

>>> print my_input

What are you waiting for?

Before cal ing raw_input, it is a good idea to print a message tel ing the user what to input. This message is cal ed a prompt. We can supply a prompt as an argument to raw_input:

>>> name = raw_input("What...is your name? ")

What...is your name? Arthur, King of the Britons!

>>> print name

Arthur, King of the Britons!

Notice that the prompt is a string, so it must be enclosed in quotation marks.

If we expect the response to be an integer, we can use the input function which evaluates the response as a Python expression:

prompt = "What...is the airspeed velocity of an unladen swallow? \n"

speed = input(prompt)

If the user types a string of digits, it is converted to an integer and assigned to speed. Unfortunately, if the user types characters that do not make up a valid Python expression, the program crashes:

>>> speed = input(prompt)

What...is the airspeed velocity of an unladen swallow?

What do you mean, an African or a European swallow?

...

SyntaxError: invalid syntax

In the last example, if the user had made the response a valid Python expression by putting quotes around it, it would not have given an error:

>>> speed = input(prompt)

What...is the airspeed velocity of an unladen swallow?

"What do you mean, an African or a European swallow?"

>>> speed

'What do you mean, an African or a European swallow?'

\>\>\>

To avoid this kind of error, it is a good idea to use raw_input to get a string and then use conversion commands to convert it to other types.

4.10. Type conversion

Each Python type comes with a built-in command that attempts to convert values of another type into that type. The int(ARGUMENT) command, for example, takes any value and converts it to an integer, if possible, or complains otherwise:

>>> int("32")

32

>>> int("Hello")

ValueError: invalid literal for int() with base 10: 'Hello'

int can also convert floating-point values to integers, but remember that it truncates the fractional part:

>>> int(-2.3)

-2

>>> int(3.99999)

3**>>>** int("42")

42

>>> int(1.0)

1

The float(ARGUMENT) command converts integers and strings to floating-point numbers:

>>> float(32)

32.0

>>> float("3.14159")

3.14159

>>> float(1)

1.0

It may seem odd that Python distinguishes the integer value 1 from the floating-point value 1.0. They may represent the same number, but they belong to different types. The reason is that they are represented differently inside the computer.

The str(ARGUMENT) command converts any argument given to it to type string:

>>> str(32)

'32'

>>> str(3.14149)

'3.14149'

>>> str(True)

'True'

>>> str(true)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'true' is not defined

str(ARGUMENT) wil work with any value and convert it into a string. As mentioned earlier, True is boolean value; true is not.

For boolean values, the situation is especial y interesting:

>>> bool(1)

True

>>> bool(0)

False

>>> bool("Ni!")

True

>>> bool("")

False

>>> bool(3.14159)

True

>>> bool(0.0)

False

Python assigns boolean values to values of other types. For numerical types like integers and floating-points, zero values are false and non-zero values are true. For strings, empty strings are false and non-empty strings are true.

4.11. GASP
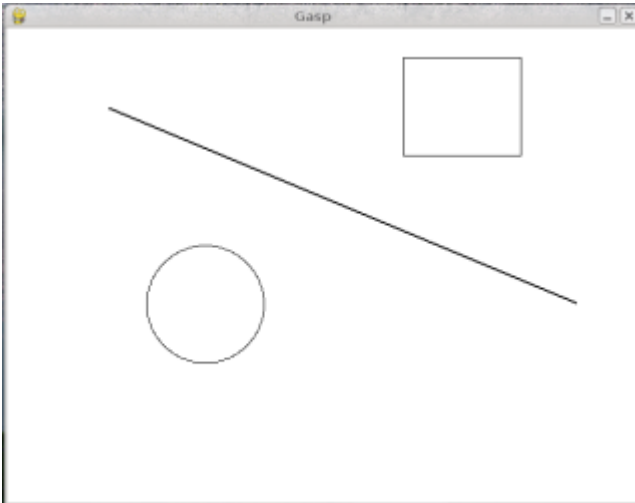
GASP (Graphics API for Students of Python) wil enable us to write programs involving graphics. Before you can use GASP, it needs to be instal ed on your machine. If you are running Ubuntu GNU/Linux, see

GASP in Appendix A. Current instructions for instal ing GASP on other platforms can be found at

http://dev.laptop.org/pub/gasp/downloads.

After instal ing gasp, try the fol owing python script:

from gasp import *

begin_graphics()

Circle((200, 200), 60)

Line((100, 400), (580, 200))

Box((400, 350), 120, 100)

update_when('key_pressed')

end_graphics()

The second to the last command pauses and waits until a key is pressed. Without it, the screen would flash by so quickly you wouldn't see it.

Running this script, you should see a graphics window that looks like this:

We wil be using gasp from here on to il ustrate (pun intended) computer programming concepts and to add to our fun while learning. You can find out more about the GASP module by reading [Appendix B](#).

4.12. Glossary

block

A group of consecutive statements with the same indentation.

body

The block of statements in a compound statement that fol ows the header.

boolean expression

An expression that is either true or false.

boolean value

There are exactly two boolean values: True and False. Boolean values result when a boolean expression is evaluated by the Python interepreter. They have type bool.

branch

One of the possible paths of the flow of execution determined by conditional execution.

chained conditional

A conditional branch with more than two possible flows of execution. In Python chained conditionals are written with if ... elif ... else statements.

comparison operator

One of the operators that compares two values: ==, !=, >, <, >=, and <=.

condition

The boolean expression in a conditional statement that determines which branch is executed.

conditional statement

A statement that controls the flow of execution depending on some condition. In Python the keywords if, elif, and else are used for conditional statements.

logical operator

One of the operators that combines boolean expressions: and, or, and not.

modulus operator

An operator, denoted with a percent sign (%), that works on integers and yields the remainder when one number is divided by another.

nesting

One program structure within another, such as a conditional statement inside a branch of another conditional statement.

prompt

A visual cue that tel s the user to input data.

type conversion

An explicit statement that takes a value of one type and computes a corresponding value of another type.

wrapping code in a function

The process of adding a function header and parameters to a sequence of program statements is often refered to as "wrapping the code in a function". This process is very useful whenever the program statements in question are going to be used multiple times.

4.13. Exercises

1. Try to evaluate the fol owing numerical expressions in your head, then use the Python interpreter to check your results:

1. >>> 5 % 2

2. >>> 9 % 5

3. >>> 15 % 12

4. >>> 12 % 15

5. >>> 6 % 6

6. >>> 0 % 7

7. >>> 7 % 0

What happened with the last example? Why? If you were able to correctly anticipate the computer's response in al but the last one, it is time to move on. If not, take time now to make up examples of your own. Explore the modulus operator until you are confident you understand how it works.

2. **if** x < y:

print x, "is less than", y

elif x > y:

print x, "is greater than", y

else:

print x, "and", y, "are equal"

Wrap this code in a function cal ed compare(x, y). Cal compare three times: one each where the first argument is less than, greater than, and equal to the second argument.

3. To better understand boolean expressions, it is helpful to construct truth tables. Two boolean expressions are logically equivalent if and only if they have the same truth table.

The fol owing Python script prints out the truth table for any boolean expression in two variables: p and q:

expression = raw_input("Enter a boolean expression in two variables, p and q: ") **print** " p q *%s*" % expression

length = len(" p q *%s*" % expression)

print length*"="

for p **in** True, False:

for q **in** True, False:

print " *%-7s %-7s %-7s*" % (p, q, eval(expression)) You wil learn how this script works in later chapters. For now, you wil use it to learn about boolean expressions. Copy this program to a file named p_and_q.py, then run it from the command line and give it: p or q, when prompted for a boolean expression. You should get the fol owing output:

p q p or q

```
=====================
```

True True True

True False True

False True True

False False False

Now that we see how it works, let's wrap it in a function to make it easier to use: **def** truth_table(expression):

print " p q *%s*" % expression

length = len(" p q *%s*" % expression)

print length*"="

for p **in** True, False:

for q **in** True, False:

print " *%-7s %-7s %-7s*" % (p, q, eval(expression)) We can import it into a Python shel and cal truth_table with a string containing our boolean expression in p and q as an argument:

>>> from p_and_q import *

>>> truth_table("p or q")

p q p or q

=====================

True True True

True False True

False True True

False False False

>>>

Use the truth_table functions with the fol owing boolean expressions, recording the truth table produced each time:

1. not(p or q)

2. p and q

3. not(p and q)

4. not(p) or not(q)

5. not(p) and not(q)

Which of these are logical y equivalent?

4. Enter the fol owing expressions into the Python shel :

True **or** False

True **and** False

not(False) **and** True

True **or** 7

False **or** 7

True **and** 0

False **or** 8

"happy" **and** "sad"

"happy" **or** "sad"

"" **and** "sad"

"happy" **and** ""

Analyze these results. What observations can you make about values of different types and logical operators? Can you write these observations in the form of simple rules about and and or expressions?

5. **if** choice == 'a':

function_a()

elif choice == 'b':

function_b()

elif choice == 'c':

function_c()

else:

print "Invalid choice."

Wrap this code in a function cal ed dispatch(choice). Then define function_a, function_b, and function_c so that they print out a message saying they were cal ed. For example:

def function_a():

print "function_a was called..."

Put the four functions (dispatch, function_a, function_b, and function_c into a script named ch04e05.py. At the bottom of this script add a cal to dispatch('b'). Your output should be: function_b was called...

Final y, modify the script so that user can enter 'a', 'b', or 'c'. Test it by importing your script into the Python shel .

6. Write a function named is_divisible_by_3 that takes a single integer as an argument and prints

"This number is divisible by three." if the argument is evenly divisible by 3 and "This number is not divisible by three." otherwise.

Now write a similar function named is_divisible_by_5.

7. Generalize the functions you wrote in the previous exercise into a function named is_divisible_by_n(x, n) that takes two integer arguments and prints out whether the first is divisible by the second. Save this in a file named ch04e07.py. Import it into a shel and try it out. A sample session might look like this:

>>> from ch04e07 import *

>>> is_divisible_by_n(20, 4)

Yes, 20 is divisible by 4
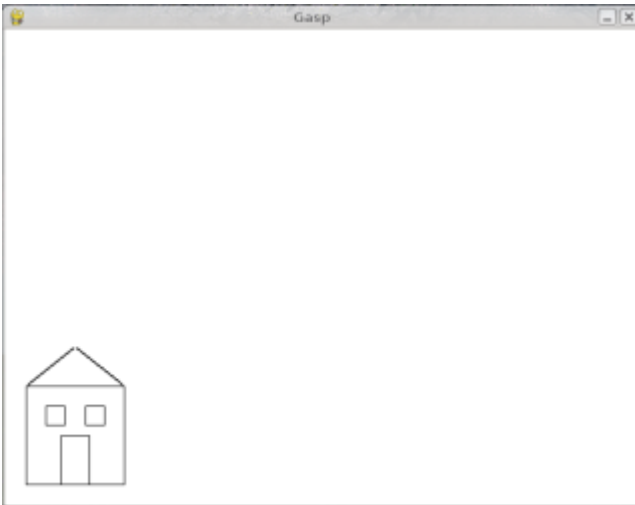
>>> is_divisible_by_n(21, 8)

No, 21 is not divisible by 8

8. What wil be the output of the fol owing?

if "Ni!":

print 'We are the Knights who say, "Ni!"'

else:

print "Stop it! No more of this!"

if 0:

print "And now for something completely different..."

else:

print "What's all this, then?"

Explain what happened and why it happened.

9. The fol owing gasp script, in a file named house.py, draws a simple house on a gasp canvas: **from gasp import** * # *import*

everything from the gasp library begin_graphics() # *open the graphics canvas*

Box((20, 20), 100, 100) # *the house*

Box((55, 20), 30, 50) # *the door*

Box((40, 80), 20, 20) # *the left window*

Box((80, 80), 20, 20) # *the right window*

Line((20, 120), (70, 160)) # *the left roof*

Line((70, 160), (120, 120)) # *the right roof*

update_when('key_pressed') # *keep the canvas open until a key is pressed*

end_graphics() # *close the canvas (which would happen*

anyway, since the script ends here, but it

is better to be explicit).

Run this script and confirm that you get a window that looks like this:

1. Wrap the house code in a function named draw_house().

2. Run the script now. Do you see a house? Why not?

3. Add a cal to draw_house() at the botton of the script so that the house returns to the screen.

4. Parameterize the function with x and y parameters – the header should then become def draw_house(x, y):, so that you can pass in the location of the house on the canvas.

5. Use draw_house to place five houses on the canvas in different locations.

10. Exploration: Read over [Appendix B](#) and write a script named houses.py that produces the fol owing when run:
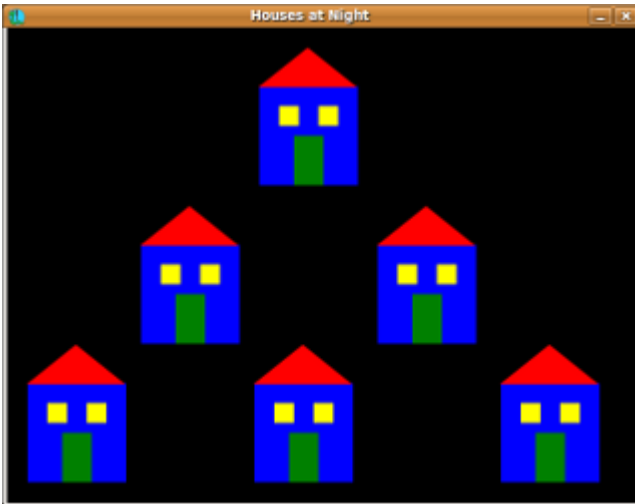
hint: You wil need to use a Polygon for the roof instead of two Lines to get filled=True to work with it.

5. Fruitful functions

5.1. Return values

The built-in functions we have used, such as abs, pow, and max, have produced results. Cal ing each of these functions generates a

value, which we usual y assign to a variable or use as part of an expression.

biggest = max(3, 7, 2, 5)

x = abs(3 - 11) + 10

But so far, none of the functions we have written has returned a value.

In this chapter, we are going to write functions that return values, which we wil cal fruitful functions, for want of a better name. The first example is area, which returns the area of a circle with the given radius:

def area(radius):

temp = 3.14159 * radius**2

return temp

We have seen the return statement before, but in a fruitful function the return statement includes a return value. This statement means: Return immediately from this function and use the fol owing expression as a return value. The expression provided can be arbitrarily complicated, so we could have written this function more concisely:

def area(radius):

return 3.14159 * radius**2

On the other hand, temporary variables like temp often make debugging easier.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional. We have already seen the built-in abs, now we see how to write our own:

```
def absolute_value(x):

if x < 0:

return -x

else:

return x
```

Since these return statements are in an alternative conditional, only one wil be executed. As soon as one is executed, the function terminates without executing any subsequent statements.

Another way to write the above function is to leave out the else and just fol ow the if condition by the second return statement.

```
def absolute_value(x):

if x < 0:

return -x

return x
```

Think about this version and convince yourself it works the same as the first one.

Code that appears after a return statement, or any other place the flow of execution can never reach, is cal ed dead code.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. The fol owing version of absolute_value fails to do this:

```
def absolute_value(x):
```

if x < 0:

return -x

elif x > 0:

return x

This version is not correct because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. In this case, the return value is a special value cal ed None:

>>> print absolute_value(0)

None

None is the unique value of a type cal ed the NoneType:

>>> type(None)

Al Python functions return None whenever they do not return another value.

5.2. Program development

At this point, you should be able to look at complete functions and tel what they do. Also, if you have been doing the exercises, you have written some smal functions. As you write larger functions, you might start to have more difficulty, especial y with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique cal ed incremental development. The goal of incremental development is to avoid long debugging sessions by adding and testing only a smal amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x1, y1) and (x2, y2). By the Pythagorean theorem, the distance is:

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function:

def distance(x1, y1, x2, y2):

return 0.0

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactical y correct, and it wil run, which means that we can test it before we make it more complicated.

To test the new function, we cal it with sample values:

>>> distance(1, 2, 4, 6)

0.0

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactical y correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be — in the last line we added.

A logical first step in the computation is to find the differences x2- x1 and y2- y1. We wil store those values in temporary variables named dx and dy and print them.

```
def distance(x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

print "dx is", dx

print "dy is", dy

return 0.0
```

If the function is working, the outputs should be 3 and 4. If so, we know that the function is getting the right parameters and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of dx and dy:

```
def distance(x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

dsquared = dx**2 + dy**2

print "dsquared is: ", dsquared

return 0.0
```

Notice that we removed the print statements we wrote in the previous step. Code like that is cal ed scaffolding because it is helpful for building the program but is not part of the final product.

Again, we would run the program at this stage and check the output (which should be 25).

Final y, using the fractional exponent 0.5 to find the square root, we compute and return the result: **def** distance(x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

dsquared = dx**2 + dy**2

result = dsquared**0.5

return result

If that works correctly, you are done. Otherwise, you might want to print the value of result before the return statement.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, the incremental development process can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make smal incremental changes. At any point, if there is an error, you wil know exactly where it is.

2. Use temporary variables to hold intermediate values so you can output and check them.

3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

5.3. Composition

As you should expect by now, you can cal one function from within another. This ability is cal ed composition.

As an example, we'l write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables xc and yc, and the perimeter point is in xp and yp.

The first step is to find the radius of the circle, which is the distance between the two points.

Fortunately, we've just written a function, distance, that does just that, so now al we have to do is use it:

radius = distance(xc, yc, xp, yp)

The second step is to find the area of a circle with that radius and return it. Again we wil use one of our earlier functions:

result = area(radius)

return result

Wrapping that up in a function, we get:

def area2(xc, yc, xp, yp):

radius = distance(xc, yc, xp, yp)

result = area(radius)

return result

We cal ed this function area2 to distinguish it from the area function defined earlier. There can only be one function with a given name within a given module.

The temporary variables radius and result are useful for development and debugging, but once the program is working, we can make it more concise by composing the function cal s:

def area2(xc, yc, xp, yp):

return area(distance(xc, yc, xp, yp))

5.4. Boolean functions

Functions can return boolean values, which is often convenient for hiding complicated tests inside functions. For example:

def is_divisible(x, y):

if x % y == 0:

return True

else:

return False

The name of this function is is_divisible. It is common to give boolean functions names that sound like yes/no questions. is_divisible returns either True or False to indicate whether the x is or is not divisible by y.

We can make the function more concise by taking advantage of the fact that the condition of the if statement is itself a boolean expression. We can return it directly, avoiding the if statement altogether: **def** is_divisible(x, y):

return x % y == 0

This session shows the new function in action:

>>> is_divisible(6, 4)

False

>>> is_divisible(6, 3)

True

Boolean functions are often used in conditional statements:

if is_divisible(x, y):

print "x is divisible by y"

else:

print "x is not divisible by y"

It might be tempting to write something like:

if is_divisible(x, y) == True:

but the extra comparison is unnecessary.

5.5. The **function** type

A function is another type in Python, joining int, float, str, bool, and NoneType.

>>> def func():

... return "function func was called..."

...

>>> type(func)

<type 'function'>

>>>

Just like the other types, functions can be passed as arguments to other functions: **def** f(n):

return 3*n - 6

def g(n):

return 5*n + 2

def h(n):

return -2*n + 17

def doto(value, func):

return func(value)

print doto(7, f)

print doto(7, g)

print doto(7, h)

doto is cal ed three times. 7 is the argument for value each time, and the functions f, g, and h are passed in for func in turn. The output of this script is:

15

37

3

This example is a bit contrived, but we wil see situations later where it is quite useful to pass a function to a function.

5.6. Programming with style

Readability is very important to programmers, since in practice programs are read and modified far more often than they are written. Al the code examples in this book wil be consistent with the Python

Enhancement Proposal 8 ([PEP 8](#)), a style guide developed by the Python community.

We'l have more to say about style as our programs become more complex, but a few pointers wil be helpful already:

use 4 spaces for indentation

imports should go at the top of the file

separate function definitions with two blank lines

keep function definitions together

keep top level statements, including function cal s, together at the bottom of the program 5.7. Triple quoted strings

In addition to the single and double quoted strings we first saw in [Values and data types](#), Python also has triple quoted strings:

```
>>> type("""This is a triple quoted string using 3 double quotes.""")
<type 'str'>
>>> type('''This triple quoted strings uses 3 single quotes.''')
<type 'str'>
>>>
```

Triple quoted strings can contain both single and double quotes inside them:

```
>>> print """"Oh no", she exclaimed, "Ben's bike is broken!"""
```

"Oh no", she exclaimed, "Ben's bike is broken!"

>>>

Final y, triple quoted strings can span multiple lines:

>>> message = """This message will

... span several

... lines."""

>>> print message

This message will

span several

lines.

>>>

5.8. Unit testing with **doctest**

It is a common best practice in software development these days to include automatic unit testing of source code. Unit testing provides a way to automatical y verify that individual pieces of code, such as functions, are working properly. This makes it possible to change the implementation of a function at a later time and quickly test that it stil does what it was intended to do.

Python has a built-in doctest module for easy unit testing. Doctests can be written within a triple quoted string on the first line of the body of a function or script. They consist of sample interpreter sessions with a series of inputs to a Python prompt fol owed by the expected output from the Python interpreter.

The doctest module automatical y runs any statement begining with >>> and compares the fol owing line with the output from the interpreter.

To see how this works, put the fol owing in a script named myfunctions.py:

```
def is_divisible_by_2_or_5(n):

    """

    >>> is_divisible_by_2_or_5(8)

    True

    """

if __name__ == '__main__':

    import doctest

    doctest.testmod()
```

The last three lines are what make doctest run. Put them at the bottom of any file that includes doctests. We wil explain how they work in Chapter 10 when we discuss modules.

Running the script wil produce the fol owing output:

```
$ python myfunctions.py

    **********************************************************************

File "myfunctions.py", line 3, in __main__.is_divisible_by_2_or_5

Failed example:

is_divisible_by_2_or_5(8)
```

Expected:

True

Got nothing

```
**********************************************************************
```

1 items had failures:

1 of 1 in __main__.is_divisible_by_2_or_5

Test Failed 1 failures.

$

This is an example of a failing test. The test says: if you cal is_divisible_by_2_or_5(8) the result should be True. Since is_divisible_by_2_or_5 as written doesn't return anything at al , the test fails, and doctest tel s us that it expected True but got nothing.

We can make this test pass by returning True:

```python
def is_divisible_by_2_or_5(n):
    """

    >>> is_divisible_by_2_or_5(8)
    True
    """
    return True

if __name__ == '__main__':
    import doctest
```

doctest.testmod()

If we run it now, there wil be no output, which indicates that the test passed. Note again that the doctest string must be placed immediately after the function definition header in order to run.

To see more detailed out put, cal the script with the -v command line option:

$ python myfunctions.py -v

Trying:

is_divisible_by_2_or_5(8)

Expecting:

True

ok

1 items had no tests:

__main__

1 items passed all tests:

1 tests in __main__.is_divisible_by_2_or_5

1 tests in 2 items.

1 passed and 0 failed.

Test passed.

$

While the test passed, our test suite is clearly inadequete, since is_divisible_by_2_or_5 wil now return True no matter what argument is passed to it. Here is a completed version with a more complete test

suite and code that makes the tests pass:

```python
def is_divisible_by_2_or_5(n):
    """

    >>> is_divisible_by_2_or_5(8)

    True

    >>> is_divisible_by_2_or_5(7)

    False

    >>> is_divisible_by_2_or_5(5)

    True

    >>> is_divisible_by_2_or_5(9)

    False

    """

    return n % 2 == 0 or n % 5 == 0

if __name__ == '__main__':

    import doctest

    doctest.testmod()
```

Run this script now with the -v command line option and see what you get.

5.9. Glossary

boolean function

A function that returns a boolean value.

composition (of functions)

Cal ing one function from within the body of another, or using the return value of one function as an argument to the cal of another.

dead code

Part of a program that can never be executed, often because it appears after a return statement.

fruitful function

A function that yields a return value.

incremental development

A program development plan intended to avoid debugging by adding and testing only a smal amount of code at a time.

None

A special Python value returned by functions that have no return statement, or a return statement without an argument. None is the only value of the type, NoneType.

return value

The value provided as the result of a function cal .

scaffolding

Code that is used during program development but is not part of the final version.

temporary variable

A variable used to store an intermediate value in a complex calculation.

unit testing

An automatic procedure used to validate that individual units of code are working properly. Python has doctest built in for this purpose.

5.10. Exercises

Al of the exercises below should be added to a file named ch05.py that contains the fol owing at the bottom:

if __name__ == '__main__':

import doctest

doctest.testmod()

After completing each exercise in turn, run the program to confirm that the doctests for your new function pass.

1. Write a compare function that returns 1 if a > b, 0 if a == b, and -1 if a < b.

def compare(a, b):

"""

>>> compare(5, 4)

1

>>> compare(7, 7)

0

>>> compare(2, 3)

-1

>>> compare(42, 1)

1

"""

 # Your function body should begin here.

Fil in the body of the function so the doctests pass.

2. Use incremental development to write a function cal ed hypotenuse that returns the length of the hypotenuse of a right triangle given the lengths of the two legs as parameters. Record each stage of the incremental development process as you go.

def hypotenuse(a, b):

"""

>>> hypotenuse(3, 4)

5.0

>>> hypotenuse(12, 5)

13.0

>>> hypotenuse(7, 24)

25.0

>>> hypotenuse(9, 12)

15.0

"""

When you are finished add your completed function with the doctests to ch05.py and confirm that the doctests pass.

3. Write a function slope(x1, y1, x2, y2) that returns the slope of the line through the points (x1, y1) and (x2, y2). Be sure your implementation of slope can pass the fol owing doctests: **def slope(x1, y1, x2, y2):**

"""

>>> slope(5, 3, 4, 2)

1.0

>>> slope(1, 2, 3, 2)

0.0

>>> slope(1, 2, 3, 3)

0.5

>>> slope(2, 4, 1, 2)

2.0

"""

Then a cal to slope in a new function named intercept(x1, y1, x2, y2) that returns the y-intercept of the line through the points (x1, y1) and

(x2, y2).

def intercept(x1, y1, x2, y2):

"""

>>> intercept(1, 6, 3, 12)

3.0

>>> intercept(6, 1, 1, 6)

7.0

>>> intercept(4, 6, 12, 8)

5.0

"""

intercept should pass the doctests above.

4. Write a function cal ed is_even(n) that takes an integer as an argument and returns True if the argument is an even number and False if it is odd.

Add your own doctests to this function.

5. Now write the function is_odd(n) that returns True when n is odd and False otherwise. Include doctests for this function as you write it.

Final y, modify it so that it uses a cal to is_even to determine if its argument is an odd integer.

6. Add the fol owing function definition header and doctests to ch05.py:

def is_factor(f, n):

```
"""
>>> is_factor(3, 12)

True

>>> is_factor(5, 12)

False

>>> is_factor(7, 14)

True

>>> is_factor(2, 14)

True

>>> is_factor(7, 15)

False

"""
```

Add a body to is_factor to make the doctests pass.

7. Add the fol owing function definition header and doctests to ch05.py:

def is_multiple(m, n):

```
"""
>>> is_multiple(12, 3)

True

>>> is_multiple(12, 4)
```

True

>>> is_multiple(12, 5)

False

>>> is_multiple(12, 6)

True

>>> is_multiple(12, 7)

False

"""

Add a body to is_multiple to make the doctests pass. Can you find a way to use is_factor in your definition of is_multiple?

8. Add the fol owing function definition header and doctests to ch05.py:

def f2c(t):

"""

>>> f2c(212)

100

>>> f2c(32)

0

>>> f2c(-40)

-40

>>> f2c(36)

2

>>> f2c(37)

3

>>> f2c(38)

3

>>> f2c(39)

4

"""

Write a body for the function definition of f2c designed to return the integer value of the nearest degree Celsius for given tempurature in Fahrenheit. (hint: you may want to make use of the builtin function, round. Try printing round.__doc__ in a Python shel and experimenting with round until you are comfortable with how it works.)

9. Add the fol owing function definition header and doctests to ch05.py:

def c2f(t):

"""

>>> c2f(0)

32

>>> c2f(100)

212

>>> c2f(-40)

-40

>>> c2f(12)

54

>>> c2f(18)

64

>>> c2f(-48)

-54

"""

Add a function body for c2f to convert from Celsius to Fahrenheit.

6. Iteration

6.1. Multiple assignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

bruce = 5

print bruce,

bruce = 7

print bruce

The output of this program is 5 7, because the first time bruce is printed, his value is 5, and the second time, his value is 7. The comma at the end of the first print statement suppresses the newline after the output, which is why both outputs appear on the same line.

Here is what multiple assignment looks like in a state diagram:

With multiple assignment it is especial y important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (=) for assignment, it is tempting to interpret a statement like a = b as a statement of equality. It is not!

First, equality is symmetric and assignment is not. For example, in mathematics, if a = 7 then 7 = a. But in Python, the statement a = 7 is legal and 7 = a is not.

Furthermore, in mathematics, a statement of equality is always true. If a = b now, then a wil always equal b. In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

a = 5

b = a # *a and b are now equal*

a = 3 # *a and b are no longer equal*

The third line changes the value of a but does not change the value of b, so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as <- or :=, to avoid confusion.)

6.2. Updating variables

One of the most common forms of multiple assignment is an update, where the new value of the variable depends on the old.

x = x + 1

This means get the current value of x, add one, and then update x with the new value.

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the expression on the right side of the assignment operator before it assigns the resulting value to the name on the left:

>>> x = x + 1

Traceback (most recent call last):

File "", line 1, in

NameError: name 'x' is not defined

Before you can update a variable, you have to initialize it, usual y with a simple assignment:

>>> x = 0

>>> x = x + 1

>>>

Updating a variable by adding 1 is cal ed an increment; subtracting 1 is cal ed a decrement.

6.3. The **while** statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do wel and people do poorly.

Repeated execution of a set of statements is cal ed iteration. Because iteration is so common, Python provides several language features to make it easier. The first feature we are going to look at is the while statement.

Here is a function cal ed countdown that demonstrates the use of the while statement: **def** countdown(n):

while n > 0:

print n

n = n-1

print "Blastoff!"

You can almost read the while statement as if it were English. It means, While n is greater than 0, continue displaying the value of n and then reducing the value of n by 1. When you get to 0, display the word Blastoff!

More formal y, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding False or True.

2. If the condition is false, exit the while statement and continue execution at the next statement.

3. If the condition is true, execute each of the statements in the body and then go back to step 1.

The body consists of al of the statements below the header with the same indentation.

This type of flow is cal ed a loop because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed.

The body of the loop should change the value of one or more variables so that eventual y the condition becomes false and the loop terminates. Otherwise the loop wil repeat forever, which is cal ed an infinite loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, Lather, rinse, repeat, are an infinite loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smal er each time through the loop, so eventual y we have to get to 0. In other cases, it is not so easy to tel . Look at the fol owing function, defined for al positive integers n:

def sequence(n):

while n != 1:

print n,

if n % 2 == 0: # *n is even*

n = n / 2

else: # *n is odd*

n = n * 3 + 1

The condition for this loop is n != 1, so the loop wil continue until n is 1, which wil make the condition false.

Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, the value of n is divided by 2. If it is odd, the value is replaced by n * 3 + 1. For example, if the starting value (the argument passed to sequence) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since n sometimes increases and sometimes decreases, there is no obvious proof that n wil ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, then the value of n wil be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for all values of n. So far, no one has been able to prove it or disprove it!

6.4. Tracing a program

To write effective computer programs a programmer needs to develop the ability to trace the execution of a computer program. Tracing involves becoming the computer and fol owing the flow of execution through a sample program run, recording the state of al variables and any output the program generates after each instruction is executed.

To understand this process, let's trace the cal to sequence(3) from the previous section. At the start of the trace, we have a local variable, n (the parameter), with an initial value of 3. Since 3 is not equal to 1, the while loop body is executed. 3 is printed and 3 % 2 == 0 is evaluated. Since it evaluates to False, the else branch is executed and 3 * 3 + 1 is evaluated and assigned to n.

To keep track of al this as you hand trace a program, make a column heading on a piece of paper for each variable created as the program runs and another one for output. Our trace so far would look something like this:

```
n  output
-- ------
3  3
10
```

Since 10 != 1 evaluates to True, the loop body is again executed, and 10 is printed. 10 % 2 == 0 is true, so the if branch is executed and n becomes 5. By the end of the trace we have:

```
n   output
--  ------
3   3
10  10
5   5
16  16
8   8
4   4
2   2
1
```

Tracing can be a bit tedious and error prone (that's why we get computers to do this stuff in the first place!), but it is an essential skil for a programmer to have. From this trace we can learn a lot about the way our code works. We can observe that as soon as n becomes a power of 2, for example, the program wil require log2(n) executions of the loop body to complete. We can also see that the final 1 wil not be printed as output.

6.5. Counting digits

The fol owing function counts the number of decimal digits in a positive integer expressed in decimal format:

def num_digits(n):

count = 0

while n:

count = count + 1

n = n / 10

return count

A cal to num_digits(710) wil return 3. Trace the execution of this function cal to convince yourself that it works.

This function demonstrates another pattern of computation cal ed a counter. The variable count is initialized to 0 and then incremented each time the loop body is executed. When the loop exits, count contains the result – the total number of times the loop body was executed, which is the same as the number of digits.

If we wanted to only count digits that are either 0 or 5, adding a conditional before incrementing the counter wil do the trick:

def num_zero_and_five_digits(n):

```
count = 0
while n:
    digit = n % 10
    if digit == 0 or digit == 5:
        count = count + 1
    n = n / 10
return count
```

Confirm that num_zero_and_five_digits(1055030250) returns 7.

6.6. Abbreviated assignment

Incrementing a variable is so common that Python provides an abbreviated syntax for it:

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count += 1
>>> count
2
>>>
```

count += 1 is an abreviation for count = count + 1 . The increment value does not have to be 1:

```
>>> n = 2
>>> n += 5
```

>>> n

7>>>

There are also abbreviations for -=, *=, /=, and %=:

>>> n = 2

>>> n *= 5

>>> n

10

>>> n -= 4

>>> n

6**>>>** n /= 2

>>> n

3**>>>** n %= 2

>>> n

1

6.7. Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions.

Creating the tables was slow and boring, and they tended to be ful of errors.

When computers appeared on the scene, one of the initial reactions was, This is great! We can use the computers to generate the tables, so there wil be no errors. That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Wel , almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium used to perform floating-point division.

Although a log table is not as useful as it once was, it stil makes a good example of iteration. The fol owing program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

x = 1

while x < 13:

print x, ' \t', 2**x

x += 1

The string '\t' represents a tab character. The backslash character in '\t' indicates the beginning of an escape sequence. Escape sequences are used to represent invisible characters like tabs and newlines.

The sequence \n represents a newline.

An escape sequence can appear anywhere in a string; in this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker cal ed the cursor keeps track of where the next character wil go. After a print statement, the cursor normal y goes to the beginning of the next line.

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program:

1 2

2 4

3 8

4 16

5 32

6 64

7 128

8 256

9 512

10 1024

11 2048

12 4096

Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

6.8. Two-dimensional tables

A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.

A good way to start is to write a loop that prints the multiples of 2, al on one line: i = 1

while i <= 6:

print 2 * i, ' ',

i += 1

print

The first line initializes a variable named i, which acts as a counter or loop variable. As the loop executes, the value of i increases from 1 to 6. When i is 7, the loop terminates. Each time through the loop, it displays the value of 2 * i, fol owed by three spaces.

Again, the comma in the print statement suppresses the newline. After the loop completes, the second print statement starts a new line.

The output of the program is:

2 4 6 8 10 12

So far, so good. The next step is to encapsulate and generalize.

6.9. Encapsulation and generalization

Encapsulation is the process of wrapping a piece of code in a function, al owing you to take advantage of al the things functions are good for. You have already seen two examples of encapsulation: print_parity in chapter 4; and is_divisible in chapter 5.

Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer.

This function encapsulates the previous loop and generalizes it to print multiples of n: **def** print_multiples(n):

i = 1

while i <= 6:

print n * i, ' \t',

i += 1

print

To encapsulate, al we had to do was add the first line, which declares the name of the function and the parameter list. To generalize, al we had to do was replace the value 2 with the parameter n.

If we cal this function with the argument 2, we get the same output as before. With the argument 3, the output is:

3 6 9 12 15 18

With the argument 4, the output is:

4 8 12 16 20 24

By now you can probably guess how to print a multiplication table — by cal ing print_multiples repeatedly with different arguments. In fact, we can use another loop:

i = 1
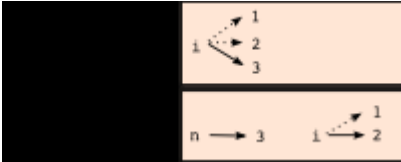
while i <= 6:

print_multiples(i)

i += 1

Notice how similar this loop is to the one inside print_multiples. Al we did was replace the print statement with a function cal .

The output of this program is a multiplication table:

1 2 3 4 5 6

2 4 6 8 10 12

3 6 9 12 15 18

4 8 12 16 20 24

5 10 15 20 25 30

6 12 18 24 30 36

6.10. More encapsulation

To demonstrate encapsulation again, let's take the code from the last section and wrap it up in a function:

def print_mult_table():

i = 1

while i <= 6:

print_multiples(i)

i += 1

This process is a common development plan. We develop code by writing lines of code outside any function, or typing them in to the interpreter. When we get the code working, we extract it and wrap it up in a function.

This development plan is particularly useful if you don't know how to divide the program into functions when you start writing. This approach lets you design as you go along.

6.11. Local variables

You might be wondering how we can use the same variable, i, in both print_multiples and print_mult_table. Doesn't it cause problems when one of the functions changes the value of the variable?

The answer is no, because the i in print_multiples and the i in print_mult_table are not the same variable.

Variables created inside a function definition are local; you can't access a local variable from outside its home function. That means you are free to have multiple variables with the same name as long as they are not in the same function.

The stack diagram for this program shows that the two variables named i are not the same variable.

They can refer to different values, and changing one does not affect the other.

The value of i in print_mult_table goes from 1 to 6. In the diagram it happens to be 3. The next time through the loop it wil be 4. Each

time through the loop, print_mult_table cal s print_multiples with the current value of i as an argument. That value gets assigned to the parameter n.

Inside print_multiples, the value of i goes from 1 to 6. In the diagram, it happens to be 2. Changing this variable has no effect on the value of i in print_mult_table.

It is common and perfectly legal to have different local variables with the same name. In particular, names like i and j are used frequently as loop variables. If you avoid using them in one function just because you used them somewhere else, you wil probably make the program harder to read.

6.12. More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the six-by-six table. You could add a parameter to print_mult_table:
def print_mult_table(high):

i = 1

while i <= high:

print_multiples(i)

i += 1

We replaced the value 6 with the parameter high. If we cal print_mult_table with the argument 7, it displays:

1 2 3 4 5 6

2 4 6 8 10 12

3 6 9 12 15 18

4 8 12 16 20 24

5 10 15 20 25 30

6 12 18 24 30 36

7 14 21 28 35 42

This is fine, except that we probably want the table to be square — with the same number of rows and columns. To do that, we add another parameter to print_multiples to specify how many columns the table should have.

Just to be annoying, we cal this parameter high, demonstrating that different functions can have parameters with the same name (just like local variables). Here's the whole program: **def** print_multiples(n, high):

i = 1

while i <= high:

print n*i, ' \t',

i += 1

print

def print_mult_table(high):

i = 1

while i <= high:

print_multiples(i, high)

i += 1

Notice that when we added a new parameter, we had to change the first line of the function (the function heading), and we also had to change the place where the function is cal ed in print_mult_table.

As expected, this program generates a square seven-by-seven table:

1 2 3 4 5 6 7

2 4 6 8 10 12 14

3 6 9 12 15 18 21

4 8 12 16 20 24 28

5 10 15 20 25 30 35

6 12 18 24 30 36 42

7 14 21 28 35 42 49

When you generalize a function appropriately, you often get a program with capabilities you didn't plan.

For example, you might notice that, because ab = ba, al the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of print_mult_table. Change

print_multiples(i, high)

to

print_multiples(i, i)

and you get:

1

2 4

3 6 9

4 8 12 16

5 10 15 20 25

6 12 18 24 30 36

7 14 21 28 35 42 49

6.13. Functions

A few times now, we have mentioned al the things functions are good for. By now, you might be wondering what exactly those things are. Here are some of them:

1. Giving a name to a sequence of statements makes your program easier to read and debug.

2. Dividing a long program into functions al ows you to separate parts of the program, debug them in isolation, and then compose them into a whole.

3. Functions facilitate the use of iteration.

4. Wel -designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

6.14. Newton's Method

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of n. If you start with almost any approximation, you can compute a better approximation with the fol owing formula:

better = (approx + n/approx)/2

By repeatedly applying this formula until the better approximation is equal to the previous one, we can write a function for computing the square root:

def sqrt(n):

approx = n/2.0

better = (approx + n/approx)/2.0

while better != approx:

approx = better

better = (approx + n/approx)/2.0

return approx

Try cal ing this function with 25 as an argument to confirm that it returns 5.0.

6.15. Algorithms

Newton's method is an example of an algorithm: it is a mechanical process for solving a category of problems (in this case, computing square roots).

It is not easy to define an algorithm. It might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were lazy, you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write n - 1 as the first digit and 10 - n as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are al algorithms. One of the characteristics of algorithms is that they do not require any intel igence to carry out. They are mechanical processes in which each step fol ows from the last according to a simple set of rules.

In our opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literal y, require no intel igence.

On the other hand, the process of designing algorithms is interesting, intel ectual y chal enging, and a central part of what we cal programming.

Some of the things that people do natural y, without difficulty or conscious thought, are the hardest to express algorithmical y. Understanding natural language is a good example. We al do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

6.16. Glossary

algorithm

A step-by-step process for solving a category of problems.

body

The statements inside a loop.

counter

A variable used to count something, usual y initialized to zero and incremented in the body of a loop.

cursor

An invisible marker that keeps track of where the next character wil be printed.

decrement

Decrease by 1.

development plan

A process for developing a program. In this chapter, we demonstrated a style of development based on developing code to do simple, specific things and then encapsulating and generalizing.

encapsulate

To divide a large complex program into components (like functions) and isolate the components from each other (by using local variables, for example).

escape sequence

An escape character, \, fol owed by one or more printable characters used to designate a nonprintable character.

generalize

To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

increment

Both as a noun and as a verb, increment means to increase by 1.

infinite loop

A loop in which the terminating condition is never satisfied.

initialization (of a variable)

To initialize a variable is to give it an initial value, usual y in the context of multiple assignment.

Since in Python variables don't exist until they are assigned values, they are initialized when they are created. In other programming languages this is not the case, and variables can be created without being initialized, in which case they have either default or garbage values.

iteration

Repeated execution of a set of programming statements.

loop

A statement or group of statements that execute repeatedly until a terminating condition is satisfied.

loop variable

A variable used as part of the terminating condition of a loop.

multiple assignment

Making more than one assignment to the same variable during the execution of a program.

newline

A special character that causes the cursor to move to the beginning of the next line.

tab

A special character that causes the cursor to move to the next tab stop on the current line.

trace

To fol ow the flow of execution of a program by hand, recording the change of state of the variables and any output produced.

6.17. Exercises

1. Write a single string that:

produces

this

output.

2. Add a print statement to the sqrt function defined in section 6.14 that prints out better each time it is calculated. Cal your modified function with 25 as an argument and record the results.

3. Trace the execution of the last version of print_mult_table and figure out how it works.

4. Write a function print_triangular_numbers(n) that prints out the first n triangular numbers. A cal to print_triangular_numbers(5) would produce the fol owing output:

1 1

2 3

3 6

4 10

5 15

(hint: use a web search to find out what a triangular number is.)

5. Open a file named ch06.py and add the fol owing:

if __name__ == '__main__':

import doctest

doctest.testmod()

Write a function, is_prime, which takes a single integral argument and returns True when the argument is a prime number and False otherwise. Add doctests to your function as you develop it.

6. What wil num_digits(0) return? Modify it to return 1 for this case. Why does a cal to num_digits(-24) result in an infinite loop (hint: -1/10 evaluates to -1)? Modify num_digits so that it works correctly with any integer value. Add the fol owing to the ch06.py file you created in the previous exercise:

def num_digits(n):

"""

```
>>> num_digits(12345)

5

>>> num_digits(0)

1

>>> num_digits(-12345)

5

"""
```

Add your function body to num_digits and confirm that it passes the doctests.

7. Add the fol owing to the ch06.py:

def num_even_digits(n):

```
"""

>>> num_even_digits(123456)

3

>>> num_even_digits(2468)

4

>>> num_even_digits(1357)

0

>>> num_even_digits(2)

1
```

```
>>> num_even_digits(20)
2
"""
```

Write a body for num_even_digits so that it works as expected.

8. Add the fol owing to ch06.py:

def print_digits(n):

```
"""

>>> print_digits(13789)
9 8 7 3 1
>>> print_digits(39874613)
3 1 6 4 7 8 9 3
>>> print_digits(213141)
1 4 1 3 1 2
"""
```

Write a body for print_digits so that it passes the given doctests.

9. Write a function sum_of_squares_of_digits that computes the sum of the squares of the digits of an integer passed to it. For example, sum_of_squares_of_digits(987) should return 194, since 9**2 + 8**2 + 7**2 == 81 + 64 + 49 == 194.

def sum_of_squares_of_digits(n):

```
"""
```

```
>>> sum_of_squares_of_digits(1)
1
>>> sum_of_squares_of_digits(9)
81
>>> sum_of_squares_of_digits(11)
2
>>> sum_of_squares_of_digits(121)
6
>>> sum_of_squares_of_digits(987)
194
"""
```

Check your solution against the doctests above.

7. Strings

7.1. A compound data type

So far we have seen five types: int, float, bool, NoneType and str. Strings are qualitatively different from the other four because they are made up of smal er pieces — characters.

Types that comprise smal er pieces are cal ed compound data types. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

The bracket operator selects a single character from a string:

>>> fruit = "banana"

>>> letter = fruit[1]

>>> print letter

The expression fruit[1] selects character number 1 from fruit. The variable letter refers to the result. When we display letter, we get a surprise:

a

The first letter of "banana" is not a, unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter (zero-eth) of "banana" is b. The 1th letter (

one-eth) is a, and the 2th (two-eth) letter is n.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets:

>>> letter = fruit[0]

>>> print letter

b

The expression in brackets is cal ed an index. An index specifies a member of an ordered set, in this case the set of characters in the string. The index indicates which one you want, hence the name. It can be any integer expression.

7.2. Length

The len function returns the number of characters in a string:

>>> fruit = "banana"

>>> len(fruit)

6

To get the last letter of a string, you might be tempted to try something like this: length = len(fruit)

last = fruit[length] # *ERROR!*

That won't work. It causes the runtime error IndexError: string index out of range. The reason is that there is no 6th letter in "banana". Since we started counting at zero, the six letters are numbered 0

to 5. To get the last character, we have to subtract 1 from length:

length = len(fruit)

last = fruit[length-1]

Alternatively, we can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

7.3. Traversal and the **for** loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn,

do something to it, and continue until the end. This pattern of processing is cal ed a traversal. One way to encode a traversal is with a while statement: index = 0

while index < len(fruit):

letter = fruit[index]

print letter

index += 1

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

Using an index to traverse a set of values is so common that Python provides an alternative, simpler syntax — the for loop:

for char **in** fruit:

print char

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

The fol owing example shows how to use concatenation and a for loop to generate an abecedarian series. Abecedarian refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book Make Way for Ducklings, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order: prefixes = "JKLMNOPQ"

suffix = "ack"

for letter **in** prefixes:

print letter + suffix

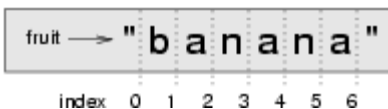The output of this program is:

Jack

Kack

Lack

Mack

Nack

Oack

Pack

Qack

Of course, that's not quite right because Ouack and Quack are misspel ed. You'l fix this as an exercise below.

7.4. String slices

A substring of a string is cal ed a slice. Selecting a slice is similar to selecting a character:

>>> s = "Peter, Paul, and Mary"

>>> **print** s[0:5]

Peter

>>> print s[7:11]

Paul

>>> print s[17:21]

Mary

The operator [n:m] returns the part of the string from the n-eth character to the m-eth character, including the first but excluding the last. This behavior is counterintuitive; it makes more sense if you imagine the indices pointing between the characters, as in the fol owing diagram:

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

>>> fruit = "banana"

>>> fruit[:3]

'ban'

>>> fruit[3:]

'ana'

What do you think s[:] means?

7.5. String comparison

The comparison operators work on strings. To see if two strings are equal:

if word == "banana":

print "Yes, we have no bananas!"

Other comparison operations are useful for putting words in [lexigraphical order:](#)

if word < "banana":

print "Your word, " + word + ", comes before banana."

elif word > "banana":

print "Your word, " + word + ", comes after banana."

else:

print "Yes, we have no bananas!"

This is similar to the alphabetical order you would use with a dictionary, except that al the uppercase letters come before al the lowercase letters. As a result:

Your word, Zebra, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as al lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

7.6. Strings are immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

greeting = "Hello, world!"

greeting[0] = 'J' # *ERROR!*

print greeting

Instead of producing the output Jello, world!, this code produces the runtime error TypeError: 'str'

object doesn't support item assignment.

Strings are immutable, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

greeting = "Hello, world!"

new_greeting = 'J' + greeting[1:]

print new_greeting

The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

7.7. The **in** operator

The in operator tests if one string is a substring of another:

>>> 'p' **in** 'apple'

True

>>> 'i' **in** 'apple'

False

>>> 'ap' **in** 'apple'

True

>>> 'pa' **in** 'apple'

False

Note that a string is a substring of itself:

>>> 'a' **in** 'a'

True

>>> 'apple' **in** 'apple'

True

Combining the in operator with string concatenation using +, we can write a function that removes al the vowels from a string:

def remove_vowels(s):

vowels = "aeiouAEIOU"

s_without_vowels = ""

for letter **in** s:

if letter **not in** vowels:

s_without_vowels += letter

return s_without_vowels

Test this function to confirm that it does what we wanted it to do.

7.8. A **find** function

What does the fol owing function do?

def find(strng, ch):

index = 0

while index < len(strng):

if strng[index] == ch:

return index

index += 1

return -1

In a sense, find is the opposite of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.

This is the first example we have seen of a return statement inside a loop. If strng[index] == ch, the function returns immediately, breaking out of the loop prematurely.

If the character doesn't appear in the string, then the program exits the loop normal y and returns -1.

This pattern of computation is sometimes cal ed a eureka traversal because as soon as we find what we are looking for, we can cry Eureka! and stop looking.

7.9. Looping and counting

The fol owing program counts the number of times the letter a appears in a string, and is another example of the counter pattern introduced in [Counting digits](): fruit = "banana"

count = 0

for char **in** fruit:

if char == 'a':

count += 1

print count

7.10. Optional parameters

To find the locations of the second or third occurence of a character in a string, we can modify the find function, adding a third parameter for the starting postion in the search string:

def find2(strng, ch, start):

index = start

while index < len(strng):

if strng[index] == ch:

return index

index += 1

return -1

The cal find2('banana', 'a', 2) now returns 3, the index of the first occurance of 'a' in 'banana' after index 2. What does find2('banana', 'n', 3) return? If you said, 4, there is a good chance you understand how find2 works.

Better stil , we can combine find and find2 using an optional parameter:

def find(strng, ch, start=0):

index = start

while index < len(strng):

if strng[index] == ch:

return index

index += 1

return -1

The cal find('banana', 'a', 2) to this version of find behaves just like find2, while in the cal find('banana', 'a'), start wil be set to the default value of 0.

Adding another optional parameter to find makes it search both forward and backward: **def** find(strng, ch, start=0, step=1):

index = start

while 0 <= index < len(strng):

if strng[index] == ch:

return index

index += step

return -1

Passing in a value of len(strng)-1 for start and -1 for step wil make it search toward the beginning of the string instead of the end. Note that we needed to check for a lower bound for index in the while loop as wel as an upper bound to accomodate this change.

7.11. The **string** module

The string module contains useful functions that manipulate strings. As usual, we have to import the module before we can use it:

>>> import string

To see what is inside it, use the dir function with the module name as an argument.

>>> dir(string)

which wil return the list of items inside the string module:

['Template', '_TemplateMetaclass', '__builtins__', '__doc__', '__file__', '__name__',

'_float', '_idmap', '_idmapL', '_int', '_long', '_multimap', '_re', 'ascii_letters',

'ascii_lowercase', 'ascii_uppercase', 'atof', 'atof_error', 'atoi', 'atoi_error', 'atol',

'atol_error', 'capitalize', 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',

'hexdigits', 'index', 'index_error', 'join', 'joinfields', 'letters', 'ljust', 'lower',

'lowercase', 'lstrip', 'maketrans', 'octdigits', 'printable', 'punctuation', 'replace',

'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitfields', 'strip', 'swapcase',

'translate', 'upper', 'uppercase', 'whitespace', 'zfill']

To find out more about an item in this list, we can use the type command. We need to specify the module name fol owed by the item using dot notation.

>>> type(string.digits)

>>> type(string.find)

<type 'function'>

Since string.digits is a string, we can print it to see what it contains:

>>> print string.digits

0123456789

Not surprisingly, it contains each of the decimal digits.

string.find is a function which does much the same thing as the function we wrote. To find out more about it, we can print out its docstring, __doc__, which contains documentation on the function:

>>> print string.find.__doc__

find(s, sub [,start [,end]]) -> in

Return the lowest index in s where substring sub is found,

such that sub is contained within s[start,end]. Optional

arguments start and end are interpreted as in slice notation.

Return -1 on failure.

The parameters in square brackets are optional parameters. We can use string.find much as we did our own find:

>>> fruit = "banana"

>>> index = string.find(fruit, "a")

>>> print index

1

This example demonstrates one of the benefits of modules — they help avoid col isions between the names of built-in functions and user-defined functions. By using dot notation we can specify which version of find we want.

Actual y, string.find is more general than our version. it can find substrings, not just characters:

>>> string.find("banana", "na")

2

Like ours, it takes an additional argument that specifies the index at which it should start:

>>> string.find("banana", "na", 3)

4

Unlike ours, its second optional parameter specifies the index at which the search should end:

>>> string.find("bob", "b", 1, 2)

-1

In this example, the search fails because the letter b does not appear in the index range from 1 to 2 (not including 2).

7.12. Character classification

It is often helpful to examine a character and test whether it is upper- or lowercase, or whether it is a character or a digit. The string module provides several constants that are useful for these purposes.

One of these, string.digits, we have already seen.

The string string.lowercase contains al of the letters that the system considers to be lowercase.

Similarly, string.uppercase contains al of the uppercase letters. Try the fol owing and see what you get: **print** string.lowercase

print string.uppercase

print string.digits

We can use these constants and find to classify characters. For example, if find(lowercase, ch) returns a value other than -1, then ch must be lowercase:

def is_lower(ch):

return string.find(string.lowercase, ch) != -1

Alternatively, we can take advantage of the in operator:

def is_lower(ch):

return ch **in** string.lowercase

As yet another alternative, we can use the comparison operator:

def is_lower(ch):

return 'a' <= ch <= 'z'

If ch is between a and z, it must be a lowercase letter.

Another constant defined in the string module may surprise you when you print it:

>>> print string.whitespace

Whitespace characters move the cursor without printing anything. They create the white space between visible characters (at least on white paper). The constant string.whitespace contains al the whitespace characters, including space, tab (\t), and newline (\n).

There are other useful functions in the string module, but this book isn't intended to be a reference manual. On the other hand, the Python Library Reference is. Along with a wealth of other documentation, it's available from the Python website, http://www.python.org.

7.13. String formatting

The most concise and powerful way to format a string in Python is to use the string formatting operator,

%, together with Python's string formatting operations. To see how this works, let's start with a few examples:

>>> "His name is *%s*." % "Arthur"

'His name is Arthur.'

>>> name = "Alice"

>>> age = 10

>>> "I am *%s* and I am *%d* years old." % (name, age)

'I am Alice and I am 10 years old.'

>>> n1 = 4

>>> n2 = 5

>>> "2**10 = *%d* and *%d* * *%d* = *%f*" % (2**10, n1, n2, n1 * n2)

'2**10 = 1024 and 4 * 5 = 20.000000'

>>>

The syntax for the string formatting operation looks like this:

"<FORMAT>" % (<VALUES>)

It begins with a format which contains a sequence of characters and conversion specifications.

Conversion specifications start with a % operator. Fol owing the format string is a single % and then a sequence of values, one per conversion specification, seperated by commas and enclosed in parenthesis. The parenthesis are optional if there is only a single value.

In the first example above, there is a single conversion specification, %s, which indicates a string. The single value, "Arthur", maps to it, and is not enclosed in parenthesis.

In the second example, name has string value, "Alice", and age has integer value, 10. These map to the two converstion specifications, %s and %d. The d in the second converstion specification indicates that the value is a decimal integer.

In the third example variables n1 and n2 have integer values 4 and 5 respectively. There are four converstion specifications in the format string: three %d's and a %f. The f indicates that the value should be represented as a floating point number. The four values that map to the four converstion specifications are: 2**10, n1, n2, and n1 * n2.

[s, d, and f are al the conversion types we wil need for this book. To see a complete list, see the String](#)

[Formatting Operations section of the Python Library Reference.](#)

The fol owing example il ustrates the real utility of string formatting:

```
i = 1

print "i\ti**2\ti**3\ti**5\ti**10\ti**20"

while i <= 10:

print i, ' \t', i**2, ' \t', i**3, ' \t', i**5, ' \t', i**10, ' \t', i**20

i += 1
```

This program prints out a table of various powers of the numbers from 1 to 10. In its current form it relies on the tab character (\t) to align the columns of values, but this breaks down when the values in the table get larger than the 8 character tab width:

i i**2 i**3 i**5 i**10 i**20

1 1 1 1 1 1

2 4 8 32 1024 1048576

3 9 27 243 59049 3486784401

4 16 64 1024 1048576 1099511627776

5 25 125 3125 9765625 95367431640625

6 36 216 7776 60466176 3656158440062976

7 49 343 16807 282475249 79792266297612001

8 64 512 32768 1073741824 1152921504606846976

9 81 729 59049 3486784401 12157665459056928801

10 100 1000 100000 10000000000 100000000000000000000

One possible solution would be to change the tab width, but the first column already has more space than it needs. The best solution would be to set the width of each column independently. As you may have guessed by now, string formatting provides the solution:

```
i = 1

print " %-4s%-5s%-6s%-8s%-13s%-15s" % \

('i', 'i**2', 'i**3', 'i**5', 'i**10', 'i**20')

while i <= 10:

print " %-4d%-5d%-6d%-8d%-13d%-15d" % (i, i**2, i**3, i**5, i**10, i**20) i += 1
```

Running this version produces the fol owing output:

```
i i**2 i**3 i**5 i**10 i**20

1 1 1 1 1 1

2 4 8 32 1024 1048576

3 9 27 243 59049 3486784401

4 16 64 1024 1048576 1099511627776

5 25 125 3125 9765625 95367431640625

6 36 216 7776 60466176 3656158440062976

7 49 343 16807 282475249 79792266297612001

8 64 512 32768 1073741824 1152921504606846976

9 81 729 59049 3486784401 12157665459056928801
```

10 100 1000 100000 10000000000 100000000000000000000

The - after each % in the converstion specifications indicates left justification. The numerical values specify the minimum length, so %-13d is a left justified number at least 13 characters wide.

7.14. Summary and First Exercises

This chapter introduced a lot of new ideas. The fol owing summary and set of exercises may prove helpful in remembering what you learned:

indexing (**[]**)

Access a single character in a string using its position (starting from 0). Example: 'This'[2]

evaluates to 'i'.

length function (**len**)

Returns the number of characters in a string. Example: len('happy') evaluates to 5.

for loop traversal (**for**)

Traversing a string means accessing each character in the string, one at a time. For example, the fol owing for loop:

for letter **in** 'Example':

print 2 * letter,

evaluates to EE xx aa mm pp ll ee

slicing (**[:]**)

A slice is a substring of a string. Example: 'bananas and cream'[3:6] evaluates to ana (so does

'bananas and cream'[1:4]).

string comparison (**>, <, >=, <=, ==**)

The comparision operators work with strings, evaluating according to [lexigraphical order.](#)

Examples: 'apple' < 'banana' evaluates to True. 'Zeta' < 'Appricot' evaluates to False . 'Zebra'

<= 'aardvark' evaluates to True because al upper case letters precede lower case letters.

in operator (**in**)

The in operator tests whether one character or string is contained inside another string. Examples:

'heck' in "I'll be checking for you." evaluates to True. 'cheese' in "I'll be checking for you." evaluates to False.

7.14.1. First Exercises

1. Write the Python interpreter's evaluation to each of the fol owing expressions:

>>> 'Python'[1]

>>> "Strings are sequences of characters."[5]

>>> len("wonderful")

>>> 'Mystery'[:4]

>>> 'p' **in** 'Pinapple'

>>> 'apple' **in** 'Pinapple'

>>> 'pear' **in** 'Pinapple'

>>> 'apple' > 'pinapple'

>>> 'pinapple' < 'Peach'

2. Write Python code to make each of the fol owing doctests pass:

```
"""

>>> type(fruit)

<type 'str'>

>>> len(fruit)

8

>>> fruit[:3]

'ram'

"""

"""

>>> group = "John, Paul, George, and Ringo"

>>> group[12:x]

'George'

>>> group[n:m]

'Paul'

>>> group[:r]
```

'John'

>>> group[s:]

'Ringo'

"""

"""

>>> len(s)

8

>>> s[4:6] == 'on'

True

"""

7.15. Glossary

compound data type

A data type in which the values are made up of components, or elements, that are themselves values.

default value

The value given to an optional parameter if no argument for it is provided in the function cal .

docstring

A string constant on the first line of a function or module definition (and as we wil see later, in class and method definitions as wel). Docstrings provide a convenient way to associate documentation

with code. Docstrings are also used by the doctest module for automated testing.

dot notation

Use of the dot operator, ., to access functions inside a module.

immutable

A compound data type whose elements cannot be assigned new values.

index

A variable or value used to select a member of an ordered set, such as a character from a string.

optional parameter

A parameter written in a function header with an assignment to a default value which it wil receive if no corresponding argument is given for it in the function cal .

slice

A part of a string (substring) specified by a range of indices. More general y, a subsequence of any sequence type in Python can be created using the slice operator (sequence[start:stop]).

traverse

To iterate through the elements of a set, performing a similar operation on each.

whitespace

Any of the characters that move the cursor without printing visible characters. The constant string.whitespace contains al the white-

space characters.

7.16. Exercises

1. Modify:

```
prefixes = "JKLMNOPQ"

suffix = "ack"

for letter in prefixes:

print letter + suffix
```

so that Ouack and Quack are spel ed correctly.

2. Encapsulate

```
fruit = "banana"

count = 0

for char in fruit:

if char == 'a':

count += 1

print count
```

in a function named count_letters, and generalize it so that it accepts the string and the letter as arguments.

3. Now rewrite the count_letters function so that instead of traversing the string, it repeatedly cal s find (the version from <u>Optional parameters</u>), with the optional third parameter to locate new occurences of the letter being counted.

4. Which version of is_lower do you think wil be fastest? Can you think of other reasons besides speed to prefer one version or the other?

5. Create a file named stringtools.py and put the fol owing in it:

```python
def reverse(s):
    """

    >>> reverse('happy')
    'yppah'
    >>> reverse('Python')
    'nohtyP'
    >>> reverse("")
    ''
    >>> reverse("P")
    'P'
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Add a function body to reverse to make the doctests pass.

6. Add mirror to stringtools.py .

```python
def mirror(s):
```

"""

    >>> mirror("good")

    'gooddoog'

    >>> mirror("yes")

    'yessey'

    >>> mirror('Python')

    'PythonnohtyP'

    >>> mirror("")

    ''

    >>> mirror("a")

    'aa'

    """

Write a function body for it that wil make it work as indicated by the doctests.

7. Include remove_letter in stringtools.py .

def remove_letter(letter, strng):

    """

    >>> remove_letter('a', 'apple')

    'pple'

    >>> remove_letter('a', 'banana')

'bnn'

>>> remove_letter('z', 'banana')

'banana'

>>> remove_letter('i', 'Mississippi')

'Msssspp'

"""

Write a function body for it that wil make it work as indicated by the doctests.

8. Final y, add bodies to each of the fol owing functions, one at a time

def is_palindrome(s):

"""

>>> is_palindrome('abba')

True

>>> is_palindrome('abab')

False

>>> is_palindrome('tenet')

True

>>> is_palindrome('banana')

False

>>> is_palindrome('straw warts')

True

"""

def count(sub, s):

"""

>>> count('is', 'Mississippi')

2

>>> count('an', 'banana')

2

>>> count('ana', 'banana')

2

>>> count('nana', 'banana')

1

>>> count('nanan', 'banana')

0

"""

def remove(sub, s):

"""

>>> remove('an', 'banana')

'bana'

>>> remove('cyc', 'bicycle')

'bile'

>>> remove('iss', 'Mississippi')

'Missippi'

>>> remove('egg', 'bicycle')

'bicycle'

"""

def remove_all(sub, s):

"""

>>> remove_all('an', 'banana')

'ba'

>>> remove_all('cyc', 'bicycle')

'bile'

>>> remove_all('iss', 'Mississippi')

'Mippi'

>>> remove_all('eggs', 'bicycle')

'bicycle'

"""

until al the doctests pass.

9. Try each of the fol owing formatted string operations in a Python shel and record the results:

1. "%s %d %f" % (5, 5, 5)

2. "%-.2f" % 3

3. "%-10.2f%-10.2f" % (7, 1.0/2)

4. print " $%5.2fn $%5.2fn $%5.2f" % (3, 4.5, 11.2)

10. The fol owing formatted strings have errors. Fix them:

1. "%s %s %s %s" % ('this', 'that', 'something')

2. "%s %s %s" % ('yes', 'no', 'up', 'down')

3. "%d %f %f" % (3, 3, 'three')
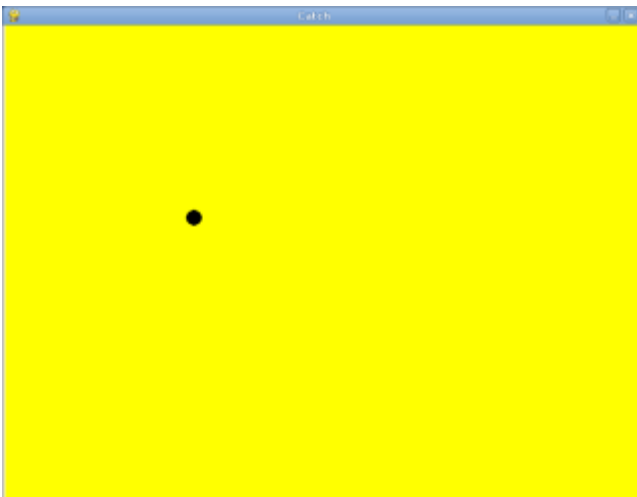
8. Case Study: Catch

8.1. Getting started

In our first case study we wil build a smal video game using the facilities in the GASP package. The game wil shoot a bal across a window from left to right and you wil manipulate a mitt at the right side of the window to catch it.

8.2. Using **while** to move a ball

while statements can be used with gasp to add motion to a program. The fol owing program moves a black bal across an 800 x 600 pixel graphics canvas. Add this to a file named pitch.py: **from gasp import** *

```
begin_graphics(800, 600, title="Catch", background=color.YELLOW)

set_speed(120)

ball_x = 10

ball_y = 300

ball = Circle((ball_x, ball_y), 10, filled=True)

dx = 4

dy = 1

while ball_x < 810:

ball_x += dx

ball_y += dy

move_to(ball, (ball_x, ball_y))
```

update_when('next_tick')

end_graphics()

As the bal moves across the screen, you wil see a graphics window that looks like this: Trace the first few iterations of this program to be sure you see what is happening to the variables x and y.

Some new things to learn about GASP from this example:

begin_graphics can take arguments for width, height, title, and background color of the graphics canvas.

set_speed can takes a frame rate in frames per second.

Adding filled=True to Circle(...) makes the resulting circle solid.

ball = Circle stores the circle (we wil talk later about what a circle actual y is) in a variable named ball so that it can be referenced later.

The move_to function in GASP al ows a programmer to pass in a shape (the bal in this case) and a location, and moves the shape to that location.

The update_when function is used to delay the action in a gasp program until a specified event occurs. The event 'next_tick' waits until the next frame, for an amount of time determined by the frame rate set with set_speed. Other valid arguments for update_when are 'key_pressed' and

'mouse_clicked'.

8.3. Varying the pitches

To make our game more interesting, we want to be able to vary the speed and direction of the bal .

GASP has a function, random_between(low, high), that returns a random integer between low and high.

To see how this works, run the fol owing program:

from gasp import *

i = 0

while i < 10:

print random_between(-5, 5)

i += 1

Each time the function is cal ed a more or less random integer is chosen between -5 and 5. When we ran this program we got:

-2

-1

-4

1-23-5-34-5

You wil probably get a different sequence of numbers.

Let's use random_between to vary the direction of the bal . Replace the line in pitch.py that assigns 1 to y:

dy = 1

with an assignment to a random number between -4 and 4:

dy = random_between(-4, 4)

8.4. Making the ball bounce

Running this new version of the program, you wil notice that bal frequently goes off either the top or bottom edges of the screen before it completes its journey. To prevent this, let's make the bal bounce off the edges by changing the sign of dy and sending the bal back in the opposite verticle direction.

Add the fol owing as the first line of the body of the while loop in pitch.py:

if ball_y >= 590 **or** ball_y <= 10:

dy *= -1

Run the program several times to see how it behaves.

8.5. The **break** statement

The break statement is used to immediately leave the body of a loop. The fol owing program implements a simple guessing game:

from gasp import *

number = random_between(1, 1000)

guesses = 1

guess = input("Guess the number between 1 and 1000: ")

while guess != number:

if guess > number:

print "Too high!"

else:

print "Too low!"

```
guess = input("Guess the number between 1 and 1000: ")

guesses += 1
```

print " \n\nCongratulations, you got it in *%d* guesses! \n\n" % guesses Using a break statement, we can rewrite this program to eliminate the duplication of the input statement:

from gasp import *

```
number = random_between(1, 1000)

guesses = 0
```

while True:

```
guess = input("Guess the number between 1 and 1000: ")

guesses += 1
```

if guess > number:

print "Too high!"

elif guess < number:

print "Too low!"

else:

print " \n\nCongratulations, you got it in *%d* guesses! \n\n" % guesses **break**

This program makes use of the mathematical law of trichotomy (given real numbers a and b, a > b, a < b, or a = b). While both versions of the program are 15 lines long, it could be argued that the logic in the second version is clearer.

Put this program in a file named guess.py.

8.6. Responding to the keyboard

The fol owing program creates a circle (or mitt) which responds to keyboard input. Pressing the j or k keys moves the mitt up and down, respectively. Add this to a file named mitt.py:

```
from gasp import *

begin_graphics(800, 600, title="Catch", background=color.YELLOW)

set_speed(120)

mitt_x = 780

mitt_y = 300

mitt = Circle((mitt_x, mitt_y), 20)

while True:

if key_pressed('k') and mitt_y <= 580:

mitt_y += 5

elif key_pressed('j') and mitt_y >= 20:

mitt_y -= 5

if key_pressed('escape'):

break

move_to(mitt, (mitt_x, mitt_y))

update_when('next_tick')

end_graphics()
```

Run mitt.py, pressing j and k to move up and down the screen.

8.7. Checking for collisions

The fol owing program moves two bal s toward each other from opposite sides of the screen. When they col ide , both bal s disappear and the program ends:

```
from gasp import *

def distance(x1, y1, x2, y2):

return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

begin_graphics(800, 600, title="Catch", background=color.YELLOW)

set_speed(120)

ball1_x = 10

ball1_y = 300

ball1 = Circle((ball1_x, ball1_y), 10, filled=True)

ball1_dx = 4

ball2_x = 790

ball2_y = 300

ball2 = Circle((ball2_x, ball2_y), 10)

ball2_dx = -4

while ball1_x < 810:

ball1_x += ball1_dx

ball2_x += ball2_dx
```

```
move_to(ball1, (ball1_x, ball1_y))

move_to(ball2, (ball2_x, ball2_y))

if distance(ball1_x, ball1_y, ball2_x, ball2_y) <= 20:

remove_from_screen(ball1)

remove_from_screen(ball2)

break

update_when('next_tick')

sleep(1)

end_graphics()
```

Put this program in a file named collide.py and run it.

8.8. Putting the pieces together

In order to combine the moving bal , moving mitt, and col ision detection, we need a single while loop that does each of these things in turn:

```
from gasp import *

def distance(x1, y1, x2, y2):

return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

begin_graphics(800, 600, title="Catch", background=color.YELLOW)

set_speed(120)

ball_x = 10

ball_y = 300
```

```
ball = Circle((ball_x, ball_y), 10, filled=True)

dx = 4

dy = random_between(-4, 4)

mitt_x = 780

mitt_y = 300

mitt = Circle((mitt_x, mitt_y), 20)

while True:

    # move the ball

    if ball_y >= 590 or ball_y <= 10:

        dy *= -1

    ball_x += dx

    if ball_x > 810: # the ball has gone off the screen

        break

    ball_y += dy

    move_to(ball, (ball_x, ball_y))

    # check on the mitt

    if key_pressed('k') and mitt_y <= 580:

        mitt_y += 5

    elif key_pressed('j') and mitt_y >= 20:

        mitt_y -= 5
```

```
if key_pressed('escape'):
    break
move_to(mitt, (mitt_x, mitt_y))
if distance(ball_x, ball_y, mitt_x, mitt_y) <= 30:  # ball is caught
    remove_from_screen(ball)
    break
update_when('next_tick')
end_graphics()
```

Put this program in a file named catch.py and run it several times. Be sure to catch the bal on some runs and miss it on others.

8.9. Displaying text

This program displays scores for both a player and the computer on the graphics screen. It generates a random number of 0 or 1 (like flipping a coin) and adds a point to the player if the value is 1 and to the computer if it is not. It then updates the display on the screen.

```
from gasp import *
begin_graphics(800, 600, title="Catch", background=color.YELLOW)
set_speed(120)
player_score = 0
comp_score = 0
player = Text("Player: %d Points" % player_score, (10, 570),
size=24) computer = Text("Computer: %d Points" % comp_score,
```

(640, 570), size=24) **while** player_score < 5 **and** comp_score < 5:

sleep(1)

winner = random_between(0, 1)

if winner:

player_score += 1

remove_from_screen(player)

player = Text("Player: *%d* Points" % player_score, (10, 570), size=24) **else**:

comp_score += 1

remove_from_screen(computer)

computer = Text("Computer: *%d* Points" % comp_score, (640, 570), size=24) **if** player_score == 5:

Text("Player Wins!", (340, 290), size=32)

else:

Text("Computer Wins!", (340, 290), size=32)

sleep(4)

end_graphics()

Put this program in a file named scores.py and run it.

We can now modify catch.py to diplay the winner. Immediately after the if ball_x > 810: conditional, add the fol owing:

Text("Computer Wins!", (340, 290), size=32)

sleep(2)

It is left as an excercise to display when the player wins.

8.10. Abstraction

Our program is getting a bit complex. To make matters worse, we are about to increase its complexity.

The next stage of development requires a nested loop. The outer loop wil handle repeating rounds of play until either the player or the computer reaches a winning score. The inner loop wil be the one we already have, which plays a single round, moving the bal and mitt, and determining if a catch or a miss has occured.

Research suggests there are clear limits to our ability to process cognitive tasks (see George A. Mil er's

[The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information](#)).

The more complex a program becomes, the more difficult it is for even an experienced programmer to develop and maintain.

To handle increasing complexity, we can wrap groups of related statements in functions, using abstraction to hide program details. This al ows us to mental y treat a group of programming statements as a single concept, freeing up mental bandwidth for further tasks. The ability to use abstraction is one of the most powerful ideas in computer programming.

Here is a completed version of catch.py:

```python
from gasp import *

COMPUTER_WINS = 1

PLAYER_WINS = 0
```

```python
QUIT = -1

def distance(x1, y1, x2, y2):

return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

def play_round():

ball_x = 10

ball_y = random_between(20, 280)

ball = Circle((ball_x, ball_y), 10, filled=True)

dx = 4

dy = random_between(-5, 5)

mitt_x = 780

mitt_y = random_between(20, 280)

mitt = Circle((mitt_x, mitt_y), 20)

while True:

if ball_y >= 590 or ball_y <= 10:

dy *= -1

ball_x += dx

ball_y += dy

if ball_x >= 810:

remove_from_screen(ball)

remove_from_screen(mitt)
```

```python
        return COMPUTER_WINS

    move_to(ball, (ball_x, ball_y))

    if key_pressed('k') and mitt_y <= 580:
        mitt_y += 5
    elif key_pressed('j') and mitt_y >= 20:
        mitt_y -= 5

    if key_pressed('escape'):
        return QUIT

    move_to(mitt, (mitt_x, mitt_y))

    if distance(ball_x, ball_y, mitt_x, mitt_y) <= 30:
        remove_from_screen(ball)
        remove_from_screen(mitt)
        return PLAYER_WINS

    update_when('next_tick')

def play_game():
    player_score = 0
    comp_score = 0
    while True:
        pmsg = Text("Player: %d Points" % player_score, (10, 570), size=24)
        cmsg = Text("Computer: %d Points" % comp_score, (640, 570), size=24)
        sleep(3)
```

```python
remove_from_screen(pmsg)

remove_from_screen(cmsg)

result = play_round()

if result == PLAYER_WINS:

player_score += 1

elif result == COMPUTER_WINS:

comp_score += 1

else:

return QUIT

if player_score == 5:

return PLAYER_WINS

elif comp_score == 5:

return COMPUTER_WINS

begin_graphics(800, 600, title="Catch", background=color.YELLOW)

set_speed(120)

result = play_game()

if result == PLAYER_WINS:

Text("Player Wins!", (340, 290), size=32)

elif result == COMPUTER_WINS:

Text("Computer Wins!", (340, 290), size=32)
```

sleep(4)

end_graphics()

Some new things to learn from this example:

Fol owing good organizational practices makes programs easier to read. Use the fol owing organization in your programs:

imports

global constants

function definitions

main body of the program

Symbolic constants like COMPUTER_WINS, PLAYER_WINS, and QUIT can be used to enhance readability of the program. It is customary to name constants with al capital letters. In Python it is up to the programmer to never assign a new value to a constant , since the language does not provide an easy way to enforce this (many other programming languages do).

We took the version of the program developed in section 8.8 and wrapped it in a function named play_round(). play_round makes use of the constants defined at the top of the program. It is much easier to remember COMPUTER_WINS than it is the arbitrary numeric value assigned to it.

A new function, play_game(), creates variables for player_score and comp_score. Using a while loop, it repeatedly cal s play_round, checking the result of each cal and updating the score appropriately. Final y, when either the player or computer reach 5 points, play_game returns the winner to the main body of the program, which then displays the winner and then quits.

There are two variables named result—one in the play_game function and one in the main body of the program. While they have the same name, they are in different namespaces, and bear no relation to each other. Each function creates its own namespace, and names defined within the body of the function are not visible to code outside the function body. Namespaces wil be discussed in greater detail in the next chapter.

8.11. Glossary

abstraction

Generalization by reducing the information content of a concept. Functions in Python can be used to group a number of program statements with a single name, abstracting out the details and making the program easier to understand.

constant

A numerical value that does not change during the execution of a program. It is conventional to use names with al uppercase letters to repesent constants, though Python programs rely on the discipline of the programmers to enforce this, since there is no language mechanism to support true constants in Python.

nested loop

A loop inside the body of another loop.

random

Having no specific pattern. Unpredictable. Computers are designed to be predicatable, and it is not possible to get a truly random value from a computer. Certain functions produce sequences of values that appear as if they were random, and it is these psuedorandom values that we get from Python.

trichotomy

Given any real numbers a and b, exactly one of the fol owing relations holds: a < b, a > b, or a = b.

Thus when you can establish that two of the relations are false, you can assume the remaining one is true.

8.12. Exercises

1. What happens when you press the key while running mitt.py? List the two lines from the program that produce this behavior and explain how they work.

2. What is the name of the counter variable in guess.py? With a proper strategy, the maximum number of guesses required to arrive at the correct number should be 11. What is this strategy?

3. What happens when the mitt in mitt.py gets to the top or bottom of the graphics window? List the lines from the program that control this behavior and explain in detail how they work.

4. Change the value of ball1_dx in collide.py to 2. How does the program behave differently? Now change ball1_dx back to 4 and set ball2_dx to -2. Explain in detail how these changes effect the behavior of the program.

5. Comment out (put a # in front of the statement) the break statement in collide.py. Do you notice any change in the behavior of the program? Now also comment out the

remove_from_screen(ball1) statement. What happens now? Experiment with commenting and uncommenting the two remove_from_screen statements and the break statement until you can describe specifically how these statements work together to produce the desired behavior in the program.

6. Where can you add the lines

Text("Player Wins!", (340, 290), size=32)

sleep(2)

to the version of catch.py in section 8.8 so that the program displays this message when the bal is caught?

7. Trace the flow of execution in the final version of catch.py when you press the escape key during the execution of play_round. What happens when you press this key? Why?

8. List the main body of the final version of catch.py. Describe in detail what each line of code does.

Which statement cal s the function that starts the game?

9. Identify the function responsible for displaying the bal and the mitt. What other operations are provided by this function?

10. Which function keeps track of the score? Is this also the function that displays the score? Justify your answer by discussing specific parts of the code which implement these operations.

8.13. Project: **pong.py**

[Pong](#) was one of the first commercial video games. With a capital P it is a registered trademark, but pong is used to refer any of the table tennis like paddle and bal video games.

catch.py already contains al the programming tools we need to develop our own version of pong.

Incremental y changing catch.py into pong.py is the goal of this project, which you wil accomplish by completing the fol owing series of exercises:

1. Copy catch.py to pong1.py and change the bal into a paddle by using Box instead of the Circle.

You can look at Appendix A for more information on Box. Make the adjustments needed to keep the paddle on the screen.

2. Copy pong1.py to pong2.py. Replace the distance function with a boolean function hit(bx, by, r, px, py, h) that returns True when the vertical coordinate of the bal (by) is between the bottom and top of the paddle, and the horizontal location of the bal (bx) is less than or equal to the radius (r) away from the front of the paddle. Use hit to determine when the bal hits the paddle, and make the bal bounce back in the opposite horizontal direction when hit returns True. Your completed function should pass these doctests:

def hit(bx, by, r, px, py, h):

"""

>>> hit(760, 100, 10, 780, 100, 100)

False

>>> hit(770, 100, 10, 780, 100, 100)

True

>>> hit(770, 200, 10, 780, 100, 100)

True

>>> hit(770, 210, 10, 780, 100, 100)

False

"""

Final y, change the scoring logic to give the player a point when the bal goes off the screen on the left.

3. Copy pong2.py to pong3.py. Add a new paddle on the left side of the screen which moves up when

'a' is pressed and down when 's' is pressed. Change the starting point for the bal to the center of the screen, (400, 300), and make it randomly move to the left or right at the start of each round.

9. Lists

A list is an ordered set of values, where each value is identified by an index. The values that make up a list are cal ed its elements. Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type. Lists and strings — and other things that behave like ordered sets — are cal ed sequences.

9.1. List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([

and]):

[10, 20, 30, 40]

["spam", "bungee", "swallow"]

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The fol owing list contains a string, a float, an integer, and (mirabile dictu) another list:

["hello", 2.0, 5, [10, 20]]

A list within another list is said to be nested.

Final y, there is a special list that contains no elements. It is cal ed the empty list, and is denoted [].

Like numeric 0 values and the empty string, the empty list is false in a boolean expression:

>>> if []:

... print 'This is true.'

... else:

... print 'This is false.'

...

This is false.

>>>

With al these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as parameters to functions. We can:

>>> vocabulary = ["ameliorate", "castigate", "defenestrate"]

>>> numbers = [17, 123]

>>> empty = []

>>> print vocabulary, numbers, empty

['ameliorate', 'castigate', 'defenestrate'] [17, 123] []

9.2. Accessing elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string—the bracket operator ([] – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0:

>>> print numbers[0]

17

Any integer expression can be used as an index:

>>> numbers[9-8]

123

>>> numbers[1.0]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: list indices must be integers

If you try to read or write an element that does not exist, you get a runtime error:

>>> numbers[2]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: list index out of range

If an index has a negative value, it counts backward from the end of the list:

>>> numbers[-1]

123

>>> numbers[-2]

17

>>> numbers[-3]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: list index out of range

numbers[-1] is the last element of the list, numbers[-2] is the second to last, and numbers[-3] doesn't exist.

It is common to use a loop variable as a list index.

horsemen = ["war", "famine", "pestilence", "death"]

i = 0

while i < 4:

print horsemen[i]

i += 1

This while loop counts from 0 to 4. When the loop variable i is 4, the condition fails and the loop terminates. So the body of the loop is

only executed when i is 0, 1, 2, and 3.

Each time through the loop, the variable i is used as an index into the list, printing the i-eth element.

This pattern of computation is cal ed a list traversal.

9.3. List length

The function len returns the length of a list, which is equal to the number of its elements. It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing al the loops; they wil work correctly for any size list:

horsemen = ["war", "famine", "pestilence", "death"]

i = 0

num = len(horsemen)

while i < num:

print horsemen[i]

i += 1

The last time the body of the loop is executed, i is len(horsemen) - 1, which is the index of the last element. When i is equal to len(horsemen), the condition fails and the body is not executed, which is a good thing, because len(horsemen) is not a legal index.

Although a list can contain another list, the nested list stil counts as a single element. The length of this list is 4:

['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]

9.4. List membership

in is a boolean operator that tests membership in a sequence. We used it previously with strings, but it also works with lists and other sequences:

>>> horsemen = ['war', 'famine', 'pestilence', 'death']

>>> 'pestilence' **in** horsemen

True

>>> 'debauchery' **in** horsemen

False

Since pestilence is a member of the horsemen list, the in operator returns True. Since debauchery is not in the list, in returns False.

We can use the not in combination with in to test whether an element is not a member of a list:

>>> 'debauchery' **not in** horsemen

True

9.5. List operations

The + operator concatenates lists:

>>> a = [1, 2, 3]

>>> b = [4, 5, 6]

>>> c = a + b

>>> print c

[1, 2, 3, 4, 5, 6]

Similarly, the * operator repeats a list a given number of times:

>>> [0] * 4

[0, 0, 0, 0]

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

[1, 2, 3, 1, 2, 3, 1, 2, 3]

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

9.6. List slices

The slice operations we saw with strings also work on lists:

>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']

>>> a_list[1:3]

['b', 'c']

>>> a_list[:4]

['a', 'b', 'c', 'd']

>>> a_list[3:]

['d', 'e', 'f']

>>> a_list[:]

['a', 'b', 'c', 'd', 'e', 'f']

9.7. The **range** function

Lists that contain consecutive integers are common, so Python provides a simple way to create them:

>>> range(1, 5)

[1, 2, 3, 4]

The range function takes two arguments and returns a list that contains al the integers from the first to the second, including the first but not the second.

There are two other forms of range. With a single argument, it creates a list that starts at 0:

>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

If there is a third argument, it specifies the space between successive values, which is cal ed the step size. This example counts from 1 to 10 by steps of 2:

>>> range(1, 10, 2)

[1, 3, 5, 7, 9]

If the step size is negative, then start must be greater than stop

>>> range(20, 4, -5)

[20, 15, 10, 5]

or the result wil be an empty list.

>>> range(10, 20, -5)

[]

9.8. Lists are mutable

Unlike strings, lists are mutable, which means we can change their elements. Using the bracket operator on the left side of an assignment, we can update one of the elements:

>>> fruit = ["banana", "apple", "quince"]

>>> fruit[0] = "pear"

>>> fruit[-1] = "orange"

>>> print fruit

['pear', 'apple', 'orange']

The bracket operator applied to a list can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of fruit has been changed from 'banana' to 'pear', and the last from 'quince' to 'orange'. An assignment to an element of a list is cal ed item assignment. Item assignment does not work for strings:

>>> my_string = 'TEST'

>>> my_string[2] = 'X'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

but it does for lists:

>>> my_list = ['T', 'E', 'S', 'T']

>>> my_list[2] = 'X'

>>> my_list

['T', 'E', 'X', 'T']

With the slice operator we can update several elements at once:
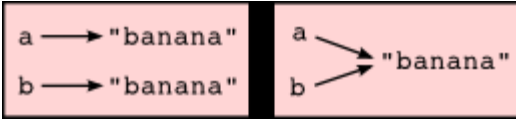
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']

>>> a_list[1:3] = ['x', 'y']

>>> print a_list

['a', 'x', 'y', 'd', 'e', 'f']

We can also remove elements from a list by assigning the empty list to them:

>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']

>>> a_list[1:3] = []

>>> print a_list

['a', 'd', 'e', 'f']

And we can add elements to a list by squeezing them into an empty slice at the desired location:

>>> a_list = ['a', 'd', 'f']

>>> a_list[1:1] = ['b', 'c']

>>> print a_list

['a', 'b', 'c', 'd', 'f']

>>> a_list[4:4] = ['e']

>>> print a_list

['a', 'b', 'c', 'd', 'e', 'f']

9.9. List deletion

Using slices to delete list elements can be awkward, and therefore error-prone. Python provides an alternative that is more readable.

del removes an element from a list:

>>> a = ['one', 'two', 'three']

>>> del a[1]

>>> a

['one', 'three']

As you might expect, del handles negative indices and causes a runtime error if the index is out of range.

You can use a slice as an index for del:

>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']

>>> del a_list[1:5]

>>> print a_list

['a', 'f']

As usual, slices select al the elements up to, but not including, the second index.

9.10. Objects and values

If we execute these assignment statements,

a = "banana"

b = "banana"

we know that a and b wil refer to a string with the letters "banana". But we don't know yet whether they point to the same string.

There are two possible states:
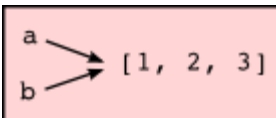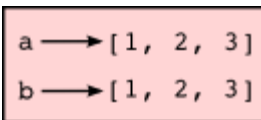
In one case, a and b refer to two different things that have the same value. In the second case, they refer to the same thing. These things have names — they are cal ed objects. An object is something a variable can refer to.

We can test whether two names have the same value using ==:

>>> a == b

True

We can test whether two names refer to the same object using the is operator:

>>> a **is** b

True

This tel s us that both a and b refer to the same object, and that it is the second of the two state diagrams that describes the relationship.

Since strings are immutable, Python optimizes resources by making two names that refer to the same string value refer to the same object.

This is not the case with lists:

>>> a = [1, 2, 3]

>>> b = [1, 2, 3]

>>> a == b

True

>>> a **is** b

False

The state diagram here looks like this:

a and b have the same value but do not refer to the same object.

9.11. Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

>>> a = [1, 2, 3]

>>> b = a

>>> a **is** b

True

In this case, the state diagram looks like this:

Because the same list has two different names, a and b, we say that it is aliased. Changes made with one alias affect the other:

>>> b[0] = 5

>>> print a

[5, 2, 3]

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there's no problem. That's why Python is free to alias strings when it sees an opportunity to economize.

9.12. Cloning lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes cal ed cloning, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator:

>>> a = [1, 2, 3]

>>> b = a[:]

>>> print b

[1, 2, 3]

Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

Now we are free to make changes to b without worrying about a:

>>> b[0] = 5

>>> print a

[1, 2, 3]

9.13. Lists and **for** loops

The for loop also works with lists. The generalized syntax of a for loop is:

for VARIABLE **in** LIST:

BODY

This statement is equivalent to:

i = 0

while i < len(LIST):

VARIABLE = LIST[i]

BODY

i += 1

The for loop is more concise because we can eliminate the loop variable, i. Here is an equivalent to the while loop from the Accessing elements section written with a for loop.

for horseman **in** horsemen:

print horseman

It almost reads like English: For (every) horseman in (the list of) horsemen, print (the name of the) horseman.

Any list expression can be used in a for loop:

```
for number in range(20):

if number % 3 == 0:

print number
```

```
for fruit in ["banana", "apple", "quince"]: print "I like to eat " + fruit +
"s!"
```


The first example prints al the multiples of 3 between 0 and 19. The second example expresses enthusiasm for various fruits.

Since lists are mutable, it is often desirable to traverse a list, modifying each of its elements. The fol owing squares al the numbers from 1 to 5:

```
numbers = [1, 2, 3, 4, 5]
```

```
for index in range(len(numbers)):
```

```
numbers[index] = numbers[index]**2
```

Take a moment to think about range(len(numbers)) until you understand how it works. We are interested here in both the value and its index within the list, so that we can assign a new value to it.

This pattern is common enough that Python provides a nicer way to implement it:

```
numbers = [1, 2, 3, 4, 5]
```

```
for index, value in enumerate(numbers):
```

```
numbers[index] = value**2
```

enumerate generates both the index and the value associated with it during the list traversal. Try this next example to see more clearly how enumerate works:

```
>>> for index, value in enumerate(['banana', 'apple', 'pear', 'quince']):
... print index, value
...
0 banana
1 apple
2 pear
3 quince
>>>
```

9.14. List parameters

Passing a list as an argument actual y passes a reference to the list, not a copy of the list. Since lists are mutable changes made to the parameter change the argument as wel . For example, the function below takes a list as an argument and multiplies each element in the list by 2:

```
def double_stuff(a_list):
    for index, value in enumerate(a_list):
        a_list[index] = 2 * value
```

If we put double_stuff in a file named ch09.py, we can test it out like this:

>>> from ch09 import double_stuff

>>> things = [2, 5, 'Spam', 9.5]

>>> double_stuff(things)

>>> things

[4, 10, 'SpamSpam', 19.0]

>>>

The parameter a_list and the variable things are aliases for the same object. The state diagram looks like this:

Since the list object is shared by two frames, we drew it between them.

If a function modifies a list parameter, the cal er sees the change.

9.15. Pure functions and modifiers

Functions which take lists as arguments and change them during execution are cal ed modifiers and the changes they make are cal ed side effects.

A pure function does not produce side effects. It communicates with the cal ing program only through parameters, which it does not modify, and a return value. Here is double_stuff written as a pure function:

def double_stuff(a_list):

new_list = []

for value **in** a_list:

new_list += [2 * value]

return new_list

This version of double_stuff does not change its arguments:

>>> from ch09 import double_stuff

>>> things = [2, 5, 'Spam', 9.5]

>>> double_stuff(things)

[4, 10, 'SpamSpam', 19.0]

>>> things

[2, 5, 'Spam', 9.5]

>>>

To use the pure function version of double_stuff to modify things, you would assign the return value back to things:

>>> things = double_stuff(things)

>>> things

[4, 10, 'SpamSpam', 19.0]

>>>

9.16. Which is better?

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only al ow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient.

In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compel ing advantage. This approach might be cal ed a functional programming style.

9.17. Nested lists

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list:

>>> nested = ["hello", 2.0, 5, [10, 20]]

If we print nested[3], we get [10, 20]. To extract an element from the nested list, we can proceed in two steps:

>>> elem = nested[3]

>>> elem[0]

10

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Or we can combine them:

>>> nested[3][1]

20

Bracket operators evaluate from left to right, so this expression gets the three-eth element of nested and extracts the one-eth element from it.

9.18. Matrices

Nested lists are often used to represent matrices. For example, the matrix:

might be represented as:

>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

matrix is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

>>> matrix[1]

[4, 5, 6]

Or we can extract a single element from the matrix using the double-index form:

>>> matrix[1][1]

5

The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A smal variation is to use a list of columns instead of a list of rows. Later we wil see a more radical alternative using a dictionary.

9.19. Test-driven development (TDD)

Test-driven development (TDD) is a software development practice which arrives at a desired feature through a series of smal , iterative steps motivated by automated tests which are written first that express increasing refinements of the desired feature.

Doctest enables us to easily demonstrate TDD. Let's say we want a function which creates a rows by columns matrix given arguments for rows and columns.

We first setup a test for this function in a file named matrices.py:

```
def make_matrix(rows, columns):
    """

    >>> make_matrix(3, 5)
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Running this returns in a failing test:

```
**********************************************************************
File "matrices.py", line 3, in __main__.make_matrix
Failed example:
make_matrix(3, 5)
Expected:
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
Got nothing
    **********************************************************************
1 items had failures:
1 of 1 in __main__.make_matrix
***Test Failed*** 1 failures.
```

The test fails because the body of the function contains only a single triple quoted string and no return statement, so it returns None. Our test indicates that we wanted it to return a matrix with 3 rows of 5

columns of zeros.

The rule in using TDD is to use the simplest thing that works in writing a solution to pass the test, so in this case we can simply return the desired result:

def make_matrix(rows, columns):

"""

>>> make_matrix(3, 5)

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

"""

return [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

Running this now the test passes, but our current implementation of make_matrix always returns the same result, which is clearly not what we intended. To fix this, we first motivate our improvement by adding a test:

def make_matrix(rows, columns):

"""

>>> make_matrix(3, 5)

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

>>> make_matrix(4, 2)

[[0, 0], [0, 0], [0, 0], [0, 0]]

"""

return [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

which as we expect fails:

    ****

File "matrices.py", line 5, in __main__.make_matrix

Failed example:

make_matrix(4, 2)

Expected:

[[0, 0], [0, 0], [0, 0], [0, 0]]

Got:

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

    ****

1 items had failures:

1 of 2 in __main__.make_matrix

Test Failed 1 failures.

This technique is cal ed test-driven because code should only be written when there is a failing test to make pass. Motivated by the failing test, we can now produce a more general solution: **def** make_matrix(rows, columns):

"""

>>> *make_matrix(3, 5)*

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

>>> make_matrix(4, 2)

[[0, 0], [0, 0], [0, 0], [0, 0]]

"""

return [[0] * columns] * rows

This solution appears to work, and we may think we are finished, but when we use the new function later we discover a bug:

>>> from matrices import *

>>> m = make_matrix(4, 3)

>>> m

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

>>> m[1][2] = 7

>>> m

[[0, 0, 7], [0, 0, 7], [0, 0, 7], [0, 0, 7]]

>>>

We wanted to assign the element in the second row and the third column the value 7, instead, all elements in the third column are 7!

Upon reflection, we realize that in our current solution, each row is an alias of the other rows. This is definitely not what we intended, so we set about fixing the problem, first by writing a failing test: **def** make_matrix(rows, columns):

"""

```
>>> make_matrix(3, 5)
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
>>> make_matrix(4, 2)
[[0, 0], [0, 0], [0, 0], [0, 0]]
>>> m = make_matrix(4, 2)
>>> m[1][1] = 7
>>> m
[[0, 0], [0, 7], [0, 0], [0, 0]]
"""
return [[0] * columns] * rows
```

With a failing test to fix, we are now driven to a better solution:

```
def make_matrix(rows, columns):
"""
>>> make_matrix(3, 5)
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
>>> make_matrix(4, 2)
[[0, 0], [0, 0], [0, 0], [0, 0]]
>>> m = make_matrix(4, 2)
>>> m[1][1] = 7
>>> m
```

[[0, 0], [0, 7], [0, 0], [0, 0]]

"""

matrix = []

for row **in** range(rows):

matrix += [[0] * columns]

return matrix

Using TDD has several benefits to our software development process. It:

helps us think concretely about the problem we are trying to solve before we attempt to solve it.

encourages breaking down complex problems into smal er, simpler problems and working our way toward a solution of the larger problem step-by-step.

assures that we have a wel developed automated test suite for our software, facilitating later additions and improvements.

9.20. Strings and lists

Python has a command cal ed list that takes a sequence type as an argument and creates a list out of its elements.

>>> list("Crunchy Frog")

['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']

There is also a str command that takes any Python value as an argument and returns a string representation of it.

>>> str(5)

'5'

>>> str(None)

'None'

>>> str(list("nope"))

"['n', 'o', 'p', 'e']"

As we can see from the last example, str can't be used to join a list of characters together. To do this we could use the join function in the string module:

>>> import string

>>> char_list = list("Frog")

>>> char_list

['F', 'r', 'o', 'g']

>>> string.join(char_list, '')

'Frog'

Two of the most useful functions in the string module involve lists of strings. The split function breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary:

>>> import string

>>> song = "The rain in Spain..."

>>> string.split(song)

['The', 'rain', 'in', 'Spain...']

An optional argument cal ed a delimiter can be used to specify which characters to use as word boundaries. The fol owing example uses the string ai as the delimiter:

>>> string.split(song, 'ai')

['The r', 'n in Sp', 'n...']

Notice that the delimiter doesn't appear in the list.

string.join is the inverse of string.split. It takes two arguments: a list of strings and a separator which wil be placed between each element in the list in the resultant string.

>>> import string

>>> words = ['crunchy', 'raw', 'unboned', 'real', 'dead', 'frog']

>>> string.join(words, ' ')

'crunchy raw unboned real dead frog'

>>> string.join(words, '**')

'crunchy**raw**unboned**real**dead**frog'

9.21. Glossary

aliases

Multiple variables that contain references to the same object.

clone

To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn't clone the object.

delimiter

A character or string used to indicate where a string should be split.

element

One of the values in a list (or other sequence). The bracket operator selects elements of a list.

index

An integer variable or value that indicates an element of a list.

list

A named col ection of objects, where each object is identified by an index.

list traversal

The sequential accessing of each element in a list.

modifier

A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

mutable type

A data type in which the elements can be modified. Al mutable types are compound types. Lists are mutable data types; strings are not.

nested list

A list that is an element of another list.

object

A thing to which a variable can refer.

pure function

A function which has no side effects. Pure functions only make changes to the cal ing program through their return values.

sequence

Any of the data types that consist of an ordered set of elements, with each element identified by an index.

side effect

A change in the state of a program made by cal ing a function that is not a result of reading the return value from the function. Side effects can only be produced by modifiers.

step size

The interval between successive elements of a linear sequence. The third (and optional argument) to the range function is cal ed the step size. If not specified, it defaults to 1.

test-driven development (TDD)

A software development practice which arrives at a desired feature through a series of smal , iterative steps motivated by automated tests which are written first that express increasing refinements of the desired feature. (see the Wikipedia article on Test-driven development for more information.)

9.22. Exercises

1. Write a loop that traverses:

['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]

and prints the length of each element. What happens if you send an integer to len? Change 1 to

'one' and run your solution again.

2. Open a file named ch09e02.py and with the fol owing content:

Add your doctests here:

"""

"""

Write your Python code here:

if __name__ == '__main__':

import doctest

doctest.testmod()

Add each of the fol owing sets of doctests to the docstring at the top of the file and write Python code to make the doctests pass.

"""

>>> a_list[3]

42

>>> a_list[6]

'Ni!'

>>> len(a_list)

8

"""

```
    """

    >>> b_list[1:]

    ['Stills', 'Nash']

    >>> group = b_list + c_list

    >>> group[-1]

    'Young'

    """

    """

    >>> 'war' in mystery_list

    False

    >>> 'peace' in mystery_list

    True

    >>> 'justice' in mystery_list

    True

    >>> 'oppression' in mystery_list

    False

    >>> 'equality' in mystery_list

    True

    """

    """
```

>>> range(a, b, c)

[5, 9, 13, 17]

"""

Only add one set of doctests at a time. The next set of doctests should not be added until the previous set pass.

3. What is the Python interpreter's response to the fol owing?

>>> range(10, 0, -2)

The three arguments to the range function are start, stop, and step, respectively. In this example, start is greater than stop. What happens if start < stop and step < 0? Write a rule for the relationships among start, stop, and step.

4. Draw a state diagram for a and b before and after the third line of the fol owing python code is executed:

a = [1, 2, 3]

b = a[:]

b[0] = 5

5. What wil be the output of the fol owing program?

this = ['I', 'am', 'not', 'a', 'crook']

that = ['I', 'am', 'not', 'a', 'crook']

print "Test 1: *%s*" % (this **is** that)

that = this

print "Test 2: *%s*" % (this **is** that)

Provide a detailed explaination of the results.

6. Open a file named ch09e06.py and use the same procedure as in exercise 2 to make the fol owing doctests pass:

"""

>>> 13 in junk

True

>>> del junk[4]

>>> junk

[3, 7, 9, 10, 17, 21, 24, 27]

>>> del junk[a:b]

>>> junk

[3, 7, 27]

"""

"""

>>> nlist[2][1]

0

>>> nlist[0][2]

17

>>> nlist[1][1]

5

```
"""

"""

>>> import string

>>> string.split(message, '??')

['this', 'and', 'that']

"""
```

7. Lists can be used to represent mathematical vectors. In this exercise and several that fol ow you wil write functions to perform standard operations on vectors. Create a file named vectors.py and write Python code to make the doctests for each function pass.

Write a function add_vectors(u, v) that takes two lists of numbers of the same length, and returns a new list containing the sums of the corresponding elements of each.

def add_vectors(u, v):

```
"""

>>> add_vectors([1, 0], [1, 1])

[2, 1]

>>> add_vectors([1, 2], [1, 4])

[2, 6]

>>> add_vectors([1, 2, 1], [1, 4, 3])

[2, 6, 4]

>>> add_vectors([11, 0, -4, 5], [2, -4, 17, 0])
```

[13, -4, 13, 5]

"""

add_vectors should pass the doctests above.

8. Write a function scalar_mult(s, v) that takes a number, s, and a list, v and returns the scalar

multiple of v by s.

def scalar_mult(s, v):

"""

>>> scalar_mult(5, [1, 2])

[5, 10]

>>> scalar_mult(3, [1, 0, -1])

[3, 0, -3]

>>> scalar_mult(7, [3, 0, 5, 11, 2])

[21, 0, 35, 77, 14]

"""

9. Write a function dot_product(u, v) that takes two lists of numbers of the same length, and returns the sum of the products of the corresponding elements of each (the dot_product).

def dot_product(u, v):

"""

>>> dot_product([1, 1], [1, 1])

2

>>> dot_product([1, 2], [1, 4])

9

>>> dot_product([1, 2, 1], [1, 4, 3])

12

>>> dot_product([2, 0, -1, 1], [1, 5, 2, 0])

0

"""

Verify that dot_product passes the doctests above.

10. Extra challenge for the mathematically inclined: Write a function cross_product(u, v) that takes two lists of numbers of length 3 and returns their [cross product](). You should write your own doctests and use the test driven development process described in the chapter.

11. Create a new module named matrices.py and add the fol owing two functions introduced in the section on test-driven development:

def add_row(matrix):

"""

>>> m = [[0, 0], [0, 0]]

>>> add_row(m)

[[0, 0], [0, 0], [0, 0]]

>>> n = [[3, 2, 5], [1, 4, 7]]

```
    >>> add_row(n)
    [[3, 2, 5], [1, 4, 7], [0, 0, 0]]
    >>> n
    [[3, 2, 5], [1, 4, 7]]
    """

def add_column(matrix):
    """
    >>> m = [[0, 0], [0, 0]]
    >>> add_column(m)
    [[0, 0, 0], [0, 0, 0]]
    >>> n = [[3, 2], [5, 1], [4, 7]]
    >>> add_column(n)
    [[3, 2, 0], [5, 1, 0], [4, 7, 0]]
    >>> n
    [[3, 2], [5, 1], [4, 7]]
    """
```

Your new functions should pass the doctests. Note that the last doctest in each function assures that add_row and add_column are pure functions. (hint: Python has a copy module with a function named deepcopy that could make your task easier here. We wil talk more about deepcopy in chapter 13, but google python copy module if you would like to try it now.)

12. Write a function add_matrices(m1, m2) that adds m1 and m2 and returns a new matrix containing their sum. You can assume that m1 and m2 are the same size. You add two matrices by adding their corresponding values.

def add_matrices(m1, m2):

"""

>>> a = [[1, 2], [3, 4]]

>>> b = [[2, 2], [2, 2]]

>>> add_matrices(a, b)

[[3, 4], [5, 6]]

>>> c = [[8, 2], [3, 4], [5, 7]]

>>> d = [[3, 2], [9, 2], [10, 12]]

>>> add_matrices(c, d)

[[11, 4], [12, 6], [15, 19]]

>>> c

[[8, 2], [3, 4], [5, 7]]

>>> d

[[3, 2], [9, 2], [10, 12]]

"""

Add your new function to matrices.py and be sure it passes the doctests above. The last two doctests confirm that add_matrices is a pure function.

13. Write a function scalar_mult(s, m) that multiplies a matrix, m, by a scalar, s.

def scalar_mult(s, m):

"""

>>> a = [[1, 2], [3, 4]]

>>> scalar_mult(3, a)

[[3, 6], [9, 12]]

>>> b = [[3, 5, 7], [1, 1, 1], [0, 2, 0], [2, 2, 3]]

>>> scalar_mult(10, b)

[[30, 50, 70], [10, 10, 10], [0, 20, 0], [20, 20, 30]]

>>> b

[[3, 5, 7], [1, 1, 1], [0, 2, 0], [2, 2, 3]]

"""

Add your new function to matrices.py and be sure it passes the doctests above.

14. Write functions row_times_column and matrix_mult:

def row_times_column(m1, row, m2, column):

"""

>>> row_times_column([[1, 2], [3, 4]], 0, [[5, 6], [7, 8]], 0)

19

>>> row_times_column([[1, 2], [3, 4]], 0, [[5, 6], [7, 8]], 1)

22

>>> row_times_column([[1, 2], [3, 4]], 1, [[5, 6], [7, 8]], 0)

43

>>> row_times_column([[1, 2], [3, 4]], 1, [[5, 6], [7, 8]], 1)

50

"""

def matrix_mult(m1, m2):

"""

>>> matrix_mult([[1, 2], [3, 4]], [[5, 6], [7, 8]])

[[19, 22], [43, 50]]

>>> matrix_mult([[1, 2, 3], [4, 5, 6]], [[7, 8], [9, 1], [2, 3]])

[[31, 19], [85, 55]]

>>> matrix_mult([[7, 8], [9, 1], [2, 3]], [[1, 2, 3], [4, 5, 6]])

[[39, 54, 69], [13, 23, 33], [14, 19, 24]]

"""

Add your new functions to matrices.py and be sure it passes the doctests above.

15. Create a new module named numberlists.py and add the fol owing functions to the module: **def** only_evens(numbers):

"""

>>> only_evens([1, 3, 4, 6, 7, 8])

[4, 6, 8]

>>> only_evens([2, 4, 6, 8, 10, 11, 0])

[2, 4, 6, 8, 10, 0]

>>> only_evens([1, 3, 5, 7, 9, 11])

[]

>>> only_evens([4, 0, -1, 2, 6, 7, -4])

[4, 0, 2, 6, -4]

>>> nums = [1, 2, 3, 4]

>>> only_evens(nums)

[2, 4]

>>> nums

[1, 2, 3, 4]

"""

def only_odds(numbers):

"""

>>> only_odds([1, 3, 4, 6, 7, 8])

[1, 3, 7]

>>> only_odds([2, 4, 6, 8, 10, 11, 0])

[11]

>>> only_odds([1, 3, 5, 7, 9, 11])

[1, 3, 5, 7, 9, 11]

>>> only_odds([4, 0, -1, 2, 6, 7, -4])

[-1, 7]

>>> nums = [1, 2, 3, 4]

>>> only_odds(nums)

[1, 3]

>>> nums

[1, 2, 3, 4]

"""

Be sure these new functions pass the doctests.

16. Add a function multiples_of(num, numlist) to numberlists.py that takes an integer (num), and a list of integers (numlist) as arguments and returns a list of those integers in numlist that are multiples of num. Add your own doctests and use TDD to develope this function.

17. Given:

import string

song = "The rain in Spain..."

Describe the relationship between string.join(string.split(song)) and song. Are they the same for al strings? When would they be different?

18. Write a function replace(s, old, new) that replaces al occurences of old with new in a string s.

```
def replace(s, old, new):
```

"""

>>> replace('Mississippi', 'i', 'I')

'MIssIssIppI'

>>> s = 'I love spom! Spom is my favorite food. Spom, spom, spom, yum!'

>>> replace(s, 'om', 'am')

'I love spam! Spam is my favorite food. Spam, spam, spam, yum!'

>>> replace(s, 'o', 'a')

'I lave spam! Spam is my favarite faad. Spam, spam, spam, yum!'

"""

Your solution should pass the doctests above. Hint: use string.split and string.join.

```
┌─────────────────────────────────────┐
│ □      pydoc      [ _ ][ □ ][ × ]    │
│   Python documentation server at     │
│        http://localhost:7464/        │
│ ┌──────────────┬──────────────────┐  │
│ │  open browser │   quit serving   │  │
│ └──────────────┴──────────────────┘  │
│ Search for │                         │
└─────────────────────────────────────┘
```
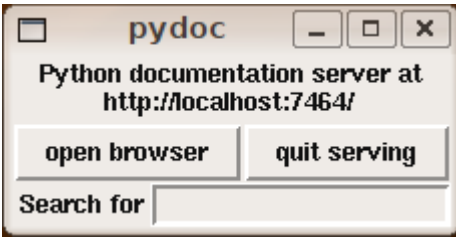
10. Modules and files

10.1. Modules

A module is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the standard library. We have seen two of these already, the doctest module and the string module.

10.2. pydoc

You can use pydoc to search through the Python libraries instal ed on your system. At the command prompt type the fol owing:

$ pydoc -g

and the fol owing wil appear:

(note: see exercise 2 if you get an error)

Click on the open browser button to launch a web browser window containing the documentation generated by pydoc:
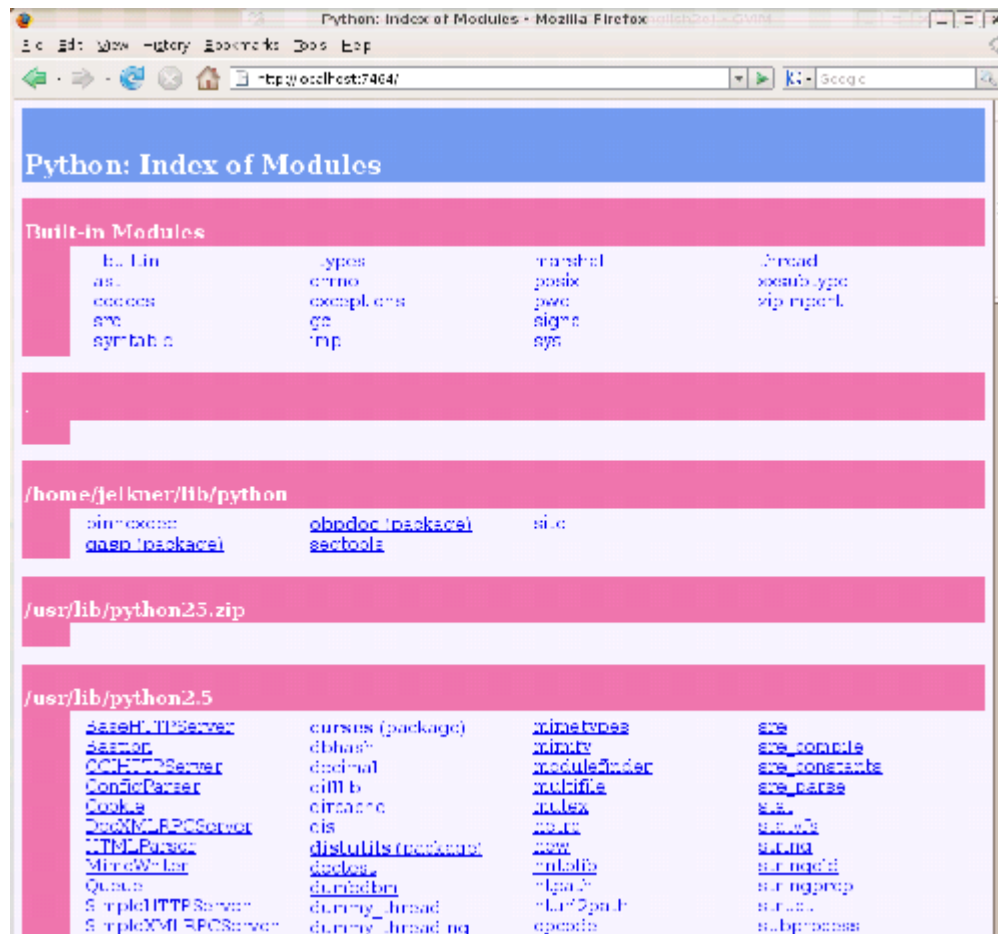
This is a listing of al the python libraries found by Python on your system. Clicking on a module name opens a new page with documenation for that module. Clicking keyword, for example, opens the fol owing page:
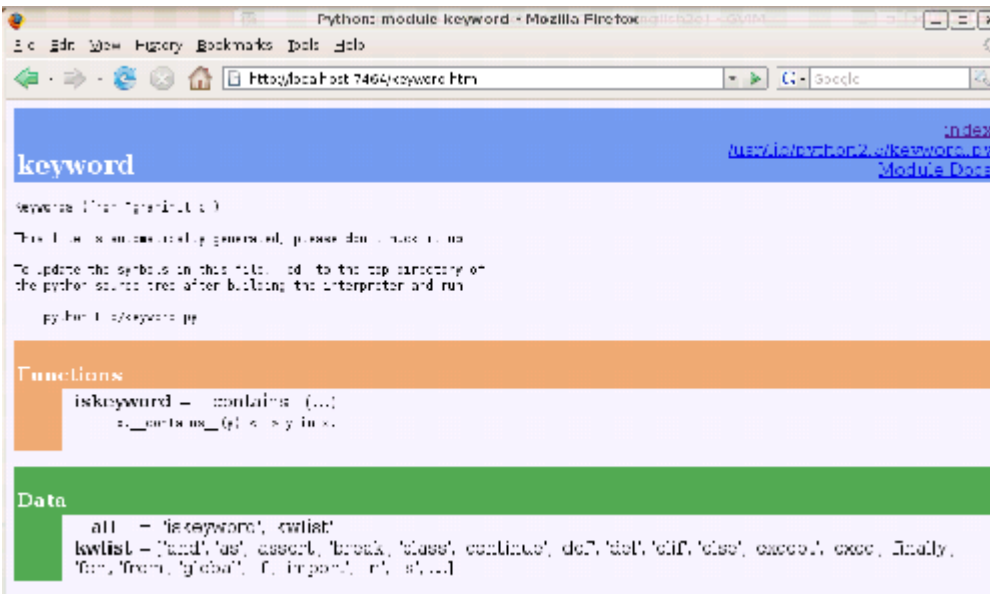
Documentation for most modules contains three color coded sections:

Classes in pink

Functions in orange

Data in green

Classes wil be discussed in later chapters, but for now we can use pydoc to see the functions and data contained within modules.

The keyword module contains a single function, iskeyword, which as its name suggests is a boolean function that returns True if a string passed to it is a keyword:

>>> from keyword import *

>>> iskeyword('for')

True

>>> iskeyword('all')

False

```
>>>
```

The data item, kwlist contains a list of al the current keywords in Python:

>>> from keyword import *

>>> print kwlist

['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif',

'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import',

'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try',

'while', 'with', 'yield']

>>>

We encourage you to use pydoc to explore the extensive libraries that come with Python. There are so many treasures to discover!

10.3. Creating modules

Al we need to create a module is a text file with a .py extension on the filename:

 # seqtools.py

#def remove_at(pos, seq):

return seq[:pos] + seq[pos+1:]

We can now use our module in both scripts and the Python shel . To do so, we must first import the module. There are two ways to do this:

>>> from seqtools import remove_at

>>> s = "A string!"

>>> remove_at(4, s)

'A sting!'

and:

>>> import seqtools

>>> s = "A string!"

>>> seqtools.remove_at(4, s)

'A sting!'

In the first example, remove_at is cal ed just like the functions we have seen previously. In the second example the name of the module and a dot (.) are written before the function name.

Notice that in either case we do not include the .py file extension when importing. Python expects the file names of Python modules to end in .py, so the file extention is not included in the import statement.

The use of modules makes it possible to break up very large programs into manageable sized parts, and to keep related parts together.

10.4. Namespaces

A namespace is a syntactic container which permits the same name to be used in different modules or functions (and as we wil see soon, in classes and methods).

Each module determines its own namespace, so we can use the same name in multiple modules without causing an identification problem.

 # module1.py

question = "What is the meaning of life, the Universe, and everything?"

answer = 42

module2.py

question = "What is your quest?"

answer = "To seek the holy grail."

We can now import both modules and access question and answer in each:

>>> import module1

>>> import module2

>>> print module1.question

What is the meaning of life, the Universe, and everything?

>>> print module2.question

What is your quest?

>>> print module1.answer

42

>>> print module2.answer

To seek the holy grail.

>>>

If we had used from module1 import * and from module2 import * instead, we would have a naming collision and would not be able to

access question and answer from module1.

Functions also have their own namespace:

def f():

n = 7

print "printing n inside of f: *%d*" % n

def g():

n = 42

print "printing n inside of g: *%d*" % n

n = 11

print "printing n before calling f: *%d*" % n

f()

print "printing n after calling f: *%d*" % n

g()

print "printing n after calling g: *%d*" % n

Running this program produces the fol owing output:

printing n before calling f: 11

printing n inside of f: 7

printing n after calling f: 11

printing n inside of g: 42

printing n after calling g: 11

The three n's here do not col ide since they are each in a different namespace.

Namespaces permit several programmers to work on the same project without having naming col isions.

10.5. Attributes and the dot operator

Variables defined inside a module are cal ed attributes of the module. They are accessed by using the dot operator (.). The question attribute of module1 and module2 are accessed using module1.question and module2.question.

Modules contain functions as wel as attributes, and the dot operator is used to access them in the same way. seqtools.remove_at refers to the remove_at function in the seqtools module.

In Chapter 7 we introduced the find function from the string module. The string module contains many other useful functions:

>>> import string

>>> string.capitalize('maryland')

'Maryland'

>>> string.capwords("what's all this, then, amen?")

"What's All This, Then, Amen?"

>>> string.center('How to Center Text Using Python', 70)

' How to Center Text Using Python '

>>> string.upper('angola')

'ANGOLA'

>>>

You should use pydoc to browse the other functions and attributes in the string module.

10.6. String and list methods

As the Python language developed, most of functions from the string module have also been added as methods of string objects. A method acts much like a function, but the syntax for cal ing it is a bit different:

>>> 'maryland'.capitalize()

'Maryland'

>>> "what's all this, then, amen?".title()

"What'S All This, Then, Amen?"

>>> 'How to Center Text Using Python'.center(70)

' How to Center Text Using Python '

>>> 'angola'.upper()

'ANGOLA'

>>>

String methods are built into string objects, and they are invoked (cal ed) by fol owing the object with the dot operator and the method name.

We wil be learning how to create our own objects with their own methods in later chapters. For now we wil only be using methods that come with Python's built-in objects.

The dot operator can also be used to access built-in methods of list objects:

>>> mylist = []

>>> mylist.append(5)

>>> mylist.append(27)

>>> mylist.append(3)

>>> mylist.append(12)

>>> mylist

[5, 27, 3, 12]

>>>

append is a list method which adds the argument passed to it to the end of the list. Continuing with this example, we show several other list methods:

>>> mylist.insert(1, 12)

>>> mylist

[5, 12, 27, 3, 12]

>>> mylist.count(12)

2**>>>** mylist.extend([5, 9, 5, 11])

>>> mylist

[5, 12, 27, 3, 12, 5, 9, 5, 11])

>>> mylist.index(9)

6**>>>** mylist.count(5)

3**>>>** mylist.reverse()

>>> mylist

[11, 5, 9, 5, 12, 3, 27, 12, 5]

>>> mylist.sort()

>>> mylist

[3, 5, 5, 5, 9, 11, 12, 12, 27]

>>> mylist.remove(12)

>>> mylist

[3, 5, 5, 5, 9, 11, 12, 27]

>>>

Experiment with the list methods in this example until you feel confident that you understand how they work.

10.7. Reading and writing text files

While a program is running, its data is stored in random access memory (RAM). RAM is fast and inexpensive, but it is also volatile, which means that when the program ends, or the computer shuts down, data in RAM disappears. To make data available the next time you turn on your computer and start your program, you have to write it to a non-volatile storage medium, such as a hard drive, usb drive, or CD-RW.

Data on non-volatile storage media is stored in named locations on the media cal ed files. By reading and writing files, programs can save information between program runs.

Working with files is a lot like working with a notebook. To use a notebook, you have to open it. When you're done, you have to close it. While the notebook is open, you can either write in it or read from it.

In either case, you know where you are in the notebook. You can read the whole notebook in its natural order or you can skip around.

Al of this applies to files as wel . To open a file, you specify its name and indicate whether you want to read or write.

Opening a file creates a file object. In this example, the variable myfile refers to the new file object.

>>> myfile = open('test.dat', 'w')

>>> print myfile

<open file 'test.dat', mode 'w' at 0x2aaaaab80cd8>

The open function takes two arguments. The first is the name of the file, and the second is the mode.

Mode 'w' means that we are opening the file for writing.

If there is no file named test.dat, it wil be created. If there already is one, it wil be replaced by the file we are writing.

When we print the file object, we see the name of the file, the mode, and the location of the object.

To put data in the file we invoke the write method on the file object:

>>> myfile.write("Now is the time")

>>> myfile.write("to close the file")

Closing the file tel s the system that we are done writing and makes the file available for reading:

>>> myfile.close()

Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is 'r' for reading:

>>> myfile = open('test.dat', 'r')

If we try to open a file that doesn't exist, we get an error:

>>> myfile = open('test.cat', 'r')

IOError: [Errno 2] No such file or directory: 'test.cat'

Not surprisingly, the read method reads data from the file. With no arguments, it reads the entire contents of the file into a single string:

>>> text = myfile.read()

>>> print text

Now is the timeto close the file

There is no space between time and to because we did not write a space between the strings.

read can also take an argument that indicates how many characters to read:

>>> myfile = open('test.dat', 'r')

>>> print myfile.read(5)

Now i

If not enough characters are left in the file, read returns the remaining characters. When we get to the end of the file, read returns the empty string:

>>> print myfile.read(1000006)

s the timeto close the file

>>> print myfile.read()

>>>

The fol owing function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

def copy_file(oldfile, newfile):

infile = open(oldfile, 'r')

outfile = open(newfile, 'w')

while True:

text = infile.read(50)

if text == "":

break

outfile.write(text)

infile.close()

outfile.close()

return

This functions continues looping, reading 50 characters from infile and writing the same 50 charaters to outfile until the end of infile is reached, at which point text is empty and the break statement is executed.

10.8. Text files

A text file is a file that contains printable characters and whitespace, organized into lines separated by newline characters. Since Python is specifical y designed to process text files, it provides methods that make the job easy.

To demonstrate, we'l create a text file with three lines of text separated by newlines:

>>> outfile = open("test.dat","w")

>>> outfile.write("line one**\n**line two**\n**line three**\n**")

>>> outfile.close()

The readline method reads al the characters up to and including the next newline character:

>>> infile = open("test.dat","r")

>>> print infile.readline()

line one

>>>

readlines returns al of the remaining lines as a list of strings:

>>> print infile.readlines()

['line two\012', 'line three\012']

In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence \\012.

At the end of the file, readline returns the empty string and readlines returns the empty list:

>>> print infile.readline()

>>> print infile.readlines()

[]

The fol owing is an example of a line-processing program. filter makes a copy of oldfile, omitting any lines that begin with #:

def filter(oldfile, newfile):

infile = open(oldfile, 'r')

outfile = open(newfile, 'w')

while True:

text = infile.readline()

if text == "":

break

if text[0] == '#':

continue

outfile.write(text)

infile.close()

outfile.close()

return

The continue statement ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the condition, and proceeds accordingly.

Thus, if text is the empty string, the loop exits. If the first character of text is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy text into the new file.

10.9. Directories

Files on non-volatile storage media are organized by a set of rules known as a file system. File systems are made up of files and directories, which are containers for both files and other directories.

When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Python looks for it in the current directory.

If you want to open a file somewhere else, you have to specify the path to the file, which is the name of the directory (or folder) where the file is located:

>>> wordsfile = open('/usr/share/dict/words', 'r')

>>> wordlist = wordsfile.readlines()

>>> print wordlist[:6]

['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']

This example opens a file named words that resides in a directory named dict, which resides in share, which resides in usr, which resides in the top-level directory of the system, cal ed /. It then reads

in each line into a list using readlines, and prints out the first 5 elements from that list.

You cannot use / as part of a filename; it is reserved as a delimiter between directory and filenames.

The file /usr/share/dict/words should exist on unix based systems, and contains a list of words in alphabetical order.

10.10. Counting Letters

The ord function returns the integer representation of a character:

>>> ord('a')

97

>>> ord('A')

65

>>>

This example explains why 'Apple' < 'apple' evaluates to True.

The chr function is the inverse of ord. It takes an integer as an argument and returns its character representation:

>>> for i **in** range(65, 71):

... print chr(i)

...

ABCDEF>>>

The fol owing program, countletters.py counts the number of times each character occurs in the book

Alice in Wonderland:

```
## countletters.py
#
def display(i):
    if i == 10: return 'LF'
    if i == 13: return 'CR'
    if i == 32: return 'SPACE'
    return chr(i)

infile = open('alice_in_wonderland.txt', 'r')
text = infile.read()
infile.close()
counts = 128 * [0]
for letter in text:
    counts[ord(letter)] += 1
outfile = open('alice_counts.dat', 'w')
outfile.write(" %-12s%s\n" % ("Character", "Count"))
outfile.write("================\n")
for i in range(len(counts)):
    if counts[i]:
        outfile.write(" %-12s%d\n" % (display(i), counts[i]))
```

outfile.close()

Run this program and look at the output file it generates using a text editor. You wil be asked to analyze the program in the exercises below.

10.11. The **sys** module and **argv**

The sys module contains functions and variables which provide access to the environment in which the python interpreter runs.

The fol owing example shows the values of a few of these variables on one of our systems:

>>> import sys

>>> sys.platform

'linux2'

>>> sys.path

['', '/home/jelkner/lib/python', '/usr/lib/python25.zip', '/usr/lib/python2.5',

'/usr/lib/python2.5/plat-linux2', '/usr/lib/python2.5/lib-tk',

'/usr/lib/python2.5/lib-dynload', '/usr/local/lib/python2.5/site-packages',

'/usr/lib/python2.5/site-packages', '/usr/lib/python2.5/site-packages/Numeric',

'/usr/lib/python2.5/site-packages/gst-0.10',

'/var/lib/python-support/python2.5', '/usr/lib/python2.5/site-packages/gtk-2.0',

'/var/lib/python-support/python2.5/gtk-2.0']

>>> sys.version

'2.5.1 (r251:54863, Mar 7 2008, 04:10:12) \n[GCC 4.1.3 20070929 (prerelease)

(Ubuntu 4.1.2-16ubuntu2)]'

>>>

Starting Jython on the same machine produces different values for the same variables:

>>> import sys

>>> sys.platform

'java1.6.0_03'

>>> sys.path

['', '/home/jelkner/.', '/usr/share/jython/Lib', '/usr/share/jython/Lib-cpython']

>>> sys.version

'2.1'

>>>

The results wil be different on your machine of course.

The argv variable holds a list of strings read in from the command line when a Python script is run.

These command line arguments can be used to pass information into a program at the same time it is invoked.

demo_argv.py

#import sys

print sys.argv

Running this program from the unix command prompt demonstrates how sys.argv works:

$ python demo_argv.py this and that 1 2 3

['demo_argv.py', 'this', 'and', 'that', '1', '2', '3']

$

argv is a list of strings. Notice that the first element is the name of the program. Arguments are separated by white space, and separated into a list in the same way that string.split operates. If you want an argument with white space in it, use quotes:

$ python demo_argv.py "this and" that "1 2" 3

['demo_argv.py', 'this and', 'that', '1 2', '3']

$

With argv we can write useful programs that take their input directly from the command line. For example, here is a program that finds the sum of a series of numbers:

sum.py

#from sys import argv

nums = argv[1:]

for index, value **in** enumerate(nums):

nums[index] = float(value)

print sum(nums)

In this program we use the from <module> import <attribute> style of importing, so argv is brought into the module's main namespace.

We can now run the program from the command prompt like this:

$ python sum.py 3 4 5 11

23.0

$ python sum.py 3.5 5 11 100

119.5

You are asked to write similar programs as exercises.

10.12. Glossary

argv

argv is short for argument vector and is a variable in the sys module which stores a list of command line arguments passed to a program at run time.

attribute

A variable defined inside a module (or class or instance – as we wil see later). Module attributes are accessed by using the dot operator (.).

command line

The sequence of characters read into the command interpreter in a command line interface (see the Wikipedia article on [command line interface](#) for more information).

command line argument

A value passed to a program along with the program's invocation at the command prompt of a command line interface (CLI).

command prompt

A string displayed by a [command line interface](#) indicating that commands can be entered.

continue statement

A statement that causes the current iteration of a loop to end. The flow of execution goes to the top of the loop, evaluates the condition, and proceeds accordingly.

delimiter

A sequence of one or more characters used to specify the boundary between separate parts of text.

directory

A named col ection of files, also cal ed a folder. Directories can contain files and other directories, which are refered to as subdirectories of the directory that contains them.

dot operator

The dot operator (.) permits access to attributes and functions of a module (or attributes and methods of a class or instance – as we wil see later).

file

A named entity, usual y stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

file system

A method for naming, accessing, and organizing files and the data they contain.

import statement

A statement which makes the objects contained in a module available for use within another module. There are two forms for the import statement. Using a hypothetical module named mymod containing functions f1 and f2, and variables v1 and v2, examples of these two forms include:

import mymod

and

from mymod import f1, f2, v1, v2

The second form brings the imported objects into the namespace of the importing module, while the first form preserves a seperate namespace for the imported module, requiring mymod.v1 to access the v1 variable.

Jython

An implementation of the Python programming language written in Java. (see the Jython home page at http://www.jython.org for more information.) method

Function-like attribute of an object. Methods are invoked (cal ed) on an object using the dot operator. For example:

>>> s = "this is a string."

>>> s.upper()

'THIS IS A STRING.'

>>>

We say that the method, upper is invoked on the string, s. s is implicitly the first argument to upper.

mode

A distinct method of operation within a computer program. Files in Python can be opened in one of three modes: read ('r'), write ('w'), and append ('a').

module

A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the import statement.

namespace

A syntactic container providing a context for names so that the same name can reside in different namespaces without ambiguity. In Python, modules, classes, functions and methods al form namespaces.

naming collision

A situation in which two or more names in a given namespace cannot be unambiguously resolved.

Using

import string

instead of

from string import *

prevents naming col isions.

non-volatile memory

Memory that can maintain its state without power. Hard drives, flash drives, and rewritable compact disks (CD-RW) are each examples of non-volatile memory.

path

The name and location of a file within a file system. For example:

/usr/share/dict/words

indicates a file named words found in the dict subdirectory of the share subdirectory of the usr directory.

pydoc

A documentation generator that comes with the Python standard library.

standard library

A library is a col ection of software used as tools in the development of other software. The standard library of a programming language is the set of such tools that are distributed with the core programming language. Python comes with an extensive standard library.

text file

A file that contains printable characters organized into lines separated by newline characters.

volatile memory

Memory which requires an electrical current to maintain state. The main memory or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.

10.13. Exercises

1. Complete the fol owing:

Start the pydoc server with the command pydoc -g at the command prompt.

Click on the open browser button in the pydoc tk window.

Find the calendar module and click on it.

While looking at the Functions section, try out the fol owing in a Python shel :

>>> import calendar

>>> year = calendar.calendar(2008)

>>> print year # *What happens here?*

Experiment with calendar.isleap. What does it expect as an argument? What does it return as a result? What kind of a function is this?

Make detailed notes about what you learned from this exercise.

2. If you don't have Tkinter instal ed on your computer, then pydoc -g wil return an error, since the graphics window that it opens requires Tkinter. An alternative is to start the web server directly: $ pydoc -p 7464

This starts the pydoc web server on port 7464. Now point your web browser at:

http://localhost:7464

and you wil be able to browse the Python libraries instal ed on your system. Use this approach to start pydoc and take a look at the math

module.

a. How many functions are there in the math module?

b. What does math.ceil do? What about math.floor? (hint: both floor and ceil expect floating point arguments.)

c. Describe how we have been computing the same value as math.sqrt without using the math module.

d. What are the two data constants in the math module?

Record detailed notes of your investigation in this exercise.

3. Use pydoc to investigate the copy module. What does deepcopy do? In which exercises from last chapter would deepcopy have come in handy?

4. Create a module named mymodule1.py. Add attributes myage set to your current age, and year set to the current year. Create another module named mymodule2.py. Add attributes myage set to 0, and year set to the year you were born. Now create a file named namespace_test.py. Import both of the modules above and write the fol owing statement:

print (mymodule2.myage - mymodule1.myage) == (mymodule2.year - mymodule1.year) When you wil run namespace_test.py you wil see either True or False as output depending on whether or not you've already had your birthday this year.

5. Add the fol owing statement to mymodule1.py, mymodule2.py, and namespace_test.py from the previous exercise:

print "My name is *%s*" % __name__

Run namespace_test.py. What happens? Why? Now add the fol owing to the bottom of

mymodule1.py:

```
if __name__ == '__main__':

print "This won't run if I'm imported."
```

Run mymodule1.py and namespace_test.py again. In which case do you see the new print statement?

6. In a Python shel try the fol owing:

>>> import this

What does Tim Peters have to say about namespaces?

7. Use pydoc to find and test three other functions from the string module. Record your findings.

8. Rewrite matrix_mult from the last chapter using what you have learned about list methods.

9. The dir function, which we first saw in Chapter 7, prints out a list of the attributes of an object passed to it as an argument. In other words, dir returns the contents of the namespace of its argument. Use dir(str) and dir(list) to find at least three string and list methods which have not been introduced in the examples in the chapter. You should ignore anything that begins with double underscore (__) for the time being. Be sure to make detailed notes of your findings, including names of the new methods and examples of their use. (hint: Print the docstring of a function you want to explore. For example, to find out how str.join works, print

str.join.__doc__)

10. Give the Python interpreter's response to each of the fol owing from a continuous interpreter session:

>>> s = "If we took the bones out, it wouldn't be crunchy, would it?"

>>> s.split()

>>> type(s.split())

>>> s.split('o')

>>> s.split('i')

>>> '0'.join(s.split('o'))

Be sure you understand why you get each result. Then apply what you have learned to fil in the body of the function below using the split and join methods of str objects:

def myreplace(old, new, s):

"""

Replace all occurences of old with new in the string s.

>>> *myreplace(',', ';', 'this, that, and, some, other, thing')*

'this; that; and; some; other; thing'

>>> *myreplace(' ', '**', 'Words will now be separated by stars.')*

*'Words**will**now**be**separated**by**stars.'*

"""

Your solution should pass al doctests.

11. Create a module named wordtools.py with the fol owing at the bottom:

if __name__ == '__main__':

import doctest

doctest.testmod()

Explain how this statement makes both using and testing this module convenient. What wil be the value of __name__ when wordtools.py is imported from another module? What wil it be when it is run as a main program? In which case wil the doctests run? Now add bodies to each of the fol owing functions to make the doctests pass:

```python
def cleanword(word):
    """

    >>> cleanword('what?')

    'what'

    >>> cleanword('"now!"')

    'now'

    >>> cleanword('?+="word!,@$()"')

    'word'

    """

def has_dashdash(s):
    """

    >>> has_dashdash('distance--but')

    True

    >>> has_dashdash('several')

    False
```

```
    >>> has_dashdash('critters')
    False
    >>> has_dashdash('spoke--fancy')
    True
    >>> has_dashdash('yo-yo')
    False
    """

def extract_words(s):
    """
    >>> extract_words('Now is the time! "Now", is the time? Yes, now.')
    ['now', 'is', 'the', 'time', 'now', 'is', 'the', 'time', 'yes', 'now']
    >>> extract_words('she tried to curtsey as she spoke--fancy')
    ['she', 'tried', 'to', 'curtsey', 'as', 'she', 'spoke', 'fancy']
    """

def wordcount(word, wordlist):
    """
    >>> wordcount('now', ['now', 'is', 'time', 'is', 'now', 'is', 'is'])
    ['now', 2]
    >>> wordcount('is', ['now', 'is', 'time', 'is', 'now', 'is', 'the', 'is'])
    ['is', 4]
```

```
>>> wordcount('time', ['now', 'is', 'time', 'is', 'now', 'is', 'is'])
['time', 1]
>>> wordcount('frog', ['now', 'is', 'time', 'is', 'now', 'is', 'is'])
['frog', 0]
"""

def wordset(wordlist):
    """

>>> wordset(['now', 'is', 'time', 'is', 'now', 'is', 'is'])
['is', 'now', 'time']
>>> wordset(['I', 'a', 'a', 'is', 'a', 'is', 'I', 'am'])
['I', 'a', 'am', 'is']
>>> wordset(['or', 'a', 'am', 'is', 'are', 'be', 'but', 'am'])
['a', 'am', 'are', 'be', 'but', 'is', 'or']
"""

def longestword(wordset):
    """

>>> longestword(['a', 'apple', 'pear', 'grape'])
5
>>> longestword(['a', 'am', 'I', 'be'])
2
```

>>> *longestword(['this', 'that', 'supercalifragilisticexpialidocious']) 34*

"""

Save this module so you can use the tools it contains in your programs.

12. [unsorted_fruits.txt](#) contains a list of 26 fruits, each one with a name that begins with a different letter of the alphabet. Write a program named sort_fruits.py that reads in the fruits from unsorted_fruits.txt and writes them out in alphabetical order to a file named

sorted_fruits.txt.

13. Answer the fol owing questions about countletters.py:

Explain in detail what the three lines do:

infile = open('alice_in_wonderland.txt', 'r')

text = infile.read()

infile.close()

What would type(text) return after these lines have been executed?

What does the expression 128 * [0] evaluate to? Read about [ASCI](#) in Wikipedia and explain why you think the variable, counts is assigned to 128 * [0] in light of what you read.

What does

for letter **in** text:

counts[ord(letter)] += 1

do to counts?

Explain the purpose of the display function. Why does it check for values 10, 13, and 32?

What is special about those values?

Describe in detail what the lines

outfile = open('alice_counts.dat', 'w')

outfile.write(" *%-12s%s\n*" % ("Character", "Count"))
outfile.write("================\n")

do. What wil be in alice_counts.dat when they finish executing?

Final y, explain in detail what

for i **in** range(len(counts)):

if counts[i]:

outfile.write(" *%-12s%d\n*" % (display(i), counts[i]))

does. What is the purpose of if counts[i]?

14. Write a program named mean.py that takes a sequence of numbers on the command line and returns the mean of their values.

$ python mean.py 3 4

3.5

$ python mean.py 3 4 5

4.0

$ python mean.py 11 15 94.5 22

35.625

A session of your program running on the same input should produce the same output as the sample session above.

15. Write a program named median.py that takes a sequence of numbers on the command line and returns the median of their values.

$ python median.py 3 7 11

7

$ python median.py 19 85 121

85

$ python median.py 11 15 16 22

15.5

A session of your program running on the same input should produce the same output as the sample session above.

16. Modify the countletters.py program so that it takes the file to open as a command line argument. How wil you handle the naming of the output file?

11. Recursion and exceptions

11.1. Tuples and mutability

So far, you have seen two compound types: strings, which are made up of characters; and lists, which are made up of elements of any type. One of the differences we noted is that the elements of a list can be modified, but the characters in a string cannot. In other words, strings are immutable and lists are mutable.

A tuple, like a list, is a sequence of items of any type. Unlike lists, however, tuples are immutable.

Syntactical y, a tuple is a comma-separated sequence of values:

>>> tup = 2, 4, 6, 8, 10

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

>>> tup = (2, 4, 6, 8, 10)

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

>>> tup = (5,)

>>> type(tup)

<type 'tuple'>

Without the comma, Python treats (5) as an integer in parentheses:

>>> tup = (5)

>>> type(tup)

Syntax issues aside, tuples support the same sequence operations as strings and lists. The index operator selects an element from a tuple.

>>> tup = ('a', 'b', 'c', 'd', 'e')

>>> tup[0]

'a'

And the slice operator selects a range of elements.

>>> tup[1:3]

('b', 'c')

But if we try to use item assignment to modify one of the elements of the tuple, we get an error:

>>> tup[0] = 'X'

TypeError: 'tuple' object does not support item assignment

Of course, even if we can't modify the elements of a tuple, we can replace it with a different tuple:

>>> tup = ('X',) + tup[1:]

>>> tup

('X', 'b', 'c', 'd', 'e')

Alternatively, we could first convert it to a list, modify it, and convert it back into a tuple:

>>> tup = ('X', 'b', 'c', 'd', 'e')

>>> tup = list(tup)

```
>>> tup
```

['X', 'b', 'c', 'd', 'e']

```
>>> tup[0] = 'a'
```

```
>>> tup = tuple(tup)
```

```
>>> tup
```

('a', 'b', 'c', 'd', 'e')

11.2. Tuple assignment

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b:

temp = a

a = b

b = temp

If we have to do this often, this approach becomes cumbersome. Python provides a form of tuple assignment that solves this problem neatly:

a, b = b, a

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

This feature makes tuple assignment quite versatile.

Natural y, the number of variables on the left and the number of values on the right have to be the same:

>>> a, b, c, d = 1, 2, 3

ValueError: need more than 3 values to unpack

11.3. Tuples as return values

Functions can return tuples as return values. For example, we could write a function that swaps two parameters:

def swap(x, y):

return y, x

Then we can assign the return value to a tuple with two variables:

a, b = swap(a, b)

In this case, there is no great advantage in making swap a function. In fact, there is a danger in trying to encapsulate swap, which is the fol owing tempting mistake:

def swap(x, y): *# incorrect version*

x, y = y, x

If we cal this function like this:

swap(a, b)

then a and x are aliases for the same value. Changing x inside swap makes x refer to a different value, but it has no effect on a in __main__. Similarly, changing y has no effect on b.

This function runs without producing an error message, but it doesn't do what we intended. This is an example of a semantic error.

11.4. Pure functions and modifiers revisited

In [Pure functions and modifiers](#) we discussed pure functions and modifiers as related to lists. Since tuples are immutable we can not write modifiers on them.

Here is a modifier that inserts a new value into the middle of a list:

seqtools.py

#

def insert_in_middle(val, lst):

middle = len(lst)/2

lst[middle:middle] = [val]

We can run it to see that it works:

>>> from seqtools import *

>>> my_list = ['a', 'b', 'd', 'e']

>>> insert_in_middle('c', my_list)

>>> my_list

['a', 'b', 'c', 'd', 'e']

If we try to use it with a tuple, however, we get an error:

>>> my_tuple = ('a', 'b', 'd', 'e')

>>> insert_in_middle('c', my_tuple)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "seqtools.py", line 7, in insert_in_middle

lst[middle:middle] = [val]

TypeError: 'tuple' object does not support item assignment

>>>

The problem is that tuples are immutable, and don't support slice assignment. A simple solution to this problem is to make insert_in_middle a pure function:

def insert_in_middle(val, tup):

middle = len(tup)/2

return tup[:middle] + (val,) + tup[middle:]

This version now works for tuples, but not for lists or strings. If we want a version that works for al sequence types, we need a way to encapsulate our value into the correct sequence type. A smal helper function does the trick:

def encapsulate(val, seq):

if type(seq) == type(""):

return str(val)

if type(seq) == type([]):

return [val]

return (val,)

Now we can write insert_in_middle to work with each of the built-in sequence types: **def** insert_in_middle(val, seq):

middle = len(seq)/2

return seq[:middle] + encapsulate(val, seq) + seq[middle:]

The last two versions of insert_in_middle are pure functions. They don't have any side effects. Adding encapsulate and the last version of insert_in_middle to the seqtools.py module, we can test it:

>>> from seqtools import *

>>> my_string = 'abde'

>>> my_list = ['a', 'b', 'd', 'e']

>>> my_tuple = ('a', 'b', 'd', 'e')

>>> insert_in_middle('c', my_string)

'abcde'

>>> insert_in_middle('c', my_list)

['a', 'b', 'c', 'd', 'e']

>>> insert_in_middle('c', my_tuple)

('a', 'b', 'c', 'd', 'e')

>>> my_string

'abde'

The values of my_string, my_list, and my_tuple are not changed. If we want to use insert_in_middle to change them, we have to assign the value returned by the function cal back to the variable:

>>> my_string = insert_in_middle('c', my_string)

>>> my_string

'abcde'

11.5. Recursive data structures

Al of the Python data types we have seen can be grouped inside lists and tuples in a variety of ways.

Lists and tuples can also be nested, providing myriad possibilities for organizing data. The organization of data for the purpose of making it easier to use is cal ed a data structure.

It's election time and we are helping to compute the votes as they come in. Votes arriving from individual wards, precincts, municipalities, counties, and states are sometimes reported as a sum total of votes and sometimes as a list of subtotals of votes. After considering how best to store the tal ies, we decide to use a nested number list, which we define as fol ows:

A nested number list is a list whose elements are either:

a. numbers

b. nested number lists

Notice that the term, nested number list is used in its own definition. Recursive definitions like this are quite common in mathematics and computer science. They provide a concise and powerful way to describe recursive data structures that are partial y composed of smal er and simpler instances of themselves. The definition is not circular, since at some point we wil reach a list that does not have any lists as elements.

Now suppose our job is to write a function that wil sum al of the values in a nested number list. Python has a built-in function which finds the sum of a sequence of numbers:

>>> sum([1, 2, 8])

11

>>> sum((3, 5, 8.5))

16.5

>>>

For our nested number list, however, sum wil not work:

>>> sum([1, 2, [11, 13], 8])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'list'

>>>

The problem is that the third element of this list, [11, 13], is itself a list, which can not be added to 1, 2, and 8.

11.6. Recursion

To sum al the numbers in our recursive nested number list we need to traverse the list, visiting each of the elements within its nested structure, adding any numeric elements to our sum, and repeating this process with any elements which are lists.

Modern programming languages general y support recursion, which means that functions can call themselves within their definitions. Thanks to recursion, the Python code needed to sum the values of a nested number list is surprisingly short:

```
def recursive_sum(nested_num_list):

sum = 0
```

```python
    for element in nested_num_list:
        if type(element) == type([]):
            sum = sum + recursive_sum(element)
        else:
            sum = sum + element
    return sum
```

The body of recursive_sum consists mainly of a for loop that traverses nested_num_list. If element is a numerical value (the else branch), it is simply added to sum. If element is a list, then recursive_sum is cal ed again, with the element as an argument. The statement inside the function definition in which the function cal s itself is known as the recursive call.

Recursion is truly one of the most beautiful and elegant tools in computer science.

A slightly more complicated problem is finding the largest value in our nested number list: **def** recursive_max(nested_num_list):

```
"""

>>> recursive_max([2, 9, [1, 13], 8, 6])

13

>>> recursive_max([2, [[100, 7], 90], [1, 13], 8, 6])

100

>>> recursive_max([2, [[13, 7], 90], [1, 100], 8, 6])

100
```

```
>>> recursive_max([[[13, 7], 90], 2, [1, 100], 8, 6])
100
"""
    largest = nested_num_list[0]
    while type(largest) == type([]):
        largest = largest[0]
    for element in nested_num_list:
        if type(element) == type([]):
            max_of_elem = recursive_max(element)
            if largest < max_of_elem:
                largest = max_of_elem
        else: # element is not a list
            if largest < element:
                largest = element
    return largest
```

Doctests are included to provide examples of recursive_max at work.

The added twist to this problem is finding a numerical value for initializing largest. We can't just use nested_num_list[0], since that my be either a number or a list. To solve this problem we use a while loop that assigns largest to the first numerical value no matter how deeply it is nested.

The two examples above each have a base case which does not lead to a recursive cal : the case where the element is a number and not a list. Without a base case, you have infinite recursion, and your program wil not work. Python stops after reaching a maximum recursion depth and returns a runtime error.

Write the fol owing in a file named infinite_recursion.py:

infinite_recursion.py

#def recursion_depth(number):

print "Recursion depth number *%d*." % number

recursion_depth(number + 1)

recursion_depth(0)

At the unix command prompt in the same directory in which you saved your program, type the fol owing:

python infinite_recursion.py

After watching the messages flash by, you wil be presented with the end of a long traceback that ends in with the fol owing:

...

File "infinite_recursion.py", line 3, **in** recursion_depth

recursion_depth(number + 1)

RuntimeError: maximum recursion depth exceeded

We would certainly never want something like this to happen to a user of one of our programs, so before finishing the recursion discussion, let's see how errors like this are handled in Python.

11.7. Exceptions

Whenever a runtime error occurs, it creates an exception. The program stops running at this point and Python prints out the traceback, which ends with the exception that occured.

For example, dividing by zero creates an exception:

>>> print 55/0

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: integer division or modulo by zero

>>>

So does accessing a nonexistent list item:

>>> a = []

>>> print a[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: list index out of range

>>>

Or trying to make an item assignment on a tuple:

>>> tup = ('a', 'b', 'd', 'd')

>>> tup[2] = 'c'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

>>>

In each case, the error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon.

Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop. We can handle the exception using the try and except statements.

For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

filename = raw_input('Enter a file name: ')

try:

f = open (filename, "r")

except:

print 'There is no file named', filename

The try statement executes the statements in the first block. If no exceptions occur, it ignores the except statement. If any exception occurs, it executes the statements in the except branch and then continues.

We can encapsulate this capability in a function: exists takes a filename and returns true if the file exists, false if it doesn't:

def exists(filename):

try:

f = open(filename)

f.close()

return True

except:

return False

[You can use multiple except blocks to handle different kinds of exceptions (see the Errors and](#)

[Exceptions lesson from Python creator Guido van Rossum's Python Tutorial for a more complet](#)e discussion of exceptions).

If your program detects an error condition, you can make it raise an exception. Here is an example that gets input from the user and checks that the number is non-negative.

learn_exceptions.py

#def get_age():

age = input('Please enter your age: ')

if age < 0:

raise ValueError, ' *%s* is not a valid age' % age

return age

The raise statement takes two arguments: the exception type, and specific information about the error.

ValueError is the built-in exception which most closely matches the kind of error we want to raise. The

complete listing of built-in exceptions is found in the Built-in Exceptions section of the Python Library

Reference, again by Python's creator, Guido van Rossum.

If the function that cal ed get_age handles the error, then the program can continue; otherwise, Python prints the traceback and exits:

>>> get_age()

Please enter your age: 42

42

>>> get_age()

Please enter your age: -2

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "learn_exceptions.py", line 4, in get_age

raise ValueError, ' *%s* is not a valid age' % age

ValueError: -2 is not a valid age

>>>

The error message includes the exception type and the additional information you provided.

Using exception handling, we can now modify infinite_recursion.py so that it stops when it reaches the maximum recursion depth allowed:

infinite_recursion.py

#**def** recursion_depth(number):

print "Recursion depth number *%d*." % number

try:

recursion_depth(number + 1)

except:

print "Maximum recursion depth exceeded."

recursion_depth(0)

Run this version and observe the results.

11.8. Tail recursion

When the only thing returned from a function is a recursive cal , it is refered to as tail recursion.

Here is a version of the countdown function from chapter 6 written using tail recursion: **def** countdown(n):

if n == 0:

print "Blastoff!"

else:

print n

```
countdown(n-1)
```

Any computation that can be made using iteration can also be made using recursion. Here is a version of find_max written using tail recursion:

```
def find_max(seq, max_so_far):

if not seq:

return max_so_far

if max_so_far < seq[0]:

return find_max(seq[1:], seq[0])

else:

return find_max(seq[1:], max_so_far)
```

Tail recursion is considered a bad practice in Python, since the Python compiler does not handle optimization for tail recursive cal s. The recursive solution in cases like this use more system resources than the equivalent iterative solution.

11.9. Recursive mathematical functions

Several wel known mathematical functions are defined recursively. [Factorial,](#) for example, is given the special operator, !, and is defined by:

0! = 1

n! = n(n-1)

We can easily code this into Python:

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

```python
if n == 0:

return 1

else:

return n * factorial(n-1)
```

Another wel know recursive relation in mathematics is the [fibonacci sequence,](#) which is defined by: fibonacci(0) = 1

fibonacci(1) = 1

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

This can also be written easily in Python:

```python
def fibonacci (n):

if n == 0 or n == 1:

return 1

else:

return fibonacci(n-1) + fibonacci(n-2)
```

Cal ing factorial(1000) wil exceed the maximum recursion depth. And try running fibonacci(35) and see how long it takes to complete (be patient, it wil complete).

You wil be asked to write an iterative version of factorial as an exercise, and we wil see a better way to handle fibonacci in the next chapter.

11.10. List comprehensions

A list comprehension is a syntactic construct that enables lists to be created from other lists using a compact, mathematical syntax:

>>> numbers = [1, 2, 3, 4]

>>> [x**2 **for** x **in** numbers]

[1, 4, 9, 16]

>>> [x**2 **for** x **in** numbers **if** x**2 > 8]

[9, 16]

>>> [(x, x**2, x**3) **for** x **in** numbers]

[(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]

>>> files = ['bin', 'Data', 'Desktop', '.bashrc', '.ssh', '.vimrc']

>>> [name **for** name **in** files **if** name[0] != '.']

['bin', 'Data', 'Desktop']

>>> letters = ['a', 'b', 'c']

>>> [n*letter **for** n **in** numbers **for** letter **in** letters]

['a', 'b', 'c', 'aa', 'bb', 'cc', 'aaa', 'bbb', 'ccc', 'aaaa', 'bbbb', 'cccc']

>>>

The general syntax for a list comprehension expression is:

[expr **for** item1 **in** seq1 **for** item2 **in** seq2 ... **for** itemx **in** seqx **if** condition]

This list expression has the same effect as:

output_sequence = []

```
for item1 in seq1:

for item2 in seq2:

...

for itemx in seqx:

if condition:

output_sequence.append(expr)
```

As you can see, the list comprehension is much more compact.

11.11. Mini case study: tree

The fol owing program implements a subset of the behavior of the Unix tree program.

```
#!/usr/bin/env python

import os

import sys

def getroot():

if len(sys.argv) == 1:

path = '

else:

path = sys.argv[1]

if os.path.isabs(path):

tree_root = path
```

```python
    else:
        tree_root = os.path.join(os.getcwd(), path)
    return tree_root

def getdirlist(path):
    dirlist = os.listdir(path)
    dirlist = [name for name in dirlist if name[0] != '.']
    dirlist.sort()
    return dirlist

def traverse(path, prefix='|--', s='. \n', f=0, d=0):
    dirlist = getdirlist(path)
    for num, file in enumerate(dirlist):
        lastprefix = prefix[:-3] + '`--'
        dirsize = len(dirlist)
        if num < dirsize - 1:
            s += ' %s %s\n' % (prefix, file)
        else:
            s += ' %s %s\n' % (lastprefix, file)
        path2file = os.path.join(path, file)
        if os.path.isdir(path2file):
            d += 1
```

```python
        if getdirlist(path2file):
            s, f, d = traverse(path2file, '| ' + prefix, s, f, d)
        else:
            f += 1
    return s, f, d

if __name__ == '__main__':
    root = getroot()
    tree_str, files, dirs = traverse(root)
    if dirs == 1:
        dirstring = 'directory'
    else:
        dirstring = 'directories'
    if files == 1:
        filestring = 'file'
    else:
        filestring = 'files'
    print tree_str
    print ' %d %s, %d %s' % (dirs, dirstring, files, filestring)
```
You wil be asked to explore this program in several of the exercises below.

11.12. Glossary

base case

A branch of the conditional statement in a recursive function that does not result in a recursive cal .

data structure

An organization of data for the purpose of making it easier to use.

exception

An error that occurs at runtime.

handle an exception

To prevent an exception from terminating a program using the try and except statements.

immutable data type

A data type which cannot be modified. Assignments to elements or slices of immutable types cause a runtime error.

infinite recursion

A function that cal s itself recursively without ever reaching the base case. Eventual y, an infinite recursion causes a runtime error.

list comprehension

A syntactic construct which enables lists to be generated from other lists using a syntax analogous to the mathematical [set-builder notation](#).

mutable data type

A data type which can be modified. Al mutable types are compound types. Lists and dictionaries (see next chapter) are mutable data

types; strings and tuples are not.

raise

To signal an exception using the raise statement.

recursion

The process of cal ing the function that is currently executing.

recursive call

The statement in a recursive function with is a cal to itself.

recursive definition

A definition which defines something in terms of itself. To be useful it must include base cases which are not recursive. In this way it differs from a circular definition. Recursive definitions often provide an elegant way to express complex data structures.

tail recursion

A recursive cal that occurs as the last statement (at the tail) of a function definition. Tail recursion is considered bad practice in Python programs since a logical y equivalent function can be written using iteration which is more efficient (see the Wikipedia article on tail recursion for more information).

tuple

A data type that contains a sequence of elements of any type, like a list, but is immutable. Tuples can be used wherever an immutable type is required, such as a key in a dictionary (see next chapter).

tuple assignment

An assignment to al of the elements in a tuple using a single assignment statement. Tuple assignment occurs in paral el rather than in sequence, making it useful for swapping values.

11.13. Exercises

1. **def** swap(x, y): # *incorrect version*

print "before swap statement: id(x):", id(x), "id(y):", id(y) x, y = y, x

print "after swap statement: id(x):", id(x), "id(y):", id(y) a, b = 0, 1

print "before swap function call: id(a):", id(a), "id(b):", id(b) swap(a, b)

print "after swap function call: id(a):", id(a), "id(b):", id(b) Run this program and describe the results. Use the results to explain why this version of swap does not work as intended. What wil be the values of a and b after the cal to swap?

2. Create a module named seqtools.py. Add the functions encapsulate and insert_in_middle from the chapter. Add doctests which test that these two functions work as intended with al three sequence types.

3. Add each of the fol owing functions to seqtools.py:

def make_empty(seq):

"""

>>> make_empty([1, 2, 3, 4])

[]

>>> make_empty(('a', 'b', 'c'))

()

```
    >>> make_empty("No, not me!")
    ''

    """

def insert_at_end(val, seq):
    """

    >>> insert_at_end(5, [1, 3, 4, 6])
    [1, 3, 4, 6, 5]
    >>> insert_at_end('x', 'abc')
    'abcx'
    >>> insert_at_end(5, (1, 3, 4, 6))
    (1, 3, 4, 6, 5)
    """

def insert_in_front(val, seq):
    """

    >>> insert_in_front(5, [1, 3, 4, 6])
    [5, 1, 3, 4, 6]
    >>> insert_in_front(5, (1, 3, 4, 6))
    (5, 1, 3, 4, 6)
    >>> insert_in_front('x', 'abc')
    'xabc'
```

"""

def index_of(val, seq, start=0):

"""

>>> *index_of(9, [1, 7, 11, 9, 10])*

3

>>> *index_of(5, (1, 2, 4, 5, 6, 10, 5, 5))*

3

>>> *index_of(5, (1, 2, 4, 5, 6, 10, 5, 5), 4)*

6

>>> *index_of('y', 'happy birthday')*

4

>>> *index_of('banana', ['apple', 'banana', 'cherry', 'date'])*

1

>>> *index_of(5, [2, 3, 4])*

-1

>>> *index_of('b', ['apple', 'banana', 'cherry', 'date'])*

-1

"""

def remove_at(index, seq):

"""

```
>>> remove_at(3, [1, 7, 11, 9, 10])
[1, 7, 11, 10]
>>> remove_at(5, (1, 4, 6, 7, 0, 9, 3, 5))
(1, 4, 6, 7, 0, 3, 5)
>>> remove_at(2, "Yomrktown")
'Yorktown'
"""

def remove_val(val, seq):
    """
    >>> remove_val(11, [1, 7, 11, 9, 10])
    [1, 7, 9, 10]
    >>> remove_val(15, (1, 15, 11, 4, 9))
    (1, 11, 4, 9)
    >>> remove_val('what', ('who', 'what', 'when', 'where', 'why', 'how'))
    ('who', 'when', 'where', 'why', 'how')
    """

def remove_all(val, seq):
    """
    >>> remove_all(11, [1, 7, 11, 9, 11, 10, 2, 11])
    [1, 7, 9, 10, 2]
```

```
>>> remove_all('i', 'Mississippi')
'Msssspp'
"""

def count(val, seq):
    """
    >>> count(5, (1, 5, 3, 7, 5, 8, 5))
    3
    >>> count('s', 'Mississippi')
    4
    >>> count((1, 2), [1, 5, (1, 2), 7, (1, 2), 8, 5])
    2
    """

def reverse(seq):
    """
    >>> reverse([1, 2, 3, 4, 5])
    [5, 4, 3, 2, 1]
    >>> reverse(('shoe', 'my', 'buckle', 2, 1))
    (1, 2, 'buckle', 'my', 'shoe')
    >>> reverse('Python')
    'nohtyP'
```

```python
    """

def sort_sequence(seq):

    """

    >>> sort_sequence([3, 4, 6, 7, 8, 2])
    [2, 3, 4, 6, 7, 8]
    >>> sort_sequence((3, 4, 6, 7, 8, 2))
    (2, 3, 4, 6, 7, 8)
    >>> sort_sequence("nothappy")
    'ahnoppty'

    """

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

As usual, work on each of these one at a time until they pass al of the doctests.

4. Write a function, recursive_min, that returns the smal est value in a nested number list: **def** recursive_min(nested_num_list):

```python
    """

    >>> recursive_min([2, 9, [1, 13], 8, 6])
    1
    >>> recursive_min([2, [[100, 1], 90], [10, 13], 8, 6])
```

1

>>> recursive_min([2, [[13, -7], 90], [1, 100], 8, 6])

-7

>>> recursive_min([[[-13, 7], 90], 2, [1, 100], 8, 6])

-13

"""

Your function should pass the doctests.

5. Write a function recursive_count that returns the number of occurences of target in nested_number_list:

def recursive_count(target, nested_num_list):

"""

>>> recursive_count(2, [2, 9, [2, 1, 13, 2], 8, [2, 6]])

4

>>> recursive_count(7, [[9, [7, 1, 13, 2], 8], [7, 6]])

2

>>> recursive_count(15, [[9, [7, 1, 13, 2], 8], [2, 6]])

0

>>> recursive_count(5, [[5, [5, [1, 5], 5], 5], [5, 6]])

6

"""

As usual, your function should pass the doctests.

6. Write a function flatten that returns a simple list of numbers containing al the values in a nested_number_list:

def flatten(nested_num_list):

"""

>>> *flatten([2, 9, [2, 1, 13, 2], 8, [2, 6]])*

[2, 9, 2, 1, 13, 2, 8, 2, 6]

>>> *flatten([[9, [7, 1, 13, 2], 8], [7, 6]])*

[9, 7, 1, 13, 2, 8, 7, 6]

>>> *flatten([[9, [7, 1, 13, 2], 8], [2, 6]])*

[9, 7, 1, 13, 2, 8, 2, 6]

>>> *flatten([[5, [5, [1, 5], 5], 5], [5, 6]])*

[5, 5, 1, 5, 5, 5, 5, 6]

"""

Run your function to confirm that the doctests pass.

7. Write a function named readposint that prompts the user for a positive integer and then checks the input to confirm that it meets the requirements. A sample session might look like this:

>>> num = readposint()

Please enter a positive integer: yes

yes is not a positive integer. Try again.

Please enter a positive integer: 3.14

3.14 is not a positive integer. Try again.

Please enter a positive integer: -6

-6 is not a positive integer. Try again.

Please enter a positive integer: 42

>>> num

42

>>> num2 = readposint("Now enter another one: ")

Now enter another one: 31

>>> num2

31

>>>

Use Python's exception handling mechanisms in confirming that the user's input is valid.

8. Give the Python interpreter's response to each of the fol owing:

1. **>>>** nums = [1, 2, 3, 4]

>>> [x**3 **for** x **in** nums]

2. **>>>** nums = [1, 2, 3, 4]

>>> [x**2 **for** x **in** nums **if** x**2 != 4]

3. **>>>** nums = [1, 2, 3, 4]

>>> [(x, y) **for** x **in** nums **for** y **in** nums]

4. **>>>** nums = [1, 2, 3, 4]

>>> [(x, y) **for** x **in** nums **for** y **in** nums **if** x != y]

You should anticipate the results before you try them in the interpreter.

9. Use either pydoc or the on-line documentation at http://pydoc.org to find out what sys.getrecursionlimit() and sys.setrecursionlimit(n) do. Create several experiments like what was done in infinite_recursion.py to test your understanding of how these module functions work.

10. Rewrite the factorial function using iteration instead of recursion. Cal your new function with 1000 as an argument and make note of how fast it returns a value.

11. Write a program named litter.py that creates an empty file named trash.txt in each subdirectory of a directory tree given the root of the tree as an argument (or the current directory as a default). Now write a program named cleanup.py that removes al these files. Hint: Use the tree program from the mini case study as a basis for these two recursive programs.

12. Dictionaries

Al of the compound data types we have studied in detail so far —
strings, lists, and tuples—are sequence types, which use integers as
indices to access the values they contain within them.

Dictionaries are a different kind of compound type. They are
Python's built-in mapping type. They map keys, which can be any
immutable type, to values, which can be any type, just like the values
of a list or tuple.

As an example, we wil create a dictionary to translate English words
into Spanish. For this dictionary, the keys are strings.

One way to create a dictionary is to start with the empty dictionary
and add key-value pairs. The empty dictionary is denoted {}:

>>> eng2sp = {}

>>> eng2sp['one'] = 'uno'

>>> eng2sp['two'] = 'dos'

The first assignment creates a dictionary named eng2sp; the other
assignments add new key-value pairs to the dictionary. We can print
the current value of the dictionary in the usual way:

>>> print eng2sp

{'two': 'dos', 'one': 'uno'}

The key-value pairs of the dictionary are seperated by commas.
Each pair contains a key and a value separated by a colon.

The order of the pairs may not be what you expected. Python uses
complex algorithms to determine where the key-value pairs are
stored in a dictionary. For our purposes we can think of this ordering
as unpredicatable.

Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output:

>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}

It doesn't matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so there is no need to care about ordering.

Here is how we use a key to look up the corresponding value:

>>> **print** eng2sp['two']

'dos'

The key 'two' yields the value 'dos'.

12.1. Dictionary operations

The del statement removes a key-value pair from a dictionary. For example, the fol owing dictionary contains the names of various fruits and the number of each fruit in stock:

>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

>>> **print** inventory

{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}

If someone buys al of the pears, we can remove the entry from the dictionary:

>>> **del** inventory['pears']

>>> **print** inventory

{'oranges': 525, 'apples': 430, 'bananas': 312}

Or if we're expecting more pears soon, we might just change the value associated with pears:

>>> inventory['pears'] = 0

>>> print inventory

{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}

The len function also works on dictionaries; it returns the number of key-value pairs:

>>> len(inventory)

4

12.2. Dictionary methods

Dictionaries have a number of useful built-in methods.

The keys method takes a dictionary and returns a list of its keys.

>>> eng2sp.keys()

['three', 'two', 'one']

As we saw earlier with strings and lists, dictionary methods use dot notation, which specifies the name of the method to the right of the dot and the name of the object on which to apply the method immediately to the left of the dot. The parentheses indicate that this method takes no parameters.

A method cal is cal ed an invocation; in this case, we would say that we are invoking the keys method on the object eng2sp. As we wil see in a few chapters when we talk about object oriented programming, the object on which a method is invoked is actual y the first argument to the method.

The values method is similar; it returns a list of the values in the dictionary:

>>> eng2sp.values()

['tres', 'dos', 'uno']

The items method returns both, in the form of a list of tuples — one for each key-value pair:

>>> eng2sp.items()

[('three', 'tres'), ('two', 'dos'), ('one', 'uno')]

The has_key method takes a key as an argument and returns True if the key appears in the dictionary and False otherwise:

>>> eng2sp.has_key('one')

True

>>> eng2sp.has_key('deux')

False

This method can be very useful, since looking up a non-existant key in a dictionary causes a runtime error:

>>> eng2esp['dog']

Traceback (most recent call last):

File "", line 1, in

KeyError: 'dog'

>>>

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

12.3. Aliasing and copying

Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

If you want to modify a dictionary and keep a copy of the original, use the copy method. For example, opposites is a dictionary that contains pairs of opposites:

>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}

>>> alias = opposites

>>> copy = opposites.copy()

alias and opposites refer to the same object; copy refers to a fresh copy of the same dictionary. If we modify alias, opposites is also changed:

>>> alias['right'] = 'left'

>>> opposites['right']

'left'

If we modify copy, opposites is unchanged:

>>> copy['right'] = 'privilege'

>>> opposites['right']

'left'

12.4. Sparse matrices

We previously used a list of lists to represent a matrix. That is a good choice for a matrix with mostly nonzero values, but consider a [sparse matrix](#) like this one: The list representation contains a lot of zeroes:

matrix = [[0, 0, 0, 1, 0],

[0, 0, 0, 0, 0],

[0, 2, 0, 0, 0],

[0, 0, 0, 0, 0],

[0, 0, 0, 3, 0]]

An alternative is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix:

matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}

We only need three key-value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.

To access an element of the matrix, we could use the [] operator:

matrix[(0, 3)]

1

Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.

fibonacci
n ⟶ 4

fibonacci
n ⟶ 3

fibonacci
n ⟶ 2

fibonacci
n ⟶ 2

fibonacci
n ⟶ 1

fibonacci
n ⟶ 1

fibonacci
n ⟶ 0

fibonacci
n ⟶ 1

fibonacci
n ⟶ 0
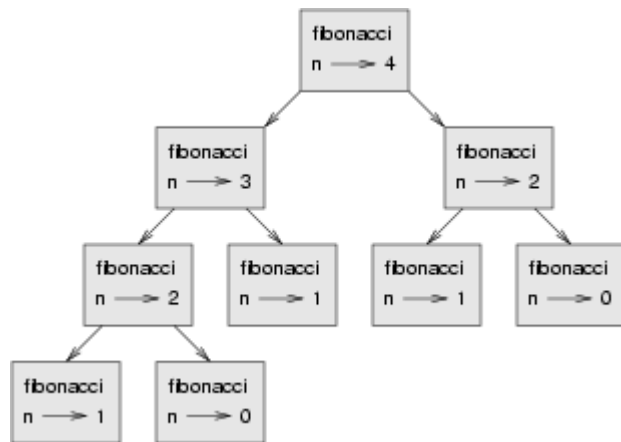
There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key:

>>> matrix[(1, 3)]

KeyError: (1, 3)

The get method solves this problem:

>>> matrix.get((0, 3), 0)

1

The first argument is the key; the second argument is the value get should return if the key is not in the dictionary:

>>> matrix.get((1, 3), 0)

0

get definitely improves the semantics of accessing a sparse matrix. Shame about the syntax.

12.5. Hints

If you played around with the fibonacci function from the last chapter, you might have noticed that the bigger the argument you provide, the

longer the function takes to run. Furthermore, the run time increases very quickly. On one of our machines, fibonacci(20) finishes instantly, fibonacci(30) takes about a second, and fibonacci(40) takes roughly forever.

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

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

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

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

previous = {0: 0, 1: 1}

def fibonacci(n):

if previous.has_key(n):

return previous[n]

else:

new_value = fibonacci(n-1) + fibonacci(n-2)

previous[n] = new_value

return new_value

The dictionary named previous keeps track of the Fibonacci numbers we already know. We start with only two pairs: 0 maps to 1; and 1 maps to 1.

Whenever fibonacci is cal ed, it checks the dictionary to determine if it contains the result. If it's there, the function can return immediately without making any more recursive cal s. If not, it has to compute the new value. The new value is added to the dictionary before the function returns.

Using this version of fibonacci, our machines can compute fibonacci(100) in an eyeblink.

>>> fibonacci(100)

354224848179261915075L

The L at the end of the number indicates that it is a long integer.

12.6. Long integers

Python provides a type cal ed long that can handle any size integer (limited only by the amount of memory you have on your computer).

There are three ways to create a long value. The first one is to compute an arithmetic expression too large to fit inside an int. We already saw this in the fibonacci(100) example above. Another way is to write an integer with a capital L at the end of your number:

>>> type(1L)

The third is to cal long with the value to be converted as an argument. long, just like int and float, can convert ints, floats, and even strings of digits to long integers:

>>> long(7)

7L

>>> long(3.9)

3L

>>> long('59')

59L

12.7. Counting letters

In Chapter 7, we wrote a function that counted the number of occurrences of a letter in a string. A more general version of this problem is to form a histogram of the letters in the string, that is, how many times each letter appears.

Such a histogram might be useful for compressing a text file. Because different letters appear with different frequencies, we can compress a file by using shorter codes for common letters and longer codes for letters that appear less frequently.

Dictionaries provide an elegant way to generate a histogram:

>>> letter_counts = {}

>>> for letter **in** "Mississippi":

... letter_counts[letter] = letter_counts.get (letter, 0) + 1

...

>>> letter_counts

{'M': 1, 's': 4, 'p': 2, 'i': 4}

We start with an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.

It might be more appealing to display the histogram in alphabetical order. We can do that with the items and sort methods:

>>> letter_items = letter_counts.items()

>>> letter_items.sort()

>>> print letter_items

[('M', 1), ('i', 4), ('p', 2), ('s', 4)]

12.8. Case Study: Robots

12.8.1. The game

In this case study we wil write a version of the classic console based game, robots.

Robots is a turn-based game in which the protagonist, you, are trying to stay alive while being chased by stupid, but relentless robots. Each robot moves one square toward you each time you move. If they catch you, you are dead, but if they col ide they die, leaving a pile of dead robot junk in their wake. If other robots col ide with the piles of junk, they die.

The basic strategy is to position yourself so that the robots col ide with each other and with piles of junk as they move toward you. To make the game playable, you also are given the ability to teleport to another location on the screen – 3 times safely and randomly thereafter, so that you don't just get forced into a corner and loose every time.

12.8.2. Setting up the world, the player, and the main loop

Let's start with a program that places the player on the screen and has a function to move her around in response to keys pressed:

robots.py

```python
#from gasp import *

SCREEN_WIDTH = 640

SCREEN_HEIGHT = 480

GRID_WIDTH = SCREEN_WIDTH/10 - 1

GRID_HEIGHT = SCREEN_HEIGHT/10 - 1

def place_player():

x = random.randint(0, GRID_WIDTH)

y = random.randint(0, GRID_HEIGHT)

return {'shape': Circle((10*x+5, 10*y+5), 5, filled=True), 'x': x, 'y': y}

def move_player(player):

update_when('key_pressed')

if key_pressed('escape'):

return True

elif key_pressed('4'):

if player['x'] > 0: player['x'] -= 1

elif key_pressed('7'):

if player['x'] > 0: player['x'] -= 1

if player['y'] < GRID_HEIGHT: player['y'] += 1

elif key_pressed('8'):

if player['y'] < GRID_HEIGHT: player['y'] += 1
```

```python
    elif key_pressed('9'):
        if player['x'] < GRID_WIDTH: player['x'] += 1
        if player['y'] < GRID_HEIGHT: player['y'] += 1
    elif key_pressed('6'):
        if player['x'] < GRID_WIDTH: player['x'] += 1
    elif key_pressed('3'):
        if player['x'] < GRID_WIDTH: player['x'] += 1
        if player['y'] > 0: player['y'] -= 1
    elif key_pressed('2'):
        if player['y'] > 0: player['y'] -= 1
    elif key_pressed('1'):
        if player['x'] > 0: player['x'] -= 1
        if player['y'] > 0: player['y'] -= 1
    else:
        return False
    move_to(player['shape'], (10*player['x']+5, 10*player['y']+5))
    return False

def play_game():
    begin_graphics(SCREEN_WIDTH, SCREEN_HEIGHT, title="Robots")
```

```
player = place_player()

finished = False

while not finished:

finished = move_player(player)

end_graphics()

if __name__ == '__main__':

play_game()
```

Programs like this one that involve interacting with the user through events such as key presses and mouse clicks are cal ed [event-driven programs](#).

The main event loop at this stage is simply:

```
while not finished:

finished = move_player(player)
```

The event handling is done inside the move_player function. update_when('key_pressed') waits until a key has been pressed before moving to the next statement. The multi-way branching statement then handles the al keys relevent to game play.

Pressing the escape key causes move_player to return True, making not finished false, thus exiting the main loop and ending the game. The 4, 7, 8, 9, 6, 3, 2, and 1 keys al cause the player to move in the appropriate direction, if she isn't blocked by the edge of a window.

12.8.3. Adding a robot

Now let's add a single robot that heads toward the player each time the player moves.

Add the fol owing place_robot function between place_player and move_player:

def place_robot():

x = random.randint(0, GRID_WIDTH)

y = random.randint(0, GRID_HEIGHT)

return {'shape': Box((10*x, 10*y), 10, 10), 'x': x, 'y': y}

Add move_robot immediately after move_player:

def move_robot(robot, player):

if robot['x'] < player['x']: robot['x'] += 1

elif robot['x'] > player['x']: robot['x'] -= 1

if robot['y'] < player['y']: robot['y'] += 1

elif robot['y'] > player['y']: robot['y'] -= 1

move_to(robot['shape'], (10*robot['x'], 10*robot['y']))

We need to pass both the robot and the player to this function so that it can compare their locations and move the robot toward the player.

Now add the line robot = place_robot() in the main body of the program immediately after the line player = place_player(), and add the move_robot(robot, player) cal inside the main loop immediately after finished = move_player(player).

12.8.4. Checking for Collisions

We now have a robot that moves relentlessly toward our player, but once it catches her it just fol ows her around wherever she goes.

What we want to happen is for the game to end as soon as the player is caught. The fol owing function wil determine if that has happened:

def collided(robot, player):

return player['x'] == robot['x'] **and** player['y'] == robot['y']

Place this new function immediately below the move_player function. Now let's modify play_game to check for col isions:

def play_game():

begin_graphics(SCREEN_WIDTH, SCREEN_HEIGHT)

player = place_player()

robot = place_robot()

defeated = False

while not defeated:

quit = move_player(player)

if quit:

break

move_robot(robot, player)

defeated = collided(robot, player)

if defeated:

remove_from_screen(player['shape'])

remove_from_screen(robot['shape'])

Text("They got you!", (240, 240), size=32)

sleep(3)

end_graphics()

We rename the variable finished to defeated, which is now set to the result of collided. The main loop runs as long as defeated is false. Pressing the key stil ends the program, since we check for quit and break out of the main loop if it is true. Final y, we check for defeated immediately after the main loop and display an appropriate message if it is true.

12.8.5. Adding more robots

There are several things we could do next:

give the player the ability to teleport to another location to escape pursuit.

provide safe placement of the player so that it never starts on top of a robot.

add more robots.

Adding the ability to teleport to a random location is the easiest task, and it has been left to you to complete as an exercise.

How we provide safe placement of the player wil depend on how we represent multiple robots, so it makes sense to tackle adding more robots first.

To add a second robot, we could just create another variable named something like robot2 with another cal to place_robot. The problem with this approach is that we wil soon want lots of robots, and giving them al their own names wil be cumbersome. A more elegant solution is to place al the robots in a list: **def** place_robots(numbots):

```
robots = []

for i in range(numbots):

robots.append(place_robot())

return robots
```

Now instead of cal ing place_robot in play_game, cal place_robots, which returns a single list containing al the robots:

```
robots = place_robots(2)
```

With more than one robot placed, we have to handle moving each one of them. We have already solved the problem of moving a single robot, however, so traversing the list and moving each one in turn does the trick:

```
def move_robots(robots, player):

for robot in robots:

move_robot(robot, player)
```

Add move_robots immediately after move_robot, and change play_game to cal move_robots instead of move_robot.

We now need to check each robot to see if it has col ided with the player:

```
def check_collisions(robots, player):

for robot in robots:

if collided(robot, player):

return True
```

return False

Add check_collisions immediately after collided and change the line in play_game that sets defeated to cal check_collisions instead of collided.

Final y, we need to loop over robots to remove each one in turn if defeated becomes true. Adding this has been left as an exercise.

12.8.6. Winning the game

The biggest problem left in our game is that there is no way to win. The robots are both relentless and indestructible. With careful maneuvering and a bit of luck teleporting, we can reach the point where it appears there is only one robot chasing the player (al the robots wil actual y just be on top of each other). This moving pile of robots wil continue chasing our hapless player until it catches it, either by a bad move on our part or a teleport that lands the player directly on the robots.

When two robots col ide they are supposed to die, leaving behind a pile of junk. A robot (or the player) is also supposed to die when it col ides with a pile of junk. The logic for doing this is quite tricky. After the player and each of the robots have moved, we need to:

1. Check whether the player has col ided with a robot or a pile of junk. If so, set defeated to true and break out of the game loop.

2. Check each robot in the robots list to see if it has col ided with a pile of junk. If it has, disgard the robot (remove it from the robots list).

3. Check each of the remaining robots to see if they have col ided with another robot. If they have, disgard al the robots that have col ided and place a pile of junk at the locations they occupied.

4. Check if any robots remain. If not, end the game and mark the player the winner.

Let's take on each of these tasks in turn.

12.8.7. Adding **junk**

Most of this work wil take place inside our check_collisions function. Let's start by modifying collided, changing the names of the parameters to reflect its more general use:

```
def collided(thing1, thing2):
    return thing1['x'] == thing2['x'] and thing1['y'] == thing2['y']
```

We now introduce a new empty list named junk immediately after the cal to place_robots: junk = []

and modify check_collisions to incorporate the new list:

```
def check_collisions(robots, junk, player):
    # check whether player has collided with anything
    for thing in robots + junk:
        if collided(thing, player):
            return True
    return False
```

Be sure to modify the cal to check_collisions (currently defeated = check_collisions(robots, player)) to include junk as a new argument.

Again, we need to fix the logic after if defeated: to remove the new junk from the screen before displaying the They got you! message:

```
for thing in robots + junk:
    remove_from_screen(thing['shape'])
```

Since at this point junk is always an empty list, we haven't changed the behavior of our program. To test whether our new logic is actual y working, we could introduce a single junk pile and run our player into it, at which point the game should remove al items from the screen and display the ending message.

It wil be helpful to modify our program temporarily to change the random placement of robots and player to predetermined locations for testing. We plan to use solid boxes to represent junk piles. We observe that placing a robot is very similar to placing a junk pile, and modify place_robot to do both: **def** place_robot(x, y, junk=False):

return {'shape': Box((10*x, 10*y), 10, 10, filled=junk), 'x': x, 'y': y}

Notice that x and y are now parameters, along with a new parameter that we wil use to set filled to true for piles of junk.

Our program is now broken, since the cal in place_robots to place_robot does not pass arguments for x and y. Fixing this and setting up the program for testing is left to you as an exercise.

12.8.8. Removing robots that hit junk

To remove robots that col ide with piles of junk, we add a nested loop to check_collisions between each robot and each pile of junk. Our first attempt at this does not work:

def check_collisions(robots, junk, player):

 # check whether player has collided with anything

for thing **in** robots + junk:

if collided(thing, player):

return True

 # remove robots that have collided with a pile of junk

for robot **in** robots:

for pile **in** junk:

if collided(robot, pile):

robots.remove(robot)

return False

Running this new code with the program as setup in exercise 11, we find a bug. It appears that the robots continue to pass through the pile of junk as before.

Actual y, the bug is more subtle. Since we have two robots on top of each other, when the col ision of the first one is detected and that robot is removed, we move the second robot into the first position in the list and it is missed by the next iteration. It is general y dangerous to modify a list while you are iterating over it. Doing so can introduce a host of difficult to find errors into your program.

The solution in this case is to loop over the robots list backwards, so that when we remove a robot from the list al the robots whose list indeces change as a result are robots we have already evaluated.

As usual, Python provides an elegant way to do this. The built-in function, reversed provides for backward iteration over a sequence. Replacing:

for robot **in** robots:

with:

for robot **in** reversed(robots):

wil make our program work the way we intended.

12.8.9. Turning robots into junk and enabling the player to win

We now want to check each robot to see if it has col ided with any other robots. We wil remove al robots that have col ided, leaving a single pile of junk in their wake. If we reach a state where there are no more robots, the player wins.

Once again we have to be careful not to introduce bugs related to removing things from a list over which we are iterating.

Here is the plan:

1. Check each robot in robots (an outer loop, traversing forward).

2. Compare it with every robot that fol ows it (an inner loop, traversing backward).

3. If the two robots have col ided, add a piece of junk at their location, mark the first robot as junk, and remove the second one.

4. Once al robots have been checked for col isions, traverse the robots list once again in reverse, removing al robots marked as junk.

5. Check to see if any robots remain. If not, declare the player the winner.

Adding the fol owing to check_collisions wil accomplish most of what we need to do:

 # remove robots that collide and leave a pile of junk

for index, robot1 **in** enumerate(robots):

for robot2 **in** reversed(robots[index+1:]):

if collided(robot1, robot2):

robot1['junk'] = True

junk.append(place_robot(robot1['x'], robot1['y'], True))

```
remove_from_screen(robot2['shape'])

robots.remove(robot2)

for robot in reversed(robots):

if robot['junk']:

remove_from_screen(robot['shape'])

robots.remove(robot)
```

We make use of the enumerate function we saw in Chapter 9 to get both the index and value of each robot as we traverse forward. Then a reverse traversal of the slice of the remaining robots, reversed(robots[index+1:]), sets up the col ision check.

Whenever two robots col ide, our plan cal s for adding a piece of junk at that location, marking the first robot for later removal (we stil need it to compare with the other robots), and immediately removing the second one. The body of the if collided(robot1, robot2): conditional is designed to do just that, but if you look careful y at the line:

```
robot1['junk'] = True
```

you should notice a problem. robot1['junk'] wil result in a systax error, since our robot dictionary does not yet contain a 'junk' key. To fix this we modify place_robot to accomodate the new key: **def** place_robot(x, y, junk=False):

```
return {'shape': Box((10*x, 10*y), 10, 10, filled=junk),

'x': x, 'y': y, 'junk': junk}
```

It is not at al unusual for data structures to change as program development proceeds. Stepwise refinement of both program data and logic is a normal part of the structured programming process.

After robot1 is marked as junk, we add a pile of junk to the junk list at the same location with junk.append(place_robot(robot1['x'], robot1['y'], True)), and then remove robot2 from the game by first removing its shape from the graphics window and then removing it from the robots list.

The next loop traverses backward over the robots list removing al the robots previously marked as junk.

Since the player wins when al the robots die, and the robot list wil be empty when it no longer contains live robots, we can simply check whether robots is empty to determine whether or not the player has won.

This can be done in check_collisions immediately after we finish checking robot col isions and removing dead robots by adding:

if not robots:

return ...

Hmmm... What should we return? In its current state, check_collisions is a boolean function that returns true when the player has col ided with something and lost the game, and false when the player has not lost and the game should continue. That is why the variable in the play_game function that catches the return value is cal ed defeated.

Now we have three possible states:

1. robots is not empty and the player has not col ided with anything – the game is stil in play 2. the player has col ided with something – the robots win

3. the player has not col ided with anything and robots is empty – the player wins

In order to handle this with as few changes as possible to our present program, we wil take advantage of the way that Python permits sequence types to live double lives as boolean values. We wil return an empty string – which is false – when game play should continue, and either "robots_win" or

"player_wins" to handle the other two cases. check_collisions should now look like this: **def** check_collisions(robots, junk, player):

 # check whether player has collided with anything

for thing **in** robots + junk:

if collided(thing, player):

return "robots_win"

 # remove robots that have collided with a pile of junk

for robot **in** reversed(robots):

for pile **in** junk:

if collided(robot, pile):

robots.remove(robot)

 # remove robots that collide and leave a pile of junk

for index, robot1 **in** enumerate(robots):

for robot2 **in** reversed(robots[index+1:]):

if collided(robot1, robot2):

robot1['junk'] = True

junk.append(place_robot(robot1['x'], robot1['y'], True))

```
remove_from_screen(robot2['shape'])

robots.remove(robot2)

for robot in reversed(robots):

if robot['junk']:

remove_from_screen(robot['shape'])

robots.remove(robot)

if not robots:

return "player_wins"

return ""
```

A few corresponding changes need to be made to play_game to use the new return values. These are left as an exercise.

12.9. Glossary

dictionary

A col ection of key-value pairs that maps from keys to values. The keys can be any immutable type, and the values can be any type.

mapping type

A mapping type is a data type comprised of a col ection of keys and associated values. Python's only built-in mapping type is the dictionary. Dictionaries implement the [associative array](#) abstract data type.

key

A data item that is mapped to a value in a dictionary. Keys are used to look up values in a dictionary.

key-value pair

One of the pairs of items in a dictionary. Values are looked up in a dictionary by key.

hint

Temporary storage of a precomputed value to avoid redundant computation.

event

A signal such as a keyboard press, mouse click, or message from another program.

event-driven program

<fil in definition here>

event loop

A programming construct that waits for events and processes them.

overflow

A numerical result that is too large to be represented in a numerical format.

12.10. Exercises

1. Write a program that reads in a string on the command line and returns a table of the letters of the alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored. A sample run of the program

would look this this: $ python letter_counts.py "ThiS is String with Upper and lower case Letters."

a 2

c 1

d 1

e 5

g 1

h 2

i 4

l 2

n 2

o 1

p 2

r 4

s 5

t 5

u 1

w 2

$

2. Give the Python interpreter's response to each of the fol owing from a continuous interpreter session:

1. **>>>** d = {'apples': 15, 'bananas': 35, 'grapes': 12}

>>> d['banana']

2. **>>>** d['oranges'] = 20

>>> len(d)

3. **>>>** d.has_key('grapes')

4. **>>>** d['pears']

5. **>>>** d.get('pears', 0)

6. **>>>** fruits = d.keys()

>>> fruits.sort()

>>> print fruits

7. **>>> del** d['apples']

>>> d.has_key('apples')

Be sure you understand why you get each result. Then apply what you have learned to fil in the body of the function below:

def add_fruit(inventory, fruit, quantity=0):

"""

Adds quantity of fruit to inventory.

>>> new_inventory = {}

>>> add_fruit(new_inventory, 'strawberries', 10)

>>> new_inventory.has_key('strawberries')

True

>>> new_inventory['strawberries']

10

>>> add_fruit(new_inventory, 'strawberries', 25)

>>> new_inventory['strawberries']

"""

Your solution should pass the doctests.

3. Write a program cal ed alice_words.py that creates a text file named alice_words.txt containing an alphabetical listing of al the words found in alice_in_wonderland.txt together with the number of times each word occurs. The first 10 lines of your output file should look something like this: Word Count

========================

a 631

a-piece 1

abide 1

able 1

about 94

above 3

absence 1

absurd 2

How many times does the word, alice, occur in the book?

4. What is the longest word in Alice in Wonderland ? How many charactes does it have?

5. Copy the code from the Setting up the world, the player, and the main loop section into a file named robots.py and run it. You should be able to move the player around the screen using the numeric keypad and to quit the program by pressing the escape key.

6. Laptops usual y have smal er keyboards than desktop computers that do not include a seperate numeric keypad. Modify the robots program so that it uses 'a', 'q', 'w', 'e', 'd', 'c', 'x', and 'z'

instead of '4', '7', '8', '9', '6', '3', '2', and '1' so that it wil work on a typical laptop keyboard.

7. Add al the code from the Adding a robot section in the places indicated. Make sure the program works and that you now have a robot fol owing around your player.

8. Add al the code from the Checking for Collisions section in the places indicated. Verify that the program ends when the robot catches the player after displaying a They got you! message for 3

seconds.

9. Modify the move_player function to add the ability for the player to jump to a random location whenever the 0 key is pressed. (hint: place_player already has the logic needed to place the player in a random location. Just add another conditional branch to move_player that uses this logic when key_pressed('0') is true.) Test the program to verify that your player can now teleport to a random place on the screen to get out of trouble.

10. Make al the changes to your program indicated in Adding more robots. Be sure to loop over the robots list, removing each robot in

turn, after defeated becomes true. Test your program to verify that there are now two robots chasing your player. Let a robot catch you to test whether you have correctly handled removing al the robots. Change the argument from 2 to 4 in robots =

place_robots(2) and confirm that you have 4 robots.

11. Make the changes to your program indicated in Adding ``junk``. Fix place_robots by moving the random generation of values for x and y to the appropriate location and passing these values as arguments in the cal to place_robot. Now we are ready to make temporary modifications to our program to remove the randomness so we can control it for testing. We can start by placing a pile of junk in the center of our game board. Change:

junk = []

to:

junk = [place_robot(GRID_WIDTH/2, GRID_HEIGHT/2, junk=True)]

Run the program and confirm that there is a black box in the center of the board.Now change place_player so that it looks like this:

def place_player():

```
# x = random.randint(0, GRID_WIDTH)

# y = random.randint(0, GRID_HEIGHT)

x, y = GRID_WIDTH/2 + 3, GRID_HEIGHT/2

return {'shape': Circle((10*x+5, 10*y+5), 5, filled=True), 'x': x, 'y': y}
```

Final y, temporarily comment out the random generation of x and y values in place_robots and the creation of numbots robots. Replace this logic with code to create two robots in fixed locations: **def** place_robots(numbots):

```
robots = []

# for i in range(numbots):

# x = random.randint(0, GRID_WIDTH)

# y = random.randint(0, GRID_HEIGHT)

# robots.append(place_robot(x, y))

robots.append(place_robot(GRID_WIDTH/2 - 4, GRID_HEIGHT/2 + 2))

robots.append(place_robot(GRID_WIDTH/2 - 4, GRID_HEIGHT/2 - 2))

return robots
```

When you start your program now, it should look like this:

When you run this program and either stay stil (by pressing the 5 repeatedly) or move away from the pile of junk, you can confirm that the robots move through it unharmed. When you move into the junk pile, on the other hand, you die.

12. Make the fol owing modifications to play_game to integrate with the changes made in Turning robots into junk and enabling the player to win:

1. Rename defeated to winner and initialize it to the empty string instead of False.

13. Classes and objects

13.1. Object-oriented programming

Python is an object-oriented programming language, which means that it provides features that support object-oriented programming (OOP).

Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1980s that it became the main programming paradigm used in the creation of new software. It was developed as a way to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time.

Up to now we have been writing programs using a procedural programming paradigm. In procedural programming the focus is on writing functions or procedures which operate on data. In object-oriented programming the focus is on the creation of objects which contain both data and functionality together.

13.2. User-defined compound types

A class in essence defines a new data type. We have been using several of Python's built-in types throughout this book, we are now ready to create our own user-defined type: the Point.

Consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated col ectively as a single object. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

A natural way to represent a point in Python is with two numeric values. The question, then, is how to group these two values into a compound object. The quick and dirty solution is to use a list or tuple, and for some applications that might be the best choice.

An alternative is to define a new user-defined compound type, also cal ed a class. This approach involves a bit more effort, but it has advantages that wil be apparent soon.

A class definition looks like this:

class Point:

pass

Class definitions can appear anywhere in a program, but they are usual y near the beginning (after the import statements). The syntax rules for a class definition are the same as for other compound statements. There is a header which begins with the keyword, class, fol owed by the name of the class, and ending with a colon.

This definition creates a new class cal ed Point. The pass statement has no effect; it is only necessary because a compound statement

must have something in its body. A docstring could serve the same purpose:

class Point:

"Point class for storing mathematical points."

By creating the Point class, we created a new type, also cal ed Point. The members of this type are cal ed instances of the type or objects. Creating a new instance is cal ed instantiation, and is accomplished by calling the class. Classes, like functions, are cal able, and we instantiate a Point object by cal ing the Point class:
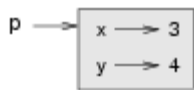
>>> type(Point)

<type 'classobj'>

>>> p = Point()

>>> type(p)

<type 'instance'>

The variable p is assigned a reference to a new Point object.

It may be helpful to think of a class as a factory for making objects, so our Point class is a factory for making points. The class itself isn't an instance of a point, but it contains the machinary to make point instances.

13.3. Attributes

Like real world objects, object instances have both form and function. The form consists of data elements contained within the instance.

We can add new data elements to an instance using dot notation:

>>> p.x = 3

>>> p.y = 4

This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.uppercase. Both modules and instances create their own namespaces, and the syntax for accessing names contained in each, cal ed attributes, is the same. In this case the attribute we are selecting is a data item from an instance.

The fol owing state diagram shows the result of these assignments:

The variable p refers to a Point object, which contains two attributes. Each attribute refers to a number.

We can read the value of an attribute using the same syntax:

>>> print p.y

4**>>>** x = p.x

>>> print x

3

The expression p.x means, "Go to the object p refers to and get the value of x". In this case, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x. The purpose of dot notation is to identify which variable you are referring to unambiguously.

You can use dot notation as part of any expression, so the fol owing statements are legal: **print** '(*%d, %d*)' % (p.x, p.y)

distance_squared = p.x * p.x + p.y * p.y

The first line outputs (3, 4); the second line calculates the value 25.

13.4. The initialization method and **self**

Since our Point class is intended to represent two dimensional mathematical points, all point instances ought to have x and y attributes, but that is not yet so with our Point objects.

>>> p2 = Point()

>>> p2.x

Traceback (most recent call last):

File "<stdin>", line 1, in ?

AttributeError: Point instance has no attribute 'x'

>>>

To solve this problem we add an initialization method to our class.

class Point:

def __init__(self, x=0, y=0):

self.x = x

self.y = y

A method behaves like a function but it is part of an object. Like a data attribute it is accessed using dot notation. The initialization method is cal ed automatical y when the class is cal ed.

Let's add another method, distance_from_origin, to see better how methods work:

class Point:

```python
def __init__(self, x=0, y=0):

self.x = x

self.y = y

def distance_from_origin(self):

return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

Let's create a few point instances, look at their attributes, and cal our new method on them:

```python
>>> p = Point(3, 4)

>>> p.x

3>>> p.y

4>>> p.distance_from_origin()

5.0

>>> q = Point(5, 12)

>>> q.x

5>>> q.y

12

>>> q.distance_from_origin()

13.0

>>> r = Point()

>>> r.x
```

0**>>>** r.y

0**>>>** r.distance_from_origin()

0.0

When defining a method, the first parameter refers to the instance being created. It is customary to name this parameter self. In the example session above, the self parameter refers to the instances p, q, and r respectively.

13.5. Instances as parameters

You can pass an instance as a parameter to a function in the usual way. For example: **def** print_point(p):

print '(*%s*, *%s*)' % (str(p.x), str(p.y))

print_point takes a point as an argument and displays it in the standard format. If you cal print_point(p) with point p as defined previously, the output is (3, 4).

13.6. Glossary

class

A user-defined compound type. A class can also be thought of as a template for the objects that are instances of it.

instantiate

To create an instance of a class.

instance

An object that belongs to a class.

object

A compound data type that is often used to model a thing or concept in the real world.

attribute

One of the named data items that makes up an instance.

13.7. Exercises

1. Create and print a Point object, and then use id to print the object's unique identifier. Translate the hexadecimal form into decimal and confirm that they match.

2. Rewrite the distance function from chapter 5 so that it takes two Points as parameters instead of four numbers.

14. Classes and functions

14.1. Time

As another example of a user-defined type, we'l define a class cal ed Time that records the time of day.

The class definition looks like this:

class Time:

pass

We can create a new Time object and assign attributes for hours, minutes, and seconds: time = Time()

time.hours = 11

time.minutes = 59

time.seconds = 30

The state diagram for the Time object looks like this:

14.2. Pure functions

In the next few sections, we'l write two versions of a function cal ed add_time, which calculates the sum of two Times. They wil demonstrate two kinds of functions: pure functions and modifiers.

The fol owing is a rough version of add_time:

def add_time(t1, t2):

sum = Time()

sum.hours = t1.hours + t2.hours

sum.minutes = t1.minutes + t2.minutes

sum.seconds = t1.seconds + t2.seconds

return sum

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is cal ed a pure function because it does not modify any of the objects passed to it as parameters and it has no side effects, such as displaying a value or getting user input.

Here is an example of how to use this function. We'l create two Time objects: current_time, which contains the current time; and bread_time, which contains the amount of time it takes for a breadmaker to make bread. Then we'l use add_time to figure out when the bread wil be done. If you haven't finished writing print_time yet, take a look ahead to Section before you try this:

>>> current_time = Time()

>>> current_time.hours = 9

>>> current_time.minutes = 14

>>> current_time.seconds = 30

>>> bread_time = Time()

>>> bread_time.hours = 3

>>> bread_time.minutes = 35

>>> bread_time.seconds = 0

>>> done_time = add_time(current_time, bread_time)

>>> print_time(done_time)

12:49:30

The output of this program is 12:49:30, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to carry the extra seconds into the minutes

column or the extra minutes into the hours column.

Here's a second corrected version of the function:

```
def add_time(t1, t2):

sum = Time()

sum.hours = t1.hours + t2.hours

sum.minutes = t1.minutes + t2.minutes

sum.seconds = t1.seconds + t2.seconds

if sum.seconds >= 60:

sum.seconds = sum.seconds - 60

sum.minutes = sum.minutes + 1

if sum.minutes >= 60:

sum.minutes = sum.minutes - 60

sum.hours = sum.hours + 1

return sum
```

Although this function is correct, it is starting to get big. Later we wil suggest an alternative approach that yields shorter code.

14.3. Modifiers

There are times when it is useful for a function to modify one or more of the objects it gets as parameters. Usual y, the cal er keeps a reference to the objects it passes, so any changes the function makes are visible to the cal er. Functions that work this way are cal ed modifiers.

increment, which adds a given number of seconds to a Time object, would be written most natural y as a modifier. A rough draft of the function looks like this:

def increment(time, seconds):

time.seconds = time.seconds + seconds

if time.seconds >= 60:

time.seconds = time.seconds - 60

time.minutes = time.minutes + 1

if time.minutes >= 60:

time.minutes = time.minutes - 60

time.hours = time.hours + 1

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if the parameter seconds is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until seconds is less than sixty. One solution is to replace the if statements with while statements:

```
def increment(time, seconds):

time.seconds = time.seconds + seconds

while time.seconds >= 60:

time.seconds = time.seconds - 60

time.minutes = time.minutes + 1

while time.minutes >= 60:

time.minutes = time.minutes - 60

time.hours = time.hours + 1
```

This function is now correct, but it is not the most efficient solution.

14.4. Prototype development versus planning

In this chapter, we demonstrated an approach to program development that we cal prototype development. In each case, we wrote a rough draft (or prototype) that performed the basic calculation and then tested it on a few cases, correcting flaws as we found them.

Although this approach can be effective, it can lead to code that is unnecessarily complicated – since it deals with many special cases – and unreliable – since it is hard to know if you have found al the errors.

An alternative is planned development, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is real y a three-digit number in base 60! The second component is the ones column, the minute component is the sixties column, and the hour component is the thirty-six hundreds column.

When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem – we can convert a Time object into a single number and take advantage of the fact that the computer knows how to do arithmetic with numbers. The fol owing function converts a Time object into an integer:

def convert_to_seconds(t):

minutes = t.hours * 60 + t.minutes

seconds = minutes * 60 + t.seconds

return seconds

Now, al we need is a way to convert from an integer to a Time object:

def make_time(seconds):

time = Time()

time.hours = seconds/3600

seconds = seconds - time.hours * 3600

time.minutes = seconds/60

seconds = seconds - time.minutes * 60

time.seconds = seconds

return time

You might have to think a bit to convince yourself that this technique to convert from one base to another is correct. Assuming you are

convinced, you can use these functions to rewrite add_time: **def** add_time(t1, t2):

seconds = convert_to_seconds(t1) + convert_to_seconds(t2)

return make_time(seconds)

This version is much shorter than the original, and it is much easier to demonstrate that it is correct (assuming, as usual, that the functions it cal s are correct).

14.5. Generalization

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (convert_to_seconds and make_time), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironical y, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

14.6. Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written

an algorithm. We mentioned this word before but did not define it careful y.

It is not easy to define, so we wil try a couple of approaches.

First, consider something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were lazy, you probably cheated by learning a few tricks. For example, to find the product of *n* and 9, you can write n-1 as the first digit and 10-n as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are al algorithms. One of the characteristics of algorithms is that they do not require any intel igence to carry out. They are mechanical processes in which each step fol ows from the last according to a simple set of rules.

In our opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literal y, require no intel igence.

On the other hand, the process of designing algorithms is interesting, intel ectual y chal enging, and a central part of what we cal programming.

Some of the things that people do natural y, without difficulty or conscious thought, are the hardest to express algorithmical y. Understanding natural language is a good example. We al do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

14.7. Glossary

pure function

A function that does not modify any of the objects it receives as parameters. Most pure functions are fruitful.

modifier

A function that changes one or more of the objects it receives as parameters. Most modifiers are void.

functional programming style

A style of program design in which the majority of functions are pure.

prototype development

A way of developing programs starting with a prototype and gradual y testing and improving it.

planned development

A way of developing programs that involves high-level insight into the problem and more planning than incremental development or prototype development.

algorithm

A set of instructions for solving a class of problems by a mechanical, unintel igent process.

14.8. Exercises

1. Write a function print_time that takes a Time object as an argument and prints it in the form hours:minutes:seconds.

2. Write a boolean function after that takes two Time objects, t1 and t2, as arguments, and returns True if t1 fol ows t2 chronological y and False otherwise.

3. Rewrite the increment function so that it doesn't contain any loops.

4. Now rewrite increment as a pure function, and write function cal s to both versions.

15. Classes and methods

15.1. Object-oriented features

Python is an object-oriented programming language, which means that it provides features that support object-oriented programming.

It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

1. Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.

2. Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the Time class defined in the last chapter corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the Point and Rectangle classes correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. Strictly speaking, these features are not necessary. For the most part, they provide an alternative syntax for things we have already done, but in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in the Time program, there is no obvious connection between the class definition and the function definitions that fol ow. With some examination, it is apparent that every function takes at least one Time object as a parameter.

This observation is the motivation for methods. We have already seen some methods, such as keys and values, which were invoked on dictionaries. Each method is associated with a class and is intended to be invoked on instances of that class.

Methods are just like functions, with two differences:

1. Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

2. The syntax for invoking a method is different from the syntax for cal ing a function.

In the next few sections, we wil take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it simply by fol owing a sequence of steps. If you are comfortable converting from one form to another, you wil be able to choose the best form for whatever you are doing.

15.2. **print_time**

In the last chapter, we defined a class named Time and you wrote a function named print_time, which should have looked something like this:

```python
class Time:

pass

def print_time(time):

print (str(time.hours) + ":" +

str(time.minutes) + ":" +

str(time.seconds))
```

To cal this function, we passed a Time object as a parameter:

```python
>>> current_time = Time()

>>> current_time.hours = 9

>>> current_time.minutes = 14

>>> current_time.seconds = 30

>>> print_time(current_time)
```

To make print_time a method, al we have to do is move the function definition inside the class definition. Notice the change in indentation.

```python
class Time:

def print_time(time):

print (str(time.hours) + ":" +

str(time.minutes) + ":" +

str(time.seconds))
```

Now we can invoke print_time using dot notation.

>>> current_time.print_time()

As usual, the object on which the method is invoked appears before the dot and the name of the method appears after the dot.

The object on which the method is invoked is assigned to the first parameter, so in this case current_time is assigned to the parameter time.

By convention, the first parameter of a method is cal ed self. The reason for this is a little convoluted, but it is based on a useful metaphor.

The syntax for a function cal , print_time(current_time), suggests that the function is the active agent.

It says something like, Hey print_time! Here's an object for you to print.

In object-oriented programming, the objects are the active agents. An invocation like current_time.print_time() says Hey current_time! Please print yourself!

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

15.3. Another example

Let's convert increment to a method. To save space, we wil leave out previously defined methods, but you should keep them in your version:

class Time:

#previous method definitions here...

def increment(self, seconds):

self.seconds = seconds + self.seconds

while self.seconds >= 60:

self.seconds = self.seconds - 60

self.minutes = self.minutes + 1

while self.minutes >= 60:

self.minutes = self.minutes - 60

self.hours = self.hours + 1

The transformation is purely mechanical - we move the method definition into the class definition and change the name of the first parameter.

Now we can invoke increment as a method.

current_time.increment(500)

Again, the object on which the method is invoked gets assigned to the first parameter, self. The second parameter, seconds gets the value 500.

15.4. A more complicated example

The after function is slightly more complicated because it operates on two Time objects, not just one.

We can only convert one of the parameters to self; the other stays the same:

class Time:

#previous method definitions here...

def after(self, time2):

if self.hour > time2.hour:

return True

if self.hour < time2.hour:

return False

if self.minute > time2.minute:

return True

if self.minute < time2.minute:

return False

if self.second > time2.second:

return True

return False

We invoke this method on one object and pass the other as an argument:

if doneTime.after(current_time):

print "The bread will be done after it starts."

You can almost read the invocation like English: If the done-time is after the current-time, then...

15.5. Optional arguments

We have seen built-in functions that take a variable number of arguments. For example, string.find can take two, three, or four arguments.

It is possible to write user-defined functions with optional argument lists. For example, we can upgrade our own version of find to do the same thing as string.find.

This is the original version:

```
def find(str, ch):

index = 0

while index < len(str):

if str[index] == ch:

return index

index = index + 1

return -1
```

This is the new and improved version:

```
def find(str, ch, start=0):

index = start

while index < len(str):

if str[index] == ch:

return index

index = index + 1
```

return -1

The third parameter, start, is optional because a default value, 0, is provided. If we invoke find with only two arguments, we use the default value and start from the beginning of the string:

>>> find("apple", "p")

1

If we provide a third parameter, it overrides the default:

>>> find("apple", "p", 2)

2**>>>** find("apple", "p", 3)

-1

15.6. The initialization method

The initialization method is a special method that is invoked when an object is created. The name of this method is __init__ (two underscore characters, fol owed by init, and then two more underscores).

An initialization method for the Time class looks like this:

class Time:

def __init__(self, hours=0, minutes=0, seconds=0):

self.hours = hours

self.minutes = minutes

self.seconds = seconds

There is no conflict between the attribute self.hours and the parameter hours. Dot notation specifies which variable we are referring to.

When we invoke the Time constructor, the arguments we provide are passed along to init:

>>> current_time = Time(9, 14, 30)

>>> current_time.print_time()

>>> 9:14:30

Because the parameters are optional, we can omit them:

>>> current_time = Time()

>>> current_time.print_time()

>>> 0:0:0

Or provide only the first parameter:

>>> current_time = Time (9)

>>> current_time.print_time()

>>> 9:0:0

Or the first two parameters:

>>> current_time = Time (9, 14)

>>> current_time.print_time()

>>> 9:14:0

Final y, we can provide a subset of the parameters by naming them explicitly:

>>> current_time = Time(seconds = 30, hours = 9)

>>> current_time.print_time()

>>> 9:0:30

15.7. Points revisited

Let's rewrite the Point class from chapter 12 in a more object-oriented style:

class Point:

def __init__(self, x=0, y=0):

self.x = x

self.y = y

def __str__(self):

return '(' + str(self.x) + ', ' + str(self.y) + ')'

The initialization method takes x and y values as optional parameters; the default for either parameter is 0.

The next method, __str__, returns a string representation of a Point object. If a class provides a method named __str__, it overrides the default behavior of the Python built-in str function.

>>> p = Point(3, 4)

>>> str(p)

'(3, 4)'

Printing a Point object implicitly invokes __str__ on the object, so defining __str__ also changes the behavior of print:

>>> p = Point(3, 4)

>>> print p

(3, 4)

When we write a new class, we almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is almost always useful for debugging.

15.8. Operator overloading

Some languages make it possible to change the definition of the built- in operators when they are applied to user-defined types. This feature is cal ed operator overloading. It is especial y useful when defining new mathematical types.

For example, to override the addition operator +, we provide a method named __add__: **class Point**:

 # previously defined methods here...

def __add__(self, other):

return Point(self.x + other.x, self.y + other.y)

As usual, the first parameter is the object on which the method is invoked. The second parameter is conveniently named other to distinguish it from self. To add two Points, we create and return a new Point that contains the sum of the x coordinates and the sum of the y coordinates.

Now, when we apply the + operator to Point objects, Python invokes __add__:

>>> p1 = Point(3, 4)

>>> p2 = Point(5, 7)

>>> p3 = p1 + p2

>>> print p3

(8, 11)

The expression p1 + p2 is equivalent to p1.__add__(p2), but obviously more elegant. As an exercise, add a method __sub__(self, other) that overloads the subtraction operator, and try it out. There are several ways to override the behavior of the multiplication operator: by defining a method named __mul__, or __rmul__, or both.

If the left operand of * is a Point, Python invokes __mul__, which assumes that the other operand is also a Point. It computes the dot product of the two points, defined according to the rules of linear algebra: **def** __mul__(self, other):

return self.x * other.x + self.y * other.y

If the left operand of * is a primitive type and the right operand is a Point, Python invokes __rmul__, which performs scalar multiplication:

def __rmul__(self, other):

return Point(other * self.x, other * self.y)

The result is a new Point whose coordinates are a multiple of the original coordinates. If other is a type that cannot be multiplied by a floating-point number, then __rmul__ wil yield an error.

This example demonstrates both kinds of multiplication:

>>> p1 = Point(3, 4)

>>> p2 = Point(5, 7)

>>> **print** p1 * p2

43

>>> **print** 2 * p2

(10, 14)

What happens if we try to evaluate p2 * 2? Since the first parameter is a Point, Python invokes __mul__

with 2 as the second argument. Inside __mul__, the program tries to access the x coordinate of other, which fails because an integer has no attributes:

>>> **print** p2 * 2

AttributeError: 'int' object has no attribute 'x'

Unfortunately, the error message is a bit opaque. This example demonstrates some of the difficulties of object-oriented programming. Sometimes it is hard enough just to figure out what code is running.

For a more complete example of operator overloading, see Appendix (reference overloading).

15.9. Polymorphism

Most of the methods we have written only work for a specific type. When you create a new object, you write methods that operate on that type.

But there are certain operations that you wil want to apply to many types, such as the arithmetic operations in the previous sections. If many types support the same set of operations, you can write functions that work on any of those types.

For example, the multadd operation (which is common in linear algebra) takes three parameters; it multiplies the first two and then adds the third. We can write it in Python like this: **def** multadd (x, y, z):

return x * y + z

This method wil work for any values of x and y that can be multiplied and for any value of z that can be added to the product.

We can invoke it with numeric values:

>>> multadd (3, 2, 1)

7

Or with Points:

>>> p1 = Point(3, 4)

>>> p2 = Point(5, 7)

>>> print multadd (2, p1, p2)

(11, 15)

>>> print multadd (p1, p2, 1)

44

In the first case, the Point is multiplied by a scalar and then added to another Point. In the second case, the dot product yields a numeric value, so the third parameter also has to be a numeric value.

A function like this that can take parameters with different types is cal ed polymorphic.

As another example, consider the method front_and_back, which prints a list twice, forward and backward:

def front_and_back(front):

import copy

back = copy.copy(front)

back.reverse()

print str(front) + str(back)

Because the reverse method is a modifier, we make a copy of the list before reversing it. That way, this method doesn't modify the list it gets as a parameter.

Here's an example that applies front_and_back to a list:

>>> myList = [1, 2, 3, 4]

>>> front_and_back(myList)

[1, 2, 3, 4][4, 3, 2, 1]

Of course, we intended to apply this function to lists, so it is not surprising that it works. What would be surprising is if we could apply it to a Point.

To determine whether a function can be applied to a new type, we apply the fundamental rule of polymorphism: If all of the operations inside the function can be applied to the type, the function can be applied to the type. The operations in the method include copy, reverse, and print.

copy works on any object, and we have already written a __str__ method for Points, so al we need is a reverse method in the Point class:

def reverse(self):

self.x , self.y = self.y, self.x

Then we can pass Points to front_and_back:

>>> p = Point(3, 4)

>>> front_and_back(p)

(3, 4)(4, 3)

The best kind of polymorphism is the unintentional kind, where you discover that a function you have already written can be applied to a type for which you never planned.

15.10. Glossary

object-oriented language

A language that provides features, such as user-defined classes and inheritance, that facilitate object-oriented programming.

object-oriented programming

A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method

A function that is defined inside a class definition and is invoked on instances of that class.

:override:: To replace a default. Examples include replacing a default parameter with a particular argument and replacing a default method by providing a new method with the same name.

initialization method

A special method that is invoked automatical y when a new object is created and that initializes the object's attributes.

operator overloading

Extending built-in operators (+, -, *, >, <, etc.) so that they work with user-defined types.

dot product

An operation defined in linear algebra that multiplies two Points and yields a numeric value.

scalar multiplication

An operation defined in linear algebra that multiplies each of the coordinates of a Point by a numeric value.

polymorphic

A function that can operate on more than one type. If al the operations in a function can be applied to a type, then the function can be applied to a type.

15.11. Exercises

1. Convert the function convertToSeconds:

def convertToSeconds(t):

minutes = t.hours * 60 + t.minutes

seconds = minutes * 60 + t.seconds

return seconds

to a method in the Time class.

2. Add a fourth parameter, end, to the find function that specifies
where to stop looking. Warning: This exercise is a bit tricky. The
default value of end should be len(str), but that doesn't work.

The default values are evaluated when the function is defined, not
when it is cal ed. When find is defined, str doesn't exist yet, so you
can't find its length.

16. Sets of objects

16.1. Composition

By now, you have seen several examples of composition. One of the
first examples was using a method invocation as part of an
expression. Another example is the nested structure of statements;
you can put an if statement within a while loop, within another if
statement, and so on.

Having seen this pattern, and having learned about lists and objects,
you should not be surprised to learn that you can create lists of
objects. You can also create objects that contain lists (as attributes);

you can create lists that contain lists; you can create objects that contain objects; and so on.

In this chapter and the next, we wil look at some examples of these combinations, using Card objects as an example.

16.2. **Card** objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King.

Depending on the game that you are playing, the rank of Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like "Spade" for suits and "Queen" for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to encode the ranks and suits. By encode, we do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by encode is to define a mapping between a sequence of numbers and the items I want to represent. For example:

Spades --> 3

Hearts --> 2

Diamonds --> 1

Clubs --> 0

An obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack --> 11

Queen --> 12

King --> 13

The reason we are using mathematical notation for these mappings is that they are not part of the Python program. They are part of the program design, but they never appear explicitly in the code. The class definition for the Card type looks like this:

class Card:

def __init__(self, suit=0, rank=0):

self.suit = suit

self.rank = rank

As usual, we provide an initialization method that takes an optional parameter for each attribute.

To create an object that represents the 3 of Clubs, use this command: three_of_clubs = Card(0, 3)

The first argument, 0, represents the suit Clubs.

16.3. Class attributes and the **__str__** method

In order to print Card objects in a way that people can easily read, we want to map the integer codes onto words. A natural way to do

that is with lists of strings. We assign these lists to class attributes at the top of the class definition:

class Card:

suits = ["Clubs", "Diamonds", "Hearts", "Spades"]

ranks = ["narf", "Ace", "2", "3", "4", "5", "6", "7",

"8", "9", "10", "Jack", "Queen", "King"]

def __init__(self, suit=0, rank=0):

self.suit = suit

self.rank = rank

def __str__(self):

return (self.ranks[self.rank] + " of " + self.suits[self.suit])

A class attribute is defined outside of any method, and it can be accessed from any of the methods in the class.

Inside __str__, we can use suits and ranks to map the numerical values of suit and rank to strings. For example, the expression self.suits[self.suit] means use the attribute suit from the object self as an index into the class attribute named suits, and select the appropriate string.

The reason for the "narf" in the first element in ranks is to act as a place keeper for the zero-eth element of the list, which wil never be used. The only valid ranks are 1 to 13. This wasted item is not entirely necessary. We could have started at 0, as usual, but it is less confusing to encode 2 as 2, 3 as 3, and so on.

With the methods we have so far, we can create and print cards:

>>> card1 = Card(1, 11)

>>> print card1

Jack of Diamonds

Class attributes like suits are shared by al Card objects. The advantage of this is that we can use any Card object to access the class attributes:

>>> card2 = Card(1, 3)

>>> print card2

3 of Diamonds

>>> print card2.suits[1]

Diamonds

The disadvantage is that if we modify a class attribute, it affects every instance of the class. For example, if we decide that Jack of Diamonds should real y be cal ed Jack of Swirly Whales, we could do this:

>>> card1.suits[1] = "Swirly Whales"

>>> print card1

Jack of Swirly Whales

The problem is that all of the Diamonds just became Swirly Whales:

>>> print card2

3 of Swirly Whales

It is usual y not a good idea to modify class attributes.

16.4. Comparing cards

For primitive types, there are conditional operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For user-defined types, we can override the behavior of the built-in operators by providing a method named __cmp__. By convention, __cmp__ takes two parameters, self and other, and returns 1 if the first object is greater, -1 if the second object is greater, and 0 if they are equal to each other.

Some types are completely ordered, which means that you can compare any two elements and tel which is bigger. For example, the integers and the floating-point numbers are completely ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why you cannot compare apples and oranges.

The set of playing cards is partial y ordered, which means that sometimes you can compare cards and sometimes not. For example, you know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to make cards comparable, you have to decide which is more important, rank or suit. To be honest, the choice is arbitrary. For the sake of choosing, we wil say that suit is more important, because a new deck of cards comes sorted with al the Clubs together, fol owed by al the Diamonds, and so on.

With that decided, we can write __cmp__:

def __cmp__(self, other):

 # check the suits

```python
if self.suit > other.suit: return 1
if self.suit < other.suit: return -1
# suits are the same... check ranks
if self.rank > other.rank: return 1
if self.rank < other.rank: return -1
# ranks are the same... it's a tie
return 0
```

In this ordering, Aces appear lower than Deuces (2s).

16.5. Decks

Now that we have objects to represent Cards, the next logical step is to define a class to represent a Deck. Of course, a deck is made up of cards, so each Deck object wil contain a list of cards as an attribute.

The fol owing is a class definition for the Deck class. The initialization method creates the attribute cards and generates the standard set of fifty-two cards:

```python
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                self.cards.append(Card(suit, rank))
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0

to 3. The inner loop enumerates the ranks from 1 to 13. Since the outer loop iterates four times, and the

inner loop iterates thirteen times, the total number of times the body is executed is fifty-two (thirteen times four). Each iteration creates a new instance of Card with the current suit and rank, and appends that card to the cards list.

The append method works on lists but not, of course, tuples.

16.6. Printing the deck

As usual, when we define a new type of object we want a method that prints the contents of an object.

To print a Deck, we traverse the list and print each Card:

class Deck:

...

def print_deck(self):

for card **in** self.cards:

print card

Here, and from now on, the el ipsis (...) indicates that we have omitted the other methods in the class.

As an alternative to print_deck, we could write a __str__ method for the Deck class. The advantage of __str__ is that it is more flexible. Rather than just printing the contents of the object, it generates a string representation that other parts of the program can manipulate before printing, or store for later use.

Here is a version of __str__ that returns a string representation of a Deck. To add a bit of pizzazz, it arranges the cards in a cascade where each card is indented one space more than the previous card:

class Deck:

...

def __str__(self):

s = ""

for i **in** range(len(self.cards)):

s = s + " " * i + str(self.cards[i]) + " **\n**"

return s

This example demonstrates several features. First, instead of traversing self.cards and assigning each card to a variable, we are using i as a loop variable and an index into the list of cards.

Second, we are using the string multiplication operator to indent each card by one more space than the last. The expression " " * i yields a number of spaces equal to the current value of i.

Third, instead of using the print command to print the cards, we use the str function. Passing an object as an argument to str is equivalent to invoking the __str__ method on the object.

Final y, we are using the variable s as an accumulator. Initial y, s is the empty string. Each time through the loop, a new string is generated and concatenated with the old value of s to get the new value. When the loop ends, s contains the complete string representation of the Deck, which looks like this:

>>> deck = Deck()

>>> print deck

Ace of Clubs

2 of Clubs

3 of Clubs

4 of Clubs

5 of Clubs

6 of Clubs

7 of Clubs

8 of Clubs

9 of Clubs

10 of Clubs

Jack of Clubs

Queen of Clubs

King of Clubs

Ace of Diamonds

And so on. Even though the result appears on 52 lines, it is one long string that contains newlines.

16.7. Shuffling the deck

If a deck is perfectly shuffled, then any card is equal y likely to appear anywhere in the deck, and any location in the deck is equal y likely to contain any card.

To shuffle the deck, we wil use the randrange function from the random module. With two integer arguments, a and b, randrange chooses a random integer in the range a <= x < b. Since the upper bound is strictly less than b, we can use the length of a list as the second parameter, and we are guaranteed to get a legal index. For example, this expression chooses the index of a random card in a deck:

```
random.randrange(0, len(self.cards))
```

An easy way to shuffle the deck is by traversing the cards and swapping each card with a randomly chosen one. It is possible that the card wil be swapped with itself, but that is fine. In fact, if we precluded that possibility, the order of the cards would be less than entirely random: **class Deck**:

...

def shuffle(self):

import random

num_cards = len(self.cards)

for i **in** range(num_cards):

j = random.randrange(i, num_cards)

self.cards[i], self.cards[j] = self.cards[j], self.cards[i]

Rather than assume that there are fifty-two cards in the deck, we get the actual length of the list and store it in num_cards.

For each card in the deck, we choose a random card from among the cards that haven't been shuffled yet. Then we swap the current card (i) with the selected card (j). To swap the cards we use a tuple assignment:

self.cards[i], self.cards[j] = self.cards[j], self.cards[i]

16.8. Removing and dealing cards

Another method that would be useful for the Deck class is remove, which takes a card as a parameter, removes it, and returns True if the card was in the deck and False otherwise:

class Deck:

...

def remove(self, card):

if card **in** self.cards:

self.cards.remove(card)

return True

else:

return False

The in operator returns True if the first operand is in the second, which must be a list or a tuple. If the first operand is an object, Python uses the object's __cmp__ method to determine equality with items in the list. Since the __cmp__ in the Card class checks for deep equality, the remove method checks for deep equality.

To deal cards, we want to remove and return the top card. The list method pop provides a convenient way to do that:

class Deck:

...

def pop(self):

return self.cards.pop()

Actual y, pop removes the last card in the list, so we are in effect dealing from the bottom of the deck.

One more operation that we are likely to want is the boolean function is_empty, which returns true if the deck contains no cards:

class Deck:

...

def is_empty(self):

return (len(self.cards) == 0)

16.9. Glossary

encode

To represent one set of values using another set of values by constructing a mapping between them.

class attribute

A variable that is defined inside a class definition but outside any method. Class attributes are accessible from any method in the class and are shared by al instances of the class.

accumulator

A variable used in a loop to accumulate a series of values, such as by concatenating them onto a string or adding them to a running sum.

16.10. Exercises

1. Modify __cmp__ so that Aces are ranked higher than Kings.

17. Inheritance

17.1. Inheritance

The language feature most often associated with object-oriented programming is inheritance.

Inheritance is the ability to define a new class that is a modified version of an existing class.

The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. It is cal ed inheritance because the new class inherits al of the methods of the existing class. Extending this metaphor, the existing class is sometimes cal ed the parent class. The new class may be cal ed the child class or sometimes subclass.

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its

definition. The relevant code may be scattered among several modules. Also, many of the things that can be done using inheritance can be done as elegantly (or more so) without it. If the natural structure of the problem does not lend itself to inheritance, this style of programming can do more harm than good.

In this chapter we wil demonstrate the use of inheritance as part of a program that plays the card game Old Maid. One of our goals is to write code that could be reused to implement other card games.

17.2. A hand of cards

For almost any card game, we need to represent a hand of cards. A hand is similar to a deck, of course.

Both are made up of a set of cards, and both require operations like adding and removing cards. Also, we might like the ability to shuffle both decks and hands.

A hand is also different from a deck. Depending on the game being played, we might want to perform some operations on hands that don't make sense for a deck. For example, in poker we might classify a hand (straight, flush, etc.) or compare it with another hand. In bridge, we might want to compute a score for a hand in order to make a bid.

This situation suggests the use of inheritance. If Hand is a subclass of Deck, it wil have al the methods of Deck, and new methods can be added.

In the class definition, the name of the parent class appears in parentheses:

class Hand(Deck):

pass

This statement indicates that the new Hand class inherits from the existing Deck class.

The Hand constructor initializes the attributes for the hand, which are name and cards. The string name identifies this hand, probably by the name of the player that holds it. The name is an optional parameter with the empty string as a default value. cards is the list of cards in the hand, initialized to the empty list:

class Hand(Deck):

def __init__(self, name=""):

self.cards = []

self.name = name

For just about any card game, it is necessary to add and remove cards from the deck. Removing cards is already taken care of, since Hand inherits remove from Deck. But we have to write add: **class Hand**(Deck):

...

def add(self,card):

self.cards.append(card)

Again, the el ipsis indicates that we have omitted other methods. The list append method adds the new card to the end of the list of cards.

17.3. Dealing cards

Now that we have a Hand class, we want to deal cards from the Deck into hands. It is not immediately obvious whether this method should go in the Hand class or in the Deck class, but since it operates on a single deck and (possibly) several hands, it is more natural to put it in Deck.

deal should be fairly general, since different games wil have different requirements. We may want to deal out the entire deck at once or add one card to each hand.

deal takes two parameters, a list (or tuple) of hands and the total number of cards to deal. If there are not enough cards in the deck, the method deals out al of the cards and stops:

class Deck :

...

def deal(self, hands, num_cards=999):

num_hands = len(hands)

for i **in** range(num_cards):

if self.is_empty(): **break** # *break if out of cards*

card = self.pop() # *take the top card*

hand = hands[i % num_hands] # *whose turn is next?*

hand.add(card) # *add the card to the hand*

The second parameter, num_cards, is optional; the default is a large number, which effectively means that al of the cards in the deck wil get dealt.

The loop variable i goes from 0 to nCards-1. Each time through the loop, a card is removed from the deck using the list method pop, which removes and returns the last item in the list.

The modulus operator (%) al ows us to deal cards in a round robin (one card at a time to each hand).

When i is equal to the number of hands in the list, the expression i % nHands wraps around to the beginning of the list (index 0).

17.4. Printing a Hand

To print the contents of a hand, we can take advantage of the printDeck and __str__ methods inherited from Deck. For example:

>>> deck = Deck()

>>> deck.shuffle()

>>> hand = Hand("frank")

>>> deck.deal([hand], 5)

>>> print hand

Hand frank contains

2 of Spades

3 of Spades

4 of Spades

Ace of Hearts

9 of Clubs

It's not a great hand, but it has the makings of a straight flush.

Although it is convenient to inherit the existing methods, there is additional information in a Hand object we might want to include when we print one. To do that, we can provide a __str__ method in the Hand class that overrides the one in the Deck class:

class Hand(Deck)

...

def __str__(self):

s = "Hand " + self.name

if self.is_empty():

s = s + " is empty**\n**"

else:

s = s + " contains**\n**"

return s + Deck.__str__(self)

Initial y, s is a string that identifies the hand. If the hand is empty, the program appends the words is empty and returns s.

Otherwise, the program appends the word contains and the string representation of the Deck, computed by invoking the __str__ method in the Deck class on self.

It may seem odd to send self, which refers to the current Hand, to a Deck method, until you remember that a Hand is a kind of Deck. Hand objects can do everything Deck objects can, so it is legal to send a Hand to a Deck method.

In general, it is always legal to use an instance of a subclass in place of an instance of a parent class.

17.5. The **CardGame** class

The CardGame class takes care of some basic chores common to al games, such as creating the deck and shuffling it:

class CardGame:

```
def __init__(self):

    self.deck = Deck()

    self.deck.shuffle()
```

This is the first case we have seen where the initialization method performs a significant computation, beyond initializing attributes.

To implement specific games, we can inherit from CardGame and add features for the new game. As an example, we'l write a simulation of Old Maid.

The object of Old Maid is to get rid of cards in your hand. You do this by matching cards by rank and color. For example, the 4 of Clubs matches the 4 of Spades since both suits are black. The Jack of Hearts matches the Jack of Diamonds since both are red.

To begin the game, the Queen of Clubs is removed from the deck so that the Queen of Spades has no match. The fifty-one remaining cards are dealt to the players in a round robin. After the deal, al players match and discard as many cards as possible.

When no more matches can be made, play begins. In turn, each player picks a card (without looking) from the closest neighbor to the left who stil has cards. If the chosen card matches a card in the player's hand, the pair is removed. Otherwise, the card is added to the player's hand. Eventual y al possible matches are made, leaving only the Queen of Spades in the loser's hand.

In our computer simulation of the game, the computer plays al hands. Unfortunately, some nuances of the real game are lost. In a real game, the player with the Old Maid goes to some effort to get their neighbor to pick that card, by displaying it a little more prominently, or perhaps failing to display it

more prominently, or even failing to fail to display that card more prominently. The computer simply picks a neighbor's card at random.

17.6. **OldMaidHand** class

A hand for playing Old Maid requires some abilities beyond the general abilities of a Hand. We wil define a new class, OldMaidHand, that inherits from Hand and provides an additional method cal ed remove_matches:

class OldMaidHand(Hand):

def remove_matches(self):

count = 0

original_cards = self.cards[:]

for card **in** original_cards:

match = Card(3 - card.suit, card.rank)

if match **in** self.cards:

self.cards.remove(card)

self.cards.remove(match)

print "Hand *%s*: *%s* matches *%s*" % (self.name, card, match) count = count + 1

return count

We start by making a copy of the list of cards, so that we can traverse the copy while removing cards from the original. Since self.cards is modified in the loop, we don't want to use it to control the traversal. Python can get quite confused if it is traversing a list that is changing!

For each card in the hand, we figure out what the matching card is and go looking for it. The match card has the same rank and the other suit of the same color. The expression 3 - card.suit turns a Club (suit 0) into a Spade (suit 3) and a Diamond (suit 1) into a Heart (suit 2). You should satisfy yourself that the opposite operations also work. If the match card is also in the hand, both cards are removed.

The fol owing example demonstrates how to use remove_matches:

>>> game = CardGame()

>>> hand = OldMaidHand("frank")

>>> game.deck.deal([hand], 13)

>>> print hand

Hand frank contains

Ace of Spades

2 of Diamonds

7 of Spades

8 of Clubs

6 of Hearts

8 of Spades

7 of Clubs

Queen of Clubs

7 of Diamonds

5 of Clubs

Jack of Diamonds

10 of Diamonds

10 of Hearts

>>> hand.remove_matches()

Hand frank: 7 of Spades matches 7 of Clubs

Hand frank: 8 of Spades matches 8 of Clubs

Hand frank: 10 of Diamonds matches 10 of Hearts

>>> print hand

Hand frank contains

Ace of Spades

2 of Diamonds

6 of Hearts

Queen of Clubs

7 of Diamonds

5 of Clubs

Jack of Diamonds

Notice that there is no __init__ method for the OldMaidHand class. We inherit it from Hand.

17.7. **OldMaidGame** class

Now we can turn our attention to the game itself. OldMaidGame is a subclass of CardGame with a new method cal ed play that takes a

list of players as a parameter.

Since __init__ is inherited from CardGame, a new OldMaidGame object contains a new shuffled deck: **class OldMaidGame**(CardGame):

def play(self, names):

remove Queen of Clubs

self.deck.remove(Card(0,12))

make a hand for each player

self.hands = []

for name **in** names:

self.hands.append(OldMaidHand(name))

deal the cards

self.deck.deal(self.hands)

print "---------- Cards have been dealt"

self.printHands()

remove initial matches

matches = self.removeAllMatches()

print "---------- Matches discarded, play begins"

self.printHands()

play until all 50 cards are matched

turn = 0

```
numHands = len(self.hands)

while matches < 25:

    matches = matches + self.playOneTurn(turn)

    turn = (turn + 1) % numHands

print "---------- Game is Over"

self.printHands()
```

The writing of printHands() is left as an exercise.

Some of the steps of the game have been separated into methods. remove_all_matches traverses the list of hands and invokes remove_matches on each:

```
class OldMaidGame(CardGame):

    ...

    def remove_all_matches(self):

        count = 0

        for hand in self.hands:

            count = count + hand.remove_matches()

        return count
```

count is an accumulator that adds up the number of matches in each hand and returns the total.

When the total number of matches reaches twenty-five, fifty cards have been removed from the hands, which means that only one card is left and the game is over.

The variable turn keeps track of which player's turn it is. It starts at 0 and increases by one each time; when it reaches numHands, the modulus operator wraps it back around to 0.

The method playOneTurn takes a parameter that indicates whose turn it is. The return value is the number of matches made during this turn:

class OldMaidGame(CardGame):

...

def play_one_turn(self, i):

if self.hands[i].is_empty():

return 0

neighbor = self.find_neighbor(i)

pickedCard = self.hands[neighbor].popCard()

self.hands[i].add(pickedCard)

print "Hand", self.hands[i].name, "picked", pickedCard

count = self.hands[i].remove_matches()

self.hands[i].shuffle()

return count

If a player's hand is empty, that player is out of the game, so he or she does nothing and returns 0.

Otherwise, a turn consists of finding the first player on the left that has cards, taking one card from the neighbor, and checking for

matches. Before returning, the cards in the hand are shuffled so that the next player's choice is random.

The method find_neighbor starts with the player to the immediate left and continues around the circle until it finds a player that stil has cards:

class OldMaidGame(CardGame):

...

def find_neighbor(self, i):

numHands = len(self.hands)

for next **in** range(1,numHands):

neighbor = (i + next) % numHands

if not self.hands[neighbor].is_empty():

return neighbor

If find_neighbor ever went al the way around the circle without finding cards, it would return None and cause an error elsewhere in the program. Fortunately, we can prove that that wil never happen (as long as the end of the game is detected correctly).

We have omitted the print_hands method. You can write that one yourself.

The fol owing output is from a truncated form of the game where only the top fifteen cards (tens and higher) were dealt to three players. With this smal deck, play stops after seven matches instead of twenty-five.

>>> import cards

```
>>> game = cards.OldMaidGame()

>>> game.play(["Allen","Jeff","Chris"])

---------- Cards have been dealt

Hand Allen contains

King of Hearts

Jack of Clubs

Queen of Spades

King of Spades

10 of Diamonds

Hand Jeff contains

Queen of Hearts

Jack of Spades

Jack of Hearts

King of Diamonds

Queen of Diamonds

Hand Chris contains

Jack of Diamonds

King of Clubs

10 of Spades

10 of Hearts
```

10 of Clubs

Hand Jeff: Queen of Hearts matches Queen of Diamonds

Hand Chris: 10 of Spades matches 10 of Clubs

---------- Matches discarded, play begins

Hand Allen contains

King of Hearts

Jack of Clubs

Queen of Spades

King of Spades

10 of Diamonds

Hand Jeff contains

Jack of Spades

Jack of Hearts

King of Diamonds

Hand Chris contains

Jack of Diamonds

King of Clubs

10 of Hearts

Hand Allen picked King of Diamonds

Hand Allen: King of Hearts matches King of Diamonds

Hand Jeff picked 10 of Hearts

Hand Chris picked Jack of Clubs

Hand Allen picked Jack of Hearts

Hand Jeff picked Jack of Diamonds

Hand Chris picked Queen of Spades

Hand Allen picked Jack of Diamonds

Hand Allen: Jack of Hearts matches Jack of Diamonds

Hand Jeff picked King of Clubs

Hand Chris picked King of Spades

Hand Allen picked 10 of Hearts

Hand Allen: 10 of Diamonds matches 10 of Hearts

Hand Jeff picked Queen of Spades

Hand Chris picked Jack of Spades

Hand Chris: Jack of Clubs matches Jack of Spades

Hand Jeff picked King of Spades

Hand Jeff: King of Clubs matches King of Spades

---------- Game is Over

Hand Allen is empty

Hand Jeff contains

Queen of Spades

Hand Chris is empty

So Jeff loses.

17.8. Glossary

inheritance

The ability to define a new class that is a modified version of a previously defined class.

parent class

The class from which a child class inherits.

child class

A new class created by inheriting from an existing class; also cal ed a subclass.

17.9. Exercises

1. Add a method, print_hands, to the OldMaidGame class which traverses self.hands and prints each hand.

18. Linked lists

18.1. Embedded references

We have seen examples of attributes that refer to other objects, which we cal ed embedded references.

A common data structure, the linked list, takes advantage of this feature.

Linked lists are made up of nodes, where each node contains a reference to the next node in the list. In addition, each node contains a unit of data cal ed the cargo.

A linked list is considered a recursive data structure because it has a recursive definition.

A linked list is either:

1. the empty list, represented by None, or

2. a node that contains a cargo object and a reference to a linked list.

Recursive data structures lend themselves to recursive methods.

18.2. The **Node** class

As usual when writing a new class, we'l start with the initialization and __str__ methods so that we can test the basic mechanism of creating and displaying the new type:

```
class Node:

def __init__(self, cargo=None, next=None):

self.cargo = cargo

self.next = next

def __str__(self):
```

return str(self.cargo)

As usual, the parameters for the initialization method are optional. By default, both the cargo and the link, next, are set to None.

The string representation of a node is just the string representation of the cargo. Since any value can be passed to the str function, we can store any value in a list.

To test the implementation so far, we can create a Node and print it:

>>> node = Node("test")

>>> print node

test

To make it interesting, we need a list with more than one node:

>>> node1 = Node(1)

>>> node2 = Node(2)

>>> node3 = Node(3)

This code creates three nodes, but we don't have a list yet because the nodes are not linked. The state diagram looks like this:

To link the nodes, we have to make the first node refer to the second and the second node refer to the third:

>>> node1.next = node2

>>> node2.next = node3

The reference of the third node is None, which indicates that it is the end of the list. Now the state diagram looks like this:

Now you know how to create nodes and link them into lists. What might be less clear at this point is why.

18.3. Lists as collections

Lists are useful because they provide a way to assemble multiple objects into a single entity, sometimes cal ed a collection. In the example, the first node of the list serves as a reference to the entire list.

To pass the list as a parameter, we only have to pass a reference to the first node. For example, the function print_list takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end:

def print_list(node):

while node:

print node,

node = node.next

print

To invoke this method, we pass a reference to the first node:

>>> print_list(node1)

1 2 3

Inside print_list we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the next value from each node to get to the next node.

To traverse a linked list, it is common to use a loop variable like node to refer to each of the nodes in succession.

This diagram shows the value of list and the values that node takes on:

18.4. Lists and recursion

It is natural to express many list operations using recursive methods. For example, the fol owing is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (cal ed the head); and the rest (cal ed the tail).

2. Print the tail backward.

3. Print the head.

Of course, Step 2, the recursive cal , assumes that we have a way of printing a list backward. But if we assume that the recursive cal works – the leap of faith – then we can convince ourselves that this algorithm works.

Al we need are a base case and a way of proving that for any list, we wil eventual y get to the base case.

Given the recursive definition of a list, a natural base case is the empty list, represented by None: **def** print_backward(list):

if list == None: **return**

head = list

tail = list.next

print_backward(tail)

print head,

The first line handles the base case by doing nothing. The next two lines split the list into head and tail.

The last two lines print the list. The comma at the end of the last line keeps Python from printing a

newline after each node.

We invoke this method as we invoked print_list:

>>> print_backward(node1)

3 2 1

The result is a backward list.

You might wonder why print_list and print_backward are functions and not methods in the Node class.

The reason is that we want to use None to represent the empty list and it is not legal to invoke a method on None. This limitation makes it awkward to write list – manipulating code in a clean object-oriented style.

Can we prove that print_backward wil always terminate? In other words, wil it always reach the base case? In fact, the answer is no. Some lists wil make this method crash.

18.5. Infinite lists

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself:

If we invoke print_list on this list, it wil loop forever. If we invoke print_backward, it wil recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasional y useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction.

Regardless, it is problematic that we cannot prove that print_list and print_backward terminate. The best we can do is the hypothetical statement, If the list contains no loops, then these methods wil terminate. This sort of claim is cal ed a precondition. It imposes a constraint on one of the parameters and describes the behavior of the method if the constraint is satisfied. You wil see more examples soon.

18.6. The fundamental ambiguity theorem

One part of print_backward might have raised an eyebrow:

head = list

tail = list.next

After the first assignment, head and list have the same type and the same value. So why did we create a new variable?

The reason is that the two variables play different roles. We think of head as a reference to a single node, and we think of list as a reference to the first node of a list. These roles are not part of the program; they are in the mind of the programmer.

In general we can't tel by looking at a program what role a variable plays. This ambiguity can be useful, but it can also make programs difficult to read. We often use variable names like node and list to document how we intend to use a variable and sometimes create additional variables to disambiguate.

We could have written print_backward without head and tail, which makes it more concise but possibly less clear:

```
def print_backward(list) :
```

```
if list == None : return
```

```
print_backward(list.next)
```

```
print list,
```

Looking at the two function cal s, we have to remember that print_backward treats its argument as a col ection and print treats its argument as a single object.

The fundamental ambiguity theorem describes the ambiguity that is inherent in a reference to a node: A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.

18.7. Modifying lists

There are two ways to modify a linked list. Obviously, we can change the cargo of one of the nodes, but the more interesting operations are the ones that add, remove, or reorder the nodes.

As an example, let's write a method that removes the second node in the list and returns a reference to the removed node:

```
def removeSecond(list):
```

```
if list == None: return
```

```
first = list
```

```
second = list.next
```

```
# make the first node refer to the third
```

```
first.next = second.next
```

 # separate the second node from the rest of the list

second.next = None

return second

Again, we are using temporary variables to make the code more readable. Here is how to use this method:

>>> print_list(node1)

1 2 3

>>> removed = removeSecond(node1)

>>> print_list(removed)

2**>>>** print_list(node1)

1 3

This state diagram shows the effect of the operation:

What happens if you invoke this method and pass a list with only one element (a singleton)? What happens if you pass the empty list as an argument? Is there a precondition for this method? If so, fix the method to handle a violation of the precondition in a reasonable way.

18.8. Wrappers and helpers

It is often useful to divide a list operation into two methods. For example, to print a list backward in the conventional list format [3, 2, 1] we can use the print_backward method to print 3, 2, but we need a separate method to print the brackets and the first node. Let's cal it print_backward_nicely: **def** print_backward_nicely(list) :

print "[",

```python
if list != None :

head = list

tail = list.next

print_backward(tail)

print head,

print "]",
```

Again, it is a good idea to check methods like this to see if they work with special cases like an empty list or a singleton.

When we use this method elsewhere in the program, we invoke print_backward_nicely directly, and it invokes print_backward on our behalf. In that sense, print_backward_nicely acts as a wrapper, and it

uses print_backward as a helper.

18.9. The **LinkedList** class

There are some subtle problems with the way we have been implementing lists. In a reversal of cause and effect, we'l propose an alternative implementation first and then explain what problems it solves.

First, we'l create a new class cal ed LinkedList. Its attributes are an integer that contains the length of the list and a reference to the first node. LinkedList objects serve as handles for manipulating lists of Node objects:

```python
class LinkedList:

def __init__(self):
```

self.length = 0

self.head = None

One nice thing about the LinkedList class is that it provides a natural place to put wrapper functions like print_backward_nicely, which we can make a method of the LinkedList class:

class LinkedList:

...

def print_backward(self):

print "[",

if self.head != None:

self.head.print_backward()

print "]",

class Node:

...

def print_backward(self):

if self.next != None:

tail = self.next

tail.print_backward()

print self.cargo,

Just to make things confusing, we renamed print_backward_nicely. Now there are two methods named print_backward: one in the Node

class (the helper); and one in the LinkedList class (the wrapper). When the wrapper invokes self.head.print_backward, it is invoking the helper, because self.head is a Node object.

Another benefit of the LinkedList class is that it makes it easier to add or remove the first element of a list. For example, addFirst is a method for LinkedLists; it takes an item of cargo as an argument and puts it at the beginning of the list:

class LinkedList:

...

def addFirst(self, cargo):

node = Node(cargo)

node.next = self.head

self.head = node

self.length = self.length + 1

As usual, you should check code like this to see if it handles the special cases. For example, what happens if the list is initial y empty?

18.10. Invariants

Some lists are wel formed ; others are not. For example, if a list contains a loop, it wil cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the length value in the LinkedList object should be equal to the actual number of nodes in the list.

Requirements like these are cal ed invariants because, ideal y, they should be true of every object al the time. Specifying invariants for objects is a useful programming practice because it makes it easier

to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are times when they are violated.

For example, in the middle of addFirst, after we have added the node but before we have incremented length, the invariant is violated. This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while. Normal y, we require that every method that violates an invariant must restore the invariant.

If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.

18.11. Glossary

embedded reference

A reference stored in an attribute of an object.

linked list

A data structure that implements a col ection using a sequence of linked nodes.

node

An element of a list, usual y implemented as an object that contains a reference to another object of the same type.

cargo

An item of data contained in a node.

link

An embedded reference used to link one object to another.

precondition

An assertion that must be true in order for a method to work correctly.

fundamental ambiguity theorem

A reference to a list node can be treated as a single object or as the first in a list of nodes.

singleton

A linked list with a single node.

wrapper

A method that acts as a middleman between a cal er and a helper method, often making the method easier or less error-prone to invoke.

helper

A method that is not invoked directly by a cal er but is used by another method to perform part of an operation.

invariant

An assertion that should be true of an object at al times (except perhaps while the object is being modified).

18.12. Exercises

1. By convention, lists are often printed in brackets with commas between the elements, as in [1, 2, 3]. Modify print_list so that it generates output in this format.

19. Stacks

19.1. Abstract data types

The data types you have seen so far are al concrete, in the sense that we have completely specified how they are implemented. For example, the Card class represents a card using two integers. As we discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An abstract data type, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

1. It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.

2. Since there are usual y many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.

3. Wel -known ADTs, such as the Stack ADT in this chapter, are often implemented in standard libraries so they can be written once

and used by many programmers.

4. The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, cal ed the client code, from the code that implements the ADT, cal ed the provider code.

19.2. The Stack ADT

In this chapter, we wil look at one common ADT, the stack. A stack is a col ection, meaning that it is a data structure that contains multiple elements. Other col ections we have seen include dictionaries and lists.

An ADT is defined by the operations that can be performed on it, which is cal ed an interface. The interface for a stack consists of these operations:

__init__

Initialize a new empty stack.

push

Add a new item to the stack.

pop

Remove and return an item from the stack. The item that is returned is always the last one that was added.

is_empty

Check whether the stack is empty.

A stack is sometimes cal ed a last in, first out or LIFO data structure, because the last item added is the first to be removed.

19.3. Implementing stacks with Python lists

The list operations that Python provides are similar to the operations that define a stack. The interface isn't exactly what it is supposed to be, but we can write code to translate from the Stack ADT to the built-in operations.

This code is cal ed an implementation of the Stack ADT. In general, an implementation is a set of methods that satisfy the syntactic and semantic requirements of an interface.

Here is an implementation of the Stack ADT that uses a Python list:

class Stack :

def __init__(self):

self.items = []

def push(self, item):

self.items.append(item)

def pop(self):

return self.items.pop()

def is_empty(self):

return (self.items == [])

A Stack object contains an attribute named items that is a list of items in the stack. The initialization method sets items to the empty list.

To push a new item onto the stack, push appends it onto items. To pop an item off the stack, pop uses the homonymous (same-named) list method to remove and return the last item on the list.

Final y, to check if the stack is empty, is_empty compares items to the empty list.

An implementation like this, in which the methods consist of simple invocations of existing methods, is cal ed a veneer. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a smal piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

19.4. Pushing and popping

A stack is a generic data structure, which means that we can add any type of item to it. The fol owing example pushes two integers and a string onto the stack:

>>> s = Stack()

>>> s.push(54)

>>> s.push(45)

>>> s.push("+")

We can use is_empty and pop to remove and print al of the items on the stack:

while not s.is_empty():

print s.pop(),

The output is + 45 54. In other words, we just used a stack to print the items backward! Granted, it's not the standard format for printing

a list, but by using a stack, it was remarkably easy to do.

You should compare this bit of code to the implementation of print_backward in the last chapter. There is a natural paral el between the recursive version of print_backward and the stack algorithm here. The difference is that print_backward uses the runtime stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, except that is use a Stack object instead of the runtime stack.

19.5. Using a stack to evaluate postfix

In most programming languages, mathematical expressions are written with the operator between the two operands, as in 1 + 2. This format is cal ed infix. An alternative used by some calculators is cal ed postfix. In postfix, the operator fol ows the operands, as in 1 2 +.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack:

1. Starting at the beginning of the expression, get one term (operator or operand) at a time.

If the term is an operand, push it on the stack.

If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.

2. When you get to the end of the expression, there should be exactly one operand left on the stack.

That operand is the result.

19.6. Parsing

To implement the previous algorithm, we need to be able to traverse a string and break it into operands and operators. This process is an example of parsing, and the results—the individual chunks of the string – are cal ed tokens. You might remember these words from Chapter 1.

Python provides a split method in both the string and re (regular expression) modules. The function string.split splits a string into a list using a single character as a delimiter. For example:

>>> import string

>>> string.split("Now is the time"," ")

['Now', 'is', 'the', 'time']

In this case, the delimiter is the space character, so the string is split at each space.

The function re.split is more powerful, al owing us to provide a regular expression instead of a delimiter. A regular expression is a way of specifying a set of strings. For example, [A-z] is the set of al letters and [0-9] is the set of al numbers. The ^ operator negates a set, so [^0-9] is the set of everything that is not a number, which is exactly the set we want to use to split up postfix expressions:

>>> import re

>>> re.split("([^0-9])", "123+456*/")

['123', '+', '456', '*', '', '/', '']

Notice that the order of the arguments is different from string.split; the delimiter comes before the string.

The resulting list includes the operands 123 and 456 and the operators * and /. It also includes two empty strings that are inserted

after the operands.

19.7. Evaluating postfix

To evaluate a postfix expression, we wil use the parser from the previous section and the algorithm from the section before that. To keep things simple, we'l start with an evaluator that only implements the operators + and *:

```
def eval_postfix(expr):

import re

token_list = re.split("([^0-9])", expr)

stack = Stack()

for token in token_list:

if token == '' or token == ' ':

continue

if token == '+':

sum = stack.pop() + stack.pop()

stack.push(sum)

elif token == '*':

product = stack.pop() * stack.pop()

stack.push(product)

else:

stack.push(int(token))
```

return stack.pop()

The first condition takes care of spaces and empty strings. The next two conditions handle operators.

We assume, for now, that anything else must be an operand. Of course, it would be better to check for erroneous input and report an error message, but we'l get to that later.

Let's test it by evaluating the postfix form of (56+47)*2:

>>> print eval_postfix ("56 47 + 2 *")

206

That's close enough.

19.8. Clients and providers

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct – in accord with the specification of the ADT – and not how it wil be used.

Conversely, the client assumes that the implementation of the ADT is correct and doesn't worry about the details. When you are using one of Python's built-in types, you have the luxury of thinking exclusively as a client.

Of course, when you implement an ADT, you also have to write client code to test it. In that case, you play both roles, which can be confusing. You should make some effort to keep track of which role you are playing at any moment.

19.9. Glossary

abstract data type (ADT)

A data type (usual y a col ection of objects) that is defined by a set of operations but that can be implemented in a variety of ways.

interface

The set of operations that define an ADT.

implementation

Code that satisfies the syntactic and semantic requirements of an interface.

client

A program (or the person who wrote it) that uses an ADT.

provider

The code (or the person who wrote it) that implements an ADT.

veneer

A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it

improves or standardizes the interface seen by the client.

generic data structure

A kind of data structure that can contain data of any type.

infix

A way of writing mathematical expressions with the operators between the operands.

postfix

A way of writing mathematical expressions with the operators after the operands.

parse

To read a string of characters or tokens and analyze its grammatical structure.

token

A set of characters that are treated as a unit for purposes of parsing, such as the words in a natural language.

delimiter

A character that is used to separate tokens, such as punctuation in a natural language.

19.10. Exercises

1. Apply the postfix algorithm to the expression 1 2 + 3 *.This example demonstrates one of the advantages of postfix—there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write (1 + 2) * 3.

2. Write a postfix expression that is equivalent to 1 + 2 * 3.

20. Queues

This chapter presents two ADTs: the Queue and the Priority Queue. In real life, a queue is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served. There are exceptions, though. At airports, customers whose flights are leaving soon are sometimes taken from the middle of the queue. At supermarkets, a polite customer might let someone with only a few items go first.

The rule that determines who goes next is cal ed the queueing policy. The simplest queueing policy is cal ed FIFO, for first- in-first-out. The most general queueing policy is priority queueing, in which each customer is assigned a priority and the customer with the highest priority goes first, regardless of the order of arrival. We say this is the most general policy because the priority can be based on anything: what time a flight leaves; how many groceries the customer has; or how important the customer is. Of course, not al queueing policies are fair, but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations. The difference is in the semantics of the operations: a queue uses the FIFO policy; and a priority queue (as the name suggests) uses the priority queueing policy.

20.1. The Queue ADT

The Queue ADT is defined by the fol owing operations:

__init__

Initialize a new empty queue.

insert

Add a new item to the queue.

remove

Remove and return an item from the queue. The item that is returned is the first one that was added.

is_empty

Check whether the queue is empty.

20.2. Linked Queue

The first implementation of the Queue ADT we wil look at is cal ed a linked queue because it is made up of linked Node objects. Here is the class definition:

class Queue:

def __init__(self):

self.length = 0

self.head = None

def is_empty(self):

return (self.length == 0)

def insert(self, cargo):

node = Node(cargo)

node.next = None

if self.head == None:

if list is empty the new node goes first

self.head = node

else:

 # find the last node in the list

last = self.head

while last.next: last = last.next

 # append the new node

last.next = node

self.length = self.length + 1

def remove(self):

cargo = self.head.cargo

self.head = self.head.next

self.length = self.length - 1

return cargo

The methods is_empty and remove are identical to the LinkedList methods is_empty and remove_first.

The insert method is new and a bit more complicated.

We want to insert new items at the end of the list. If the queue is empty, we just set head to refer to the new node.

Otherwise, we traverse the list to the last node and tack the new node on the end. We can identify the last node because its next attribute is None.

There are two invariants for a properly formed Queue object. The value of length should be the number of nodes in the queue, and the last node should have next equal to None. Convince yourself that this method preserves both invariants.

20.3. Performance characteristics

Normal y when we invoke a method, we are not concerned with the details of its implementation. But there is one detail we might want to know—the performance characteristics of the method. How long does it take, and how does the run time change as the number of items in the col ection increases?

First look at remove. There are no loops or function cal s here, suggesting that the runtime of this method is the same every time. Such a method is cal ed a constant-time operation. In reality, the method might be slightly faster when the list is empty since it skips the body of the conditional, but that difference is not significant.

The performance of insert is very different. In the general case, we have to traverse the list to find the last element.

This traversal takes time proportional to the length of the list. Since the runtime is a linear function of the length, this method is cal ed linear time. Compared to constant time, that's very bad.

20.4. Improved Linked Queue

We would like an implementation of the Queue ADT that can perform al operations in constant time.

One way to do that is to modify the Queue class so that it maintains a reference to both the first and the last node, as shown in the figure:

The ImprovedQueue implementation looks like this:

class ImprovedQueue:

def __init__(self):

self.length = 0

self.head = None

```python
        self.last = None

    def is_empty(self):

        return (self.length == 0)
```

So far, the only change is the attribute last. It is used in insert and remove methods:

```python
class ImprovedQueue:

    ...

    def insert(self, cargo):

        node = Node(cargo)

        node.next = None

        if self.length == 0:

            # if list is empty, the new node is head and last

            self.head = self.last = node

        else:

            # find the last node

            last = self.last

            # append the new node

            last.next = node

            self.last = node

        self.length = self.length + 1
```

Since last keeps track of the last node, we don't have to search for it. As a result, this method is constant time.

There is a price to pay for that speed. We have to add a special case to remove to set last to None when the last node is removed:

class ImprovedQueue:

...

def remove(self):

cargo = self.head.cargo

self.head = self.head.next

self.length = self.length - 1

if self.length == 0:

self.last = None

return cargo

This implementation is more complicated than the Linked Queue implementation, and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal – both insert and remove are constant-time operations.

20.5. Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. Again, the interface is:

__init__

Initialize a new empty queue.

insert

Add a new item to the queue.

remove

Remove and return an item from the queue. The item that is returned is the one with the highest priority.

is_empty

Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is the item in the queue that has the highest priority. What the priorities are and how they compare to each other are not specified by the Priority Queue implementation. It depends on which items are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might go from highest to lowest, but if they are golf scores, we would go from

lowest to highest. As long as we can compare the items in the queue, we can find and remove the one with the highest priority.

This implementation of Priority Queue has as an attribute a Python list that contains the items in the queue.

class PriorityQueue:

def __init__(self):

self.items = []

def is_empty(self):

return self.items == []

```
def insert(self, item):

    self.items.append(item)
```

The initialization method, is_empty, and insert are al veneers on list operations. The only interesting method is remove:

```
class PriorityQueue:

...

def remove(self):

    maxi = 0

    for i in range(1, len(self.items)):

        if self.items[i] > self.items[maxi]: maxi = i

    item = self.items[maxi]

    self.items[maxi:maxi+1] = []

    return item
```

At the beginning of each iteration, maxi holds the index of the biggest item (highest priority) we have seen so far. Each time through the loop, the program compares the i-eth item to the champion. If the new item is bigger, the value of maxi if set to i.

When the for statement completes, maxi is the index of the biggest item. This item is removed from the list and returned.

Let's test the implementation:

```
>>> q = PriorityQueue()

>>> q.insert(11)
```

>>> q.insert(12)

>>> q.insert(14)

>>> q.insert(13)

>>> while not q.is_empty(): **print** q.remove()

14

13

12

11

If the queue contains simple numbers or strings, they are removed in numerical or alphabetical order, from highest to lowest. Python can find the biggest integer or string because it can compare them using the built-in comparison operators.

If the queue contains an object type, it has to provide a __cmp__ method. When remove uses the > operator to compare items, it invokes the __cmp__ for one of the items and passes the other as a parameter. As long as the __cmp__ method works correctly, the Priority Queue wil work.

20.6. The **Golfer** class

As an example of an object with an unusual definition of priority, let's implement a class cal ed Golfer that keeps track of the names and scores of golfers. As usual, we start by defining __init__ and

__str__:

class Golfer:

def __init__(self, name, score):

```
self.name = name

self.score= score

def __str__(self):

return " %-16s: %d" % (self.name, self.score)
```

__str__ uses the format operator to put the names and scores in neat columns.

Next we define a version of __cmp__ where the lowest score gets highest priority. As always, __cmp__

returns 1 if self is greater than other, -1 if self is less than other, and 0 if they are equal.

```
class Golfer:

...

def __cmp__(self, other):

if self.score < other.score: return 1 # less is more

if self.score > other.score: return -1

return 0
```

Now we are ready to test the priority queue with the Golfer class:

```
>>> tiger = Golfer("Tiger Woods", 61)

>>> phil = Golfer("Phil Mickelson", 72)

>>> hal = Golfer("Hal Sutton", 69)

>>>
```

```
>>> pq = PriorityQueue()

>>> pq.insert(tiger)

>>> pq.insert(phil)

>>> pq.insert(hal)

>>> while not pq.is_empty(): print pq.remove()
```

Tiger Woods : 61

Hal Sutton : 69

Phil Mickelson : 72

20.7. Glossary

queue

An ordered set of objects waiting for a service of some kind.

Queue

An ADT that performs the operations one might perform on a queue.

queueing policy

The rules that determine which member of a queue is removed next.

FIFO

First In, First Out, a queueing policy in which the first member to arrive is the first to be removed.

priority queue

A queueing policy in which each member has a priority determined by external factors. The member with the highest priority is the first

to be removed.

Priority Queue

An ADT that defines the operations one might perform on a priority queue.

linked queue

An implementation of a queue using a linked list.

constant time

An operation whose runtime does not depend on the size of the data structure.

linear time

An operation whose runtime is a linear function of the size of the data structure.

20.8. Exercises

1. Write an implementation of the Queue ADT using a Python list. Compare the performance of this implementation to the ImprovedQueue for a range of queue lengths. #. Write an implementation of the Priority Queue ADT using a linked list. You should keep the list sorted so that removal is a constant time operation. Compare the performance of this implementation with the Python list implementation.

21. Trees

Like linked lists, trees are made up of nodes. A common kind of tree is a binary tree, in which each node contains a reference to two other nodes (possibly None). These references are referred to as the left and right subtrees. Like list nodes, tree nodes also contain cargo. A state diagram for a tree looks like this: To avoid cluttering up the picture, we often omit the Nones.

The top of the tree (the node tree refers to) is cal ed the root. In keeping with the tree metaphor, the other nodes are cal ed branches and the nodes at the tips with nul references are cal ed leaves. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in another metaphor – the family tree. The top node is sometimes cal ed a parent and the nodes it refers to are its children. Nodes with the same parent are cal ed siblings.

Final y, there is a geometric vocabulary for talking about trees. We already mentioned left and right, but there is also up (toward the parent/root) and down (toward the children/leaves). Also, al of the nodes that are the same distance from the root comprise a level of the tree.

We probably don't need three metaphors for talking about trees, but there they are.

Like linked lists, trees are recursive data structures because they are defined recursively. A tree is either: 1. the empty tree, represented by None, or

2. a node that contains an object reference (cargo) and two tree references.

21.1. Building trees

The process of assembling a tree is similar to the process of assembling a linked list. Each constructor invocation builds a single node.

class Tree:

def __init__(self, cargo, left=None, right=None):

self.cargo = cargo

self.left = left

self.right = right

def __str__(self):

return str(self.cargo)

The cargo can be any type, but the left and right parameters should be tree nodes. left and right are optional; the default value is None.

To print a node, we just print the cargo.

One way to build a tree is from the bottom up. Al ocate the child nodes first:

left = Tree(2)

right = Tree(3)

Then create the parent node and link it to the children:

tree = Tree(1, left, right);

We can write this code more concisely by nesting constructor invocations:

>>> tree = Tree(1, Tree(2), Tree(3))

Either way, the result is the tree at the beginning of the chapter.

21.2. Traversing trees

Any time you see a new data structure, your first question should be, How do I traverse it? The most natural way to traverse a tree is recursively. For example, if the tree contains integers as cargo, this function returns their sum:

def total(tree):

if tree == None: **return** 0

return total(tree.left) + total(tree.right) + tree.cargo

The base case is the empty tree, which contains no cargo, so the sum is 0. The recursive step makes two recursive cal s to find the sum of the child trees. When the recursive cal s complete, we add the cargo of the parent and return the total.

21.3. Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the computation unambiguously. For example, the infix expression 1 + 2 * 3 is ambiguous unless we know that the multiplication happens before the addition.

This expression tree represents the same computation:

The nodes of an expression tree can be operands like 1 and 2 or operators like + and *. Operands are leaf nodes; operator nodes contain references to their operands. (Al of these operators are binary, meaning they have exactly two operands.)

We can build this tree like this:

>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))

Looking at the figure, there is no question what the order of operations is; the multiplication happens first in order to compute the second operand of the addition.

Expression trees have many uses. The example in this chapter uses trees to translate expressions to postfix, prefix, and infix. Similar trees are used inside compilers to parse, optimize, and translate programs.

21.4. Tree traversal

We can traverse an expression tree and print the contents like this:

def print_tree(tree):

if tree == None: **return**

print tree.cargo,

print_tree(tree.left)

print_tree(tree.right)

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, and then print the entire right subtree. This way of traversing a tree is cal ed a preorder, because the contents of the root appear before the contents of the children. For the previous example, the output is:

>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))

>>> print_tree(tree)

+ 1 * 2 3

This format is different from both postfix and infix; it is another notation cal ed prefix, in which the operators appear before their operands.

You might suspect that if you traverse the tree in a different order, you wil get the expression in a different notation. For example, if you print the subtrees first and then the root node, you get: **def** print_tree_postorder(tree):

if tree == None: **return**

print_tree_postorder(tree.left)

print_tree_postorder(tree.right)

print tree.cargo,

The result, 1 2 3 * +, is in postfix! This order of traversal is cal ed postorder.

Final y, to traverse a tree inorder, you print the left tree, then the root, and then the right tree: **def** print_tree_inorder(tree):

if tree == None: **return**

print_tree_inorder(tree.left)

print tree.cargo,

print_tree_inorder(tree.right)

The result is 1 + 2 * 3, which is the expression in infix.

To be fair, we should point out that we have omitted an important complication. Sometimes when we write an expression in infix, we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

If we do an inorder traversal and keep track of what level in the tree we are on, we can generate a graphical representation of a tree:

```
def print_tree_indented(tree, level=0):

    if tree == None: return

    print_tree_indented(tree.right, level+1)

    print ' ' * level + str(tree.cargo)

    print_tree_indented(tree.left, level+1)
```

The parameter level keeps track of where we are in the tree. By default, it is initial y 0. Each time we make a recursive cal , we pass level+1 because the child's level is always one greater than the parent's.

Each item is indented by two spaces per level. The result for the example tree is:

```
>>> print_tree_indented(tree)

3

  *

2

+ 1
```

If you look at the output sideways, you see a simplified version of the original figure.

21.5. Building an expression tree

In this section, we parse infix expressions and build the corresponding expression trees. For example, the expression (3 + 7) * 9 yields the fol owing tree:

Notice that we have simplified the diagram by leaving out the names of the attributes.

The parser we wil write handles expressions that include numbers, parentheses, and the operators +

and *. We assume that the input string has already been tokenized into a Python list (producing this list is left as an exercise). The token list for (3 + 7) * 9 is:

['(', 3, '+', 7, ')', '*', 9, 'end']

The end token is useful for preventing the parser from reading past the end of the list.

The first function we'l write is get_token, which takes a token list and an expected token as parameters.

It compares the expected token to the first token on the list: if they match, it removes the token from the list and returns True; otherwise, it returns False:

```
def get_token(token_list, expected):

    if token_list[0] == expected:

        del token_list[0]

        return True

    else:
```

return False

Since token_list refers to a mutable object, the changes made here are visible to any other variable that refers to the same object.

The next function, get_number, handles operands. If the next token in token_list is a number, get_number removes it and returns a leaf node containing the number; otherwise, it returns None.

def get_number(token_list):

x = token_list[0]

if type(x) != type(0): **return** None

del token_list[0]

return Tree (x, None, None)

Before continuing, we should test get_number in isolation. We assign a list of numbers to token_list, extract the first, print the result, and print what remains of the token list:

>>> token_list = [9, 11, 'end']

>>> x = get_number(token_list)

>>> print_tree_postorder(x)

9**>>> print** token_list

[11, 'end']

The next method we need is get_product, which builds an expression tree for products. A simple product has two numbers as operands, like 3 * 7.

Here is a version of get_product that handles simple products.

```python
def get_product(token_list):

    a = get_number(token_list)

    if get_token(token_list, '*'):

        b = get_number(token_list)

        return Tree ('*', a, b)

    else:

        return a
```

Assuming that get_number succeeds and returns a singleton tree, we assign the first operand to a. If the next character is *, we get the second number and build an expression tree with a, b, and the operator.

If the next character is anything else, then we just return the leaf node with a. Here are two examples:

```
>>> token_list = [9, '*', 11, 'end']

>>> tree = get_product(token_list)

>>> print_tree_postorder(tree)

9 11 *

>>> token_list = [9, '+', 11, 'end']

>>> tree = get_product(token_list)

>>> print_tree_postorder(tree)

9
```

The second example implies that we consider a single operand to be a kind of product. This definition of product is counterintuitive, but it turns out to be useful.

Now we have to deal with compound products, like like 3 * 5 * 13. We treat this expression as a product of products, namely 3 * (5 * 13). The resulting tree is:

With a smal change in get_product, we can handle an arbitrarily long product:

def get_product(token_list):

a = get_number(token_list)

if get_token(token_list, '*'):

b = get_product(token_list) *# this line changed*

return Tree ('*', a, b)

else:

return a

In other words, a product can be either a singleton or a tree with * at the root, a number on the left, and a product on the right. This kind of recursive definition should be starting to feel familiar.

Let's test the new version with a compound product:

>>> token_list = [2, '*', 3, '*', 5 , '*', 7, 'end']

>>> tree = get_product(token_list)

>>> print_tree_postorder(tree)

2 3 5 7 * * *

Next we wil add the ability to parse sums. Again, we use a slightly counterintuitive definition of sum.

For us, a sum can be a tree with + at the root, a product on the left, and a sum on the right. Or, a sum can be just a product.

If you are wil ing to play along with this definition, it has a nice property: we can represent any expression (without parentheses) as a sum of products. This property is the basis of our parsing algorithm.

get_sum tries to build a tree with a product on the left and a sum on the right. But if it doesn't find a +, it just builds a product.

```
def get_sum(token_list):

a = get_product(token_list)

if get_token(token_list, '+'):

b = get_sum(token_list)

return Tree ('+', a, b)

else:

return a
```

Let's test it with 9 * 11 + 5 * 7:

```
>>> token_list = [9, '*', 11, '+', 5, '*', 7, 'end']

>>> tree = get_sum(token_list)

>>> print_tree_postorder(tree)

9 11 * 5 7 * +
```

We are almost done, but we stil have to handle parentheses. Anywhere in an expression where there can be a number, there can also be an entire sum enclosed in parentheses. We just need to modify get_number to handle subexpressions:

def get_number(token_list):

if get_token(token_list, '('):

x = get_sum(token_list) *# get the subexpression*

get_token(token_list, ')') *# remove the closing parenthesis*

return x

else:

x = token_list[0]

if type(x) != type(0): **return** None

token_list[0:1] = []

return Tree (x, None, None)

Let's test this code with 9 * (11 + 5) * 7:

>>> token_list = [9, '*', '(', 11, '+', 5, ')', '*', 7, 'end']

>>> tree = get_sum(token_list)

>>> print_tree_postorder(tree)

9 11 5 + 7 * *

The parser handled the parentheses correctly; the addition happens before the multiplication.

In the final version of the program, it would be a good idea to give get_number a name more descriptive of its new role.

21.6. Handling errors

Throughout the parser, we've been assuming that expressions are wel -formed. For example, when we reach the end of a subexpression, we assume that the next character is a close parenthesis. If there is an error and the next character is something else, we should deal with it.

def get_number(token_list):

if get_token(token_list, '('):

x = get_sum(token_list)

if not get_token(token_list, ')'):

raise 'BadExpressionError', 'missing parenthesis'

return x

else:

the rest of the function omitted

The raise statement creates an exception; in this case we create a new kind of exception, cal ed a BadExpressionError. If the function that cal ed get_number, or one of the other functions in the traceback, handles the exception, then the program can continue. Otherwise, Python wil print an error message and quit.

21.7. The animal tree

In this section, we develop a smal program that uses a tree to represent a knowledge base.

The program interacts with the user to create a tree of questions and animal names. Here is a sample run:

Are you thinking of an animal? y

Is it a bird? n

What is the animals name? dog

What question would distinguish a dog from a bird? Can it fly

If the animal were dog the answer would be? n

Are you thinking of an animal? y

Can it fly? n

Is it a dog? n

What is the animals name? cat

What question would distinguish a cat from a dog? Does it bark

If the animal were cat the answer would be? n

Are you thinking of an animal? y

Can it fly? n

Does it bark? y

Is it a dog? y

I rule!

Are you thinking of an animal? n

Here is the tree this dialog builds:

At the beginning of each round, the program starts at the top of the tree and asks the first question.

Depending on the answer, it moves to the left or right child and continues until it gets to a leaf node. At that point, it makes a guess. If the guess is not correct, it asks the user for the name of the new animal and a question that distinguishes the (bad) guess from the new animal. Then it adds a node to the tree with the new question and the new animal.

Here is the code:

```
def yes(ques):

ans = raw_input(ques).lower()

return ans[0] == 'y'

def animal():

# start with a singleton

root = Tree("bird")

# loop until the user quits

while True:

print

if not yes("Are you thinking of an animal? "): break

# walk the tree

tree = root

while tree.left != None:
```

```python
    prompt = tree.cargo + "? "
    if yes(prompt):
        tree = tree.right
    else:
        tree = tree.left

    # make a guess
    guess = tree.cargo
    prompt = "Is it a " + guess + "? "
    if yes(prompt):
        print "I rule!"
        continue

    # get new information
    prompt = "What is the animal's name? "
    animal = raw_input(prompt)
    prompt = "What question would distinguish a %s from a %s? "
    question = raw_input(prompt % (animal, guess))

    # add new information to the tree
    tree.cargo = question
    prompt = "If the animal were %s the answer would be? "
    if yes(prompt % animal):
```

```
tree.left = Tree(guess)

tree.right = Tree(animal)
```

else:

```
tree.left = Tree(animal)

tree.right = Tree(guess)
```

The function yes is a helper; it prints a prompt and then takes input from the user. If the response begins with y or Y, the function returns True.

The condition of the outer loop of animal is True, which means it wil continue until the break statement executes, if the user is not thinking of an animal.

The inner while loop walks the tree from top to bottom, guided by the user's responses.

When a new node is added to the tree, the new question replaces the cargo, and the two children are the new animal and the original cargo.

One shortcoming of the program is that when it exits, it forgets everything you careful y taught it! Fixing this problem is left as an exercise.

21.8. Glossary

binary tree

A tree in which each node refers to zero, one, or two dependent nodes.

root

The topmost node in a tree, with no parent.

leaf

A bottom-most node in a tree, with no children.

parent

The node that refers to a given node.

child

One of the nodes referred to by a node.

siblings

Nodes that share a common parent.

level

The set of nodes equidistant from the root.

binary operator

An operator that takes two operands.

subexpression

An expression in parentheses that acts as a single operand in a larger expression.

preorder

A way to traverse a tree, visiting each node before its children.

prefix notation

A way of writing a mathematical expression with each operator appearing before its operands.

postorder

A way to traverse a tree, visiting the children of each node before the node itself.

inorder

A way to traverse a tree, visiting the left subtree, then the root, and then the right subtree.

21.9. Exercises

1. Modify print_tree_inorder so that it puts parentheses around every operator and pair of operands. Is the output correct and unambiguous? Are the parentheses always necessary?

2. Write a function that takes an expression string and returns a token list.

3. Find other places in the expression tree functions where errors can occur and add appropriate raise statements. Test your code with improperly formed expressions.

4. Think of various ways you might save the animal knowledge tree in a file. Implement the one you think is easiest.

Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

1. Syntax errors are produced by Python when it is translating the source code into byte code. They usual y indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a def statement yields the somewhat redundant message SyntaxError: invalid syntax.

2. Runtime errors are produced by the runtime system if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventual y causes a runtime error of maximum recursion depth exceeded.

3. Semantic errors are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the fol owing sections are organized by error type, some techniques are applicable in more than one situation.

Syntax errors

Syntax errors are usual y easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.

On the other hand, the message does tel you where in the program the problem occurred. Actual y, it tel s you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incremental y, you should have a good idea about where the error is. It wil be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very careful y.

Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.

2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.

3. Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.

4. Make sure that any strings in the code have matching quotation marks.

5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the fol owing part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at al !

6. An unclosed bracket – (, {, or [– makes Python continue with the next line as part of the current statement. General y, an error occurs almost immediately in the next line.

7. Check for the classic = instead of == inside a conditional.

If nothing works, move on to the next section...

I can't get my program to run no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run. If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run (or import) it again. If the compiler doesn't find the new error, there is probably something wrong with the way your environment is set up.

If this happens, one approach is to start again with a new program like Hel o, World!, and make sure you can get a known program to run. Then gradual y add the pieces of the new program to the working one.

Runtime errors

Once your program is syntactical y correct, Python can import it and at least start running it. What could possibly go wrong?

My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actual y invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the Flow of Execution section below.

My program hangs.

If a program stops and seems to be doing nothing, we say it is hanging. Often that means that it is caught in an infinite loop or an infinite recursion.

1. If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says entering the loop and another immediately after that says exiting the loop.

2. Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the Infinite Loop section below.

3. Most of the time, an infinite recursion wil cause the program to run for a while and then produce a RuntimeError: Maximum recursion depth exceeded error. If that happens, go to the Infinite Recursion section below.

4. If you are not getting this error but you suspect there is a problem with a recursive method or function, you can stil use the techniques in the Infinite Recursion section.

5. If neither of those steps works, start testing other loops and other recursive functions and methods.

6. If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the Flow of Execution section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

while x > 0 **and** y < 0:

 # do something to x

 # do something to y

print "x: ", x

print "y: ", y

print "condition: ", (x > 0 **and** y < 0)

Now when you run the program, you wil see three lines of output for each time through the loop. The last time through the loop, the condition should be false. If the loop keeps going, you wil be able to see the values of x and y, and you might figure out why they are not being updated correctly.

Infinite Recursion

Most of the time, an infinite recursion wil cause the program to run for a while and then produce a Maximum recursion depth exceeded error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that wil cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a print statement at the beginning of the function or method that prints the parameters. Now when you run the program, you wil see a few lines of output every time the function or method is invoked, and you wil see the parameters. If the parameters are not moving toward the base case, you wil get some ideas about why not.

Flow of Execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each function with a message like entering function foo, where foo is the name of the function.

Now when you run the program, it wil print a trace of each function as it is invoked.

When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked that, and so on. In other words, it traces the path of function invocations that got you to where you are. It also includes the line number in your file where each of these cal s occurs.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError

You are trying to use a variable that doesn't exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.

TypeError

There are several possible causes:

1. You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.

2. There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is cal ed for.

3. You are passing the wrong number of arguments to a function or method. For methods, look at the method definition and check that the first parameter is self. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

KeyError

You are trying to access an element of a dictionary using a key value that the dictionary does not contain.

AttributeError

You are trying to access an attribute or method that does not exist.

IndexError

The index you are using to access a list, string, or tuple is greater than its length minus one.

Immediately before the site of the error, add a print statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

I added so many **print** statements I get inundated with output.

One of the problems with using print statements for debugging is that you can end up buried in output.

There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output

so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting an array, sort a small array. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smal er functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

Semantic errors

In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actual y doing. One of the things that makes that hard is that computers run so fast.

You wil often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few wel -placed print statements is often short compared to

setting up the debugger, inserting and removing breakpoints, and walking the program to where the error is occurring.

My program doesn't work.

You should ask yourself these questions:

1. Is there something the program was supposed to do but which doesn't seem to be happening?

Find the section of the code that performs that function and make sure it is executing when you think it should.

2. Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.

3. Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especial y if it involves invocations to functions or methods in other Python modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usual y the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a smal amount of new code that is not known to be correct.

I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

self.hands[i].addCard (self.hands[self.findNeighbor(i)].popCard())

This can be rewritten as:

neighbor = self.findNeighbor (i)

pickedCard = self.hands[neighbor].popCard()

self.hands[i].addCard (pickedCard)

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression x/2pi into Python, you might write: y = x / 2 * math.pi;

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes (x/2)pi.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit: y = x / (2 * math.pi);

Whenever you are not sure of the order of evaluation, use parentheses. Not only wil the program be correct (in the sense of

doing what you intended), it wil also be more readable for other people who haven't memorized the rules of precedence.

I've got a function or method that doesn't return what I expect.

If you have a return statement with a complex expression, you don't have a chance to print the return value before returning. Again, you can use a temporary variable. For example, instead of: **return** self.hands[i].removeMatches()

you could write:

count = self.hands[i].removeMatches()

return count

Now you have the opportunity to display the value of count before returning.

I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these effects:

1. Frustration and/or rage.

2. Superstitious beliefs (the computer hates me) and magical thinking (the program only works when I wear my hat backward).

3. Random-walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. We often find bugs when we are away from the computer and let our minds wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fal asleep.

No, I really need help.

It happens. Even the best programmers occasional y get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you have exhausted the techniques described here. Your program should be as simple as possible, and you should be working on the smal est input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem wel enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

1. If there is an error message, what is it and what part of the program does it indicate?

2. What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?

3. What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you wil be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

Graphics API for Students of Python: GASP

Introduction

Describe gasp here...

Coordinates

(0, 0) is at the bottom left of the window. The window is 640 pixels by 480, by default. (You can make it a different size if you want to.) Coordinates are given in units of one pixel.

Al functions that take coordinates take them as a tuple (x, y).

Circle((300, 200), 10) # *:) This is good*

Circle(300, 200, 10) # *:(This is bad*

Colors

To access the color module GASP has to offer. Cal color.* where * is the color you wish to cal . For example: ` color.BLACK ` This is the color black. Check out the gasp color refrence chart to see al of the availble color options.

The Essentials

from gasp import *

begin_graphics()

... # all of your code

end_graphics()

These are the essentials. ` from gasp import * `imports the gasp module, begin_graphics() starts the graphics window, and end_graphics() quits the graphics window and ends the program. It's dead simple, but also dead necessary.

Graphics Functions

begin_graphics()

begin_graphics(width=800, height=600, title="My Game", background=color.YELLOW) This creates a graphics window with the dimensions 800x600, a title of My Game , and a background color of yel ow. With no arguments you get a white 640x480 graphics window titled Gasp .

width

The width of the window in pixels.

height

The windows height in pixels.

title

A string that wil be the title of the window.

background

It is the background of the graphics window. It can either be a color or an image end_graphics()

endgraphics()

Ends a graphics window.

clear_screen()

clear_screen()

Clears everything off of the graphics window. It looks like a new graphcs window as if you just cal ed begin_graphics().

remove_from_screen()

remove_from_screen(obj)

removes those objects from the screen

obj

A screen object of a list of screen_objects you would like to remove from the screen Screen Objects

The objects that you wil be displayed in your graphics window. You can manipulate these objects using the screen object methods

Plot

Plot(pos, color=color.BLACK, size=1)

It puts a dot on the screen.

pos

The coordinate on the screen that you wish to plot.

color

The color you wish the dot to be.

size

An integer that determinse the size the of the dot

Line

Line(start, end, color=color.BLACK)

Creates a line on the screen.

start

The starting coordinate of the line.

end

The coordinate at which the line wil end.

color

The color of the line

Box

Box(center, width, height, filled=False, color=color.BLACK, thickness=1)

This creates a Box on the screen

center

A coorinate where the center of your box wil be.

width

The width in pixels of the box.

height

The height of the box in pixels.

fil ed

A boolean value that determines if your box wil be fil ed

color

The color of your box.

thickness

The thickness in pixels of your box's lines.

Polygon

Polygon(points, filled=False, color=color.BLACK, thickness=1)

Creates a polygon on the screen

points

A list of coorinates that is each point on the polygon. The must be more than two items in the list fil ed

A boolean value. If it is False the polygon wil not be fil ed. Else, the polygon wil not be fil ed color

The color of the polygon's lines

thickness

An integer that determines the thickness of the lines.

Circle

Circle(center, radius, filled=False, color=color.BLACK, thickness=1)

Draws a circle, its center is a set of coordinates, and the radius is in pixels. It defaults to not being fil ed and the color black.

center

The circle's center coordinate.

width

An integer that is the radius of the circle

fil ed

A boolean value that determines if your circle wil be fil ed

color

The color of your circle.

thickness

The thickness in pixels of the circles lines.

Arc

Arc(center, radius, start_angle, end_angle, filled=False, color=color.BLACK, thickness=1) Creates an arc on the screen.

center

A coordinate that is the center of the arc.

radius

An integer that is the distance between the center and the outer edge of the arc.

start_angle

The start angle in degrees of the arc

end_angle

The end angle in degrees of your arc

fil ed

A boolean value that if True it fil s the arc

color

The color the arc

thickness

The thickness in pixels of the arc

Oval

Oval(center, width, height, filled=False, color=color.BLACK, thickness=1)

Puts an oval on the screen wherever you want.

center

The center coordinate of the Oval

width

The width in pixels of the oval

height

The height of the oval in pixels

fil ed

A boolean value determining if the oval wil be fil es or not.

color

The oval's color

thickness

The thickness of the ovals lines

Image

Image(file_path, center, width=None, height=None):

Loads an image onto the screen. If you only pass a width and not a height it automatical y scales the height to fit the width you passed it. It behaves likewise when you pass just a height.

file_path

The path to the image

center

The center coordinates of the image

width

The width of the image in pixels. If width equals None then it defaults to the image file's width height

The height of the image in pixels. If no height is passed it defaults to the image file's height Screen Object Methods

The methods that manipulates screen objects

move_to()

move_to(obj, pos)

Move a screen object to a pos

obj

A screen object you wish to move.

pos

The coordinate on the screen that the object wil move to

move_by()

move_by(obj, dx, dy)

Move a screen object relative to it's position

obj

The screen object you wish to move

dx

How much the object wil move in the 'x' direction. Positive or negative.

dy

How much the object wil move in the 'y' direction. A pixel value.

rotate_to()

rotate_to(obj, angle)

Rotate an object to an angle

obj

The screen object that wil be rotated

angle

The angle in degrees that the object wil be rotated to

rotate_by()

rotate_by(obj, angle)

Rotate an object a certain degree.

obj

The screen object you wish to rotate

angle

The degree that the object wil be rotate. Can be positive or negative.

Text

Text()

Text(text, pos, color=color.BLACK, size=12)

Puts text on the screen

text

A string of the text that wil be displayed

pos

The center coordinate of the text

color

The color of the text

size

The font size

Mouse

mouse_position()

mouse_position()

Returns the current mouse coordinate

mouse_buttons()

mouse_buttons()

returns a dictionary of the buttons state. There is a 'left', 'middle', and 'right' key.

Keyboard

keys_pressed()

keys_pressed()

returns a list of al of the keys pressed at that moment.

Gasp Tools

screen_shot

screen_shot(filename)

Saves a screenshot of the current graphics screen to a png file.

filename

The file path relative to the current directory that the image wil be written to.

Index

Symbols

"hel o world!"

``join``

``for`` loop

``rangè` function

`ìs`òperator

``split``

A

abecedarian series

ambiguity

abstract data type (ADT)

argument, [1]

abstraction

argv

accumulator

assignment, [1]

algorithm, [1], [2], [3], [4]

multiple