Katie Cunningham

Sams Teach Yourself

Python

in 24 Hours

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Katie Cunningham

Sams Teach Yourself Python in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Python in 24 Hours

ISBN-13: 978-0-672-33687-4
ISBN-10: 0-672-33687-1

Library of Congress Control Number: 2013944085

Printed in the United States of America

Third Printing: November 2014

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Editor-in-Chief
Mark Taub

Executive Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Bart Reed

Indexer
Lisa Stumpf

Proofreader
Dan Knott

Technical Editors
Doug Hellmann
Gabriel Nilsson

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Mark Shirar

Senior Compositor
Gloria Schurick

Contents at a Glance

Table of Contents

About the Author

Katie Cunningham is a Python developer at Cox Media Group. She's a fervent advocate for Python, open source software, and teaching people how to program. She's a frequent speaker at open source conferences, such as PyCon and DjangoCon, speaking on beginners' topics such as someone's first site in the cloud and making a site that is accessible to everyone.

She also helps organize PyLadies in the DC area, a program designed to increase diversity in the Python community. She has taught classes for the organization, bringing novices from installation to writing their first app in 48 hours.

Katie is an active blogger at her website (http://therealkatie.net), covering issues such as Python, accessibility, and the trials and tribulations of working from home.

Katie lives in the DC area with her husband and two children.

Dedication

This is dedicated to my family, who helps keep me sane every time
I decide to do this again. Jim, thank you for picking up the slack.
Mom, thank you for taking the kids and offering help every time
I started to look like I was going to fall over.
Kids, thank you for being okay with all the delivery food.

Acknowledgments

This book wouldn't have happened without the help from quite a few people.

First, my editor, Debra Williams Cauley, has been both patient and enthusiastic. Without her, I don't know if I would have ever hit the deadline.

A special thanks goes to my tech editors, Doug Hellmann and Gabriel Nilsson. They were machines when it came to catching my glaring errors, and their suggestions only made this book stronger. Also, a thanks goes out to Richard Jones, who took the time to review my PyGame chapter.

Thanks to Michael Thurston, who made me sound fabulous. I swear, one of these days, I'll learn to spell "installation" right.

Finally, a thank you goes out to the Python community, who has been on hand every time I had a question, needed a sanity check, or just needed some inspiration. You guys are my home.

Preface

Why Python?

I get this question quite a bit. Why should someone learning to program learn Python? Why not a language that was made for beginners, such as Scratch? Why not learn Java or C++, which most colleges seem to be using?

Personally, I believe that Python is an ideal language for beginners. It runs on multiple systems. The syntax (the grammar of the language) isn't fussy. It's easy to read, and many people can walk through a simple script and understand what it's doing without ever having written a single line of code.

It's also ideal because it's easy for a beginner to move on to more advanced projects. Python is used in a number of areas, from scientific computing to game development. A new programmer can almost always find one, if not multiple, projects to fit their tastes.

Who This Book Is For

This book is for those who have never programmed before and for those who have programmed some but now want to learn Python. This is not a book for those who are already experienced developers.

It is assumed you have a computer you have admin rights to. You'll need to install Python, as well as multiple libraries and applications later in the book. The computer does not need to be terribly powerful.

You should also have an Internet connection in order to access some of the resources.

How This Book Is Organized

This book covers the basics of programming in Python as well as some advanced concepts such as object-oriented programming.

- The Introduction and Hour 1 cover the background of Python and installation.
- Hours 2–7 cover some basics of programming, such as variables, math, strings, and getting input.

- Hours 8–12 cover advanced topics. Functions, dictionaries, and object-oriented programming will be discussed.
- Hours 13–15 discuss using libraries and modules, as well as creating your own module.
- Hours 16–19 cover working with data, such as saving to files, using standard formats, and using databases.
- Hours 20 and 21 give a taste of some projects outside of the standard library. In these hours, you will explore creating dynamic websites and making games. These hours are not meant to be complete lessons, but serve instead as a starting point for learning more.
- Hours 22 and 23 go over how to save your code properly, and how to find answers when something has gone wrong.
- Hour 24 goes over what projects you can get involved with, what resources can help you learn more, and how to get more involved in the Python community.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: consumer@samspublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

Many people idly contemplate learning how to code. It seems like something that could be of use, but many are too intimidated to jump in and try. Maybe they believe it's too late to start learning a skill like programming, or they believe they don't have enough time. Maybe they get lost too quickly, because the book they found is written for someone with previous experience with coding. It seems like an impossible task. The goal of this book is to break down the concepts behind programming into bite-sized chunks that are easy to digest as well as immediately useful.

Learning to Program

For many people, learning to program seems like an impossible task. It's painted as a field that requires a crazy amount of math, years of education and training, and, once you're done with that, endless hours of constantly banging away at a keyboard.

The truth is, although becoming a full-time developer can take quite a bit of dedication, learning how to write code can be easy. As more of our life touches computers, learning to write code to control them can enhance any career, no matter how nontechnical it may seem. An elementary school teacher might make a website to help students learn their vocabulary. An accountant could automate calculations that normally have to be done by hand. A parent could create a home inventory system to help with generating grocery lists. Nearly every profession and hobby can be enhanced through learning to program.

To put it simply, computers are stupid. Without human input, they don't know what to do. Code is a set of instructions that tells the computer not only what to do, but how to do it. Everything on your computer, from the largest applications (such as Word and video games) to the smallest (such as a calculator), is based on code.

Most code on your computer will be compiled already as an .exe or .app file. For the exercises in this book, we'll either be running them from a file or using the interpreter (which we'll get to in Hour 1, "Installing and Running Python").

Why Python?

Python is a language that is lauded for its readability, its lack of fussiness, and how easy it is to teach. Also, unlike some languages that are created specifically for teaching, it's used in countless places outside of the classroom. People have used Python to write everything from websites to tools for scientific work, from simple scripts to video games. The following is a non-exhaustive list of programs written in Python:

- **YouTube**—A popular site for viewing and sharing videos.
- **The Onion**—A parody news site.
- **Eve Online**—A video game set in space.
- **The Washington Post**—The website runs off of Django, a framework written in Python.
- **Paint Shop Pro**—An image-editing software package.
- **Google**—A significant number of applications at Google use Python.
- **Civilization IV**—A turn-based simulation game.

Python may appear simple, but it's incredibly powerful.

Getting Started

Before we get started, let's go over a list of some things you're going to need. You absolutely must have all these things before you can start learning Python. Here's what you will need:

- **Admin access**—Python doesn't require a very powerful computer to run, but you will need a computer that you have permission to install things on.
- **Internet access**—We're going to be downloading installers, and, later on, talking to web services. It doesn't need to be a fast connection, because many of the items we'll be downloading are rather small.
- **A computer**—It doesn't need to be brand new, but the faster your computer is, the faster your code should run. A computer built in the past five years should be fine.
- **Space**—A dedicated workspace can greatly enhance your ability to pick up new concepts. It should be free from distractions, such as TV.
- **No distractions**—It's almost impossible to learn something new if you have family members interrupting you, phones buzzing, or a TV blaring in the background. A good pair of noise-canceling headphones can be a wonderful asset—if you can't get rid of people and ambient noise.

For most people, the last two items can be the most difficult to get in place, but they're invaluable. Not only will you need them while learning, but you'll need them once you're done with this book and moving on to your own projects. Writing code is a creative endeavor, and requires time and space to do.

How This Book Works

Each chapter is meant to be completed in one hour or less. That includes reading the text and doing the exercises. Ideally, the exercises should be done directly after reading a chapter, so try to set aside time when you not only can focus, but have access to your computer. Not every chapter will require Internet access (those that do will warn you before you dive in).

It may be tempting to dive in to the next chapter after finishing one, but try to give yourself a break. Your brain needs time to integrate the new information, and you need to be rested before diving into more new material.

What to Do If You Get Stuck

There is one thing that applies to every person who writes code: You will get stuck. Sometimes a new concept doesn't seem to be clicking. Sometimes an error won't go away. There are days when everything you touch seems to break.

The key to getting past days like these is to not give in to frustration. Get up, move away from the computer, and go for a walk. Make a cup of tea. Talk to a friend about anything but your misbehaving code. Give yourself a chance to unwind.

When you've given yourself some space from the problem, do a quick self-assessment. Are you tired? A tired developer is a bad developer, no matter how experienced he or she is. Sometimes a bit of coffee helps, but most of the time what you need is some sleep.

If you're not tired, try re-reading the chapter. It might be time to break out the highlighters or take notes. Are some of the terms unfamiliar? Try searching for these terms online.

Is the code not working? Sometimes, you need to delete what you have (or save it in another file) and try again. Later in the book, we'll talk about better ways to debug your code, but rest assured, every developer has had to toss code at some point in his or her life.

This page intentionally left blank

HOUR 4

Storing Text in Strings

What You'll Learn in This Hour:

- How to create and print strings
- How to get information about stored text
- How to use math with stored text
- How to format strings
- When to use strings in the real world

When Python wants to store text in a variable, it creates a variable called a *string*. A string's sole purpose is to hold text for the program. It can hold anything—from nothing at all ('') to enough to fill up all the memory on your computer.

Creating Strings

Creating a string in Python is very similar to how we stored numbers in the last hour. One difference, however, is that we need to wrap the text we want to use as our string in quotes. Open your Python shell and type in the following:

```
>>> s = "Hello, world"
>>> s
'Hello, world'
```

The quotes can be either single (') or double ("). Keep in mind, though, that if you start with a double quote, you need to end with a double quote (and the same goes for single quotes). Mixing them up only confuses Python, and your program will refuse to run. Look at the following code, where the text "Harold" starts with a double quote but ends with a single quote:

```
>>> name = "Harold'
File "<stdin>", line 1
name = "Harold'
^ SyntaxError: EOL while scanning string literal
```

As you can see, we got an error. We have to make the quote types match:

```
>>> name = "Harold"
>>> name
'Harold'
>>> name2 = 'Harold'
'Harold'
```

Printing Strings

In the examples so far, Python prints out strings with the quotes still around them. If you want to get rid of these quotes, use a `print` statement:

```
>>> greeting = "Hello"
>>> print greeting
Hello
```

A `print` statement usually prints out the string, then moves to the next line. What if you don't want to move to the next line? In this case, you can add a comma (,) to the end of the `print` statement. This signals Python not to move to a new line yet. This only works in a file, though, because the shell will always move to the next line.

In this example, we print out an item along with the price on the same line:

```
print 'Apple: ',
print '$ 1.99 / lb'
```

When we run it, we get this:

```
Apple:  $ 1.99 / lb
```

We can even do calculations between the two `print` statements, if we need to. Python will not move to a new line until we tell it to.

Getting Information About a String

In Hour 2, "Putting Numbers to Work in Python," variables were compared to cups because they can hold a number of things. Cups themselves have some basic functions, too, whether they contain something or not. You can move them around, you can touch their side to see if what's in them is hot or cold, and you can even look inside them to see if there's anything in there. The same goes with strings.

Python comes with a number of built-ins that are useful for getting information about the stored text and changing how it's formatted. For example, we can use `len()` to see how long a string is.

In the following example, we want to see how long a name is:

```
>>> name = "katie"
>>> len(name)
5
```

In this case, the length of the string held in `name` is five.

In Python, variables also come with some extra capabilities that allow us to find out some basic information about what they happen to be storing. We call these methods. Methods are tacked on to the end of a variable name and are followed by parentheses. The parentheses hold any information the method might need. Many times, we leave the parentheses blank because the method already has all the information it requires.

One set of methods that comes with strings is used to change how the letters are formatted. Strings can be converted to all caps, all lowercase, initial capped (where the first letter of the string is capitalized), or title case (where the first letter and every letter after a space is capitalized). These methods are detailed in Table 4.1.

TABLE 4.1 String-Formatting Methods

Method	Description	Example
`.upper()`	Converts all letters to uppercase (a.k.a. all caps).	'HELLO WORLD'
`.lower()`	Converts all letters to lowercase.	'hello world'
`.capitalize()`	Converts the first letter in a string to uppercase and converts the rest of the letters to lowercase.	'Hello world'
`.title()`	Converts the first letter, and every letter after a space or punctuation, to uppercase. The other letters are converted to lowercase.	'Hello World'

These methods are appended to the end of a string (or variable containing a string):

```
>>> title = "wind in the willows"
>>> title.upper()
'WIND IN THE WILLOWS'
>>> title.lower()
'wind in the willows'
>>> title.capitalize()
'Wind in the willows'
>>> title.title()
'Wind In The Willows'
```

These methods are nondestructive. They don't change what's stored in the variable. In the following example, note that the string stored in `movie_title` isn't changed, even though we used `.upper()` on it:

```
>>> movie_title = "the mousetrap"
>>> movie_title.upper()
'THE MOUSETRAP'
>>> movie_title '
the mousetrap'
```

We can also see if certain things are true about a string. `is_alpha()` and `is_digit()` are two popular methods, especially when checking to see if a user put in the correct type of data for a string.

In the following string, we check to see that `birth_year` is composed of all digits and that `state` is nothing but letters:

```
>>> birth_year = "1980"
>>> state = "VA"
>>> birth_year.isdigit()
True
>>> state.isalpha()
True
```

Had `birth_year` contained any letters or symbols (or even spaces), `isdigit()` would have returned `False`. With `state`, had it contained any numbers or symbols, we would have gotten `False` as well.

```
>>> state = "VA"
>>> state.isdigit()
False
```

Math and Comparison

Just as with numbers, you can perform certain kinds of math on strings as well as compare them. Not every operator works, though, and some of the operators don't work as you might expect.

Adding Strings Together

Strings can also be added together to create new strings. Python will simply make a new string out of the smaller strings, appending one after the next.

In the following example, we take the strings stored in two variables (in this case, someone's first name and last name) and print them out together:

```
>>> first_name = "Jacob"
>>> last_name = "Fulton"
>>> first_name + last_name
'JacobFulton'
```

Note that Python doesn't add any space between the two strings. One way to add spaces to strings is to add them explicitly to the expression.

Let's add a space between the user's first and last names:

```
>>> first_name + " " + last_name
'Jacob Fulton'
```

Multiplication

You can do some funny things with multiplication and strings. When you multiply a string by an integer, Python returns a new string. This new string is the original string, repeated X number of times (where X is the value of the integer).

In the following example, we're going to multiply the string 'hello' by a few integers. Take note of the results.

```
>>> s = 'hello '
>>> s * 5
'hello hello hello hello hello'
>>> s * 10
'hello hello hello hello hello hello hello hello hello hello '
>>> s * 0
''
```

What happens if we store an integer in a string?

```
>>> s = '5'
>>> s * 5
55555
```

Normally, if we multiplied 5 by 5, Python would give us 25. In this case, however, `'5'` is stored as a string, so it's treated as a string and repeated five times.

There's some limitations to string multiplication, however. Multiplying by a negative number gives an empty string.

```
>>> s = "hello"
>>> s * -5
''
```

Multiplying by a `float` gives an error:

```
>>> s * 1.0
Traceback (most recent call last):
File "<stdin>", line 1, in <module> TypeError: can't multiply sequence by
non-int of type 'float'
```

Comparing Strings

It's possible to compare strings just as you would numbers. Keep in mind, however, that Python is picky about strings being equal to each other. If the two strings differ, even slightly, they're not considered the same. Consider the following example:

```
>>> a = "Virginia"
>>> b = "virginia"
>>> a == b
False
```

Although `a` and `b` are very similar, one is capitalized and one isn't. Because they aren't exactly alike, Python returns `False` when we ask whether they are alike.

Whitespace matters, too. Consider the following code snippet:

```
>>> greet1 = "Hello "
>>> greet2 = "Hello"
>>> greet1 == greet2
False
```

`greet1` has a space at the end of its string whereas `greet2` does not. Python looks at whitespace when comparing strings, so the two aren't considered equal.

Operators That Don't Work with Strings

In Python, the only operators that work with strings are addition and multiplication. You can't use strings if you're subtracting or dividing. If you try this, Python will throw an error and your program will stop running.

```
>>> s = "5"
>>> s / 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

If you ever see an error like this one (unsupported operand type), it usually means that the data type you're trying to use doesn't know how to use that operator.

Formatting Strings

There are many ways to format strings—from removing extra spaces to forcing new lines. You can also add in tabs as well as search and replace specified text.

Controlling Spacing with Escapes

Until now, we've been printing strings out on one line. What if we need to print out something on multiple lines? We can use the special combination of a backslash and "n" (`\n`). Every time we insert this into a string, Python will start printing on the next line.

```
>>> rhyme = "Little Miss Muffett\nSat on a tuffet\nEating her curds and whey."
>>> print rhyme
Little Miss Muffett
Sat on a tuffet
Eating her curds and whey.
```

The backslash is a special character in strings. It's called an escape, and it clues Python into the fact that you have some special formatting in mind. You can also use an escape to put a string onto several lines in your code so it's easier to read. The preceding string isn't so easy to read as it is, but we can fix that as follows:

```
>>> rhyme = "Little Miss Muffett\n\
... Sat on a Tuffet\n\
... Eating her curds and whey."
>>> print rhyme
Little Miss Muffett
Sat on a Tuffet
Eating her curds and whey.
```

A new line isn't the only thing you can do with an escape, though. You can also insert tabs with `\t`.

Take note of the spacing in the following example. Each `\t` is replaced with tab when the string is printed.

```
>>> header = "Dish\tPrice\tType"
>>> print header
Dish    Price   Type
```

The escape is also useful for when you have quotes in a string. If you're creating a string that has quotes in it, this can cause some confusion for Python. "Escaping" them lets Python know that you're not done with the string quite yet.

In the following example, the name has a single quote in it. If we don't escape it, Python gives us an error. If we do, however, Python has no problem storing the string.

```
>>> name = 'Harry O'Conner'
File "<stdin>", line 1
name = 'Harry O'Conner'
               ^ SyntaxError: invalid syntax
>>> name = 'Harry O\'Conner'
>>> print name
Harry O'Conner
```

NOTE

Another Way to Deal with Single Quotes

If you don't want to use an escape, you can use double quotes if your string contains single quotes, or vice versa. So, Python will have no issues saving "Harry O'Conner" or 'He said, "Hello" as he opened the door.'

But what if you need to use a backslash in a string? Simple: Just escape the backslash. In other words, if you want to display one backslash, you'll need to enter two backslashes.

In the following example, we want to save a path for a Windows machine. These always include backslashes, so we need to escape the backslash. When we print it, only one backslash appears.

```
>>> path = "C:\\Applications\\"
>>> print path
C:\Applications\
```

Removing Whitespace

Sometimes, a user might put extra whitespace when typing in something for your program. This can be annoying when trying to print out several strings on one line, and it can be downright disastrous if you're trying to compare strings.

In the following example, extra whitespace makes printing out a name difficult. It looks like there's too much space between the first name and middle name. To make matters more difficult, the extra whitespace means that the comparison `first_name == "Hannah"` fails.

```
>>> first_name = "Hannah "
>>> middle_name = "Marie"
>>> print first_name + " " + middle_name
Hannah  Marie
>>> if first_name == "Hannah":
...   print "Hi, Hannah!"
... else:
...   print "Who are you?"
...
Who are you?
```

Strings come with a method, `strip()`, that allows you to strip out all the whitespace at the beginning and end of a string. In the following code snippet, the name Hannah has an extra space tacked onto the end. Using `strip()` removes that space.

```
>>> first_name = "Hannah "
>>> first_name.strip()
'Hannah'
```

`strip()` not only removes all whitespace from around a string, it can remove other characters you specify. This time, Hannah is surrounded by a number of asterisks. Passing an asterisk to `strip()` removes all the asterisks in the string:

```
>>> bad_input = "****Hannah****"
>>> bad_input.strip('*')
'Hannah'
```

If you only want to strip the beginning or end of a string, you can use `rstrip()` or `lstrip()`, respectively. Here, the name Hannah has asterisks before and after it. If we pass an asterisk to `rstrip()`, only asterisks at the end of the string are removed. If we pass an asterisk to `lstrip()`, only asterisks at the beginning of the string are removed.

```
>>> bad_input = "****Hannah****"
>>> bad_input.rstrip('*')
'****Hannah'
>>> bad_input.lstrip('*')
'Hannah****'
```

Searching and Replacing Text

Sometimes, you need to find a piece of text that is located in a string. Strings come with a number of methods that let you search for text. These methods can tell you how many times the text occurs, and let you replace one substring with another.

`count()` returns how many times one string appears in another string. In this example, we're using a rather lengthy bit of text stored in a variable called `long_text`. Let's find how many times the word "the" appears:

```
>>> long_text.count('the')
5
```

Apparently, "the" appears five times.

What if we want to find out where the first instance of "ugly" appears? We can use `find()`. In this example, we want to find where the first instance of the word "ugly" appears in `long_text`.

```
>>> long_text.find('ugly')
25
```

In this example, "ugly" appears starting at the 25th character. A character is one letter, number, space, or symbol.

NOTE

When `find()` Finds Nothing

If `find()` doesn't find anything, it returns -1.

Strings in Python also come with the ability to replace substrings in strings. You can pass two strings to `replace()`, and Python will find all instances of the first string and replace it with the second string.

For example, if we don't like the term "ugly," we can replace it with "meh" by using `replace()` and giving it `'ugly'` and `'meh'` as parameters.

```
>>> long_text.replace('ugly', 'meh')
"Beautiful is better than meh.\n    Explicit is better ...[snip]"
```

NOTE

Zen of Python

Want to see what text I used for this section? In your interpreter, type `import this`. The Zen of Python will print out! This is the main philosophy behind Python, and is one of the Easter eggs in the Python library.

Using Strings in the Real World

In previous hours, we've gone over how Python might help the waiter in our imaginary restaurant. What about the chef? How can strings benefit her?

Most obviously, she can store the specials of the day in a script that can be run later by the waiter. That way, he can run it and see what the specials are without bothering her.

In the following script, the chef has saved a number of specials. She then prints them out in a formatted list of the specials of the day.

```
breakfast_special = "Texas Omelet"
breakfast_notes = "Contains brisket, horseradish cheddar"
lunch_special = "Greek patty melt"
lunch_notes = "Like the regular one, but with tzatziki sauce"
dinner_special = "Buffalo steak"
dinner_notes = "Top loin with hot sauce and blue cheese. NOT BUFFALO MEAT."

print "Today's specials"
print "*"*20
print "Breakfast: ",
print breakfast_special
print breakfast_notes
print
print "Lunch: ",
print lunch_special
print lunch_notes
print
```

```
print "Dinner: ",
print dinner_special
print dinner_notes
```

When the waiter runs it, the following is printed out:

```
Today's specials
********************
Breakfast: Texas Omelet
Contains brisket, horseradish cheddar
Lunch: Greek patty melt
Like the regular one, but with tzatziki sauce
Dinner: Buffalo steak
Top loin with hot sauce and blue cheese. NOT BUFFALO MEAT.
```

If the cook wants to change the specials later, she can edit the first few lines in the file.

Summary

During this hour, you learned that text is stored in something called a string. Python allows you to do certain kinds of math operations on strings, and offers some extra methods for strings, such as removing whitespace.

Q&A

Q. Is there any way to see all of the things I can do with a string without looking it up online?

A. If you want to see everything you can do with strings, type this into your Python shell:

```
>>> s = ""
>>> help(type(s))
```

A list of everything you can do with strings will pop up. Pressing Enter will move you down one line, your up arrow will move you up one line, spacebar will move you down one page, and "q" will close the help menu. Note that this behavior is slightly different in IDLE, where all the text is printed at once.

Incidentally, you can get this screen with any kind of Python data type. If you wanted to find out all the methods that come with the integer type, you could do something like this:

```
>>> s = 1
>>> help(type(s))
```

Q. Why are the methods to remove whitespace from the beginning and end of a string called "right strip" and "left strip"? Why not "beginning" and "end"?

A. In quite a few languages, text isn't printed from left to right. Arabic and Hebrew are both written from right to left, whereas many Eastern scripts are written from top to bottom. "Right" and "left" are more universal than "beginning" and "end".

Q. How big can a string be?

A. That depends on how much memory and hard drive space your computer has. Some languages limit the size of a string, but Python has no hard limit. In theory, one string in your program could fill up your whole hard drive!

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the answers that follow.

Quiz

1. What characters can be stored in strings?
2. What math operators work with strings?
3. What is the backslash character (\) called? What is it used for?

Answers

1. Alphabetic characters, numbers, and symbols can all be stored in strings, as well as whitespace characters such as spaces and tabs.
2. Addition and multiplication operators work with strings.
3. The backslash is called an "escape" and indicates that you want to include some special formatting, such as a tab, new line, a single or double quote, or a backslash.

Exercise

In your program, you're given a string that contains the body of an email. If the email contains the word "emergency," print out "Do you want to make this email urgent?" If it contains the word "joke," print out "Do you want to set this email as non-urgent?"

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T